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

WE, THE UNDERSIGNED MEMBERS OF THE COMMITTEE,

HAVE APPROVED THIS THESIS

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

By

Lucas Wukmer

COMMITTEE MEMBERS

Jen-Mei Chang, Ph.D. (Chair)	Mathematics and Statistics
James von Brecht, Ph.D.	Mathematics and Statistics
William Ziemer, Ph.D.	Mathematics and Statistics

ACCEPTED AND APPROVED ON BEHALF OF THE UNIVERSITY

Tangan Gao, Ph.D.
Department Chair, Mathematics and Statistics

California State University, Long Beach

December 2018

ACKNOWLEDGEMENTS

Acknowledgments go here.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS iii

LIST OF TABLES vi

LIST OF FIGURES vii

CHAPTER

1. INTRODUCTION 1

The Applied Problem 1
Research Goals 1

2. MATHEMATICAL METHODS 2

Problem Setup in Image Processing 2
Differential Geometry 3
Preliminaries of Differential Geometry 3
Curvature of a surface and its calculation 6
Principal Curvatures and Principal Directions 12
The Weingarten map and Principal Curvatures of a Cylindrical Ridge 17
The Frangi Filter: Uniscale 21
Anisotropy Measure 22
Structureness measure 24
The Frangi vesselness measure 24
The Frangi vesselness filter: Choosing parameters β and γ . 25
Linear Scale Space Theory 26
Axioms 26
Uniqueness of the Gaussian Kernel 28
Scale Spaces over Discrete Structures 32
The Frangi Filter: A multiscale approach 34
Thresholding 35
Calculating the 2D Hessian 36
Convolution Speedup via FFT 36
Fourier Transform of a continuous 1D signal 36
Fourier Transform of a Discrete 1D signal 37

APPENDIX Page

2D DFT Convolution Theorem 37
FFT 40

3. IMPLEMENTATIONS 44

Calculating the Hessian 44

4. RESEARCH PROTOCOL 45

Samples / Image Domain 45
A representative sample 45
Knowns and Unknowns 47
Data Cleaning and Preprocessing 48
Boundary Dilation 50
Deglaring 53
Variations in the Data Set and Imperfections of the Ground Truth 56
Multiscale Setup 57
Applying Vesselness Measure 58
Scale-space post-processing 58
Multiscale Merging 58
Cleanup/Postprocessing 58
Measurements 58
Erode plate / dilate boundary 59

5. RESULTS AND ANALYSIS 60

Sample visual output 60
The confusion matrix 60
A Source of "False Negatives" in the NCS data set 60
Results 61
Answer Research Questions 61

6. CONCLUSION 62

Review of Work 62
Future research directions 62

APPENDICES 63

A. CODE LISTINGS 64

BIBLIOGRAPHY 124

LIST OF TABLES

TABLE Page
1 Vessel width color code 47

LIST OF FIGURES

FIGURE		Page
1	Tangent plane of a graph	6
2	The graph of a cylindrical ridge of radius r	18
3	The principal eigenvectors at a ridge like structure	23
4	A representative placental sample and tracing	46
5	Preprocessed files from an NCS sample	49
6	Effect of boundary dilation on Frangi responses	51
7	Deglaring a sample using a hybrid inpainting method	54
8	Comparison of glare inpainting methods (detail)	55
9	Frangi vesselness score at several scales	58
10	Frangi vesselness score at several scales (inset)	59
11	″True″ false positives and “False″ false positives	61

CHAPTER 1

INTRODUCTION

The Applied Problem

From [cite salafia], it is useful to develop a neonatal test for high risk of Autism Spectrum Disorder. There is some evidence as in [6] that there is some correlation between risk and placental health. Most ASD cases are not diagnosed until the child reaches three or four, so the benefit of any neonatal testing would be very beneficial, as the brain may be more receptive to treatment at a young age. In particular, it was shown in [6] that measurements of the placental chorionic surface vascular network (PCSVN) may be useful in identifying such risk. Whereas previous studies have required manual tracing of the PCSVN in order to make these measurements, there was been work to automate this procedure, as in [13] [8], and so on. We continue the work of developing a procedure to automate extraction of the PCSVN.

Our basic goal of ”vascular network extraction” is a frequent one in image processing. There are have been many techniques adapted to extracting vascular networks. The placenta in particular presents a greater degree of difficulty due to the nature of the vascular network. It’s a surface network and the ”background” has a great degree of topology itself, causing many naive approaches to fail that work with other image domains.

Research Goals

Our present work is to improve upon the vascular network extraction developed in [13] and test our extraction against the manual traces developed.

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 suitiable 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 *MxN* array of pixels, whose intensity is given by an integer value between 0 and 255.

Definition 2.1 (Image as a pixel matrix)**.**

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

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)**.**

$$h : \mathbb{R}^2 \to \mathbb{R} \quad with \quad h \in C^2\left(\mathbb{R}^2\right), \quad where \quad h(i,j) = \mathtt{I}_{ij}\ \forall (i,j) \in \{0,...,M\} \times \{0,...,N\} \subset \mathbb{N}^2$$

That is, the function h is identical to the pixel matrix $\mathtt{I}$ 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 `I` is not really a perfect representation of the underlying "content" within the picture. Not only is information lost when `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 [11], especially when the goal is noise reduction. However, we will be content to simply represent the pixels of `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 `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 [17] (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 $h : U \to \mathbb{R}$ (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*

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

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.
$p = f(u) = f(u_1,u_2) \;=\; \Big(u_1,u_2,h(u_1,u_2)\Big)$, 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*

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

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

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

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 $U \to \mathbb{R}^3$ 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)**.**

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

This 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.

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

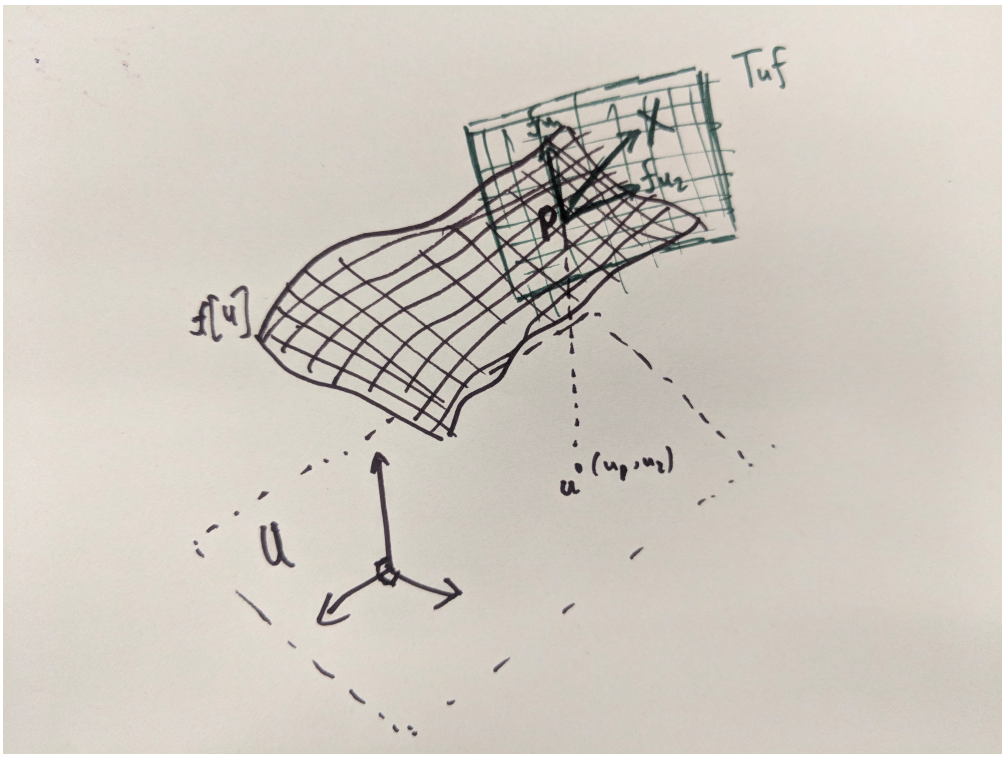


FIGURE 1: Tangent plane of a graph

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. □

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 $c : I \to \mathbb{R}^3$ 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 $c : I \to \mathbb{R}^3$ 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,*

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

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

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:

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

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

$$= \theta_1'(t)\left(\frac{\partial f}{\partial u_1}\right) + \theta_2'(t)\left(\frac{\partial f}{\partial u_2}\right) \in T_u f. \tag{2.3}$$

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 $X \in T_u f$, any regular curve on the surface $c : I \to \text{image}(f)$ with $p \in I : \theta_c(p) = u$ where $c'(p) = X$ will have the same curvature.*

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*

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

Each partial above understood to be evaluated at the input $u \in U$; that is, we calculate $\frac{\partial f}{\partial u_i}\Big|_u$. 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 $T_u f$. However, we can also show it directly:

To show that $\left\{\frac{\partial \nu}{\partial u_1}, \frac{\partial \nu}{\partial u_2}\right\} \in T_u f$, 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 $\text{span}\left\{\frac{\partial \nu}{\partial u_1}, \frac{\partial \nu}{\partial u_2}\right\} \subset T_u f$ 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 $X \in T_u f$ be given and consider some curve where $\frac{\partial c}{\partial t}(u) = X$ where $X \in T_u f$. 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 $u = \theta_c(p)$)

$$c'' = \langle c'', X \rangle X + \langle c'', \nu \rangle \nu \tag{2.4}$$

.

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:

$$\langle c'', v\rangle = \frac{\partial}{\partial t}[\langle c', v\rangle] - \langle c', \frac{\partial v}{\partial t}\rangle \tag{2.5}$$
$$= \frac{\partial}{\partial t}[\langle X, v\rangle] - \langle c', \frac{\partial v}{\partial t}\rangle \tag{2.6}$$
$$= 0 - \langle X, \frac{\partial v}{\partial t}\rangle \tag{2.7}$$

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

$$\|c''\| = \|\langle c'', X\rangle X + \langle c'', v\rangle v\| = \|0 + \langle c'', v\rangle v\| \tag{2.8}$$
$$= -\langle X, \frac{\partial v}{\partial t}\rangle \|v\| \tag{2.9}$$
$$= -\langle X, \frac{\partial v}{\partial t}\rangle \tag{2.10}$$
$$= \langle X, -\frac{\partial v}{\partial t}\rangle \tag{2.11}$$

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

$$-\frac{dv}{dt} = -\frac{d}{dt}\Big[v(u_1, u_2)\Big] \tag{2.12}$$
$$= -\frac{d}{dt}\Big[v(\theta_1(t), \theta_2(t))\Big] \tag{2.13}$$
$$= \theta_1'(t)\left(-\frac{\partial v}{\partial u_1}\right) + \theta_2'(t)\left(-\frac{\partial v}{\partial u_2}\right) \tag{2.14}$$

Identifying span$\left\{-\frac{\partial v}{\partial u_i}\right\}_{i=1,2}$ 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)**.**

$$\mathsf{L} : T_u f \to T_u f \quad \textit{given by the composition} \quad \mathsf{L} = Dv \circ (Df)^{-1}.$$

That is, $\mathsf{L}(\frac{\partial f}{\partial u_i}) = -\frac{\partial v}{\partial u_i}$ for $i = 1, 2$, where the negative sign comes about from blind adherence to eq. (2.14) and eq. (2.11). We will This allows us to rewrite the time

derivative of the Gauss map eq. (2.12) as

$$-\frac{dv}{dt} = \theta_1'(t)\left(-\frac{\partial v}{\partial u_1}\right) + \theta_2'(t)\left(-\frac{\partial v}{\partial u_2}\right) \tag{2.15}$$

$$= \theta_1'(t)\left(\mathsf{L}\left(\frac{\partial f}{\partial u_1}\right)\right) + \theta_2'(t)\left(\mathsf{L}\left(\frac{\partial f}{\partial u_2}\right)\right) \tag{2.16}$$

$$= \mathsf{L}\left[\theta_1'(t)\left(\frac{\partial f}{\partial u_1}\right) + \theta_2'(t)\left(\frac{\partial f}{\partial u_2}\right)\right] \tag{2.17}$$

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

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

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

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. □

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_v at point u in the direction X is given by*

$$\kappa_v := \langle X, \mathsf{L}(X)\rangle$$

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 $\mathsf{L}(\frac{\partial f}{\partial u_i}) = -\frac{\partial v}{\partial u_i}$.

That is, we may trace any $X \in T_u f$ 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 in the general case, which is certainly useful for surfaces f that are not graphs. We refer to [17] for the general proof. 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 $\widehat{\mathsf{L}} \in R^{2\times 2}$) we simply wish to find a linear transformation such that $\widehat{\mathsf{L}} \left.\frac{\partial f}{\partial u_i}\right|_{T_u f} = -\left.\frac{\partial \nu}{\partial u_i}\right|_{T_u f}$ for $i = 1, 2$ where $-X|_{T_u f}$ denotes that $X \in T_u f$ is being represented in so-called 'local coordinates' for $T_u f$ (Strictly speaking, of course $T_u f \subset \mathbb{R}^3$ and thus $\frac{\partial f}{\partial u_i} \in \mathbb{R}^3$. Thus when we say $\left.\frac{\partial f}{\partial u_i}\right|_{T_u f}$ we are referring to this 3-vector expanded with respect to the two-dimensional basis for $T_u f$). In matrix form, we describe this situation as

$$\left[\widehat{\mathsf{L}}\right]\begin{bmatrix} | & | \\ \left.\frac{\partial f}{\partial u_1}\right|_{T_u f} & \left.\frac{\partial f}{\partial u_2}\right|_{T_u f} \\ | & | \end{bmatrix} = \begin{bmatrix} | & | \\ \widehat{\mathsf{L}}\left.\frac{\partial f}{\partial u_1}\right|_{T_u f} & \widehat{\mathsf{L}}\left.\frac{\partial f}{\partial u_2}\right|_{T_u f} \\ | & | \end{bmatrix} \tag{2.20}$$

$$= \begin{bmatrix} | & | \\ -\left.\frac{\partial \nu}{\partial u_1}\right|_{T_u f} & -\left.\frac{\partial \nu}{\partial u_2}\right|_{T_u f} \\ | & | \end{bmatrix} \tag{2.21}$$

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

$$\implies \left[\widehat{\mathsf{L}}\right]\begin{bmatrix} - & \frac{\partial f}{\partial u_1} & - \\ - & \frac{\partial f}{\partial u_2} & - \end{bmatrix}\begin{bmatrix} | & | \\ \frac{\partial f}{\partial u_1} & \frac{\partial f}{\partial u_2} \\ | & | \end{bmatrix} = \begin{bmatrix} - & \frac{\partial f}{\partial u_1} & - \\ - & \frac{\partial f}{\partial u_2} & - \end{bmatrix}\begin{bmatrix} | & | \\ -\frac{\partial \nu}{\partial u_1} & -\frac{\partial \nu}{\partial u_2} \\ | & | \end{bmatrix} \tag{2.22}$$

We can simplify this greatly by defining

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

so that

$$\begin{bmatrix} \widehat{\mathsf{L}} \end{bmatrix} \begin{bmatrix} g_{11} & g_{12} \\ g_{21} & g_{22} \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} \tag{2.24}$$

Then we rearrange to solve for $\widehat{\mathsf{L}}$ as

$$\widehat{\mathsf{L}} = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} g_{11} & g_{12} \\ g_{21} & g_{22} \end{bmatrix}^{-1} \tag{2.25}$$

where $\left[g_{ij}\right]$ 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 $f : U \to \mathbb{R}^3$ 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 to 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 $\nabla_v \langle v, Av \rangle$:

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

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

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

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

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

$$\implies \nabla_v\langle v, Av\rangle = 2Av. \tag{2.31}$$

□

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 $\kappa_\nu = \langle \mathsf{L}X, X\rangle$ subject to $\langle X, X\rangle = 1$ iff X is a (normalized) eigenvector of the Weingarten map 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 L freely as the 2×2 matrix $\widehat{\mathsf{L}}$. Using the method of Lagrange multipliers, we define the Lagrangian:

$$\mathscr{L}(X;\lambda) := \langle \widehat{\mathsf{L}}X, X\rangle - \lambda\Big(\langle X, X\rangle - 1\Big) \tag{2.32}$$

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

$$\begin{cases} \nabla_X\langle \widehat{\mathsf{L}}X, X\rangle - \lambda\nabla_X\left(\langle X, X\rangle - 1\right) = 0 \\ \qquad\qquad\qquad\qquad \langle X, X\rangle - 1 = 0 \end{cases} \tag{2.33}$$

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

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

which implies that X is an eigenvector of $\widehat{\mathsf{L}}$ 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. □

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 (Relationship between Hessian and Weingarten Map of a Graph)**.** *Given the graph $f : U \to \mathbb{R}^3$ where $(x, y) \mapsto (x, y, h(x, y))$, the matrix representation of its Weingarten map*

is given by

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

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

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

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

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

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

and we have the unit normal vector (Gauss map)

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

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

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

We then calculate each h_{ij} as

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

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

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

To calcuate the second, we use

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

and thus

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

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

Thus the matrix of the Weingarten map $\widehat{\mathsf{L}}$ 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 $\widehat{\mathsf{L}}$ and $\text{Hess}(h)$ have the same eigenvalues and eigenvectors at these points.

But this observation is more broadly useful than that, since if $\tilde{G}$ above is close to identity, then the eigenvalues and eigenvectors of $\widehat{\mathsf{L}}$ will be similarly close to the eigenvalues of the Hessian. We can rewrite $\tilde{G}$ from eq. (2.35) as identity plus a small matrix:

$$\tilde{G} = I + [\delta], \quad [\delta] := \begin{bmatrix} h_y^2 & -h_x h_y \\ -h_x h_y & h_x^2 \end{bmatrix} \tag{2.44}$$

We can then rewrite eq. (2.35) as

$$\widehat{\mathsf{L}} = \frac{1}{\sqrt{1+h_x^2+h_y^2}}\text{Hess}(h) + \frac{1}{\sqrt{1+h_x^2+h_y^2}}\text{Hess}(h)[\delta] \tag{2.45}$$

We can see that as h_x, h_y are close to zero, $[\delta]$ will be very close to the zero matrix (and the constant $\frac{1}{\sqrt{1+h_x^2+h_y^2}}$ will be very close to 1 as well), and we should not expect the addition of a "close to 0" matrix to have much effect on the eigenvectors or eigenvalues. This intuition is confirmed by a result from Wilkinson [31], which we state without rigorous proof.

Theorem 2.7. *If A, B are matrices such that $|A_{ij}| < 1, |B_{ij}| < 1$ (a condition that can be ignored with scaling) and λ is a simple eigenvalue of A, then given $\epsilon > 0$, there exists a simple eigenvalue $\tilde{\lambda}$ of the matrix $A + \epsilon B$ with $|\lambda - \tilde{\lambda}| = O(\epsilon)$. Similarly, if v is an eigenvector of A, then $\tilde{v}$ is an eigenvector of $A + \epsilon B$ with $|v - \tilde{v}| = O(\epsilon)$.*

The proof ultimately relies on a general result of analysis, that the zeros of a polynomial are continuous with respect to its coefficients. In this case, the polynomial in question is the characteristic polynomial $p(\lambda) = \det(\lambda I - A - \epsilon B)$, whose coefficients will scale with ϵ. Thus $\widehat{\mathsf{L}} \approx \text{Hess}(h)$ for any point where the gradient $\nabla h \approx 0$. We shall see that we're only concerned with regions where h_x, h_y is small anyway, and we do not expect

In the event that we do wish to rigorously compute the Weingarten map should want to be rigorously computed "without approximation"–that is, without concern for the magnitude of the gradient–we refer to [14] and survey papers mentioned therein.

To make the Weingarten map and its relationship to the Hessian 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

$$f : \mathbb{R}^2 \to \mathbb{R}^3 \text{ by } f(x,y) = (x,y,h(x,y)), \text{ with } h(x,y) = \begin{cases} \sqrt{r^2 - x^2} & -r \le x \le r \\ 0 & \text{else} \end{cases} \tag{2.46}$$

The graph is shown in fig. 2. We calculate the necessary partial derivatives of f as follows:

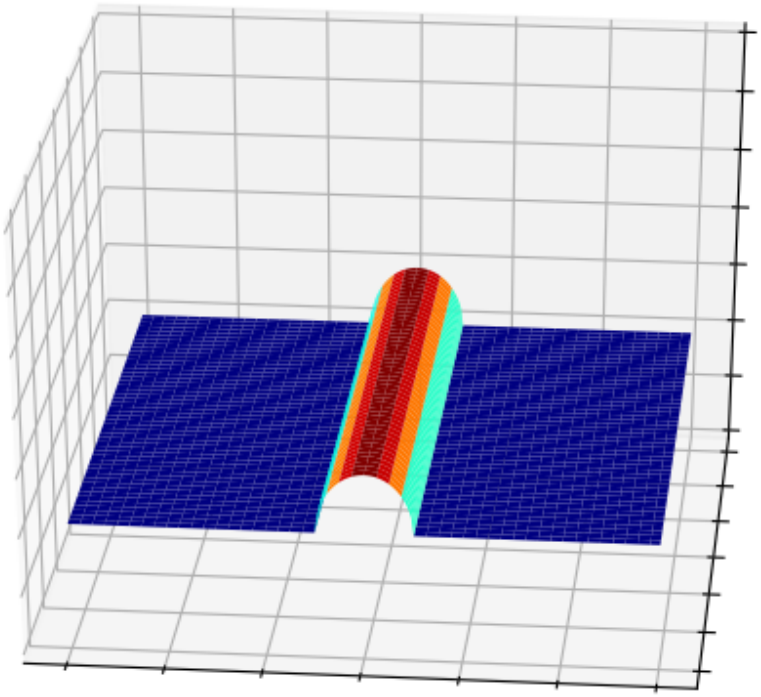


FIGURE 2: The graph of a cylindrical ridge of radius r

$$\frac{\partial f}{\partial x} = \left(1, 0, \frac{-x}{\sqrt{r^2 - x^2}}\right) \quad , \quad \frac{\partial^2 f}{\partial x^2} = \left(0, 0, \frac{-r^2}{\left(\sqrt{r^2 - x^2}\right)^3}\right) \tag{2.47}$$

$$\frac{\partial f}{\partial y} = (0, 1, 0) \quad , \quad \frac{\partial^2 f}{\partial y^2} = \frac{\partial^2 f}{\partial x \partial y} = 0 \tag{2.48}$$

The gauss map is given by

$$\nu(x, y) = \frac{\frac{\partial f}{\partial x} \times \frac{\partial f}{\partial y}}{\| \frac{\partial f}{\partial x} \times \frac{\partial f}{\partial y} \|} = \left(\frac{x}{r},\, 0,\, \frac{\sqrt{r^2 - x^2}}{r}\right) \tag{2.49}$$

$$\Longrightarrow \frac{\partial \nu}{\partial x} = \left(\frac{1}{r}\,,\, 0\,,\, \frac{-x}{r\sqrt{r^2 - x^2}}\right) \quad , \quad \frac{\partial \nu}{\partial y} = (0, 0, 0)\,. \tag{2.50}$$

We then calculate matrix elements of the Weingarten map's construction as given in eq. (2.41) and eq. (2.43) :

$$[h_{ij}] = \frac{1}{\sqrt{1+h_x^2+h_y^2}}\text{Hess}(h) = \frac{1}{\sqrt{1+\left(\frac{x^2}{r^2-x^2}\right)}}\begin{bmatrix}\frac{-r^2}{\sqrt{r^2-x^2}^3} & 0\\ 0 & 0\end{bmatrix} = \begin{bmatrix}\frac{-r}{r^2-x^2} & 0\\ 0 & 0\end{bmatrix} \tag{2.51}$$

$$[g_{ij}]^{-1} = \begin{bmatrix}\frac{r^2-x^2}{r^2} & 0\\ 0 & 1\end{bmatrix} \tag{2.52}$$

$$\implies \widehat{\mathsf{L}} = [h_{ij}][g_{ij}]^{-1} = \begin{bmatrix}\frac{-r}{r^2-x^2} & 0\\ 0 & 0\end{bmatrix}\begin{bmatrix}\frac{r^2-x^2}{r^2} & 0\\ 0 & 1\end{bmatrix} \tag{2.53}$$

$$= \begin{bmatrix}-\frac{1}{r} & 0\\ 0 & 0\end{bmatrix} \tag{2.54}$$

We see that $u_2 = (0,1)$ and $u_1 = (1,0)$ are eigenvectors for $\widehat{\mathsf{L}}$ 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 theorem 2.2 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 $\text{Hess}(x,y)$ of the surface L at a point (x,y) on the surface as the matrix of its second partial derivatives:

$$\text{Hess}(x,y) = \begin{bmatrix}L_{xx}(x,y) & L_{xy}(x,y)\\ L_{yx}(x,y) & L_{yy}(x,y)\end{bmatrix} \tag{2.55}$$

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

$$\text{Hess}(x,y)u_i = \kappa_i u_i\,, \quad i = 1,2 \tag{2.56}$$

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, 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 basic result from linear algebra, [4]:

Lemma 2.8 (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 $v_1^T v_2 = 0$.

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

In the case that $\lambda_1 = \lambda_2 =: \lambda$, we can define (as in Gram-Schmidt orthogonalization) $u = v_2 - \frac{v_1^T v_2}{v_1^T v_1} v_1$. 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: Uniscale

The Frangi filter, first described by Alejandro Frangi et al. in [10] 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 [28] and [24], as well as a comparison given in [27]. 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 orignally 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:

$$V_\sigma(x_0, y_0) = \begin{cases} 0 & \text{if} \quad \lambda_2 > 0 \\ \exp\left(-\frac{A^2}{2\beta^2}\right)\left(1 - \exp(-\frac{S^2}{2\gamma^2})\right) & \text{otherwise} \end{cases} \tag{2.57}$$

where

$$A := |\lambda_1/\lambda_2| \quad \text{and} \quad S := \sqrt{\lambda_1^2 + \lambda_2^2} \tag{2.58}$$

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. (2.57), and in particular, the ratios defined in eq. (2.58). 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 tor represent their magnitudes, as u_2 should have a

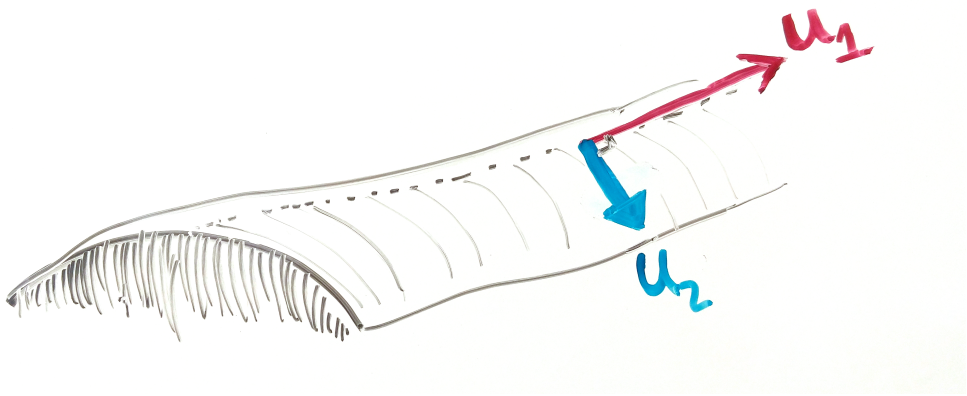


FIGURE 3: The principal eigenvectors at a ridge like structure

much larger relative magnitude by design!

Of course, if the the ridge is perfectly circular along its cross section (as was in section 2.2.4, 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 (section 4.1), there will be points where we seem to ”lose” our ridgelike structure, but this is simply due to an error in the sample.

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 2.5.

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. (2.57) in which a factor of exp{...} 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. (2.57), 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 indentify 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 $\exp\{\cdots\}$ and $1-\exp\{\cdots\}$ 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 [16] 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

$$\Delta K(x,y,\sigma) = K_\sigma(x,y,\sigma) \text{ for } \sigma \geq 0 \text{ such that } K(x,y,0) = u_0(x,y). \tag{2.59}$$

where $K : \mathbb{R}^3 \to \mathbb{R}$ and $u_0 : \mathbb{R}^2 \to \mathbb{R}$: is the original image (viewed as a continuous surface) and σ is a resolution parameter. Much work has been done to formalize this approach [29]. There is a long list of desired properties–we will to 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.

Axiom 2.1 (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.*

Axiom 2.2 (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 not immediately clear though. An argument from functional analysis (see [12]) implies that there is a so-called "infinitesimal generator" A which is a limit case of our desired operator T; that is

$$Au_0 = \lim_{\sigma \downarrow 0} \frac{T_\sigma u_0 - u_0}{\sigma} \tag{2.60}$$

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

$$\partial_\sigma K(x,y;\sigma) = \lim_{\sigma \downarrow 0} \frac{K(\cdot;\sigma+h) - K(\cdot;\sigma)}{h} = A(T_\sigma u) = A(K(\cdot,\sigma)) \tag{2.61}$$

We shall return to this idea later and more concretely describe A once we actually characterize the generating operator T_σ.

Axiom 2.3 (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.*

$$T_\sigma u = T_{\sigma_1+\sigma_2} u \tag{2.62}$$

Axiom 2.4 (Causality Conditon)**.** *The following 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.

$$\begin{cases} \nabla K(x_0, y_0; \sigma_0) = 0 \\ \Delta K(x_0, y_0; \sigma_0) < 0 \end{cases} \implies K(x_0, y_0; \sigma_1) \leq K(x_0, y_0; \sigma_0) \; \forall \; \sigma_1 \geq \sigma_0 \tag{2.63}$$

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.

$$\begin{cases} \nabla K(x_0, y_0; \sigma_0) = 0 \\ \Delta K(x_0, y_0; \sigma_0) > 0 \end{cases} \implies K(x_0, y_0; \sigma_1) \geq K(x_0, y_0; \sigma_0) \; \forall \; \sigma_1 \geq \sigma_0 \tag{2.64}$$

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. (2.59) must hold. This has been shown in various ways, both by Koenderink [16], Babaud [3], as well as Lindeberg in [29]. In fact, it is shown that the Gaussian is the unique convolution kernel that works.

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,

$$K(x,y;\sigma) = T_\sigma u_0 = G_\sigma \star u_0 \quad \text{where} \quad G_\sigma := \frac{1}{2\pi\sigma^2} e^{\left(-|x|^2/(2\sigma^2)\right)} \tag{2.65}$$

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.

$$u : \mathbb{R}^2 \supset \Omega \to \mathbb{R} \text{ with } u(\boldsymbol{x},t) : \begin{cases} \frac{\partial u}{\partial t}(\boldsymbol{x},t) = \Delta u(\boldsymbol{x},t) \quad , \ t \geq 0 \\ u(\boldsymbol{x},0) = u_0(\boldsymbol{x}) \end{cases} \tag{2.66}$$

We show that

$$u(\boldsymbol{x},t) = \left(G_{\sqrt{2t}} \star u_0\right)(\boldsymbol{x}) \tag{2.67}$$

solves (the above tagged equation), where

s

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

Lemma 2.9. *Derivative of a convolution is the way that it is (obviously rewrite this).*

Proof. For a single variable,

$$\frac{\partial}{\partial\alpha}\left[f(\alpha) \star g(\alpha)\right] = \frac{\partial}{\partial\alpha}\left[\int f(t)g(\alpha - t)dt\right] \tag{2.68}$$

$$= \int f(t)\frac{\partial}{\partial\alpha}\left[g(\alpha - t)\right]dt \tag{2.69}$$

$$= \int f(t)\left(\frac{\partial g}{\partial\alpha}\right)g(\alpha - t)dt \tag{2.70}$$

$$= f(\alpha) \star g'(\alpha) \tag{2.71}$$

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

$$\Delta(f \star g) = \Delta(f) \star g = f \star \Delta(g) \tag{2.72}$$

□

Theorem 2.10. *$u(\boldsymbol{x}, t) = \left(G_{\sqrt{2t}} \star u_0\right)(\boldsymbol{x})$ solves the heat equation.*

Proof. We focus on the particular kernel

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

Then

$$\begin{aligned}
\frac{\partial u}{\partial t}(\boldsymbol{x}, t) &= \frac{\partial}{\partial t}\left(G_{\sqrt{2t}}(\boldsymbol{x}, t) \star u_0(\boldsymbol{x})\right) && (2.73)\\
&= \frac{\partial}{\partial t}\left(G_{\sqrt{2t}}(\boldsymbol{x}, t)\right) \star u_0(\boldsymbol{x}) && (2.74)\\
&= \frac{\partial}{\partial t}\left(\frac{1}{4\pi t} e^{(-|x|^2/(4t))}\right) \star u_0(\boldsymbol{x}) && (2.75)\\
&= \left[-\frac{1}{4\pi t^2} e^{(-|x|^2/(4t))} + \frac{1}{4\pi t}\left(\frac{-|x|^2}{4t^2}\right) e^{-|x|^2/(4t)}\right] \star u_0(\boldsymbol{x}) && (2.76)\\
&= -\frac{1}{4t^2}\left(e^{(-|x|^2/(4t))} + |\boldsymbol{x}|^2 G_{\sqrt{2t}}(\boldsymbol{x}, t)\right) \star u_0(\boldsymbol{x}) && (2.77)
\end{aligned}$$

and from the previous lemma,

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

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

$$\frac{\partial}{\partial x}G_\sigma(x,y) = A\left(\frac{-2x}{2\sigma^2}\right)\exp\left(-\frac{x^2+y^2}{2\sigma^2}\right)$$

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

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

$$\Delta G_\sigma(x,y) = \frac{\partial^2}{\partial x^2}(G_\sigma)+\frac{\partial^2}{\partial y^2}(G_\sigma) = \frac{1}{\sigma^2}G_\sigma(x,y)\left[\frac{x^2+y^2}{\sigma^2}-2\right] \tag{2.78}$$

Then, given lemma 2.9, we conclude

$$\Delta\left[G_\sigma(x,y)\star u_0(x,y)\right] = \left(\frac{1}{\sigma^2}G_\sigma(x,y)\left[\frac{x^2+y^2}{\sigma^2}-2\right]\right)\star u_0(x,y) \tag{2.79}$$

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

$$\Delta\left[G_{\sqrt{2t}}(x,y)\star u_0(x,y)\right] = \left(\frac{1}{2t}G_{\sqrt{2t}}(x,y)\left[\frac{x^2+y^2}{2t}-2\right]\right)\star u_0(x,y) \tag{2.80}$$

$$= \left(G_{\sqrt{2t}}(x,y)\left[\frac{x^2+y^2}{4t^2}-\frac{1}{t}\right]\right)\star u_0(x,y) \tag{2.81}$$

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

$$\frac{\partial}{\partial t}\Big[G_{\sigma(t)}(x,y) \star u_0(x,y)\Big] = \frac{\partial}{\partial t}\Big[G_{\sigma(t)}(x,y)\Big] \star u_0(x,y) \tag{2.82}$$

$$= \frac{\partial}{\partial t}\Big[G_{\sqrt{2t}}(x,y)\Big] \star u_0(x,y) \tag{2.83}$$

$$= \frac{\partial}{\partial t}\left[\frac{1}{4\pi t}\exp\left(-\frac{x^2+y^2}{4t}\right)\right] \star u_0(x,y) \tag{2.84}$$

$$= \left[-\frac{1}{4\pi t^2}\exp\left(-\frac{x^2+y^2}{4t}\right) + \frac{1}{4\pi t}\left(\frac{x^2+y^2}{4t^2}\exp\left(-\frac{x^2+y^2}{4t}\right)\right)\right] \star u_0(x,y) \tag{2.85}$$

$$= \left(G_{\sqrt{2t}}(x,y)\left[\frac{x^2+y^2}{4t^2} - \frac{1}{t}\right]\right) \star u_0(x,y) \tag{2.86}$$

Combining these results, we find that

$$\frac{\partial}{\partial t}\Big[G_{\sqrt{2t}} \star u_0\Big] = \Delta\Big[G_{\sqrt{2t}} \star u_0\Big] \tag{2.87}$$

as desired. □

Scale Spaces over Discrete Structures

The above developments from scale space axioms have (since their first appearance) been recast in terms of discrete structures (rather than continuous surfaces) as in [21]. However, we've chosen to present the above in their original continuous surface for clarity of argument. The discrete case is not much different– we still have the same axioms, and it can be shown that the family of scaled images must simply satisfy a discrete version of the However, viewing our actual image definition 2.1 as a sample of a continuous surface definition 2.2, we might naïvely expect our convolution by the Gaussian to "commute" with our supposed sampling of the continuous signal, or even that we could simply convolve our discrete signal with a discretely sampled Gaussian kernel. The latter in fact, seems to be an often implemented interpretation of scale space theory.

To be clear, the "sampled" 1D Gaussian Kernel we have in mind might be given by:

Definition 2.13 (Sampled Gaussian Kernel and Generated Family)**.**

$$g(n;\sigma) = \frac{1}{2\pi\sigma} e^{-n^2/2\sigma}, \quad -\infty < n < \infty$$

and the resulting (1D) convolution would be given by

$$K(x,\sigma) = \sum_{n=-\infty}^{\infty} g(n;\sigma) f(x-n) \quad \text{for} \quad x \in \mathbb{Z},\ \sigma > 0$$

The reality of the matter is that a discretely sampled Gaussian is not an appropriate kernel for creating discrete scale space. In [21] and in particular [20], Lindeberg demonstrated that the sampled Gaussian kernel violates not only semigroup property (axiom 2.3), but–much less forgivably–the causality property (axiom 2.3). There is absolutely no guarantee that convolution with a sampled Gaussian kernel will not create "spurious" structures as resolution increases.

Fortunately, Lindeberg was immediately able to remedy this by providing a discrete analogue of the Gaussian kernel, which does satisfy axiom 2.4 and axiom 2.3:

Definition 2.14 (Discrete Gaussian Kernel)**.** *The discrete Gaussian kernel, which can be shown to be a suitable generator for scale space, is given by*

$$T(n;\sigma) = e^{-\alpha\sigma} I_n(\alpha\sigma), \quad I_n(\sigma) = I_{-n}(\sigma) = (-1)^n J_n(i\sigma) \quad n \geq 0, \sigma, \alpha > 0 \tag{2.88}$$

where I_n are the modified Bessel functions of integer order based on the ordinary Bessel functions J_n, i.e.

$$I_n(x) = \sum_{m=0}^{\infty} \frac{1}{m!(m+n)!} \left(\frac{x}{2}\right)^{2m+n}, \quad n \geq 0$$

where we have taken the liberty of simplifying the typical definition [1] (which involves the gamma function), since we only desire Bessel functions of integer order. The

parameter α above is simply an optional scaling parameter which is simply set to 1 hereforth.

The derived family of 1D signals is then given by

$$K(x,\sigma) = \sum_{n=-\infty}^{\infty} T(n;t) f(x-n) \quad \text{for} \quad x \in \mathbb{Z},\ t > 0 \tag{2.89}$$

The compatibility of scale space theory and derivatives on discrete structures and extension to two dimensions was also demonstrated by Lindeberg in [22] and [23]. In particular, we may take derivatives of the convolutions of our discrete images using, say, a central difference. Lastly, the 2D version of the family given in eq. (2.89) can be obtained by independent convolution of its dimensions (i.e. it is separable). We will make these ideas explicit in chapter 3 and the Appendix.

With the ideas of scale established, we may return to our discussion of the Frangi filter.

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 [10] approached this problem by simply aggregating vesselness measure over all scales:

$$V(x_0, y_0) = \max_{\sigma \in \Sigma} V_\sigma(x_0, y_0) \tag{2.90}$$

where $\Sigma := \{\sigma_0, \sigma_1, \cdots, \sigma_N\}$ 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:

$$\mathsf{V}_\Sigma := [V(x,y)]_{\substack{0\le x<M \\ 0\le y<N}} \tag{2.91}$$

Notably, Frangi [10] refrained from explicitly interpreting the probablility assigned by eq. (2.90); that is–whether a particular point (x, y) in the image definitely a vessel or not. Instead, he cautioned that the result should not be used as a segmentation method alone, and that the size of the vasculature cannot be determined rigorously from the filter alone.

However, for the purposes of obtaining an intermediate result, we wish to be final about the whole matter and ultimately say whether or not a pixel does in fact corresponds to a curvilinear structure. A straightforward enough approach is to simply threshold at some fixed value. The resulting matrix can be given in terms of either eq. (2.90) or eq. (2.91)

$$V_{\Sigma,\alpha}(x,y) = \begin{cases} 1 & \text{if} \quad V(x,y) \ge \alpha \\ 0 & \text{else} \end{cases}, \quad \alpha > 0 \text{ for } \alpha \text{ fixed.} \tag{2.92}$$

We will discuss alternatives methods of aggregating results from our multiscale method, as well as optimal values for parameters and scales in chapter 3. As a final note, we admit that any future extensions of this work (as will be discussed in chapter 6) should not hold too much stock in this thresholded result, and analyzing the raw vesselness score eq. (2.91), or even the un-merged scale-wise scores, would be far more rewarding.

All that remains to describe mathematically is how to actually calculate the derivatives of our images and deal with the ultimately discrete nature of our samples.

Calculating the 2D Hessian

According to section 2.4.3, we may calculate derivatives of our structure by calculating a gradient on our convolved image. Our method of calculating the gradient of a matrix uses a second-order accurate central difference, as in [9]. Specific implementation will be discussed in chapter 3.

We note in passing that we may take the derivative of the Gaussian kernel and then convolve it, and the effect will be the same as if we had taken the derivative subsequently [11]. This could offer some computational speedup if we wish to run this procedure on many samples and fixed scale sizes, although we have implemented our scale spaces in the conventional way, as discussed in chapter 3.

Convolution Speedup via FFT

In practice, the convolutions described above are very slow for large scales (σ), as the size of the kernel is very large. Instead, we will perform a fast Fourier transform, which requires only $\mathcal{O}\left(N \cdot \log_2 N\right)$ operations for a one dimension signal of length N, as compared to the N^2 operations required of a conventional discrete Fourier transform [11]. We will briefly outline the theory of 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:

$$f(t) = \sum_{-\infty}^{\infty} c_n e^{i\frac{2\pi n}{T}t}, \quad c_n = \frac{1}{T}\int_{-T/2}^{T/2} f(t)e^{-i\frac{2\pi n}{T}t}dt \tag{2.93}$$

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

$$F(\mu) := \mathscr{F}\{f(t)\} = \int_{-\infty}^{\infty} f(t)e^{i2\pi\mu}dt \tag{2.94}$$

An inverse transform will then recover our original signal:

$$f(t) = \mathscr{F}^{-1}\{F(\mu)\} = \int_{-\infty}^{\infty} F(\mu)e^{i2\pi\mu t}dt \tag{2.95}$$

Together, eq. (2.94) and eq. (2.95) 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., following [11]. 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 (or impulse) function, an infinite series of discrete impulses with equal spacing ΔT:

$$s_{\Delta T}(t) := \sum_{n=-\infty}^{\infty} \delta[t - n\Delta T] \,, \quad \delta[t] = \begin{cases} 1 \,, & t = 0 \\ 0 \,, & t \neq 0 \end{cases} \tag{2.96}$$

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

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

$$\tilde{f}(t) = f(t) s_{\Delta T}(t) \tag{2.97}$$

From this we can calculate $\tilde{F}(t)$. Given the discrete signal $\tilde{f}$, we construct the transform $\tilde{F}(\mu) = \mathscr{F}\{\tilde{f}(t)\}$. by expanding $\tilde{f}$ in the same infinite basis as the continuous case.

$$\tilde{F}(\mu) = \sum_{n=-\infty}^{\infty} f_n e^{-i2\pi\mu n\Delta T} \,, \quad f_n = \tilde{f}(n) = f(n\Delta T) \tag{2.98}$$

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

2D DFT Convolution Theorem .

Theorem 2.11 (2D DFT Convolution Theorem)**.** *Given two discrete functions are sequences with the same length.* $f(x, y)$ *and* $h(x, y)$ *for integers* $0 < x < M$ *and* $0 < y < N$*, we can take the*

discrete fourier transform (DFT) of each:

$$F(u,v) := \mathcal{D}\{f(x,y)\} = \sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y)e^{-2\pi i\left(\frac{ux}{M}+\frac{vy}{N}\right)} \tag{2.99}$$

$$H(u,v) := \mathcal{D}\{h(x,y)\} = \sum_{x=0}^{M-1}\sum_{y=0}^{N-1} h(x,y)e^{-2\pi i\left(\frac{ux}{M}+\frac{vy}{N}\right)} \tag{2.100}$$

and given the convolution of the two functions

$$(f \star h)(x,y) = \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} f(m,n)h(x-m,y-n) \tag{2.101}$$

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

$$(f \star h)(x,y) = \mathcal{D}^{-1}\{MN \cdot F(u,v)H(u,v)\} \tag{2.102}$$

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

Proof.

$$(f \star h)(x,y) = \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} f(m,n)h(x-m,y-n) \tag{2.103}$$

$$= \sum_{m=0}^{M-1}\sum_{n=0}^{N-1}\left(\sum_{p=0}^{M-1}\sum_{q=0}^{N-1} F(p,q)e^{2\pi i\left(\frac{mp}{M}+\frac{nq}{N}\right)}\right)\left(\sum_{u=0}^{M-1}\sum_{v=0}^{N-1} H(u,v)e^{2\pi i\left(\frac{u(x-m)}{M}+\frac{v(y-n)}{N}\right)}\right) \tag{2.104}$$

$$= \left(\sum_{u=0}^{M-1}\sum_{v=0}^{N-1} H(u,v)e^{2\pi i\left(\frac{ux}{M}+\frac{vy}{N}\right)}\right)\left(\sum_{p=0}^{M-1}\sum_{q=0}^{N-1} F(p,q)\left(\sum_{m=0}^{M-1} e^{2\pi i\left(\frac{m(p-u)}{M}\right)}\right)\left(\sum_{n=0}^{N-1} e^{2\pi i\left(\frac{n(q-v)}{N}\right)}\right)\right) \tag{2.105}$$

$$= \left(\sum_{u=0}^{M-1}\sum_{v=0}^{N-1} H(u,v)e^{2\pi i\left(\frac{ux}{M}+\frac{vy}{N}\right)}\right)\left(\sum_{p=0}^{M-1}\sum_{q=0}^{N-1} F(p,q)\left(M\cdot\hat{\delta}_M(p-u)\right)\left(N\cdot\hat{\delta}_M(q-v)\right)\right) \tag{2.106}$$

$$= \left(\sum_{u=0}^{M-1}\sum_{v=0}^{N-1} H(u,v)e^{2\pi i\left(\frac{ux}{M}+\frac{vy}{N}\right)}\right)\cdot MNF(u,v) \tag{2.107}$$

$$= MN\cdot\sum_{u=0}^{M-1}\sum_{v=0}^{N-1} F(u,v)H(u,v)e^{2\pi i\left(\frac{ux}{M}+\frac{vy}{N}\right)} \tag{2.108}$$

$$= MN\cdot\mathcal{D}^{-1}\{FH\} \tag{2.109}$$

where

$$\hat{\delta}_N(k) = \begin{cases} 1 & \text{when } k = 0 \mod N \\ 0 & \text{else} \end{cases} \tag{2.110}$$

□

Above, we make use of the following lemma

Lemma 2.12. *Let j and k be integers and let N be a positive integer. Then*

$$\sum_{n=0}^{N-1} e^{2\pi i\left(\frac{n(j-k)}{N}\right)} = N\cdot\hat{\delta}_N(j-k) \tag{2.111}$$

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

$$z^N - 1 \;=\; (z-1)\Big(z^{n-1} + \cdots + z + 1\Big) \;=\; (z-1)\sum_{n=0}^{N-1} z^n. \tag{2.112}$$

thus giving us

$$0 = \Big(e^{2\pi i(j-k)/N} - 1\Big)\sum_{n=0}^{N-1} e^{2\pi in(j-k)/N} \tag{2.113}$$

To prove the claim in eq. (2.111), we consider two cases: First, if $j-k$ is a multiple of N, we of course have $e^{2\pi in(j-k)/N} = \Big(e^{2\pi i}\Big)^{n(j-k)/N} = 1$ and thus the left side of eq. (2.111) reduces to

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

.

In the case that $j-k$ is *not* a multiple of N, we refer to eq. (2.113). The first factor is not zero since, $\Big(e^{2\pi i(j-k)/N}\Big) \neq 1$ (simply since $(j-k)/N$ is not an integer), and thus it must be that the second factor is 0:

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

We can combine these two cases by invoking the definition of eq. (2.110), giving us the result. □

FFT

As noted, the above result applies to the Discrete Fourier Transform. We actually achieve a convolution speedup using a Fast Fourier Transform (FFT) instead. We follow the developments of [11]. 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:

$$F(\mu,\nu) = \sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y)e^{-i2\pi(\mu x/M+\nu y/N)} \tag{2.114}$$

$$= \sum_{x=0}^{M-1} e^{-i2\pi\mu x/M}\left[\sum_{y=0}^{N-1} f(x,y)e^{-i2\pi\nu y/N}\right] \tag{2.115}$$

$$= \sum_{x=0}^{M-1} e^{-i2\pi\mu x/M}\,\mathscr{F}_x\{f(x,y)\} \tag{2.116}$$

$$= \mathscr{F}_y\{\mathscr{F}_x\{f(x,y)\}\} \tag{2.117}$$

where $\mathscr{F}_{x'}$ 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 $x \in 0,\cdots,M-1$. 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 $(MN)^2$ We'll also mention that 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

$$F(u) = \sum_{x=0}^{M-1} f(x)W_M^{ux}, \quad \text{where} \quad W_m := e^{-i2\pi/m} \tag{2.118}$$

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. (2.118) and split it into odd and even entries in the summation

$$F(u) = \sum_{x=0}^{2K-1} f(x) W_{2K}^{ux} \tag{2.119}$$

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

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

$$W_{(2m)}^{(2n)} = e^{\frac{-i2\pi(2m)}{2m}} = e^{\frac{-i2\pi m}{n}} = W_m^n \tag{2.121}$$

$$W_m^{(u+m)x} = e^{\frac{-i2\pi(u+m)x}{m}} = e^{\frac{-i2\pi unx}{m}} e^{\frac{-i2\pi mx}{m}} = e^{\frac{-i2\pi ux}{m}}(1) = W_m^{ux} \tag{2.122}$$

$$W_{2m}^{(u+m)} = e^{\frac{-i2\pi(u+m)}{2m}} = e^{\frac{-i2\pi ux}{2m}} e^{-i\pi} = W_{2m}^u e^{-i\pi} = -W_{2m}^u \tag{2.123}$$

Thus we can rewrite eq. (2.120) as

$$F(u) = \sum_{x=0}^{K-1} f(2x) W_{2K}^{2ux} + \sum_{x=0}^{K-1} f(2x+1) W_{2K}^{2ux} W_{2K}^u \tag{2.124}$$

$$\implies \quad F(u) = \left(\sum_{x=0}^{K-1} f(2x) W_K^{ux} \right) + \left(\sum_{x=0}^{K-1} f(2x+1) W_K^{ux} \right) W_{2K}^u \tag{2.125}$$

The major advance comes via using the identities eq. (2.121) to consider the Fourier transform K frequencies later :

$$F(u+K) = \left(\sum_{x=0}^{K-1} f(2x) W_K^{(u+K)x} \right) + \left(\sum_{x=0}^{K-1} f(2x+1) W_K^{(u+K)x} \right) W_{2K}^{(u+K)} \tag{2.126}$$

$$\implies F(u+K) = \left(\sum_{x=0}^{K-1} f(2x) W_K^{ux} \right) - \left(\sum_{x=0}^{K-1} f(2x+1) W_K^{ux} \right) W_K^u \tag{2.127}$$

Comparing eq. (2.125) and eq. (2.127), 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:

$$\mathscr{D}_u\{f_{\text{even}}(t)\} := \sum_{x=0}^{K-1} f(2x) W_K^{ux} \tag{2.128}$$

$$\mathscr{D}_u\{f_{\text{odd}}(t)\} := \sum_{x=0}^{K-1} f(2x+1) W_K^{ux} \tag{2.129}$$

If we're calculating an M point transform (i.e. we're wishing to calculate $F(1),...,F(M)$), once we've calculated the first K discrete frequencies (i.e. $F(1),\cdots,F(K)$) we may simply reuse the two values we've calculated in eq. (2.128) to calculate the next $F(K+1),..,F(K+K) = F(M)$. 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 $F(1),..,F(M)$. 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 $M\log_2 M$.

Of course, since eq. (2.128) 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 eq. (2.114).

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 that you just do the part that's a power of 2 and then do the rest manually or pick a different power.

Finally we note the inverse DFT can actually be found via a DFT of the complex conjugate of the original signal, and of course we may translate that operation to a FFT.

CHAPTER 3

IMPLEMENTATIONS

Calculating the Hessian

Pseudocode for `np.gradient` which is used in calculating Hessian (code below)

```
gaussian_filtered = fftgauss(image, sigma=sigma)

Lx, Ly = np.gradient(gaussian_filtered)

Lxx, Lxy = np.gradient(Lx)

Lxy, Lyy = np.gradient(Ly)
```

CHAPTER 4

RESEARCH PROTOCOL

Samples / Image Domain

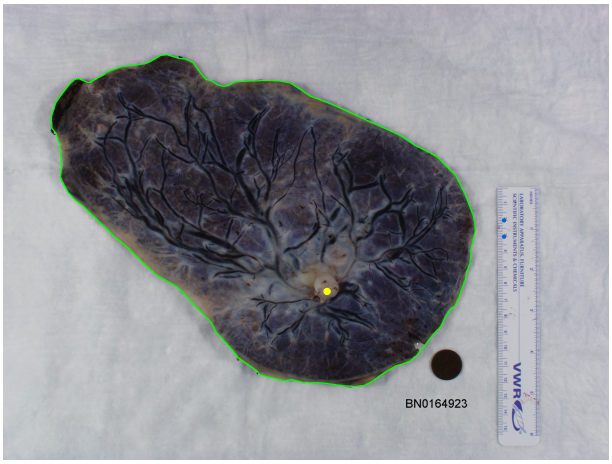
We ultimately perform a PCSVN extraction on a subset of 201 color placental images from a private database provided by the National Children's Study, which had been prepared for a different study. A detailed description of the data set is given in [5], and a description of the cleaning and fixing procedure is given in [2]. The samples are provided as XCF files (the native project file for GIMP) and contain four major layers.

A representative sample

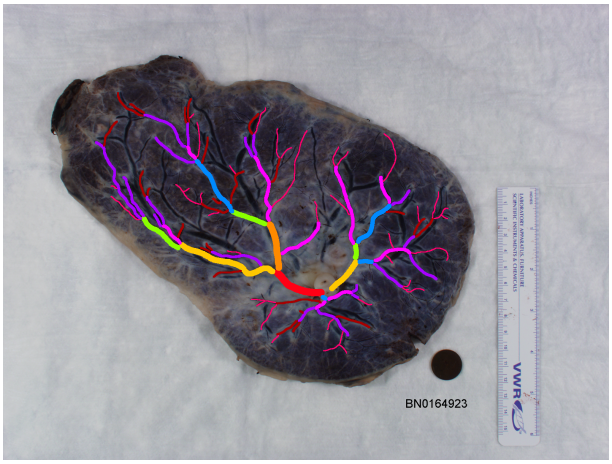
The layers together give a hand tracing of the vascular network and perimeter. A sample of overlaid layers in a representative sample (with ID number "BN0164923") is given in fig. 4.

Each layer is roughly 1954x1200 pixels (with some subtle variation). In fig. 4a, a cleaned, fixed placenta is placed on a table with a camera a fixed distance away, and a rule and penny (presumably for redundancy) to aid registration and calibration of the resolution. fig. 4b is a tracing (in green) of the perimeter of the placenta. The point of umbilical cord insertion is notated in yellow. Two cyan marks are placed on consecutive centimeter markings on the ruler (the dots are enlarged and shown as a darker blue here for clarity). fig. 4c and fig. 4d are both hand traces of the PCSVN, with a layer for each the arteries and veins. These layers are simultaneously overlain on the base image infig. 4e. The coloration is meant to indicate the diameter of each vessel. The diameters are binned into 9 discrete widths, odd integers from 3 to 19 pixels. Vessels of smaller diameter are either binned to three or (quite frequently) left untraced. The

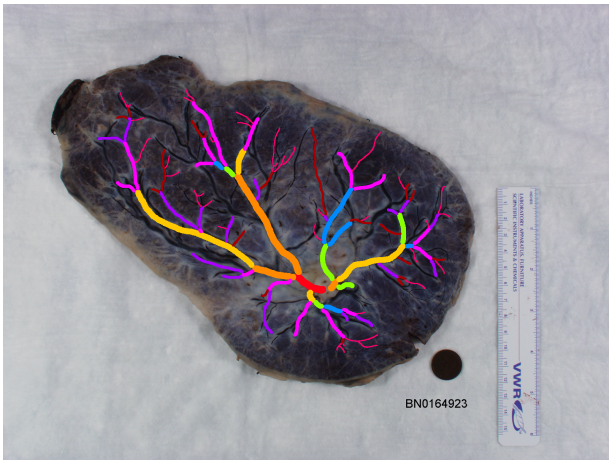
(a) Fixed Placental Sample



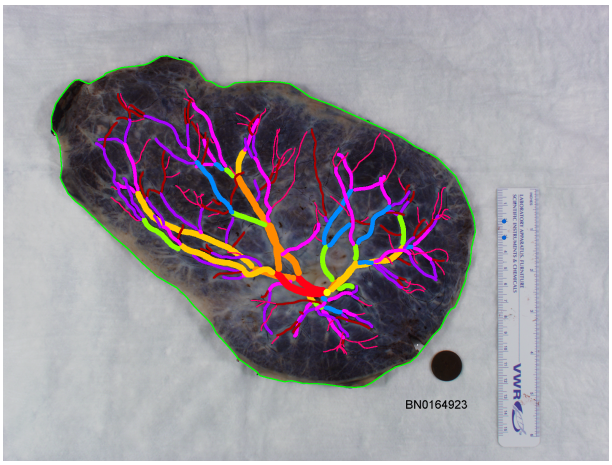
(b) Perimeter tracing



(c) Arterial tracing



(d) Venous tracing



(e) Total Vascular Network

FIGURE 4: A representative placental sample and tracing

vessel width	color (hex value)	color name
3 pixels	#ff006f	magenta
5 pixels	#a80000	dark red
7 pixels	#a800ff	purple
9 pixel s	#ff00ff	light pink
11 pixels	#008aff	blue
13 pixels	#8aff00	green
15 pixels	#ffc800	gold
17 pixels	#ff8a00	orange
19 pixels	#ff0015	bright red

TABLE 1: Vessel width color code

correspondence between pencil color and (binned) vessel width is given in table 1.

All in all, these hand-traced and rather labor intensive–requiring between 4 and 8 hours to trace a single sample. A closer look at many of the samples often reveals that a great deal of subjectivity in providing this "ground truth," as it is not often clear what the underlying truth really is; often it's hard to see where the vein is, vascular networks are obscured by the umbilical stem, the blood in the vessels dries unevenly or ruptures, and the vessel seems to disappear momentarily. These situations and more will be showcased in our results section, where we will discuss methods to simulate the subjectivity of decision.

Knowns and Unknowns

Of course, we wish to simply operate on the placental sample itself, without any understanding of its provided tracing (except for judging the strength of our algorithm); our goal is to develop an algorithm that can produce a "ground truth" tracing similar to fig. 4e or fig. 5d without any user intervention.

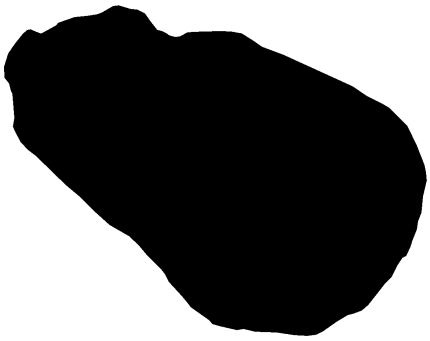
For our purposes however, we will use a limited amount of information from the tracings, namely the provided placental perimeter (shown in green in fig. 4). In developing a fully automated algorithm, it would be relatively straightforward to obtain this boundary ourselves using various techniques, such as an Active Contour Model [26] or, or even a simple edge finding algorithm followed by watershedding and largest object selection as in [13]. We leave that for future work. We do use the traced placental perimeter at our own peril, however, since often there are tears in the side of the plate or large amounts of non-vascular content with large changes in height that are not adequately accounted for in the perimeter tracing.

Finally, we will consider the location of the umbilical insertion point as a "known", as the vessels around it are frequently impossible to see and we wish to exclude them from consideration. It is not unreasonable, however, to consider this to be a known–in future preparations of samples, we could simply require that this point be centered in image in a predictable location. Furthermore, we use its location as a convenience in data analysis–knowledge of this point does not inform our algorithm at all.

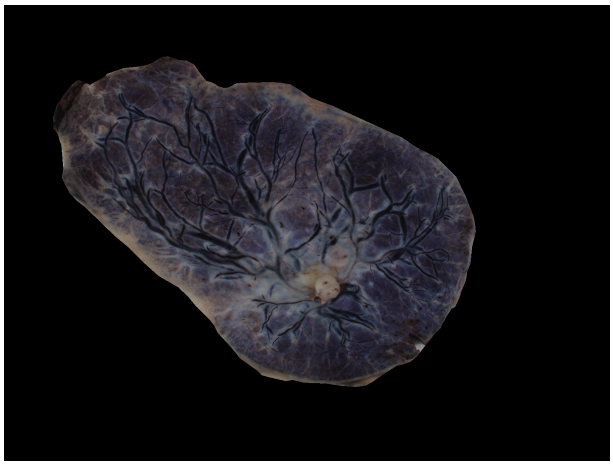
Data Cleaning and Preprocessing

Building a sample suitable for use in our algorithm from fig. 4 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. Our preprocessed samples used by the algorithm are given in section 4.2.

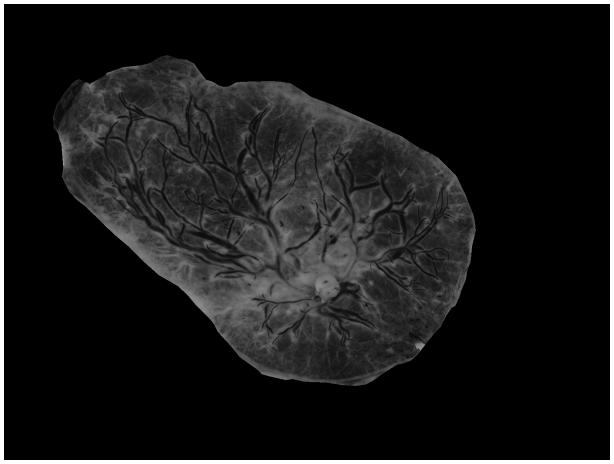
These procedures are performed automatically on the 201 images in our data set using a custom GIMP plug-in, which performs various "bucket fill" operations, layer mergings, and thresholdings. For completeness sake, this plug-in (and an associated Scheme script which turns it into a batch operation) can be found in the Appendix.



(a) Background Mask (in white)



(b) Sample with BG removed



(c) Grayscale

(d) Trace / "Ground Truth"

FIGURE 5: Preprocessed files from an NCS sample

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 N-dimensional input [10], 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 [15], or "luminance" transform:

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

It should be noted that this choice is not automatic–several other attempts have used the green channel unmodified, as in [2] and [13].

Boundary Dilation

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 potentially useful to adapt such methods in a way to, say, calculate 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 "zero out" unwanted background pixels and simply exclude adjacent areas from consideration. This excluding function, `plate_morphology.dilate_plate`, ultimately relies on two functions provided by the Python library `scikit-image` [30]. 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 (which should have the same affect as using the perimeter labeled in fig. 4b directly, though we've chosen to not include that in our sample). That boundary itself is then

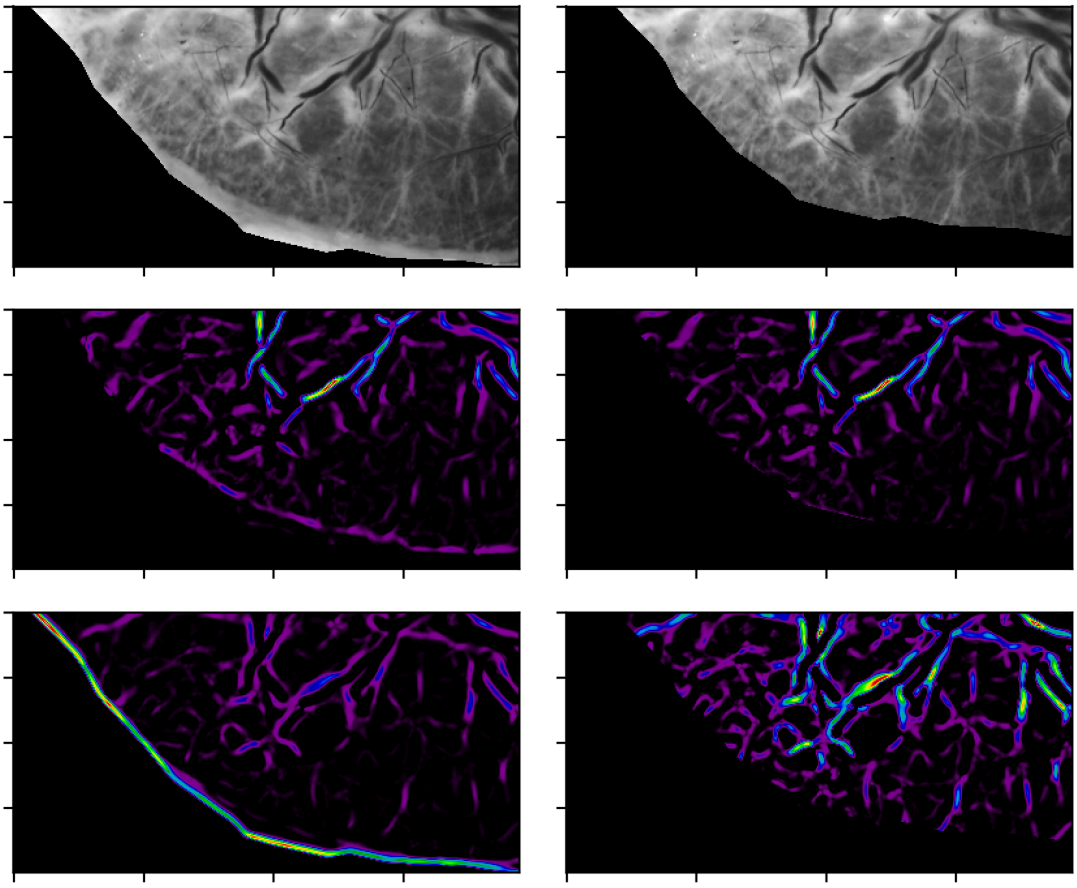


FIGURE 6: Effect of boundary dilation on Frangi responses

dilated by the desired factor. The second is a much quicker implementation of binary dilation that is particularly efficient for our problem: we iterate through an array of indices for the image where the boundary occurs and simply extend the mask R pixels in each direction (like a giant plus sign). Since these pixels are all connected, the effect is very similar to convolving with a disk of radius R, but is much faster.

fig. 6 shows the effect of this so-called "boundary dilation." In the image above, $\sigma = 3$ and border radius is 25 to exaggerate the effect. The first row shows the unaltered boundary of the sample (left) and the sample after boundary dilation (with radius dilation of 25 pixels). The second row shows the Frangi vesselness measure at single

scale ($\sigma = 3$) where `DARK_BG=False` to target dark curvilinear structures performed on the altered sample (left) and the boundary dilated sample (right). Removing an unnecessary part of the placental plate prevents a small response to a non-vascular yet mildly curvilinear background feature from appearing. The third row of fig. 6 shows the Frangi vesselness measure at the same scale ($\sigma = 3$) when we are probing for bright curvilinear structures (i.e. `DARK_BG=True`. Here, wherever the very edge of the placental plate is *any* brighter than adjacent interior, a very large Frangi response will occur, as seen on the left. Dilating the boundary completely avoids this issue, as seen by the figure on the right. Thus we prevent a visual artifact that is present in much prior work on this problem (see [13], [2]). It should be noted that, while the figure on the right shows a much larger interior response, this is simply because the intensity of the output in each of these images is being independently scaled between the minimum and maximum intensity in the image. However, we argue that this is an appropriate and desired depiction of the situation, as we will frequently consider only the relative maxima of Frangi response per scale in our analysis.

We end our discussion by noting that we perform this boundary dilation within the Frangi algorithm itself when we set the structureness parameter γ as half of the maximum Hessian norm found at that scale–this ensures that the maximum occurs sufficiently away from the boundary of the plate.

The code for generating fig. 6 is found in the within the "`if __name__ == __main__`" block of the file `plate_morphology.py`, (so the figure will be generated when running `plate_morphology.py` as a top-level script from the command line). See appendix.

Deglaring

Despite best efforts when harvesting samples, some placental images have substantial glare, which leads to inaccuracies in identifying curvilinear content. Our protocol for deglaring is analogous to that performed in [2] and [13]. Unfortunately, the method relied upon by those previous papers (`MATLAB`'s `imfill`, which relies on inpainting by solving the Dirichlet problem for masked regions) was not immediately available in a Python environment. Instead, we used an already implemented inpainting algorithm, `scikit-image`'s `inpaint_biharmonic()`, which should be expected to achieve similar results, at the expense of processing time.

The function `inpaint_biharmonic` is based on [7], and relies on solving a biharmonic equation i.e. $\nabla\nabla f = 0$ for the surface f subject to boundary conditions (as compared to `imfill`'s solving the Laplace equation $\nabla f = 0$ in regions marked as glare.

The method for deciding what is considered glare is similar to [2], in which we consider any intensities close the maximum intensity in the image (Almoussa et al. used 80% of max intensity, and we use $175/255 \approx 68\%$). This threshold is dependent on the image domain.

Inpainting in the above way is rather resource intensive, so we implemented two faster and less precise methods of inpainting, which can be found in `preprocessing.py`. The first, called `inpaint_glare()` (change this name) replaces any masked pixel with the average of all non-masked values within a certain distance (default 15 pixels). The second, called `inpaint_with_boundary_median` calculates the median value of the (non-masked) boundary and fills any masked region with that value. We argue that these less-exact methods are adequate for smaller regions, while larger regions of glare deserve a more thoughtful application of inpainting. Our final method of inpainting, `inpaint_hybrid` implements this idea–smaller glare regions are inpainted with a boundary median, while larger areas are inpainted with the more expensive but more

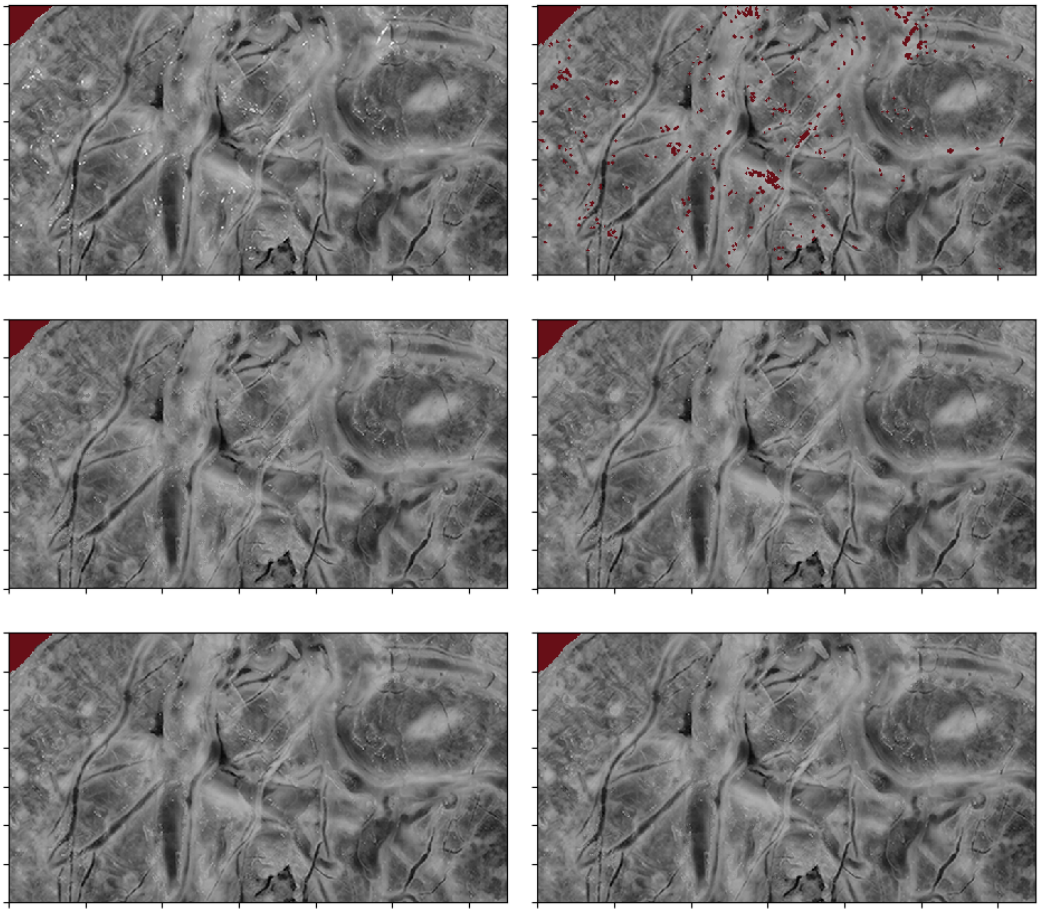


FIGURE 7: Deglaring a sample using a hybrid inpainting method

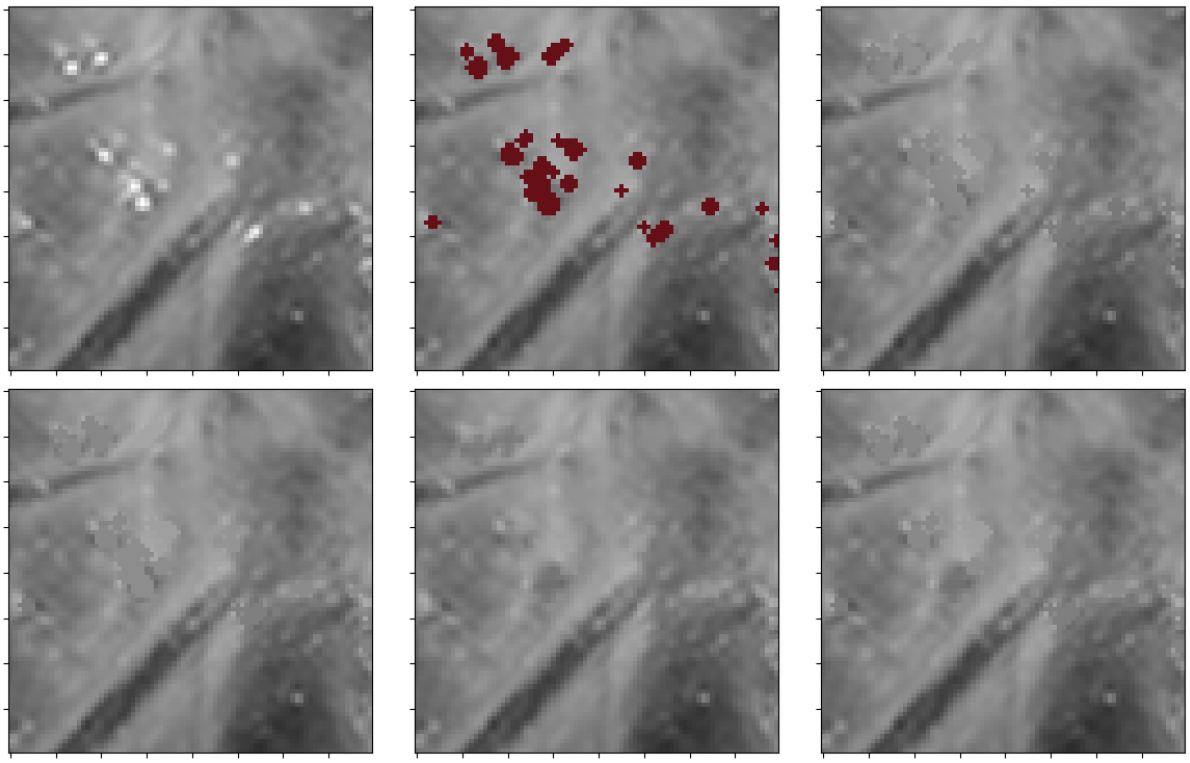


FIGURE 8: Comparison of glare inpainting methods (detail)

accurate biharmonic inpainting.

A comparison of these methods is shown in fig. 7, and a zoomed in portion is shown for in fig. 8. In the top left, the glary image is shown. In the top middle, regions above the threshold intensity are masked (shown in dark red, along with the background). In the top right, the strategy is "mean window" with a window size of 15 pixels. The bottom left uses "boundary median" strategy. The middle is the more expensive "biharmonic inpainting" strategy, and the bottom right uses a "hybrid" strategy.

The following timing demonstrates that the "hybrid" strategy is over 3 times faster than biharmonic inpainting, and that biharmonic painting takes 22 seconds, even when only 1% of the placental plate is to be inpainted.

```
In [1]: %timeit inpaint_with_boundary_median(img)
1 loop, best of 3: 3.99 s per loop

In [2]: %timeit inpaint_with_biharmonic(img)
1 loop, best of 3: 22.3 s per loop

In [3]: %timeit inpaint_hybrid(img)
1 loop, best of 3: 6.49 s per loop

In [4]: (np.logical_and(masked.mask, np.invert(img.mask))).sum()
Out[4]: 10055

In [5]: np.invert(img.mask).sum()
Out[5]: 878591

In [6]: 10055/878591
Out[6]: 0.011444460505513942

```

We stress again that only a small subset our image domain exhibits disruptive amounts of glare. Future improvements in this direction should probably seek to implement more robust method such as [18] that is not dependent on an arbitrary global threshold for deciding what regions exhibit glare.

Where should this go?

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 $\Sigma := \{\sigma_1, \sigma_2, \ldots, \sigma_N\}$.

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 [16] it is reasonable and natural to select these logarithmically; that is, for some selected inputs $m < M$ we have

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

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}

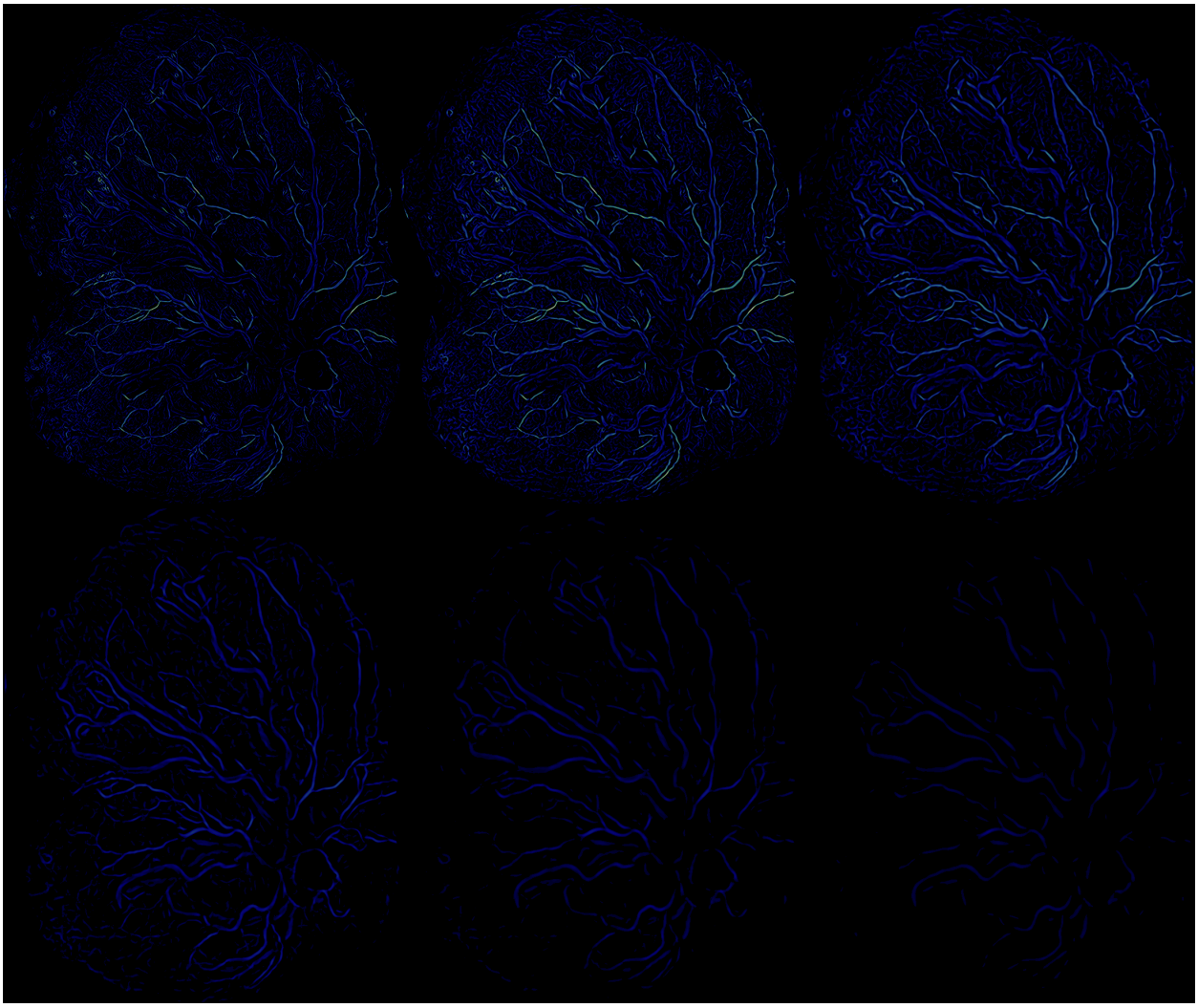


FIGURE 9: Frangi vesselness score at several scales

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

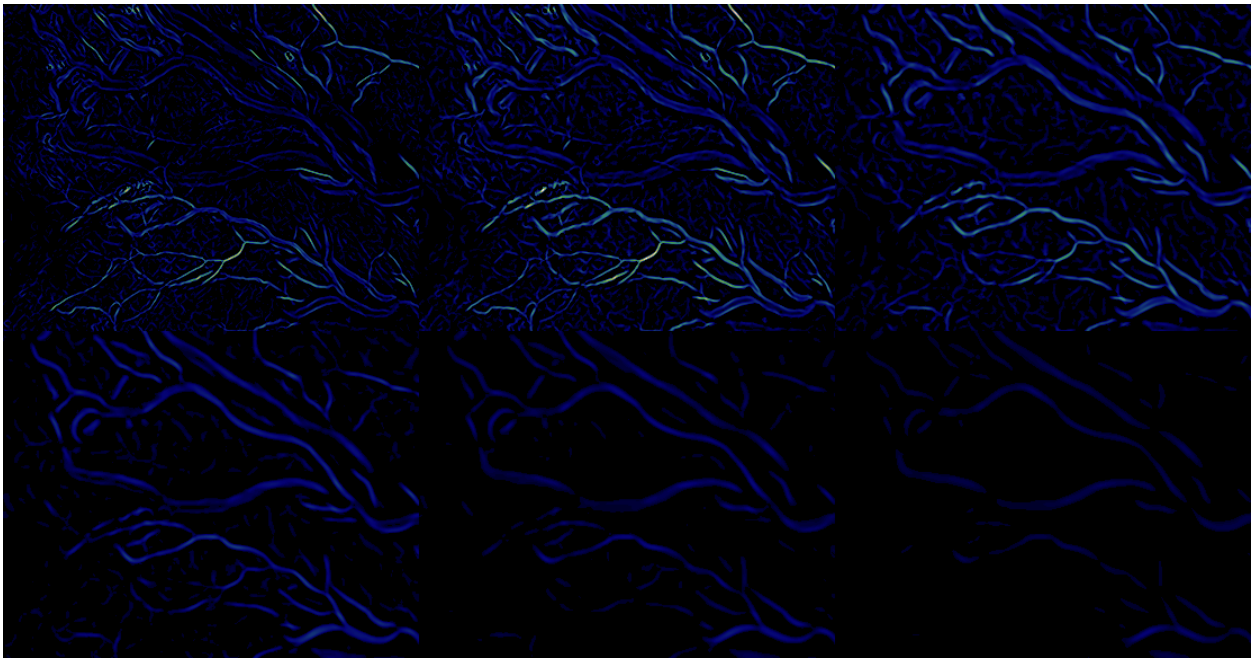


FIGURE 10: Frangi vesselness score at several scales (inset)

Erode plate / dilate boundary

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

CHAPTER 5

RESULTS AND ANALYSIS

use MCC [25]

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.

Variations in the Data Set and Imperfections of the Ground Truth

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 obviouslly 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. Umbillical cord insertion point is stupid and obscures a lot. The tracer guesses but

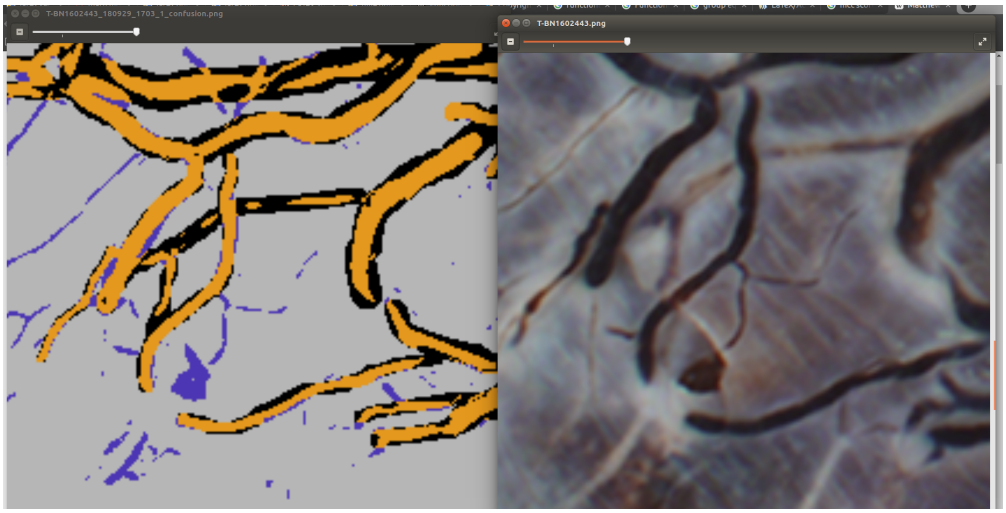


FIGURE 11: ”True” false positives and “False” false positives

there’s no real guiding principle AFAIK..

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

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) Try something like [19].
- Refine variable thresholding and automate.
- Combine with a ridge search.
- Use this as pre-processing for a Neural Network or something (cite kara's work, katalinas work)
- Apply to more image domains (STARE, WORSE PLACENTAS, ETC.)
- Automate Measurements (more quantative 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/add_margins.py

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

from skimage.filters import sobel
from frangi import frangi_from_image
from plate_morphology import dilate_boundary
from skimage.morphology import remove_small_holes, remove_small_objects
from merging import nz_percentile

s = sobel(img)
s = dilate_boundary(s, mask=img.mask, radius=20)
finv = frangi_from_image(s, sigma=0.8, dark_bg=True)

finv_thresh = nz_percentile(finv, 80)

margins = remove_small_objects((finv > ft).filled(0), min_size=32)

margins_added = remove_small_holes(np.logical_or(margins, approx),
                                   min_size=100, connectivity=2)

markers = np.zeros(img.shape, dtype=np.uint8)

markers[ Fmax < .1 ] = 1
markers[ margins_added] = 2

rw = random_walker(img, markers)
approx_rw = (rw==2)
confusion_rw = confusion(approx_rw, trace, bg_mask=ucip_mask)
mccs_rw = mccs(approx_rw, trace, bg_mask=ucip_mask)
pnc_rw = np.logical_and(skeltrace, rw2==2).sum() / skeltrace.sum()
```

listings/diffgeo.py

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


import numpy as np
import numpy.ma as ma

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

from hfft import fft_hessian


def principal_curvatures(img, sigma=1.0, H=None):
    """
    Return the principal curvatures { 1 ,  2 } of an image, that is, the
    eigenvalues of the Hessian at each point (x,y). The output is arranged such
    that |  1 | <= |  2 |.

    Input:

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

                PLEASE ADD SOME INFO HERE ABOUT WHAT SORT OF DTYPES ARE
                EXPECTED/REQUIRED, IF ANY

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

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

        (K1, K2):   A tuple where K1, K2 each are the exact dimension of the
                    input image, ordered in magnitude such that |  1 | <= |  2 |
                    in all locations. If *signed* option is used, then elements
                    of K1, K2 may be negative.

    Example:

        >>> K1, K2 = principal_curvatures(img)
        >>> K1.shape == img.shape
        True
        >>> (K1 <= K2).all()
        True

        >> K1.mask == img.mask
        True
    """

    # determine if multichannel
    multichannel = (img.ndim == 3)

    if not multichannel:
        # add a trivial dimension
        img = img[:,:,np.newaxis]

    K1 = np.zeros_like(img, dtype='float64')
    K2 = np.zeros_like(img, dtype='float64')

```

```
    for ic in range(img.shape[2]):

        channel = img[:,:,ic]

        # returns the tuple (Hxx, Hxy, Hyy)
        if H is None:
            H = hessian_matrix(channel, sigma=sigma)

        # returns tuple (l1,l2) where l1 >= l2 but this *includes sign*
        L = hessian_matrix_eigvals(*H)
        L = np.dstack(L)

        mag = np.argsort(abs(L), axis=-1)

        # just some slice nonsense
        ix = np.ogrid[0:L.shape[0], 0:L.shape[1], 0:L.shape[2]]

        L = L[ix[0], ix[1], mag]

        # now k2 is larger in absolute value, as consistent with Frangi paper

        K1[:,:,ic] = L[:,:,0]
        K2[:,:,ic] = L[:,:,1]

    try:
        mask = img.mask
    except AttributeError:
        pass
    else:
        K1 = ma.masked_array(K1, mask=mask)
        K2 = ma.masked_array(K2, mask=mask)

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

    return K1, K2

def reorder_eigs(L1,L2):
    """
    L1, L2 contain a 2D matrix of eigenvalues at each point
    so that L1 <= L2 at each element.

    this reorders this so that |L1| <= |L2| instead.

    this could (if desired) also return the permutation array
    but does not do so presently
    """

    L = np.dstack((L1,L2))
    mag =  np.argsort(abs(L), axis=-1)

    ix = np.ogrid[0:L.shape[0], 0:L.shape[1], 0:L.shape[2]]

    L = L[ix[0], ix[1], mag]

    # now L2 is larger in absolute value, as consistent with Frangi paper

    return (L[:,:,0] , L[:,:,1])

def principal_directions(img, sigma, H=None, mask=None):
    """
    will ignore calculation of principal directions of masked areas
```

```
127
128     mask should  be positive where the PD's should *NOT* be calculated
129     this function actually returns the theta corresponding to
130     leading and trailing principal directions, i.e. angle w / x axis
131     """
132
133     if H is None:
134         H = hessian_matrix(img, sigma)
135
136     Hxx, Hxy, Hyy = H
137
138
139     # determine if we have a mask or not (and try to get one off image if
140     # possible)
141     if mask is None:
142         try:
143             mask = img.mask
144         except AttributeError:
145             masked = False
146         else:
147             masked = True
148     else:
149         masked = True
150
151     dims = img.shape
152
153     # where to store
154     trailing_thetas = np.zeros_like(img, dtype='float64')
155     leading_thetas = np.zeros_like(img, dtype='float64')
156
157
158     # maybe implement a small angle correction
159     for i, (xx, xy, yy) in enumerate(np.nditer([Hxx, Hxy, Hyy])):
160
161         # grab the (x,y) coordinate of the hxx, hxy, hyy you're using
162         subs = np.unravel_index(i, dims)
163
164         # ignore masked areas (if masked array)
165         if masked and mask[subs]:
166             continue
167
168         h = np.array([[xx, xy], [xy, yy]]) # per-pixel hessian
169         l, v = eig(h) # eigenvectors as columns
170
171         # reorder eigenvectors by (increasing) magnitude of eigenvalues
172         v = v[:,np.argsort(np.abs(l))]
173
174         # angle between each eigenvector and positive x-axis
175         # arccos of first element (dot product with (1,0) and eigvec is already
176         # normalized)
177         trailing_thetas[subs] = np.arccos(v[0,0]) # first component of each
178         leading_thetas[subs] = np.arccos(v[0,1]) # first component of each
179
180     if masked:
181         leading_thetas = ma.masked_array(leading_thetas, mask)
182         trailing_thetas = ma.masked_array(trailing_thetas, mask)
183
184
185     return trailing_thetas, leading_thetas
186
187
188
189
190 if __name__ == "__main__":
```

```
191
192     pass
193
194     #from get_base import get_preprocessed
195     #import matplotlib.pyplot as plt
196     #from functools import partial
197     #from fpd import get_targets
198     #b = partial(plt.imshow, cmap=plt.cm.Blues)
199     #sp = partial(plt.imshow, cmap=plt.cm.spectral)
200     #s = plt.show
201
202     #import time
203
204     #img = get_preprocessed(mode='G')
205
206     #for sigma in [0.5, 1, 2, 3, 5, 10]:
207
208     #    print('-'*80)
209     #    print('  =',sigma)
210     #    print('calculating hessian H')
211
212     #    tic = time.time()
213     #    H = hessian_matrix(img, sigma=sigma)
214
215     #    toc = time.time()
216     #    print('time elapsed: ', toc - tic)
217     #    tic = time.time()
218     #    print('calculating hessian via FFT (F)')
219     #    h = fft_hessian(img, sigma)
220
221     #    toc = time.time()
222     #    print('time elapsed: ', toc - tic)
223     #    tic = time.time()
224     #    print('calculating principal curvatures for   ={}'.format(sigma))
225     #    K1,K2 = principal_curvatures(img, sigma=sigma, H=H)
226     #    toc = time.time()
227     #    print('time elapsed: ', toc - tic)
228     #    tic = time.time()
229     #    print('calculating principal curvatures for   ={} (fast)'.format(sigma))
230     #    k1,k2 = principal_curvatures(img, sigma=sigma, H=h)
231
232     #    toc = time.time()
233     #    print('time elapsed: ', toc - tic)
234     #    tic = time.time()
235
236     #    #####
237
238     #    print('calculating targets for   ={}'.format(sigma))
239     #    T = get_targets(K1,K2, threshold=False)
240
241     #    toc = time.time()
242     #    print('time elapsed: ', toc - tic)
243     #    tic = time.time()
244
245     #    print('calculating targets for   ={} (fast)'.format(sigma))
246     #    t = get_targets(k1,k2, threshold=False)
247
248     #    toc = time.time()
249     #    print('time elapsed: ', toc - tic)
250
251     #    ######
252
253     #    print('extending masks')
254
```

```
    #     # extend mask over nontargets items
    #     img1 = ma.masked_where( T < T.mean(), img)
    #     img2 = ma.masked_where( t < t.mean(), img)

    #     tic = time.time()
    #     print('calculating principal directions for   ={}'.format(sigma))
    #     T1,T2 = principal_directions(img1, sigma=sigma, H=H)
    #     toc = time.time()
    #     print('time elapsed: ', toc - tic)
    #     tic = time.time()

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

listings/extract_NCS_pcsvn.py

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

"""
This is the main program. It approximates the PCSVN


"""

from placenta import (get_named_placenta, cropped_args, cropped_view,
                      list_placentas, list_by_quality, open_typefile,
                      open_tracefile, add_ucip_to_mask, measure_ncs_markings)

from merging import nz_percentile, apply_threshold
from scoring import (compare_trace, rgb_to_widths, merge_widths_from_traces,
                     filter_widths, mcc, confusion, skeletonize_trace)

from pcsvn import extract_pcsvn, scale_label_figure, get_outname_lambda
from preprocessing import inpaint_hybrid

import numpy as np
import numpy.ma as ma

import matplotlib.pyplot as plt

import os.path
import os
import json
import datetime

# for some post_processing, this needs to be moved elsewhere
from skimage.filters import sobel
from frangi import frangi_from_image
from plate_morphology import dilate_boundary
from skimage.morphology import remove_small_holes, remove_small_objects
from skimage.segmentation import random_walker



# INITIALIZE SAMPLES ____________________________________________________

# initialize a list of samples (several different ways)
#placentas = list_by_quality(0)
placentas = list_placentas('T-BN')  # load allllll placentas
#placentas = list_by_quality(json_file='manual_batch.json')
```

```
#placentas = ['T-BN0204423.png'] # for a single sample, use a 1 element list.

n_samples = len(placentas)

# RUNTIME OPTIONS ____________________________________________________________

MAKE_NPZ_FILES = True # pickle frangi targets if you can
USE_NPZ_FILES = True  # use old npz files if you can
NPZ_DIR = 'output/181117-deglared' # where to look for npz files
OUTPUT_DIR = 'output/181117-deglared' # where to save outputs



# EXTRACT_PCSVN OPTIONS ______________________________________________________

# find bright curvilinear structure against a dark background -> True
# find dark curvilinear structure against a bright background -> False
# DARK_BG -> ignore and return signed Frangi scores
DARK_BG = False

# along with the above, this will return "opposite" signed frangi scores.
# if this is True, then DARK_BG controls the "polarity" of the filter.
# See frangi.get_frangi_targets for details.
SIGNED_FRANGI = False

# do not calculate hessian scores close to the boundary (this is important
# mainly in terms of ensuring that the hessian is very large on the edge of
# the plate (which would influence gamma calculation)
DILATE_PER_SCALE = True

# use preprocessing.inpaint_with_boundary_median() to replace high
# glare regions
REMOVE_GLARE = True

log_range = (-3, 6)
n_scales = 40

# when showing "large scales only", this is where to start
# (some index between 0 and n_scales)
LO_offset = 24


scales = np.logspace(log_range[0], log_range[1], num=n_scales, base=2)
alphas = [0.15 for s in scales]
betas = None  # will be given default parameters
gammas =  None  # will be given default parameters

mccs = dict()  # empty dict to store MCC's of each sample
pncs = dict() # empty dict to store percent network covered for each sample

if not os.path.exists(OUTPUT_DIR):
    os.makedirs(OUTPUT_DIR)

print(n_samples, "samples total!")

for i, filename in enumerate(placentas):

    print('*'*80)
    print(f'extracting PCSVN of {filename}\t ({i} of {n_samples})')

    if USE_NPZ_FILES:
        # find the first npz file with the sample name in it in the
        # specified directory.
        stub = filename.rstrip('.png')
```

```
        for f in os.scandir(NPZ_DIR):
            if f.name.endswith('npz') and f.name.startswith(stub):
                npz_filename = os.path.join(NPZ_DIR, f.name)
                print(f'using the npz file {npz_filename}')
                break # just use the first one.
        else:
            print(f'no npz file found for {filename}.')
            npz_filename = None
    else:
        npz_filename = None

    # set a lambda function to make output file names
    outname = get_outname_lambda(filename, output_dir=OUTPUT_DIR)

    raw_img = get_named_placenta(filename, maskfile=None)
    if npz_filename is not None:
        F = np.load(npz_filename)['F']
        if REMOVE_GLARE:
            img = inpaint_hybrid(raw_img)
        else:
            img = raw_img
        print('successfully loaded the frangi targets!')

    else:
        print('finding multiscale frangi targets')

        F, img = extract_pcsvn(filename, DARK_BG=DARK_BG, alphas=alphas,
                               betas=betas, scales=scales, gammas=gammas,
                               kernel='discrete', dilate_per_scale=True,
                               verbose=False, signed_frangi=SIGNED_FRANGI,
                               generate_json=True, output_dir=OUTPUT_DIR,
                               remove_glare=REMOVE_GLARE)

        if MAKE_NPZ_FILES:
            npzfile = ".".join((outname("F").rsplit('.', maxsplit=1)[0],'npz'))
            print("saving frangi targets to ", npzfile)
            np.savez_compressed(npzfile, F=F)

    crop = cropped_args(img) # these indices crop out the mask significantly

    print("...making outputs")

    # get the 99% frangi filter score
    print("rewriting alphas with 1% scores")

    p_alphas = [nz_percentile(F[:,:,k],95.0) for k in range(n_scales)]
    alphas = np.array(p_alphas)

    approx, labs = apply_threshold(F, alphas, return_labels=True)

    # get the main tracefile
    trace = open_tracefile(filename, as_binary=True)

    ucip_midpoint, resolution = measure_ncs_markings(filename=filename)

    Fmax = F.max(axis=-1)

    # print(f"The umbilicial cord insertion point is at {ucip_midpoint}")
    # print(f"The resolution of the image is {resolution} pixels per cm.")

    # mask anywhere close to (within 90px L2 distance of) the UCIP
    # this is imperically how the it is, although could be bigger
    ucip_mask = add_ucip_to_mask(ucip_midpoint, radius=90, mask=img.mask)

```

```
    # open up RGB tracefiles (keep for visualizing?)
    A_trace = open_typefile(filename, 'arteries')
    V_trace = open_typefile(filename, 'veins')

    skeltrace = skeletonize_trace(A_trace, V_trace)

    # matrix of widths of traced image
    widths = merge_widths_from_traces(A_trace, V_trace, strategy='arteries')
    # min_widths = merge_widths_from_traces(A_trace, V_trace,
    #                                       strategy='minimum')

    # trace ignoring largest vessels (19 pixels wide)
    # trace_smaller_only = filter_widths(min_widths, min_width=3, max_width=17)
    # trace_smaller_only != 0
    # use limited scales
    approx_L0, labs_L0 = apply_threshold(F[:,:, L0_offset:], alphas[L0_offset:])

    # fix labels to incorporate offset
    labs_L0 = (labs_L0 != 0)*(labs_L0 + L0_offset)

    # confusion matrix against default trace
    confuse = confusion(approx, trace, bg_mask=ucip_mask)
    confuse_L0 = confusion(approx_L0, trace,
                                    bg_mask=ucip_mask)

    m_score, counts = mcc(approx, trace, ucip_mask, return_counts=True)
    m_score_L0, counts_L0 = mcc(approx_L0, trace, ucip_mask,
                                return_counts=True)

    # this all just verifies that the 4 categories were added up
    # correctly and match the total number of pixels in the reported
    # placental plate.
    TP, TN, FP, FN = counts # return these for more analysis?

    total = np.invert(ucip_mask).sum()
    print('TP: {}\t TN: {}\nFP: {}\tFN: {}'.format(TP,TN,FP,FN))
    print('TP+TN+FP+FN={}\ttotal pixels={}'.format(TP+TN+FP+FN,total))

    # MOVE THIS ELSEWHERE
    s = sobel(img)
    s = dilate_boundary(s, mask=img.mask, radius=20)
    finv = frangi_from_image(s, sigma=0.8, dark_bg=True)
    finv_thresh = nz_percentile(finv, 80)
    margins = remove_small_objects((finv > finv_thresh).filled(0), min_size=32)
    margins_added = np.logical_or(margins, approx)
    margins_added = remove_small_holes(margins_added, min_size=100,
                                       connectivity=2)
    # random walker markers
    markers = np.zeros(img.shape, dtype=np.uint8)
    markers[Fmax < .1] = 1
    markers[margins_added] = 2
    rw = random_walker(img, markers, beta=1000)
    approx_rw = (rw==2)
    confuse_rw = confusion(approx_rw, trace, bg_mask=ucip_mask)
    m_score_rw = mcc(approx_rw, trace, ucip_mask)
    pnc_rw = np.logical_and(skeltrace, approx_rw).sum() / skeltrace.sum()

    mccs[filename] =  (m_score, m_score_L0, m_score_rw)

    print(f'mcc score of {m_score:.3} for {filename}')
    print(f'mcc score of {m_score_L0:.3} with larger sigmas only')
    print(f'mcc score of {m_score_rw:.3} after random walker')

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

```
    plt.imsave(outname('1_img'), img[crop].filled(0), cmap=plt.cm.gray)
    plt.imsave(outname('4_confusion'), confuse[crop])
    plt.imsave(outname('7_confusion_LO'), confuse_LO[crop])
    plt.imsave(outname('9_confusion_rw'), confuse_rw[crop])

    percent_covered = np.logical_and(skeltrace, approx).sum() / skeltrace.sum()
    percent_covered_LO = np.logical_and(skeltrace, approx_LO).sum() / skeltrace.sum()

    pncs[filename] = (percent_covered, percent_covered_LO, pnc_rw)

    print('percentage of skeltrace covered:', f'{percent_covered:.2%}')
    print('percentage of skeltrace covered (larger sigmas only):',
          f'{percent_covered_LO:.2%}')
    print('percentage of skeltrace covered (random_walker):',
          f'{pnc_rw:.2%}')
    plt.imsave(outname('5_coverage'), confusion(approx, skeltrace)[crop])
    plt.imsave(outname('8_coverage_LO'), confusion(approx_LO, skeltrace)[crop])
    plt.imsave(outname('9_coverage_rw'), confusion(approx_rw, skeltrace)[crop])

    # only save the colorbar the first time
    save_colorbar = (i==0)
    # make the graph that shows what scale the max was pulled from
    scale_label_figure(labs, scales, crop=crop,
                       savefilename=outname('3_labeled'), image_only=True,
                       save_colorbar_separate=save_colorbar,
                       output_dir=OUTPUT_DIR)

    scale_label_figure(labs_LO, scales, crop=crop,
                       savefilename=outname('4_labeled'), image_only=True,
                       save_colorbar_separate=False, output_dir=OUTPUT_DIR)
    # save the maximum frangi output
    plt.imsave(outname('2_fmax'), F.max(axis=-1)[crop],
               vmin=0, vmax=1.0, cmap=plt.cm.nipy_spectral)
    plt.close('all')  # something's leaking :(

# json file with mccs and other runtime info
timestring = datetime.datetime.now()
timestring = timestring.strftime("%y%m%d_%H%M")


mccfile = os.path.join(OUTPUT_DIR, f"runlog_{timestring}.json")

runlog = {
    'time': timestring,
    'dark_bg': DARK_BG,
    'dilate_per_scale': DILATE_PER_SCALE,
    'log_range': log_range,
    'n_scales': n_scales,
    'scales': list(scales),
    'alphas': list(alphas),
    'betas': None,
    'use_npz_files': False,
    'remove_glare': REMOVE_GLARE,
    'files': list(placentas),
    'MCCS': mccs,
    'PNC': pncs
}

with open(mccfile, 'w') as f:
    json.dump(runlog, f, indent=True)
```

listings/frangi.py

```
import numpy as np
import numpy.ma
from hfft import fft_hessian
from diffgeo import principal_curvatures
from plate_morphology import dilate_boundary


def frangi_from_image(img, sigma, beta=0.5, gamma=None, dark_bg=True,
                      dilation_radius=None, signed_frangi=False,
                      return_debug_info=False):
    """
    Perform a frangi filtering on img
    if None, gamma returns half of Frobenius norm on the image
    if dilation radius is specified, that amount is dilated from the
    boundary of the image (mask must be specified)

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

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


    BIGGER TODO:

        THIS OVERLAPS WITH pcsvn.make_multiscale
        USE THIS THERE

    """
    # principal_directions() calculates the frangi filter with
    # standard convolution and takes forever. FIX THIS!
    hesh = fft_hessian(img, sigma)  # the triple (Hxx,Hxy,Hyy)

    k1, k2 = principal_curvatures(img, sigma, H=hesh)

    if dilation_radius is not None:

        # pass None to just get the mask back
        collar = dilate_boundary(None, radius=dilation_radius,
                                 mask=img.mask)

        # get rid of "bad" K values before you calculate gamma
        k1[collar] = 0
        k2[collar] = 0

    # set default gamma value if not supplies
    if gamma is None:
        gamma = .5 * max_hessian_norm(hesh)
        if np.isclose(gamma, 0):
            print("WARNING: gamma is close to 0. should skip this layer.")

    targets = get_frangi_targets(k1, k2, beta=beta, gamma=gamma,
                                 dark_bg=dark_bg, signed=signed_frangi)

    if not return_debug_info:
        return targets
    else:
        return targets, (R, S, gamma)

def get_frangi_targets(K1, K2, beta=0.5, gamma=None, dark_bg=True,
```

```
                    signed=False):
    """Calculate the Frangi vesselness measure from eigenvalues.

    Parameters
    ----------
        K1, K2 : ndarray (each)
            each is an ndarray of eigenvalues (approximated principal
            curvatures) for some image.
        beta: float
            the anisotropy parameter (default is 0.5)
        gamma: float or None
            the structureness parmeter. if gamma is None (default), use
            half of L2 norm of hessian (calculated from K2).  if you want to
            use half of frobenius norm, calculate it outside here.
        dark_bg: boolean or None
            if True, then frangi will select only for bright curvilinear
            features; if False, then Frangi will select only for dark
            curvilinear structures. if None instead of a bool, then curvilinear
            structures of either type will be reported.
        signed: boolean
            if signed is True, the result will be the same as if dark_bg is set
            to None, except that the sign will change to match the desired
            features. See example below.

    Returns
    -------
        F: ndarray, same shape as K1
            the Frangi vesselness measure.
    Examples
    --------
    >>>f1 = get_frangi_targets(K1,K2, dark_bg=True, signed=True)
    >>>f2 = get_frangi_targets(K1,K2, dark_bg=False, signed=True)
    >>>f1 == -f2
    True

    """
    R = anisotropy(K1,K2)
    S = structureness(K1,K2)

    if gamma is None:
        # half of max hessian norm (using L2 norm)
        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...")
            print("sigma={:.3f}, gamma={}".format(sigma,gamma))
            print("returning an empty array")
            return np.zeros_like(img)

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

    # now just filter/ change sign as appropriate.
    if not signed:
        # calculate the regular frangi filter
        if dark_bg is None:
            #keep F the way it is
            pass
        elif dark_bg:
            # zero responses from positive curvatures
            F = (K2 < 0)*F
        else:
            # zero responses from negative curvatures
            F = (K2 > 0)*F
```

```
    else:
        if dark_bg is None:
            # output is already signed
            pass
        elif dark_bg:
            # positive curvature spots will be made negative
            F[K2 > 0] = -1 * F[K2 > 0]
        else:
            # negative curvature spots will be made positive
            F[K2 < 0] = -1 * F[K2 < 0]

    # reapply the mask if the inputs came with one
    if numpy.ma.is_masked(K1):
        F = numpy.ma.masked_array(F, mask=K1.mask)

    return F

def max_hessian_norm(hesh):
    """Calculate max norm of Hessian.
    calculates the maximal value (over all pixels of the image) of the
    Frobenius norm of the Hessian.

    Parameters
    ----------
    hesh: a tuple of ndarrays
        The tuple hxx,hxy,hyy which are all the same shape. The hessian at
        the point (m,n) is then [[hxx[m,n], hxy[m,n]],
                                 [hxy[m,n], hyy[m,n]]]

    Returns
    -------

    """
    hxx, hxy, hyy = hesh

    # frob norm is just sqrt(trace(AA^T)) which is easy for a 2x2
    max_norm = np.sqrt((hxx**2 + 2*hxy*2 + hyy**2).max())

    return max_norm

def anisotropy(K1,K2):
    """
    according to Frangi (1998) this is technically A**2
    """

    return (K1/K2) **2

def structureness(K1,K2):
    """
    according to Frangi (1998) this is technically S**2
    """
    return K1**2 + K2**2
```

listings/hfft_accuracy.py

```
"""

here you want to show  the accuracy of hfft.py

BOILERPLATE

show that gaussian blur of hfft is accurate, except potentially around the
boundary proportional to sigma.
```

```
or if they're off by a scaling factor, show that the derivates
(taken the same way) are proportional.

pseudocode

A = gaussian_blur(image, sigma, method='convential')
B = gaussian_blue(image, sigma, method='fourier')

zero_order_accurate = isclose(A, B, tol)

J_A= get_jacobian(A)
J_B = get_jacobian(B)

first_order_accurate = isclose(J_A, J_B, tol)

A_eroded = zero_around_plate(A, sigma)
B_eroded = zero_around_plate(B, sigma)

J_A_eroded = zero_around_plate(A, sigma)
J_B_eroded = zero_around_plate(B, sigma)

zero_order_accurate_no_boundary = isclose(A_eroded, B_eroded, tol)
first_order_accurate = isclose(J_A_eroded, J_B_eroded, tol)

"""

from placenta import get_named_placenta

from hfft import fft_hessian, fft_gaussian
from scipy.ndimage import gaussian_filter
import matplotlib.pyplot as plt
from placenta import mimshow

from scoring import mean_squared_error
import numpy as np
from scipy.ndimage import laplace
import numpy.ma as ma

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

from diffgeo import principal_curvatures
from frangi import structureness, anisotropy, get_frangi_targets

def erode_plate(img, sigma, mask=None):
    """
    Apply an eroded mask to an image
    assume (if helpful) that the boundary of the placenta is a connected loop
    that is, there is a single inside and outside of the shape, and that
    the placenta is more or less convex

    alternatively, if img is None, simply erode the mask
    this function should probably be renamed "dilate mask"
    and erode plate should be one that just acts on masked inputs
    """

    if mask is None:
        mask = img.mask

    # get a boolean array that is 1 along the border of the mask, zero elsewhere
    # default mode is 'thick' which is fine
    bounds = find_boundaries(mask)

```

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

    new_mask = np.logical_or(mask, dilated_border)

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


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

imgfile = 'barium1.png'
maskfile = 'barium1.mask.png'

img_raw = get_named_placenta(imgfile, maskfile=maskfile)

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

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

sigma = 5

print('applying standard gauss blur')
# THIS USES THE SAME DTYPE AS THE INPUT SO DEAR LORD MAKE SURE IT'S A FLOAT
A = gaussian_filter(img.astype('f'), sigma, mode='constant') #zero padding
print('applying fft gauss blur')
B = fft_gaussian(img, sigma)
B_unnormalized = B.copy()
B = B / (2*(sigma**2)*np.pi)

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

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


print('calculating second derivatives')

# you can verify np.isclose(Axy,Ayx) && np.isclose(Bxy,Byx) -> True
Axx, Axy = np.gradient(Ax)
Ayx, Ayy = np.gradient(Ay)

Bxx, Bxy = np.gradient(Bx)
Byx, Byy = np.gradient(By)



print('calculating eigenvalues of hessian')
ak1, ak2 = principal_curvatures(A, sigma=sigma, H=(Axx,Axy,Ayy))
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)
148 #R1[new_mask] = 0
149 #R2[new_mask] = 0
150 #S1[new_mask] = 0
151 #S2[new_mask] = 0
152
153 FA = get_frangi_targets(ak1,ak2)
154 FB = get_frangi_targets(bk1,bk2)
155
156 FA[new_mask] = 0
157 FB[new_mask] = 0
158
159 # even without scaling (which occurs below) the second derivates should be
160 # close. normalize matrices using frobenius norm of the hessian?
161 # note: A & B are off but have the same shape
162
163
164 # rescale to [0,255] (actually should keep as 0,1? )
165 #A_unscaled = A.copy()
166 #B_unscaled = B.copy()
167
168 #Ascaled = (A-A.min())/(A.max()-A.min())
169 #Bscaled = (B-B.min())/(B.max()-B.min())
170
171 # the following shows a random vertical slice of A & B (when scaled)
172 # the results are even more fitting when you scale B to coincide with A's max
173 # (which obviously isn't feasible in practice)
174
175 # FIXEDISH AFTER SCALING!
176
177 plt.plot(np.arange(A.shape[1]),A[A.shape[0]//2,:],
178          label='scipy.ndimage,gaussian_filter')
179 plt.plot(np.arange(B.shape[1]), B[B.shape[0]//2,:],
180          label='fft_gaussian')
181 plt.legend()
182
183 #MSE = ((A-B)**2).sum() / A.size
184 MSE = mean_squared_error(A,B)
```

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)
24
25 fft_results = list()
26 std_results = list()
27
28 for sigma in sigmas:
29     # test statements to compare (fft-based gaussian vs convolution-based)
30     fft_test_statement = 'fft_gaussian(img,{})'.format(sigma)
31     std_test_statement = 'gaussian_filter(img,{})'.format(sigma)
32     # run each statement 1 times (with 2 runs in each trial)
33     # returns/appends the average of 3 runs
34     fft_results.append(timeit.timeit(fft_test_statement,
35                                      number=1, globals=globals()))
36     std_results.append(timeit.timeit(std_test_statement,
37                                      number=1, globals=globals()))
38
39     # now actually evaluate both to compare
40     f = eval(fft_test_statement)
41     s = eval(std_test_statement)
42
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
50
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/hfft.py

```
1 #!/usr/bin/env python3
2
3 import numpy as np
4 from scipy import signal
5 import scipy.fftpack as fftpack
6 from scipy.special import iv
7
8 """
9 hfft.py is the implementation of calculating the hessian of a real
10
11 image based in frequency space (rather than direct convolution with a gaussian
12 as is standard in scipy, for example).
13
```

```
TODO: PROVIDE MAIN USAGE NOTES
"""

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

    NOTE:
        this function is/should be? for illustrative purposes only--
        we can actually build this much faster using the builtin
        scipy.signal.gaussian rather than a roll-your-own

    build a shape=(M,N) sized gaussian kernel in frequency space
    with size

    (due to the convolution theorem for fourier transforms, the function
    created here may simply be *multiplied" against the signal.

    """

    M,  N = shape
    fgauss = np.fromfunction(lambda   ,  : ((  +M+1)/2)**2 + ((  +N+1)/2)**2, shape=shape)

    # is this used?
    coeff = (1 / (2*np.pi *   **2))

    return np.exp(-fgauss / (2*  **2))

def blur(img, sigma):
    """
    DEPRECATED
    a roll-your-own FFT-implemented gaussian blur.
    fft_gaussian below is preferred (it is more efficient)
    """

    I = fftpack.fft2(img) # get 2D transform of the image

    # do whatever

    I *= gauss_freq(I.shape, sigma)


    return fftpack.ifft2(I).real

def fft_gaussian(img,sigma,A=None):

    """
    https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.fftconvolve.html

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

    along with the comment:
    "Gaussian blur implemented using FFT convolution. Notice the dark borders
    around the image, due to the zero-padding beyond its boundaries. The
    convolve2d function allows for other types of image boundaries, but is far
    slower"

    (i.e. doesn't use FFT).

    note that here, you actually take the FFT of a gaussian (rather than
    build it in frequency space). there are ~6 ways to do this.
    """
    #create a 2D gaussian kernel to take the FFT of

```

```
78      # scale factor!
79      A = 1 / (2*np.pi*sigma**2)
80      kernel = np.outer(A*signal.gaussian(img.shape[0], sigma),
81                        A*signal.gaussian(img.shape[1],sigma))
82
83      return signal.fftconvolve(img, kernel, mode='same')
84
85  def discrete_gaussian_kernel(n_samples, t):
86      """
87      t is the scale, n_samples is the number of samples to compute
88      will return a window centered a zero
89      i.e. arange(-n_samples//2, n_samples//2+1)
90
91      this is UNnormalized, oops
92      """
93      dom = np.arange(-n_samples//2, n_samples // 2 + 1)
94      #there should be a scaling parameter alpha but whatever
95      return np.exp(-t) * iv(dom,t)
96
97  def fft_dgk(img,sigma,order=0,A=None):
98      """
99      A is scaling factor.
100     This is the discrete gaussian kernel which is supposedly less crappy
101     than using a sampled gaussian.
102     """
103     m,n = img.shape
104     # i don't know if this will suck if there are odd dimensions
105     kernel = np.outer(discrete_gaussian_kernel(m,sigma),
106                       discrete_gaussian_kernel(n,sigma))
107
108     return signal.fftconvolve(img, kernel, mode='same')
109
110 def fft_fdgk(img,sigma):
111     """
112     convolve with discrete gaussian kernel in freq. space
113     """
114     # this would be a lot better since you wouldn't have to deal
115     # with an arbitrary cutoff of size of the discrete kernel
116     # since the freq. space version is just
117     # exp{\alpha*t (cos\theta - 1)}
118     # see formula 22 of lindeberg discrete paper
119
120     pass
121
122 def fft_hessian(image, sigma=1., kernel=None):
123     """
124     a reworking of skimage.feature.hessian_matrix that uses
125     e FFT to compute gaussian, which results in a considerable speedup
126
127     INPUT:
128         image - a 2D image (which type?)
129         sigma - coefficient for gaussian blur
130
131     OUTPUT:
132         (Lxx, Lxy, Lyy) - a triple containing three arrays
133             each of size image.shape containing the xx, xy, yy derivatives
134             respectively at each pixel. That is, for the pixel value given
135             by image[j][k] has a calculated 2x2 hessian of
136             [ [Lxx[j][k], Lxy[j][k]],
137               [Lxy[j][k], Lyy[j][k]] ]
138     """
139     if kernel == 'discrete':
140         #print('using discrete kernel!')
141         gaussian_filtered = fft_dgk(image, sigma=sigma)
```

```
142     else:
143         #print('using sampled gauss kernel')
144         gaussian_filtered = fft_gaussian(image, sigma=sigma)
145 
146     Lx, Ly = np.gradient(gaussian_filtered)
147 
148     Lxx, Lxy = np.gradient(Lx)
149     Lxy, Lyy = np.gradient(Ly)
150 
151     return (Lxx, Lxy, Lyy)
152 
153 def fft_gradient(image, sigma=1.):
154     """ returns gradient norm """
155 
156     gaussian_filtered = fft_gaussian(image, sigma=sigma)
157 
158     Lx, Ly = np.gradient(gaussian_filtered)
159 
160     return np.sqrt(Lx**2 + Ly**2)
161 def _old_test():
162     """
163     old main function for testing.
164 
165     This simply tests fft_gaussian on a test image, exemplifying the speedup
166     compared to a traditional gaussian.
167     """
168     import matplotlib.pyplot as plt
169 
170     from skimage.data import camera
171 
172     img = camera() / 255.
173 
174     sample_sigmas = (.2, 2, 10, 30)
175 
176     outputs = (fft_gaussian(img, sample_sigmas[0]),
177                fft_gaussian(img, sample_sigmas[1]),
178                fft_gaussian(img, sample_sigmas[2]),
179                fft_gaussian(img, sample_sigmas[3]),
180                )
181 
182 
183     fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(10, 10))
184 
185     axes[0, 0].imshow(outputs[0], cmap='gray')
186     axes[0, 0].set_title('fft_gaussian   ={}'.format(sample_sigmas[0]))
187     axes[0, 0].axis('off')
188 
189     axes[0, 1].imshow(outputs[1], cmap='gray')
190     axes[0, 1].set_title('fft_gaussian   ={}'.format(sample_sigmas[1]))
191     axes[0, 1].axis('off')
192 
193     axes[1, 0].imshow(outputs[2], cmap='gray')
194     axes[1, 0].set_title('fft_gaussian   ={}'.format(sample_sigmas[2]))
195     axes[1, 0].axis('off')
196 
197     axes[1, 1].imshow(outputs[3], cmap='gray')
198     axes[1, 1].set_title('fft_gaussian   ={}'.format(sample_sigmas[3]))
199     axes[1, 1].axis('off')
200 
201     plt.tight_layout()
202     plt.show()
203 
204 if __name__ == "__main__":
205 
```

```
    pass
```

listings/merging.py

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

import numpy as np
import numpy.ma as ma


def nz_percentile(A, q, axis=None, interpolation='linear'):
    """calculate np.percentile(...,q) on an array's nonzero elements only

    Parameters
    ----------
    A : ndarray
        matrix from which percentiles will be calculated. Percentiles
        are calculated on an elementwise basis, so the shape is not important
    q : a float
        Percentile to compute, between 0 and 100.0 (inclusive).

    (other arguments): see numpy.percentile docstring
    ...

    Returns
    -------
    out: float

    """

    if ma.is_masked(A):
        A = A.filled(0)

    return np.percentile(A[A > 0], q, axis=axis, interpolation=interpolation)


def apply_threshold(targets, alphas, return_labels=True):
    """Threshold targets at each scale, then return max target over all scales.

    A unique alpha can be given for each scale (see below). Return a 2D boolean
    array, and optionally another array representing what at what scale the max
    filter response occurred.

    Parameters
    ----------
    targets : ndarray
        a 3D array, where targets[:,:,k] is the result of the Frangi filter
        at the kth scale.
    alphas : float or array_like
        a list / 1d array of length targets.shape[-1]. each alphas[k] is a
        float which thresholds the Frangi response at the kth scale. Due to
        broadcasting, this can also be a single float, which will be applied
        to each scale.
    return_labels : bool, optional
        If True, return another ndarray representing the scale (see Notes
        below). Default is True.

    Returns
    -------
    out : ndarray, dtype=bool
        if return labels is true, this will return both the final
        threshold and the labels as two separate matrices. This is
        a convenience, since you could easily find labels with
```

```
    labels : ndarray, optional, dtype=uint8
        The scale at which the largest filter response was found after
        thresholding. Element is 0 if no scale passed the threshold,
        otherwise an int between 1 and targets.shape[-1] See Notes below.

    Notes / Examples
    ----------------
    Despite the name, this does *NOT* return the thresholded targets itself,
    but instead the maximum value after thresholding. If you wanted the
    thresholded filter responses alone, you should simply run

    >>>(targets > alphas)*targets

    The optional output `labels` is a 2D matrix indicating where the max filter
    response occured. For example, if the label is K, the max filter response
    will occur at targets[:,:,K-1].  In other words,

    >>>passed, labels = apply_threshold(targets,alphas)
    >>>targets.max(axis=-1) == targets[:,:,labels -1 ]
    True

    It should be noted that returning labels is really just for convenience
    only; you could construct it as shown in the following example:

    >>>manual_labels = (targets.argmax(axis=-1) + 1)*np.invert(passed)
    >>>labels == manual_labels
    True

    Similarly, the standard boolean output could just as easily be obtained.
    >>>passed == (labels != 0)
    True
    """

    # threshold as an array (even if it's a single element) to broadcast
    alphas = np.array(alphas)

    # if input's just a MxN matrix, expand it trivially so it works below
    if targets.ndim == 2:
        targets = np.expand_dims(targets, 2)

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

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

    if not return_labels:
        return passed  # we're done already

    wheres = targets.argmax(axis=-1)  # get label of where maximum occurs
    wheres += 1  # increment to reserve 0 label for no match

    # then remove anything that didn't pass the threshold
    wheres[np.invert(passed)] = 0

    assert np.all(passed == (wheres > 0))

    return passed, wheres
```

listings/pcsvn.py

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

```
2
3  from placenta import get_named_placenta
4  from hfft import fft_hessian
5  from diffgeo import principal_curvatures, principal_directions
6  from frangi import get_frangi_targets
7  from skimage.util import img_as_float
8  import numpy as np
9  from preprocessing import (inpaint_glare, inpaint_with_boundary_median,
10                             inpaint_hybrid)
11
12 from plate_morphology import dilate_boundary
13
14 import matplotlib.pyplot as plt
15 import matplotlib as mpl
16 import numpy.ma as ma
17
18 import os.path
19 import json
20 import datetime
21
22
23 def make_multiscale(img, scales, betas, gammas,
24                     find_principal_directions=False, dilate_per_scale=True,
25                     signed_frangi=False, dark_bg=True, kernel=None,
26                     VERBOSE=True):
27     """Returns an ordered list of dictionaries for each scale of Frangi info.
28
29     Each element in the output contains the following info:
30         {'sigma': sigma,
31          'beta': beta,
32          'gamma': gamma,
33          'H': hesh,
34          'F': targets,
35          'k1': k1,
36          'k2': k2,
37          't1': t1, # if find_principal_directions
38          't2': t2 # if find_principal_directions
39          }
40
41     """
42
43     # store results of each scale (create as empty list)
44     multiscale = list()
45
46     img = ma.masked_array(img_as_float(img), mask=img.mask)
47
48     for i, sigma, beta, gamma in zip(range(len(scales)), scales,
49                                      betas, gammas):
50         if dilate_per_scale:
51             if sigma < 2.5:
52                 radius = 10
53             elif sigma > 20:
54                 radius = int(sigma*2)
55             else:
56                 radius = int(sigma*4)  # a little aggressive
57         else:
58             radius = None
59
60         if VERBOSE:
61             print('  ={}'.format(sigma))
62
63         # get hessian components at each pixel as a triplet (Lxx, Lxy, Lyy)
64         hesh = fft_hessian(img, sigma, kernel=kernel)
```

```
65
66          if VERBOSE:
67              print('finding principal curvatures')
68
69          # calculate principal curvatures with |k1| <= |k2|
70          k1, k2 = principal_curvatures(img, sigma=sigma, H=hesh)
71
72          # area of influence to zero out
73          if dilate_per_scale:
74              collar = dilate_boundary(None, radius=radius, mask=img.mask)
75
76              k1[collar] = 0
77              k2[collar] = 0
78
79          # set anisotropy parameter if not specified
80          if gamma is None:
81              # Frangi suggested 'half the max Hessian norm' as an empirical
82              # half the max spectral radius is easier to calculate so do that
83              # shouldn't be affected by mask data but should make sure the
84              # mask is *well* far away from perimeter
85              # we actually calculate half of max hessian norm
86              # using frob norm = sqrt(trace(AA^T))
87              hxx, hxy, hyy = hesh
88              hessian_norm = np.sqrt((hxx**2 + 2*hxy**2 + hyy**2))
89
90              # make sure the max doesn't occur on a boundary
91              dilation_radius = int(max(np.ceil(sigma), 10))
92              collar = dilate_boundary(None, radius=dilation_radius,
93                                       mask=img.mask)
94              hessian_norm[collar] = 0
95              max_hessian_norm = hessian_norm.max()
96              gamma = .5*max_hessian_norm
97
98              if VERBOSE:
99                  # compare to other method of calculating gamma
100                 gamma_alt = .5 * np.abs(k2).max()
101                 print(f"half of k2 max is {gamma_alt}")
102
103         if VERBOSE:
104             print(f"gamma (half of max hessian (frob) norm is {gamma}")
105             print(f'finding Frangi targets with   ={beta} and   ={gamma:.2}')
106
107         # calculate frangi targets at this scale
108         targets = get_frangi_targets(k1, k2, beta=beta, gamma=gamma,
109                                      dark_bg=dark_bg, signed=signed_frangi)
110
111
112         # store results as a dictionary
113         this_scale = {'sigma': sigma,
114                       'beta': beta,
115                       'gamma': gamma,
116                       'H': hesh,
117                       'F': targets,
118                       'k1': k1,
119                       'k2': k2,
120                       'border_radius': radius
121                       }
122
123         if find_principal_directions:
124             # principal directions should only be computed for critical regions
125             # ignore anything less than a std deviation over the mean
126             # this mask is where PD's will *NOT* be calculated
127             cutoff = targets.mean() + targets.std()
128             pd_mask = np.bitwise_or(targets < cutoff, img.mask).filled(1)
```

```
            percent_calculated = (pd_mask.size - pd_mask.sum()) / pd_mask.size

            if VERBOSE:
                print(f"finding PD's for {percent_calculated:.2%} of image"
                      f"anything above vesselness score {cutoff:.6f}"
                      )
            t1, t2 = principal_directions(img, sigma=sigma, H=hesh,
                                          mask=pd_mask)

            # add them to this scale's output
            this_scale['t1'] = t1
            this_scale['t2'] = t2
        else:
            if VERBOSE:
                print('skipping principal direction calculation')

        # store results as a list of dictionaries
        multiscale.append(this_scale)

    return multiscale


def extract_pcsvn(filename, scales, alphas=None, betas=None, gammas=None,
                  DARK_BG=True, dilate_per_scale=True, verbose=True,
                  generate_json=True, output_dir=None, kernel=None,
                  signed_frangi=False, remove_glare=False):
    """Run PCSVN extraction on the sample given in the file.

    Despite the name, this simply returns the Frangi filter responses at
    each provided scale without explicitly making any decisions about what
    is or is not part of the PCSVN.

    TODO:Finish docstring!
    """

    raw_img = get_named_placenta(filename, maskfile=None)

    # Multiscale & Frangi Parameters######################

    # set default alphas and betas if undeclared
    if alphas is None:
        alphas = [.15 for s in scales]  # threshold constant
    if betas is None:
        betas = [0.5 for s in scales]  # anisotropy constant

    # declare None here to calculate half of hessian's norm
    if gammas is None:
        gammas = [None for s in scales]  # structureness parameter

    # Preprocessing###############
    if remove_glare:
        if verbose:
            print('removing glare from sample')
        # img = inpaint_glare(raw_img)
        # img = inpaint_with_boundary_median(raw_img)
        img = inpaint_hybrid(raw_img)
    else:
        img = raw_img.copy()  # in case we alter the mask or something

    # Multiscale Frangi Filter###############################

    # output is a dictionary of relevant info at each scale
    multiscale = make_multiscale(img, scales, betas, gammas,
                                 find_principal_directions=False,
```

```
                                  dilate_per_scale=dilate_per_scale,
                                  kernel=kernel,
                                  signed_frangi=signed_frangi,
                                 dark_bg=DARK_BG,
                                  VERBOSE=verbose)

    # extract these for logging
    gammas = [scale['gamma'] for scale in multiscale]
    border_radii = [scale['border_radius'] for scale in multiscale]

    ###Process Multiscale Targets##############################

    # ignore targets too close to edge of plate
    # wait are we doing this twice?
    if dilate_per_scale:
        if verbose:
            print('trimming collars of plates (per scale)')

        for i in range(len(multiscale)):
            f = multiscale[i]['F']
            # twice the buffer (be conservative!)
            radius = int(multiscale[i]['sigma']*2)
            if verbose:
                print('dilating plate for radius={}'.format(radius))
            f = dilate_boundary(f, radius=radius, mask=img.mask)
            # get rid of mask
            multiscale[i]['F'] = f.filled(0)
    else:
        for i in range(len(multiscale)):
            # get rid of mask
            multiscale[i]['F'] = multiscale[i]['F'].filled(0)

    ###Make Composite#########################################

    # get a M x N x n_scales array of Frangi targets at each level
    F_all = np.dstack([scale['F'] for scale in multiscale])

    if generate_json:

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

        logdata = {'time': timestring,
                'filename': filename,
                'alphas': list(alphas),
                'betas': list(betas),
                'gammas': gammas,
                'sigmas': list(scales),
                }
        if dilate_per_scale:
            logdata['border_radii'] = border_radii

        if output_dir is None:
            output_dir = 'output'

        base = os.path.basename(filename)
        *base, suffix = base.split('.')
        dumpfile = os.path.join(output_dir,
                                ''.join(base) + '_' + str(timestring)
                                + '.json')

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

```
257     # this function used to returns scales and alphas too but now doesn't,
258     return F_all, img
259
260
261 def get_outname_lambda(filename, output_dir=None, timestring=None):
262     """
263     return a lambda function which can build output filenames
264     """
265
266     if output_dir is None:
267         output_dir = 'output'
268
269     base = os.path.basename(filename)
270     *base, suffix = base.split('.')
271
272     if timestring is None:
273         time_of_run = datetime.datetime.now()
274         timestring = time_of_run.strftime("%y%m%d_%H%M")
275
276     outputstub = ''.join(base) +'_' + timestring +  '_{}.'+ suffix
277     return lambda s: os.path.join(output_dir, outputstub.format(s))
278
279
280 def _build_scale_colormap(N_scales, base_colormap, basecolor=(0,0,0,1)):
281     """
282     returns a mpl.colors.ListedColormap with N samples,
283     based on the colormap named "default_colormap" (a string)
284
285     the N colors are given by the default colormap, and
286     basecolor (default black) is added to map to 0.
287     (you could change this, for example, to (1,1,1,1) for white)
288
289     reversed colormaps often work better if the basecolor is black
290     you should make sure there's good contrast between the basecolor
291     and the first color in the colormap
292     """
293
294     map_range = np.linspace(0, 1, num=N_scales)
295
296     colormap = plt.get_cmap(base_colormap)
297
298     colorlist = colormap(map_range)
299
300     # add basecolor as the first entry
301     colorlist = np.vstack((basecolor, colorlist))
302
303     return mpl.colors.ListedColormap(colorlist)
304
305 def scale_label_figure(wheres, scales, savefilename=None,
306                        crop=None, show_only=False, image_only=False,
307                        save_colorbar_separate=False, savecolorbarfile=None,
308                        output_dir=None):
309     """
310     crop is a slice object.
311     if show_only, then just plt.show (interactive).
312     if image_only, then this will *not* be printed with the colorbar
313
314     if save_colormap_separate, then the colormap will be saved as a separate
315     file
316     """
317     if crop is not None:
318         wheres = wheres[crop]
319
320     fig, ax = plt.subplots() # not sure about figsize
```

```
    N = len(scales) # number of scales / labels

    tabemap = _build_scale_colormap(N, 'viridis_r')

    if image_only:
        plt.imsave(savefilename, wheres, cmap=tabemap, vmin=0, vmax=N)
        plt.close()
    else:
        imgplot = ax.imshow(wheres, cmap=tabemap, vmin=0, vmax=N)
        # discrete colorbar
        cbar = plt.colorbar(imgplot)

        # this is apparently hackish, beats me
        tick_locs = (np.arange(N+1) + 0.5)*(N-1)/N

        cbar.set_ticks(tick_locs)
        # label each tick with the sigma value
        scalelabels = [r"$\sigma = {:.2f}$".format(s) for s in scales]
        scalelabels.insert(0, "(no match)")
        # label with their sigma value
        cbar.set_ticklabels(scalelabels)
        #ax.set_title(r"Scale ($\sigma$) of maximum vesselness ")
        plt.tight_layout()

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

        plt.close()

    if save_colorbar_separate:
        if savecolorbarfile is None:
            savecolorbarfile = os.path.join(output_dir, "scale_colorbar.png")
        fig = plt.figure(figsize=(1, 8))
        ax1 = fig.add_axes([0.05, 0.05, 0.15, 0.9])
        tick_locs = (np.arange(N+1) + 0.5)*(N-1)/N
        scalelabels = [r"$\sigma = {:.2f}$".format(s) for s in scales]
        scalelabels.insert(0, "n/a")
        cbar = mpl.colorbar.ColorbarBase(ax1, cmap=tabemap,
                                         norm=mpl.colors.Normalize(vmin=0,
                                                                   vmax=N),
                                         orientation='vertical',
                                         ticks=tick_locs)
        cbar.set_ticklabels(scalelabels)
        plt.savefig(savecolorbarfile, dpi=300)
```

listings/placenta.py

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

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

Again, there is no support for unregistered placental pictures.
A mask file must be provided.

There is currently no support for color images.
"""

```

```
import numpy as np
import numpy.ma as ma
from skimage import segmentation, morphology
import os.path
import os
import json
from scipy.ndimage import imread

def open_typefile(filename, filetype, sample_dir=None, mode=None):
    """
    filetype is either 'mask' or 'trace'
    mask -> 'L' mode
    trace -> 'RGB' mode
    use mode keyword to override this behavior (for example if you
    want a binary trace)

    typefiles that aren't the above will be treated as 'L'
    """
    # try to open what the mask *should* be named
    # this should be done less hackishly
    # for example, if filename is 'ncs.1029.jpg' then
    # this would set the maskfile as 'ncs.1029.mask.jpg'

    #if filetype not in ("mask", "trace"):
    #    raise NotImplementedError("Can only deal with mask or trace files.")

    # get the base of filename and build the type filename
    *base, suffix = filename.split('.')
    base = ''.join(base)
    typefile = '.'.join((base, filetype ,suffix))

    if sample_dir is None:
        sample_dir = 'samples'

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

    if mode is not None:
        if filetype == 'mask':
            mode = 'L'
        elif filetype in ('ctrace', 'veins', 'arteries'):
            mode = 'RGB'
        else:
            # handle this if you need to?
            mode = 'L'
    try:
        img = imread(typefile, mode=mode)

    except FileNotFoundError:
        print('Could not find file', typefile)
        raise

    return img


def open_tracefile(base_filename, as_binary=True,
                   sample_dir=None):

    """

    ###width parsing is no longer done here. instead, this function
    should handle the venous/arterial difference.

    this currently only serves to open the RGB traces as binary
    files instead of RGB, which is processed later
```

```
78
79      #TODO: expand this later to handle arterial traces and venous traces
80      INPUT:
81          base_filename: the name of the base file, not the tracefile itself
82          as_binary: if True
83      """
84
85      if as_binary:
86          mode = 'L'
87      else:
88          mode = 'RGB'
89
90      T = open_typefile(base_filename, 'trace', sample_dir=sample_dir, mode=mode)
91
92      if as_binary:
93
94          return np.invert(T != 0)
95
96      else:
97          return T
98
99
100
101
102
103 def get_named_placenta(filename, sample_dir=None, masked=True,
104                        maskfile=None):
105     """
106     This function is to be replaced by a more ingenious/natural
107     way of accessing a database of unregistered and/or registered
108     placental samples.
109
110     INPUT:
111         filename: name of file (including suffix?) but NOT directory
112         masked: return it masked.
113         maskfile: if supplied, this use the file will use a supplied 1-channel
114                 mask (where 1 represents an invalid/masked pixel, and 0
115                 represents a valid/unmasked pixel. the supplied image must be
116                 the same shape as the image. if not provided, the mask is
117                 calculated (unless masked=False)
118                 the file must be located within the sample directory
119
120                 If maskfile is 'None' then this function will look for
121                 a default maskname with the following  pattern:
122
123                     test.jpg -> test.mask.jpg
124                     ncs.1029.jpg  -> ncs.1029.mask.jpg
125
126         sample_directory: Relative path where sample (and mask file) is located.
127                 defaults to './samples'
128
129     if masked is true (default), this returns a masked array.
130
131     NOTE: A previous logical incongruity has been corrected. Masks should have
132     1 as the invalid/background/mask value (to mask), and 0 as the
133     valid/plate/foreground value (to not mask)
134     """
135     if sample_dir is None:
136         sample_dir = 'samples'
137
138     full_filename = os.path.join(sample_dir, filename)
139
140     raw_img = imread(full_filename, mode='L')
141
```

```
    if maskfile is None:
        # try to open what the mask *should* be named
        # this should be done less hackishly
        # for example, if filename is 'ncs.1029.jpg' then
        # this would set the maskfile as 'ncs.1029.mask.jpg'
        *base, suffix = filename.split('.')
        test_maskfile = ''.join(base) + '.mask.' + suffix
        test_maskfile = os.path.join(sample_dir, test_maskfile)
        try:
            mask = imread(test_maskfile, mode='L')
        except FileNotFoundError:
            print('Could not find maskfile', test_maskfile)
            print('Please supply a maskfile. Autogeneration of mask',
                  'files is slow and buggy and therefore not supported.')
            raise
            #return mask_background(raw_img)
    else:
        # set maskfile name relative to path
        maskfile = os.path.join(sample_dir, maskfile)
        mask = imread(maskfile, mode='L')

    return ma.masked_array(raw_img, mask=mask)


def list_by_quality(quality=0, N=None, json_file=None,
                            return_empty=False):
    """
    returns a list of filenames that are of quality ``quality''

    quality is either "good" or 0
                      "OK" or 1
                      "fair" or 2
                      "bad" or 3

    N is the number of placentas to return (will return # of placentas
    of that quality or N, whichever is smaller)

    if json_name is not None just use that filename directly

    if return_empty then silenty failing is OK
    """
    if quality == 0:
        quality = 'good'
    elif quality == 1:
        quality = 'okay'
    elif quality == 2:
        quality = 'fair'
    elif quality == 3:
        quality = 'bad'
    else:
        try:
            quality = quality.lower()
        except AttributeError:
            if return_empty:
                return list()
            else:
                print(f'unknown quality {quality}')
                raise

    # json file
    if json_file is None:
        json_file = f"{quality}-mccs.json"

    # else the quality is irrelevant and hopefully the jsonfile
```

```
206     # was provided
207     try:
208         with open(json_file, 'r') as f:
209             D = json.load(f)
210     except FileNotFoundError:
211         if return_empty:
212             return list()
213         else:
214             print('cannot find', json_file)
215             raise
216 
217     placentas = [f'{d}.png' for d in D.keys()]
218 
219     return placentas
220 
221 def check_filetype(filename, assert_png=True, assert_standard=False):
222     """
223     'T-BN8333878.raw.png' returns 'raw'
224     'T-BN8333878.mask.png' returns 'mask'
225     'T-BN8333878.png' returns 'base'
226 
227     if assert_png is True, then raise assertion error if the file
228     is not of type png
229 
230     if assert_standard, then assert the filetype is
231     mask, base, trace, or raw.
232 
233     etc.
234     """
235     basename, ext = os.path.splitext(filename)
236 
237     if ext != '.png':
238         if assert_png:
239             assert ext == '.png'
240 
241     sample_name, typestub = os.path.splitext(basename)
242 
243     if typestub == '':
244         # it's just something like  'T-BN8333878.png'
245         return 'base'
246     elif typestub in ('.mask','.trace', '.raw', '.ctrace', '.arteries', '.veins', '.ucip'):
247         # return 'mask' or 'trace' or 'raw'
248         return typestub.strip('.')
249     else:
250         print('unknown filetype:', typestub)
251         print('is it a weird filename?')
252 
253         print('warning: lookup failed, unknown filetype:' + typestub)
254 
255         return typestub
256 
257 def list_placentas(label=None, sample_dir=None):
258     """
259     label is the specifier, basically just ''.startswith()
260 
261     only real use is to find all the T-BN* files
262 
263     this is hackish, if you ever decide to use a file other than
264     png then this needs to change
265     """
266 
267     if sample_dir is None:
268         sample_dir = 'samples'
269 
```

```
    if label is None:
        label = '' # str.startswith('') is always True

    placentas = list()

    for f in os.listdir(sample_dir):

        if f.startswith(label):
            # oh man they gotta be png files
            if check_filetype(f) == 'base':
                placentas.append(f)

    return sorted(placentas)

def show_mask(img, interactive=True, mask_color=None):
    """
    show a masked grayscale image with a dark blue masked region

    custom version of imshow that shows grayscale images with the right colormap
    and, if they're masked arrays, sets makes the mask a dark blue)
    a better function might make the grayscale value dark blue
    (so there's no confusion)

    if interactive, this operates like "plt.imshow"
    if interactive==False, return the RGB matrix
    """

    from numpy.ma import is_masked
    from skimage.color import gray2rgb
    import matplotlib.pyplot as plt

    if mask_color is None:
        mask_color = (0,0,60)

    if not is_masked(img):
        if interactive:
            plt.imshow(img, cmap=plt.cm.gray)
        else:
            # return as an rgb image so output is uniform
            return gray2rgb(img)

    # otherwise, get an RGB array, black where the mask is
    mimg = gray2rgb(img.filled(0))
    # fill masked regions with the mask color
    mimg[img.mask, :] = mask_color

    if interactive:
        plt.imshow(mimg)
    else:
        return mimg

def _cropped_bounds(img, mask=None):

    if mask is not None:

        img = ma.masked_array(img, mask=mask)

    X,Y = (np.argwhere(np.invert(img.mask).any(axis=k)).squeeze() for k in (0,1))

    if X.size == 0:
        X = [None,None] # these will slice correctly
    if Y.size == 0:
        Y = [None,None]

```

```
334     return Y[0],Y[-1],X[0],X[-1]
335
336 def cropped_args(img, mask=None):
337     """
338     get a slice that would crop image
339     i.e. img[cropped_args(img)] would be a cropped view
340     """
341
342     x0, x1, y0,y1 = _cropped_bounds(img, mask=None)
343
344     return np.s_[x0:x1,y0:y1]
345
346 def cropped_view(img, mask=None):
347     """
348     removes entire masked rows and columns from the borders of a masked array.
349     will return a masked array of smaller size
350
351     don't ask me about data
352
353     the name sucks too
354     """
355
356     # find first and last row with content
357     x0, x1, y0,y1 = _cropped_bounds(img, mask=mask)
358
359     return img[x0:x1,y0:y1]
360
361 CYAN = [0,255,255]
362 YELLOW = [255,255,0]
363
364
365 def measure_ncs_markings(ucip_img=None, filename=None, verbose=True):
366     """
367     find location of ucip and resolution of image based on input
368     (similar to perimeter layer in original NCS data set
369
370     Parameters
371     ----------
372
373     ucip_img: an RGB ndarray or None
374         The perimeter layer of an NCS sample (colorations according to the
375         tracing protocol). if None, filename must be included. Default is None.
376     filename:
377         the filename of the SAMPLE (not the ucip image file itself)
378
379     Returns
380     -------
381     m : tuple of ints
382         the coordinates of (the center of) of the umbilical cord point
383         (depicted as a yellow dot) in the original image.
384     resolution: a float
385         measured distance between the two cyan dots
386     """
387
388     if ucip_img is None:
389         ucip_img = open_typefile(filename, 'ucip')
390
391     # just in case it's got an alpha channel, remove it
392     img = ucip_img[:,:,0:3]
393
394     # given the image img (make sure no alpha channel)
395     # find all cyan pixels (there are two boxes of 3 pixels each and we
396     # just want to extract the middle of each
397     if verbose:
```

```
        print('the image size is {}x{}'.format(img.shape[0], img.shape[1]))

    rulemarks = np.all(img == CYAN, axis=-1)

    # turn into two pixels (these should each be shape (18,))
    X,Y = np.where(rulemarks)

    assert X.shape == Y.shape

    # if they followed the protocol correctly...
    if X.size == 18:
        # get the two pixels at the center of each box
        A, B = (X[4], Y[4]) , (X[13], Y[13])
    else:
        # dots are a nonstandard size for some reason. this works too.
        thinned = morphology.thin(rulemarks)
        X, Y = np.where(thinned)
        assert(thinned.sum() == 2) # there should be just two pixels now.
        A, B = (X[0], Y[0]) , (X[1], Y[1])

    ruler_distance =  np.sqrt( (A[0] - B[0])**2 + (A[1] - B[1])**2 )
    if verbose:
        print(f'one cm equals {ruler_distance} pixels')

    # the umbillical cord insertion point (UCIP) is a yellow circle, radius 19
    ucipmarks = np.all(img == YELLOW, axis=-1)
    X,Y = np.where(ucipmarks)

    # find midpoint of the x & y cooridnates
    assert X.max() - X.min() == Y.max() - Y.min()
    radius = (X.max() - X.min()) // 2

    mid = (X.min() + radius, Y.min() + radius)

    if verbose:
        print('the middle of the UCIP location is', mid)
        print('the radius outward is', radius)
        print('the total measurable diameter is', radius*2 + 1)

    return mid, ruler_distance


def add_ucip_to_mask(m, radius=100, mask=None, size_like=None):
    """
    - m is a tuple (2x1) representing the (coordinate) midpoint of the UCIP
    - radius around which to dilate the UCIP is the dilation radius as it
    works in morphology--this is passed directly to skimage.morphology.disk.
    thus a circle centered at point m with diameter 2*radius + 1
    - if no mask is supplied, dilate the point in an array of zeros the shape
    of `size_like` (would be the same as passing mask=np.zeros_like(size_like))

    Note: this behaves much faster than binary dilation on the point
    """
    if mask is None:
        if size_like is not None:
            mask = np.zeros_like(size_like)
        else:
            raise ValueError("No mask info supplied!")

    # an empty mask (since we need to merge--we don't want to copy the
    # zeros of the dilated UCIP -- just the ones!)
    to_add = np.zeros_like(mask)

    # this is way faster than dilating the point in the matrix,
```

```
462     # just set this at the centered point
463
464     # doesn't check for out of bounds stuff. use at your own peril
465     D = morphology.disk(radius)
466     to_add[m[0]-radius:m[0]+radius+1 , m[1]-radius:m[1]+radius+1] = D
467
468     # merge with supplied mask
469     return np.logical_or(mask, to_add)
470
471
472 if __name__ == "__main__":
473
474     """test that this works on an easy image."""
475
476     from scipy.ndimage import imread
477     import matplotlib.pyplot as plt
478     test_filename = 'barium1.png'
479     #test_maskfile = 'barium1.mask.png'
480
481     img =  get_named_placenta(test_filename, maskfile=None)
482
483     print('showing the mask of', test_filename)
484     print('run plt.show() to see masked output')
485     show_mask(img)
```

listings/plate_morphology.py

```
1  #!/usr/bin/env python3
2
3  from skimage.morphology import disk, binary_erosion, binary_dilation
4  from skimage.morphology import convex_hull_image
5  from skimage.segmentation import find_boundaries
6
7  import numpy as np
8  import numpy.ma as ma
9
10 def dilate_boundary(img, radius=10, mask=None):
11     """
12     grows the mask by a specified radius of a masked 2D array
13     Manually remove (erode) the outside boundary of a plate.
14     The goal is remove any influence of the zeroed background
15     on reporting derivative information.
16
17     There is varying functionality here (maybe should be multiple functions
18     instead?)
19
20     If img is a masked array and mask=None, the mask will be dilated and a
21     masked array is outputted.
22
23     If img is any 2D array (masked or unmasked), if mask is specified, then
24     the mask will be dilated and the original image will be returned as a
25     masked array with a new mask.
26
27     If the img is None, then the specified mask will be dilated and returned
28     as a regular 2D array.
29
30     """
31
32     if mask is None:
33         # grab the mask from input image
34         # if img is None this will break too but not handled
35         try:
```

```
            mask = img.mask
        except AttributeError:
            raise('Need to supply mask information')

    perimeter = find_boundaries(mask, mode='inner')

    maskpad = np.zeros_like(perimeter)

    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

    new_mask = np.bitwise_or(maskpad, mask)

    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)

##############
# DEMO FOR SHOWING OFF DILATE_BOUNDARY EFFECT

if __name__ == "__main__":

    from placenta import get_named_placenta
    from frangi import frangi_from_image
    import matplotlib.pyplot as plt
    import numpy as np
    from skimage.exposure import rescale_intensity

    import os.path

    dest_dir = 'demo_output'
    img = get_named_placenta('T-BN0164923.png')

    #radius = 30
    sigma = 3
    radius = 80
    # IMG WITHOUT DILATING, THEN IMAGE WITH DILATING
    #sigma = radius/4

    #inset = np.s_[:,:]
    inset = np.s_[800:1000,500:890]
    #inset = np.s_[100:300,300:500]

    D = dilate_boundary(img, radius=radius)


    # SAVE IT IN THE RIGHT DIRECTORY, ETC plt.savefig(

    # NOW SHOW FRANGI ON THESE IMAGES

    # NOW SHOW THE SAME PICTURE

    Fimg = frangi_from_image(img, sigma, dark_bg=False, dilation_radius=None)
    #FD = frangi_from_image(img, sigma, dark_bg=False, dilation_radius=radius)
    FD = frangi_from_image(D, sigma, dark_bg=False)

    #Fimg = rescale_intensity(Fimg)
    #FD = rescale_intensity(FD)
```

```
100
101     fig, axes = plt.subplots(ncols=2, nrows=2, figsize=(14, 7))
102
103     axes[0,0].imshow(img[inset].filled(0), cmap=plt.cm.gray)
104     axes[0,1].imshow(D[inset].filled(0), cmap=plt.cm.gray)
105
106     axes[1,0].imshow(Fimg[inset].filled(0), cmap=plt.cm.nipy_spectral)
107     axes[1,1].imshow(FD[inset].filled(0), cmap=plt.cm.nipy_spectral)
108
109     # this is a hack directly from matplotlib, that's why it's so ugly.
110     plt.setp([a.get_xticklabels() for a in axes[0, :]], visible=False)
111     plt.setp([a.get_yticklabels() for a in axes[:, 1]], visible=False)
112
113     fig.tight_layout()
114
115     plt.savefig(os.path.join(dest_dir, "boundary_dilation_demo.png"), dpi=300)
```

listings/preprocessing.py

```
1  #!/usr/bin/env python3
2
3  from skimage.morphology import binary_dilation, disk, remove_small_objects
4  from skimage.restoration import inpaint_biharmonic
5  import numpy as np
6  import numpy.ma as ma
7  from scipy.ndimage import label
8  from skimage.util import img_as_float
9  from skimage.segmentation import find_boundaries
10 from plate_morphology import dilate_boundary
11
12
13 def inpaint_glare(img, threshold=175, window_size=15, mask=None):
14     """
15     img is a masked array type uint [0,255]
16     """
17
18     # bool array, true where glare
19     if mask is None:
20         glared = mask_glare(img, threshold=threshold, mask_only=True)
21     else:
22         glared = mask
23
24     B = ma.masked_array(img, mask=glared)  # masked background *and* glare
25     new_img = img.copy()  # copy values of original image (will rewrite)
26     d = int(window_size)
27
28     for j, k in zip(*np.where(glared)):
29         # rewrite all glared pixels with the mean of nonmasked elements
30         # in a window_size window. (this doesn't check OoB, be careful!)
31         new_img[j, k] = B[j-d:j+d, k-d:k+d].compressed().mean()
32
33     return new_img
34
35
36 def inpaint_with_boundary_median(img, threshold=175, mask=None):
37     """
38     mask glare pixels, then replace by the median value on the mask's boundary
39     """
40     if mask is None:
41         glared = mask_glare(img, threshold=threshold, mask_only=True)
42     else:
43         glared = mask
```

```
44
45      B = ma.masked_array(img, mask=glared)
46
47      new_img = img.copy()  # copy values of original image (will rewrite)
48      bounds = find_boundaries(glared)
49      lb, _ = label(bounds)
50      fill_vals = np.zeros_like(img.data)
51
52      # for each boundary of masked region, find the median value of the img
53      for lab in range(1, lb.max()+1):
54          inds = np.where(lb == lab)
55          fill_vals[inds] = nz_median(B[inds])
56
57      # label masked regions together with their boundaries (they'll be
58      # connected)
59      lm, _ = label(np.logical_or(glared, lb != 0))
60
61      # fill the masked areas with the corresponding fill value
62      for lab in range(1, lm.max()+1):
63          inds = np.where(lm == lab)
64          # find locations of filled values corresponding to this label
65          # median in case there's overlapped regions? (sloppy)
66          replace_value = nz_median(fill_vals[inds])
67
68          if replace_value == 0:
69              raise
70
71          fill_vals[inds] = replace_value
72
73      # now fill in the values
74      new_img[glared] = fill_vals[glared]
75
76      return new_img
77
78
79  def nz_median(A):
80
81      if ma.is_masked(A):
82          relevant = A[A > 0].compressed()
83      else:
84          relevant = A[A > 0]
85
86      return np.median(relevant)
87
88
89  def inpaint_hybrid(img, threshold=175, min_size=64, boundary_radius=10):
90      """
91      use biharmonic inpainting in larger, inner areas (important stuff)
92      and median inpainting in smaller areas and along boundary
93      """
94
95      glare = mask_glare(img, threshold=threshold, mask_only=True)
96
97      glare_inside = dilate_boundary(glare, mask=img.mask,
98                                     radius=boundary_radius).filled(0)
99
100     large_glare = remove_small_objects(glare_inside, min_size=min_size,
101                                        connectivity=2)
102     small_glare = np.logical_and(glare, np.invert(large_glare))
103
104     # inpaint smaller and less important values with less expensive method
105     inpainted = inpaint_with_boundary_median(img, mask=small_glare)
106     hybrid = img_as_float(inpainted) # scale 0 to 1
107
```

```
    # inpaint larger regions with biharmonic inpainting
    large_inpainted = inpaint_biharmonic(img.filled(0), mask=large_glare)

    # now overwrite with these values
    hybrid[large_glare] = large_inpainted[large_glare]

    # put on old image mask
    return ma.masked_array(hybrid, mask=img.mask)

def inpaint_with_biharmonic(img, threshold=175):
    """
    use biharmonic inpainting *all* glare
    """
    glare = mask_glare(img, threshold=threshold, mask_only=True)
    inpainted = inpaint_biharmonic(img_as_float(img.filled(0)), mask=glare)

    if ma.is_masked(img):
        return ma.masked_array(inpainted, mask=img.mask)
    else:
        return inpainted

def mask_glare(img, threshold=175, mask_only=False):
    """
    for demoing purposes, with placenta.show_mask

    if mask_only, just return the mask. Otherwise return a copy of img with
    that added to the mask. If you want the original mask to be ignored,
    just pass img.filled(0) ya doofus

    threshold is expected to be of the same dtype as img *unless# it assumes
    its default value, in which case the threshold will be converted to a float

    """
    # if img.dtype is floating but threshold value is still the default
    # this could be generalized
    if np.issubdtype(img.dtype, np.floating) and (threshold == 175):
        threshold = 175 / 255
    # region to inpaint
    inp = (img > threshold)

    # get a larger area around the specks
    inp = binary_dilation(inp, selem=disk(1))

    # remove anything large
    #inp = white_tophat(inp, selem=disk(3))

    if mask_only:
        return inp
    else:
        # both the original background *and* these new glared regions
        # are masked
        return ma.masked_array(img, mask=inp)


DARK_RED = np.array([103, 15, 23]) / 255.

# test it on a particularly bad sample
if __name__ == "__main__":

    from placenta import get_named_placenta, show_mask
    import matplotlib.pyplot as plt

    filename = 'T-BN0204423.png'  # a particularly glary sample
    img = get_named_placenta(filename)
```

```
172
173     img = ma.masked_array(img_as_float(img), mask=img.mask)
174     crop = np.s_[150:500, 150:800]  # indices to zoom in on the region
175     zoom = np.s_[300:380, 300:380]  # even smaller region
176
177     slc = zoom  # which view to use
178
179     masked = mask_glare(img)  # for viewing
180
181     inpainted = inpaint_glare(img)
182     minpainted = inpaint_with_boundary_median(img)
183     hinpainted = inpaint_hybrid(img)
184     binpainted = inpaint_with_biharmonic(img)
185
186     # view the closeup like this
187     minpainted_view = show_mask(minpainted[slc], interactive=False,
188                                 mask_color=DARK_RED)
189     inpainted_view = show_mask(inpainted[slc], interactive=False,
190                                mask_color=DARK_RED)
191     masked_view = show_mask(masked[slc], interactive=False,
192                             mask_color=DARK_RED)
193     img_view = show_mask(img[slc], interactive=False,
194                          mask_color=DARK_RED)
195     hinpainted_view = show_mask(hinpainted[slc], interactive=False,
196                                 mask_color=DARK_RED)
197     binpainted_view = show_mask(binpainted[slc], interactive=False,
198                                 mask_color=DARK_RED)
199
200     # view them all next to each other
201
202     IMGS = np.vstack((
203         np.hstack((img_view, masked_view, inpainted_view)),
204         np.hstack((minpainted_view, binpainted_view, hinpainted_view))))
205
206     # THEN IMSAVE
207
208     # plt.imsave('preprocessing_comparison_cropped.png',  IMGS)
209     # plt.imsave('preprocessing_comparison_zoomed.png',  IMGS)
210
211     # if it's zoomed, then rescale the output in GIMP to 4x
```

listings/process_NCS_xcfs.py

```
1  #!/usr/bin/env python
2
3  """
4  This should be a plugin to take images from the folder NCS_vessel_GIMP_xcf
5  and create trace, mask, and backgrounded images from each xcf file.
6
7  to use:
8  chmod +x and then copy or link to  ~/gimp-2.x/plug-ins/
9  """
10
11 from gimpfu import *
12 import os.path
13 from functools import partial
14
15 #basefile, ext = os.path.splitext(xcffile)
16
17 def _outname(base, s=None):
18
19     #base = base.split("_", maxsplit=1)[0]
20     if s is None:
```

```
21          stubs = (base, 'png')
22      else:
23          stubs = (base, s, 'png')
24      file
25      filename = '.'.join(stubs)
26
27      return os.path.join(os.getcwd(), filename)
28
29  # get active image
30  def process_NCS_xcf(timg,tdrawable):
31      img = timg
32      basename, _ = os.path.splitext(img.name) # split off extension .xcf
33      basename = basename.split("_")[0] # only get T-BN-kjlksf part
34      print "*"*80
35      print '\n\n'
36
37      print "Processing " , img.name
38      # generate output names easier
39      outname = partial(_outname, base=basename)
40
41      # get coordinates of the center
42      cx, cy = img.height // 2 , img.width // 2
43
44      # disable the undo buffer
45      img.disable_undo()
46
47      #perimeter = pdb.gimp_image_get_layer_by_name(img, 'perimeter')
48
49      for layer in img.layers:
50          if layer.name.lower() in ('perimeter', 'perimeters'):
51              # .copy() has optional arg of "add_alpha_channel"
52              mask = layer.copy()
53              break
54      else:
55          print "Could not find a perimeter layer."
56          print "Layers of this image are:"
57          for n,layer in enumerate(img.layers):
58              print "\t", n, ":", layer.name
59          print "Skipping this file."
60
61          return
62
63      for layer in img.layers:
64          layer.visible = False
65
66      mask.name = "mask" # name the new layer
67      img.add_layer(mask,0) # add in position 0 (top)
68
69      pdb.gimp_layer_flatten(mask) # Remove Alpha Channel.
70
71      # save the annotated perimeter file (for calculations later)
72      pdb.gimp_file_save(img,mask, outname(s="ucip"), '')
73
74      # remove unneeded annotations from mask layer
75      # color exchange yellow & blue to black
76      pdb.plug_in_exchange(img,mask,255,255,0,0,0,0,1,1,1)
77      pdb.plug_in_exchange(img,mask,0,0,255,0,0,0,1,1,1)
78
79      # set FG color to black (for tools, not of image)
80      gimp.set_foreground(0,0,0)
81
82      # Bucket Fill Inside black (center pixel is hopefully fine,
83      # do rest manually
84      pdb.gimp_edit_bucket_fill(mask,0,0,100,0,0,cx,cy)
```

```
85
86      # Color Exchange Green to White.
87      pdb.plug_in_exchange(img,mask,0,255,0,255,255,255,1,1,1)
88
89      # Color Exchange Cyan (00ffff) to White.
90      pdb.plug_in_exchange(img,mask,0,255,255,255,255,255,1,1,1)
91
92      # Export Layer as Image called "f".mask.png
93      pdb.gimp_file_save(img,mask, outname(s="mask"), '')
94
95      # invert (so exterior is now black)
96      pdb.gimp_invert(mask)
97      mask.mode = DARKEN_ONLY_MODE # the constant 9
98
99      # set bottom layer (placenta) to visible
100     raw = img.layers[-1]
101     raw.visible = True
102
103     # now make a new layer called 'raw_img' from visible
104     base = pdb.gimp_layer_new_from_visible(img,img,'base')
105     img.add_layer(base,0)
106     pdb.gimp_file_save(img , base, outname(s=None) , '')
107
108     # now get rid of mask and save the raw image
109     mask.visible = False
110     pdb.gimp_file_save(img, base, outname(s='raw') ,'')
111
112
113     # now make the other one visible (this is dumb)
114     for layer in img.layers:
115         if layer.name.lower() in ("arteries", "veins"):
116             layer.visible = True
117         else:
118             layer.visible = False
119     # now with these two visible, merge them and add layer
120     trace = pdb.gimp_layer_new_from_visible(img,img,'trace')
121     img.add_layer(trace,0)
122
123     pdb.gimp_layer_flatten(trace) # remove alpha channel
124
125     # don't turn binary anymore
126     #pdb.gimp_desaturate(trace) # turn to grayscale
127     #pdb.gimp_threshold(trace,255,255) # anything not 255 turns black
128
129     pdb.gimp_file_save(img, trace, outname(s='ctrace') ,'')
130
131     # now extract an each type individually.
132     found = 0
133     for subtype in ("arteries", "veins"):
134         for layer in img.layers:
135             if layer.name.lower() == subtype:
136                 layer.visible = True
137                 pdb.gimp_layer_flatten(layer) # remove alpha channel
138                 pdb.gimp_file_save(img, layer, outname(s=subtype), '')
139                 layer.mode = 9 # set to darken only (for merging)
140                 found += 1
141             else:
142                 layer.visible = False
143     if found < 2:
144         print "WARNING! Could not find appropriate artery/vein layers."
145
146
147     print "Saved. "
148
```

```
register(
    "process_NCS_xcf",
    "Create base image + trace + mask from an NCS xcf file",
    "Create base image + trace + mask from an NCS xcf file",
    "Luke Wukmer",
    "Luke Wukmer",
    "2018",
    "<Image>/Image/Process_NCS_xcf...",
    "RGB*, GRAY*",
    [],
    [],
    process_NCS_xcf)

main()
```

listings/score.py

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

import numpy as np
from placenta import open_typefile, open_tracefile

def rgb_to_widths(T):
    """
    this will take an RGB trace image (MxNx3) and return a 2D (MxN)
    "labeled" trace corresponding to the traced pixel length.
    there is no distinguishing between arteries and vessels

    it's preferrable to do this in real-time so only one tracefile
    needs to be stored (making the sample folder less cluttered)
    although obviously at the expense of storing a larger image
    which is only needed for visualization purposes.

    Input:
        T: a MxNx3 RGB (uint8) array, where the colorations are
        assumed as described in NOTES below.

    Output:
        widthtrace: a MxN array whose inputs describe the width of the
        vessel (in pixels), see NOTES.

    Notes:

        The correspondence is as follows:
        3 pixels: "#ff006f",  # magenta
        5 pixels: "#a80000",  # dark red
        7 pixels: "#a800ff",  # purple
        9 pixels: "#ff00ff",  # light pink
        11 pixels: "#008aff",  # blue
        13 pixels: "#8aff00",  # green
        15 pixels: "#ffc800",  # dark yellow
        17 pixels: "#ff8a00",  # orange
        19 pixels: "#ff0015"   # bright red

    According to the original tracing protocol, the traced vessels are
    binned into these 9 sizes. Vessels with a diameter smaller than 3px
    are not traced (unless they're binned into 3px).

    Note: this does *not* deal with collisions. If you pass anything
    with addition (blended colors) as the ctraces are, you will have
    trouble, as those will not be registered as any of the colors above
    and will thus be ignored. If you want to handle data from both
```

```
    arterial *and* venous layers, you should do so outside of this
    function.
    """

    # a 2D picture to fix in with the pixel widths
    W = np.zeros_like(T[:,:,0])

    for pix, color in TRACE_COLORS.items():

        #ignore pixelwidths outside the specified range
        # get the 2D indices that are that color
        idx = np.where(np.all(T == color, axis=-1))
        W[idx] = pix


    return W

def merge_widths_from_traces(A_trace, V_trace, strategy='minimum'):
    """
    combine the widths from two RGB-traces A_trace and V_trace
    and return one width matrix according to `strategy`

    Parameters
    ----------
    A_trace: ndarray
        an MxNx3 matrix, where each pixel (along the
        last dimension) is an RGB triplet (i.e. each entry
        is an integer between [0,256). The colors each
        correspond to those in TRACE_COLORS, and (255,255,255)
        signifies "no vessel". This will normally correspond to
        the sample's arterial trace.
    V_trace: ndarray
        an MxNx3 matrix the same shape and other
        requirements as A_trace (see above). This will normally
        correspond to the sample's venous trace.
    strategy: keyword string
        when A_trace and V_trace coincide at some entry,
        this is the merging strategy. It should be a keyword
        of one of the following choices:

        "minimum": take the minimum width of the two traces
            (default). this is the sensible option if you
            are filtering out larger widths.
        "maximum": take the maximum width of the two traces
        "artery" or "A" or "top": take the width from A_trace
        "vein" or "V" or "bottom": take the width from V_trace

    Returns
    -------
        W : ndarray
        a width-matrix where each entry is a number 0 (no vessel), 3,5,7,...19

    Notes
    -----
    Since arteries grow over the veins on the PCSVN and are generally easier
    to extract, it might be preferable to indicate "arteries". In reality,
    each strategy is a compromise, and only by keeping track of both would
    you get the complete picture.

    No filtering out widths is done here.
    """
    assert A_trace.shape == V_trace.shape

    A = rgb_to_widths(A_trace)
```

```
    V = rgb_to_widths(V_trace)

    # collisions (where are widths both reported)
    c = np.logical_and(A!=0, V!=0)

    W = np.maximum(A,V)  # get the nonzero value
    if strategy == 'maximum':
        pass # already done, else rewrite the collisions
    elif strategy in ('arteries', 'A', 'top'):
        W[c] = A[c]
    elif strategy in ('veins', 'V', 'bottom'):
        W[c] = V[c]
    else:
        if strategy != 'minimum':
            print(f"Warning: unknown merge strategy: {strategy}")
            print("Defaulting to minimum strategy")

        W[c] = np.minimum(A[c], V[c])

    return W

def filter_widths(W, widths=None, min_width=3, max_width=19):
    """
    Filter a width matrix, removing widths according to rules.

    This function will take a 2D matrix of vessel widths and
    remove any widths outside a particular range (or alternatively,
    that are not included in a particular list)

    Should be roughly as easy as doing it by hand, except that you
    won't have to rewrite the code each time.

    Inputs:

    W:  a width matrix (2D matrix with elements 0,3,5,7,...19

    min_width: widths below this will be excluded (default is
                3, the min recorded width). assuming these
                are ints

    max_width: widths above this will be excluded (default is
                19, the max recorded width)

    widths: an explicit list of widths that should be returned.
            in this case the above min & max are ignored.
            this way you could include widths = [3, 17, 19] only
            """

    Wout = W.copy()
    if widths is None:
        Wout[W < min_width] = 0
        Wout[W > max_width] = 0

    else:
        # use numpy.isin(T, widths) but that's only in version 1.13 and up
        # of numpy this is basically the code for that though
        to_keep = np.in1d(W, widths, assume_unique=True).reshape(W.shape)
        Wout[np.invert(to_keep)] = 0
    return Wout


TRACE_COLORS = {
    3: (255, 0, 111),
    5: (168, 0, 0),
```

```
    7: (168, 0, 255),
    9: (255, 0, 255),
    11: (0, 138, 255),
    13: (138, 255, 0),
    15: (255, 200, 0),
    17: (255, 138, 0),
    19: (255, 0, 21)
}


def widths_to_rgb(w, show_non_matches=False):
    """Convert width matrix back to RGB values.

    For display purposes/convenience. Return an RGB matrix
    converting back from [3,5,7, ... , 19] -> TRACE_COLORS

    this doesn't do any rounding (i.e. it ignores anything outside of
    the default widths), but maybe you'd want to?
    """
    B = np.zeros((w.shape[0], w.shape[1], 3))

    for px, rgb_triplet in TRACE_COLORS.items():
        B[w == px, : ] = rgb_triplet

    if show_non_matches:
        # everything in w not found in TRACE_COLORS will be black
        B[w == 0, : ] = (255, 255, 255)
    else:
        non_filled = (B == 0).all(axis=-1)

        B[non_filled,:] = (255,255,255) # make everything white

    # matplotlib likes the colors as [0,1], so....
    return B / 255.


def _hex_to_rgb(hexstring):
    """
    there's a function that does this in matplotlib.colors
    but its scaled between 0 and 1 but not even as an
    array so this is just as much work

    ##TODO rewrite everything so this is useful if it's not been
    rewritten already.
    """
    triple = hexstring.strip("#")
    return tuple(int(x,16) for x in (triple[:2],triple[2:4],triple[4:]))


def confusion(test, truth, bg_mask=None, colordict=None, tint_mask=True):
    """
    distinct coloration of false positives and negatives.

    colors output matrix with
        true_pos if test[-] == truth[-] == 1
        true_neg if test[-] == truth[-] == 0
        false_neg if test[-] == 0 and truth[-] == 1
        false_pos if test[-] == 1 and truth[-] == 0

    if colordict is supplied: you supply a dictionary of how to
    color the four cases. Spec given by the default below:

    if tint mask, then the mask is overlaid on the image, not replacing totally
    colordict = {
```

```
        'TN': (247, 247, 247), # true negative
        'TP': (0, 0, 0) # true positive
        'FN': (241, 163, 64), # false negative
        'FP': (153, 142, 195), # false positive
        'mask': (247, 200, 200) # mask color (not used in MCC calculation)
        }
    """

    if colordict is None:
        colordict = {
            'TN': (247, 247, 247),  # true negative# 'f7f7f7'
            'TP': (0, 0, 0),  # true positive  # '000000'
            'FN': (241, 163, 64),  # false negative # 'f1a340' orange
            'FP': (153, 142, 195),  # false positive # '998ec4' purple
            'mask': (247, 200, 200)  # mask color (not used in MCC calculation)
            }

    #TODO: else check if mask is specified and add it as color of TN otherwise

    true_neg_color = np.array(colordict['TN'], dtype='f')/255
    true_pos_color = np.array(colordict['TP'], dtype='f')/255
    false_neg_color = np.array(colordict['FN'], dtype='f') /255
    false_pos_color = np.array(colordict['FP'], dtype='f')/255
    mask_color = np.array(colordict['mask'], dtype='f') /255

    assert test.shape == truth.shape

    # convert to bool
    test, truth = test.astype('bool'), truth.astype('bool')

    # RGB array size of test and truth for output
    output = np.zeros((test.shape[0], test.shape[1], 3), dtype='f')

    # truth conditions
    true_pos = np.bitwise_and(test==truth, truth)
    true_neg = np.bitwise_and(test==truth, np.invert(truth))
    false_neg = np.bitwise_and(truth, np.invert(test))
    false_pos = np.bitwise_and(test, np.invert(truth))

    output[true_pos,:] = true_pos_color
    output[true_neg,:] = true_neg_color
    output[false_pos,:] = false_pos_color
    output[false_neg,:] = false_neg_color

    # try to find a mask
    if bg_mask is None:
        try:
            bg_mask =  test.mask
        except AttributeError:
            # no mask is specified, we're done.
            return output

    # color the mask
    if tint_mask:
        output[bg_mask,:] += mask_color
        output[bg_mask,:] /= 2
    else:
        output[bg_mask,:] = mask_color

    return output


def compare_trace(approx, trace=None, filename=None,
                  sample_dir=None, colordict=None):
```

```
    """
    compare approx matrix to trace matrix and output a confusion matrix.
    if trace is not supplied, open the image from the tracefile.
    if tracefile is not supplied, filename must be supplied, and
    tracefile will be opened according to the standard pattern

    colordict are parameters to pass to confusion()

    returns a matrix
    """

    # load the tracefile if not supplied
    if trace is None:
        if filename is not None:
            try:
                trace = open_typefile(filename, 'trace')
            except FileNotFoundError:
                print("No trace file found matching ", filename)
                print("no trace found. generating dummy trace.")
                trace = np.zeros_like(approx)
        else:
            print("no trace supplied/found. generating dummy trace.")
            trace = np.zeros_like(approx)

    C = confusion(approx, trace, colordict=colordict)

    return C


def mcc(test, truth, bg_mask=None, score_bg=False, return_counts=False):
    """
    Matthews correlation coefficient
    returns a float between -1 and 1
    -1 is total disagreement between test & truth
    0 is "no better than random guessing"
    1 is perfect prediction

    bg_mask is a mask of pixels to ignore from the statistics
    for example, things outside the placental plate will be counted
    as "TRUE NEGATIVES" when there wasn't any chance of them not being
    scored as negative. therefore, it's not really a measure of the
    test's accuracy, but instead artificially pads the score higher.

    setting bg_mask to None when test and truth are not masked
    arrays should give you this artificially inflated score.
    Passing score_bg=True makes this decision explicit, i.e.
    any masks (even if supplied) will be ignored, and your count of
    false positives will be inflated.

    """
    true_pos = np.logical_and(test==truth, truth)
    true_neg = np.logical_and(test==truth, np.invert(truth))
    false_neg = np.logical_and(truth, np.invert(test))
    false_pos = np.logical_and(test, np.invert(truth))

    if score_bg:
        # take the classifications above as they are (nothing is masked)
        pass
    else:
        # if no specified mask, check the test array itself?
        if bg_mask is None:
            try:
                bg_mask = test.mask
            except AttributeError:
```

```
                # no mask is specified, we're done.
                bg_mask = np.zeros_like(test)

        # only get stats in the plate
        true_pos[bg_mask] = 0
        true_neg[bg_mask] = 0
        false_pos[bg_mask] = 0
        false_neg[bg_mask] = 0

    # now tally
    TP = true_pos.sum()
    TN = true_neg.sum()
    FP = false_pos.sum()
    FN = false_neg.sum()

    if not score_bg:
        total = np.invert(bg_mask).sum()
    else:
        total = test.size
    #print('TP: {}\t TN: {}\nFP: {}\tFN: {}'.format(TP,TN,FP,FN))
    #print('TP+TN+FN+FP={}\ntotal pixels={}'.format(TP+TN+FP+TN,total))
    # prevent potential overflow
    denom = np.sqrt(TP+FP)*np.sqrt(TP+FN)*np.sqrt(TN+FP)*np.sqrt(TN+FN)

    if denom == 0:
        # set MCC to zero if any are zero
        m_score =  0
    else:
        m_score = ((TP*TN) - (FP*FN)) / denom

    if return_counts:
        return m_score, (TP,TN,FP,FN)
    else:
        return m_score


def mean_squared_error(A,B):
    """
    get mean squared error between two matrices of the same size

    input:
        A, B : two ndarrays of the same size.

    output:

        mse:   a single number.
    """

    try:
        mse = ((A-B)**2).sum() / A.size

    except ValueError:
        print("inputs must be of the same size")
        raise

    return mse


if __name__ == "__main__":

    import matplotlib.pyplot as plt
    from skimage.data import binary_blobs

    A = binary_blobs()
```

```
    B = binary_blobs()

    true_neg_color = np.array([247,247,247], dtype='f') # 'f7f7f7'
    true_pos_color = np.array([0, 0, 0] , dtype='f')  # '000000'
    false_neg_color = np.array([241,163,64], dtype='f')# 'f1a340'
    false_pos_color = np.array([153,142,195], dtype='f') # '998ec4'

    C = confusion(A,B)

    fig, (ax0, ax1, ax2) = plt.subplots(nrows=1,
                                        ncols=3,
                                        figsize=(8, 2.5),
                                        sharex=True,
                                        sharey=True)

    ax0.imshow(A, cmap='gray')
    ax0.set_title('A')
    ax0.axis('off')
    ax0.set_adjustable('box-forced')

    ax1.imshow(B, cmap='gray')
    ax1.set_title('B')
    ax1.axis('off')
    ax1.set_adjustable('box-forced')

    ax2.imshow(C)
    ax2.set_title('confusion matrix of A and B')
    ax2.axis('off')
    ax2.set_adjustable('box-forced')

    fig.tight_layout()
```

listings/scoring.py

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

import numpy as np
from placenta import open_typefile, open_tracefile
from skimage.morphology import thin

def rgb_to_widths(T):
    """
    this will take an RGB trace image (MxNx3) and return a 2D (MxN)
    "labeled" trace corresponding to the traced pixel length.
    there is no distinguishing between arteries and vessels

    it's preferrable to do this in real-time so only one tracefile
    needs to be stored (making the sample folder less cluttered)
    although obviously at the expense of storing a larger image
    which is only needed for visualization purposes.

    Input:
        T: a MxNx3 RGB (uint8) array, where the colorations are
        assumed as described in NOTES below.

    Output:
        widthtrace: a MxN array whose inputs describe the width of the
        vessel (in pixels), see NOTES.

    Notes:

        The correspondence is as follows:
        3 pixels: "#ff006f",  # magenta
```

```
        5 pixels: "#a80000",  # dark red
        7 pixels: "#a800ff",  # purple
        9 pixels: "#ff00ff",  # light pink
        11 pixels: "#008aff",  # blue
        13 pixels: "#8aff00",  # green
        15 pixels: "#ffc800",  # dark yellow
        17 pixels: "#ff8a00",  # orange
        19 pixels: "#ff0015"   # bright red

    According to the original tracing protocol, the traced vessels are
    binned into these 9 sizes. Vessels with a diameter smaller than 3px
    are not traced (unless they're binned into 3px).

    Note: this does *not* deal with collisions. If you pass anything
    with addition (blended colors) as the ctraces are, you will have
    trouble, as those will not be registered as any of the colors above
    and will thus be ignored. If you want to handle data from both
    arterial *and* venous layers, you should do so outside of this
    function.
    """

    # a 2D picture to fix in with the pixel widths
    W = np.zeros_like(T[:,:,0])

    for pix, color in TRACE_COLORS.items():

        #ignore pixelwidths outside the specified range
        # get the 2D indices that are that color
        idx = np.where(np.all(T == color, axis=-1))
        W[idx] = pix


    return W

def merge_widths_from_traces(A_trace, V_trace, strategy='minimum'):
    """
    combine the widths from two RGB-traces A_trace and V_trace
    and return one width matrix according to `strategy`

    Parameters
    ----------
    A_trace: ndarray
        an MxNx3 matrix, where each pixel (along the
        last dimension) is an RGB triplet (i.e. each entry
        is an integer between [0,256). The colors each
        correspond to those in TRACE_COLORS, and (255,255,255)
        signifies "no vessel". This will normally correspond to
        the sample's arterial trace.
    V_trace: ndarray
        an MxNx3 matrix the same shape and other
        requirements as A_trace (see above). This will normally
        correspond to the sample's venous trace.
    strategy: keyword string
        when A_trace and V_trace coincide at some entry,
        this is the merging strategy. It should be a keyword
        of one of the following choices:

        "minimum": take the minimum width of the two traces
            (default). this is the sensible option if you
            are filtering out larger widths.
        "maximum": take the maximum width of the two traces
        "artery" or "A" or "top": take the width from A_trace
        "vein" or "V" or "bottom": take the width from V_trace

```

```
    Returns
    -------
        W : ndarray
        a width-matrix where each entry is a number 0 (no vessel), 3,5,7,...19

    Notes
    -----
    Since arteries grow over the veins on the PCSVN and are generally easier
    to extract, it might be preferable to indicate "arteries". In reality,
    each strategy is a compromise, and only by keeping track of both would
    you get the complete picture.

    No filtering out widths is done here.
    """
    assert A_trace.shape == V_trace.shape

    A = rgb_to_widths(A_trace)
    V = rgb_to_widths(V_trace)

    # collisions (where are widths both reported)
    c = np.logical_and(A!=0, V!=0)

    W = np.maximum(A,V)  # get the nonzero value
    if strategy == 'maximum':
        pass # already done, else rewrite the collisions
    elif strategy in ('arteries', 'A', 'top'):
        W[c] = A[c]
    elif strategy in ('veins', 'V', 'bottom'):
        W[c] = V[c]
    else:
        if strategy != 'minimum':
            print(f"Warning: unknown merge strategy: {strategy}")
            print("Defaulting to minimum strategy")

        W[c] = np.minimum(A[c], V[c])

    return W

def filter_widths(W, widths=None, min_width=3, max_width=19):
    """
    Filter a width matrix, removing widths according to rules.

    This function will take a 2D matrix of vessel widths and
    remove any widths outside a particular range (or alternatively,
    that are not included in a particular list)

    Should be roughly as easy as doing it by hand, except that you
    won't have to rewrite the code each time.

    Inputs:

    W:  a width matrix (2D matrix with elements 0,3,5,7,...19

    min_width: widths below this will be excluded (default is
                3, the min recorded width). assuming these
                are ints

    max_width: widths above this will be excluded (default is
                19, the max recorded width)

    widths: an explicit list of widths that should be returned.
            in this case the above min & max are ignored.
            this way you could include widths = [3, 17, 19] only
            """
```

```
158
159     Wout = W.copy()
160     if widths is None:
161         Wout[W < min_width] = 0
162         Wout[W > max_width] = 0
163
164     else:
165         # use numpy.isin(T, widths) but that's only in version 1.13 and up
166         # of numpy this is basically the code for that though
167         to_keep = np.in1d(W, widths, assume_unique=True).reshape(W.shape)
168         Wout[np.invert(to_keep)] = 0
169     return Wout
170
171
172 TRACE_COLORS = {
173     3: (255, 0, 111),
174     5: (168, 0, 0),
175     7: (168, 0, 255),
176     9: (255, 0, 255),
177     11: (0, 138, 255),
178     13: (138, 255, 0),
179     15: (255, 200, 0),
180     17: (255, 138, 0),
181     19: (255, 0, 21)
182 }
183
184
185 def widths_to_rgb(w, show_non_matches=False):
186     """Convert width matrix back to RGB values.
187
188     For display purposes/convenience. Return an RGB matrix
189     converting back from [3,5,7, ... , 19] -> TRACE_COLORS
190
191     this doesn't do any rounding (i.e. it ignores anything outside of
192     the default widths), but maybe you'd want to?
193     """
194     B = np.zeros((w.shape[0], w.shape[1], 3))
195
196     for px, rgb_triplet in TRACE_COLORS.items():
197         B[w == px, : ] = rgb_triplet
198
199     if show_non_matches:
200         # everything in w not found in TRACE_COLORS will be black
201         B[w == 0, : ] = (255, 255, 255)
202     else:
203         non_filled = (B == 0).all(axis=-1)
204
205         B[non_filled,:] = (255,255,255) # make everything white
206
207     # matplotlib likes the colors as [0,1], so....
208     return B / 255.
209
210
211 def _hex_to_rgb(hexstring):
212     """
213     there's a function that does this in matplotlib.colors
214     but its scaled between 0 and 1 but not even as an
215     array so this is just as much work
216
217     ##TODO rewrite everything so this is useful if it's not been
218     rewritten already.
219     """
220     triple = hexstring.strip("#")
221     return tuple(int(x, 16) for x in (triple[:2], triple[2:4], triple[4:]))
```

```
222
223
224 def skeletonize_trace(T, T2=None):
225     """
226     if T is a boolean matrix representing a trace, then thin it
227
228     if T is an RGB trace, then register it according to the
229     tracing protocol then thin it
230
231     if T2 is provided, do the same thing to T2 and then merge the two
232     """
233     if T.ndim == 3:
234         trace = (rgb_to_widths(T) > 0)  # booleanize it
235
236     thinned = thin(trace)
237
238     if T2 is None:
239         return thinned
240
241     else:
242         # do the same thing to second trace and merge it
243         if T2.ndim == 3:
244             trace_2 = (rgb_to_widths(T2) > 0)  # booleanize it
245         thinned_2 = thin(trace_2)
246
247         return np.logical_or(thinned, thinned_2)
248
249
250 def confusion(test, truth, bg_mask=None, colordict=None, tint_mask=True):
251     """
252     distinct coloration of false positives and negatives.
253
254     colors output matrix with
255         true_pos if test[-] == truth[-] == 1
256         true_neg if test[-] == truth[-] == 0
257         false_neg if test[-] == 0 and truth[-] == 1
258         false_pos if test[-] == 1 and truth[-] == 0
259
260     if colordict is supplied: you supply a dictionary of how to
261     color the four cases. Spec given by the default below:
262
263     if tint mask, then the mask is overlaid on the image, not replacing totally
264     colordict = {
265         'TN': (247, 247, 247), # true negative
266         'TP': (0, 0, 0) # true positive
267         'FN': (241, 163, 64), # false negative
268         'FP': (153, 142, 195), # false positive
269         'mask': (247, 200, 200) # mask color (not used in MCC calculation)
270         }
271     """
272
273     if colordict is None:
274         colordict = {
275             'TN': (247, 247, 247),  # true negative# 'f7f7f7'
276             'TP': (0, 0, 0),  # true positive  # '000000'
277             'FN': (241, 163, 64),  # false negative # 'f1a340' orange
278             'FP': (153, 142, 195),  # false positive # '998ec4' purple
279             'mask': (247, 200, 200)  # mask color (not used in MCC calculation)
280             }
281
282     #TODO: else check if mask is specified and add it as color of TN otherwise
283
284     true_neg_color = np.array(colordict['TN'], dtype='f')/255
285     true_pos_color = np.array(colordict['TP'], dtype='f')/255
```

```
286     false_neg_color = np.array(colordict['FN'], dtype='f') /255
287     false_pos_color = np.array(colordict['FP'], dtype='f')/255
288     mask_color = np.array(colordict['mask'], dtype='f') /255
289 
290     assert test.shape == truth.shape
291 
292     # convert to bool
293     test, truth = test.astype('bool'), truth.astype('bool')
294 
295     # RGB array size of test and truth for output
296     output = np.zeros((test.shape[0], test.shape[1], 3), dtype='f')
297 
298     # truth conditions
299     true_pos = np.bitwise_and(test==truth, truth)
300     true_neg = np.bitwise_and(test==truth, np.invert(truth))
301     false_neg = np.bitwise_and(truth, np.invert(test))
302     false_pos = np.bitwise_and(test, np.invert(truth))
303 
304     output[true_pos,:] = true_pos_color
305     output[true_neg,:] = true_neg_color
306     output[false_pos,:] = false_pos_color
307     output[false_neg,:] = false_neg_color
308 
309     # try to find a mask
310     if bg_mask is None:
311         try:
312             bg_mask =  test.mask
313         except AttributeError:
314             # no mask is specified, we're done.
315             return output
316 
317     # color the mask
318     if tint_mask:
319         output[bg_mask,:] += mask_color
320         output[bg_mask,:] /= 2
321     else:
322         output[bg_mask,:] = mask_color
323 
324     return output
325 
326 
327 def compare_trace(approx, trace=None, filename=None,
328                   sample_dir=None, colordict=None):
329     """
330     compare approx matrix to trace matrix and output a confusion matrix.
331     if trace is not supplied, open the image from the tracefile.
332     if tracefile is not supplied, filename must be supplied, and
333     tracefile will be opened according to the standard pattern
334 
335     colordict are parameters to pass to confusion()
336 
337     returns a matrix
338     """
339 
340     # load the tracefile if not supplied
341     if trace is None:
342         if filename is not None:
343             try:
344                 trace = open_typefile(filename, 'trace')
345             except FileNotFoundError:
346                 print("No trace file found matching ", filename)
347                 print("no trace found. generating dummy trace.")
348                 trace = np.zeros_like(approx)
349         else:
```

```
            print("no trace supplied/found. generating dummy trace.")
            trace = np.zeros_like(approx)

    C = confusion(approx, trace, colordict=colordict)

    return C


def mcc(test, truth, bg_mask=None, score_bg=False, return_counts=False):
    """
    Matthews correlation coefficient
    returns a float between -1 and 1
    -1 is total disagreement between test & truth
    0 is "no better than random guessing"
    1 is perfect prediction

    bg_mask is a mask of pixels to ignore from the statistics
    for example, things outside the placental plate will be counted
    as "TRUE NEGATIVES" when there wasn't any chance of them not being
    scored as negative. therefore, it's not really a measure of the
    test's accuracy, but instead artificially pads the score higher.

    setting bg_mask to None when test and truth are not masked
    arrays should give you this artificially inflated score.
    Passing score_bg=True makes this decision explicit, i.e.
    any masks (even if supplied) will be ignored, and your count of
    false positives will be inflated.

    """
    true_pos = np.logical_and(test==truth, truth)
    true_neg = np.logical_and(test==truth, np.invert(truth))
    false_neg = np.logical_and(truth, np.invert(test))
    false_pos = np.logical_and(test, np.invert(truth))

    if score_bg:
        # take the classifications above as they are (nothing is masked)
        pass
    else:
        # if no specified mask, check the test array itself?
        if bg_mask is None:
            try:
                bg_mask = test.mask
            except AttributeError:
                # no mask is specified, we're done.
                bg_mask = np.zeros_like(test)

        # only get stats in the plate
        true_pos[bg_mask] = 0
        true_neg[bg_mask] = 0
        false_pos[bg_mask] = 0
        false_neg[bg_mask] = 0

    # now tally
    TP = true_pos.sum()
    TN = true_neg.sum()
    FP = false_pos.sum()
    FN = false_neg.sum()

    if not score_bg:
        total = np.invert(bg_mask).sum()
    else:
        total = test.size
    #print('TP: {}\t TN: {}\nFP: {}\tFN: {}'.format(TP,TN,FP,FN))
    #print('TP+TN+FN+FP={}\ntotal pixels={}'.format(TP+TN+FP+TN,total))
```

```
    # prevent potential overflow
    denom = np.sqrt(TP+FP)*np.sqrt(TP+FN)*np.sqrt(TN+FP)*np.sqrt(TN+FN)

    if denom == 0:
        # set MCC to zero if any are zero
        m_score =  0
    else:
        m_score = ((TP*TN) - (FP*FN)) / denom

    if return_counts:
        return m_score, (TP,TN,FP,FN)
    else:
        return m_score


def mean_squared_error(A,B):
    """
    get mean squared error between two matrices of the same size

    input:
        A, B : two ndarrays of the same size.

    output:

        mse:   a single number.
    """

    try:
        mse = ((A-B)**2).sum() / A.size

    except ValueError:
        print("inputs must be of the same size")
        raise

    return mse


if __name__ == "__main__":

    import matplotlib.pyplot as plt
    from skimage.data import binary_blobs

    A = binary_blobs()
    B = binary_blobs()

    true_neg_color = np.array([247,247,247], dtype='f') # 'f7f7f7'
    true_pos_color = np.array([0, 0, 0] , dtype='f')  # '000000'
    false_neg_color = np.array([241,163,64], dtype='f')# 'f1a340'
    false_pos_color = np.array([153,142,195], dtype='f') # '998ec4'

    C = confusion(A,B)

    fig, (ax0, ax1, ax2) = plt.subplots(nrows=1,
                                        ncols=3,
                                        figsize=(8, 2.5),
                                        sharex=True,
                                        sharey=True)

    ax0.imshow(A, cmap='gray')
    ax0.set_title('A')
    ax0.axis('off')
    ax0.set_adjustable('box-forced')

    ax1.imshow(B, cmap='gray')
```

```
478     ax1.set_title('B')
479     ax1.axis('off')
480     ax1.set_adjustable('box-forced')
481 
482     ax2.imshow(C)
483     ax2.set_title('confusion matrix of A and B')
484     ax2.axis('off')
485     ax2.set_adjustable('box-forced')
486 
487     fig.tight_layout()
```

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Milton Abramowitz and Irene A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover, New York, ninth dover printing, tenth gpo printing edition, 1964.

[2] Nizar Almoussa, Brittany Dutra, Bryce Lampe, Pascal Getreuer, Todd Wittman, Carolyn Salafia, and Luminita Vese. Automated vasculature extraction from placenta images. In *Medical Imaging 2011: Image Processing*, volume 7962, page 79621L. International Society for Optics and Photonics, 2011.

UCLA REU

[3] 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.

[4] R Burden and J Faires. *Numerical Analysis*. Brooks/Cole, 9 edition, 2011.

[5] Jen-Mei Chang, Hui Zeng, Ruxu Han, Ya-Mei Chang, Ruchit Shah, Carolyn M Salafia, Craig Newschaffer, Richard K Miller, Philip Katzman, Jack Moye, et al. Autism risk classification using placental chorionic surface vascular network features. *BMC medical informatics and decision making*, 17(1):162, 2017.

[6] Ya-Mei Chang, Ruxu Han, Hui Zeng, Ruchit Shah, Craig Newschaffer, Richard Miller, Philip Katzman, John Moye, Carolyn Salafia, et al. Whole chorionic surface vessel feature analysis with the boruta method, and autism risk. *Placenta*, 45:75, 2016.

[7] SB Damelin and NS Hoang. On surface completion and image inpainting by biharmonic functions: Numerical aspects. *International Journal of Mathematics and Mathematical Sciences*, 2018, 2018.

[8] Karamatou Yacoubou Djima, Carolyn Salafia, Richard K Miller, Ronald Wood, Philip Katzman, Chris Stodgell, and Jen-Mei Chang. Enhancing placental chorionic surface vasculature from barium-perfused images with directional and multiscale methods. *Placenta*, 57:292–293, 2017.

[9] Bengt Fornberg. Generation of finite difference formulas on arbitrarily spaced grids. *Mathematics of computation*, 51(184):699–706, 1988.

[10] 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.

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

[12] 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

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

[14] Xiangmin Jiao and Hongyuan Zha. Consistent computation of first-and second-order differential quantities for surface meshes. In *Proceedings of the 2008 ACM symposium on Solid and physical modeling*, pages 159–170. ACM, 2008.

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

[16] Jan J. Koenderink. The structure of images. *Biological Cybernetics*, 50(5):363–370, Aug 1984.

[17] W. Kühnel, B. Hunt, and American Mathematical Society. *Differential Geometry: Curves - Surfaces - Manifolds*. Student mathematical library. American Mathematical Society, 2006.

[18] Holger Lange. Automatic glare removal in reflectance imagery of the uterine cervix. In *Medical Imaging 2005: Image Processing*, volume 5747, pages 2183–2193. International Society for Optics and Photonics, 2005.

[19] Ivan Laptev, Helmut Mayer, Tony Lindeberg, Wolfgang Eckstein, Carsten Steger, and Albert Baumgartner. Automatic extraction of roads from aerial images based on scale space and snakes. *Machine Vision and Applications*, 12(1):23–31, 2000.

[20] Tony Lindeberg. *On the construction of a scale-space for discrete images*. KTH Royal Institute of Technology, 1988.

[21] Tony Lindeberg. Scale-space for discrete signals. *IEEE transactions on pattern analysis and machine intelligence*, 12(3):234–254, 1990.

[22] Tony Lindeberg. Discrete derivative approximations with scale-space properties: A basis for low-level feature extraction. *Journal of Mathematical Imaging and Vision*, 3(4):349–376, 1993.

[23] Tony Lindeberg. Feature detection with automatic scale selection. *International journal of computer vision*, 30(2):79–116, 1998.

[24] 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.

[25] 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.

[26] Anca Morar, Florica Moldoveanu, and Eduard Gröller. Image segmentation based on active contours without edges. In *2012 IEEE 8th International Conference on Intelligent Computer Communication and Processing*, pages 213–220. IEEE, 2012.

[27] 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.

[28] 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.

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

[30] 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.

[31] J. H. Wilkinson, editor. *The Algebraic Eigenvalue Problem*. Oxford University Press, Inc., New York, NY, USA, 1988.