

ABSTRACT

AN OPTIMIZED FRANGI FILTER FOR FAST VASCULAR SEGMENTATION TOWARDS A FAST AUTOMATIC PLACENTAL CHORIONIC SURFACE VASCULAR NETWORK EXTRACTION

By

Lucas Allen 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 [5]. 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 technique to study placental images, but this approach is greatly hindered by the comparative irregularity of the placental surface as a whole, which introduces significant noise into the image domain. Prior work [13] has 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 provide an in depth mathematical background of the Frangi filter and a reasonable introduction to Gaussian scale space theory. Finally, we discuss an important advancement in implementation–scale space conversion for differentiation (i.e. gaussian blur) via Fast Fourier Transform, which offers a significant speedup. This allows us faster calculation of the eigenvalues of the Hessian, from which we calculate the Frangi filter, a vesselness measure.

We demonstrate the effectiveness of our sped-up implementation of the Frangi filter by performing a large ($N=40$) multiscale Frangi filter on a set of 201 placental images from a private database provided by the National Children’s Study (NCS). We then compare several approaches of merging the multiscale result into an approximation of the PCSVN and compare them to manual tracings of the network. We finally suggest several ways to improve upon our approximation, namely by using the Frangi result as a prefilter for more robust techniques, providing a brief demo using a random walker segmentation.

**AN OPTIMIZED FRANGI FILTER FOR FAST VASCULAR SEGMENTATION
TOWARDS A FAST AUTOMATIC PLACENTAL CHORIONIC SURFACE
VASCULAR NETWORK EXTRACTION**

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 Allen Wukmer

B.S., 2013, University of California, Los Angeles

December 2018

WE, THE UNDERSIGNED MEMBERS OF THE COMMITTEE,
HAVE APPROVED THIS THESIS

**AN OPTIMIZED FRANGI FILTER FOR FAST VASCULAR SEGMENTATION
TOWARDS A FAST AUTOMATIC PLACENTAL CHORIONIC SURFACE
VASCULAR NETWORK EXTRACTION**

By

Lucas Allen 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

Thank you to people for things.

Thank you for reading this.

TABLE OF CONTENTS

	Page
ACKNOWLEDGEMENTS	iii
LIST OF TABLES	vi
LIST OF FIGURES	vii
 CHAPTER	
1. INTRODUCTION	1
2. MATHEMATICAL METHODS	3
Problem Setup in Image Processing	3
Differential Geometry	4
Preliminaries of Differential Geometry	4
Curvature of a surface and its calculation	7
Principal Curvatures and Principal Directions	14
The Weingarten map and Principal Curvatures of a Cylindrical Ridge	20
The Frangi Filter: Uniscale	24
Anisotropy Measure	25
Structureness measure	27
The Frangi vesselness measure	27
The Frangi vesselness filter: Choosing parameters β and γ ..	28
Linear Scale Space Theory	29
Axioms	29
Uniqueness of the Gaussian Kernel	31
Scale Spaces over Discrete Structures	36
The Frangi Filter: A multiscale approach	38
Thresholding	38
Calculating the 2D Hessian	39
Convolution Speedup via FFT	40
Fourier Transform of a continuous 1D signal	40
Fourier Transform of a Discrete 1D signal	41
2D DFT Convolution Theorem	41
FFT	44

	Page
APPENDIX	Page
3. IMPLEMENTATIONS	49
Calculating the Hessian.....	49
4. RESEARCH PROTOCOL	50
Samples / Image Domain	50
A representative sample	50
Knowns and Unknowns	52
Data Cleaning and Preprocessing	53
Boundary Dilation.....	55
Cut Removal.....	57
Deglaring	58
Multiscale Setup	62
Scale-space post-processing.....	64
Multiscale Merging.....	64
Cleanup/Postprocessing	64
Measurements.....	64
Erode plate / dilate boundary	64
5. RESULTS AND ANALYSIS	68
Sample visual output	68
The confusion matrix.....	68
A Source of “False Negatives” in the NCS data set	68
Variations in the Data Set and Imperfections of the Ground Truth	68
Results.....	69
Answer Research Questions	69
6. CONCLUSION	70
Review of Work	70
Future research directions	70
APPENDICES	73
A. CODE LISTINGS	74
BIBLIOGRAPHY	134

LIST OF TABLES

TABLE		Page
1	Vessel width color code	52

LIST OF FIGURES

FIGURE	Page
1 Tangent plane of a graph	8
2 The graph of a cylindrical ridge of radius r	20
3 The principal eigenvectors at a ridge like structure	26
4 A representative placental sample and tracing.....	51
5 Preprocessed files from an NCS sample.....	54
6 Effect of boundary dilation on Frangi responses.....	56
7 Removing a “cut” from the placental plate.	58
8 Deglaring a sample using a hybrid inpainting method	59
9 Comparison of glare inpainting methods (detail)	60
10 Signed Frangi output (plate and inset) (Example 1).....	65
11 Signed Frangi output (plate and inset) (Example 1).....	66
12 “True” false positives and “False” false positives.....	69
13 Signed Frangi output (plate and inset) (Example 1).....	71
14 Signed Frangi output (plate and inset) (Example 1).....	72

CHAPTER 1

INTRODUCTION

From [5], 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. [5] has provided a method of automatically calculating such features from an extracted vascular network, but does so with manual tracing of the PCSVN in order to make these measurements. These manual tracings are labor-intensive, requiring 4 to 8 hours of labor for each trace. There has been work to automate this procedure [2] [13] [8]. 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. 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 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 naïve approaches that work with other image domains to fail completely.

Much work has been done 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 technique to study placental images, but this approach is greatly hindered by the comparative irregularity of the placental surface as a whole, which introduces significant noise into the image domain. Prior work [13] solved this problem by provided an additional local curvilinear filter to the Frangi result in an effort to remove some noise from the final extraction.

Here we provide an in depth mathematical background of the Frangi filter and a reasonable introduction to Gaussian scale space theory. Finally, we discuss an important advancement in implementation–scale space conversion for differentiation (i.e. gaussian blur) via Fast Fourier Transform, which offers a significant speedup. This allows us faster calculation of the eigenvalues of the Hessian, from which we calculate the Frangi filter, a vesselness measure.

We demonstrate the effectiveness of our sped-up implementation of the Frangi filter by performing a large ($N=40$) multiscale Frangi filter on a set of 201 placental images from a private database provided by the National Children’s Study (NCS). We then compare several approaches of merging the multiscale result into an approximation of the PCSVN and compare them to manual tracings of the network. Our ability to take many more scales into consideration allows us to be pickier about our thresholding, which significantly reduces noise experienced in previous efforts. We finally suggest several ways to improve upon our approximation, namely by using the Frangi result as a prefilter for more robust techniques, providing a brief demo using a random walker segmentation.

CHAPTER 2

MATHEMATICAL METHODS

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

Problem Setup in Image Processing

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

Definition 2.1 (Image as a pixel matrix).

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

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

Definition 2.2 (Image as an interpolated surface).

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

That is, the function h is identical to the pixel matrix \mathbf{I} at all integer inputs, and

simply a “smooth enough” interpolation of those points for all other values.

It is of course necessary to admit that 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 $h : U \rightarrow \mathbb{R}$ (as in definition 2.2) we define the graph, f , of h in the following definition.

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

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

Since the graph f is clearly one-to-one by definition, we may readily associate any

input $u \in U$ with its corresponding output $p \in f[U]$, i.e.

$p = f(u) = f(u_1, u_2) = (u_1, u_2, h(u_1, u_2))$, depending on whether we wish to focus on a point of a graph in terms of its input or in terms of the structure of the graph itself.

Our development of curvature ultimately will hinge upon a careful consideration of the tangent plane of f at a point p , for we will require a concrete definition of both the tangent space within the domain and image of f , as well as the so called "differential" of f , the lattermost of which we will only define for the immediate case required. Seeing that f is one-to-one should make a lot of this futzing about complete overkill, but I've yet to find a way to distill it. That is, this development works for any parametrized surface element, not necessarily a graph. Whatever for now.

Definition 2.4 (Tangent space of U at u).

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

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

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

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

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

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

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

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

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

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

Definition 2.7 (Tangent plane of a graph).

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

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 .

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

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

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

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

Curvature of a surface and its calculation

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

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

Definition 2.8 (Surface curve). *Given a closed interval $I \subset \mathbb{R}$, we call the regular curve $c : I \rightarrow \mathbb{R}^3$ a surface curve in the event that $\text{image}(c) \subset \text{image}(f)$ entirely. The one-to-one-ness of*

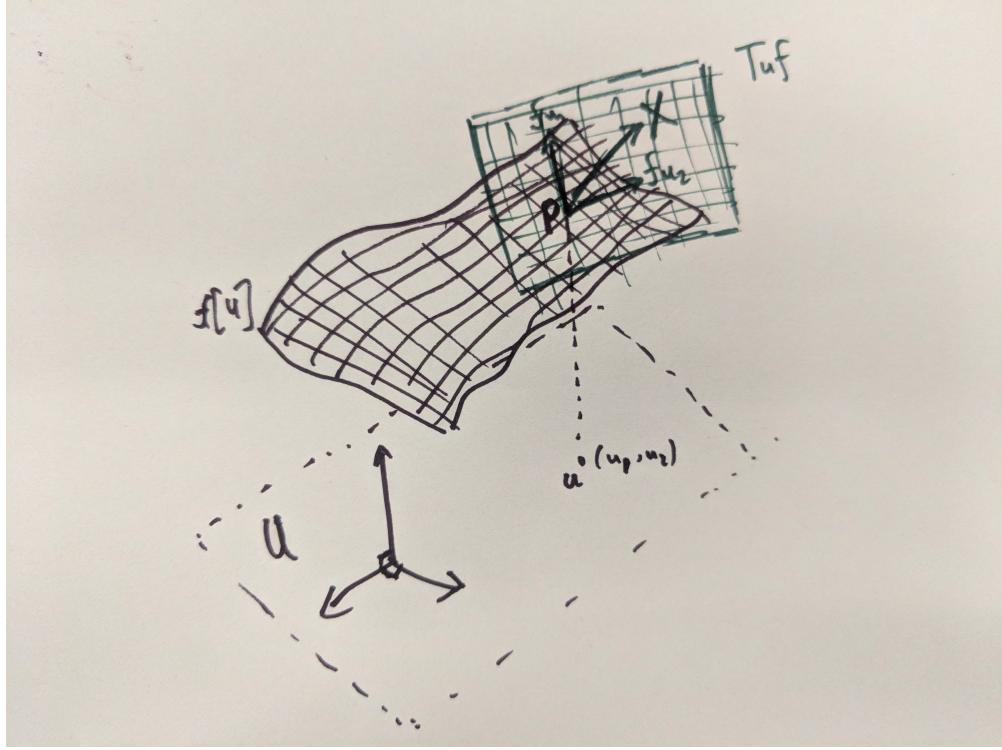


FIGURE 1: Tangent plane of a graph

the graph f ensures that we can define (for the given curve) an intermediary parametrization θ_c so that $c = f \circ \theta_c$. That is,

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

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

Note as well that the velocity of this particular curve lies within T_{uf} . This can be seen by an elementary application of chain rule:

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

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

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

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 $c : I \rightarrow \text{image}(f)$ 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*

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

Each partial above understood to be evaluated at the input $u \in U$; that is, we calculate $\frac{\partial f}{\partial u_i} \Big|_u$. The existence of the cross product in its definition makes it clear that $v \perp \frac{\partial f}{\partial u_i}$ 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 $\left\{ \frac{\partial v}{\partial u_1}, \frac{\partial v}{\partial u_2} \right\} \subset T_u f$, first note that at any particular $u \in U$, $\langle v, v \rangle = 1 \implies \frac{\partial}{\partial u_i} \langle v, v \rangle = 0$, and so by chain rule $2 \langle \frac{\partial v}{\partial u_i}, v \rangle = 0 \implies \frac{\partial v}{\partial u_i} \perp v$. Since $v \perp \text{span} \left\{ \frac{\partial f}{\partial u_i} \right\}$ as well (since v its outer product), in \mathbb{R}^3 , this implies $\text{span} \left\{ \frac{\partial v}{\partial u_i} \right\} \parallel \text{span} \left\{ \frac{\partial f}{\partial u_i} \right\}$. Thus, we have $\text{span} \left\{ \frac{\partial v}{\partial u_1}, \frac{\partial v}{\partial u_2} \right\} \subset T_u f$ 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 $\frac{dc}{dt}(u) = X$ 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 $u = \theta_c(p)$)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 $\left\{ -\frac{\partial v}{\partial u_1}, -\frac{\partial v}{\partial u_2} \right\}$.

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 $\widehat{L} \in R^{2 \times 2}$) we simply wish to find a linear transformation such that $\widehat{L} \left. \frac{\partial f}{\partial u_i} \right|_{T_u f} = - \left. \frac{\partial v}{\partial u_i} \right|_{T_u f}$ 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 $\frac{\partial f}{\partial u_i}|_{T_u f}$ 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

$$\left[\widehat{\mathcal{L}} \right] \begin{bmatrix} \frac{\partial f}{\partial u_1} \Big|_{T_u f} & \frac{\partial f}{\partial u_2} \Big|_{T_u f} \\ \hline \end{bmatrix} = \begin{bmatrix} \widehat{\mathcal{L}} \frac{\partial f}{\partial u_1} \Big|_{T_u f} & \widehat{\mathcal{L}} \frac{\partial f}{\partial u_2} \Big|_{T_u f} \\ \hline \end{bmatrix} \quad (2.20)$$

$$= \begin{bmatrix} \frac{\partial v}{\partial u_1} \Big|_{T_u f} & \frac{\partial v}{\partial u_2} \Big|_{T_u f} \\ \hline \end{bmatrix} \quad (2.21)$$

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

$$\Rightarrow \left[\widehat{\mathcal{L}} \right] \begin{bmatrix} -\frac{\partial f}{\partial u_1} & \\ -\frac{\partial f}{\partial u_2} & \end{bmatrix} \begin{bmatrix} \frac{\partial f}{\partial u_1} & \frac{\partial f}{\partial u_2} \\ \hline \end{bmatrix} = \begin{bmatrix} -\frac{\partial f}{\partial u_1} & \\ -\frac{\partial f}{\partial u_2} & \end{bmatrix} \begin{bmatrix} \frac{\partial v}{\partial u_1} & \frac{\partial v}{\partial u_2} \\ \hline \end{bmatrix} \quad (2.22)$$

We can simplify this greatly by defining

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

so that

$$\left[\widehat{\mathcal{L}} \right] \begin{bmatrix} g_{11} & g_{12} \\ g_{21} & g_{22} \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} \quad (2.24)$$

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

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

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

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

Our final goal 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_v 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 $\nabla_v \langle v, Av \rangle$:

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

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

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

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

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

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

□

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

Theorem 2.5 (Theorem of Olinde Rodrigues). *Fixing a point $u \in U$, a direction $X \in T_u f$ minimizes the normal curvature $\kappa_v = \langle LX, X \rangle$ 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{L} . Using the method of Lagrange multipliers, we define the Lagrangian:

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

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

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

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

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

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

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

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

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

principal directions and curvatures in a simple example.

Theorem 2.6 (Relationship between Hessian and Weingarten Map of a Graph). *Given the graph $f : U \rightarrow \mathbb{R}^3$ where $(x, y) \mapsto (x, y, h(x, y))$, the matrix representation of its Weingarten map is given by*

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

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

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

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

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

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

and we have the unit normal vector (Gauss map)

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

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

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

We then calculate each h_{ij} as

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

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

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

To calculate the second, we use

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

and thus

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

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

Thus the matrix of the Weingarten map \widehat{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{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{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:

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

We can then rewrite eq. (2.35) as

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

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 $\widehat{L} \approx \text{Hess}(h)$ 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

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

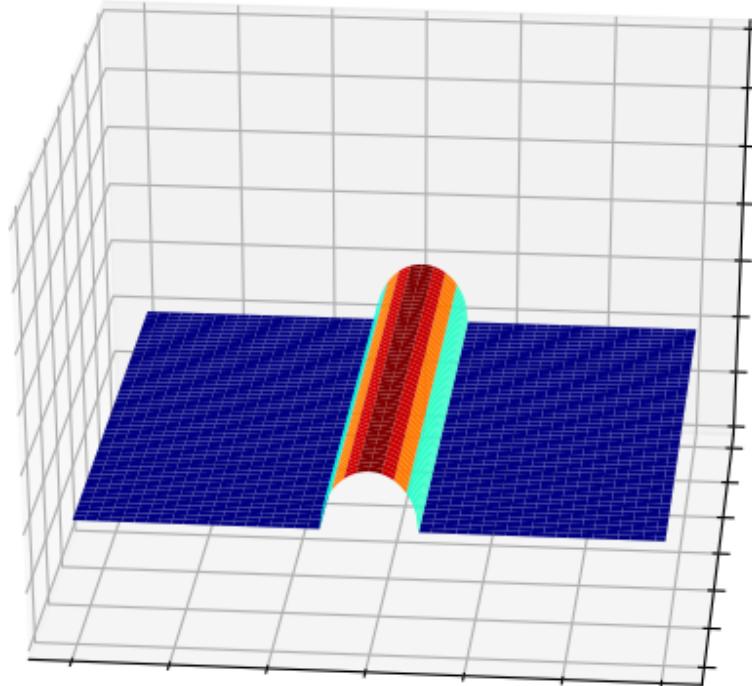


FIGURE 2: The graph of a cylindrical ridge of radius r

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

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

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

The gauss map is given by

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

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

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

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

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

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

$$= \begin{bmatrix} -\frac{1}{r} & 0 \\ 0 & 0 \end{bmatrix} \quad (2.54)$$

We see that $u_2 = (0, 1)$ and $u_1 = (1, 0)$ are eigenvectors for \widehat{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:

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

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

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

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, $\text{Hess}(x, y)$ is a real, symmetric matrix (since $L_{xy} = L_{yx}$ and L is a real function) and thus its eigenvalues are real and its eigenvectors are orthonormal to each other, as given by following 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

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

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

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

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

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

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

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

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

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 originally developed for vascular segmentation in images such as MRIs and it excels in that context.

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

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

where

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

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

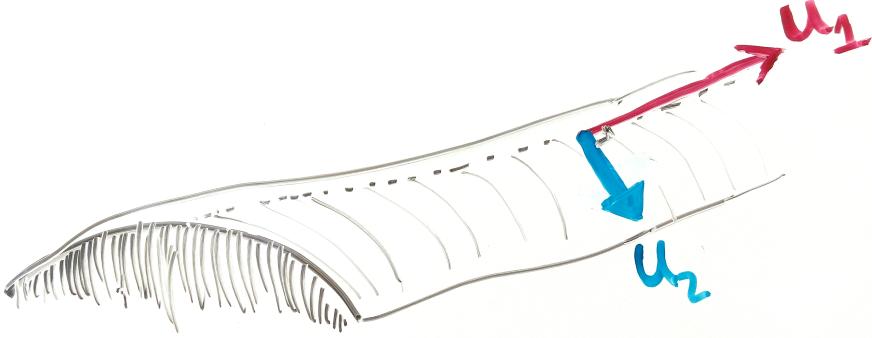


FIGURE 3: The principal eigenvectors at a ridge like structure

(in the direction of u_1). $\lambda_1 \approx 0$ and λ_2 is large and negative Note that the length of these vectors in this picture is not meant to represent their magnitudes, as u_2 should have a much larger relative magnitude by design!

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 $\exp\{\dots\}$ and $(1 - \exp\{\})$ are multiplied together are simply to ensure that the final vesselness measure V is largest when A is small and S is large enough, with rapidly decay in other situations.

Frangi further strengthened the filter by adding an additional case to in eq. (2.57), 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 identify when we are finding crests. A will still be small and S will still be large however if we identify a “trough”.

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

The Frangi vesselness filter: Choosing parameters β and γ

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

Frangi suggested for γ that half of the Frobenius norm of the Hessian matrix is appropriate, simply because the minimum value of S is zero, and its maximum value is approx the 2 norm of the Hessian. For β Frangi chose an innocuous intermediate point, $\beta = 1/2$ (and thus $2\beta^2 = 1/2$). As we will show later, choosing the structureness parameter γ is rather important for the context especially if the background (non-ridgelike structure) is significant and noisy. β should be strengthened/relaxed depending on how “flat” the ridgelike structure is. If there is a lot of gain then β should be smaller. If this is

not the case, a stronger filter can be created by requiring A to be much smaller.

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

Linear Scale Space Theory

There is obviously a major disconnect in the ideas presented above. Although the ideas presented above require differentiation of continuous surfaces, our image is in fact a discrete pixel. That is, our previous discussions have been in terms of an image as the continuous surface in definition 2.2, rather than the more realistic discrete pixel matrix as in definition 2.1. The present section seeks to address this disconnect. In particular, we seek to mitigate the bias of our limited sampling of the “true” 3D surface. Our main goal is to counter against some of the bias of our particular sampling. In particular, we wish to not over-represent structures that are clear at our resolution without giving appropriate weight to larger structures as well. Koenderink [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

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

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

Axioms

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

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

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

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

$$Au_0 = \lim_{\sigma \downarrow 0} \frac{T_\sigma u_0 - u_0}{\sigma} \quad (2.60)$$

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

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

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.*

$$T_\sigma u = T_{\sigma_1 + \sigma_2} u \quad (2.62)$$

Axiom 2.4 (Causality Condition). *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.

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

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.

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

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,

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

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.

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

We show that

$$u(\mathbf{x}, t) = (G_{\sqrt{2t}} \star u_0)(\mathbf{x}) \quad (2.67)$$

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,

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

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

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

$$= f(\alpha) \star g'(\alpha) \quad (2.71)$$

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

$$\Delta(f \star g) = \Delta(f) \star g = f \star \Delta(g) \quad (2.72)$$

□

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

Proof. We focus on the particular kernel

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

Then

$$\frac{\partial u}{\partial t}(\mathbf{x}, t) = \frac{\partial}{\partial t} \left(G_{\sqrt{2t}}(\mathbf{x}, t) \star u_0(\mathbf{x}) \right) \quad (2.73)$$

$$= \frac{\partial}{\partial t} \left(G_{\sqrt{2t}}(\mathbf{x}, t) \right) \star u_0(\mathbf{x}) \quad (2.74)$$

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

$$= \left[-\frac{1}{4\pi t^2} e^{(-|\mathbf{x}|^2/(4t))} + \frac{1}{4\pi t} \left(\frac{-|\mathbf{x}|^2}{4t^2} \right) e^{(-|\mathbf{x}|^2/(4t))} \right] \star u_0(\mathbf{x}) \quad (2.76)$$

$$= -\frac{1}{4t^2} \left(e^{(-|\mathbf{x}|^2/(4t))} + |\mathbf{x}|^2 G_{\sqrt{2t}}(\mathbf{x}, t) \right) \star u_0(\mathbf{x}) \quad (2.77)$$

and from the previous lemma,

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

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

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

By symmetry of argument we also may conclude

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

and so

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

Then, given lemma 2.9, we conclude

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

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

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

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

We then calculate the time derivative, using our particular choice of $\sigma(t) = \sqrt{2t}$ and

$A = \frac{1}{4\pi t}$ as:

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

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

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

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

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

Combining these results, we find that

$$\frac{\partial}{\partial t} [G_{\sqrt{2t}} \star u_0] = \Delta [G_{\sqrt{2t}} \star u_0] \quad (2.87)$$

as desired. \square

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 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

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

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*

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

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

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

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:

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

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

Thresholding

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

$$V_\Sigma := [V(x, y)]_{\substack{0 \leq x < M \\ 0 \leq y < N}} \quad (2.91)$$

Notably, Frangi [10] refrained from explicitly interpreting the probability

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)

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

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 $\mathcal{O}(N \cdot \log_2 N)$ 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:

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

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

$$F(\mu) := \mathcal{F}\{f(t)\} = \int_{-\infty}^{\infty} f(t) e^{i 2\pi \mu t} dt \quad (2.94)$$

An inverse transform will then recover our original signal:

$$f(t) = \mathcal{F}^{-1}\{F(\mu)\} = \int_{-\infty}^{\infty} F(\mu) e^{i 2\pi \mu t} dt \quad (2.95)$$

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 :

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

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

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

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

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

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

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:

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

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

and given the convolution of the two functions

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

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

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

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

Proof.

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

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

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

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

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

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

$$= MN \cdot \mathcal{D}^{-1} \{ FH \} \quad (2.109)$$

where

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

□

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

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

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

$$z^N - 1 = (z - 1)(z^{n-1} + \dots + z + 1) = (z - 1) \sum_{n=0}^{N-1} z^n. \quad (2.112)$$

thus giving us

$$0 = \left(e^{2\pi i(j-k)/N} - 1\right) \sum_{n=0}^{N-1} e^{2\pi i n(j-k)/N} \quad (2.113)$$

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 $e^{2\pi i n(j-k)/N} = \left(e^{2\pi i}\right)^{n(j-k)/N} = 1$ and thus the left side of eq. (2.111) reduces to

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

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

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

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

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:

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

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

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

$$= \mathcal{F}_y\{\mathcal{F}_x\{f(x, y)\}\} \quad (2.117)$$

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

Thus, to calculate the fourier transform $F(u, v)$ at the point u, v requires the computation of the transform of length N for each iterated point $x \in 0, \dots, M-1$. 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

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

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

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

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

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

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

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

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

Thus we can rewrite eq. (2.120) as

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

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

The major advance comes via using the identities eq. (2.121) to consider the

Fourier transform K frequencies later :

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

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

Comparing eq. (2.125) and eq. (2.127), we see that the expressions within parentheses are identical. What's more, these parenetical expressions are functionally identical to discrete fourier transforms themselves. Let's notate them as follows:

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

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

If we're calculating an M point transform (i.e. we're wishing to calculate $F(1), \dots, F(M)$), once we've calculated the first K discrete frequencies (i.e. $F(1), \dots, F(K)$) we may simply reuse the two values we've calculated in eq. (2.128) to calculate the next $F(K+1), \dots, F(K+K) = F(M)$. Since each expression in parentheses involves K complex multiplications and $K-1$ complex additions, we are effectively saving $K(2K-1)$ calculations in computing the entire spectrum $F(1), \dots, F(M)$. When M is large, the payoff is undeniable.

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

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 ruler 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 in fig. 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

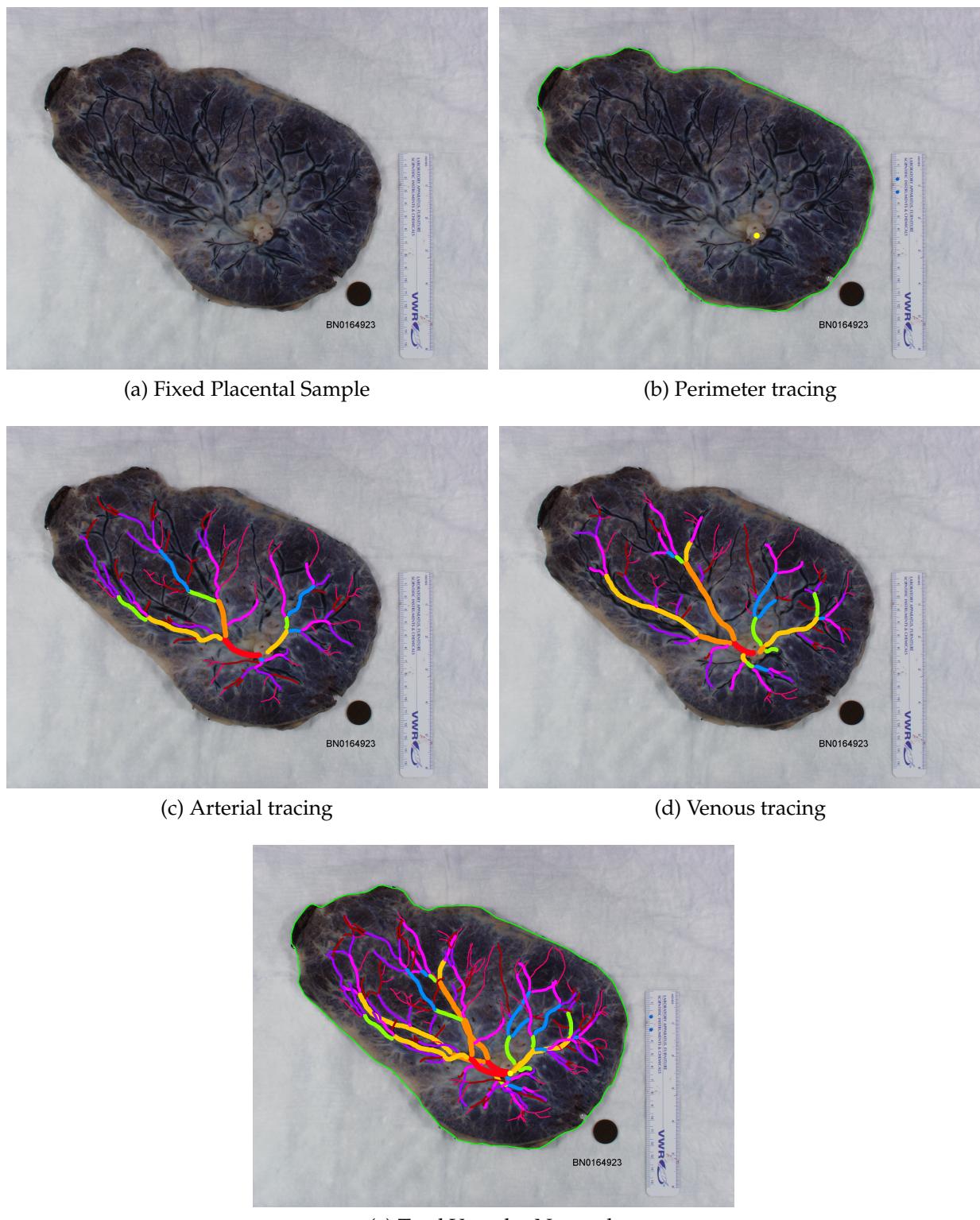


FIGURE 4: A representative placental sample and tracing

vessel width	color (hex value)	color name
3 pixels	#ff006f	magenta
5 pixels	#a80000	dark red
7 pixels	#a800ff	purple
9 pixels	#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.

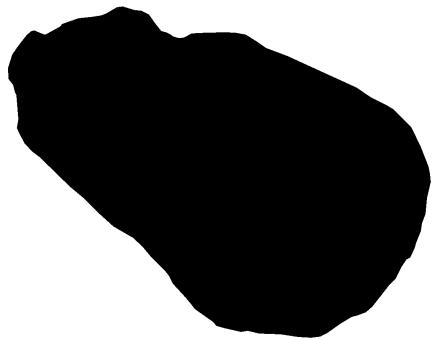
Finally, we will consider the location of the umbilical insertion point as a “known”, as the vessels around it are frequently impossible to see and we wish to exclude them from consideration. It is not unreasonable, however, to consider this to be a known—in future preparations of samples, we could simply 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.

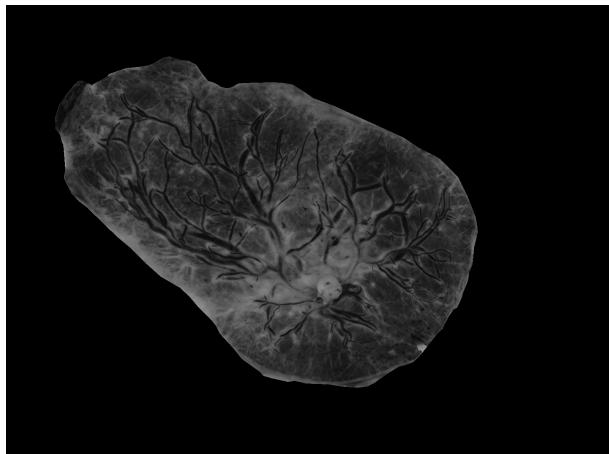
As a point of technicality, the grayscale image in fig. 5c is not actually produced directly by the extractor plug-in, but created when the 3 channel RGB image fig. 5b is imported at the start of the algorithm. This grayscale conversion is simply done for ease



(a) Background Mask (in white)



(b) Sample with BG removed



(c) Grayscale



(d) Trace / “Ground Truth”

FIGURE 5: Preprocessed files from an NCS sample

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:

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

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 effect 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 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

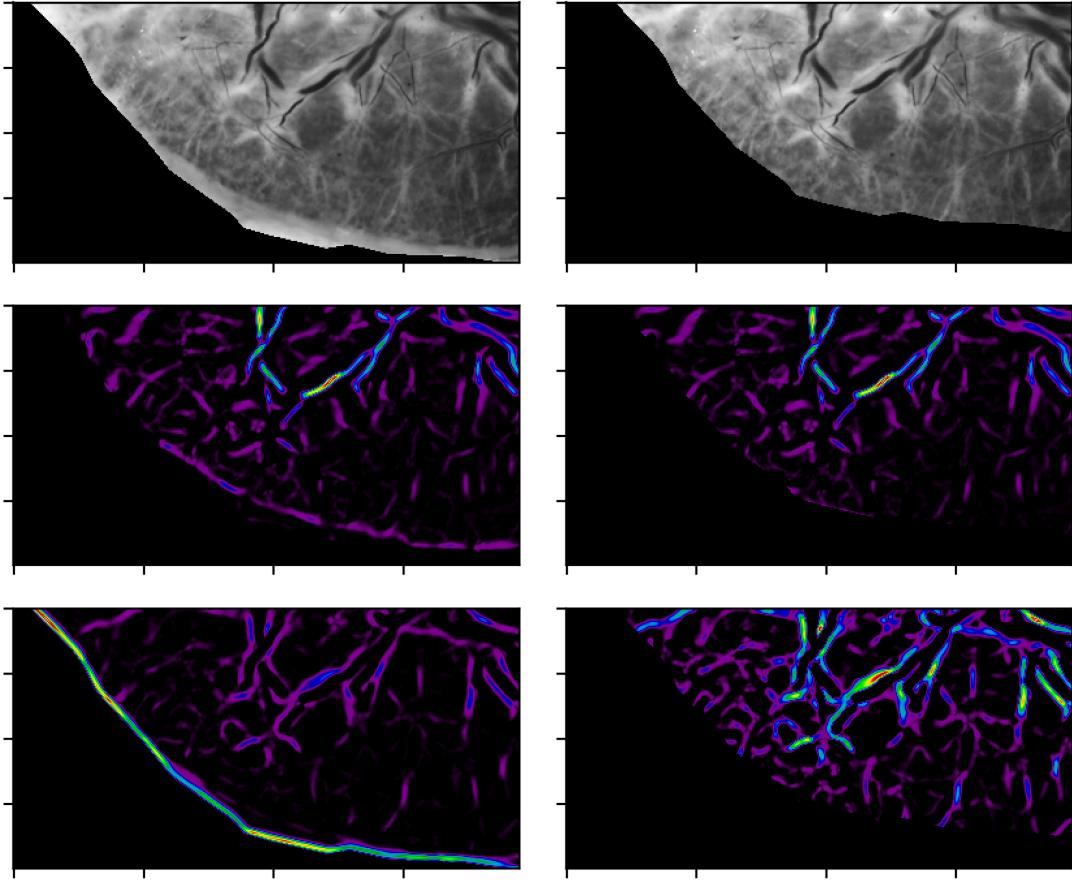


FIGURE 6: Effect of boundary dilation on Frangi responses

each direction (like a giant plus scale). 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.

Cut Removal

Sometimes there are small cuts that appear on the side of the placental plate, which can lead to large filter responses. We would like to filter these out to eliminate false positives in any way we can. These places are identified somewhat reliably by the tracing protocol with a blue dot. We perform a morphological watershedding in this area in an attempt to add this area to the mask. The threshold for the watershed is ultimately based on the background value of the blue dot; if this is incorrectly placed (or anything

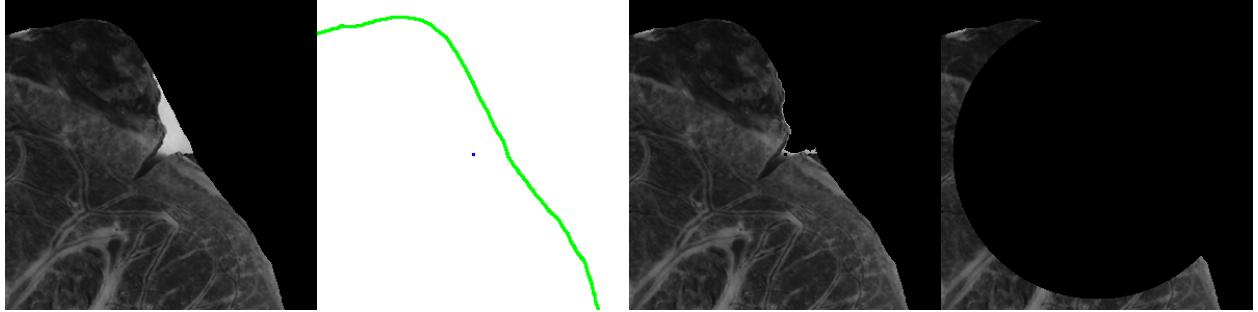


FIGURE 7: Removing a “cut” from the placental plate.

else weird happens), we can manually opt to simply remove a large area from consideration from the plate, as we do the umbilical cord insertion point. This is demonstrated in fig. 7. From left to right, we have the original masked placental sample, the cut mark noted on the perimeter layer, the improved mask after watershedding, and finally, the “scorched earth” approach in case the previous attempt failed.

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

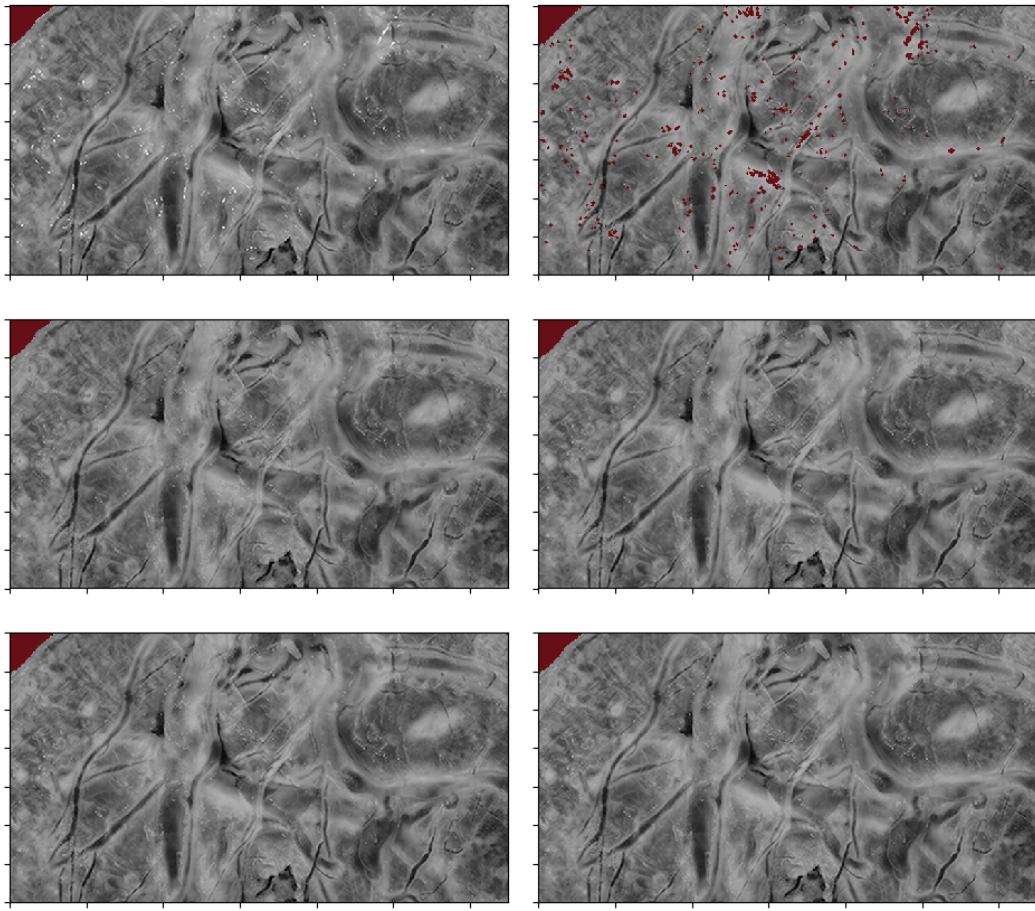


FIGURE 8: Deglaring a sample using a hybrid inpainting method

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

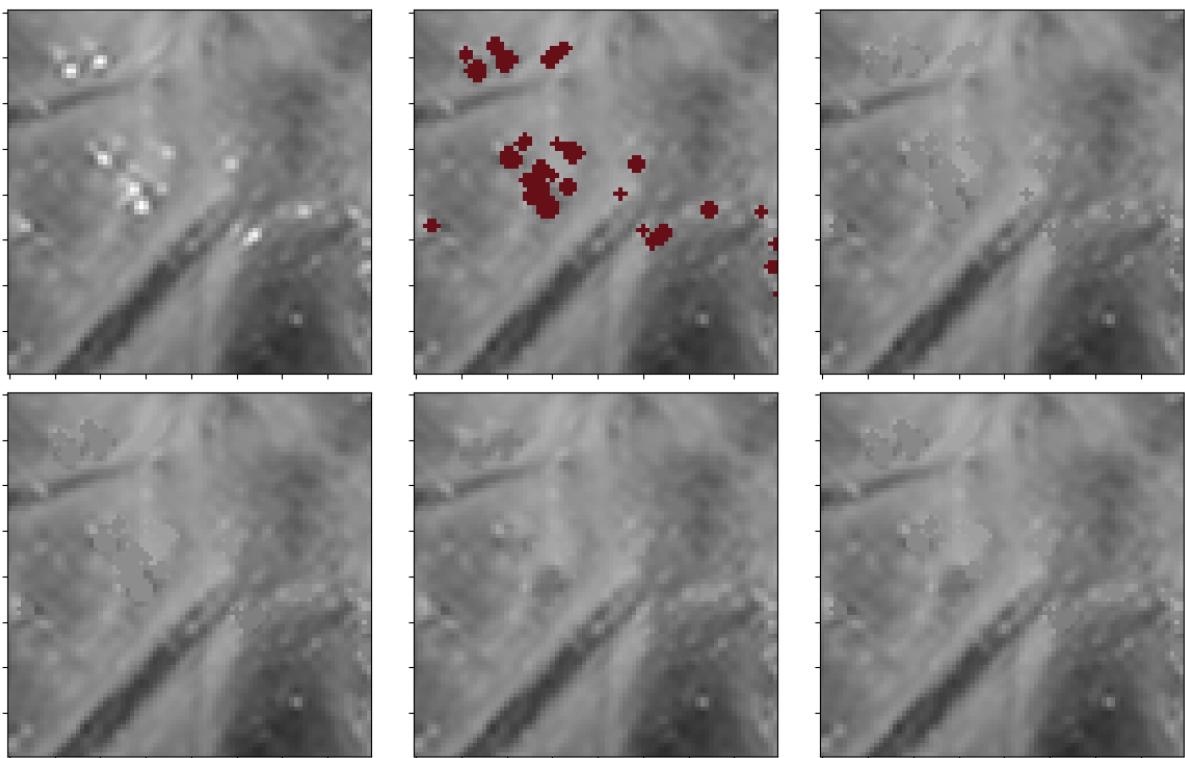


FIGURE 9: Comparison of glare inpainting methods (detail)

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 accurate biharmonic inpainting.

A comparison of these methods is shown in fig. 8, and a zoomed in portion is shown for in fig. 9. 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.

```

1 In [1]: %timeit inpaint_with_boundary_median(img)
2   1 loop, best of 3: 3.99 s per loop
3
4 In [2]: %timeit inpaint_with_biharmonic(img)
5   1 loop, best of 3: 22.3 s per loop
6
7 In [3]: %timeit inpaint_hybrid(img)
8   1 loop, best of 3: 6.49 s per loop
9
10 In [4]: inpaint_count = np.sum(np.logical_and(masked.mask, np.invert(img.mask)))
11 In [5]: plate_count = np.sum(np.invert(img.mask))
12 In [6]: inpaint_count / plate_count # ratio of inpainted pixels to total plate
13 Out[6]: 0.011444460505513942
14

```

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.

Multiscale Setup

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

Although we cannot expect *a priori* that there is an exact proportionality between our scale size σ and the width of a particular vessel [10], we generally expect to isolate narrower curvilinear structures at smaller scales, and thicker curvilinear structures at larger scales. 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

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

That is, the exponents are spaced linearly from m to M . This is achieved by the command `np.logspace(m, M, num=N)`. The idea is that the curvilinear content of the image will respond better at some particular scale, but there are diminishing returns as σ increases; while the filter's response may vary substantially between, say $\sigma = 1$ and $\sigma = 2$, there will probably not be a substantial difference in response between, say, $\sigma = 46$ and $\sigma = 47$. Historically, there was another benefit of using a logarithmic scale spice: computing the vesselness measure was very expensive, and thus it was simply not feasible to collect so many large scale readings. This is much less of an issue with the present implementation.

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.”

Once we have this chosen set Σ , we simply convolve the image with a discrete gaussian kernel with that standard deviation, then take gradients enough to get a matrix of partial second derivatives, the Hessian. We take its eigenvalues and then compute the Frangi filter according to section 2.3 and section 2.5. We use these to provide a couple examples of estimating the PCSVN network. The entire decision tree can be shown in the outline below. Indentations with “+” and “.” characters are lists of options at that point, where “+” is the default and “.” is for alternatives discussed elsewhere in the text.

For each sample:

- A) Preprocessing
 - 1) RGB to single channel
 - + Luminance transform
 - . Isolate green channel only (Almoussa, Huynh)
 - 2) Cut removal
 - 3) Remove glare
 - a) Mask glare
 - + Threshold (*175/255*)
 - . Threshold at *80%* of max intensity (Almoussa)
 - . Lange (2005) (multistep procedure, done in RGB space actually)
 - b) Post-process mask
 - + Dilate with radius *2*
 - . Do nothing
 - c) Inpaint glare
 - + Hybrid inpainting, with size threshold *32*
 - . Biharmonic inpainting
 - . Mean value of boundary
 - . Median value of boundary
 - . Windowed mean (radius: *15*)
- B) Multiscale Frangi filter
 - 1) Define parameters
 - a) Scales
 - = n_scales (default: *40*)
 - = scale_range (default *[-2, 3.5]*)
 - = scale_type (*logarithmic base 2* or linear or custom)
 - > build scales
 - b) Betas
 - = *0.5* each scale or custom range
 - c) Gammas
 - = strategy: (half L2 hessian norm or *half hessian frobenius norm*)
 - or custom value each scale
 - = redilate plate per scale (?)

- d) Dilate per scale
 - + Custom function of scale
(default $\max\{10, \text{int}(4\sigma)\}$ if $(\sigma < 20)$ else $\text{int}(2\sigma)$)*)
 - . No dilation
- e) Scale space convolution method
 - + Discrete Gaussian kernel with FFT
 - . Sampled gaussian kernel with FFT
 - . Sample gaussian kernel, standard convolution
- 2) For each sigma: do Uniscale Frangi Filter
 - a) gauss blur image with method from (1e)
 - b) take gradient across each axis, take gradient across each axis of gradient to get H_{xx} , H_{xy} , H_{yy}
 - c) find eigenvalues of hessian at each point (using np.eig)
and sort by magnitude
 - d) zero out principal directions according to Dilate Per Scale
 - e) zero out hessian according to $\max(\text{ceil}(\sigma), 10)$
 - f) Calculate Frangi Vesselness Measure
- C) Estimate PCSVN
 - 1) Approximate using strategy
 - a) Calculate F_{\max} and $F_{\max.\text{where}}$ -> F_{\max}
 - b) Threshold at 95th percentile -> approx
 - c) Threshold at fixed alpha
 - d) Margin adding to one of the above
 - e) Random walker after margin adding
 - 2) Compare to Trace
 - 3) Calculate Network Coverage and MCC score

Scale-space post-processing

Multiscale Merging

Cleanup/Postprocessing

Measurements

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 area which is affected should be larger than just the standard dropoff the

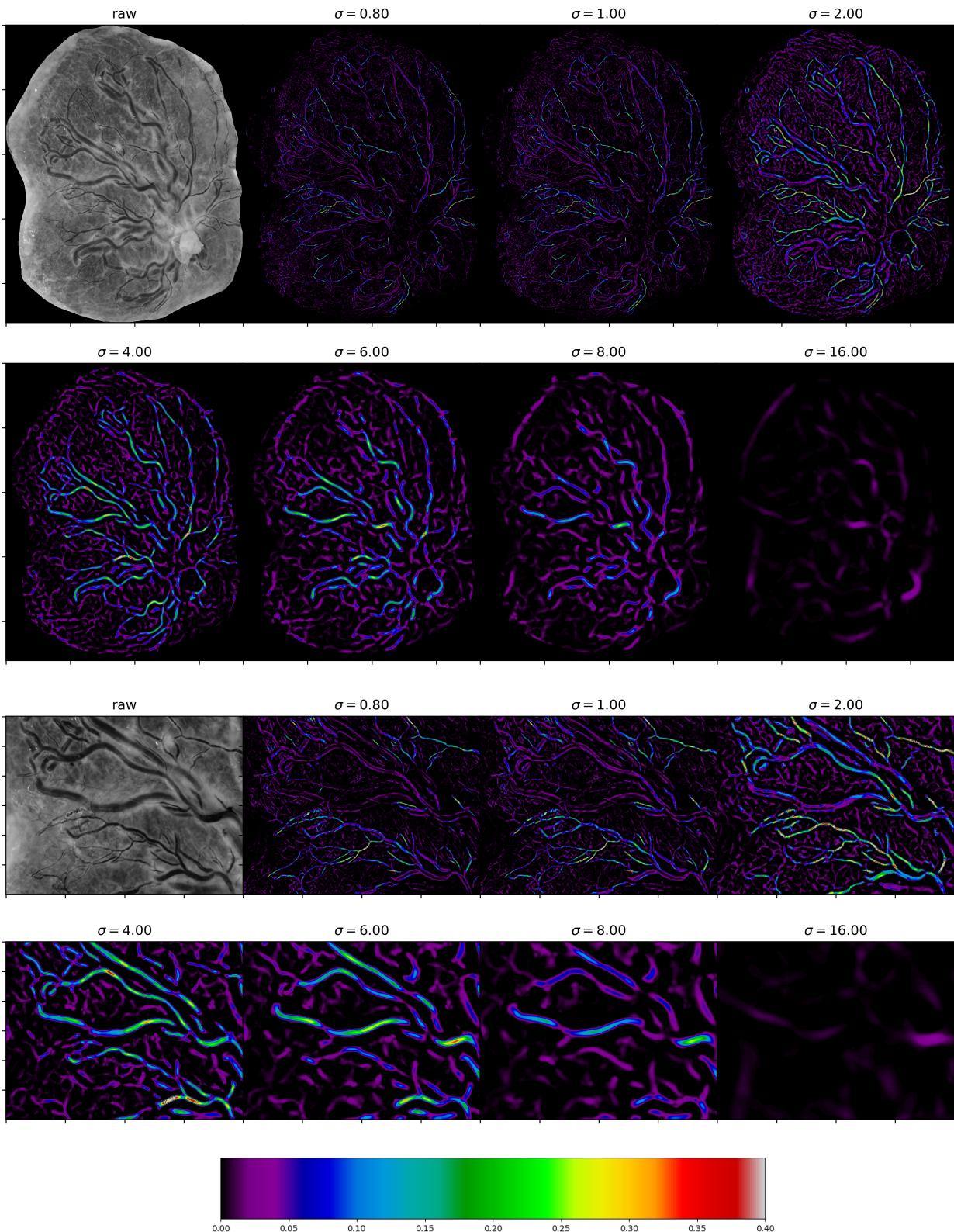


FIGURE 10: Signed Frangi output (plate and inset) (Example 1)

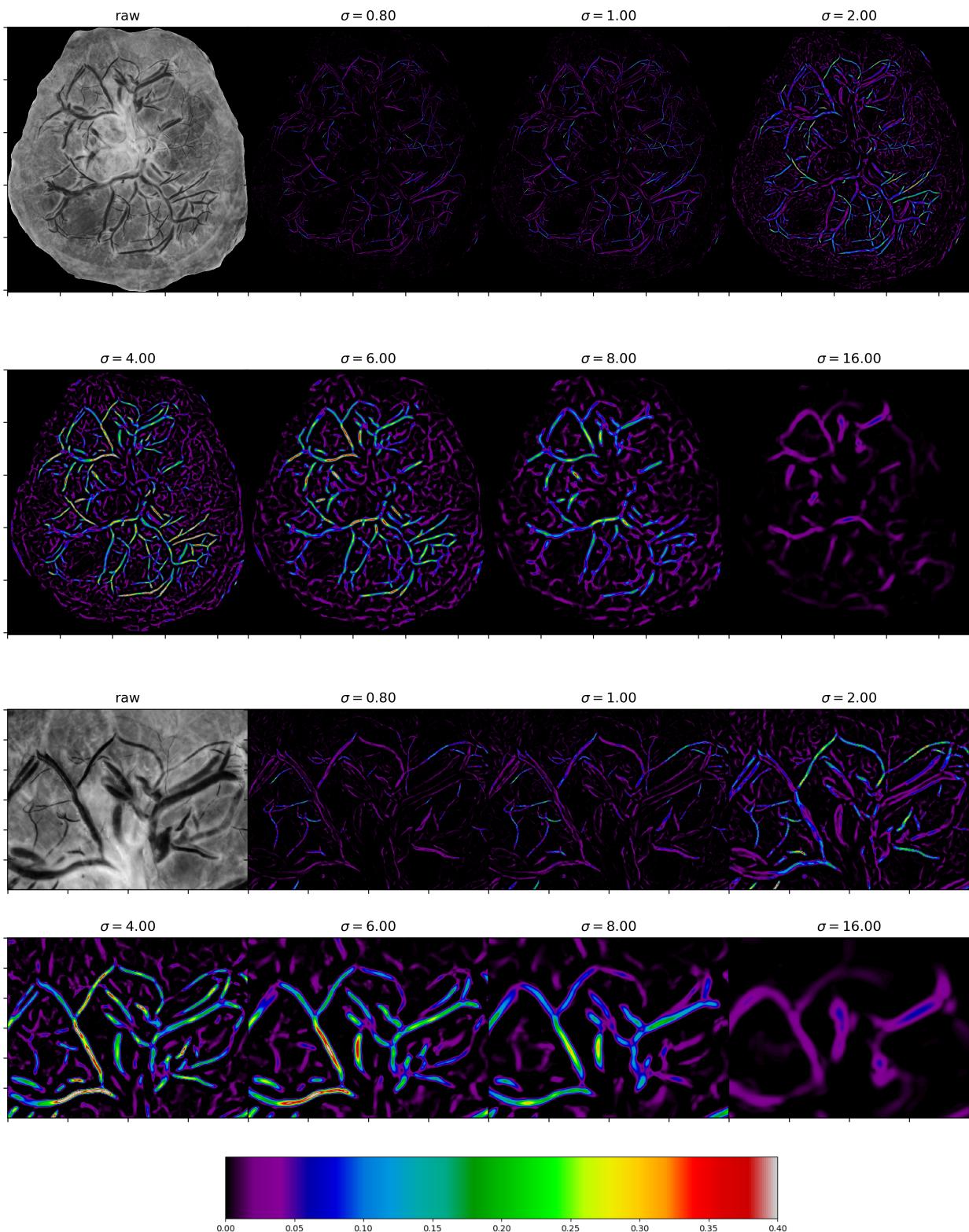


FIGURE 11: Signed Frangi output (plate and inset) (Example 1)

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 obviously a vessel doesn't get defined.
3. Too much blood (not enough?? no idea) is left in the vessels. leading to the weird white border around some vessels. you could identify these along with black center and combine them somehow. no idea. Also, holy shit, some of the white vessel “sleeves” ARE identified in the tracing, and some aren't. Find an example of this and whine about it.
4. Umbilical cord insertion point is stupid and obscures a lot. The tracer guesses but

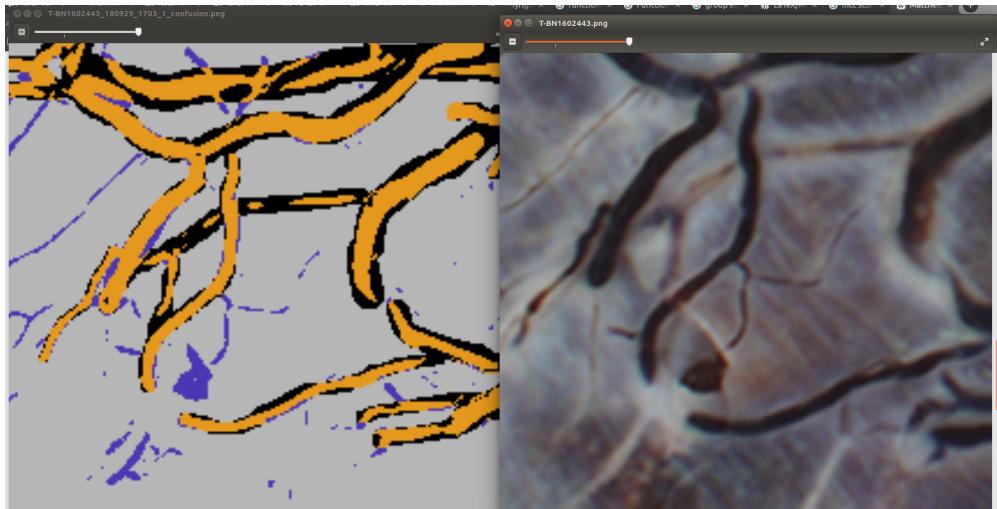


FIGURE 12: "True" false positives and "False" false positives

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

This is a list of particular things I'd like to explore if I have time:

- Relationship between traced pixelwidths vs the scale they were pulled from.
- Frangi behavior at max scale length and if there's anything that gets too large (related to first derivatives maybe?)
- calculate the actual weingarten map eigenvectors (although this is probably gonna be very fake in a discrete sense).
- difference between using green channel and non-green channel.

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 quantitative results too)
- Optimize; Better Use of Scales
- Use of Color Data

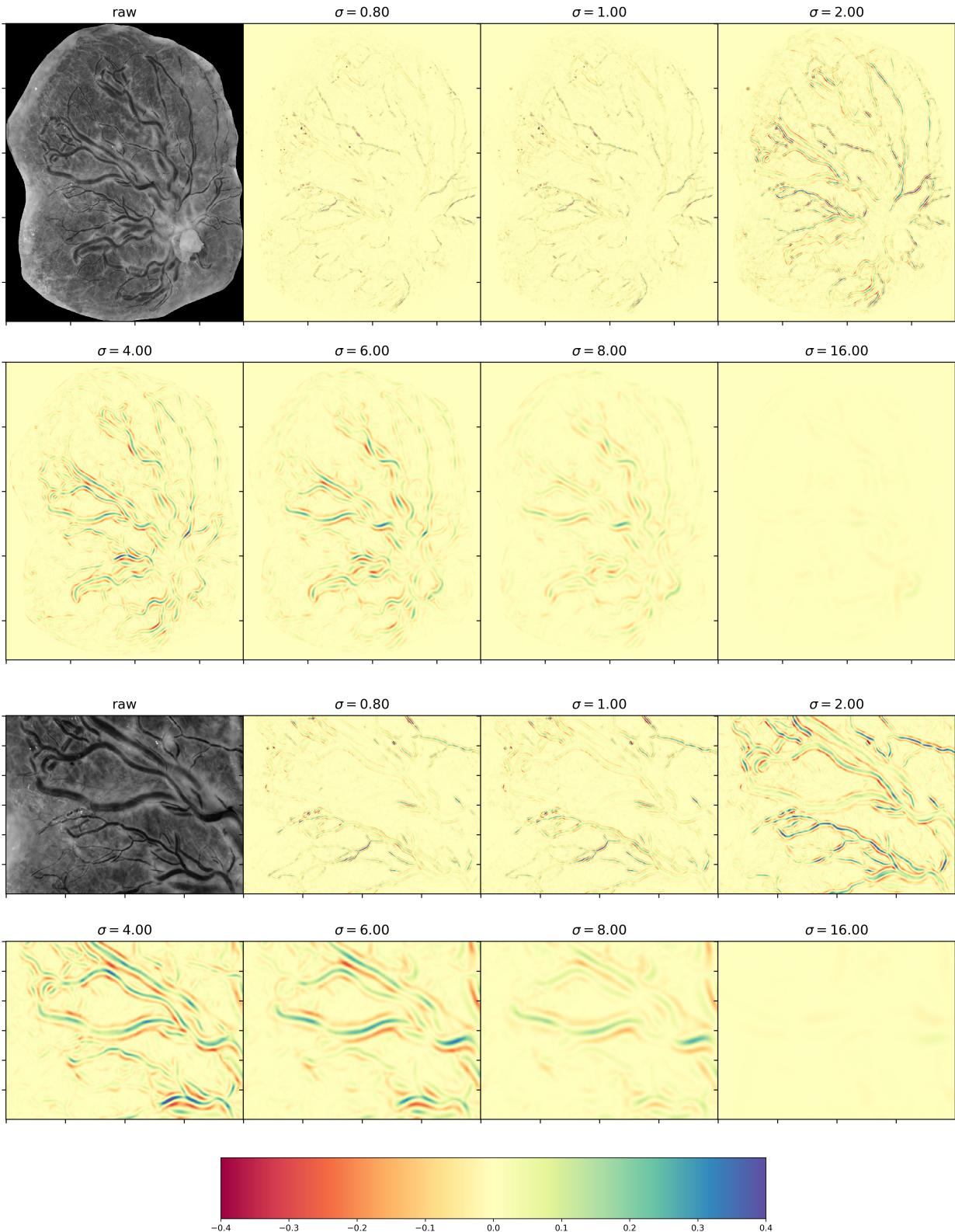


FIGURE 13: Signed Frangi output (plate and inset) (Example 1)

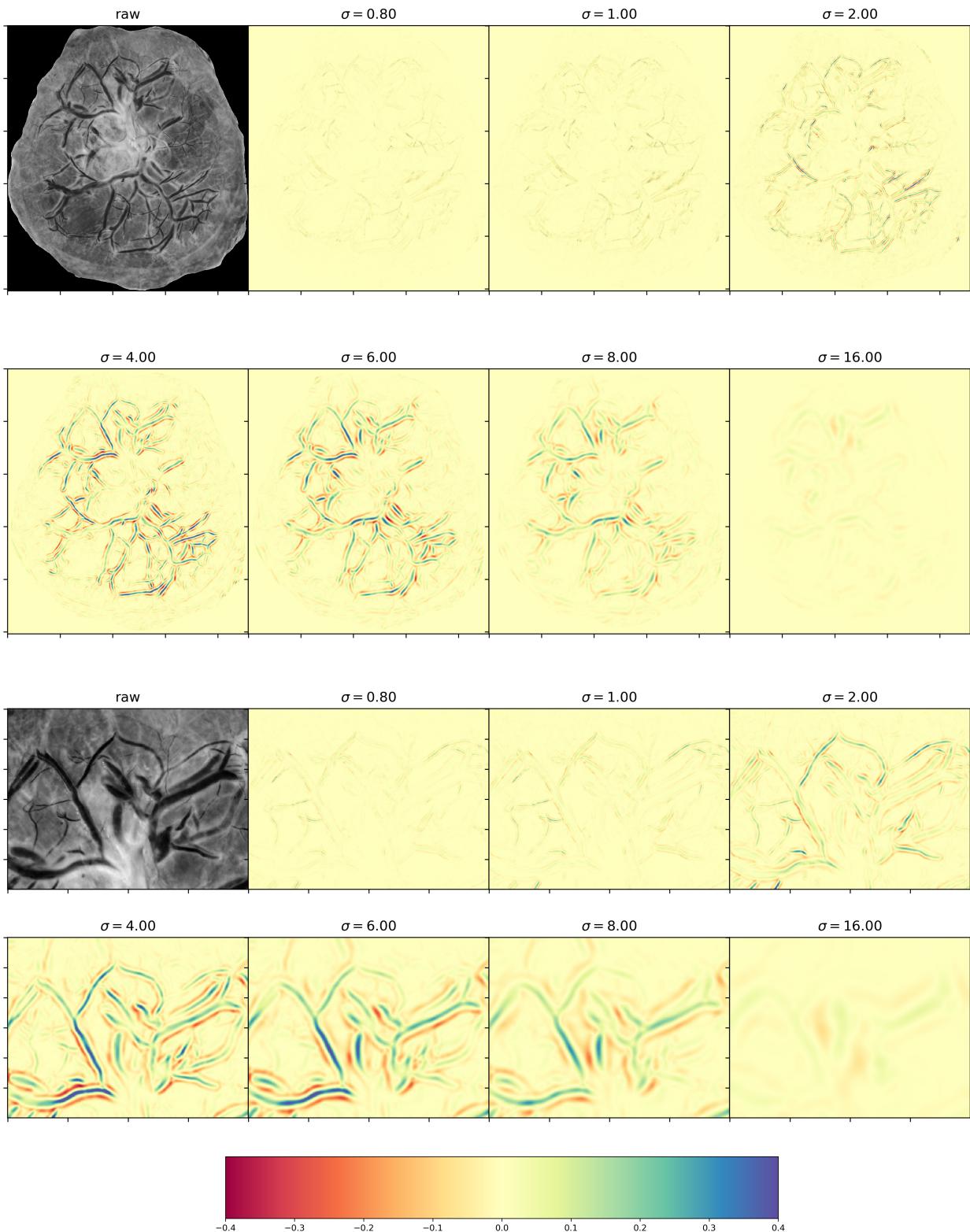


FIGURE 14: Signed Frangi output (plate and inset) (Example 1)

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

```
1 #!/usr/bin/env python3
2
3 from skimage.filters import sobel
4 from frangi import frangi_from_image
5 from plate_morphology import dilate_boundary
6 from skimage.morphology import remove_small_holes, remove_small_objects
7 from merging import nz_percentile
8
9 s = sobel(img)
10 s = dilate_boundary(s, mask=img.mask, radius=20)
11 finv = frangi_from_image(s, sigma=0.8, dark_bg=True)
12
13 finv_thresh = nz_percentile(finv, 80)
14
15 margins = remove_small_objects((finv > ft).filled(0), min_size=32)
16
17 margins_added = remove_small_holes(np.logical_or(margins, approx),
18                                     min_size=100, connectivity=2)
19
20 markers = np.zeros(img.shape, dtype=np.uint8)
21
22 markers[Fmax < .1] = 1
23 markers[margins_added] = 2
24
25 rw = random_walker(img, markers)
26 approx_rw = (rw==2)
27 confusion_rw = confusion(approx_rw, trace, bg_mask=ucip_mask)
28 mccs_rw = mccs(approx_rw, trace, bg_mask=ucip_mask)
29 pnc_rw = np.logical_and(skeltrace, rw2==2).sum() / skeltrace.sum()
```

listings/diffgeo.py

```
1 #!/usr/bin/env python3
2
3
4 import numpy as np
5 import numpy.ma as ma
6
7 from skimage.feature import hessian_matrix, hessian_matrix_eigvals
8 from numpy.linalg import eig
9 from functools import partial
10
11 from hfft import fft_hessian
12
13
14 def principal_curvatures(img, sigma=1.0, H=None):
15     """
16     Return the principal curvatures { 1 , 2 } of an image, that is, the
17     eigenvalues of the Hessian at each point (x,y). The output is arranged such
18     that | 1 | <= | 2 |.
19
20     Input:
21
22         img: An ndarray representing a 2D or multichannel image. If the image
23               is multichannel (e.g. RGB), then each channel will be proccessed
24               individually. Additionally, the input image may be a masked
25               array-- in which case the output will preserve this mask
26               identically.
27
28         PLEASE ADD SOME INFO HERE ABOUT WHAT SORT OF DTYPES ARE
29         EXPECTED/REQUIRED , IF ANY
30
31         sigma: (optional) The scale at which the Hessian is calculated.
32
33         H: (optional) provide sigma (else it will be calculated)
34
35     Output:
36
37         (K1, K2): A tuple where K1, K2 each are the exact dimension of the
38                   input image, ordered in magnitude such that | 1 | <= | 2 |
39                   in all locations. If *signed* option is used, then elements
40                   of K1, K2 may be negative.
41
42     Example:
43
44         >>> K1, K2 = principal_curvatures(img)
45         >>> K1.shape == img.shape
46         True
47         >>> (K1 <= K2).all()
48         True
49
50         >> K1.mask == img.mask
51         True
52         """
53
54     # determine if multichannel
55     multichannel = (img.ndim == 3)
56
57     if not multichannel:
58         # add a trivial dimension
59         img = img[:, :, np.newaxis]
60
61     K1 = np.zeros_like(img, dtype='float64')
62     K2 = np.zeros_like(img, dtype='float64')
```

```

63     for ic in range(img.shape[2]):
64         channel = img[:, :, ic]
65
66         # returns the tuple (Hxx, Hxy, Hyy)
67         if H is None:
68             H = hessian_matrix(channel, sigma=sigma)
69
70         # returns tuple (l1,l2) where l1 >= l2 but this *includes sign*
71         L = hessian_matrix_eigvals(*H)
72         L = np.dstack(L)
73
74         mag = np.argsort(abs(L), axis=-1)
75
76         # just some slice nonsense
77         ix = np.ogrid[0:L.shape[0], 0:L.shape[1], 0:L.shape[2]]
78
79         L = L[ix[0], ix[1], mag]
80
81         # now k2 is larger in absolute value, as consistent with Frangi paper
82
83         K1[:, :, ic] = L[:, :, 0]
84         K2[:, :, ic] = L[:, :, 1]
85
86     try:
87         mask = img.mask
88     except AttributeError:
89         pass
90     else:
91         K1 = ma.masked_array(K1, mask=mask)
92         K2 = ma.masked_array(K2, mask=mask)
93
94     # now undo the trivial dimension
95     if not multichannel:
96         K1 = np.squeeze(K1)
97         K2 = np.squeeze(K2)
98
99     return K1, K2
100
101 def reorder_eigs(L1, L2):
102     """
103     L1, L2 contain a 2D matrix of eigenvalues at each point
104     so that L1 <= L2 at each element.
105
106     this reorders this so that |L1| <= |L2| instead.
107
108     this could (if desired) also return the permutation array
109     but does not do so presently
110     """
111
112     L = np.dstack((L1, L2))
113     mag = np.argsort(abs(L), axis=-1)
114
115     ix = np.ogrid[0:L.shape[0], 0:L.shape[1], 0:L.shape[2]]
116
117     L = L[ix[0], ix[1], mag]
118
119     # now L2 is larger in absolute value, as consistent with Frangi paper
120
121     return (L[:, :, 0], L[:, :, 1])
122
123 def principal_directions(img, sigma, H=None, mask=None):
124     """
125     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, Hyx = 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, Hyx])):
160
161     # grab the (x,y) coordinate of the hxx, hxy, hyx you're using
162     subs = np.unravel_index(i, dims)
163
164     # ignore masked areas (if masked array)
165     if masked and mask[sub]:
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[sub] = np.arccos(v[0,0]) # first component of each
178     leading_thetas[sub] = 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 if __name__ == "__main__":
190

```

```

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

```

```

255     # extend mask over nontargets items
256     # img1 = ma.masked_where( T < T.mean(), img)
257     # img2 = ma.masked_where( t < t.mean(), img)
258
259     # tic = time.time()
260     # print('calculating principal directions for ={}'.format(sigma))
261     # T1,T2 = principal_directions(img1, sigma=sigma, H=H)
262     # toc = time.time()
263     # print('time elapsed: ', toc - tic)
264     # tic = time.time()
265
266     # print('calculating principal directions for ={} (fast)'.format(sigma))
267     # t1,t2 = principal_directions(img2, sigma=sigma, H=h)
268     # toc = time.time()
269     # print('time elapsed: ', toc - tic)

```

listings/extract_NCS_pcsvn.py

```

1 #!/usr/bin/env python3
2
3 """
4 This is the main program. It approximates the PCSVN
5
6 """
7
8
9 from placenta import (get_named_placenta, cropped_args, cropped_view,
10                      list_placentas, list_by_quality, open_typefile,
11                      open_tracefile, add_uci_to_mask, measure_ncs_markings)
12
13 from merging import nz_percentile, apply_threshold
14 from scoring import (compare_trace, rgb_to_widths, merge_widths_from_traces,
15                      filter_widths, mcc, confusion, skeletonize_trace)
16
17 from pcsvn import extract_pcsvn, scale_label_figure, get_outname_lambda
18 from preprocessing import inpaint_hybrid
19
20 import numpy as np
21 import numpy.ma as ma
22
23 import matplotlib.pyplot as plt
24
25 import os.path
26 import os
27 import json
28 import datetime
29
30 # for some post_processing, this needs to be moved elsewhere
31 from skimage.filters import sobel
32 from frangi import frangi_from_image
33 from plate_morphology import dilate_boundary
34 from skimage.morphology import remove_small_holes, remove_small_objects
35 from skimage.segmentation import random_walker
36
37
38
39 # INITIALIZE SAMPLES -----
40
41 # initialize a list of samples (several different ways)
42 #placentas = list_by_quality(0)
43 placentas = list_placentas('T-BN') # load alllllll placentas
44 #placentas = list_by_quality(json_file='manual_batch.json')

```

```

45 #placentas = ['T-BN0204423.png'] # for a single sample, use a 1 element list.
46
47 n_samples = len(placentas)
48
49 # RUNTIME OPTIONS -----
50
51 MAKE_NPZ_FILES = True # pickle frangi targets if you can
52 USE_NPZ_FILES = True # use old npz files if you can
53 NPZ_DIR = 'output/181117-deglared' # where to look for npz files
54 OUTPUT_DIR = 'output/181117-deglared' # where to save outputs
55
56
57 # EXTRACT_PCSVN OPTIONS -----
58
59 # find bright curvilinear structure against a dark background -> True
60 # find dark curvilinear structure against a bright background -> False
61 # DARK_BG -> ignore and return signed Frangi scores
62 DARK_BG = False
63
64 # along with the above, this will return "opposite" signed frangi scores.
65 # if this is True, then DARK_BG controls the "polarity" of the filter.
66 # See frangi.get_frangi_targets for details.
67 SIGNED_FRANGI = False
68
69
70 # do not calculate hessian scores close to the boundary (this is important
71 # mainly in terms of ensuring that the hessian is very large on the edge of
72 # the plate (which would influence gamma calculation)
73 DILATE_PER_SCALE = True
74
75 # use preprocessing.inpaint_with_boundary_median() to replace high
76 # glare regions
77 REMOVE_GLARE = True
78
79 log_range = (-3, 6)
80 n_scales = 40
81
82 # when showing "large scales only", this is where to start
83 # (some index between 0 and n_scales)
84 LO_offset = 24
85
86
87 scales = np.logspace(log_range[0], log_range[1], num=n_scales, base=2)
88 alphas = [0.15 for s in scales]
89 betas = None # will be given default parameters
90 gammas = None # will be given default parameters
91
92 mccs = dict() # empty dict to store MCC's of each sample
93 pnccs = dict() # empty dict to store percent network covered for each sample
94
95 if not os.path.exists(OUTPUT_DIR):
96     os.makedirs(OUTPUT_DIR)
97
98 print(n_samples, "samples total!")
99
100 for i, filename in enumerate(placentas):
101
102     print('*'*80)
103     print(f'extracting PCSVN of {filename}\t ({i} of {n_samples})')
104
105     if USE_NPZ_FILES:
106         # find the first npz file with the sample name in it in the
107         # specified directory.
108         stub = filename.rstrip('.png')

```

```

109     for f in os.scandir(NPZ_DIR):
110         if f.name.endswith('npz') and f.name.startswith(stub):
111             npz_filename = os.path.join(NPZ_DIR, f.name)
112             print(f'using the npz file {npz_filename}')
113             break # just use the first one.
114     else:
115         print(f'no npz file found for {filename}.')
116         npz_filename = None
117     else:
118         npz_filename = None
119
120     # set a lambda function to make output file names
121     outname = get_outname_lambda(filename, output_dir=OUTPUT_DIR)
122
123     raw_img = get_named_placenta(filename, maskfile=None)
124     if npz_filename is not None:
125         F = np.load(npz_filename)[‘F’]
126         if REMOVE_GLARE:
127             img = inpaint_hybrid(raw_img)
128         else:
129             img = raw_img
130         print('successfully loaded the frangi targets!')
131     else:
132         print('finding multiscale frangi targets')
133
134     F, img = extract_pcsvn(filename, DARK_BG=DARK_BG, alphas=alphas,
135                             betas=betas, scales=scales, gammas=gammas,
136                             kernel='discrete', dilate_per_scale=True,
137                             verbose=False, signed_frangi=SIGNED_FRANGI,
138                             generate_json=True, output_dir=OUTPUT_DIR,
139                             remove_glare=REMOVE_GLARE)
140
141     if MAKE_NPZ_FILES:
142         npzfile = ‘.’.join((outname("F").rsplit(‘.’, maxsplit=1)[0], ‘npz’))
143         print("saving frangi targets to ", npzfile)
144         np.savez_compressed(npzfile, F=F)
145
146     crop = cropped_args(img) # these indices crop out the mask significantly
147
148     print("...making outputs")
149
150     # get the 99% frangi filter score
151     print("rewriting alphas with 1% scores")
152
153     p_alphas = [nz_percentile(F[:, :, k], 95.0) for k in range(n_scales)]
154     alphas = np.array(p_alphas)
155
156     approx, labs = apply_threshold(F, alphas, return_labels=True)
157
158     # get the main tracefile
159     trace = open_tracefile(filename, as_binary=True)
160
161     ucip_midpoint, resolution = measure_ncs_markings(filename=filename)
162
163     Fmax = F.max(axis=-1)
164
165     # print(f"The umbilical cord insertion point is at {ucip_midpoint}")
166     # print(f"The resolution of the image is {resolution} pixels per cm.")
167
168     # mask anywhere close to (within 90px L2 distance of) the UCIP
169     # this is empirically how the it is, although could be bigger
170     ucip_mask = add_ucip_to_mask(ucip_midpoint, radius=90, mask=img.mask)
171
172

```

```

173 # open up RGB tracefiles (keep for visualizing?)
174 A_trace = open_typefile(filename, 'arteries')
175 V_trace = open_typefile(filename, 'veins')
176
177 skeltrace = skeletonize_trace(A_trace, V_trace)
178
179 # matrix of widths of traced image
180 widths = merge_widths_from_traces(A_trace, V_trace, strategy='arteries')
181 # min_widths = merge_widths_from_traces(A_trace, V_trace,
182 #                                         strategy='minimum')
183
184 # trace ignoring largest vessels (19 pixels wide)
185 # trace_smaller_only = filter_widths(min_widths, min_width=3, max_width=17)
186 # trace_smaller_only != 0
187 # use limited scales
188 approx_L0, labs_L0 = apply_threshold(F[:, :, L0_offset:], alphas[L0_offset:])
189
190 # fix labels to incorporate offset
191 labs_L0 = (labs_L0 != 0)*(labs_L0 + L0_offset)
192
193 # confusion matrix against default trace
194 confuse = confusion(approx, trace, bg_mask=ucip_mask)
195 confuse_L0 = confusion(approx_L0, trace,
196                         bg_mask=ucip_mask)
197
198 m_score, counts = mcc(approx, trace, ucip_mask, return_counts=True)
199 m_score_L0, counts_L0 = mcc(approx_L0, trace, ucip_mask,
200                             return_counts=True)
201
202 # this all just verifies that the 4 categories were added up
203 # correctly and match the total number of pixels in the reported
204 # placental plate.
205 TP, TN, FP, FN = counts # return these for more analysis?
206
207 total = np.invert(ucip_mask).sum()
208 print('TP: {} \t TN: {} \nFP: {} \t FN: {}'.format(TP, TN, FP, FN))
209 print('TP+TN+FP+FN={} \t total pixels={}'.format(TP+TN+FP+FN, total))
210
211 # MOVE THIS ELSEWHERE
212 s = sobel(img)
213 s = dilate_boundary(s, mask=img.mask, radius=20)
214 finv = frangi_from_image(s, sigma=0.8, dark_bg=True)
215 finv_thresh = nz_percentile(finv, 80)
216 margins = remove_small_objects((finv > finv_thresh).filled(0), min_size=32)
217 margins_added = np.logical_or(margins, approx)
218 margins_added = remove_small_holes(margins_added, min_size=100,
219                                     connectivity=2)
220
221 # random walker markers
222 markers = np.zeros(img.shape, dtype=np.uint8)
223 markers[Fmax < .1] = 1
224 markers[margins_added] = 2
225 rw = random_walker(img, markers, beta=1000)
226 approx_rw = (rw==2)
227 confuse_rw = confusion(approx_rw, trace, bg_mask=ucip_mask)
228 m_score_rw = mcc(approx_rw, trace, ucip_mask)
229 pnc_rw = np.logical_and(skeltrace, approx_rw).sum() / skeltrace.sum()
230
231 mccs[filename] = (m_score, m_score_L0, m_score_rw)
232
233 print(f'mcc score of {m_score:.3} for {filename}')
234 print(f'mcc score of {m_score_L0:.3} with larger sigmas only')
235 print(f'mcc score of {m_score_rw:.3} after random walker')
236
plt.imsave(outname('0_raw'), raw_img[crop].filled(0), cmap=plt.cm.gray)

```

```

237 plt.imsave(outname('1_img'), img[crop].filled(0), cmap=plt.cm.gray)
238 plt.imsave(outname('4_confusion'), confuse[crop])
239 plt.imsave(outname('7_confusion_L0'), confuse_L0[crop])
240 plt.imsave(outname('9_confusion_rw'), confuse_rw[crop])
241
242 percent_covered = np.logical_and(skeltrace, approx).sum() / skeltrace.sum()
243 percent_covered_L0 = np.logical_and(skeltrace, approx_L0).sum() / skeltrace.sum()
244
245 pncs[filename] = (percent_covered, percent_covered_L0, pnc_rw)
246
247 print('percentage of skeltrace covered:', f'{percent_covered:.2%}')
248 print('percentage of skeltrace covered (larger sigmas only):',
249       f'{percent_covered_L0:.2%}')
250 print('percentage of skeltrace covered (random_walker):',
251       f'{pnc_rw:.2%}')
252 plt.imsave(outname('5_coverage'), confusion(approx, skeltrace)[crop])
253 plt.imsave(outname('8_coverage_L0'), confusion(approx_L0, skeltrace)[crop])
254 plt.imsave(outname('9_coverage_rw'), confusion(approx_rw, skeltrace)[crop])
255
256 # only save the colorbar the first time
257 save_colorbar = (i==0)
258 # make the graph that shows what scale the max was pulled from
259 scale_label_figure(labs, scales, crop=crop,
260                     savefilename=outname('3_labeled'), image_only=True,
261                     save_colorbar_separate=save_colorbar,
262                     output_dir=OUTPUT_DIR)
263
264 scale_label_figure(labs_L0, scales, crop=crop,
265                     savefilename=outname('4_labeled'), image_only=True,
266                     save_colorbar_separate=False, output_dir=OUTPUT_DIR)
267 # save the maximum frangi output
268 plt.imsave(outname('2_fmax'), F.max(axis=-1)[crop],
269             vmin=0, vmax=1.0, cmap=plt.cm.nipy_spectral)
270 plt.close('all') # something's leaking :(
271
272 # json file with mccs and other runtime info
273 timestamp = datetime.datetime.now()
274 timestamp = timestamp.strftime("%y%m%d_%H%M")
275
276
277 mccfile = os.path.join(OUTPUT_DIR, f"runlog_{timestamp}.json")
278
279 runlog = {
280     'time': timestamp,
281     'dark_bg': DARK_BG,
282     'dilate_per_scale': DILATE_PER_SCALE,
283     'log_range': log_range,
284     'n_scales': n_scales,
285     'scales': list(scales),
286     'alphas': list(alphas),
287     'betas': None,
288     'use_npz_files': False,
289     'remove_glare': REMOVE_GLARE,
290     'files': list(placentas),
291     'MCCS': mccs,
292     'PNC': pncs
293 }
294
295 with open(mccfile, 'w') as f:
296     json.dump(runlog, f, indent=True)

```

listings/frangi.py

```

1 import numpy as np
2 import numpy.ma
3 from hfft import fft_hessian
4 from diffgeo import principal_curvatures
5 from plate_morphology import dilate_boundary
6
7
8 def frangi_from_image(img, sigma, beta=0.5, gamma=None, dark_bg=True,
9                      dilation_radius=None, signed_frangi=False,
10                     return_debug_info=False):
11     """
12     Perform a frangi filtering on img
13     if None, gamma returns half of Frobenius norm on the image
14     if dilation radius is specified, that amount is dilated from the
15     boundary of the image (mask must be specified)
16
17     input image *must* be a masked array. To implement: supply mask
18     or create a dummy mask if not specified so this can work out of the
19     box on arbitrary images.
20
21     return_debug info will return anisotropy, structureness measures, as
22     well as the calculated gamma. will return a tuple of
23     (R, S, gamma) where R and S are matrices of shape img.shape
24     and gamma is a float.
25
26
27 BIGGER TODO:
28
29     THIS OVERLAPS WITH pcsvn.make_multiscale
30     USE THIS THERE
31
32     """
33
34     # principal_directions() calculates the frangi filter with
35     # standard convolution and takes forever. FIX THIS!
36     hesh = fft_hessian(img, sigma) # the triple (Hxx,Hxy,Hyy)
37
38     k1, k2 = principal_curvatures(img, sigma, H=hesh)
39
40     if dilation_radius is not None:
41
42         # pass None to just get the mask back
43         collar = dilate_boundary(None, radius=dilation_radius,
44                               mask=img.mask)
45
46         # get rid of "bad" K values before you calculate gamma
47         k1[collar] = 0
48         k2[collar] = 0
49
50     # set default gamma value if not supplies
51     if gamma is None:
52         gamma = .5 * max_hessian_norm(hesh)
53         if np.isclose(gamma, 0):
54             print("WARNING: gamma is close to 0. should skip this layer.")
55
56     targets = get_frangi_targets(k1, k2, beta=beta, gamma=gamma,
57                                 dark_bg=dark_bg, signed=signed_frangi)
58
59     if not return_debug_info:
60         return targets
61     else:
62         return targets, (R, S, gamma)
63
64 def get_frangi_targets(K1, K2, beta=0.5, gamma=None, dark_bg=True,

```

```

64                     signed=False):
65     """Calculate the Frangi vesselness measure from eigenvalues.
66
67     Parameters
68     -----
69         K1, K2 : ndarray (each)
70             each is an ndarray of eigenvalues (approximated principal
71             curvatures) for some image.
72         beta: float
73             the anisotropy parameter (default is 0.5)
74         gamma: float or None
75             the structureness parameter. if gamma is None (default), use
76             half of L2 norm of hessian (calculated from K2). if you want to
77             use half of frobenius norm, calculate it outside here.
78         dark_bg: boolean or None
79             if True, then frangi will select only for bright curvilinear
80             features; if False, then Frangi will select only for dark
81             curvilinear structures. if None instead of a bool, then curvilinear
82             structures of either type will be reported.
83         signed: boolean
84             if signed is True, the result will be the same as if dark_bg is set
85             to None, except that the sign will change to match the desired
86             features. See example below.
87
88     Returns
89     -----
90         F: ndarray, same shape as K1
91             the Frangi vesselness measure.
92
93     Examples
94     -----
95     >>>f1 = get_frangi_targets(K1,K2, dark_bg=True, signed=True)
96     >>>f2 = get_frangi_targets(K1,K2, dark_bg=False, signed=True)
97     >>>f1 == -f2
98     True
99
100    """
101    R = anisotropy(K1,K2)
102    S = structureness(K1,K2)
103
104    if gamma is None:
105        # half of max hessian norm (using L2 norm)
106        gamma = .5 * np.abs(K2).max()
107        if np.isclose(gamma, 0):
108            print("warning! gamma is very close to zero."
109                  "maybe this layer isn't worth it...")
110            print("sigma={:.3f}, gamma={}".format(sigma, gamma))
111            print("returning an empty array")
112            return np.zeros_like(img)
113
114    F = np.exp(-R / (2*beta**2))
115    F *= 1 - np.exp( -S / (2*gamma**2))
116
117    # now just filter/ change sign as appropriate.
118    if not signed:
119        # calculate the regular frangi filter
120        if dark_bg is None:
121            #keep F the way it is
122            pass
123        elif dark_bg:
124            # zero responses from positive curvatures
125            F = (K2 < 0)*F
126        else:
127            # zero responses from negative curvatures
128            F = (K2 > 0)*F

```

```

128     else:
129         if dark_bg is None:
130             # output is already signed
131             pass
132         elif dark_bg:
133             # positive curvature spots will be made negative
134             F[K2 > 0] = -1 * F[K2 > 0]
135         else:
136             # negative curvature spots will be made positive
137             F[K2 < 0] = -1 * F[K2 < 0]
138
139     # reapply the mask if the inputs came with one
140     if numpy.ma.is_masked(K1):
141         F = numpy.ma.masked_array(F, mask=K1.mask)
142
143     return F
144
145 def max_hessian_norm(hesh):
146     """Calculate max norm of Hessian.
147     calculates the maximal value (over all pixels of the image) of the
148     Frobenius norm of the Hessian.
149
150     Parameters
151     -----
152     hesh: a tuple of ndarrays
153         The tuple hxx,hxy,hyy which are all the same shape. The hessian at
154         the point (m,n) is then [[hxx[m,n], hxy[m,n]],
155                               [hxy[m,n], hyy[m,n]]]
156
157     Returns
158     -----
159
160     """
161     hxx, hxy, hyy = hesh
162
163     # frob norm is just sqrt(trace(AA^T)) which is easy for a 2x2
164     max_norm = np.sqrt((hxx**2 + 2*hxy**2 + hyy**2).max())
165
166     return max_norm
167
168 def anisotropy(K1,K2):
169     """
170     according to Frangi (1998) this is technically A**2
171     """
172
173     return (K1/K2)**2
174
175 def structureness(K1,K2):
176     """
177     according to Frangi (1998) this is technically S**2
178     """
179     return K1**2 + K2**2

```

listings/hfft_accuracy.py

```

1 """
2
3 here you want to show the accuracy of hfft.py
4
5 BOILERPLATE
6
7 show that gaussian blur of hfft is accurate, except potentially around the
8 boundary proportional to sigma.

```

```

9
10 or if they're off by a scaling factor, show that the derivates
11 (taken the same way) are proportional.
12
13 pseudocode
14
15 A = gaussian_blur(image, sigma, method='conventional')
16 B = gaussian_blue(image, sigma, method='fourier')
17
18 zero_order_accurate = isclose(A, B, tol)
19
20 J_A= get_jacobian(A)
21 J_B = get_jacobian(B)
22
23 first_order_accurate = isclose(J_A, J_B, tol)
24
25 A_eroded = zero_around_plate(A, sigma)
26 B_eroded = zero_around_plate(B, sigma)
27
28 J_A_eroded = zero_around_plate(A, sigma)
29 J_B_eroded = zero_around_plate(B, sigma)
30
31 zero_order_accurate_no_boundary = isclose(A_eroded, B_eroded, tol)
32 first_order_accurate = isclose(J_A_eroded, J_B_eroded, tol)
33
34 """
35
36 from placenta import get_named_placenta
37
38 from hfft import fft_hessian, fft_gaussian
39 from scipy.ndimage import gaussian_filter
40 import matplotlib.pyplot as plt
41 from placenta import mimshow
42
43 from scoring import mean_squared_error
44 import numpy as np
45 from scipy.ndimage import laplace
46 import numpy.ma as ma
47
48 from skimage.segmentation import find_boundaries
49 from skimage.morphology import disk, binary_dilation
50
51 from diffgeo import principal_curvatures
52 from frangi import structureness, anisotropy, get_frangi_targets
53
54 def erode_plate(img, sigma, mask=None):
55     """
56         Apply an eroded mask to an image
57         assume (if helpful) that the boundary of the placenta is a connected loop
58         that is, there is a single inside and outside of the shape, and that
59         the placenta is more or less convex
60
61         alternatively, if img is None, simply erode the mask
62         this function should probably be renamed "dilate mask"
63         and erode plate should be one that just acts on masked inputs
64     """
65
66     if mask is None:
67         mask = img.mask
68
69     # get a boolean array that is 1 along the border of the mask, zero elsewhere
70     # default mode is 'thick' which is fine
71     bounds = find_boundaries(mask)
72

```

```

73     # structure element to dilate by is a disk of diameter sigma
74     # rounded up to the nearest integer. this may be too conservative.
75     selem = disk(np.ceil(sigma))
76     dilated_border = binary_dilation(bounds, selem=selem)
77
78     new_mask = np.logical_or(mask, dilated_border)
79
80     # see comment in docstring. alternatively, the behavior here
81     # could be handled by an "apply mask" parameter
82     if img is None:
83         return new_mask
84     else:
85         return ma.masked_array(img, mask=new_mask)
86
87
88 # FIX SOME ISSUES, BINARY DILATION IS TAKING HELLA LONG AND ALSO
89 # THERE ARE RANDOM BLIPS INSIDE THE MASK!!!
90 # FIX IN GIMP!:
91
92 imgfile = 'barium1.png'
93 maskfile = 'barium1.mask.png'
94
95 img_raw = get_named_placenta(imgfile, maskfile=maskfile)
96
97 # so that scipy.ndimage.gaussian_filter doesn't use uint8 precision (jesus)
98 img = img_raw / 255.
99
100 # convenience function to show a matrix with img.mask mask
101 ms = lambda x: mimshow(ma.masked_array(x, img.mask))
102
103 sigma = 5
104
105 print('applying standard gauss blur')
106 # THIS USES THE SAME DTYPE AS THE INPUT SO DEAR LORD MAKE SURE IT'S A FLOAT
107 A = gaussian_filter(img.astype('f'), sigma, mode='constant') #zero padding
108 print('applying fft gauss blur')
109 B = fft_gaussian(img, sigma)
110 B_unnormalized = B.copy()
111 B = B / (2*(sigma**2)*np.pi)
112
113 #A = erode_plate(A, sigma, mask=img.mask)
114 #B = erode_plate(B, sigma, mask=img.mask)
115 print('calculating first derivatives')
116
117 # zero the masks before calculating derivates if they're masked
118 Ax, Ay = np.gradient(A)
119 Bx, By = np.gradient(B)
120
121 print('calculating second derivatives')
122
123 # you can verify np.isclose(Axy,Ayx) && np.isclose(Bxy,Byx) -> True
124 Axx, Axy = np.gradient(Ax)
125 Ayy, Axy = np.gradient(Ay)
126
127 Bxx, Bxy = np.gradient(Bx)
128 Bxy, Byy = np.gradient(By)
129
130
131
132 print('calculating eigenvalues of hessian')
133 ak1, ak2 = principal_curvatures(A, sigma=sigma, H=(Axx,Axy,Ayy))
134 bk1, bk2 = principal_curvatures(B, sigma=sigma, H=(Bxx,Bxy,Byy))
135
136

```

```

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 #Bscalled = (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

```

```

14 TODO: PROVIDE MAIN USAGE NOTES
15 """
16
17 def gauss_freq(shape,    =1.):
18     """
19     DEPRECATED
20
21     NOTE:
22         this function is/should be? for illustrative purposes only--
23         we can actually build this much faster using the builtin
24         scipy.signal.gaussian rather than a roll-your-own
25
26     build a shape=(M,N) sized gaussian kernel in frequency space
27     with size
28
29     (due to the convolution theorem for fourier transforms, the function
30     created here may simply be *multiplied* against the signal.
31
32     """
33
34     M, N = shape
35     fgauss = np.fromfunction(lambda , : (( +M+1)/2)**2 + (( +N+1)/2)**2, shape=shape)
36
37     # is this used?
38     coeff = (1 / (2*np.pi * **2))
39
40     return np.exp(-fgauss / (2* **2))
41
42 def blur(img, sigma):
43     """
44     DEPRECATED
45     a roll-your-own FFT-implemented gaussian blur.
46     fft_gaussian below is preferred (it is more efficient)
47
48
49     I = fftpack.fft2(img) # get 2D transform of the image
50
51     # do whatever
52
53     I *= gauss_freq(I.shape, sigma)
54
55
56     return fftpack.ifft2(I).real
57
58 def fft_gaussian(img,sigma,A=None):
59     """
60
61     https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.fftconvolve.html
62
63     in particular the example in which a gaussian blur is implemented.
64
65     along with the comment:
66     "Gaussian blur implemented using FFT convolution. Notice the dark borders
67     around the image, due to the zero-padding beyond its boundaries. The
68     convolve2d function allows for other types of image boundaries, but is far
69     slower"
70
71     (i.e. doesn't use FFT).
72
73     note that here, you actually take the FFT of a gaussian (rather than
74     build it in frequency space). there are ~6 ways to do this.
75
76     #create a 2D gaussian kernel to take the FFT of
77

```

```

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     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

```

listings/merging.py

```

1 #!/usr/bin/env python3
2
3 import numpy as np
4 import numpy.ma as ma
5
6
7 def nz_percentile(A, q, axis=None, interpolation='linear'):
8     """calculate np.percentile(...,q) on an array's nonzero elements only
9
10    Parameters
11    -----
12    A : ndarray
13        matrix from which percentiles will be calculated. Percentiles
14        are calculated on an elementwise basis, so the shape is not important
15    q : a float
16        Percentile to compute, between 0 and 100.0 (inclusive).
17
18    (other arguments): see numpy.percentile docstring
19    ...
20
21    Returns
22    -----
23    out: float
24
25    """
26
27    if ma.is_masked(A):
28        A = A.filled(0)
29
30    return np.percentile(A[A > 0], q, axis=axis, interpolation=interpolation)
31
32
33 def apply_threshold(targets, alphas, return_labels=True):
34     """Threshold targets at each scale, then return max target over all scales.
35
36     A unique alpha can be given for each scale (see below). Return a 2D boolean
37     array, and optionally another array representing what at what scale the max
38     filter response occurred.
39
40    Parameters
41    -----
42    targets : ndarray
43        a 3D array, where targets[:, :, k] is the result of the Frangi filter
44        at the kth scale.
45    alphas : float or array_like
46        a list / 1d array of length targets.shape[-1]. each alphas[k] is a
47        float which thresholds the Frangi response at the kth scale. Due to
48        broadcasting, this can also be a single float, which will be applied
49        to each scale.
50    return_labels : bool, optional
51        If True, return another ndarray representing the scale (see Notes
52        below). Default is True.
53
54    Returns
55    -----
56    out : ndarray, dtype=bool
57        if return_labels is true, this will return both the final
58        threshold and the labels as two separate matrices. This is
59        a convenience, since you could easily find labels with

```

```

60     labels : ndarray, optional, dtype=uint8
61         The scale at which the largest filter response was found after
62         thresholding. Element is 0 if no scale passed the threshold,
63         otherwise an int between 1 and targets.shape[-1] See Notes below.
64
65     Notes / Examples
66     -----
67     Despite the name, this does *NOT* return the thresholded targets itself,
68     but instead the maximum value after thresholding. If you wanted the
69     thresholded filter responses alone, you should simply run
70
71     >>>(targets > alphas)*targets
72
73     The optional output ‘labels‘ is a 2D matrix indicating where the max filter
74     response occurred. For example, if the label is K, the max filter response
75     will occur at targets[:, :, K-1]. In other words,
76
77     >>>passed, labels = apply_threshold(targets, alphas)
78     >>>targets.max(axis=-1) == targets[:, :, labels - 1 ]
79     True
80
81     It should be noted that returning labels is really just for convenience
82     only; you could construct it as shown in the following example:
83
84     >>>manual_labels = (targets.argmax(axis=-1) + 1)*np.invert(passed)
85     >>>labels == manual_labels
86     True
87
88     Similarly, the standard boolean output could just as easily be obtained.
89     >>>passed == (labels != 0)
90     True
91     """
92
93     # threshold as an array (even if it's a single element) to broadcast
94     alphas = np.array(alphas)
95
96     # if input's just a MxN matrix, expand it trivially so it works below
97     if targets.ndim == 2:
98         targets = np.expand_dims(targets, 2)
99
100    # either there's an alpha for each channel or there's a single
101    # alpha to be broadcast across all channels
102    assert (targets.shape[-1] == alphas.size) or (alphas.size == 1)
103
104    # pixels that passed the threshold at any level
105    passed = (targets >= alphas).any(axis=-1)
106
107    if not return_labels:
108        return passed # we're done already
109
110    wheres = targets.argmax(axis=-1) # get label of where maximum occurs
111    wheres += 1 # increment to reserve 0 label for no match
112
113    # then remove anything that didn't pass the threshold
114    wheres[np.invert(passed)] = 0
115
116    assert np.all(passed == (wheres > 0))
117
118    return passed, wheres

```

listings/pcsvn.py

¹ `#!/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):
49         if dilate_per_scale:
50             if sigma < 2.5:
51                 radius = 10
52             elif sigma > 20:
53                 radius = int(sigma*2)
54             else:
55                 radius = int(sigma*4) # a little aggressive
56         else:
57             radius = None
58
59         if VERBOSE:
60             print('  ={}'.format(sigma))
61
62         # get hessian components at each pixel as a triplet (Lxx, Lxy, Lyy)
63         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)
93         hessian_norm[collar] = 0
94         max_hessian_norm = hessian_norm.max()
95         gamma = .5*max_hessian_norm
96
97     if VERBOSE:
98         # compare to other method of calculating gamma
99         gamma_alt = .5 * np.abs(k2).max()
100        print(f"half of k2 max is {gamma_alt}")
101
102    if VERBOSE:
103        print(f"gamma (half of max hessian (frob) norm is {gamma})")
104        print(f'finding Frangi targets with beta={beta} and gamma={gamma:.2f}')
105
106    # calculate frangi targets at this scale
107    targets = get_frangi_targets(k1, k2, beta=beta, gamma=gamma,
108                                 dark_bg=dark_bg, signed=signed_frangi)
109
110
111    # store results as a dictionary
112    this_scale = {'sigma': sigma,
113                  'beta': beta,
114                  'gamma': gamma,
115                  'H': hesh,
116                  'F': targets,
117                  'k1': k1,
118                  'k2': k2,
119                  'border_radius': radius
120                  }
121
122    if find_principal_directions:
123        # principal directions should only be computed for critical regions
124        # ignore anything less than a std deviation over the mean
125        # this mask is where PD's will *NOT* be calculated
126        cutoff = targets.mean() + targets.std()
127        pd_mask = np.bitwise_or(targets < cutoff, img.mask).filled(1)

```

```

129     percent_calculated = (pd_mask.size - pd_mask.sum()) / pd_mask.size
130
131     if VERBOSE:
132         print(f"finding PD's for {percent_calculated:.2%} of image"
133             f"anything above vesselness score {cutoff:.6f}")
134
135     t1, t2 = principal_directions(img, sigma=sigma, H=hesh,
136                                     mask=pd_mask)
137
138     # add them to this scale's output
139     this_scale['t1'] = t1
140     this_scale['t2'] = t2
141 else:
142     if VERBOSE:
143         print('skipping principal direction calculation')
144
145     # store results as a list of dictionaries
146     multiscale.append(this_scale)
147
148 return multiscale
149
150
151 def extract_pcsvn(filename, scales, alphas=None, betas=None, gammas=None,
152                     DARK_BG=True, dilate_per_scale=True, verbose=True,
153                     generate_json=True, output_dir=None, kernel=None,
154                     signed_frangi=False, remove_glare=False):
155     """Run PCSVN extraction on the sample given in the file.
156
157     Despite the name, this simply returns the Frangi filter responses at
158     each provided scale without explicitly making any decisions about what
159     is or is not part of the PCSVN.
160
161     TODO:Finish docstring!
162     """
163
164     raw_img = get_named_placenta(filename, maskfile=None)
165
166     # Multiscale & Frangi Parameters#####
167
168     # set default alphas and betas if undeclared
169     if alphas is None:
170         alphas = [.15 for s in scales] # threshold constant
171     if betas is None:
172         betas = [0.5 for s in scales] # anisotropy constant
173
174     # declare None here to calculate half of hessian's norm
175     if gammas is None:
176         gammas = [None for s in scales] # structureness parameter
177
178     # Preprocessing#####
179     if remove_glare:
180         if verbose:
181             print('removing glare from sample')
182             # img = inpaint_glare(raw_img)
183             # img = inpaint_with_boundary_median(raw_img)
184             img = inpaint_hybrid(raw_img)
185     else:
186         img = raw_img.copy() # in case we alter the mask or something
187
188     # Multiscale Frangi Filter#####
189
190     # output is a dictionary of relevant info at each scale
191     multiscale = make_multiscale(img, scales, betas, gammas,
192                                 find_principal_directions=False,

```

```

193         dilate_per_scale=dilate_per_scale,
194         kernel=kernel,
195         signed_frangi=signed_frangi,
196         dark_bg=DARK_BG,
197         VERBOSE=verbose)
198
199 # extract these for logging
200 gammas = [scale['gamma'] for scale in multiscale]
201 border_radii = [scale['border_radius'] for scale in multiscale]
202
203 #####Process Multiscale Targets#####
204
205 # ignore targets too close to edge of plate
206 # wait are we doing this twice?
207 if dilate_per_scale:
208     if verbose:
209         print('trimming collars of plates (per scale)')
210
211     for i in range(len(multiscale)):
212         f = multiscale[i]['F']
213         # twice the buffer (be conservative!)
214         radius = int(multiscale[i]['sigma']*2)
215         if verbose:
216             print('dilating plate for radius={}'.format(radius))
217         f = dilate_boundary(f, radius=radius, mask=img.mask)
218         # get rid of mask
219         multiscale[i]['F'] = f.filled(0)
220     else:
221         for i in range(len(multiscale)):
222             # get rid of mask
223             multiscale[i]['F'] = multiscale[i]['F'].filled(0)
224
225 #####Make Composite#####
226
227 # get a M x N x n_scales array of Frangi targets at each level
228 F_all = np.dstack([scale['F'] for scale in multiscale])
229
230 if generate_json:
231
232     time_of_run = datetime.datetime.now()
233     timestamp = time_of_run.strftime("%y%m%d_%H%M")
234
235     logdata = {'time': timestamp,
236                'filename': filename,
237                'alphas': list(alphas),
238                'betas': list(betas),
239                'gammas': gammas,
240                'sigmas': list(scales),
241                }
242     if dilate_per_scale:
243         logdata['border_radii'] = border_radii
244
245     if output_dir is None:
246         output_dir = 'output'
247
248     base = os.path.basename(filename)
249     *base, suffix = base.split('.')
250     dumpfile = os.path.join(output_dir,
251                            ''.join(base) + '_' + str(timestamp)
252                            + '.json')
253
254     with open(dumpfile, 'w') as f:
255         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(whereis, 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     whereis = whereis[crop]
319
320 fig, ax = plt.subplots() # not sure about figsize

```

```

321 N = len(scales) # number of scales / labels
322 tabemap = _build_scale_colormap(N, 'viridis_r')
324
325 if image_only:
326     plt.imsave(savefilename, wheres, cmap=tabemap, vmin=0, vmax=N)
327     plt.close()
328 else:
329     imgplot = ax.imshow(wheres, cmap=tabemap, vmin=0, vmax=N)
330     # discrete colorbar
331     cbar = plt.colorbar(imgplot)
332
333     # this is apparently hackish, beats me
334     tick_locs = (np.arange(N+1) + 0.5)*(N-1)/N
335
336     cbar.set_ticks(tick_locs)
337     # label each tick with the sigma value
338     scalelabels = [r"$\sigma = {:.2f}$".format(s) for s in scales]
339     scalelabels.insert(0, "(no match)")
340     # label with their sigma value
341     cbar.set_ticklabels(scalelabels)
342     #ax.set_title(r"Scale ($\sigma$) of maximum vesselness ")
343     plt.tight_layout()
344
345     #plt.savefig(outname('labeled'), dpi=300)
346     if show_only or (savefilename is None):
347         plt.show()
348     else:
349         plt.savefig(savefilename, dpi=300)
350
351     plt.close()
352
353 if save_colorbar_separate:
354     if savecolorbarfile is None:
355         savecolorbarfile = os.path.join(output_dir, "scale_colorbar.png")
356     fig = plt.figure(figsize=(1, 8))
357     ax1 = fig.add_axes([0.05, 0.05, 0.15, 0.9])
358     tick_locs = (np.arange(N+1) + 0.5)*(N-1)/N
359     scalelabels = [r"$\sigma = {:.2f}$".format(s) for s in scales]
360     scalelabels.insert(0, "n/a")
361     cbar = mpl.colorbar.ColorbarBase(ax1, cmap=tabemap,
362                                     norm=mpl.colors.Normalize(vmin=0,
363                                     vmax=N),
364                                     orientation='vertical',
365                                     ticks=tick_locs)
366     cbar.set_ticklabels(scalelabels)
367     plt.savefig(savecolorbarfile, dpi=300)

```

listings/placenta.py

```

1 #!/usr/bin/env python3
2
3 """
4 Get registered, unpreprocessed placental images. No automatic registration
5 (i.e. segmentation of placental plate) takes place here. The background,
6 however, *is* masked.
7
8 Again, there is no support for unregistered placental pictures.
9 A mask file must be provided.
10
11 There is currently no support for color images.
12 """
13

```

```

14 import numpy as np
15 import numpy.ma as ma
16 from skimage import segmentation, morphology
17 import os.path
18 import os
19 import json
20 from scipy.ndimage import imread
21
22 def open_typefile(filename, filetype, sample_dir=None, mode=None):
23     """
24         filetype is either 'mask' or 'trace'
25         mask -> 'L' mode
26         trace -> 'RGB' mode
27         use mode keyword to override this behavior (for example if you
28         want a binary trace)
29
30     typefiles that aren't the above will be treated as 'L'
31     """
32     # try to open what the mask *should* be named
33     # this should be done less hackishly
34     # for example, if filename is 'ncs.1029.jpg' then
35     # this would set the maskfile as 'ncs.1029.mask.jpg'
36
37     #if filetype not in ("mask", "trace"):
38     #    raise NotImplementedError("Can only deal with mask or trace files.")
39
40     # get the base of filename and build the type filename
41     *base, suffix = filename.split('.')
42     base = ''.join(base)
43     typefile = '.'.join((base, filetype, suffix))
44
45     if sample_dir is None:
46         sample_dir = 'samples'
47
48     typefile = os.path.join(sample_dir, typefile)
49
50     if mode is not None:
51         if filetype == 'mask':
52             mode = 'L'
53         elif filetype in ('ctrace', 'veins', 'arteries'):
54             mode = 'RGB'
55         else:
56             # handle this if you need to?
57             mode = 'L'
58     try:
59         img = imread(typefile, mode=mode)
60
61     except FileNotFoundError:
62         print('Could not find file', typefile)
63         raise
64
65     return img
66
67
68 def open_tracefile(base_filename, as_binary=True,
69                     sample_dir=None):
70     """
71
72     ###width parsing is no longer done here. instead, this function
73     should handle the venous/arterial difference.
74
75     this currently only serves to open the RGB traces as binary
76     files instead of RGB, which is processed later
77

```

```

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 raw_img = imread(full_filename, mode='L')
140
141

```

```

142 if maskfile is None:
143     # try to open what the mask *should* be named
144     # this should be done less hackishly
145     # for example, if filename is 'ncs.1029.jpg' then
146     # this would set the maskfile as 'ncs.1029.mask.jpg'
147     base, suffix = filename.split('.')
148     test_maskfile = ''.join(base) + '.mask.' + suffix
149     test_maskfile = os.path.join(sample_dir, test_maskfile)
150     try:
151         mask = imread(test_maskfile, mode='L')
152     except FileNotFoundError:
153         print('Could not find maskfile', test_maskfile)
154         print('Please supply a maskfile. Autogeneration of mask',
155             'files is slow and buggy and therefore not supported.')
156         raise
157     #return mask_background(raw_img)
158 else:
159     # set maskfile name relative to path
160     maskfile = os.path.join(sample_dir, maskfile)
161     mask = imread(maskfile, mode='L')
162
163 return ma.masked_array(raw_img, mask=mask)
164
165
166 def list_by_quality(quality=0, N=None, json_file=None,
167                     return_empty=False):
168     """
169     returns a list of filenames that are of quality ``quality``
170
171     quality is either "good" or 0
172             "OK" or 1
173             "fair" or 2
174             "bad" or 3
175
176     N is the number of placentas to return (will return # of placentas
177     of that quality or N, whichever is smaller)
178
179     if json_name is not None just use that filename directly
180
181     if return_empty then silently failing is OK
182     """
183     if quality == 0:
184         quality = 'good'
185     elif quality == 1:
186         quality = 'okay'
187     elif quality == 2:
188         quality = 'fair'
189     elif quality == 3:
190         quality = 'bad'
191     else:
192         try:
193             quality = quality.lower()
194         except AttributeError:
195             if return_empty:
196                 return list()
197             else:
198                 print(f'unknown quality {quality}')
199                 raise
200
201     # json file
202     if json_file is None:
203         json_file = f'{quality}-mccs.json'
204
205     # 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

```

```

270     if label is None:
271         label = '' # str.startswith('') is always True
272
273     placentas = list()
274
275     for f in os.listdir(sample_dir):
276
277         if f.startswith(label):
278             # oh man they gotta be png files
279             if check_filetype(f) == 'base':
280                 placentas.append(f)
281
282     return sorted(placentas)
283
284 def show_mask(img, interactive=True, mask_color=None):
285     """
286     show a masked grayscale image with a dark blue masked region
287
288     custom version of imshow that shows grayscale images with the right colormap
289     and, if they're masked arrays, sets makes the mask a dark blue
290     a better function might make the grayscale value dark blue
291     (so there's no confusion)
292
293     if interactive, this operates like "plt.imshow"
294     if interactive==False, return the RGB matrix
295     """
296
297     from numpy.ma import is_masked
298     from skimage.color import gray2rgb
299     import matplotlib.pyplot as plt
300
301     if mask_color is None:
302         mask_color = (0,0,60)
303
304     if not is_masked(img):
305         if interactive:
306             plt.imshow(img, cmap=plt.cm.gray)
307         else:
308             # return as an rgb image so output is uniform
309             return gray2rgb(img)
310
311     # otherwise, get an RGB array, black where the mask is
312     mimg = gray2rgb(img.filled(0))
313     # fill masked regions with the mask color
314     mimg[img.mask, :] = mask_color
315
316     if interactive:
317         plt.imshow(mimg)
318     else:
319         return mimg
320
321 def _cropped_bounds(img, mask=None):
322
323     if mask is not None:
324
325         img = ma.masked_array(img, mask=mask)
326
327     X,Y = (np.argwhere(np.invert(img.mask)).any(axis=k)).squeeze() for k in (0,1))
328
329     if X.size == 0:
330         X = [None, None] # these will slice correctly
331     if Y.size == 0:
332         Y = [None, None]
333

```

```

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:

```

```

398     print('the image size is {}x{}'.format(img.shape[0], img.shape[1]))
399
400 rulemarks = np.all(img == CYAN, axis=-1)
401
402 # turn into two pixels (these should each be shape (18,))
403 X,Y = np.where(rulemarks)
404
405 assert X.shape == Y.shape
406
407 # if they followed the protocol correctly...
408 if X.size == 18:
409     # get the two pixels at the center of each box
410     A, B = (X[4], Y[4]), (X[13], Y[13])
411 else:
412     # dots are a nonstandard size for some reason. this works too.
413     thinned = morphology.thin(rulemarks)
414     X, Y = np.where(thinned)
415     assert(thinned.sum() == 2) # there should be just two pixels now.
416     A, B = (X[0], Y[0]), (X[1], Y[1])
417
418 ruler_distance = np.sqrt( (A[0] - B[0])**2 + (A[1] - B[1])**2 )
419 if verbose:
420     print(f'one cm equals {ruler_distance} pixels')
421
422 # the umbilical cord insertion point (UCIP) is a yellow circle, radius 19
423 ucipmarks = np.all(img == YELLOW, axis=-1)
424 X,Y = np.where(ucipmarks)
425
426 # find midpoint of the x & y coordinates
427 assert X.max() - X.min() == Y.max() - Y.min()
428 radius = (X.max() - X.min()) // 2
429
430 mid = (X.min() + radius, Y.min() + radius)
431
432 if verbose:
433     print('the middle of the UCIP location is', mid)
434     print('the radius outward is', radius)
435     print('the total measurable diameter is', radius*2 + 1)
436
437 return mid, ruler_distance
438
439
440 def add_ucip_to_mask(m, radius=100, mask=None, size_like=None):
441     """
442     - m is a tuple (2x1) representing the (coordinate) midpoint of the UCIP
443     - radius around which to dilate the UCIP is the dilation radius as it
444     works in morphology--this is passed directly to skimage.morphology.disk.
445     thus a circle centered at point m with diameter 2*radius + 1
446     - if no mask is supplied, dilate the point in an array of zeros the shape
447     of 'size_like' (would be the same as passing mask=np.zeros_like(size_like))
448
449 Note: this behaves much faster than binary dilation on the point
450 """
451 if mask is None:
452     if size_like is not None:
453         mask = np.zeros_like(size_like)
454     else:
455         raise ValueError("No mask info supplied!")
456
457 # an empty mask (since we need to merge--we don't want to copy the
458 # zeros of the dilated UCIP -- just the ones!)
459 to_add = np.zeros_like(mask)
460
461 # 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 if __name__ == "__main__":
472
473     """test that this works on an easy image."""
474
475     from scipy.ndimage import imread
476     import matplotlib.pyplot as plt
477     test_filename = 'barium1.png'
478     #test_maskfile = 'barium1.mask.png'
479
480     img = get_named_placenta(test_filename, maskfile=None)
481
482     print('showing the mask of', test_filename)
483     print('run plt.show() to see masked output')
484     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:

```

```

36         mask = img.mask
37     except AttributeError:
38         raise('Need to supply mask information')
39
40 perimeter = find_boundaries(mask, mode='inner')
41
42 maskpad = np.zeros_like(perimeter)
43
44 M,N = maskpad.shape
45 for i,j in np.argwhere(perimeter):
46     # just make a cross shape on each of those points
47     # these will silently fail if slice is OOB thus ranges are limited.
48     maskpad[max(i-radius,0):min(i+radius,M),j] = 1
49     maskpad[i,max(j-radius,0):min(j+radius,N)] = 1
50
51 new_mask = np.bitwise_or(maskpad, mask)
52
53 if img is None:
54     return new_mask # return a 2D array
55 else:
56     # replace the original mask or create a new masked array
57     return ma.masked_array(img, mask=new_mask)
58
59 ######
60 # DEMO FOR SHOWING OFF DILATE_BOUNDARY EFFECT
61
62 if __name__ == "__main__":
63
64     from placenta import get_named_placenta
65     from frangi import frangi_from_image
66     import matplotlib.pyplot as plt
67     import numpy as np
68     from skimage.exposure import rescale_intensity
69
70     import os.path
71
72     dest_dir = 'demo_output'
73     img = get_named_placenta('T-BN0164923.png')
74
75     #radius = 30
76     sigma = 3
77     radius = 80
78     # IMG WITHOUT DILATING, THEN IMAGE WITH DILATING
79     #sigma = radius/4
80
81     #inset = np.s_[:, :]
82     inset = np.s_[800:1000, 500:890]
83     #inset = np.s_[100:300, 300:500]
84
85     D = dilate_boundary(img, radius=radius)
86
87
88     # SAVE IT IN THE RIGHT DIRECTORY, ETC plt.savefig(
89
90     # NOW SHOW FRANGI ON THESE IMAGES
91
92     # NOW SHOW THE SAME PICTURE
93
94     Fimg = frangi_from_image(img, sigma, dark_bg=False, dilation_radius=None)
95     #FD = frangi_from_image(img, sigma, dark_bg=False, dilation_radius=radius)
96     FD = frangi_from_image(D, sigma, dark_bg=False)
97
98     #Fimg = rescale_intensity(Fimg)
99     #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))
56
60     # fill the masked areas with the corresponding fill value
61     for lab in range(1, lm.max()+1):
62         inds = np.where(lm == lab)
63         # find locations of filled values corresponding to this label
64         # median in case there's overlapped regions? (sloppy)
65         replace_value = nz_median(fill_vals[inds])
66
66         if replace_value == 0:
67             raise
68
69         fill_vals[inds] = replace_value
70
71     # now fill in the values
72     new_img[glared] = fill_vals[glared]
73
74     return new_img
75
76
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

```

```

108 # inpaint larger regions with biharmonic inpainting
109 large_inpainted = inpaint_biharmonic(img.filled(0), mask=large_glare)
110
111 # now overwrite with these values
112 hybrid[large_glare] = large_inpainted[large_glare]
113
114 # put on old image mask
115 return ma.masked_array(hybrid, mask=img.mask)
116
117 def inpaint_with_biharmonic(img, threshold=175):
118     """
119     use biharmonic inpainting *all* glare
120     """
121     glare = mask_glare(img, threshold=threshold, mask_only=True)
122     inpainted = inpaint_biharmonic(img_as_float(img.filled(0)), mask=glare)
123
124     if ma.is_masked(img):
125         return ma.masked_array(inpainted, mask=img.mask)
126     else:
127         return inpainted
128
129 def mask_glare(img, threshold=175, mask_only=False):
130     """
131     for demoing purposes, with placenta.show_mask
132
133     if mask_only, just return the mask. Otherwise return a copy of img with
134     that added to the mask. If you want the original mask to be ignored,
135     just pass img.filled(0) ya doofus
136
137     threshold is expected to be of the same dtype as img *unless# it assumes
138     its default value, in which case the threshold will be converted to a float
139     """
140
141     # if img.dtype is floating but threshold value is still the default
142     # this could be generalized
143     if np.issubdtype(img.dtype, np.floating) and (threshold == 175):
144         threshold = 175 / 255
145     # region to inpaint
146     inp = (img > threshold)
147
148     # get a larger area around the specks
149     inp = binary_dilation(inp, selem=disk(1))
150
151     # remove anything large
152     #inp = white_tophat(inp, selem=disk(3))
153
154     if mask_only:
155         return inp
156     else:
157         # both the original background *and* these new glared regions
158         # are masked
159         return ma.masked_array(img, mask=inp)
160
161
162 DARK_RED = np.array([103, 15, 23]) / 255.
163
164 # test it on a particularly bad sample
165 if __name__ == "__main__":
166
167     from placenta import get_named_placenta, show_mask
168     import matplotlib.pyplot as plt
169
170     filename = 'T-BN0204423.png' # a particularly glary sample
171     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(xcfffile)
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

```

```

149
150 register(
151     "process_NCS_xcf",
152     "Create base image + trace + mask from an NCS xcf file",
153     "Create base image + trace + mask from an NCS xcf file",
154     "Luke Wukmer",
155     "Luke Wukmer",
156     "2018",
157     "<Image>/Image/Process_NCS_xcf...",
158     "RGB*", "GRAY*",
159     [],
160     [],
161     process_NCS_xcf)
162
163 main()

```

listings/score.py

```

1 #!/usr/bin/env python3
2
3 import numpy as np
4 from placenta import open_typefile, open_tracefile
5
6 def rgb_to_widths(T):
7     """
8         this will take an RGB trace image (MxNx3) and return a 2D (MxN)
9         "labeled" trace corresponding to the traced pixel length.
10        there is no distinguishing between arteries and vessels
11
12        it's preferable to do this in real-time so only one tracefile
13        needs to be stored (making the sample folder less cluttered)
14        although obviously at the expense of storing a larger image
15        which is only needed for visualization purposes.
16
17    Input:
18        T: a MxNx3 RGB (uint8) array, where the colorations are
19        assumed as described in NOTES below.
20
21    Output:
22        widthtrace: a MxN array whose inputs describe the width of the
23        vessel (in pixels), see NOTES.
24
25    Notes:
26
27        The correspondence is as follows:
28        3 pixels: "#ff006f", # magenta
29        5 pixels: "#a80000", # dark red
30        7 pixels: "#a800ff", # purple
31        9 pixels: "#ff00ff", # light pink
32        11 pixels: "#008aff", # blue
33        13 pixels: "#8aff00", # green
34        15 pixels: "#ffc800", # dark yellow
35        17 pixels: "#ff8a00", # orange
36        19 pixels: "#ff0015" # bright red
37
38    According to the original tracing protocol, the traced vessels are
39    binned into these 9 sizes. Vessels with a diameter smaller than 3px
40    are not traced (unless they're binned into 3px).
41
42    Note: this does *not* deal with collisions. If you pass anything
43    with addition (blended colors) as the ctraces are, you will have
44    trouble, as those will not be registered as any of the colors above
45    and will thus be ignored. If you want to handle data from both

```

```

46 arterial *and* venous layers, you should do so outside of this
47 function.
48 """
49
50 # a 2D picture to fix in with the pixel widths
51 W = np.zeros_like(T[:, :, 0])
52
53 for pix, color in TRACE_COLORS.items():
54
55     #ignore pixelwidths outside the specified range
56     # get the 2D indices that are that color
57     idx = np.where(np.all(T == color, axis=-1))
58     W[idx] = pix
59
60
61 return W
62
63 def merge_widths_from_traces(A_trace, V_trace, strategy='minimum'):
64 """
65 combine the widths from two RGB-traces A_trace and V_trace
66 and return one width matrix according to 'strategy'
67
68 Parameters
69 -----
70 A_trace: ndarray
71     an MxNx3 matrix, where each pixel (along the
72     last dimension) is an RGB triplet (i.e. each entry
73     is an integer between [0,256]). The colors each
74     correspond to those in TRACE_COLORS, and (255,255,255)
75     signifies "no vessel". This will normally correspond to
76     the sample's arterial trace.
77 V_trace: ndarray
78     an MxNx3 matrix the same shape and other
79     requirements as A_trace (see above). This will normally
80     correspond to the sample's venous trace.
81 strategy: keyword string
82     when A_trace and V_trace coincide at some entry,
83     this is the merging strategy. It should be a keyword
84     of one of the following choices:
85
86     "minimum": take the minimum width of the two traces
87         (default). this is the sensible option if you
88         are filtering out larger widths.
89     "maximum": take the maximum width of the two traces
90     "artery" or "A" or "top": take the width from A_trace
91     "vein" or "V" or "bottom": take the width from V_trace
92
93 Returns
94 -----
95     W : ndarray
96     a width-matrix where each entry is a number 0 (no vessel), 3,5,7,...19
97
98 Notes
99 -----
100 Since arteries grow over the veins on the PCSVN and are generally easier
101 to extract, it might be preferable to indicate "arteries". In reality,
102 each strategy is a compromise, and only by keeping track of both would
103 you get the complete picture.
104
105 No filtering out widths is done here.
106 """
107 assert A_trace.shape == V_trace.shape
108 A = rgb_to_widths(A_trace)
109
```

```

110 V = rgb_to_widths(V_trace)
111
112 # collisions (where are widths both reported)
113 c = np.logical_and(A!=0, V!=0)
114
115 W = np.maximum(A,V) # get the nonzero value
116 if strategy == 'maximum':
117     pass # already done, else rewrite the collisions
118 elif strategy in ('arteries', 'A', 'top'):
119     W[c] = A[c]
120 elif strategy in ('veins', 'V', 'bottom'):
121     W[c] = V[c]
122 else:
123     if strategy != 'minimum':
124         print(f"Warning: unknown merge strategy: {strategy}")
125         print("Defaulting to minimum strategy")
126
127     W[c] = np.minimum(A[c], V[c])
128
129 return W
130
131 def filter_widths(W, widths=None, min_width=3, max_width=19):
132 """
133 Filter a width matrix, removing widths according to rules.
134
135 This function will take a 2D matrix of vessel widths and
136 remove any widths outside a particular range (or alternatively,
137 that are not included in a particular list)
138
139 Should be roughly as easy as doing it by hand, except that you
140 won't have to rewrite the code each time.
141
142 Inputs:
143
144 W: a width matrix (2D matrix with elements 0,3,5,7,...19
145
146 min_width: widths below this will be excluded (default is
147             3, the min recorded width). assuming these
148             are ints
149
150 max_width: widths above this will be excluded (default is
151             19, the max recorded width)
152
153 widths: an explicit list of widths that should be returned.
154             in this case the above min & max are ignored.
155             this way you could include widths = [3, 17, 19] only
156             """
157
158 Wout = W.copy()
159 if widths is None:
160     Wout[W < min_width] = 0
161     Wout[W > max_width] = 0
162
163 else:
164     # use numpy.isin(T, widths) but that's only in version 1.13 and up
165     # of numpy this is basically the code for that though
166     to_keep = np.in1d(W, widths, assume_unique=True).reshape(W.shape)
167     Wout[np.invert(to_keep)] = 0
168
169 return Wout
170
171 TRACE_COLORS = {
172     3: (255, 0, 111),
173     5: (168, 0, 0),

```

```

174     7: (168, 0, 255),
175     9: (255, 0, 255),
176    11: (0, 138, 255),
177    13: (138, 255, 0),
178    15: (255, 200, 0),
179    17: (255, 138, 0),
180    19: (255, 0, 21)
181 }
182
183
184 def widths_to_rgb(w, show_non_matches=False):
185     """Convert width matrix back to RGB values.
186
187     For display purposes/convenience. Return an RGB matrix
188     converting back from [3,5,7, ..., 19] -> TRACE_COLORS
189
190     this doesn't do any rounding (i.e. it ignores anything outside of
191     the default widths), but maybe you'd want to?
192     """
193     B = np.zeros((w.shape[0], w.shape[1], 3))
194
195     for px, rgb_triplet in TRACE_COLORS.items():
196         B[w == px, :] = rgb_triplet
197
198     if show_non_matches:
199         # everything in w not found in TRACE_COLORS will be black
200         B[w == 0, :] = (255, 255, 255)
201     else:
202         non_filled = (B == 0).all(axis=-1)
203
204         B[non_filled, :] = (255, 255, 255) # make everything white
205
206     # matplotlib likes the colors as [0,1], so....
207     return B / 255.
208
209
210 def _hex_to_rgb(hexstring):
211     """
212     there's a function that does this in matplotlib.colors
213     but its scaled between 0 and 1 but not even as an
214     array so this is just as much work
215
216     ##TODO rewrite everything so this is useful if it's not been
217     rewritten already.
218     """
219     triple = hexstring.strip("#")
220     return tuple(int(x,16) for x in (triple[:2],triple[2:4],triple[4:]))
221
222
223 def confusion(test, truth, bg_mask=None, colordict=None, tint_mask=True):
224     """
225     distinct coloration of false positives and negatives.
226
227     colors output matrix with
228         true_pos if test[-] == truth[-] == 1
229         true_neg if test[-] == truth[-] == 0
230         false_neg if test[-] == 0 and truth[-] == 1
231         false_pos if test[-] == 1 and truth[-] == 0
232
233     if colordict is supplied: you supply a dictionary of how to
234     color the four cases. Spec given by the default below:
235
236     if tint mask, then the mask is overlaid on the image, not replacing totally
237     colordict = {

```

```

238     'TN': (247, 247, 247), # true negative
239     'TP': (0, 0, 0) # true positive
240     'FN': (241, 163, 64), # false negative
241     'FP': (153, 142, 195), # false positive
242     'mask': (247, 200, 200) # mask color (not used in MCC calculation)
243   }
244   """
245
246   if colordict is None:
247     colordict = {
248       'TN': (247, 247, 247), # true negative# 'f7f7f7'
249       'TP': (0, 0, 0), # true positive # '000000'
250       'FN': (241, 163, 64), # false negative # 'f1a340' orange
251       'FP': (153, 142, 195), # false positive # '998ec4' purple
252       'mask': (247, 200, 200) # mask color (not used in MCC calculation)
253     }
254
255   #TODO: else check if mask is specified and add it as color of TN otherwise
256
257   true_neg_color = np.array(colordict['TN'], dtype='f')/255
258   true_pos_color = np.array(colordict['TP'], dtype='f')/255
259   false_neg_color = np.array(colordict['FN'], dtype='f') /255
260   false_pos_color = np.array(colordict['FP'], dtype='f')/255
261   mask_color = np.array(colordict['mask'], dtype='f') /255
262
263   assert test.shape == truth.shape
264
265   # convert to bool
266   test, truth = test.astype('bool'), truth.astype('bool')
267
268   # RGB array size of test and truth for output
269   output = np.zeros((test.shape[0], test.shape[1], 3), dtype='f')
270
271   # truth conditions
272   true_pos = np.bitwise_and(test==truth, truth)
273   true_neg = np.bitwise_and(test==truth, np.invert(truth))
274   false_neg = np.bitwise_and(truth, np.invert(test))
275   false_pos = np.bitwise_and(test, np.invert(truth))
276
277   output[true_pos,:,:] = true_pos_color
278   output[true_neg,:,:] = true_neg_color
279   output[false_pos,:,:] = false_pos_color
280   output[false_neg,:,:] = false_neg_color
281
282   # try to find a mask
283   if bg_mask is None:
284     try:
285       bg_mask = test.mask
286     except AttributeError:
287       # no mask is specified, we're done.
288       return output
289
290   # color the mask
291   if tint_mask:
292     output[bg_mask,:,:] += mask_color
293     output[bg_mask,:,:] /= 2
294   else:
295     output[bg_mask,:,:] = mask_color
296
297   return output
298
299
300 def compare_trace(approx, trace=None, filename=None,
301                   sample_dir=None, colordict=None):

```

```

302 """
303 compare approx matrix to trace matrix and output a confusion matrix.
304 if trace is not supplied, open the image from the tracefile.
305 if tracefile is not supplied, filename must be supplied, and
306 tracefile will be opened according to the standard pattern
307
308 colordict are parameters to pass to confusion()
309
310 returns a matrix
311 """
312
313 # load the tracefile if not supplied
314 if trace is None:
315     if filename is not None:
316         try:
317             trace = open_typefile(filename, 'trace')
318         except FileNotFoundError:
319             print("No trace file found matching ", filename)
320             print("no trace found. generating dummy trace.")
321             trace = np.zeros_like(approx)
322     else:
323         print("no trace supplied/found. generating dummy trace.")
324         trace = np.zeros_like(approx)
325
326 C = confusion(approx, trace, colordict=colordict)
327
328 return C
329
330
331 def mcc(test, truth, bg_mask=None, score_bg=False, return_counts=False):
332 """
333 Matthews correlation coefficient
334 returns a float between -1 and 1
335 -1 is total disagreement between test & truth
336 0 is "no better than random guessing"
337 1 is perfect prediction
338
339 bg_mask is a mask of pixels to ignore from the statistics
340 for example, things outside the placental plate will be counted
341 as "TRUE NEGATIVES" when there wasn't any chance of them not being
342 scored as negative. therefore, it's not really a measure of the
343 test's accuracy, but instead artificially pads the score higher.
344
345 setting bg_mask to None when test and truth are not masked
346 arrays should give you this artificially inflated score.
347 Passing score_bg=True makes this decision explicit, i.e.
348 any masks (even if supplied) will be ignored, and your count of
349 false positives will be inflated.
350
351 """
352 true_pos = np.logical_and(test==truth, truth)
353 true_neg = np.logical_and(test==truth, np.invert(truth))
354 false_neg = np.logical_and(truth, np.invert(test))
355 false_pos = np.logical_and(test, np.invert(truth))
356
357 if score_bg:
358     # take the classifications above as they are (nothing is masked)
359     pass
360 else:
361     # if no specified mask, check the test array itself?
362     if bg_mask is None:
363         try:
364             bg_mask = test.mask
365         except AttributeError:

```

```

366         # no mask is specified, we're done.
367         bg_mask = np.zeros_like(test)
368
369         # only get stats in the plate
370         true_pos[bg_mask] = 0
371         true_neg[bg_mask] = 0
372         false_pos[bg_mask] = 0
373         false_neg[bg_mask] = 0
374
375     # now tally
376     TP = true_pos.sum()
377     TN = true_neg.sum()
378     FP = false_pos.sum()
379     FN = false_neg.sum()
380
381     if not score_bg:
382         total = np.invert(bg_mask).sum()
383     else:
384         total = test.size
385     #print('TP: {} \t TN: {} \nFP: {} \tFN: {}'.format(TP, TN, FP, FN))
386     #print('TP+TN+FN+FP={} \ntotal pixels={}'.format(TP+TN+FP+FN, total))
387     # prevent potential overflow
388     denom = np.sqrt(TP+FP)*np.sqrt(TP+FN)*np.sqrt(TN+FP)*np.sqrt(TN+FN)
389
390     if denom == 0:
391         # set MCC to zero if any are zero
392         m_score = 0
393     else:
394         m_score = ((TP*TN) - (FP*FN)) / denom
395
396     if return_counts:
397         return m_score, (TP, TN, FP, FN)
398     else:
399         return m_score
400
401
402 def mean_squared_error(A,B):
403     """
404     get mean squared error between two matrices of the same size
405
406     input:
407         A, B : two ndarrays of the same size.
408
409     output:
410
411         mse:    a single number.
412     """
413
414     try:
415         mse = ((A-B)**2).sum() / A.size
416
417     except ValueError:
418         print("inputs must be of the same size")
419         raise
420
421     return mse
422
423
424 if __name__ == "__main__":
425
426     import matplotlib.pyplot as plt
427     from skimage.data import binary_blobs
428
429     A = binary_blobs()

```

```

430 B = binary_blobs()
431
432 true_neg_color = np.array([247,247,247], dtype='f') # 'f7f7f7'
433 true_pos_color = np.array([0, 0, 0], dtype='f') # '000000'
434 false_neg_color = np.array([241,163,64], dtype='f')# 'f1a340'
435 false_pos_color = np.array([153,142,195], dtype='f') # '998ec4'
436
437 C = confusion(A,B)
438
439 fig, (ax0, ax1, ax2) = plt.subplots(nrows=1,
440                                     ncols=3,
441                                     figsize=(8, 2.5),
442                                     sharex=True,
443                                     sharey=True)
444
445 ax0.imshow(A, cmap='gray')
446 ax0.set_title('A')
447 ax0.axis('off')
448 ax0.set_adjustable('box-forced')
449
450 ax1.imshow(B, cmap='gray')
451 ax1.set_title('B')
452 ax1.axis('off')
453 ax1.set_adjustable('box-forced')
454
455 ax2.imshow(C)
456 ax2.set_title('confusion matrix of A and B')
457 ax2.axis('off')
458 ax2.set_adjustable('box-forced')
459
460 fig.tight_layout()

```

listings/scoring.py

```

1 #!/usr/bin/env python3
2
3 import numpy as np
4 from placenta import open_typefile, open_tracefile
5 from skimage.morphology import thin
6
7 def rgb_to_widths(T):
8     """
9         this will take an RGB trace image (MxNx3) and return a 2D (MxN)
10        "labeled" trace corresponding to the traced pixel length.
11        there is no distinguishing between arteries and vessels
12
13        it's preferable to do this in real-time so only one tracefile
14        needs to be stored (making the sample folder less cluttered)
15        although obviously at the expense of storing a larger image
16        which is only needed for visualization purposes.
17
18    Input:
19        T: a MxNx3 RGB (uint8) array, where the colorations are
20            assumed as described in NOTES below.
21
22    Output:
23        widthtrace: a MxN array whose inputs describe the width of the
24            vessel (in pixels), see NOTES.
25
26    Notes:
27
28        The correspondence is as follows:
29        3 pixels: "#ff006f", # magenta

```

```

30     5 pixels: "#a80000", # dark red
31     7 pixels: "#a800ff", # purple
32     9 pixels: "#ff00ff", # light pink
33    11 pixels: "#008aff", # blue
34    13 pixels: "#8aff00", # green
35    15 pixels: "#ffc800", # dark yellow
36    17 pixels: "#ff8a00", # orange
37    19 pixels: "#ff0015" # bright red
38
39 According to the original tracing protocol, the traced vessels are
40 binned into these 9 sizes. Vessels with a diameter smaller than 3px
41 are not traced (unless they're binned into 3px).
42
43 Note: this does *not* deal with collisions. If you pass anything
44 with addition (blended colors) as the ctraces are, you will have
45 trouble, as those will not be registered as any of the colors above
46 and will thus be ignored. If you want to handle data from both
47 arterial *and* venous layers, you should do so outside of this
48 function.
49 """
50
51 # a 2D picture to fix in with the pixel widths
52 W = np.zeros_like(T[:, :, 0])
53
54 for pix, color in TRACE_COLORS.items():
55
56     #ignore pixelwidths outside the specified range
57     # get the 2D indices that are that color
58     idx = np.where(np.all(T == color, axis=-1))
59     W[idx] = pix
60
61
62 return W
63
64 def merge_widths_from_traces(A_trace, V_trace, strategy='minimum'):
65 """
66 combine the widths from two RGB-traces A_trace and V_trace
67 and return one width matrix according to 'strategy'
68
69 Parameters
70 -----
71 A_trace: ndarray
72     an MxNx3 matrix, where each pixel (along the
73     last dimension) is an RGB triplet (i.e. each entry
74     is an integer between [0,256]). The colors each
75     correspond to those in TRACE_COLORS, and (255,255,255)
76     signifies "no vessel". This will normally correspond to
77     the sample's arterial trace.
78 V_trace: ndarray
79     an MxNx3 matrix the same shape and other
80     requirements as A_trace (see above). This will normally
81     correspond to the sample's venous trace.
82 strategy: keyword string
83     when A_trace and V_trace coincide at some entry,
84     this is the merging strategy. It should be a keyword
85     of one of the following choices:
86
87     "minimum": take the minimum width of the two traces
88             (default). this is the sensible option if you
89             are filtering out larger widths.
90     "maximum": take the maximum width of the two traces
91     "artery" or "A" or "top": take the width from A_trace
92     "vein" or "V" or "bottom": take the width from V_trace
93

```

```

94     Returns
95     -----
96     W : ndarray
97         a width-matrix where each entry is a number 0 (no vessel), 3,5,7,...19
98
99     Notes
100    -----
101    Since arteries grow over the veins on the PCSVN and are generally easier
102    to extract, it might be preferable to indicate "arteries". In reality,
103    each strategy is a compromise, and only by keeping track of both would
104    you get the complete picture.
105
106    No filtering out widths is done here.
107    """
108    assert A_trace.shape == V_trace.shape
109
110    A = rgb_to_widths(A_trace)
111    V = rgb_to_widths(V_trace)
112
113    # collisions (where are widths both reported)
114    c = np.logical_and(A!=0, V!=0)
115
116    W = np.maximum(A,V) # get the nonzero value
117    if strategy == 'maximum':
118        pass # already done, else rewrite the collisions
119    elif strategy in ('arteries', 'A', 'top'):
120        W[c] = A[c]
121    elif strategy in ('veins', 'V', 'bottom'):
122        W[c] = V[c]
123    else:
124        if strategy != 'minimum':
125            print(f"Warning: unknown merge strategy: {strategy}")
126            print("Defaulting to minimum strategy")
127
128    W[c] = np.minimum(A[c], V[c])
129
130    return W
131
132 def filter_widths(W, widths=None, min_width=3, max_width=19):
133     """
134     Filter a width matrix, removing widths according to rules.
135
136     This function will take a 2D matrix of vessel widths and
137     remove any widths outside a particular range (or alternatively,
138     that are not included in a particular list)
139
140     Should be roughly as easy as doing it by hand, except that you
141     won't have to rewrite the code each time.
142
143     Inputs:
144
145     W: a width matrix (2D matrix with elements 0,3,5,7,...19
146
147     min_width: widths below this will be excluded (default is
148             3, the min recorded width). assuming these
149             are ints
150
151     max_width: widths above this will be excluded (default is
152             19, the max recorded width)
153
154     widths: an explicit list of widths that should be returned.
155             in this case the above min & max are ignored.
156             this way you could include widths = [3, 17, 19] only
157     """

```

```

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
170 return Wout
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:

```

```

350     print("no trace supplied/found. generating dummy trace.")
351     trace = np.zeros_like(approx)
352
353 C = confusion(approx, trace, colordict=colordict)
354
355 return C
356
357
358 def mcc(test, truth, bg_mask=None, score_bg=False, return_counts=False):
359     """
360     Matthews correlation coefficient
361     returns a float between -1 and 1
362     -1 is total disagreement between test & truth
363     0 is "no better than random guessing"
364     1 is perfect prediction
365
366     bg_mask is a mask of pixels to ignore from the statistics
367     for example, things outside the placental plate will be counted
368     as "TRUE NEGATIVES" when there wasn't any chance of them not being
369     scored as negative. therefore, it's not really a measure of the
370     test's accuracy, but instead artificially pads the score higher.
371
372     setting bg_mask to None when test and truth are not masked
373     arrays should give you this artificially inflated score.
374     Passing score_bg=True makes this decision explicit, i.e.
375     any masks (even if supplied) will be ignored, and your count of
376     false positives will be inflated.
377
378     """
379     true_pos = np.logical_and(test==truth, truth)
380     true_neg = np.logical_and(test==truth, np.invert(truth))
381     false_neg = np.logical_and(truth, np.invert(test))
382     false_pos = np.logical_and(test, np.invert(truth))
383
384     if score_bg:
385         # take the classifications above as they are (nothing is masked)
386         pass
387     else:
388         # if no specified mask, check the test array itself?
389         if bg_mask is None:
390             try:
391                 bg_mask = test.mask
392             except AttributeError:
393                 # no mask is specified, we're done.
394                 bg_mask = np.zeros_like(test)
395
396         # only get stats in the plate
397         true_pos[bg_mask] = 0
398         true_neg[bg_mask] = 0
399         false_pos[bg_mask] = 0
400         false_neg[bg_mask] = 0
401
402     # now tally
403     TP = true_pos.sum()
404     TN = true_neg.sum()
405     FP = false_pos.sum()
406     FN = false_neg.sum()
407
408     if not score_bg:
409         total = np.invert(bg_mask).sum()
410     else:
411         total = test.size
412     #print('TP: {} \t TN: {} \nFP: {} \tFN: {}'.format(TP, TN, FP, FN))
413     #print('TP+TN+FN+FP={} \ntotal pixels={}'.format(TP+TN+FP+TN, total))

```

```

414 # prevent potential overflow
415 denom = np.sqrt(TP+FP)*np.sqrt(TP+FN)*np.sqrt(TN+FP)*np.sqrt(TN+FN)
416
417 if denom == 0:
418     # set MCC to zero if any are zero
419     m_score = 0
420 else:
421     m_score = ((TP*TN) - (FP*FN)) / denom
422
423 if return_counts:
424     return m_score, (TP,TN,FP,FN)
425 else:
426     return m_score
427
428
429 def mean_squared_error(A,B):
430     """
431     get mean squared error between two matrices of the same size
432
433     input:
434         A, B : two ndarrays of the same size.
435
436     output:
437         mse:    a single number.
438     """
439
440
441 try:
442     mse = ((A-B)**2).sum() / A.size
443
444 except ValueError:
445     print("inputs must be of the same size")
446     raise
447
448 return mse
449
450
451 if __name__ == "__main__":
452
453     import matplotlib.pyplot as plt
454     from skimage.data import binary_blobs
455
456     A = binary_blobs()
457     B = binary_blobs()
458
459     true_neg_color = np.array([247,247,247], dtype='f') # 'f7f7f7'
460     true_pos_color = np.array([0, 0, 0], dtype='f') # '000000'
461     false_neg_color = np.array([241,163,64], dtype='f')# 'f1a340'
462     false_pos_color = np.array([153,142,195], dtype='f') # '998ec4'
463
464     C = confusion(A,B)
465
466     fig, (ax0, ax1, ax2) = plt.subplots(nrows=1,
467                                         ncols=3,
468                                         figsize=(8, 2.5),
469                                         sharex=True,
470                                         sharey=True)
471
472     ax0.imshow(A, cmap='gray')
473     ax0.set_title('A')
474     ax0.axis('off')
475     ax0.set_adjustable('box-forced')
476
477     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.