

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 [1]. 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 [2] 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 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	28
The Frangi vesselness filter: Choosing parameters β and c .	28
Linear Scale Space Theory	30
Axioms	33
Uniqueness of the Gaussian Kernel.....	35
Scale Spaces over Discrete Structures.....	39
The Frangi Filter: A multiscale approach	41
Thresholding	42
Calculating the 2D Hessian	43
Convolution Speedup via FFT	43
Fourier Transform of a continuous 1D signal.....	43
Fourier Transform of a Discrete 1D signal	44
2D DFT Convolution Theorem	45
FFT	47

APPENDIX	Page
3. RESEARCH PROTOCOL	52
Samples / Image Domain	52
A representative sample	52
Knowns and Unknowns	54
Data Cleaning and Preprocessing	55
Boundary Dilatation	57
Cut Removal	59
Deglaring	60
Multiscale Setup	64
4. IMPLEMENTATIONS	69
Calculating the Hessian via FFT	69
Postprocessing Techniques	74
Method A: Fixed Threshold	75
Method B: Percentile Based Merging	75
Method C: Scale-Based Random Walker	76
Method D: Scale Based Sieving	77
5. RESULTS AND ANALYSIS	80
Sample visual output	80
Binary Classificaitons and the confusion matrix	80
Variations in the Data Set and Imperfections of the Ground Truth	80
Results	86
Answer Research Questions	86
6. CONCLUSION	87
Future research directions	87
APPENDICES	90
A. CODE LISTINGS	91
B. 3D VISUALIZATION OF THE FRANGI FILTER	173
BIBLIOGRAPHY	180

LIST OF TABLES

	Page
1 Vessel width color code	54
2 MSE of Gaussian blurs of an image ($\sigma = 0.3$)	71
3 MSE of Frangi scores $\sigma = 0.3$	71
4 MSE of Gaussian blurs of an image ($\sigma = 5$)	73
5 MSE of Frangi scores $\sigma = 5$	73

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 Dependence of the Anisotropy Factor on its Parameter	31
5 Dependence of the Structureness Factor on its Parameter.....	32
6 A representative placental sample and tracing.....	53
7 Preprocessed files from an NCS sample.....	56
8 Effect of boundary dilation on Frangi responses.....	58
9 Removing a “cut” from the placental plate.	60
10 Deglaring a sample using a hybrid inpainting method	61
11 Comparison of glare inpainting methods (detail)	62
12 Signed Frangi output (plate and inset) (Example 1)	67
13 Signed Frangi output (plate and inset) (Example 1).....	68
14 Compatibility of Gaussian convolution strategies	70
15 Iterative Gaussian blur	72
16 Image cross-section of Gaussian blurred images	72
17 Image cross-section of Frangi targets images.....	72
18 Image cross-section of Gaussian blurred images $\sigma = 5$	73
19 Image cross-section of Frangi targets images $\sigma = 5$	73
20 Time required.....	74
21 Scale-wise random walker segmentation (select scales)	78
22 Random walker segmentation (result and sample).....	79
23 Sample Multiscale Frangi output ($\beta = 0.35$) with simple segmentation strategies (Example 1).....	81
24 Sample Multiscale Frangi output ($\beta = 0.35$) with simple segmentation strategies (Example 2).....	82
25 Demonstration of postprocessing techniques	83
26 Issues with ground truth and sample quality	84
27 Signed Frangi output (plate and inset) (Example 1)	88
28 Signed Frangi output (plate and inset) (Example 1)	89
29 3D graph of the Frangi Vesselness Measure, variable $\gamma, \beta = 0.1$	174
30 3D graph of the Frangi Vesselness Measure, variable $\gamma, \beta = 0.25$	175
31 3D graph of the Frangi Vesselness Measure, variable $\gamma, \beta = 0.5$	176
32 3D graph of the Frangi Vesselness Measure, variable $\gamma, \beta = 0.9$	177
33 3D graph of the Frangi Vesselness Measure, variable $\gamma, \beta = 1$	178
34 3D graph of the Frangi Vesselness Measure, variable $\gamma, \beta = 1.5$	179

CHAPTER 1

INTRODUCTION

From [1], it is useful to develop a neonatal test for high risk of Autism Spectrum Disorder. There is some evidence as in [3] 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 [3] that measurements of the placental chorionic surface vascular network (PCSVN) may be useful in identifying such risk. [1] 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 [4] [2] [5]. 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 [2] 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 its justification as an image-processing technique, as well as an introduction to the Gaussian scale space theory common to many multiscale methods. 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 ($N = 20$) multiscale Frangi filter on a set of 201 placental images from a private database provided by the National Children’s Study (NCS). We then demonstrate several approaches to merging the multiscale result into an approximation of the PCSVN and compare these to manual tracings of the network. Our ability to take many more scales into consideration allows us to be pickier about our thresholding, as well as our choice of parameters, 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.

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 [6], 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 section 3.1), 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 [7] (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*

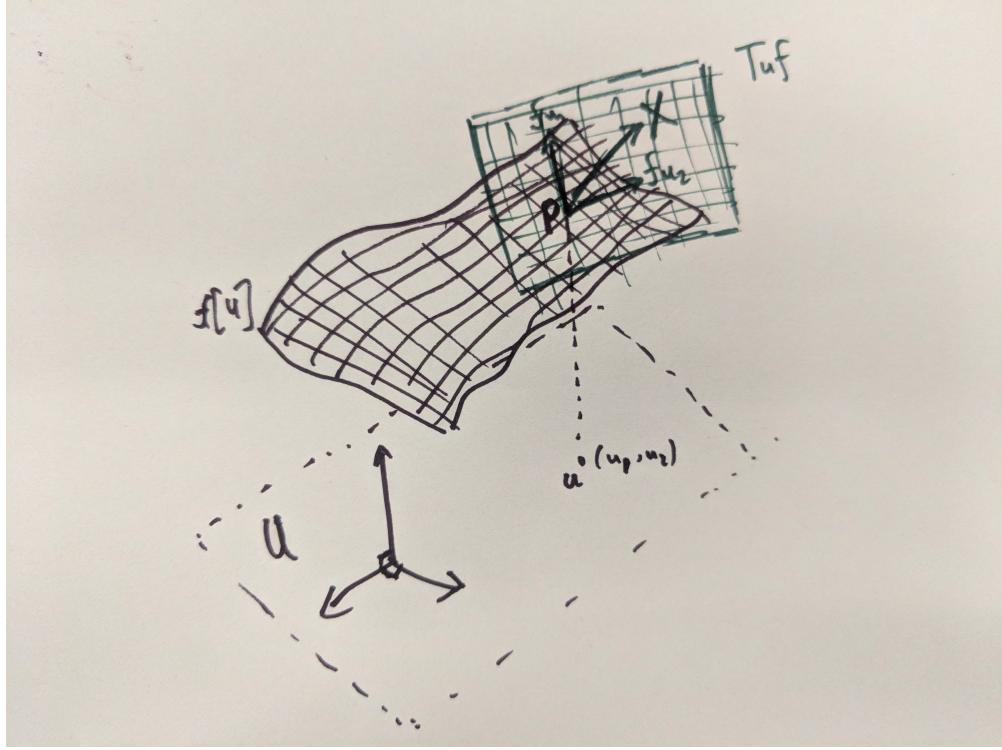


FIGURE 1: Tangent plane of a graph

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 [7] 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 $\widehat{L} \left. \frac{\partial f}{\partial u_i} \right|_{T_u f} = - \left. \frac{\partial v}{\partial u_i} \right|_{T_u f} \quad \text{for } i = 1, 2 \text{ where}$

$- X|_{T_u f}$ 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

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

$$= \begin{bmatrix} | & | \\ -\left. \frac{\partial v}{\partial u_1} \right|_{T_u f} & -\left. \frac{\partial v}{\partial u_2} \right|_{T_u f} \\ | & | \end{bmatrix} \quad (2.21)$$

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

$$\Rightarrow \left[\widehat{\mathbf{L}} \right] \begin{bmatrix} -\frac{\partial f}{\partial u_1} \\ -\frac{\partial f}{\partial u_2} \end{bmatrix} \begin{bmatrix} | & | \\ \left. \frac{\partial f}{\partial u_1} \right|_{T_u f} & \left. \frac{\partial f}{\partial u_2} \right|_{T_u f} \\ | & | \end{bmatrix} = \begin{bmatrix} -\frac{\partial f}{\partial u_1} \\ -\frac{\partial f}{\partial u_2} \end{bmatrix} \begin{bmatrix} | & | \\ -\left. \frac{\partial v}{\partial u_1} \right|_{T_u f} & -\left. \frac{\partial v}{\partial u_2} \right|_{T_u f} \\ | & | \end{bmatrix} \quad (2.22)$$

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

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

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 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 [8], 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 [9] 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

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

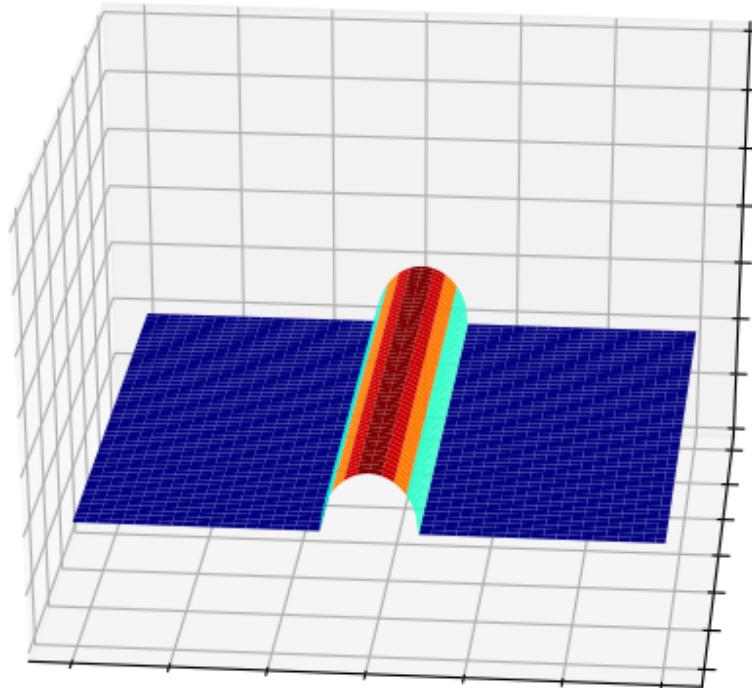


FIGURE 2: The graph of a cylindrical ridge of radius r

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) :

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

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

$$= \begin{bmatrix} -\frac{1}{r} & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad (2.53)$$

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:

$$\text{Hess}(x, y) = \begin{bmatrix} L_{xx}(x, y) & L_{xy}(x, y) & L_{yx}(x, y) & L_{yy}(x, y) \end{bmatrix} \quad (2.54)$$

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

$$\text{Hess}(x, y)u_i = \kappa_i u_i, \quad i = 1, 2 \quad (2.55)$$

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, [10]:

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 [11] is a widely used (cite) Hessian-based filter within image processing. Hessian-based filters make use of the logical “proximity” of the Hessian to notions of curvature of surfaces, as developed in section 2.2. Several such Hessian-based filters exist—see [12] and [13], as well as a

comparison given in [14]. 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:

$$V_\sigma(x_0, y_0) = \begin{cases} 0 & \text{if } \lambda_2 > 0 \\ \exp\left\{-\frac{A^2}{2\beta^2}\right\} \left(1 - \exp\left\{-\frac{S^2}{2c^2}\right\}\right) & \text{otherwise} \end{cases} \quad (2.56)$$

where

$$A := |\lambda_1/\lambda_2| \quad \text{and} \quad S := \sqrt{\lambda_1^2 + \lambda_2^2} \quad (2.57)$$

and β and c are tuning parameters. Before we discuss appropriate values for β and c , we first seek to highlight the significance of eq. (2.56), and in particular, the ratios defined in eq. (2.57). A and S are known as the anisotropy measure and structureness measure, respectively. Consequently, we'll refer to the two factors in eq. (2.56) as the anisotropy factor and structureness factor, 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

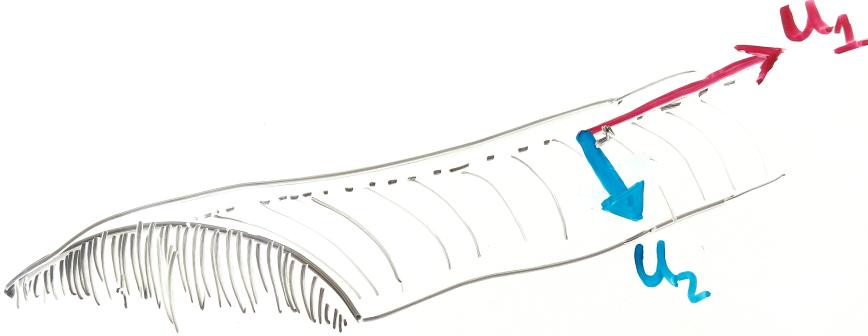


FIGURE 3: The principal eigenvectors at a ridge like structure

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

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 naïve 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, but instead, somewhere along it.

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 3.1), 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 6.1.

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 Frobenius 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. (2.56) in which a factor of $\exp\{\dots\}$ and $(1 - \exp\{\dots\})$ 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 rapid decay in other situations.

Frangi further strengthened the filter by adding an additional case to in eq. (2.56), 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. Conversely, we could only seek to find valley, or local minima, as thus require $\lambda_2 > 0$, and set the vesselness measure to zero when $\lambda_2 < 0$.

The Frangi vesselness filter: Choosing parameters β and c

The parameters β and c are meant to scale so that the peaks of the anisotropy factor $\exp\{\frac{-A^2}{2\beta^2}\}$ and the structureness factor $(1 - \exp\{\frac{-S^2}{2c^2}\})$ coincide enough to be statistically significant at highly curvilinear structures, but rapidly decay in areas not associated with curvilinear content. What values of these parameters are appropriate is ultimately dependent on the context of the problem.

Frangi suggested for c 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 exactly the max Frobenius norm. With this in mind we would like to introduce the scaling factor γ , so that $c = \gamma S_{\max}$. This creates a minor annoyance though: although the anisotropy factor can certainly attain a value of 1, if c is to take this “appropriate” value, the maximum value of the structureness factor is somewhat smaller than 1. In fact,

$$\begin{aligned}\max\{V_\sigma\} &\leq \max\left(\exp\left\{\frac{-A^2}{2\beta^2}\right\}\right) \max\left(\left(1 - \exp\left\{\frac{-S^2}{2(\gamma S_{\max})^2}\right\}\right)\right) \\ &\leq \max\left\{\left(1 - \exp\left\{\frac{-S^2}{2(\gamma S_{\max})^2}\right\}\right)\right\} \\ &= \left(1 - \exp\left\{\frac{-(S_{\max})^2}{2(\gamma S_{\max})^2}\right\}\right) = \left(1 - \exp\left\{\frac{-1}{2\gamma^2}\right\}\right)\end{aligned}\tag{2.58}$$

Thus, when γ takes the suggested value of $\gamma = 1/2$, the above calculation suggests that the maximum theoretical value that the Frangi filter could attain is $\max\{V_\sigma\} \leq 1 - \exp\{-1\} \approx .8647$. This (among other obvious reasons) certainly justifies Frangi’s description of the vesselness measure as only “probability-like.” Still, we would like the filter’s sensitivity to relative structureness to not have the effect of dampening the Filter as a whole, so we will introduce a rescaling factor a_γ , which is an explicit function of γ that rescales V so that the structureness factor has a maximum output score of 1 regardless of choice of γ . Our final Frangi vesselness measure is thus

$$V_\sigma(x_0, y_0) = \begin{cases} 0 & \text{if } \lambda_2 > 0 \\ a_\gamma \exp\left\{\frac{-A^2}{2\beta^2}\right\} \left(1 - \exp\left\{\frac{-S^2}{2(\gamma S_{\max})^2}\right\}\right) & \text{otherwise} \end{cases}\tag{2.59}$$

where, as before,

$$A := |\lambda_1/\lambda_2|, S := \sqrt{\lambda_1^2 + \lambda_2^2} \text{ and } a_\gamma = \left(1 - \exp\left(\frac{-1}{2\gamma^2}\right)\right)^{-1}$$

and

$$|\lambda_1| \leq |\lambda_2| \text{ are eigenvalues of } \text{Hess}_\sigma(\mathbf{I}(x_0, y_0))$$

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.

Considering as the anisotropy measure $(\lambda_1/\lambda_2) \in [0, 1]$ (simply since $|\lambda_1| \geq |\lambda_2|$), we can actually visualize how much the anisotropy factor varies depending on our choice of β , as seen in fig. 4.

We make a similar presentation of the dependence of the structureness kernel on its parameter γ , as you can see in fig. 5

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 [15] argued that “any image

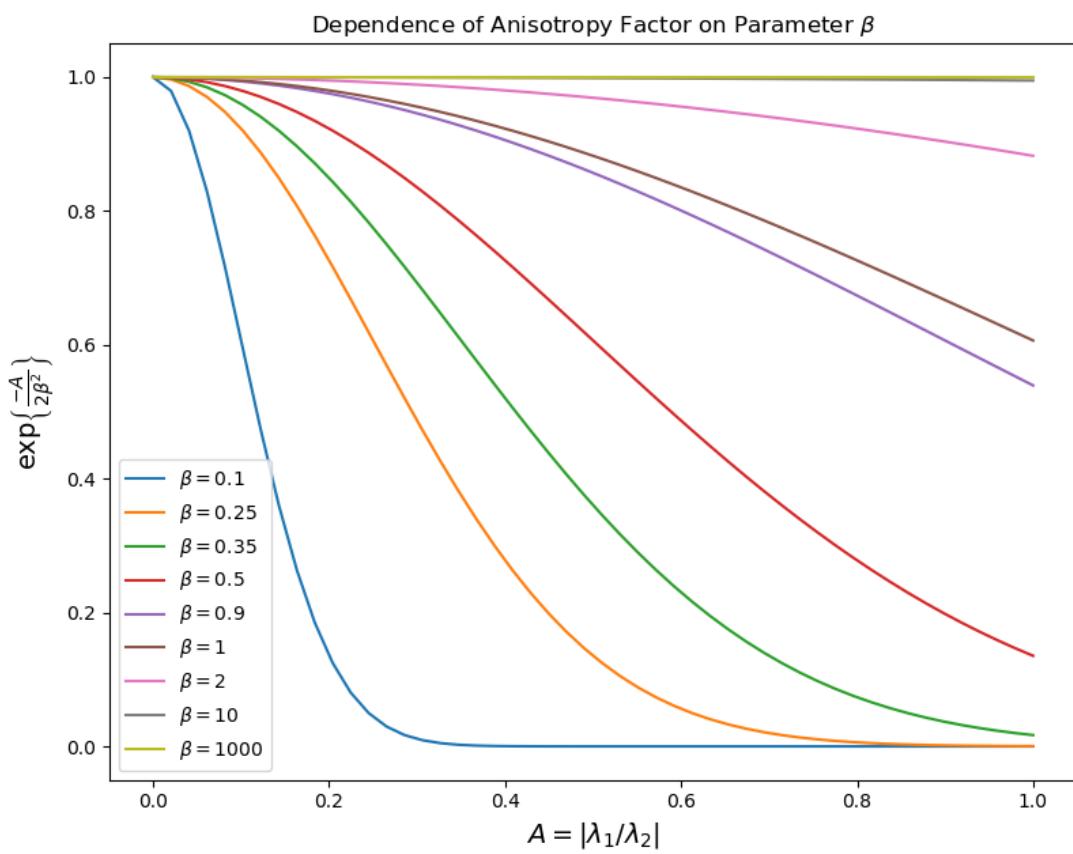


FIGURE 4: Dependence of the Anisotropy Factor on its Parameter

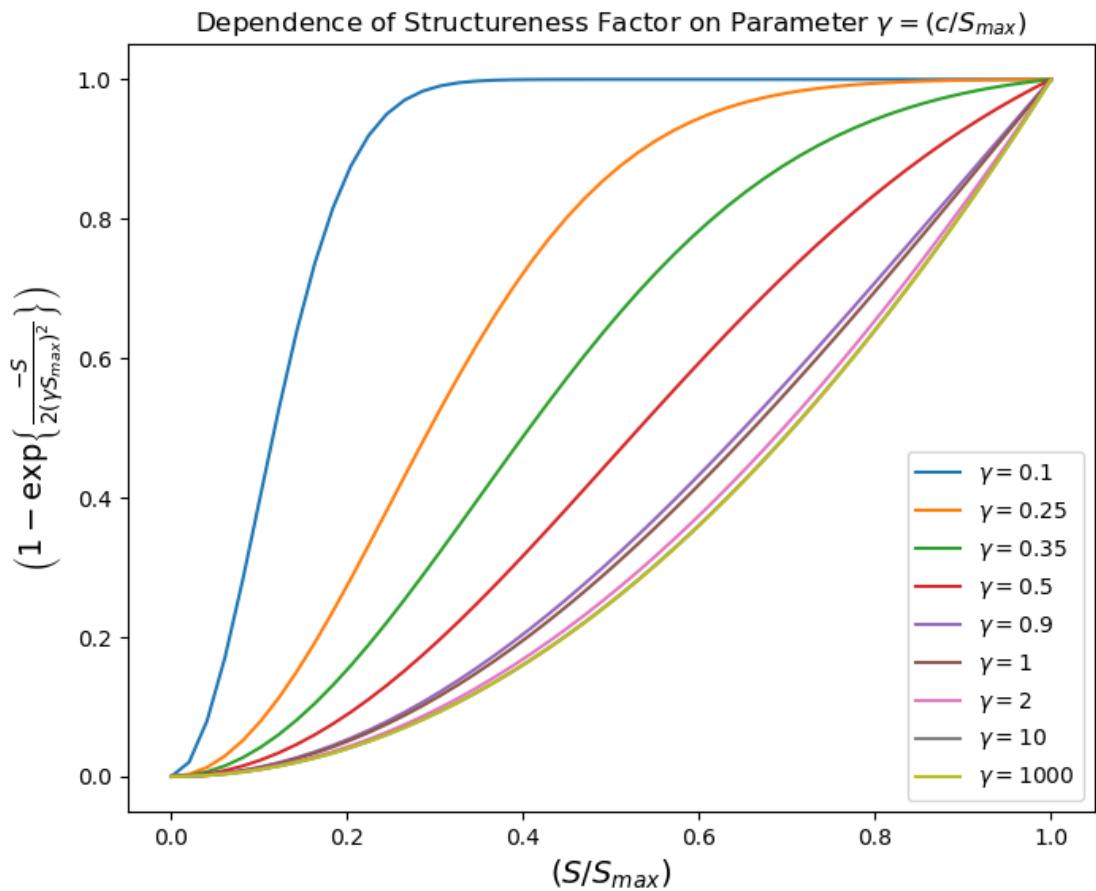


FIGURE 5: Dependence of the Structureness Factor on its Parameter

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

$$\Delta K(x, y, \sigma) = K_\sigma(x, y, \sigma) \text{ for } \sigma \geq 0 \text{ such that } K(x, y, 0) = u_0(x, y). \quad (2.60)$$

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 [16]. 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 [17]) implies that there is a so-called “infinitesimal generator” A which is a limit case of our desired operator T ; that is

$$Au_0 = \lim_{\sigma \downarrow 0} \frac{T_\sigma u_0 - u_0}{\sigma} \quad (2.61)$$

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

$$\partial_\sigma K(x, y; \sigma) = \lim_{\sigma \downarrow 0} \frac{K(\cdot; \sigma + h) - K(\cdot; \sigma)}{h} = A(T_\sigma u) = A(K(\cdot, \sigma)) \quad (2.62)$$

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.*

$$T_\sigma u = T_{\sigma_1 + \sigma_2} u \quad (2.63)$$

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.

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

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.

$$\begin{cases} \nabla K(x_0, y_0; \sigma_0) = 0 \\ \Delta K(x_0, y_0; \sigma_0) > 0 \end{cases} \implies K(x_0, y_0; \sigma_1) \geq K(x_0, y_0; \sigma_0) \forall \sigma_1 \geq \sigma_0 \quad (2.65)$$

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. (2.60) must hold. This has been shown in various ways, both by Koenderink [15], Babaud [18], as well as Lindeberg in [16]. 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,

$$K(x, y; \sigma) = T_\sigma u_0 = G_\sigma \star u_0 \quad \text{where} \quad G_\sigma := \frac{1}{2\pi\sigma^2} e^{(-|x|^2/(2\sigma^2))} \quad (2.66)$$

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.

$$u : \mathbb{R}^2 \supset \Omega \rightarrow \mathbb{R} \text{ with } u(\mathbf{x}, t) : \begin{cases} \frac{\partial u}{\partial t}(\mathbf{x}, t) = \Delta u(\mathbf{x}, t) & , t \geq 0 \\ u(\mathbf{x}, 0) = u_0(\mathbf{x}) \end{cases} \quad (2.67)$$

We show that

$$u(\mathbf{x}, t) = (G_{\sqrt{2t}} \star u_0)(\mathbf{x}) \quad (2.68)$$

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,

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

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

$$= \int f(t) \left(\frac{\partial g}{\partial \alpha} \right) g(\alpha - t) dt \quad (2.71)$$

$$= f(\alpha) \star g'(\alpha) \quad (2.72)$$

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

$$\Delta(f \star g) = \Delta(f) \star g = f \star \Delta(g) \quad (2.73)$$

□

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

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

$$= \frac{\partial}{\partial t} (G_{\sqrt{2t}}(\mathbf{x}, t)) \star u_0(\mathbf{x}) \quad (2.75)$$

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

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

$$= -\frac{1}{4t^2} \left(e^{(-|\mathbf{x}|^2/(4t))} + |\mathbf{x}|^2 G_{\sqrt{2t}}(\mathbf{x}, t) \right) \star u_0(\mathbf{x}) \quad (2.78)$$

and from the previous lemma,

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

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

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

By symmetry of argument we also may conclude

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

and so

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

Then, given lemma 2.9, we conclude

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

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

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

$$= \left(G_{\sqrt{2t}}(x, y) \left[\frac{x^2 + y^2}{4t^2} - \frac{1}{t} \right] \right) \star u_0(x, y) \quad (2.82)$$

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

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

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

$$= \frac{\partial}{\partial t} [G_{\sqrt{2t}}(x, y)] \star u_0(x, y) \quad (2.84)$$

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

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

$$= \left(G_{\sqrt{2t}}(x, y) \left[\frac{x^2 + y^2}{4t^2} - \frac{1}{t} \right] \right) \star u_0(x, y) \quad (2.87)$$

Combining these results, we find that

$$\frac{\partial}{\partial t} [G_{\sqrt{2t}} \star u_0] = \Delta [G_{\sqrt{2t}} \star u_0] \quad (2.88)$$

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 [19]. However, we've chosen to present the above in their original continuous surface for clarity of argument. The discrete case is not much different– we still have the same axioms, and it can be shown that the family of scaled images must simply satisfy a discrete version of the However, viewing our actual image definition 2.1 as a sample of a continuous surface definition 2.2, we might naively 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 [19] 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*

$$T(n; \sigma) = e^{-\alpha\sigma} I_n(\alpha\sigma), \quad I_n(\sigma) = I_{-n}(\sigma) = (-1)^n J_n(i\sigma) \quad n \geq 0, \sigma, \alpha > 0 \quad (2.89)$$

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 [21] (which involves the gamma function), since we only desire Bessel functions of integer order. The parameter α above is simply an optional scaling parameter which is simply set to 1 hereforth.

The derived family of 1D signals is then given by

$$K(x, \sigma) = \sum_{n=-\infty}^{\infty} T(n; t) f(x-n) \quad \text{for } x \in \mathbb{Z}, t > 0 \quad (2.90)$$

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. (2.90) can be obtained by independent convolution of its dimensions (i.e. it is separable). We will make these ideas explicit in chapter 4 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 [11] approached this problem by simply aggregating vesselness measure over all scales:

$$V(x_0, y_0) = \max_{\sigma \in \Sigma} V_\sigma(x_0, y_0) \quad (2.91)$$

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:

$$V_\Sigma := [V(x, y)]_{0 \leq x < M, 0 \leq y < N} \quad (2.92)$$

Notably, Frangi [11] refrained from explicitly interpreting the probability assigned by eq. (2.91); 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. (2.91) or eq. (2.92)

$$V_{\Sigma, \alpha}(x, y) = \begin{cases} 1 & \text{if } V(x, y) \geq \alpha \\ 0 & \text{else} \end{cases}, \quad \alpha > 0 \text{ for } \alpha \text{ fixed.} \quad (2.93)$$

We will discuss alternatives methods of aggregating results from our multiscale

method, as well as optimal values for parameters and scales in chapter 4. 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. (2.92), 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 [24]. Specific implementation will be discussed in chapter 4.

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 [6]. 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 4.

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 [6]. 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:

$$f(t) = \sum_{n=-\infty}^{\infty} c_n e^{j\frac{2\pi n}{T}t}, \quad c_n = \frac{1}{T} \int_{-T/2}^{T/2} f(t) e^{-i\frac{2\pi n}{T}t} dt \quad (2.94)$$

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

$$F(\mu) := \mathcal{F}\{f(t)\} = \int_{-\infty}^{\infty} f(t) e^{j2\pi\mu t} dt \quad (2.95)$$

An inverse transform will then recover our original signal:

$$f(t) = \mathcal{F}^{-1}\{F(\mu)\} = \int_{-\infty}^{\infty} F(\mu) e^{j2\pi\mu t} dt \quad (2.96)$$

Together, eq. (2.95) and eq. (2.96) 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 [6]. 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 :

$$s_{\Delta T}(t) := \sum_{n=-\infty}^{\infty} \delta[t - n\Delta T], \quad \delta[t] = \begin{cases} 1, & t = 0 \\ 0, & t \neq 0 \end{cases} \quad (2.97)$$

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

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

$$\tilde{f}(t) = f(t)s_{\Delta T}(t) \quad (2.98)$$

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.

$$\tilde{F}(\mu) = \sum_{n=-\infty}^{\infty} f_n e^{-i2\pi\mu n \Delta T}, \quad f_n = \tilde{f}(n) = f(n\Delta T) \quad (2.99)$$

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:*

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

$$H(u, v) := \mathcal{D}\{h(x, y)\} = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} h(x, y) e^{-2\pi i (\frac{ux}{M} + \frac{vy}{N})} \quad (2.101)$$

and given the convolution of the two functions

$$(f \star h)(x, y) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m, n) h(x - m, y - n) \quad (2.102)$$

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

$$(f \star h)(x, y) = \mathcal{D}^{-1}\{MN \cdot F(u, v)H(u, v)\} \quad (2.103)$$

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

Proof.

$$(f \star h)(x, y) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m, n)h(x-m, y-n) \quad (2.104)$$

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

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

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

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

$$= MN \cdot \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u, v) H(u, v) e^{2\pi i (\frac{ux}{M} + \frac{vy}{N})} \quad (2.109)$$

$$= MN \cdot \mathcal{D}^{-1} \{ FH \} \quad (2.110)$$

where

$$\hat{\delta}_N(k) = \begin{cases} 1 & \text{when } k = 0 \pmod{N} \\ 0 & \text{else} \end{cases} \quad (2.111)$$

□

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

$$\sum_{n=0}^{N-1} e^{2\pi i \left(\frac{n(j-k)}{N} \right)} = N \cdot \hat{\delta}_N(j-k) \quad (2.112)$$

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

$$z^N - 1 = (z - 1)(z^{n-1} + \dots + z + 1) = (z - 1) \sum_{n=0}^{N-1} z^n. \quad (2.113)$$

thus giving us

$$0 = \left(e^{2\pi i(j-k)/N} - 1\right) \sum_{n=0}^{N-1} e^{2\pi i n(j-k)/N} \quad (2.114)$$

To prove the claim in eq. (2.112), 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. (2.112) 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. (2.114). 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. (2.111), 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 [6]. 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:

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

$$= \sum_{x=0}^{M-1} e^{-i2\pi\mu x/M} \left[\sum_{y=0}^{N-1} f(x, y) e^{-i2\pi\nu y/N} \right] \quad (2.116)$$

$$= \sum_{x=0}^{M-1} e^{-i2\pi\mu x/M} \mathcal{F}_x\{f(x, y)\} \quad (2.117)$$

$$= \mathcal{F}_y\{\mathcal{F}_x\{f(x, y)\}\} \quad (2.118)$$

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

$$F(u) = \sum_{x=0}^{M-1} f(x) W_M^{ux}, \quad \text{where} \quad W_m := e^{-i2\pi/m} \quad (2.119)$$

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

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

$$= \sum_{x=0}^{K-1} f(2x) W_{2K}^{u(2x)} + \sum_{x=0}^{K-1} f(2x+1) W_{2K}^{u(2x+1)} \quad (2.121)$$

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

$$W_{(2m)}^{(2n)} = e^{\frac{-i2\pi(2m)}{2m}} = e^{\frac{-i2\pi m}{n}} = W_m^n \quad (2.122)$$

$$W_m^{(u+m)x} = e^{\frac{-i2\pi(u+m)x}{m}} = e^{\frac{-i2\pi unx}{m}} e^{\frac{-i2\pi mx}{m}} = e^{\frac{-i2\pi ux}{m}} (1) = W_m^{ux} \quad (2.123)$$

$$W_{2m}^{(u+m)} = e^{\frac{-i2\pi(u+m)}{2m}} = e^{\frac{-i2\pi ux}{2m}} e^{-i\pi} = W_{2m}^u e^{-i\pi} = -W_{2m}^u \quad (2.124)$$

Thus we can rewrite eq. (2.121) as

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

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

The major advance comes via using the identities eq. (2.122) to consider the

Fourier transform K frequencies later :

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

$$\implies F(u+K) = \left(\sum_{x=0}^{K-1} f(2x) W_K^{ux} \right) - \left(\sum_{x=0}^{K-1} f(2x+1) W_K^{ux} \right) W_K^u \quad (2.128)$$

Comparing eq. (2.126) and eq. (2.128), 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:

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

$$\mathcal{D}_u\{f_{\text{odd}}(t)\} := \sum_{x=0}^{K-1} f(2x+1) W_K^{ux} \quad (2.130)$$

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. (2.129) 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. (2.129) 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. (2.115).

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

RESEARCH PROTOCOL

Samples / Image Domain

We ultimately perform a multiscale Frangi filter to estimate the PCSVN from 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 [1], and a description of the cleaning and fixing procedure is given in [4]. 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. 6.

Each layer is roughly 1954x1200 pixels (with some occasional variation). In fig. 6a, 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. 6b 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. 6c and fig. 6d 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. 6e. 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

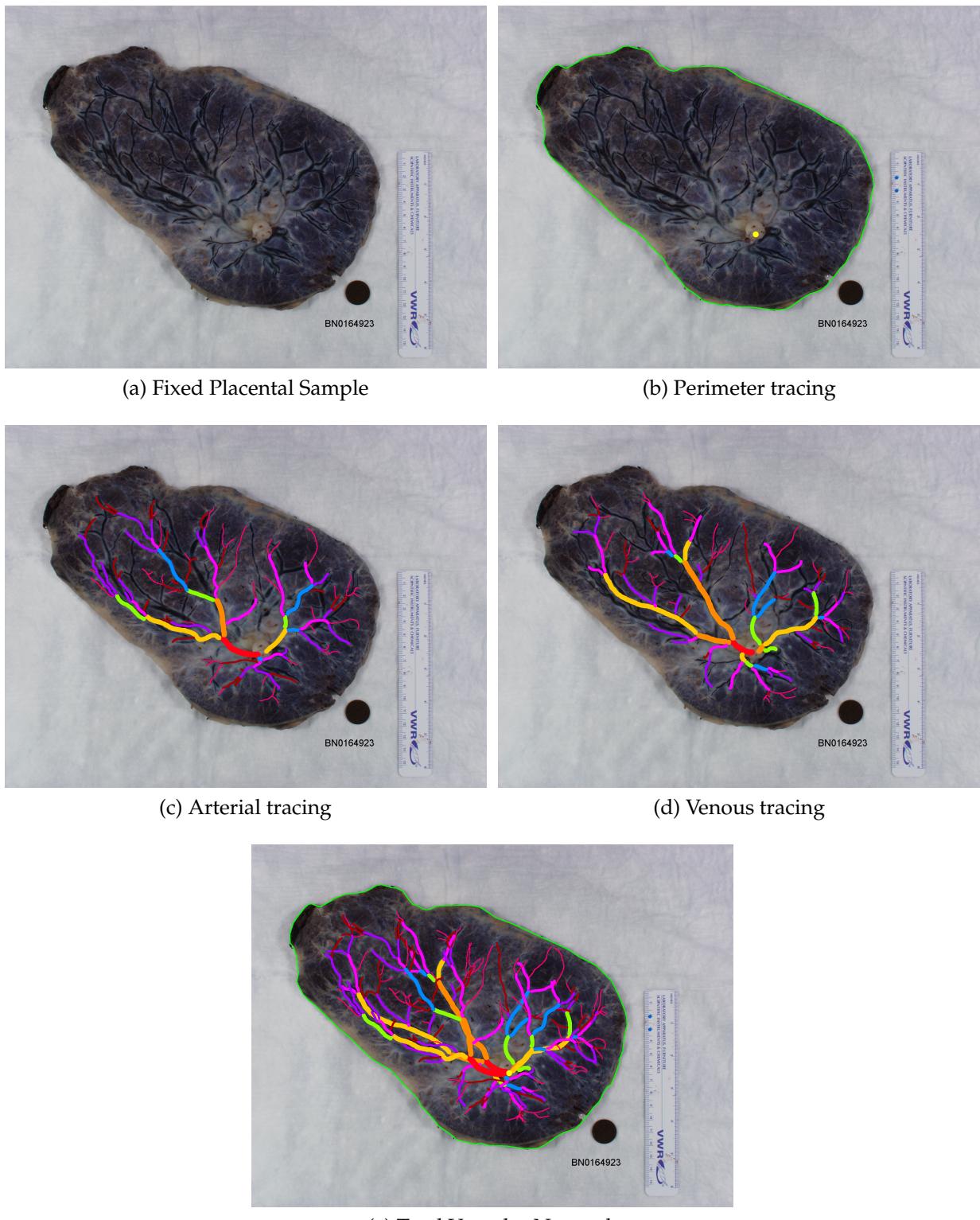


FIGURE 6: A representative placental sample and tracing

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

diameter are either binned to three or (quite frequently) left untraced. The 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. 6e or fig. 7d 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. 6). 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 [25] or, or even a simple edge finding algorithm followed by watershedding and largest object selection as in [2]. 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.

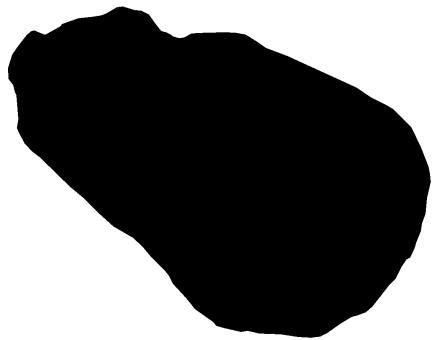
Finally, we will consider the location of the umbilical insertion point as a “known”, as the vessels around it are frequently impossible to see and we wish to exclude them from consideration. It is not unreasonable, however, to consider this to be a known—in future preparations of samples, we could simply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

Data Cleaning and Preprocessing

Building a sample suitable for use in our algorithm from fig. 6 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 3.2.

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.

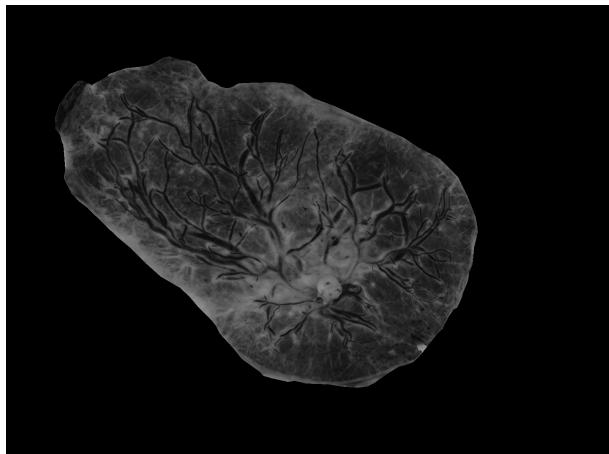
As a point of technicality, the grayscale image in fig. 7c is not actually produced directly by the extractor plug-in, but created when the 3 channel RGB image fig. 7b is



(a) Background Mask (in white)



(b) Sample with BG removed



(c) Grayscale



(d) Trace / “Ground Truth”

FIGURE 7: Preprocessed files from an NCS sample

imported at the start of the algorithm. This grayscale conversion is simply done for ease of analysis on the sample: although the Frangi filter is designed for arbitrary N-dimensional input [11], 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 [26], or “luminance” transform:

$$L = \frac{299}{1000} R + \frac{587}{1000} G + \frac{114}{1000} B \quad (3.1)$$

It should be noted that this choice is not automatic—several other attempts have used the green channel unmodified, as in [4] and [2].

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` [27]. The first, `skimage.segmentation.find_boundaries()`, takes the mask input (such as fig. 7a) and calculates where differences in a morphological erosion and dilation occur (which should have the same effect as using the perimeter labeled in fig. 6b 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

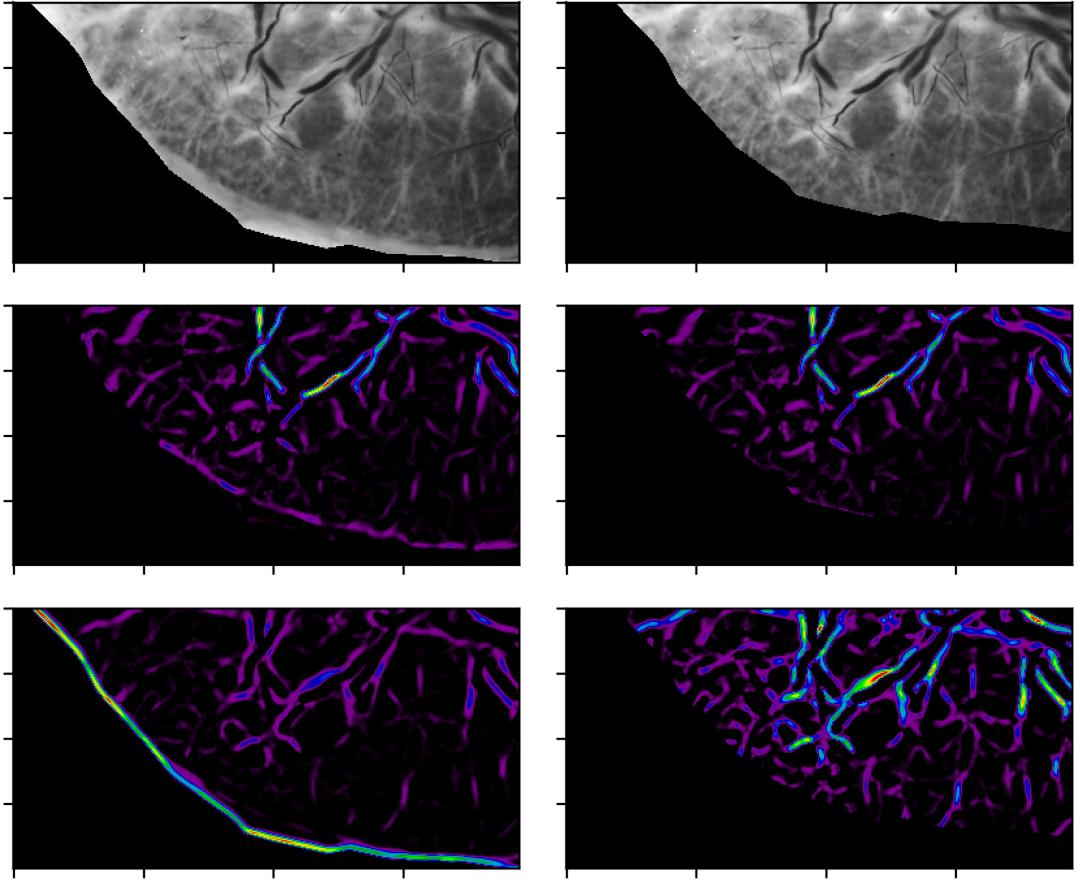


FIGURE 8: Effect of boundary dilation on Frangi responses

indices for the image where the boundary occurs and simply extend the mask R pixels in 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. 8 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. 8 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 [2], [4]). 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. 8 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

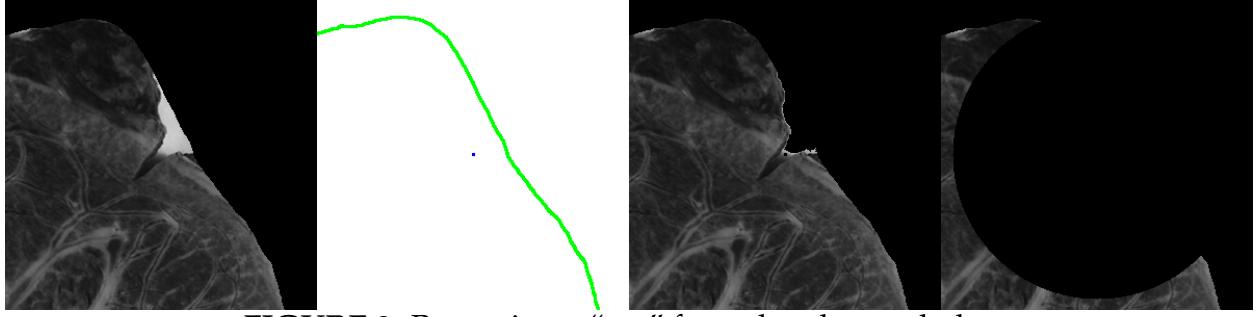


FIGURE 9: Removing a “cut” from the placental plate.

based on the background value of the blue dot; if this is incorrectly placed (or anything 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. 9. 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. In few additional cases, neither approach is adequate. We again will stress that a fully automated algorithm would have no knowledge of our traced perimeter (as in fig. 6b or the second from left image in fig. 9), so we anticipate a fully automated method of handling that problem should also be able to correct for these “cuts” as well.

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 [4] and [2]. 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.

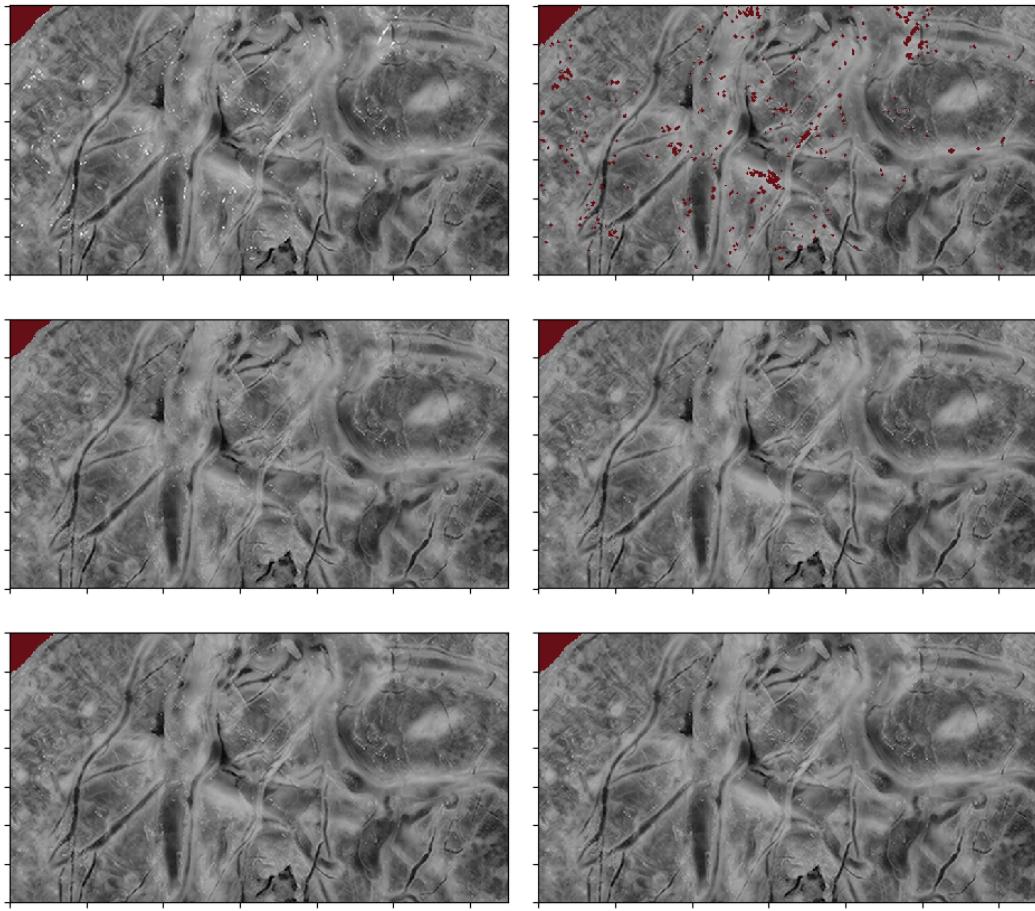


FIGURE 10: Deglaring a sample using a hybrid inpainting method

The function `inpaint_biharmonic` is based on [28], 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 [4], in which we consider any intensities close the maximum intensity in the image (Almoussa et al. used 80% of max intensity, and we use $175/255 \approx 68\%$). This threshold is dependent on the image domain.

Inpainting in the above way is rather resource intensive, so we implemented two faster and less precise methods of inpainting that work well enough for removing small

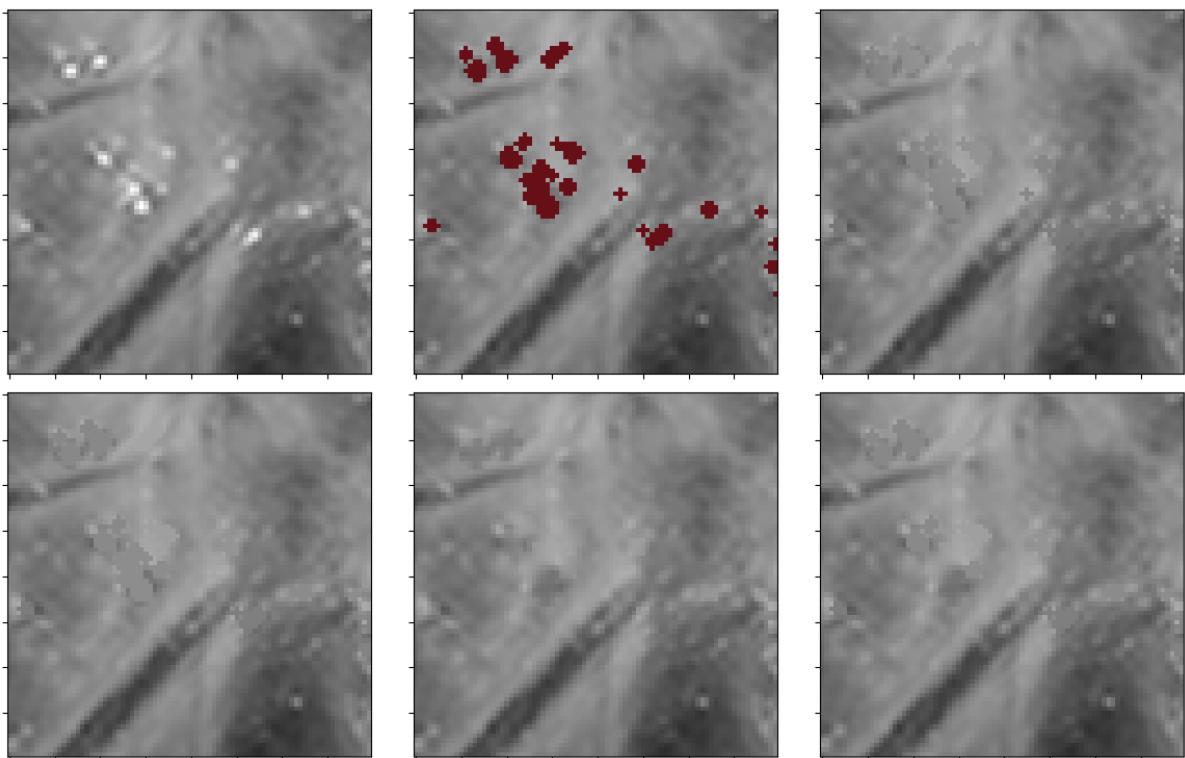


FIGURE 11: Comparison of glare inpainting methods (detail)

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

A comparison of these methods is shown in fig. 10, and a zoomed in portion is shown in fig. 11. 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]: px_inpainted = np.sum(np.logical_and(masked.mask, np.invert(img.mask)))
11 In [5]: px_plate = np.sum(np.invert(img.mask))
12 In [6]: px_inpainted / px_plate # 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 [29] 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 (i.e. vessel diameter) of a particular size. 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 direct proportionality between our scale size σ and (even some function of) the width of a particular vessel [11], 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 isolated, and the largest should be an effective size as well. In fact, following [15], it is reasonable and natural to select these logarithmically; that is, for some selected inputs $m < M$ we have

$$\sigma_1 = 2^m, \sigma_j = 2^{(m + \frac{M-m}{N-1}j)}, \sigma_N = 2^M \quad (3.2)$$

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 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.” Our final remark is that this choice of scale size is intuitively dependent on the resolution of the image.

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.

```
% DECISION TREE
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, int(4sigma)} if (sigma < 20) else 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 Hxx, Hxy, Hy
    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(ceil(sigma),10)
    f) Calculate Frangi Vesselness Measure
C) Estimate PCSVN
1) Approximate using strategy
    a) Calculate Fmax and Fmax.where -> Fmax
    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

```

We will discuss our various demonstrations of merging techniques in the Results section.

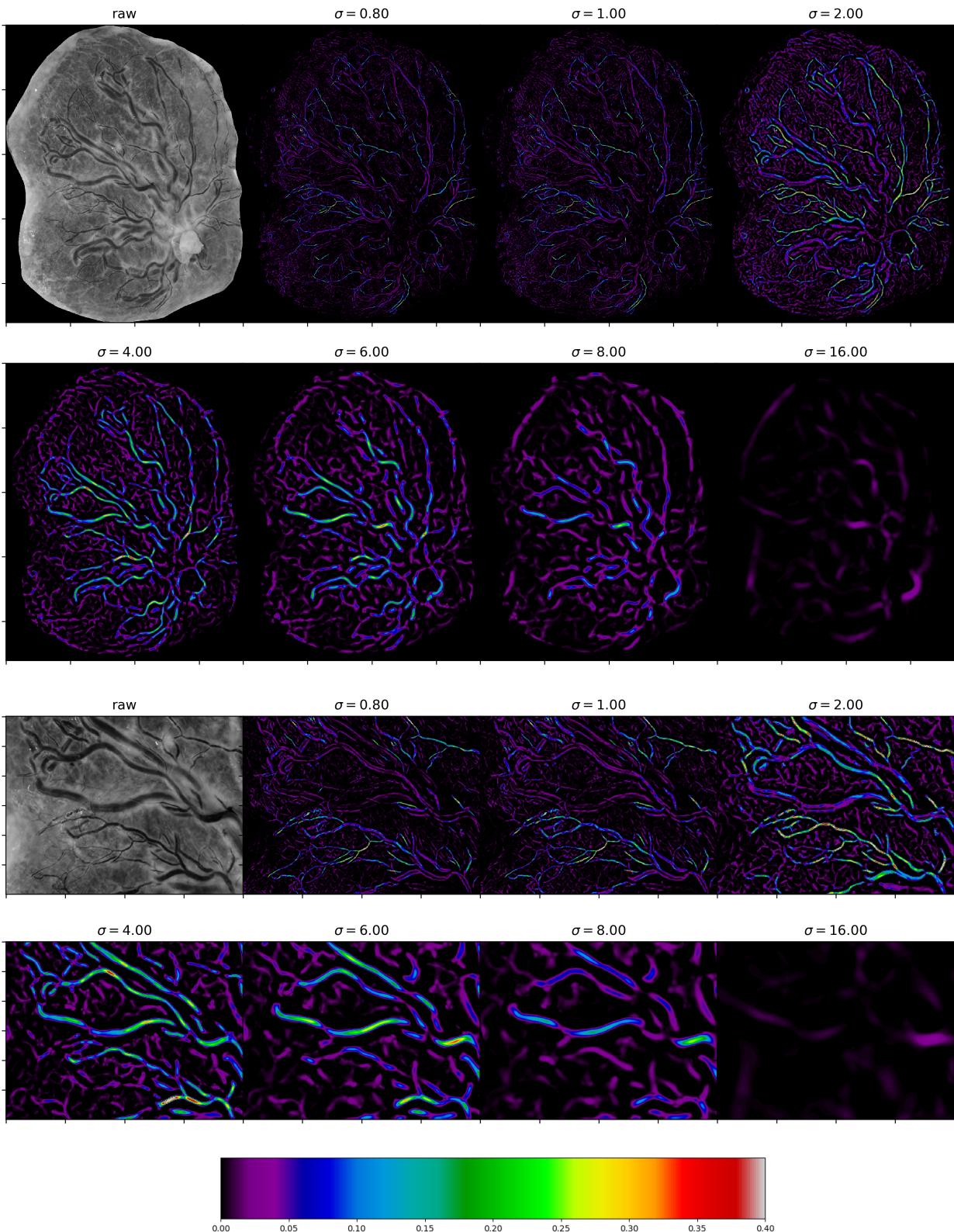


FIGURE 12: Signed Frangi output (plate and inset) (Example 1)

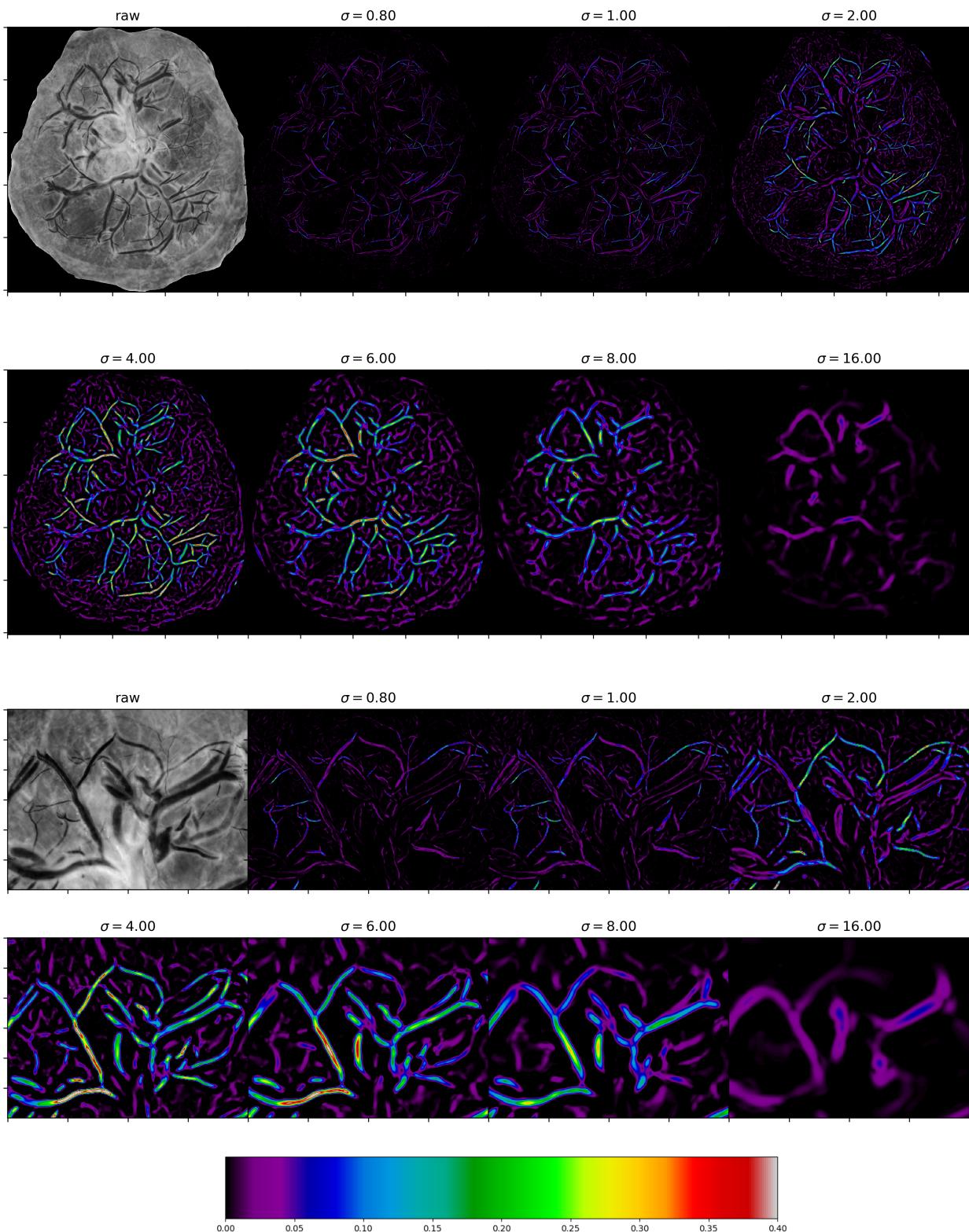


FIGURE 13: Signed Frangi output (plate and inset) (Example 1)

CHAPTER 4

IMPLEMENTATIONS

Calculating the Hessian via FFT

Efficient implementation of the Frangi filter ultimately relies on performing a 2D Gaussian blur in frequency space. Here we demonstrate that our FFT implementation of Gaussian blur is commensurate with other implementations.

In fig. 14, we demonstrate the compatibility of standard convolution and FFT convolve. Each row corresponds to a different scale at which Gaussian blurring occurs. Column (a) is standard convolution with a sampled Gaussian kernel, column (b) is FFT-convolution with a Gaussian kernel, and column (c) is a FFT-convolution with the “discrete Gaussian kernel”. In column (d), the 1D discrete Gaussian kernel (in green) is plotted against the sampled continuous Gaussian kernel (in black). Note that each of the images in the first three columns are scaled the same.

In fig. 15, we show these same three methods of Gaussian blur but for a large scale ($\sigma = 45$). For each method of taking the Gaussian blur ((a) - standard convolution with sampled kernel, (b) fft with sampled kernel, (c) fft with discrete kernel), the top row is one round of Gaussian blur with $\sigma = 45$ and the bottom row is two progressive passes of Gaussian blur ($\sigma_1 = 10, \sigma_2 = 35$). The mean squared error and mean absolute error between the one-pass and two-pass versions are outputted below. Code for this demo can be found in `hfft.semigroup_demo`. The discrete kernel performs very slightly better than the sampled versions. We originally attempted this demonstration with a much larger sigma (say $\sigma = 150$) and multiple iterations, but unfortunately multiple passes cause the “noise” from zeroing out around the boundaries to become very noticeable after

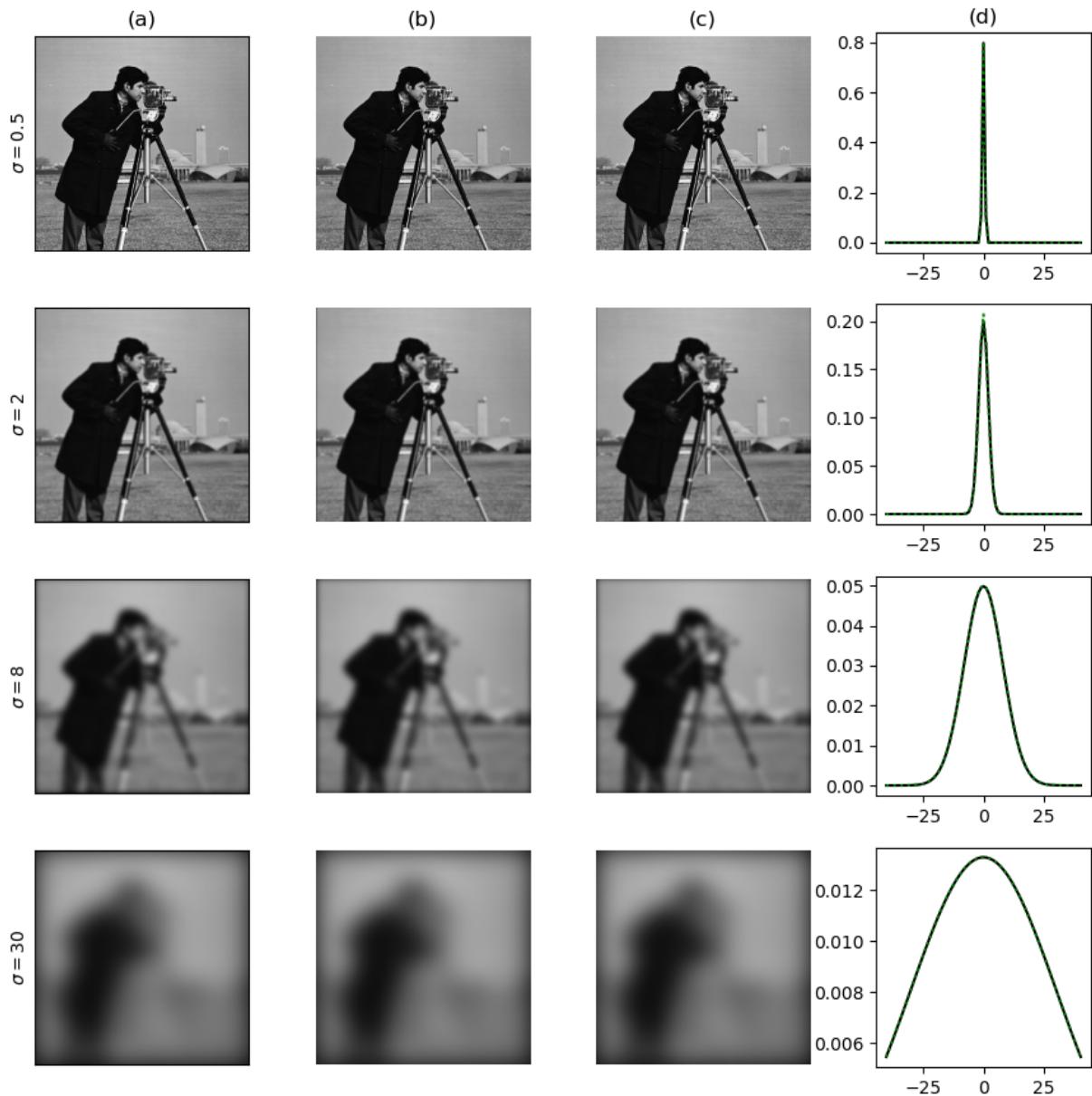


FIGURE 14: Compatibility of Gaussian convolution strategies

blurring method	MSE	MAE
spatial convolution, sampled kernel	0.00054426	0.02015643
FFT convolution, sampled kernel	0.00055205	0.02029916
FFT convolution, discrete kernel	0.00054406	0.02015336

	A	B	C
A	-	1.296e-03	6.772e-06
B	-	-	1.247e-03
C	-	-	-

TABLE 2: MSE of Gaussian blurs of an image ($\sigma = 0.3$)

	A	B	C
A	-	4.256e-06	5.537e-08
B	-	-	4.337e-06
C	-	-	-

TABLE 3: MSE of Frangi scores $\sigma = 0.3$

several iterations (here, we've opted to crop out a radius of pixels from around the edges equal to the standard deviation of the Gaussian before we calculated the MAE or MSE).

We could show this again with a zero border or maybe even just a 1D signal.

We further confirm the commensurate nature of Gaussian blur techniques by comparing the three techniques on a placental image and using each to calculate Frangi targets. The code can be found in `hfft_accuracy.py`. In table 2, table 3, table 4 and table 5 we compare the mean squared error of a single image blurred (A) with standard spatial convolution, (B) with FFT sampled Gaussian kernel, and (C) with the discrete kernel. We see that the standard convolution and discrete convolution are very similar, while the sampled discrete Gaussian is off by two orders of magnitude, but still reasonably small. We further confirm these by viewing the intensity of the images and the Frangi targets themselves across an arbitrarily chosen horizontal cross section of the image. As seen in fig. 16, fig. 17, fig. 18, fig. 19, the peaks of the Gaussian blurred image all still occur at the same places, as do the Frangi responses. We repeated this procedure up to $\sigma = 90$ and found a situation similar to $\sigma = 5$; it was only in very small scales where there was any noticeable difference at all.

Finally, we wish to demonstrate the point of this comparison—that *FFT-based* convolution is much faster than spatial convolution. We took a much larger sample

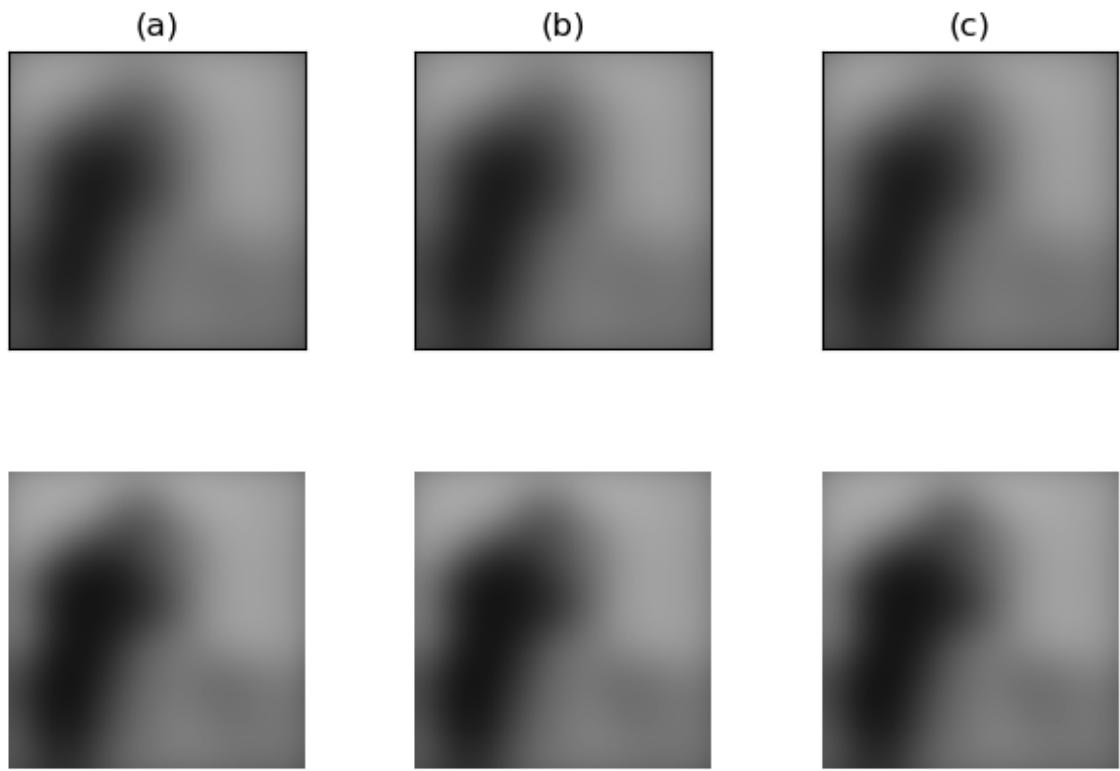


FIGURE 15: Iterative Gaussian blur

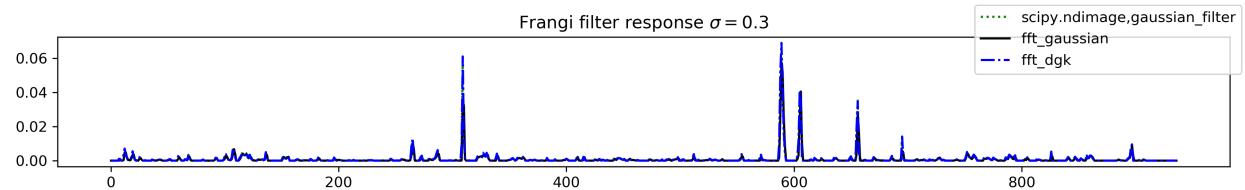


FIGURE 16: Image cross-section of Gaussian blurred images

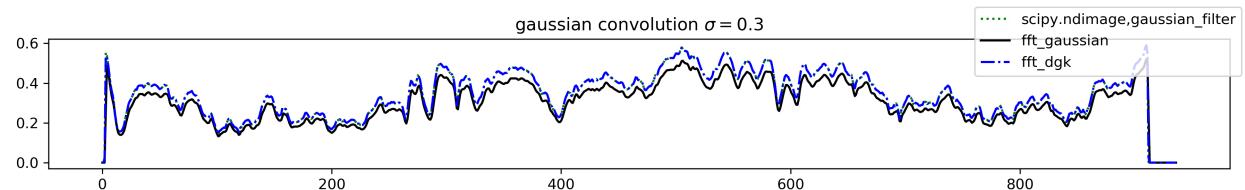


FIGURE 17: Image cross-section of Frangi targets images

	A	B	C
A	-	9.012e-06	8.629e-09
B	-	-	9.031e-06
C	-	-	-

TABLE 4: MSE of Gaussian blurs of an image ($\sigma = 5$)

	A	B	C
A	-	9.388e-05	8.383e-07
B	-	-	9.599e-05
C	-	-	-

TABLE 5: MSE of Frangi scores $\sigma = 5$

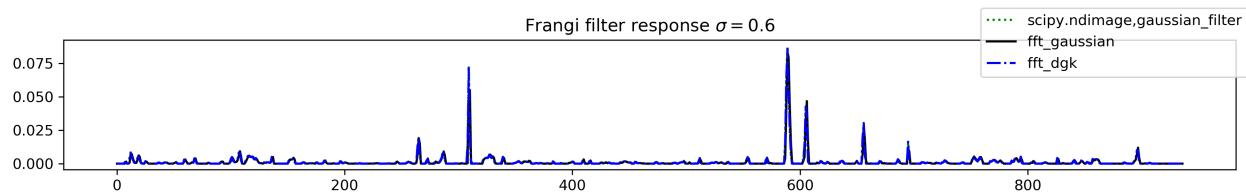


FIGURE 18: Image cross-section of Gaussian blurred images $\sigma = 5$

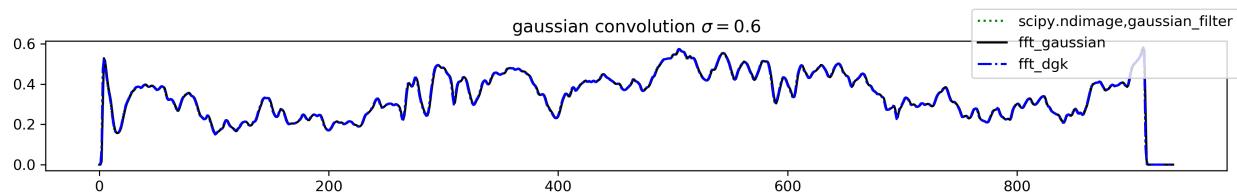


FIGURE 19: Image cross-section of Frangi targets images $\sigma = 5$

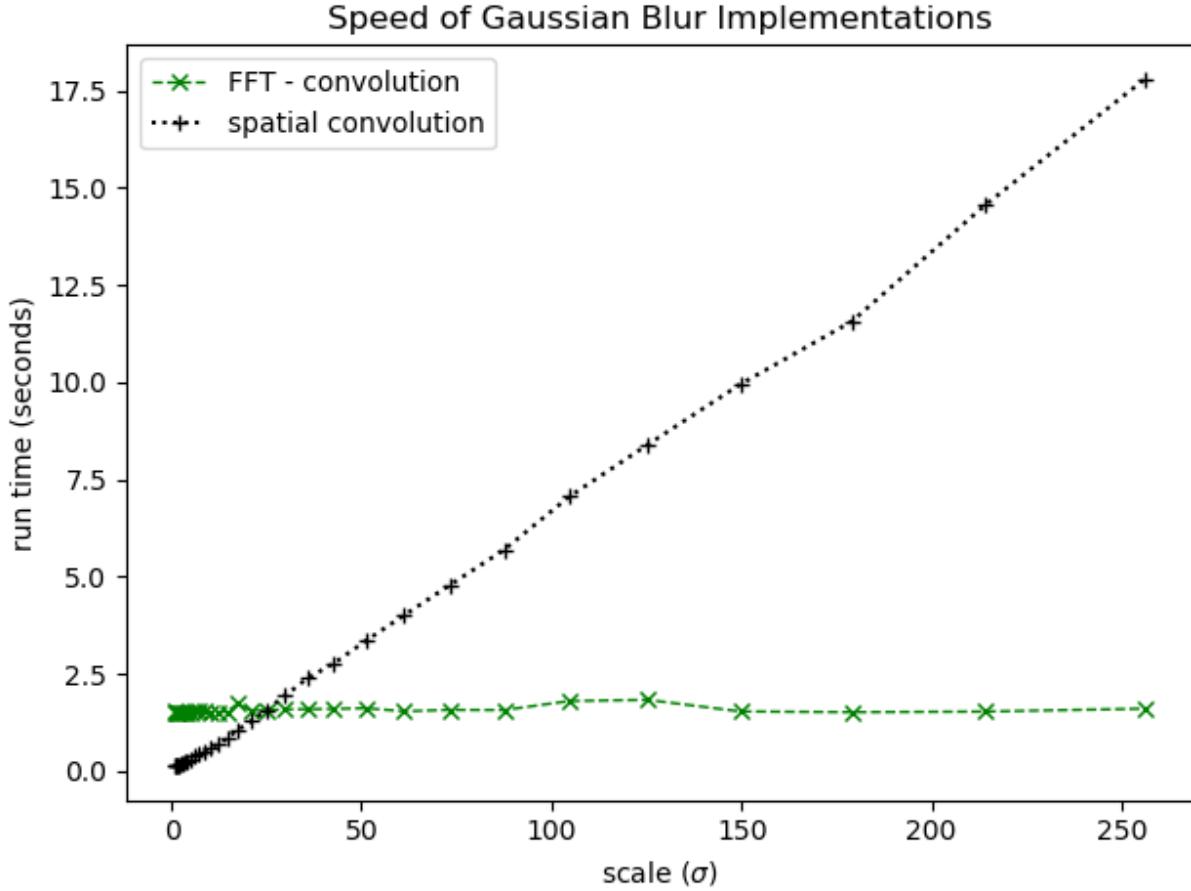


FIGURE 20: Time required

(2200×2561) and timed each method of convolution (average of three trials) for a large number of samples: logarithmic between $\sigma = 1$ and $\sigma = 128$ with 32 steps. The result shows that the convolution time seems to at least linearly increase with the size of the kernel, whereas FFT is independent of choice of scale.

Postprocessing Techniques

We display several four relatively immediate postprocessing techniques on the multiscale Frangi output to obtain an actual PCSVN extraction. Again we stress that the Frangi filter itself does not produce a segmentation, but instead could be used as a preprocessing step. In fact, Frangi in his original paper [11] refrained from any explicit

analysis of the Frangi score apart from taking the maximum across all scales, as in eq. (2.91). Still, we wish to demonstrate some several immediate methods of postprocessing these samples in order to illustrate the usefulness of this optimized Frangi filter.

Unfortunately, even with our “rescaled” Frangi filter, this α cannot be picked without regard for the particular image domain. Equally problematic, we cannot guarantee that the Frangi filter will decay as our scale exceeds the bounds where structure is expected to be found. Ideally we could create a filter that would do that.

Method A: Fixed Threshold

In the fixed threshold method, we say that a pixel (x, y) of the image corresponds to a vessel if $V_{\max} > \alpha$. This α , as noted above, is unfortunately highly dependent on the image domain, and this merging method will tend to happily allow noise generated from scales that are too large or too small. . Another issue with this is the individual scales of the Frangi filter in the extreme cases are not known to scale—although Lindeberg introduced a normalization factor based on the scale to apply to the derivatives, we do not know of an optimal factor to use.

Method B: Percentile Based Merging

The idea behind percentile-based merging is beneficial for large multiscale methods. At each scale, we would like to assume that there is *some* curvilinear content that could be identified. With that in mind, we could simply accept from each scales scores in a very high percentile. We chose for our demonstration a fairly large percentile, 95, and furthermore bolster this by requiring that any selected pixels be in the 95th percentile of nonzero and unmasked pixels—otherwise the average is artificially low due to the large background and pixels with zero Frangi score. The use of percentiles removes dependence of picking a particular threshold on the problem, while allowing the most prominent features to emerge at each scale, but of course it unfortunately treats

all scales equally, so the success of the multiscale approach here is very dependent on choice of σ_{\min} and σ_{\max} .

Method C: Scale-Based Random Walker

We observed that areas where Frangi scores are zero in well-behaved samples seem to neatly outline prominent vascular features. Following this idea, we employed a random walker segmentation [30] (which is implemented by `scikit-image`). Random walk segmentation comes about by solving a diffusion problem over a discrete array (in this case, the Frangi vesselness score itself) given starting markers. At each scale, we positively labeled pixels whose Frangi score was very high ($V_\sigma(x_0, y_0) > .4$), and negatively labeled pixels whose score was 0 (i.e. where the leading principal eigenvalue was positive). The result of this technique is demonstrated in ?? and the result (along with the original sample for comparison) is shown in fig. 22. In ??, the first column is the Frangi vesselness score at that scale, where black is a score of 0, to emphasize the difference between a score of zero and even a very small positive score, which appear in blue. The middle score are markers passed to the random walker—blue are seeds labelled with a “1” (where the Frangi vesselness score is 0), green is labeled “2” (where $V > .4$), and purple represents unknowns that will either assigned either label. In the last column, the result of the random walker is given—areas that have been added to the label “2” are shown in yellow. Although the result of random walker segmentation is technically a binary matrix, we still show the original seeds of label 2 in green for easier comparison. Similarly, the purple in the right column has actually been labeled “1” for non-vascular, but is left in its original color to emphasize what was assigned background. In fig. 22 we show the original image and the result of merging all positively marked pixels at each scale. Black means the pixel was unmatched, while increasing colors of blue (larger scales) to white (smaller scales) indicate the smallest scale from which a pixel was marked after the random walker technique. Though we

shall set up the multiscale method slightly differently in chapter 5, we used a Frangi anisotropy coefficient of $\beta = 0.35$, and 12 scales logarithmically spaced from $\sigma_1 = 2^{-1.5}$ to $\sigma_{12} = 2^{3.5}$ to generate these figures. There is a coefficient (also called β) which serves as a diffusion penalization coefficient (larger values making diffusion over the image less likely). We used scikit-image's default value of 130.

Method D: Scale Based Sieving

Our final approach seeks to include not only pixels at each scale that pass a high threshold, but also adjacent pixels at that scale that pass a lower threshold. We proceed as follows. At each scale, take a low threshold, then label each connected region. Then, iterate through each labeled region and add to the final approximation any labeled region that contains a pixel that passes a higher threshold.

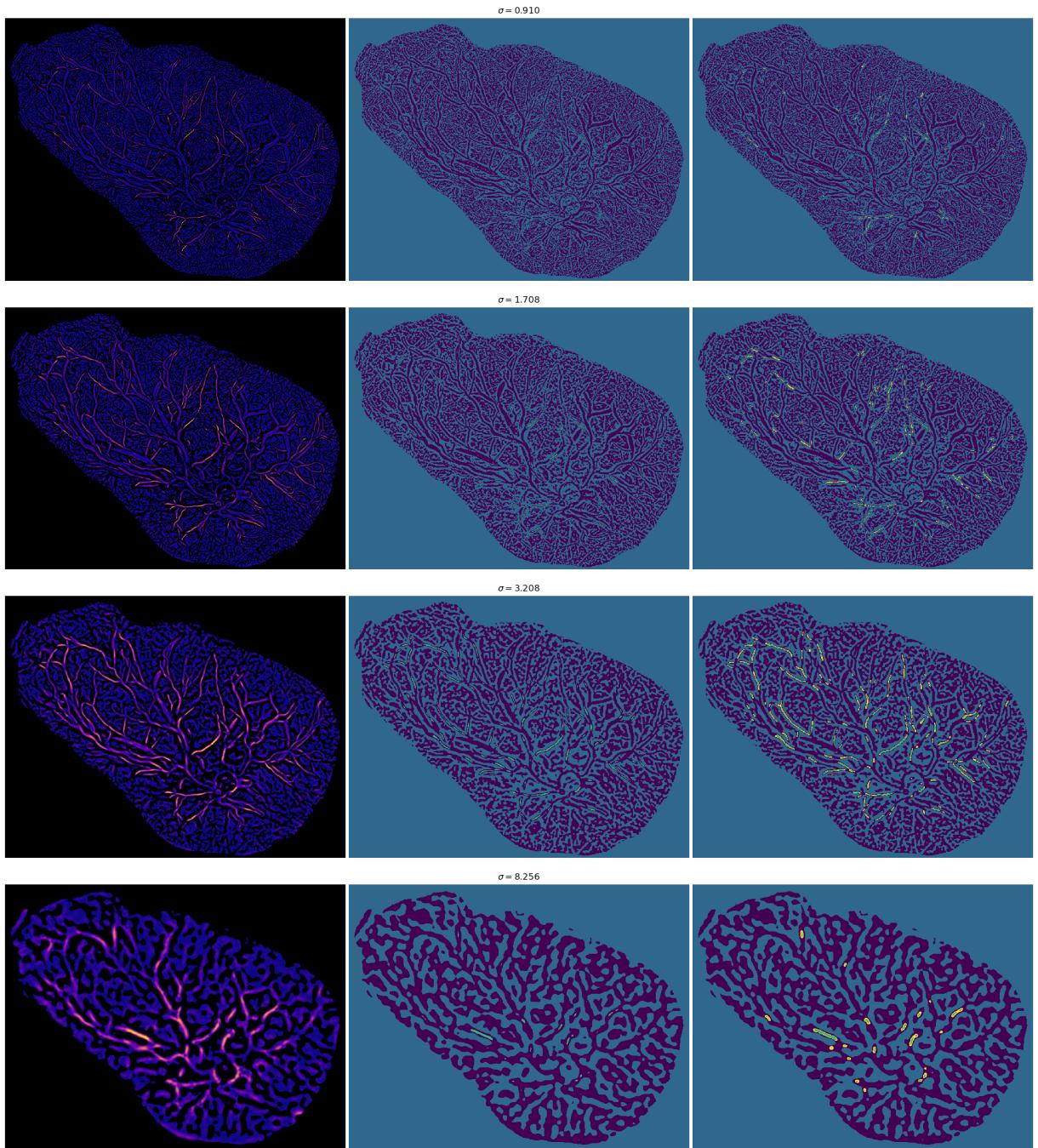


FIGURE 21: Scale-wise random walker segmentation (select scales)



FIGURE 22: Random walker segmentation (result and sample)

CHAPTER 5

RESULTS AND ANALYSIS

We demonstrate the output of the Frangi filter on our samples after running a multiscale technique with $N = 20$ scales with a stricter anisotropy $\beta = .35$, with scales spaced logarithmically from $\sigma_1 = 2^{-1}$ to $\sigma_N = 2^{3.5}$, performing glare and cut removal in preprocessing, and using a discrete gaussian kernel and dilation border of 20.

Sample visual output

In fig. 23 and fig. 24 we take a partial look at the Frangi output for two particularly well-behaved samples. In the top-left, the preprocessed placenta is shown. In the top-right, the maximum of the Frangi output over N scales. The bottom left and right images are simple segmentation strategies of merging the result.

Binary Classifications and the confusion matrix

Here, we demonstrate the visual outputs produced by `extract_NCS_pcsvn.py`. This particular sample, BN4569506, is a relatively well-behaved sample, and segmentation was comparatively successful.

use MCC [31]

Variations in the Data Set and Imperfections of the Ground Truth

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.

As seen in fig. 26, there are several issues with the samples that will cause trouble in our efforts toward segmentation. Our representative sample is BN0392644. The top left is the original (color) image, the top right is the full vessel width trace. The bottom left is

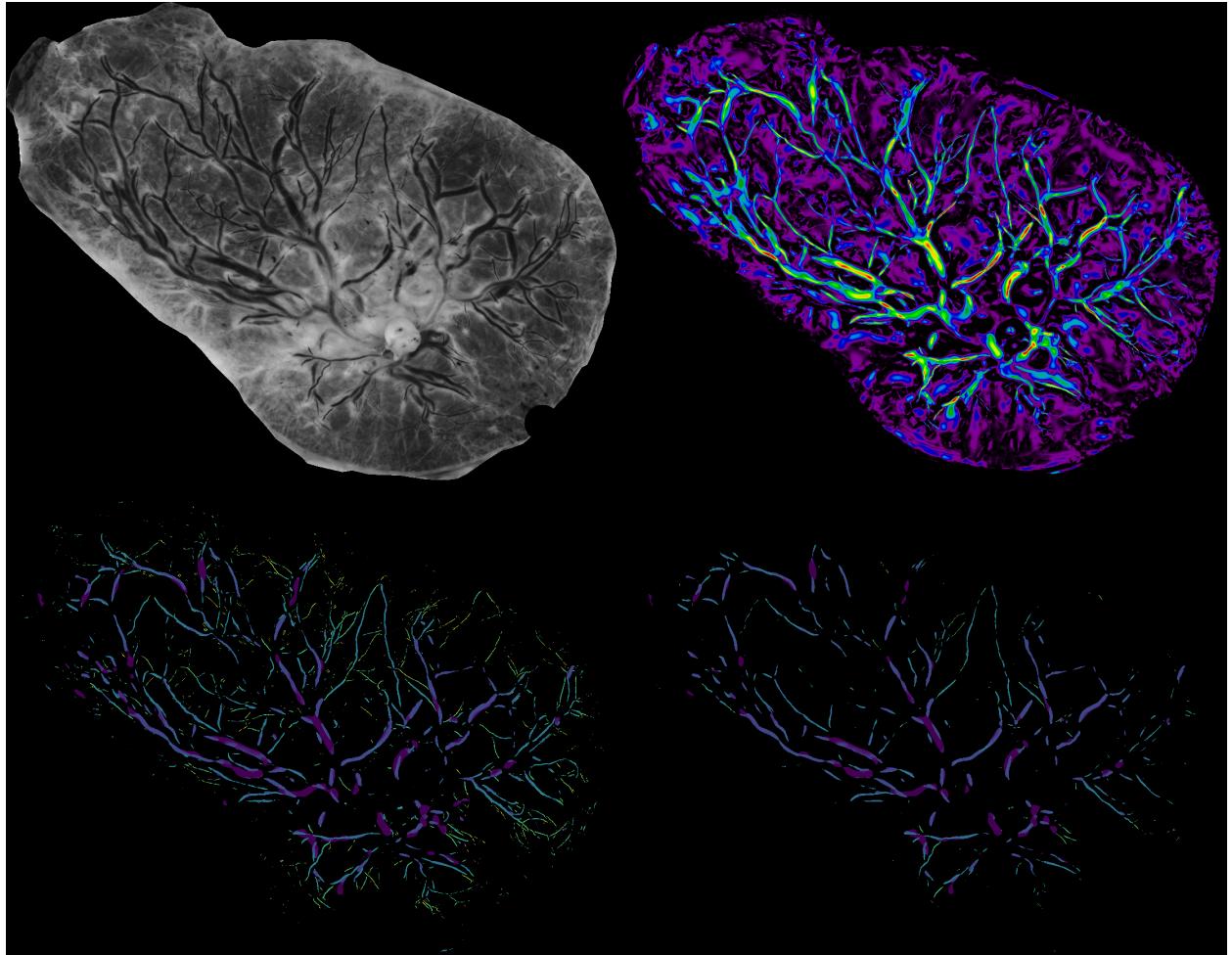


FIGURE 23: Sample Multiscale Frangi output ($\beta = 0.35$) with simple segmentation strate-gies (Example 1)

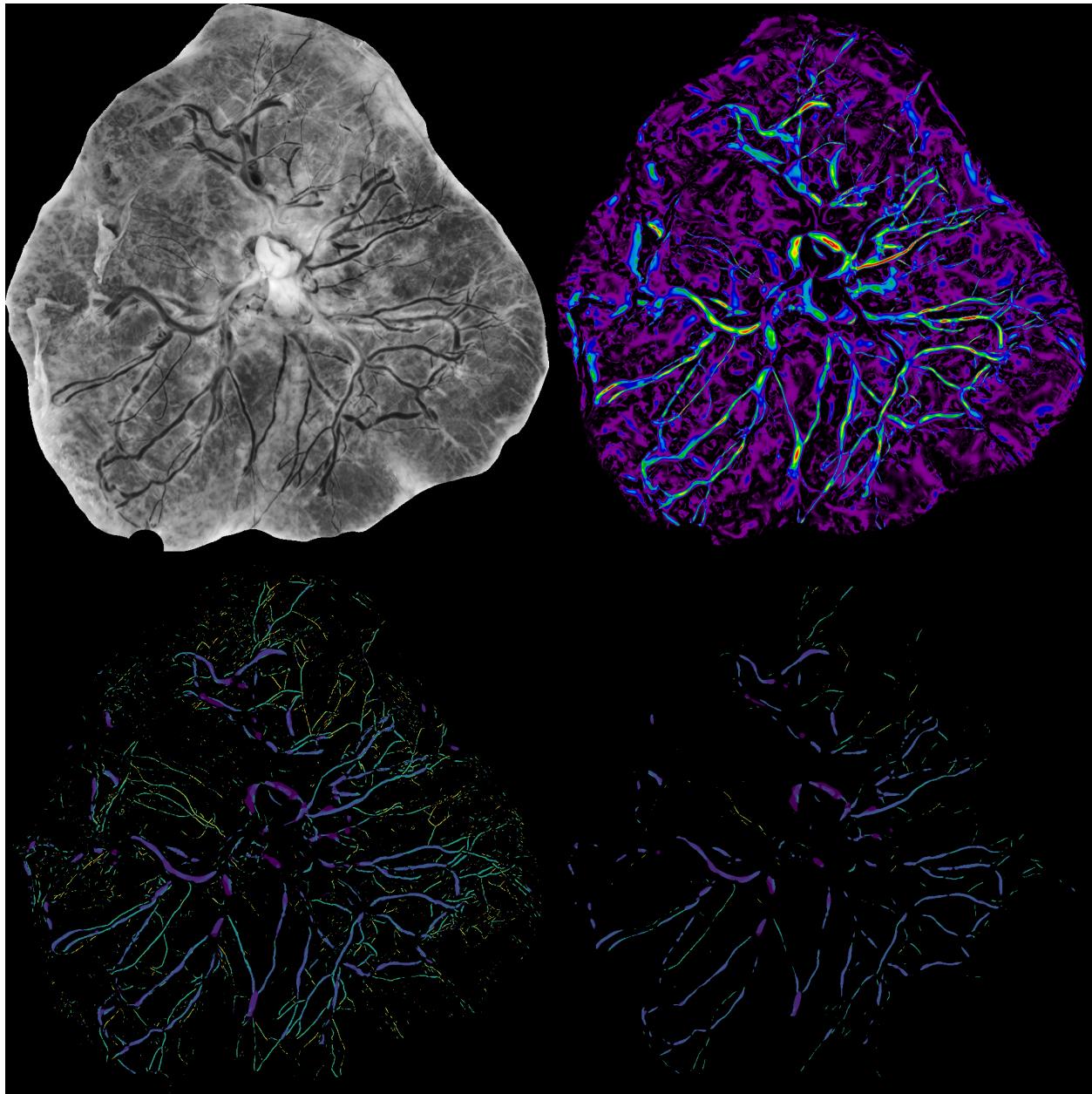


FIGURE 24: Sample Multiscale Frangi output ($\beta = 0.35$) with simple segmentation strategies (Example 2)

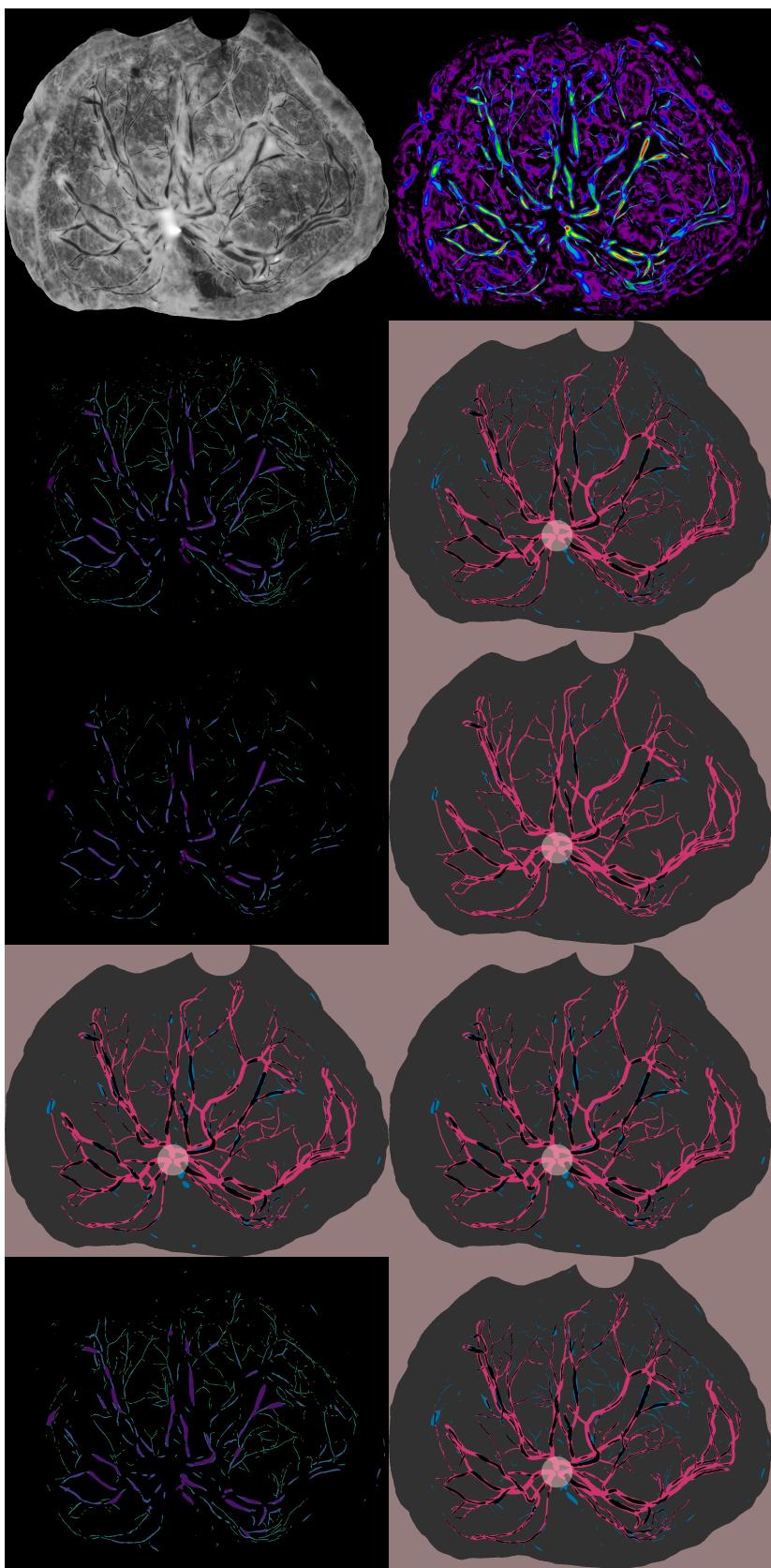


FIGURE 25: Demonstration of postprocessing techniques
83

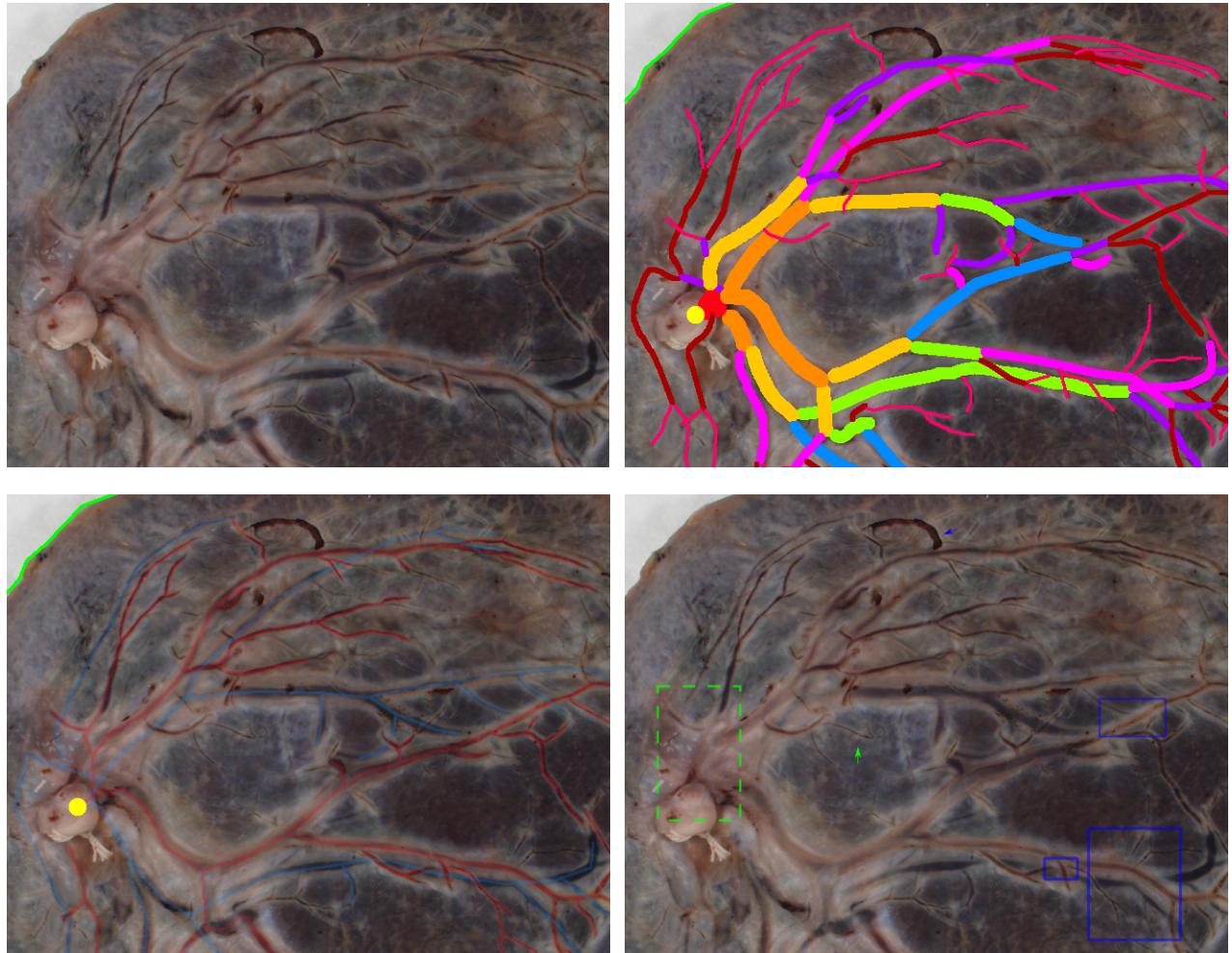


FIGURE 26: Issues with ground truth and sample quality

a smaller skeletonization (sketch), where arteries are shown in red and veins are shown in blue. The bottom right figure contains some annotations. At the top, a blue arrow indicates a large curvilinear patch of dried blood that is not part of the vascular network. The green arrow in the middle indicates some vessels that are too small for the diameter binning and are thus not reported. We will see later that our Frangi result perfectly captures these, yet they will be reported as a false positive since they are not part of the tracing. However, there are other vessels of similar visual width in this same inset that are traced. In blue boxes (and in many other spots) the vessels cross each other. The border around these will prevent us from being able to extract the vessel directly. In the green dotted box, a major arterial and a major venous branch each connect to the umbilical cord insertion point. Whereas the arterial branch (on the right) can be seen, it will not be reported by the Frangi filter, since those points are not darker relative to the background. You can also see how much variation there is as you look along a blood vessel. There are some areas where the Frangi filter will have a very limited response.

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.

	PF (q=95)	FA (alpha=0.4)	RW	PS
MCC	0.505366	0.434502	0.511190	0.505440
% skeltrace coverage	0.498603	0.306664	0.411479	0.527201
precision	0.857878	0.906386	0.821095	0.879843

TABLE 6: various metrics for gauging success of Frangi thresholding demos

4. Umbilical cord insertion point is stupid and obscures a lot. The tracer guesses but 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

We justified the use of differential geometry in 2D discrete image processing, and vastly improved upon the implementation of the Frangi filter. Our improved implementation allowed us to take more steps in our multiscale method and thus choose stricter parameters for Frangi scale. We used our multiscale Frangi vesselness measure to suggest several alternative approaches at merging the vesselness and compared their effectiveness as a precursor to segmentation and eventually network completion.

Future research directions

Our goal is to eventually solve the network connection problem.

- Solve the Network Connection Problem (PICTURE OF GAPS) Try something like [32] or use of principal curvatures.
- Implement the automatic scale selection and normalization of derivates as mentioned in Lindeberg [23] to relieve ourselves of our current dependency on manual selection of σ_{\min} and σ_{\max} .
- Look into gradient prefiltering more as well as varying γ more, especially in areas where we suspect the network could be completed.
- Look into using signed frangi arguments.
- Use this as preprocessing for a Neural Network. (cite kara's work, katalinas work)
- Apply to more image domains (STARE, other placental domains).

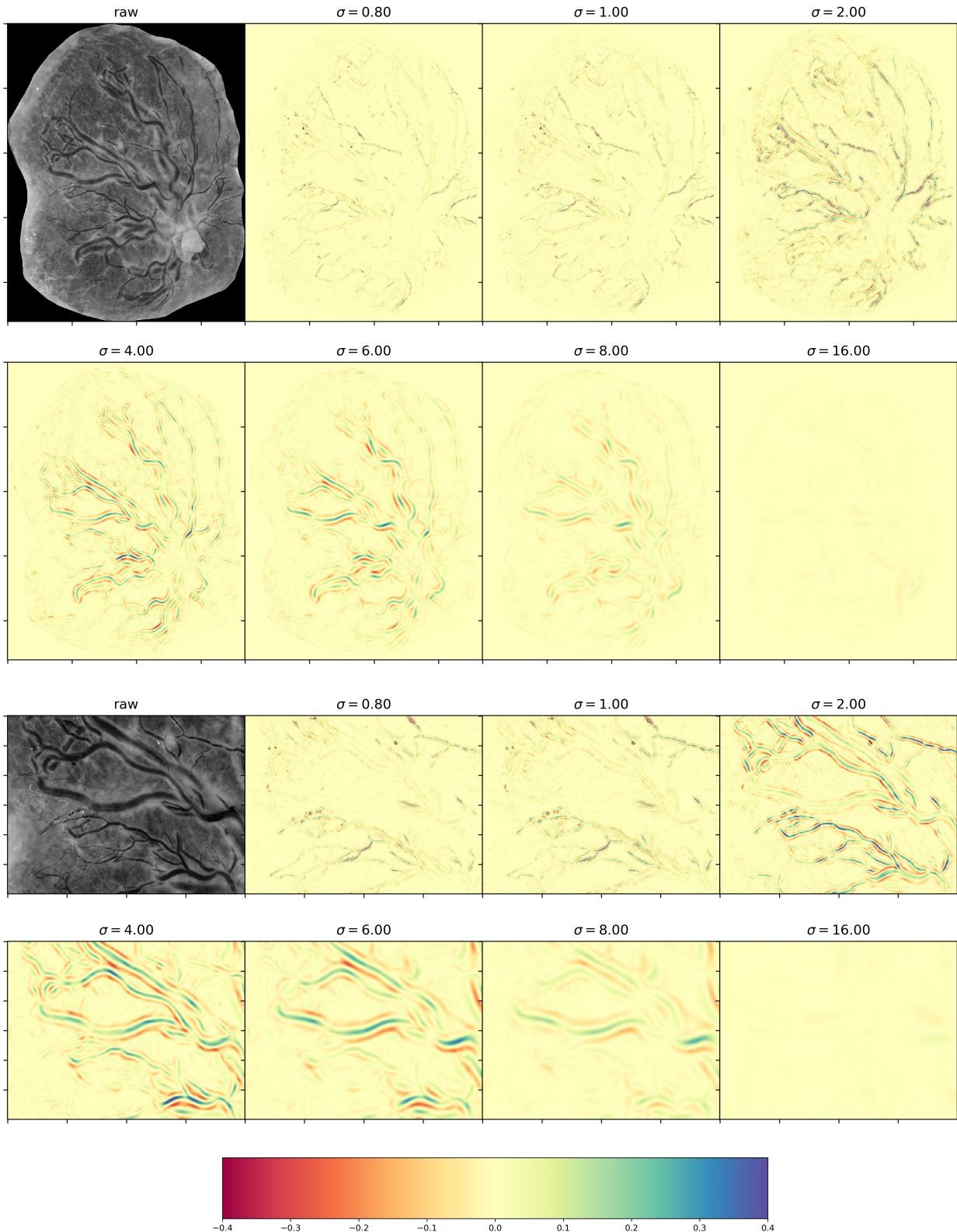


FIGURE 27: Signed Frangi output (plate and inset) (Example 1)

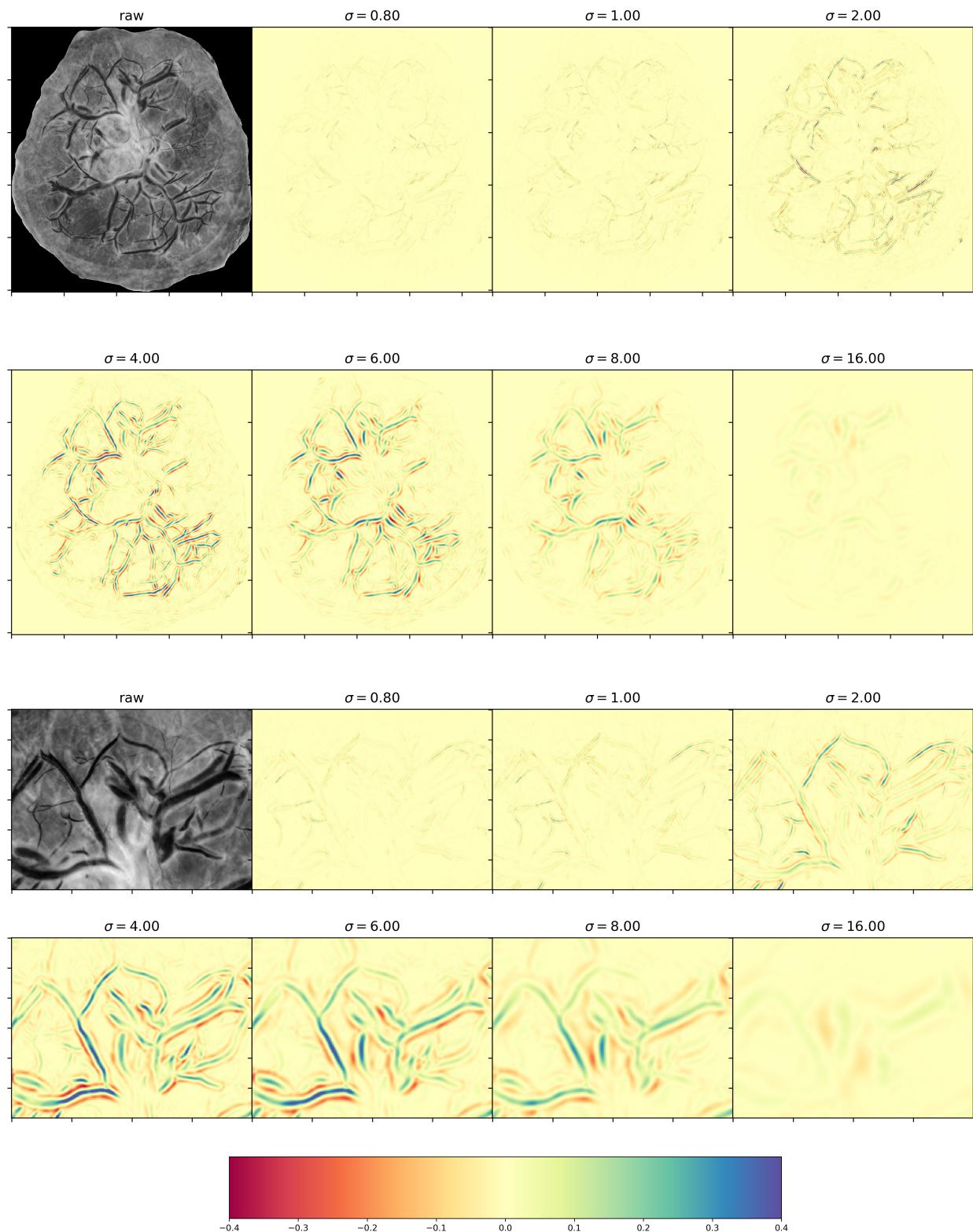


FIGURE 28: Signed Frangi output (plate and inset) (Example 1)

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/cut_demo.py

```
1 #!/usr/bin/env python3
2
3 import numpy as np
4 import matplotlib.pyplot as plt
5
6 from placenta import open_typefile, list_placentas, get_named_placenta
7 from plate_morphology import mask_cuts_simple, dilate_boundary
8
9 from skimage.color import gray2rgb
10 from skimage.morphology import thin, binary_dilation, disk, square
11 import numpy.ma as ma
12 import os.path
13
14 def l2_dist(p,q):
15     return int(np.round(np.sqrt((p[0]-q[0])**2 + (p[1]-q[1])**2)))
16
17 placentas = list_placentas('T-BN')
18 samples_with_cuts = list()
19
20 for filename in list_placentas('T-BN'):
21
22     # this one has two cuts, another one has two cuts as well
23     #if filename != "T-BN2459820.png":
24     #    continue
25
26     img = get_named_placenta(filename)
27     ucip = open_typefile(filename, 'ucip')
28
29     #C, has_cut = mask_cuts(img, ucip, return_success=True, in_place=False)
30
31     #if has_cut:
32
33         # dilcut = img.copy()
34
35         # print(filename, "has a cut!")
36         # samples_with_cuts.append(filename)
37
38         # B = np.all(ucip==(0,0,255), axis=-1)
39         # G = np.all(ucip==(0,255,0), axis=-1)
40
41         # cutmarks = np.nonzero(thin(B))
42         # perimeter = np.nonzero(G)
43
44         # #for array in the tuple that comes out of np.nonzero(thin(B))
45         # # or just one if it's just a single thing i guess?
46
47         # # the x, y points of the cutmarks are in columns
48         # cutinds = np.stack(cutmarks)
49
50         # for P in cutinds.T:
51
52             # consider larger and larger window sizes
53             # for W in [100,200,300]:
54             #     # consider all perimeter elements within these bounds
55
56             #     rmin, rmax = max(0, P[0]-W), min(img.shape[0], P[0]+W)
57             #     cmin, cmax = max(0, P[1]-W), min(img.shape[1], P[1]+W)
58             #     window = np.s_[rmin:rmax, cmin:cmax]
59
60             #     # perimeter indices within the window
61             #     pinds = [(x,y) for x, y in zip(*perimeter)
62             #             if x > rmin and x < rmax and y > cmin and y < cmax]
```

```

63     #
64     #           if pinds:
65     #               break
66     #       if pinds:
67
68     #           # max distance to boundary point in the window
69     #           # we really only need to keep the largest; deque?
70     #           dists = sorted([(pp, l2_dist(P, pp)) for pp in pinds],
71     #                           key=lambda t: t[1])
72     #           r = int(dists[-1][1]) + 1 # get largest radius but closest point
73     #           P = dists[0][0]
74     #           B = np.zeros_like(img.mask)
75
76     #           B[cutmarks] = True
77
78     #           # center a disk of found radius there
79     #           D = disk(r)
80     #           winx = max(P[0]-r,0), min(P[0]+r+1,B.shape[0])
81     #           winy = max(P[1]-r,0), min(P[1]+r+1,B.shape[1])
82     #           try:
83     #               B[winx[0]:winx[1], winy[0]:winy[1]] = D
84     #           except ValueError:
85     #               # they're out of bounds so it's a size mismatch. fix it
86     #               # by starting/ending D index with opposite sign of the initial
87     #               # p +/- radius that was out of bounds
88     #               # for example P[0]-r was -9 and everything else was fine
89     #               # so you just need to set left side to D[9:,:]
90     #               # but you should wrap this up in a function so the three times
91     #               # you do it here and the one time in ucip all gets the same
92     #               # code
93     #               pass
94     #           dilcut[B] = ma.masked
95
96     #       else:
97     #           # this is probably not going to happen, but just in case no
98     #           # nearby perimeter was found, just... give up
99     #           pass
100
101    #       rminv, rmaxv = max(0, rmin-W//2), min(img.shape[0], rmax+W//2)
102    #       cminv, cmaxv = max(0, cmin-W//2), min(img.shape[1], cmax+W//2)
103    #       view = np.s_[rminv:rmaxv, cminv:cmaxv]
104    #       montage = np.hstack((gray2rgb(img.filled(0)[view]),
105    #                             ucip[view],
106    #                             gray2rgb(C.filled(0)[view]),
107    #                             gray2rgb(dilcut.filled(0)[view])))
108    #       filestub, _ = os.path.splitext(filename)
109    #       plt.imsave(f'demo_output/cut_demo/{filestub}_cutopts.png', montage)
110    #       #plt.imshow(montage)
111    #       plt.show()
112    #       plt.close()
113    mimg, success = mask_cuts_simple(img, ucip, return_success=True)
114    if success:
115        montage = np.hstack((img.filled(0),
116                             mimg.filled(0)))
117        plt.imshow(montage)
118        plt.show()
119        plt.close()
120    print("*"*80)
121    print(f"there were {len(placentas)} total samples",
122          f"and {len(samples_with_cuts)} of them had cuts")

```

listings/diffgeo.py

```

1 #!/usr/bin/env python3
2
3 import numpy as np
4 import numpy.ma as ma
5
6 from skimage.feature import hessian_matrix, hessian_matrix_eigvals
7 from numpy.linalg import eig
8
9
10 def principal_curvatures(img, sigma=1.0, H=None):
11     """Calculate the approximated principal curvatures of an image
12
13     Return the (approximated) principal curvatures { 1 , 2 } of an image,
14     that is, the eigenvalues of the Hessian at each point (x,y). The output
15     is arranged such that | 1 | <= | 2 |. Note that the Hessian of the image,
16     if not provided, is computed using skimage.feature.hessian_matrix, which
17     can be very slow for large sigmas.
18
19     Parameters
20     -----
21     img: array or ma.MaskedArray
22
23         An ndarray representing a 2D or multichannel image. If the image is
24         multichannel (e.g. RGB), then each channel will be processed
25         individually. Additionally, the input image may be a masked array-- in
26         which case the output will preserve this mask identically.
27
28     sigma: float, optional
29         Standard deviation of the Gaussian (used to calculate the hessian
30         matrix).
31     H: list of array, optional
32         The hessian itself (Hxx,Hxy,Hyy) whose eigenvalues will be calculated.
33         Use this option if you're going to calculate the Hessian using faster
34         means, e.g. via FFT.
35
36     Returns
37     -----
38     (K1, K2): tuple of arrays
39         K1, K2 each are the exact dimension of the input image, ordered in
40         magnitude such that | 1 | <= | 2 | in all locations.
41
42     Examples
43     -----
44     >>> K1, K2 = principal_curvatures(img)
45     >>> K1.shape == img.shape
46     True
47     >>> (K1 <= K2).all()
48     True
49     >>> K1.mask == img.mask
50     True
51     """
52
53     # determine if multichannel
54     multichannel = (img.ndim == 3)
55
56     if not multichannel:
57         # add a trivial dimension
58         img = img[:, :, np.newaxis]
59
60     K1 = np.zeros_like(img, dtype='float64')
61     K2 = np.zeros_like(img, dtype='float64')
62
63     for ic in range(img.shape[2]):
```

```

64     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 = reorder_eigs(L)
73
74     # Make K2 larger in magnitude, as consistent with Frangi paper
75     K1[:, :, ic] = L[0, :, :]
76     K2[:, :, ic] = L[1, :, :]
77
78 try:
79     mask = img.mask # get mask to add to each if input was a masked array
80
81 except AttributeError:
82     pass # there's no mask, so do nothing
83
84 else:
85     K1 = ma.masked_array(K1, mask=mask)
86     K2 = ma.masked_array(K2, mask=mask)
87
88 # now undo the trivial dimension
89 if not multichannel:
90     K1 = np.squeeze(K1)
91     K2 = np.squeeze(K2)
92
93 return K1, K2
94
95
96
97 def reorder_eigs(L):
98     """reorder eigenvalues by decreasing magnitude.
99
100    Eigenvalues are outputted from hessian_matrix_eigvals so that L1 >= L2.
101    This reorders this so that |L1| >= |L2| instead (where L1,L2=L)
102    Parameters
103    -----
104    L: ndarray or iterable of ndarrays
105        As outputted by, say, hessian_matrix_eigs. If a single ndarray, it
106        should be the shape (N, *img.shape) where there are N eigenvalues to
107        reorder. You may also input a tuple like (L1,L2).
108    Returns
109    -----
110    eigs: ndarray
111        The eigenvalues in decreasing order of magnitude; that is
112        eigs[i,j,k] is the ith-largest eigenvalue at position (j, k).
113        Each of these is the same shape the original inputs, but
114        np.abs(L1r) >= np.abs(L2r) will be true. See warning below.
115
116
117 Warnings / Notes
118 -----
119 Please note the order! Outputs are given in *decreasing* magnitude. This is
120 done to align with the behavior of skimage.feature.hessian_matrix_eigvals,
121 but if you want to label them according to the Frangi filter (where k2
122 denotes the *larger* magnitude eigenvalue, you should reverse the labels:
123
124     >>>k2, k1 = reorder_eigs(L) # k2, k1 as frangi labeled them
125     >>>np.all(np.abs(k2) >= np.abs(k1))
126     True
127

```

```

128
129 It doesn't actually matter the order in which inputs are inputted (they
130 will be sorted the same regardless).
131
132 Example
133 -----
134 >>>K1,K2 = hessian_matrix_eigvals(H)
135 >>>(K1 >= K2).all()
136 True
137 >>>(np.abs(K1) <= np.abs(K2)).all()
138 False
139 >>>K1r, K2r = reorder_eigs(K1,K2)
140 >>>(K1r <= K2r).all()
141 False
142 >>>(np.abs(K1r) <= np.abs(K2r)).all()
143 True
144
145 TODO
146 -----
147 Support out= keyword
148 """
149 # this will do nothing if L is already an array but will make it an array
150 # if it's a tuple/list/iterable
151 L = np.stack(L)
152 mag = np.argsort(np.abs(L), axis=0)
153
154 # now L2 is larger in absolute value, as consistent with Frangi paper
155 return np.take_along_axis(L, mag, axis=0)
156
157
158 def principal_directions(img, sigma, H=None, mask=None):
159     """Calculate principal directions of
160     will ignore calculation of principal directions of masked areas
161
162     mask should be positive where the PD's should *NOT* be calculated
163     this function actually returns the theta corresponding to
164     leading and trailing principal directions, i.e. angle w / x axis
165     """
166
167     if H is None:
168         H = hessian_matrix(img, sigma)
169
170     Hxx, Hxy, Hyy = H
171
172
173     # determine if there was a supplied mask or use images if it exists
174     if mask is None:
175         try:
176             mask = img.mask
177         except AttributeError:
178             masked = False
179         else:
180             masked = True
181     else:
182         masked = True
183
184     dims = img.shape
185
186     # where to store
187     trailing_thetas = np.zeros_like(img, dtype='float64')
188     leading_thetas = np.zeros_like(img, dtype='float64')
189
190     # maybe implement a small angle correction
191     for i, (xx, xy, yy) in enumerate(np.nditer([Hxx, Hxy, Hyy])):

```

```

192
193     # grab the (x,y) coordinate of the hxx, hxy, hyy you're using
194     subs = np.unravel_index(i, dims)
195
196     # ignore masked areas (if masked array)
197     if masked and mask[sub]:
198         continue
199
200     h = np.array([[xx, xy], [xy, yy]]) # per-pixel hessian
201     l, v = eig(h) # eigenvectors as columns
202
203     # reorder eigenvectors by (increasing) magnitude of eigenvalues
204     v = v[:, np.argsort(np.abs(l))]
205
206     # angle between each eigenvector and positive x-axis
207     # arccos of first element (dot product with (1,0) and eigvec is already
208     # normalized)
209     trailing_thetas[sub] = np.arccos(v[0, 0]) # first component of each
210     leading_thetas[sub] = np.arccos(v[0, 1]) # first component of each
211
212     if masked:
213         leading_thetas = ma.masked_array(leading_thetas, mask)
214         trailing_thetas = ma.masked_array(trailing_thetas, mask)
215
216     return trailing_thetas, leading_thetas
217
218
219 if __name__ == "__main__":
220     pass
221
222
223     #from get_base import get_preprocessed
224     #import matplotlib.pyplot as plt
225     #from functools import partial
226     #from fpd import get_targets
227     #b = partial(plt.imshow, cmap=plt.cm.Blues)
228     #sp = partial(plt.imshow, cmap=plt.cm.spectral)
229     #s = plt.show
230
231     #import time
232
233     #img = get_preprocessed(mode='G')
234
235     #for sigma in [0.5, 1, 2, 3, 5, 10]:
236
237     #    print('-'*80)
238     #    print(' =', sigma)
239     #    print('calculating hessian H')
240
241     #    tic = time.time()
242     #    H = hessian_matrix(img, sigma=sigma)
243
244     #    toc = time.time()
245     #    print('time elapsed: ', toc - tic)
246     #    tic = time.time()
247     #    print('calculating hessian via FFT (F)')
248     #    h = fft_hessian(img, sigma)
249
250     #    toc = time.time()
251     #    print('time elapsed: ', toc - tic)
252     #    tic = time.time()
253     #    print('calculating principal curvatures for ={}'.format(sigma))
254     #    K1,K2 = principal_curvatures(img, sigma=sigma, H=H)
255     #    toc = time.time()

```

```

256     #     print('time elapsed: ', toc - tic)
257     #     tic = time.time()
258     #     print('calculating principal curvatures for  ={} (fast)'.format(sigma))
259     #     k1,k2 = principal_curvatures(img, sigma=sigma, H=h)
260
261     #     toc = time.time()
262     #     print('time elapsed: ', toc - tic)
263     #     tic = time.time()
264
265     ######
266
267     #     print('calculating targets for  ={}'.format(sigma))
268     #     T = get_targets(K1,K2, threshold=False)
269
270     #     toc = time.time()
271     #     print('time elapsed: ', toc - tic)
272     #     tic = time.time()
273
274     #     print('calculating targets for  ={} (fast)'.format(sigma))
275     #     t = get_targets(k1,k2, threshold=False)
276
277     #     toc = time.time()
278     #     print('time elapsed: ', toc - tic)
279
280     ######
281
282     #     print('extending masks')
283
284     #     # extend mask over nontargets items
285     #     img1 = ma.masked_where( T < T.mean(), img)
286     #     img2 = ma.masked_where( t < t.mean(), img)
287
288     #     tic = time.time()
289     #     print('calculating principal directions for  ={}'.format(sigma))
290     #     T1,T2 = principal_directions(img1, sigma=sigma, H=H)
291     #     toc = time.time()
292     #     print('time elapsed: ', toc - tic)
293     #     tic = time.time()
294
295     #     print('calculating principal directions for  ={} (fast)'.format(sigma))
296     #     t1,t2 = principal_directions(img2, sigma=sigma, H=h)
297     #     toc = time.time()
298     #     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 PCCSVN of a list of samples.
5 It does not do network completion.
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_ucip_to_mask, measure_ncs_markings)
12
13 from merging import nz_percentile, apply_threshold, sieve_scales, view_slices
14
15 from scoring import (compare_trace, rgb_to_widths, merge_widths_from_traces,
16                      filter_widths, mcc, confusion, skeletonize_trace)
17

```

```

18 from pcsvn import extract_pcsvn, scale_label_figure, get_outname_lambda
19 from preprocessing import inpaint_hybrid
20
21 import numpy as np
22 import numpy.ma as ma
23
24 import matplotlib.pyplot as plt
25
26 import os.path
27 import os
28 import json
29 import datetime
30 import pandas
31
32 # for some post_processing, this needs to be moved elsewhere
33 from skimage.filters import sobel
34 from frangi import frangi_from_image
35 from plate_morphology import dilate_boundary, mask_cuts_simple
36 from skimage.morphology import remove_small_holes, remove_small_objects
37 from skimage.segmentation import random_walker
38 from postprocessing import random_walk_fill, random_walk_scalewise
39
40
41 # INITIALIZE SAMPLES -----
42 #     There are several ways to initialize samples. Uncomment one.
43
44 # load all 201 samples
45 # placenta = list_placentas('T-BN')
46 # load placenta from a certain quality category 0=good, 1=okay, 2=fair, 3=poor
47
48 placenta = list_by_quality(0, N=1)
49 #placenta = list()
50 #placenta.extend(list_by_quality(1))
51 #placenta.extend(list_by_quality(2))
52 #placenta.extend(list_by_quality(3))
53
54 # load from a file (sample names are keys of the json file)
55 # placenta = list_by_quality(json_file='manual_batch.json')
56
57 # for a single named sample, use a 1 element list.
58 # placenta = ['T-BN0204423.png']
59
60 #placenta = ['barium1.png',]
61 # RUNTIME OPTIONS -----
62 #     Where to save and whether or not to use old targets.
63
64 MAKE_NPZ_FILES = False # pickle frangi targets if you can
65 USE_NPZ_FILES = False # use old npz files if you can
66 NPZ_DIR = 'output/181201-L' # where to look for npz files
67 OUTPUT_DIR = 'output/181201-L' # where to save outputs
68
69 # add in a meta switch for verbosity (or levels)
70 #VERBOSE = False
71
72 # FRANGI / EXTRACT_PCSVN OPTIONS -----
73
74 # Find bright curvilinear structure against a dark background -> True
75 # Find dark curvilinear structure against a bright background -> False
76 # DARK_BG -> ignore and return signed Frangi scores
77 DARK_BG = False
78
79 # Along with the above, this will return "opposite" signed frangi scores.
80 # if this is True, then DARK_BG controls the "polarity" of the filter.
81 # See frangi.get_frangi_targets for details.

```

```

82 SIGNED_FRANGI = False
83
84 # Do not calculate hessian scores close to the boundary (this is important
85 # mainly in terms of ensuring that the hessian is very large on the edge of
86 # the plate (which would influence gamma calculation)
87 DILATE_PER_SCALE = True
88
89 # Attempt to remove glare from sample (some are OK, some are bad)
90 FLATTEN_MODE = 'L' # 'G' or 'L'
91 REMOVE_GLARE = True
92 REMOVE_CUTS = True
93
94 # Which scales to use
95 SCALE_RANGE = (-1.0, 3.5); SCALE_TYPE = 'logarithmic'
96 #SCALE_RANGE = (.2, 12); SCALE_TYPE = 'linear'
97 N_SCALES = 20
98
99 # use this if you want to use a custom argument (comment out the above)
100 SCALES = None
101 #SCALE_RANGE = None, SCALE_TYPE == 'custom'
102
103
104 # Explicit Frangi Parameters (pass a scalar, array as long as scales)
105 BETAS = 0.35
106 GAMMAS = 0.5
107 CS = None # pass scalar, array, or None
108 ALPHAS = None # set custom alphas or calculate later
109 FIXED_ALPHA = .3
110
111 RESCALE_FRANGI = True
112 GRADIENT_FILTER = False
113
114
115 # Scoring Decisions (don't need to touch these)
116 UCIP_RADIUS = 50 # area around the umbilical cord insertion point to ignore
117 INV_SIGMA = 0.8
118 # some other initializations, don't mind me
119
120
121
122
123 # CODE BEGINS HERE -----
124
125 if SCALES is None:
126     if SCALE_TYPE == 'linear':
127         scales = np.linspace(*SCALE_RANGE, num=N_SCALES)
128     elif SCALE_TYPE == 'logarithmic':
129         scales = np.logspace(*SCALE_RANGE, num=N_SCALES, base=2)
130 else:
131     scales = SCALES
132     SCALE_TYPE = 'custom' # this and the next three lines are just for logging
133     N_SCALES = len(SCALES)
134     SCALES = (min(SCALES), max(SCALES))
135
136 mccs = dict() # empty dict to store MCC's of each sample
137 pnccs = dict() # empty dict to store percent network covered for each sample
138
139 n_samples = len(placentas)
140
141 if not os.path.exists(OUTPUT_DIR):
142     os.makedirs(OUTPUT_DIR)
143
144 print(n_samples, "samples total!")
145

```

```

146 for i, filename in enumerate(placentas):
147
148     print('*'*80)
149     print(f'extracting PCSVN of {filename}\t({i} of {n_samples})')
150
151     # --- Setup, Preprocessing, Frangi Filter (it's mixed up) -----
152
153     raw_img = get_named_placenta(filename, mode=FLATTEN_MODE)
154
155     ucip = open_typefile(filename, 'ucip')
156
157     if REMOVE_CUTS:
158         img, has_cut = mask_cuts_simple(raw_img, ucip, return_success=True)
159         img.data[img.mask] = 0 # actually zero out that area
160     else:
161         img = raw_img.copy()
162
163     if REMOVE_GLARE:
164         img = inpaint_hybrid(img)
165
166
167     if USE_NPZ_FILES:
168         # find the first npz file with the sample name in it in the
169         # specified directory.
170         stub = filename.rstrip('.png')
171         for f in os.scandir(NPZ_DIR):
172             if f.name.endswith('npz') and f.name.startswith(stub):
173                 npz_filename = os.path.join(NPZ_DIR, f.name)
174                 print(f'using the npz file {npz_filename}')
175                 break # we'll just use the first one we can find.
176         else:
177             print(f'no npz file found for {filename}.')
178         npz_filename = None
179     else:
180         npz_filename = None
181
182     # set a lambda function to make output file names
183     outname = get_outname_lambda(filename, output_dir=OUTPUT_DIR)
184
185     if npz_filename is not None:
186
187         F = np.load(npz_filename)['F']
188
189         # in case preprocessing happens inside extract_pcsvn, do it out here
190
191         print('successfully loaded the frangi targets!')
192
193     else:
194         print('finding multiscale frangi targets')
195
196         # F is an array of frangi scores of shape (*img.shape, N_SCALES)
197         F, jfile = extract_pcsvn(img, filename, dark_bg=DARK_BG, beta=BETAS,
198                                  scales=scales, gamma=GAMMAS, c=CS,
199                                  kernel='discrete', dilate_per_scale=True,
200                                  verbose=False, signed_frangi=SIGNED_FRANGI,
201                                  generate_json=True, output_dir=OUTPUT_DIR,
202                                  rescale_frangi=RESCALE_FRANGI,
203                                  gradient_filter=GRADIENT_FILTER)
204
205     if MAKE_NPZ_FILES:
206         npzfile = ".".join((outname("F").rsplit('.', maxsplit=1)[0], 'npz'))
207
208         print("saving frangi targets to ", npzfile)
209         np.savez_compressed(npzfile, F=F)

```

```

210
211 # --- Merging & Postprocessing -----
212
213 # This is the maximum frangi response over all scales at each location
214 Fmax = F.max(axis=-1)
215
216 print("...making outputs")
217
218 if ALPHAS is None:
219     print("thresholding ALPHAS with top 5% scores at each scale")
220     ALPHAS = np.array([nz_percentile(F[..., k], 95.0)
221                         for k in range(N_SCALES)])
222
223 # the maximum value of the entire image at each scale
224 scale_maxes = np.array([F[..., i].max() for i in range(F.shape[-1])])
225
226 table = pandas.DataFrame(np.dstack((scales, ALPHAS, scale_maxes)).squeeze(),
227                           columns=(' ', ' ', '_p', 'max(F_)'))
228
229 print(table)
230 # threshold the responses at each of these values and get labels of max
231 approx, labs = apply_threshold(F, ALPHAS, return_labels=True)
232
233 # --- Scoring and Outputs -----
234
235 # get the main (boolean) tracefile and the RGB tracefiles
236 trace = open_tracefile(filename, as_binary=True)
237 A_trace = open_typefile(filename, 'arteries')
238 if A_trace is None:
239     # there are no special trace files for this sample
240     skeltrace = skeletonize_trace(trace)
241 else:
242     V_trace = open_typefile(filename, 'veins')
243     skeltrace = skeletonize_trace(A_trace, V_trace)
244
245 # get a matrix of pixel widths in the trace
246 widths = merge_widths_from_traces(A_trace, V_trace, strategy='arteries')
247
248 # find cord insertion point and resolution of the image
249 ucip_midpoint, resolution = measure_ncs_markings(ucip)
250 # if verbose:
251 # print(f"The umbilical cord insertion point is at {ucip_midpoint}")
252 # print(f"The resolution of the image is {resolution} pixels per cm.")
253
254 if ucip_midpoint is None:
255     ucip_mask = img.mask
256 # mask anywhere close to the UCIP
257 else:
258     ucip_mask = add_ucip_to_mask(ucip_midpoint,
259                                   radius=int(UCIP_RADIUS), mask=img.mask)
260
261 # The following are examples of things you can do:
262
263 # matrix of widths of traced image
264 # min_widths = merge_widths_from_traces(A_trace, V_trace,
265 #                                         strategy='minimum')
266
267 # trace ignoring largest vessels (19 pixels wide)
268 # trace_smaller_only = filter_widths(min_widths, min_width=3, max_width=17)
269 # trace_smaller_only != 0
270
271 # use only some scales
272 #approx_LO, labs_LO = apply_threshold(F[:, :, LO_offset:], ALPHAS[LO_offset:])
273 approx_FA, labs_FA = apply_threshold(F, FIXED_ALPHA)

```

```

274
275 # fix labels to incorporate offset
276 #labs_L0 = (labs_L0 != 0)*(labs_L0 + LO_offset)
277
278 # confusion matrix against default trace
279 confuse = confusion(approx, trace, bg_mask=ucip_mask)
280 #confuse_L0 = confusion(approx_L0, trace, bg_mask=ucip_mask)
281 confuse_FA = confusion(approx_FA, trace, bg_mask=ucip_mask)
282
283 m_score, counts = mcc(approx, trace, ucip_mask, return_counts=True)
284 m_score_FA, counts_FA = mcc(approx_FA, trace, ucip_mask,
285                             return_counts=True)
286
287 # this all just verifies that the 4 categories were added up
288 # correctly and match the total number of pixels in the reported
289 # placental plate.
290 TP, TN, FP, FN = counts # return these for more analysis?
291
292 total = np.sum(~ucip_mask)
293 #print(f'TP: {TP}\t TN: {TN}\nFP: {FP}\tFN: {FN}')
294 # just a sanity check
295 #print(f'TP+TN+FP+FN={TP+TN+FP+FN}\ttotal pixels={total}')
296
297
298 #approx_rw, markers, margins_added = random_walk_fill(img, Fmax, .3, .01,
299 #                                                    DARK_BG)
300
301 approx_rw, labs_rw = random_walk_scalewise(F, .4, return_labels=True)
302 #confuse_margins = confusion(margins_added, trace, bg_mask=ucip_mask)
303
304 high_alphas = np.array([nz_percentile(F[..., k], 98.0)
305                         for k in range(N_SCALES)])
306
307 high_frangi = apply_threshold(F, high_alphas, return_labels=False)
308 confuse_rw = confusion(approx_rw, trace, bg_mask=ucip_mask)
309 m_score_rw, counts_rw = mcc(approx_rw, trace, ucip_mask,
310                             return_counts=True)
311 pnc_rw = (skeltrace & approx_rw).sum() / skeltrace.sum()
312
313
314 # --- Generating Visual Outputs-----
315 crop = cropped_args(img) # these indices crop out the mask significantly
316
317 # save the raw, unaltered image
318 plt.imsave(outname('0_raw'), raw_img[crop].filled(0), cmap=plt.cm.gray)
319
320 # save the preprocessed image
321 plt.imsave(outname('1_img'), img[crop].filled(0), cmap=plt.cm.gray)
322
323 # save the maximum frangi output over all scales
324 plt.imsave(outname('2_fmax'), Fmax[crop], vmin=0, vmax=1.0,
325             cmap=plt.cm.nipy_spectral)
326
327 # only save the colorbar the first time
328 save_colorbar = (i==0)
329 scale_label_figure(labs, scales, crop=crop,
330                     savefilename=outname('3_labeled'), image_only=True,
331                     save_colorbar_separate=save_colorbar,
332                     output_dir=OUTPUT_DIR)
333
334 scale_label_figure(labs_rw, scales, crop=crop,
335                     savefilename=outname('A_labeled_rw'), image_only=True,
336                     save_colorbar_separate=save_colorbar,
337                     output_dir=OUTPUT_DIR)

```

```

338 plt.imsave(outname('4_confusion'), confuse[crop])
339
340 plt.imsave(outname('7_confusion_FA'), confuse_FA[crop])
341 plt.imsave(outname('B_confusion_rw'), confuse_rw[crop])
342 #plt.imsave(outname('A_markers_rw'), markers[crop])
343 #plt.imsave(outname('9_margin_for_rw'), confuse_margins[crop])
344 percent_covered = (skeltrace & approx).sum() / skeltrace.sum()
345 percent_covered_FA = (skeltrace & approx_FA).sum() / skeltrace.sum()
346
347
348
349 st_colors = {
350     'TN': (79, 79, 79),  # true negative# 'f7f7f7'
351     'TP': (0, 0, 0),    # true positive  # '000000'
352     'FN': (201, 53, 108), # false negative # 'f1a340' orange
353     'FP': (92, 92, 92),  # false positive
354     'mask': (247, 200, 200) # mask color (not used in MCC calculation)
355 }
356
357 plt.imsave(outname('5_coverage'), confusion(approx, skeltrace,
358                                              colordict=st_colors)[crop])
359 plt.imsave(outname('8_coverage_FA'), confusion(approx_FA, skeltrace,
360                                              colordict=st_colors)[crop])
361 plt.imsave(outname('C_coverage_rw'), confusion(approx_rw, skeltrace,
362                                              colordict=st_colors)[crop])
363
364 # make the graph that shows what scale the max was pulled from
365
366 scale_label_figure(labs_FA, scales, crop=crop,
367                     savefilename=outname('6_labeled_FA'), image_only=True,
368                     save_colorbar_separate=False, output_dir=OUTPUT_DIR)
369
370 V = np.transpose(F, axes=(2, 0, 1))
371
372 #view_slices(F[crop], axis=-1, scales=scales)
373
374 print('starting to sieve')
375 sieved = sieve_scales(V, 98, 95)
376
377 approx_S, labs_S = (sieved != 0), sieved
378 confuse_S = confusion(approx_S, trace, bg_mask=ucip_mask)
379
380 scale_label_figure(labs_S, scales, crop=crop,
381                     savefilename=outname('D_labeled_S'), image_only=True,
382                     save_colorbar_separate=False, output_dir=OUTPUT_DIR)
383
384 plt.imsave(outname('E_confusion_S'), confuse_S[crop])
385
386 m_score_S, counts_S = mcc(approx_S, trace, ucip_mask, return_counts=True)
387 pnc_S = (skeltrace & approx_S).sum() / skeltrace.sum()
388
389 mccs[filename] = (m_score, m_score_FA, m_score_rw, m_score_S )
390 pncs[filename] = (percent_covered, percent_covered_FA, pnc_rw, pnc_S)
391
392 print('percentage of skeltrace covered:(percentile filtering)',
393       f'{percent_covered:.2%}')
394 print('percentage of skeltrace covered (fixed alpha):',
395       f'{percent_covered_FA:.2%}')
396 print('percentage of skeltrace covered (random_walker):',
397       f'{pnc_rw:.2%}')
398 print('percentage of skeltrace covered (sieving):',
399       f'{pnc_S:.2%}')
400
401
```

```

402 print(f'mcc score of {m_score:.3} for percentile filtering')
403 print(f'mcc score of {m_score_FA:.3} with fixed alpha {FIXED_ALPHA}')
404 print(f'mcc score of {m_score_rw:.3} after random walker')
405 print(f'mcc score of {m_score_S:.3} after sieving')
406 ##### THIS IS ALL A HORRIBLE MESS. FIX IT
407
408 # why don't you just return the dict instead
409 with open(jfile, 'r') as f:
410     slog = json.load(f)
411
412 c2d = lambda t: dict(zip(['TP', 'TN', 'FP', 'FN'], [int(c) for c in t]))
413
414 slog['counts'] = c2d(counts)
415 slog['counts_FA'] = c2d(counts_FA)
416 slog['counts_rw'] = c2d(counts_rw)
417 slog['pnc'] = pncts[filename]
418 slog['mcc'] = mcss[filename]
419 slog['scale_maxes'] = list(scale_maxes)
420 slog['ALPHAS'] = list(ALPHAS)
421
422 with open(jfile, 'w') as f:
423     json.dump(slog, f)
424
425 plt.close('all')
426
427 # Post-run Meta-Output and Logging -----
428
429 timestamp = datetime.datetime.now()
430 timestamp = timestamp.strftime("%y%m%d_%H%M")
431
432 mccfile = os.path.join(OUTPUT_DIR, f"runlog_{timestamp}.json")
433
434 runlog = {
435     'time': timestamp,
436     'DARK_BG': DARK_BG,
437     'DILATE_PER_SCALE': DILATE_PER_SCALE,
438     'SCALE_RANGE': SCALE_RANGE,
439     'SCALE_TYPE': SCALE_TYPE,
440     'N_SCALES': N_SCALES,
441     'scales': list(scales),
442     'ALPHAS': list(ALPHAS),
443     'BETAS': None,
444     'use_npz_files': False,
445     'remove_glare': REMOVE_GLARE,
446     'files': list(placentas),
447     'MCCS': mcss,
448     'PNC': pncts
449 }
450
451 # save to a json file
452 with open(mccfile, 'w') as f:
453     json.dump(runlog, f, indent=True)

```

listings/frangi_graphing.py

```

1 import matplotlib as mpl
2 from mpl_toolkits.mplot3d import Axes3D
3 import matplotlib.pyplot as plt
4 import numpy as np
5 from numpy import exp
6 from skimage.io import imread
7 from skimage.util import montage

```

```

8 from itertools import product
9
10 def s(x, gamma):
11     """normalized structureness factor.
12     x is the ratio of the input to the maximum possible structureness factor
13     (smax) at that scale (which would cancel out with the c in the denominator)
14     """
15     return (1 - exp(-x**2 / (2*gamma**2))) / (1 - exp(-1/(2*gamma**2)))
16
17 def r(y, beta):
18     """normalized anisotropy factor
19     y is the ratio |k1 / k2|, so y=0 corresponds to perfectly isotropic
20     and y->1 corresponds to highly anisotropic
21     """
22     return np.exp(-y**2 / (2*beta**2))
23
24 #plt.rc('text', usetex=True)
25 #plt.rc('font', family='serif')
26
27
28 dom = np.linspace(0, 1)
29 plt.close('all')
30
31 # show dependence of structureness factor on its parameter
32 for gamma in [0.1, 0.25, 0.35, 0.5, 0.9, 1, 2, 10, 1000]:
33     plt.plot(dom, s(dom, gamma), label=r'$\gamma=\{}$'.format(gamma))
34
35 plt.ylabel(r'$\left(1-\exp\left(\frac{-S}{2(\gamma_{\max})^2}\right)\right)$',
36         fontsize=14)
37 plt.xlabel(r'$S / S_{\max}$', fontsize=14)
38 plt.title(r'Dependence of Structureness Factor on Parameter $\gamma=(c/S_{\max})$')
39 plt.legend()
40
41 #plt.show()
42 plt.close('all')
43
44 # show dependence of anisotropy factor on its parameter
45 for beta in [0.1, 0.25, 0.35, 0.5, 0.9, 1, 2, 10, 1000]:
46     plt.plot(dom, r(dom, beta), label=r'$\beta=\{}$'.format(beta))
47
48 plt.ylabel(r'$\exp\left(-\frac{A}{2\beta^2}\right)$',
49         fontsize=14)
50 plt.xlabel(r'$A = \lambda_1 / \lambda_2$', fontsize=14)
51 plt.title(r'Dependence of Anisotropy Factor on Parameter $\beta$')
52 plt.legend()
53
54 #plt.show()
55 plt.close('all')
56
57 prange = [0.1, 0.25, 0.5, 0.9, 1, 1.5]
58
59 for n, (beta, gamma) in enumerate(product(prange, prange)):
60
61     fig = plt.figure(figsize=(8,5))
62     ax = fig.gca(projection='3d')
63     X, Y = np.meshgrid(dom, dom)
64
65     Z = r(X,beta)*s(Y,gamma)
66
67     surf = ax.plot_surface(X,Y,Z, cmap='coolwarm', linewidth=0)
68
69     ax.set_xlabel(r'$\lambda_1 / \lambda_2$')
70     ax.set_ylabel(r'$S / S_{\max}$')
71

```

```

72
73     ax.set_title(r"Rescaled Frangi filter, "
74                 r"$\beta={}$, $\gamma={}$".format(beta, gamma))
75
76     #fig.colorbar(surf, shrink=0.5, aspect=5)
77     fig.tight_layout()
78
79     plt.savefig(f'demo_output/frangi3d/{n}.png', dpi=300)
80
81     plt.close()
82
83 imgs = [imread(f'demo_output/frangi3d/{n}.png') for n in range(36*1)]
84 imgs = np.stack(imgs)
85
86 for n in range(6):
87     plt.imsave(f'demo_output/frangi3d/frangi3dpart{n}.png',
88                montage(imgs[(n*6):((n+1)*6)], multichannel=True,
89                        grid_shape=(3,2)))

```

listings/frangi.py

```

1 import numpy as np
2 import numpy.ma
3 from hfft import fft_hessian, fft_gradient
4 from diffgeo import principal_curvatures
5 from plate_morphology import dilate_boundary
6 from merging import nz_percentile
7
8 def frangi_from_image(img, sigma, beta=0.5, gamma=0.5, c=None, dark_bg=True,
9                       dilation_radius=None, kernel=None, signed_frangi=False,
10                      return_debug_info=False, verbose=False,
11                      rescale_frangi=False, gradient_filter=False):
12     """Calculate the (uniscale) Frangi vesselness measure on a grayscale image
13
14     Parameters
15     -----
16     img: ndarray or ma.MaskedArray
17         a one-channel image. If this is a masked array (preferred), ignore the
18         masked regions of the image
19     sigma: float
20         Standard deviation of the gaussian, used to calculate derivatives.
21     beta: float, optional
22         The anisotropy parameter of the Frangi filter (default is 0.5)
23     gamma: float, optional
24         Scaling factor for the structureness parameter of the Frangi
25         filter. The structureness parameter will be set to gamma * maximum
26         of the hessian norm. (Default is 0.5)
27     c: float or None, optional
28         The strutureness parameter of the Frangi filter. If this is set then
29         gamma is ignored. (Default is None).
30     dilation_radius: int or None
31         If dilation radius is supplied, then areas within that amount of pixels
32         will not be calculated. This is preferable in certain contexts,
33         especially when there is a dark background and dark_bg=True. This is
34         especially recommended for small sigmas and when gamma is not provided.
35         None to forgo this procedure (default). A mask must be supplied for
36         this to make sense.
37     dark_bg: boolean or None
38         if True, then frangi will select only for bright curvilinear
39         features; if False, then Frangi will select only for dark
40         curvilinear structures. if None instead of a bool, then curvilinear
41         structures of either type will be reported.

```

```

42     signed_frangi: bool, optional
43         if signed is True, the result will be the same as if dark_bg is set
44         to None, except that the sign will change to match the desired
45         features. See example below.
46     return_debug info: bool, optional
47         will return a large dict consisting of several large matrices,
48         calculated hessian, etc.
49
50         scale_dict = {'sigma': sigma,
51                         'beta': beta,
52                         'gamma': gamma,
53                         'c': c,
54                         'H': hesh,
55                         'F': targets,
56                         'k1': k1,
57                         'k2': k2,
58                         'border_radius': dilation_radius
59                     }
60
61     Returns
62     -----
63     ...
64
65     Notes
66     -----
67     Although default is 0.5, this means that the structureness factor of the
68     Frangi score will only be 0.86 at its maximum. Larger values of gamma
69     will only dampen the frangi filter more. Smaller values toward 0 will
70     result in a "looser" filter. For example, if gamma = .25, then the
71     maximum score is (1-exp{-8}) around .999 (it may be desirable that the
72     franginess score should be able to achieve a score of 1).
73
74     This function will accept 0 an input, and the structureness factor will
75     be set to 1 everywhere (the limiting case as gamma -> 0)
76
77     Frangi structureness factor is (1 - exp((-S**2)/(2*c**2)))
78     """
79     hesh = fft_hessian(img, sigma, kernel=kernel) # the triple (Hxx,Hxy,Hyy)
80     # calculate principal curvatures with |k1| <= |k2|
81
82
83     k1, k2 = principal_curvatures(img, sigma, H=hesh)
84
85     if dilation_radius is not None:
86         # pass None to just get the mask back
87         collar = dilate_boundary(None, radius=dilation_radius, mask=img.mask)
88
89         # get rid of "bad" K values before you calculate gamma and Frangi
90         k1[collar] = 0
91         k2[collar] = 0
92         hesh[0][collar] = 0
93         hesh[1][collar] = 0
94         hesh[2][collar] = 0
95     else:
96         collar = img.mask.copy()
97
98
99     # no need to set gamma or c anymore. will be set inside get_frangi_targets
100    #if c is None:
101    #    Frangi suggested 'half the max Hessian norm' as an empirical
102    #    half the max spectral radius is easier to calculate so do that
103    #    shouldn't be affected by mask data but should make sure the
104    #    mask is *well* far away from perimeter
105    #    we actually calculate half of max hessian norm

```

```

106 # using frob_norm = sqrt(trace(AA^T))
107 # alternatively you could use gamma = .5 * np.abs(k2).max()
108 #hnorm = hessian_norm(hesh, mask=collar)
109 #print(f' ={sigma}:2f}')
110 #gamma0 = .5*hessian_norm(hesh).max()
111 #print(f'\t{gamma0:.5f} = frob-norm    pre-dilation')
112
113 #gamma1 = .5*hessian_norm(hesh, mask=collar).max()
114 #print(f'\t{gamma1:.5f} = frob-norm    post-collar dilation {dilation_radius}')
115 #l2gamma = .5*np.max(np.abs(k2))
116 #print(f'\t{l2gamma:.5f} = from L2-norm    (K2 with collar)')
117
118 #hdilation = int(max(np.ceil(sigma),10))
119 #hcollar = dilate_boundary(None, radius=hdilation, mask=img.mask)
120 #gamma = .5 * max_hessian_norm(hesh, mask=hcollar)
121
122 #print(f'\t{gamma:.5f} =    post-hdilation (radius {hdilation}) (old    )')
123
124 #print('changing      to L2-norm with collar')
125 #gamma = max(gamma1, l2gamma, gamma0)
126
127 # wish this scaled a little better
128
129 # a very large gamma here will make the Frangi score zero
130 # a very small gamma means that we are artificially inflating the
131 # structureness measure
132 #import matplotlib.pyplot as plt
133 #plt.imshow(hnorm*(~collar))
134 #plt.show()
135 #print(hnorm[~collar].min(), hnorm[~collar].max())
136 #if hnorm[~collar].max() < 0.1:
137 #    print(f'max hessian norm is very small at this scale ({sigma},{hnorm[~collar].max():.5f}')
138 #          'you should maybe skip this scale')
139 #elif hnorm[~collar].min() < 0.001:
140 #    # only trigger if the first one didn't
141 #    print(f'min hessian norm is very small at this scale ({sigma}, {hnorm[~collar].min():.5f}')
142 #          'be carefully of artificially inflated scores')
143 #S = np.sqrt(k1**2 + k2**2)
144 #import matplotlib.pyplot as plt
145 #plt.imshow(S)
146 #plt.show()
147 #plt.close()
148 #print('max hessian norm (Frob): ', hnorm.max())
149 #print('max structureness: ', S.max())
150 #c = gamma*S.max()

151
152 if verbose:
153     print(f'finding Frangi targets with   ={beta} and   ={c:.2f}')
154
155 targets = get_frangi_targets(k1, k2, beta=beta, gamma=gamma, c=c,
156                             dark_bg=dark_bg, signed=signed_frangi,
157                             rescale_frangi=rescale_frangi)

158
159 if gradient_filter:
160     # obviously you could compute this at the same time as the hessian :/
161
162     g = fft_gradient(img, sigma)
163     g = dilate_boundary(g, radius=20, mask=img.mask)
164
165     # you could technically pass a function to switch between these
166     # behaviors
167     low_g = (g < nz_percentile(g, 50)).filled(0)
168
169     targets[~low_g] = 0

```

```

170
171     if not return_debug_info:
172         return targets
173     else:
174
175         # for logging we have to recalculate this
176         if c is not None:
177             c = gamma * max(np.sqrt(k1**2 + k2**2))
178
179         scale_dict = {'sigma': sigma,
180                      'beta': beta,
181                      'gamma': gamma,
182                      'c': c,
183                      'H': hesh,
184                      'F': targets,
185                      'k1': k1,
186                      'k2': k2,
187                      'border_radius': dilation_radius
188                      }
189
190     return targets, scale_dict
191
192
193 def get_frangi_targets(K1, K2, beta=0.5, gamma=0.5, c=None,
194                         dark_bg=True, signed=False, rescale_frangi=False):
195     """Calculate the Frangi vesselness measure from eigenvalues.
196
197     Parameters
198     -----
199     K1, K2 : ndarray (each)
200         each is an ndarray of eigenvalues (approximated principal
201         curvatures) for some image.
202     beta: float
203         the anisotropy parameter (default is 0.5)
204     gamma: float or None
205         Scaling factor for the the structureness parameter. The structureness
206         parameter c will be set to gamma times the maximum of the Hessian
207         norm, sqrt(K1**2 + K2**2). Default is 0.5
208     c: float or None
209         The frangi structureness parameter. If this is set, gamma above will
210         be ignored.
211     dark_bg: boolean or None
212         if True, then frangi will select only for bright curvilinear
213         features; if False, then Frangi will select only for dark
214         curvilinear structures. if None instead of a bool, then curvilinear
215         structures of either type will be reported.
216     signed: boolean
217         if signed is True, the result will be the same as if dark_bg is set
218         to None, except that the sign will change to match the desired
219         features. See example below.
220
221     Returns
222     -----
223     F: ndarray, same shape as K1
224         the Frangi vesselness measure.
225
226     Notes
227     -----
228     If beta or gamma are set to 0, then the frangi anisotropy factor will be
229     set to 0 or 1 everywhere (which is the limiting case as beta->0 or
230     gamma->0) You can set beta = 'inf' or np.inf to set anisotropy factor to 1.
231
232     Examples
233     -----

```

```

234 >>>f1 = get_frangi_targets(K1,K2, dark_bg=True, signed=True)
235 >>>f2 = get_frangi_targets(K1,K2, dark_bg=False, signed=True)
236 >>>np.all(f1 == -f2)
237 True
238
239 >>> F = get_frangi_targets(K1,K2, gamma=0.5)
240 >>> Falt = get_frangi_targets(K1,K2, c=0.5*np.sqrt(K1**2 + K2**2))
241 >>> np.all(F == Falt)
242 True
243 """
244
245 A = anisotropy(K1, K2, beta=beta)
246 S = structureness(K1, K2, gamma=gamma, c=c)
247
248 anisotropy_factor = np.exp(-A)
249 structureness_factor = (1 - np.exp(-S))
250
251 F = anisotropy_factor * structureness_factor
252
253 if rescale_frangi:
254     if c is not None:
255         # more like will not
256         print('c was set to an arbitrary value. cannot rescale')
257     else:
258         max_theoretical = (1 - np.exp(-1/(2*gamma**2)))
259         F = F / max_theoretical
260
261 # now just filter/ change sign as appropriate.
262 if not signed:
263     # calculate the regular frangi filter
264     if dark_bg is None:
265         #keep F the way it is
266         pass
267     elif dark_bg:
268         # zero responses from positive curvatures
269         F = (K2 < 0)*F
270     else:
271         # zero responses from negative curvatures
272         F = (K2 > 0)*F
273 else:
274     if dark_bg is None:
275         # output is already signed
276         pass
277     elif dark_bg:
278         # positive curvature spots will be made negative
279         F[K2 > 0] = -1 * F[K2 > 0]
280     else:
281         # negative curvature spots will be made positive
282         F[K2 < 0] = -1 * F[K2 < 0]
283
284 # finally, reapply the mask if the inputs came with one
285 if numpy.ma.is_masked(K1):
286     F = numpy.ma.masked_array(F, mask=K1.mask)
287
288 return F
289
290
291 def hessian_norm(hesh, mask=None):
292     """Calculate Frobenius norm of Hessian.
293
294     Calculates the maximal value (over all pixels of the image) of the
295     Frobenius norm of the Hessian. This should be the same as the square root
296     of unscaled structureness.
297

```

```

298 Parameters
299 -----
300 hesh: a tuple of ndarrays
301     The tuple hxx,hxy,hyy which are all the same shape. The hessian at
302     the point (m,n) is then [[hxx[m,n], hxy[m,n]],
303                             [hxy[m,n], hyy[m,n]]]
304
305 Returns
306 -----
307 float
308 """
309
310 hxx, hxy, hyy = hesh
311
312 # frob norm is just sqrt(trace(AA^T)) which is easy for a 2x2
313 hnorm = (hxx**2 + 2*hxy**2 + hyy**2)
314
315 if mask is not None:
316     hnorm[mask] = 0
317
318 hnorm = np.sqrt(hnorm)
319 return hnorm
320
321
322 def anisotropy(K1,K2, beta=0.5):
323     """Convenience function for the exponential argument in the Frangi
324     anisotropy factor.
325
326     According to Frangi (1998) this is technically (A**2) / (2*beta**2)
327     unless beta is None, in which case just A**2 is returned
328
329     The frangi vesselness factor is formally (np.exp(-R))
330     where R is what's returned by this function
331 """
332
333 A = (K1 / K2) ** 2
334 #print(f'inside anisotropy,   ={beta}')
335 if beta == 0:
336     return np.zeros_like(A) # the limiting case as beta -> 0
337
338 elif beta == 'inf' or np.isinf(beta):
339     return np.ones_like(A) # the limiting case as beta -> inf
340
341 elif beta is None:
342     return A # just return the A**2 part (why though)
343 else:
344     return A / (2*beta**2)
345
346
347 def structureness(K1, K2, gamma=0.5, c=None):
348     """Convenience function for Structureness measure.
349     According to Frangi (1998) this is technically S**2
350 """
351 S = K1**2 + K2**2
352
353
354 # is c is not provided, calculate it
355 if c is None:
356     c = gamma * np.sqrt(S).max() # the max Frob norm of the Hessian
357
358 #print(f'inside structureness,   ={gamma}, c={c}')
359
360 if c == 0:
361     return np.zeros_like(S)

```

```

362
363     elif c == 'inf' or np.isinf(c):
364         return np.ones_like(S)
365
366     elif c is None:
367         return S
368
369     else:
370         return S / (2*c**2)

```

listings/gradient_filter_demo.py

```

1 #!/usr/bin/env python3
2
3 import numpy as np
4 import matplotlib.pyplot as plt
5 from skimage.util import img_as_float
6 from skimage.io import imread
7 from placenta import (get_named_placenta, list_by_quality, cropped_args,
8                         img_as_float)
9
10 from frangi import frangi_from_image
11 from hfft import fft_gradient, fft_hessian, fft_gaussian
12 from merging import nz_percentile
13 from plate_morphology import dilate_boundary
14 import os.path, os
15
16 MAKE_OUTPUTS = False
17 OUTPUT_DIR = 'demo_output/gradient_filter_demo'
18
19 BETA = .5
20
21 if not os.path.exists(OUTPUT_DIR):
22     os.makedirs(OUTPUT_DIR)
23
24 filename = list_by_quality(N=1)[0]
25 img = get_named_placenta(filename)
26 crop = cropped_args(img)
27
28 F0 = list()
29 F1 = list()
30
31 scales = np.logspace(-1, 3, num=12, base=2)
32
33 for n, sigma in enumerate(scales):
34
35     f0 = frangi_from_image(img, sigma, beta=BETA, dark_bg=False,
36                            dilation_radius=20, gradient_filter=False)
37
38     f1 = frangi_from_image(img, sigma, beta=BETA, dark_bg=False,
39                            dilation_radius=20, gradient_filter=True)
40
41     # simulate g
42     #g = fft_gradient(img, sigma)
43     #g = dilate_boundary(g, radius=20, mask=img.mask)
44     #g = g < nz_percentile(g, 50)
45     #g_filter = (~g).filled(0)
46
47 if MAKE_OUTPUTS:
48     fig, ax = plt.subplots(ncols=2, nrows=1, figsize=(10,4))
49
50     ax[0].imshow(f0.filled(0)[crop], vmin=0, vmax=1, cmap='nipy_spectral')

```

```

51     ax[0].axis('off')
52     ax[0].set_title(f'Standard Frangi    ={sigma:.2f}', fontsize=10)
53
54     ax[1].imshow(f1.filled(0)[crop], vmin=0, vmax=1, cmap='nipy_spectral')
55     ax[1].axis('off')
56     ax[1].set_title(f'w/ gradient filter', fontsize=10)
57
58     #ax[2].imshow(g_filter[crop].T, cmap='nipy_spectral')
59     #ax[2].axis('off')
60     #ax[2].set_title(f'Gradient filter    ={sigma:.2f}')
61
62     #plt.show()
63     plt.tight_layout()
64     plt.savefig(os.path.join(OUTPUT_DIR, f'gf_scale_{n:0{2}}.png'))
65     plt.close('all')
66
67 F0.append(f0)
68 F1.append(f1)
69
70 F0 = np.stack(F0)
71 F1 = np.stack(F1)
72
73
74 if MAKE_OUTPUTS:
75     fig, ax = plt.subplots(ncols=2, nrows=1, figsize=(10,4))
76
77     ax[0].imshow(F0.max(axis=0).filled(0)[crop], vmin=0, vmax=1,
78                   cmap='nipy_spectral')
79     ax[0].axis('off')
80     ax[0].set_title(f'Standard Frangi F_max', fontsize=10)
81
82     ax[1].imshow(F1.max(axis=0).filled(0)[crop], vmin=0, vmax=1,
83                   cmap='nipy_spectral')
84     ax[1].axis('off')
85     ax[1].set_title(f'w/ gradient filter', fontsize=10)
86
87     plt.tight_layout()
88     #plt.show()
89     plt.savefig(os.path.join(OUTPUT_DIR, f'gf_Fmax.png'))
90     plt.close('all')

```

listings/hfft_accuracy.py

```

1 #!/usr/bin/env python3
2 """
3 here you want to show the accuracy of hfft.py
4 BOILERPLATE
5
6 show that gaussian blur of hfft is accurate, except potentially around the
7 boundary proportional to sigma.
8 or if they're off by a scaling factor, show that the derivates
9 (taken the same way) are proportional.
10 pseudocode
11 A = gaussian_blur(image, sigma, method='conventional')
12 B = gaussian_blur(image, sigma, method='fourier')
13 zero_order_accurate = isclose(A, B, tol)
14
15
16
17
18
19
20

```

```

21 J_A= get_jacobian(A)
22 J_B = get_jacobian(B)
23
24 first_order_accurate = isclose(J_A, J_B, tol)
25
26 A_eroded = zero_around_plate(A, sigma)
27 B_eroded = zero_around_plate(B, sigma)
28
29 J_A_eroded = zero_around_plate(A, sigma)
30 J_B_eroded = zero_around_plate(B, sigma)
31
32 zero_order_accurate_no_boundary = isclose(A_eroded, B_eroded, tol)
33 first_order_accurate = isclose(J_A_eroded, J_B_eroded, tol)
34 """
35
36
37 from placenta import get_named_placenta, cropped_args
38
39 from itertools import combinations_with_replacement
40 from skimage.exposure import rescale_intensity
41
42 from hfft import fft_hessian, fft_gaussian, fft_dgk
43 from scipy.ndimage import gaussian_filter
44 import matplotlib.pyplot as plt
45 from placenta import show_mask, list_by_quality
46
47 from scoring import mean_squared_error
48 from itertools import combinations
49 import numpy as np
50 from scipy.ndimage import laplace
51 import numpy.ma as ma
52
53 from skimage.segmentation import find_boundaries
54 from skimage.morphology import disk, binary_dilation
55
56 from plate_morphology import dilate_boundary
57
58 from diffgeo import principal_curvatures
59 from frangi import structureness, anisotropy, get_frangi_targets
60
61 from skimage.util import img_as_float
62
63 def plot_image_slices(arrs, fixed_axis=0, fixed_index=None, labels=None,
64                      formats=None, title=None):
65     """
66     arrs needs to be the same shape and dimension
67     could pass it to np.stack and check for a value error?
68
69     """
70     fig, ax = plt.subplots(figsize=(12,2))
71     # hopefully the fixed axis is 0 or 1. this gets the other one
72     it_axis = 1 if fixed_axis==0 else 0
73
74     # if it's a tuple, make it an array, etc. etc.
75     arrs = np.stack(arrs)
76
77     # make sure we can iterate over it if there's just as single image
78     if arrs.ndim < 3:
79         arrs = np.expand_dims(arrs,0)
80
81     if labels is None:
82         labels = [None for a in arrs]
83     if formats is None:

```

```

84     formats = [ '' for a in arrs]
85
86     if fixed_index is None:
87         # find halfway point of the appropriate dimension from the first array
88         fixed_index = arrs[0].shape[fixed_axis] // 2
89
90     for a, lab, fmt in zip(arrs, labels, formats):
91         ax.plot(np.arange(a.shape[it_axis]),
92                 np.moveaxis(a, fixed_axis, 0)[fixed_index, :],
93                 fmt, label=lab)
94
95     if title is not None:
96         ax.set_title(title)
97
98     # can this be at least a little object-oriented? :(
99     fig.legend()
100
101 def multiway_comparison(arrs, scorefunc):
102
103     scores = np.zeros((len(arrs), len(arrs)))
104
105     for j in range(len(arrs)):
106         for k in range(j+1, len(arrs)):
107             scores[j,k] = scorefunc(arrs[j], arrs[k])
108
109     return scores
110
111 filename = list_by_quality(0)[5]
112
113 img = get_named_placenta(filename)
114
115 # so that scipy.ndimage.gaussian_filter doesn't use uint8 precision (jesus)
116 img = ma.masked_array(img_as_float(img), mask=img.mask)
117
118 test_sigmas = [.3, .6, 1.0, 5.0, 15, 30, 60, 90]
119
120 for sigma in test_sigmas:
121
122     print("*"*80, '\n\n', f"  ={sigma} ")
123     #print('applying standard gauss blur')
124
125     # this is exactly how it's passed to skimage.feature.hessian_matrix(...)
126     A = gaussian_filter(img.filled(0), sigma, mode='constant', cval=0)
127
128     #print('applying fft gauss blur')
129     B = fft_gaussian(img, sigma, kernel='sampled')
130     C = fft_gaussian(img, sigma, kernel='discrete')
131
132     #print('calculating first derivatives')
133     # zero the masks before calculating derivates if they're masked
134     Agrad = np.gradient(A)
135     Bgrad = np.gradient(B)
136     Cgrad = np.gradient(C)
137
138
139 axes = range(img.ndim)
140
141 #print('calculating second derivatives')
142 # this is the same way it's done in skimage.feature.hessian_matrix(...)
143 H_A = [np.gradient(Agrad[ax0], axis=ax1)
144        for ax0, ax1 in combinations_with_replacement(axes, 2)]
145 H_B = [np.gradient(Bgrad[ax0], axis=ax1)
146        for ax0, ax1 in combinations_with_replacement(axes, 2)]
147 H_C = [np.gradient(Cgrad[ax0], axis=ax1)

```

```

148     for ax0, ax1 in combinations_with_replacement(axes, 2)]
149
150     #print('calculating eigenvalues of hessian')
151     ak1, ak2 = principal_curvatures(img, sigma=sigma, H=H_A)
152     bk1, bk2 = principal_curvatures(img, sigma=sigma, H=H_B)
153     ck1, ck2 = principal_curvatures(img, sigma=sigma, H=H_C)
154
155
156     #RA = anisotropy(ak1,ak2)
157     #RB = anisotropy(bk1,bk2)
158     #RC = anisotropy(ck1,ck2)
159
160     #SA = structureness(ak1, ak2)
161     #SB = structureness(bk1, bk2)
162     #SC = structureness(ck1, ck2)
163
164     ## ugh, apply masks here. too large to be conservative?
165     ## otherwise structureness only shows up for small sizes
166     new_mask = dilate_boundary(None, radius=int(3*sigma), mask=img.mask)
167
168     crop = cropped_args(img)
169
170     A = A[crop]
171     B = B[crop]
172     C = C[crop]
173
174     ak1 = ma.masked_array(ak1,new_mask)[crop]
175     ak2 = ma.masked_array(ak2,new_mask)[crop]
176     bk1 = ma.masked_array(bk1,new_mask)[crop]
177     bk2 = ma.masked_array(bk2,new_mask)[crop]
178     ck1 = ma.masked_array(ck1,new_mask)[crop]
179     ck2 = ma.masked_array(ck2,new_mask)[crop]
180
181     FA = get_frangi_targets(ak1,ak2, dark_bg=False).filled(0)
182     FB = get_frangi_targets(bk1,bk2, dark_bg=False).filled(0)
183     FC = get_frangi_targets(ck1,ck2, dark_bg=False).filled(0)
184
185
186     # the following shows a random vertical slice of A & B (when scaled)
187     labels = ('scipy.ndimage,gaussian_filter', 'fft_gaussian', 'fft_dgk')
188     formats = ('g:', 'k', 'b-')
189     plot_image_slices((A,B,C), labels=labels, formats=formats,
190                         title=r'gaussian convolution $\sigma={}$'.format(sigma))
191     plt.tight_layout()
192     plt.savefig('Gslice_sigma={:d}.png'.format(int(sigma*10)), dpi=300)
193     plot_image_slices((FA,FB,FC), labels=labels, formats=formats,
194                         title=r'Frangi filter response $\sigma={}$'.format(sigma))
195     plt.tight_layout()
196     plt.savefig('Fslice_sigma={:d}.png'.format(int(sigma*10)), dpi=300)
197     #plt.show()
198
199     print('comparing gaussians (mean squared error)')
200     print(multiway_comparison((A,B,C), mean_squared_error))
201     print('comparing frangi response (mean squared error)')
202     print(multiway_comparison((FA,FB,FC), mean_squared_error))

```

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  from skimage.util import img_as_float
7
8  import matplotlib.pyplot as plt
9  from hfft import fft_gaussian, fft_hessian, fft_dgk
10 from scipy.ndimage import gaussian_filter
11
12 from scipy.linalg import norm
13 import timeit
14
15 #img = camera() / 255.
16 img = imread('samples/barium1.png', as_grey=True) / 255.
17 mask = imread('samples/barium1.mask.png', as_grey=True)
18
19 img = img_as_float(img)
20
21 # compare computation speed over sigmas
22
23 # N logarithmically spaced scales between 1 and 2^m
24 N = 32
25 m = 8
26 sigmas = np.logspace(0,m, num=N, base=2)
27
28 fft_results = list()
29 std_results = list()
30
31 for sigma in sigmas:
32     # test statements to compare (fft-based gaussian vs convolution-based)
33     fft_test_statement = "fft_gaussian(img,{},kernel='discrete')".format(sigma)
34     std_test_statement = "gaussian_filter(img,{})".format(sigma)
35     # run each statement 1 times (with 2 runs in each trial)
36     # returns/appends the average of 3 runs
37     fft_results.append(timeit.timeit(fft_test_statement,
38                                     number=1, globals=globals()))
39     std_results.append(timeit.timeit(std_test_statement,
40                                     number=1, globals=globals()))
41
42     # now actually evaluate both to compare
43     f = eval(fft_test_statement)
44     s = eval(std_test_statement)
45
46     # normalize each matrix by frobenius norm and take difference
47     # ideally should try to zero out the "mask" area
48     diff = np.abs(f / norm(f) - s / norm(s))
49     raw_diff = np.abs(f - s)
50     # don't care if it's the background
51     diff[mask==1] = 0
52     raw_diff[mask==1] = 0
53
54     # should format this stuff better into a legible table
55     print(sigma, diff.max(), raw_diff.max())
56
57 lines = plt.plot(sigmas, fft_results, 'go', sigmas, std_results, 'bo')
58 plt.xlabel('sigma (gaussian blur parameter)')
59 plt.ylabel('run time (seconds)')
60 plt.legend(lines, ('fft-gaussian', 'conv-gaussian'))
61 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, ive
7 from scipy.ndimage import gaussian_filter
8 from itertools import combinations_with_replacement
9
10 # for demos
11 import matplotlib.pyplot as plt
12
13 from skimage.data import camera
14 from skimage.util import img_as_float
15 from skimage.measure import compare_mse, compare_nrmse
16 """
17 hfft.py is the implementation of calculating the hessian of a real
18
19 image based in frequency space (rather than direct convolution with a gaussian
20 as is standard in scipy, for example).
21
22 TODO: PROVIDE MAIN USAGE NOTES
23 """
24
25 def fft_gaussian(img, sigma, kernel=None):
26     """
27         https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.fftconvolve.html
28
29     in particular the example in which a gaussian blur is implemented.
30
31     along with the comment:
32     "Gaussian blur implemented using FFT convolution. Notice the dark borders
33     around the image, due to the zero-padding beyond its boundaries. The
34     convolve2d function allows for other types of image boundaries, but is far
35     slower"
36
37     (i.e. doesn't use FFT).
38
39     note that here, you actually take the FFT of a gaussian (rather than
40     build it in frequency space). there are ~6 ways to do this.
41     """
42
43     #create a 2D gaussian kernel to take the FFT of
44     # output of signal.gaussian is normalized to 1 so you need to scale
45     # it back to work
46     #A = 1 / (2*np.pi*sigma**2) # scale factor for 2D
47
48     if kernel in ('discrete', None):
49         kern_x = discrete_gaussian_kernel(img.shape[0], sigma)
50         kern_y = discrete_gaussian_kernel(img.shape[1], sigma)
51     elif kernel == 'sampled':
52         A = 1 / (np.sqrt((2*np.pi)*sigma**2))
53         kern_x = A*signal.gaussian(img.shape[0], sigma)
54         kern_y = A*signal.gaussian(img.shape[1], sigma)
55     else:
56         raise ValueError("Key must be 'discrete' or 'sampled'")
57
58     kernel = np.outer(kern_x, kern_y)
59
60     return signal.fftconvolve(img, kernel, mode='same')
61
62 def discrete_gaussian_kernel(n_samples, sigma):
63     """
64     sigma is the scale, n_samples is the number of samples to compute
65     will return a window centered a zero
66     i.e. arange(-n_samples//2, n_samples//2+1)

```

```

67
68 note! to make this work similarly to fft_gaussian, this uses
69 sigma = np.sqrt(t). Usually you'll find this in terms of t
70
71 by using scipy.special.iv instead we prevent blowups
72 """
73 dom = np.arange(-(n_samples//2), (n_samples//2) + 1)
74 #there should be a scaling parameter alpha but whatever
75 #return np.exp(-t) * iv(dom,t)
76 return ive(dom,sigma**2)

77
78 def fft_dgk(img, sigma, order=0, A=None):
79 """
80 This is the discrete gaussian kernel which is supposedly less crappy
81 than using a sampled gaussian.
82 """
83 m,n = img.shape
84 # i don't know if this will suck if there are odd dimensions
85 kernel = np.outer(discrete_gaussian_kernel(m,sigma**2),
86 discrete_gaussian_kernel(n,sigma**2))

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

89
90 def fft_fdgk(img, sigma):
91 """
92 convolve with discrete gaussian kernel in freq. space
93 """
94 # this would be a lot better since you wouldn't have to deal
95 # with an arbitrary cutoff of size of the discrete kernel
96 # since the freq. space version is just
97 # exp{\alpha*t (cos\theta - 1)}
98 # see formula 22 of lindeberg discrete paper
99
100 pass

101
102 def fft_hessian(image, sigma=1., kernel=None):
103 """
104 a reworking of skimage.feature.hessian_matrix that uses
105 FFT to compute gaussian, which results in a considerable speedup
106
107 INPUT:
108     image - a 2D image (which type?)
109     sigma - coefficient for gaussian blur
110     kernel - input to fft_gaussian
111     gradient - if you've already computed this
112
113 OUTPUT:
114     (Lxx, Lxy, Lyy) - a triple containing three arrays
115         each of size image.shape containing the xx, xy, yy derivatives
116         respectively at each pixel. That is, for the pixel value given
117         by image[j][k] has a calculated 2x2 hessian of
118         [ [Lxx[j][k], Lxy[j][k]], ,
119           [Lxy[j][k], Lyy[j][k]] ]
120 """
121
122 gaussian_filtered = fft_gaussian(image, sigma=sigma, kernel=kernel)
123
124 gradients = np.gradient(gaussian_filtered)
125
126 axes = range(image.ndim)
127
128 H_elems = [np.gradient(gradients[ax0], axis=ax1)
129             for ax0, ax1 in combinations_with_replacement(axes, 2)]
130

```

```

131     return H_elems
132
133
134 def fft_gradient(image, sigma=1.):
135     """ returns gradient norm """
136
137     gaussian_filtered = fft_gaussian(image, sigma=sigma)
138
139     Lx, Ly = np.gradient(gaussian_filtered)
140
141     return np.sqrt(Lx**2 + Ly**2)
142
143
144 def demo(img=None):
145     """
146     old main function for testing.
147
148     This simply tests fft_gaussian on a test image,
149     """
150
151     if img is None:
152         img = img_as_float(camera())
153     else:
154         img = img_as_float(img)
155
156     sample_sigmas = (.5, 2, 8, 30)
157     #sample_sigmas = (.2, 2)
158
159     # build the graphs here side by side
160     # show regular blur, sampled blur, discrete blur, 1d plot of signals
161     # so a 4 by 4 grid
162
163     fig, axes = plt.subplots(nrows=len(sample_sigmas), ncols=4,
164                             figsize=(10, 10))
165
166     for cax, sigma in enumerate(sample_sigmas):
167
168         # convolve the image with a gaussian kernel, one of three ways
169         fft_dgk = fft_gaussian(img, sigma, kernel='discrete')
170         fft_sampled = fft_gaussian(img, sigma, kernel='sampled')
171         xy_sampled = gaussian_filter(img, sigma, mode='constant', cval=0)
172
173         # make the fancy sample
174         N = 80
175         dom = np.arange(-(N//2), N//2 + 1)
176         dgk = discrete_gaussian_kernel(N, sigma)
177         A = np.sqrt(2*np.pi*sigma**2)
178         A = 1 / A
179         sgk = A * signal.gaussian(N+1, sigma)
180
181         axes[cax, 0].imshow(xy_sampled, cmap='gray', vmin=0, vmax=1)
182         axes[cax, 0].set_ylabel(r'$\sigma=$' .format(sigma))
183         #axes[cax, 0].set_title(f'ndi.gaussian_filter,   ={sigma}')
184         #axes[cax, 0].axis('off')
185         axes[cax, 0].set_xticks([])
186         axes[cax, 0].set_yticks([])
187
188         axes[cax, 1].imshow(fft_sampled, cmap='gray', vmin=0, vmax=1)
189         #axes[cax, 1].imshow(fft_sampled, cmap='gray')
190         #axes[cax, 1].set_title(f'fft sampled kernel,   ={sigma}')
191         axes[cax, 1].axis('off')
192
193         axes[cax, 2].imshow(fft_dgk, cmap='gray', vmin=0, vmax=1)
194         #axes[cax, 2].set_title(f'fft discrete kernel,   ={sigma}')

```

```

195     axes[cax, 2].axis('off')
196
197
198     axes[cax, 3].plot(dom, sgk, 'k', dom, dgk, 'g:')
199     #axes[cax, 3].set_title(f'discrete vs. sampled kernel ={sigma}')
200     #axes[cax, 3].axes.set_aspect('equal')
201
202     # set titles for the first column
203     axes[0,0].set_title('(a)')
204     axes[0,1].set_title('(b)')
205     axes[0,2].set_title('(c)')
206     axes[0,3].set_title('(d)')
207
208     plt.tight_layout()
209     plt.show()
210
211 def compare_mae(arr1, arr2):
212
213     assert arr1.shape == arr2.shape
214     return np.abs(arr1 - arr2).sum() / arr1.size
215
216
217 def semigroup_demo(img=None):
218     """
219     the step ones don't look anywhere near as blurred as the initial image
220     in any case! don't use this till it's good!
221     """
222     if img is None:
223         img = img_as_float(camera())
224     else:
225         img = img_as_float(img)
226
227     sigma = 45.
228     n_steps = 2
229     sigmas = (10, 35)
230
231     fft_discrete = fft_gaussian(img, sigma, kernel='discrete')
232     fft_sampled = fft_gaussian(img, sigma, kernel='sampled')
233     xy_sampled = gaussian_filter(img, sigma, mode='constant', cval=0)
234
235     step_discrete = img.copy()
236     step_fft_sampled = img.copy()
237     step_xy_sampled = img.copy()
238
239     #sigma_n = sigma / n_steps
240     #sigma_n = np.power(sigma, 1/n_steps)
241     counter = 0
242     for sigma_n in sigmas:
243         step_discrete = fft_gaussian(step_discrete, sigma_n, kernel='discrete')
244         step_fft_sampled = fft_gaussian(step_fft_sampled, sigma_n,
245                                         kernel='sampled')
246         step_xy_sampled = gaussian_filter(step_xy_sampled, sigma_n,
247                                           mode='constant', cval=0)
248         counter += sigma_n
249         #print(counter, end=' ')
250
251     print()
252     er = int(sigma)
253     crop = np.s_[er:-er, er:-er]
254
255     fft_discrete = fft_discrete[crop]
256     fft_sampled = fft_sampled[crop]
257     xy_sampled = xy_sampled[crop]
258     step_discrete = step_discrete[crop]

```

```

259 step_fft_sampled = step_fft_sampled[crop]
260 step_xy_sampled = step_xy_sampled[crop]
261
262 fig, axes = plt.subplots(ncols=3, nrows=2)
263 axes[0,0].imshow(xy_sampled, vmin=0, vmax=1, cmap='gray')
264 axes[0,0].set_title('(a)')
265 axes[0,0].set_xticks([]), axes[0,0].set_yticks([])
266
267 axes[0,1].imshow(fft_sampled, vmin=0, vmax=1, cmap='gray')
268 axes[0,1].set_title('(b)')
269 axes[0,1].set_xticks([]), axes[0,1].set_yticks([])
270
271 axes[0,2].imshow(fft_discrete, vmin=0, vmax=1, cmap='gray')
272 axes[0,2].set_title('(c)')
273 axes[0,2].set_xticks([]), axes[0,2].set_yticks([])
274
275 axes[1,0].imshow(step_xy_sampled, vmin=0, vmax=1, cmap='gray')
276 axes[1,0].axis('off')
277 axes[1,1].imshow(step_fft_sampled, vmin=0, vmax=1, cmap='gray')
278 axes[1,1].axis('off')
279 axes[1,2].imshow(step_discrete, vmin=0, vmax=1, cmap='gray')
280 axes[1,2].axis('off')
281
282 MSE_sampled = compare_mse(xy_sampled, step_xy_sampled)
283 MSE_fft_sampled = compare_mse(fft_sampled, step_fft_sampled)
284 MSE_discrete = compare_mse(fft_discrete, step_discrete)
285
286 print(f'MSE sampled:{MSE_sampled}')
287 print(f'MSE fft_sampled:{MSE_fft_sampled}')
288 print(f'MSE discrete:{MSE_discrete}')
289 print()
290 #NRMSE_sampled = compare_nrmse(xy_sampled, step_xy_sampled)
291 #NRMSE_fft_sampled = compare_nrmse(fft_sampled, step_fft_sampled)
292 #NRMSE_discrete = compare_nrmse(fft_discrete, step_discrete)
293
294 #print(f'NRMSE sampled:{NRMSE_sampled}')
295 #print(f'MSE fft_sampled:{NRMSE_fft_sampled}')
296 #print(f'NRMSE discrete:{NRMSE_discrete}')
297 #for ax, title in zip(axes.ravel(), ['(a)', '(b)', '(c)', '(d)']):
298 #    ax.axis('off')
299 #    ax.set_title(title)
300
301
302 MAE_sampled = compare_mae(xy_sampled, step_xy_sampled)
303 MAE_fft_sampled = compare_mae(fft_sampled, step_fft_sampled)
304 MAE_discrete = compare_mae(fft_discrete, step_discrete)
305
306 print(f'MAE sampled:{MAE_sampled}')
307 print(f'MAE fft_sampled:{MAE_fft_sampled}')
308 print(f'MAE discrete:{MAE_discrete}')
309
310 plt.show()
311
312 if __name__ == "__main__":
313     from skimage.io import imread
314     from placenta import list_by_quality, get_named_placenta
315     #A = list_by_quality(0)[0]
316     #A = get_named_placenta(A)
317     #A = imread('samples/5.3.02.tiff')
318     A = None
319     demo(A)
320     semigroup_demo(A)

```

listings/merging.py

```
1 #!/usr/bin/env python3
2
3 import numpy as np
4 import numpy.ma as ma
5 from scipy.ndimage import label
6 from skimage.morphology import remove_small_objects
7 import matplotlib.pyplot as plt
8
9 def nz_percentile(A, q, axis=None, interpolation='linear'):
10     """calculate np.percentile(...,q) on an array's nonzero elements only
11
12     Parameters
13     -----
14     A : ndarray
15         matrix from which percentiles will be calculated. Percentiles
16         are calculated on an elementwise basis, so the shape is not important
17     q : a float
18         Percentile to compute, between 0 and 100.0 (inclusive).
19
20     (other arguments): see numpy.percentile docstring
21     ...
22
23     Returns
24     -----
25     out: float
26
27     """
28
29     if ma.is_masked(A):
30         A = A.filled(0)
31
32     return np.percentile(A[A > 0], q, axis=axis, interpolation=interpolation)
33
34
35 def apply_threshold(targets, alphas, return_labels=True):
36     """Threshold targets at each scale, then return max target over all scales.
37
38     A unique alpha can be given for each scale (see below). Return a 2D boolean
39     array, and optionally another array representing what at what scale the max
40     filter response occurred.
41
42     Parameters
43     -----
44     targets : ndarray
45         a 3D array, where targets[:, :, k] is the result of the Frangi filter
46         at the kth scale.
47     alphas : float or array_like
48         a list / 1d array of length targets.shape[-1]. each alphas[k] is a
49         float which thresholds the Frangi response at the kth scale. Due to
50         broadcasting, this can also be a single float, which will be applied
51         to each scale.
52     return_labels : bool, optional
53         If True, return another ndarray representing the scale (see Notes
54         below). Default is True.
55
56     Returns
57     -----
58     out : ndarray, dtype=bool
59         if return labels is true, this will return both the final
60         threshold and the labels as two separate matrices. This is
61         a convenience, since you could easily find labels with
62     labels : ndarray, optional, dtype=uint8
```

```

63     The scale at which the largest filter response was found after
64     thresholding. Element is 0 if no scale passed the threshold,
65     otherwise an int between 1 and targets.shape[-1] See Notes below.
66
67 Notes / Examples
68 -----
69 Despite the name, this does *NOT* return the thresholded targets itself,
70 but instead the maximum value after thresholding. If you wanted the
71 thresholded filter responses alone, you should simply run
72
73 >>>(targets > alphas)*targets
74
75 The optional output ‘labels’ is a 2D matrix indicating where the max filter
76 response occurred. For example, if the label is K, the max filter response
77 will occur at targets[:, :, K-1]. In other words,
78
79 >>>passed, labels = apply_threshold(targets, alphas)
80 >>>targets.max(axis=-1) == targets[:, :, labels - 1 ]
81 True
82
83 It should be noted that returning labels is really just for convenience
84 only; you could construct it as shown in the following example:
85
86 >>>manual_labels = (targets.argmax(axis=-1) + 1)*np.invert(passed)
87 >>>labels == manual_labels
88 True
89
90 Similarly, the standard boolean output could just as easily be obtained.
91 >>>passed == (labels != 0)
92 True
93 """
94
95 # threshold as an array (even if it's a single element) to broadcast
96 alphas = np.array(alphas)
97
98 # if input's just a MxN matrix, expand it trivially so it works below
99 if targets.ndim == 2:
100     targets = np.expand_dims(targets, 2)
101
102 # either there's an alpha for each channel or there's a single
103 # alpha to be broadcast across all channels
104 assert (targets.shape[-1] == alphas.size) or (alphas.size == 1)
105
106 # pixels that passed the threshold at any level
107 passed = (targets >= alphas).any(axis=-1)
108
109 if not return_labels:
110     return passed # we're done already
111
112 wheres = targets.argmax(axis=-1) # get label of where maximum occurs
113 wheres += 1 # increment to reserve 0 label for no match
114
115 # then remove anything that didn't pass the threshold
116 wheres[np.invert(passed)] = 0
117
118 assert np.all(passed == (wheres > 0))
119
120 return passed, wheres
121
122 def sieve_scales(multiscale, high_percentile, low_percentile, min_size=None,
123                  axis=0):
124     """
125     multiscale is a 3 dimensional where 2 dimensions are image and ‘axis’

```

```

127 parameter is which one is the scale space (i.e. resolution). hopefully
128 axis is 0 or 1 (this won't handle stupider cases)
129
130 this gathers points contiguous points at a low threshold and adds them
131 to the output it contains at least only if that blob contains at least one
132 high percentile point.
133
134 min_size is a size requirement can either be an integer or an array of
135 integers """
136
137 assert multiscale.ndim == 3
138
139 if axis in (-1, 2):
140     # this won't change the input, just creates a view
141     V = np.transpose(multiscale, axes=(2, 0, 1))
142
143 elif axis == 0:
144     V = multiscale # just to use the same variable name
145 else:
146     raise ValueError('Please make resolution the first or last dimension.')
147
148 if np.isscalar(min_size):
149     min_size = [min_size for x in range(multiscale.shape[0])]
150
151 # label matrix the size of one of the images
152 sieved = np.zeros(V.shape[1:], dtype=np.int32)
153
154 print('sieving ', end='')
155 for n, v in enumerate(V):
156     print(' ', end='', flush=True)
157
158     if min_size is not None:
159         z = remove_small_objects(v, min_size=min_size[n])
160     else:
161         z = v # relabel to use same variable
162
163     high_thresh = nz_percentile(v, high_percentile)
164     low_thresh = nz_percentile(v, low_percentile)
165
166     labeled, n_labels = label(z > low_thresh)
167     high_passed = (z > high_thresh)
168
169     for lab in range(n_labels):
170         if lab == 0:
171             continue
172         if np.any(high_passed[labeled == lab]):
173             sieved[labeled == lab] = n
174
175 print()
176 return sieved
177
178 def view_slices(multiscale, axis=0, scales=None, cmap='nipy_spectral',
179                 vmin=0, vmax=1.0, outnames=None, show_colorbar=True):
180     """ scales is just to use for a figure title
181     crop before you get in here.
182
183     if outname is an iterable returning filenames, then we'll assume
184     non-interative mode
185     """
186
187     assert multiscale.ndim == 3
188
189     if axis in (-1, 2):
190         # this won't change the input, just creates a view
191         V = np.transpose(multiscale, axes=(2, 0, 1))

```

```

191
192     elif axis == 0:
193         V = multiscale # just to use the same variable name
194     else:
195         raise ValueError('Please make resolution the first or last dimension.')
196
197     if scales is None:
198         scales = [None for x in range(multiscale.shape[0])]
199     if outnames is None:
200         outnames = [None for x in range(multiscale.shape[0])]
201
202     plt.close('all')
203     for v, sigma, outname in zip(V, scales, outnames):
204
205         if outname is None:
206             plt.imshow(v, cmap=cmap, vmin=vmin, vmax=vmax)
207             mng = plt.get_current_fig_manager()
208             mng.window.showMaximized()
209             plt.tight_layout()
210             if sigma is not None:
211                 plt.title(r'$\sigma={:.2f}$'.format(sigma))
212             plt.axis('off')
213             if show_colorbar:
214                 plt.colorbar()
215             plt.tight_layout()
216             plt.show()
217             plt.close()
218
219     else:
220         # save them non interactively with imsave
221         plt.imsave(outname, v, cmap=cmap, vmin=vmin, vmax=vmax)

```

listings/pcsvn.py

```

1 #!/usr/bin/env python3
2
3 from placenta import get_named_placenta
4 from diffgeo import principal_directions
5 from frangi import frangi_from_image
6 from skimage.util import img_as_float
7 import numpy as np
8 from preprocessing import inpaint_hybrid
9
10 from merging import nz_percentile
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 def make_multiscale(img, scales, beta=0.5, gamma=0.5, c=None, dark_bg=True,
23                     find_principal_directions=False, dilate_per_scale=True,
24                     signed_frangi=False, kernel=None, verbose=True,
25                     rescale_frangi=False, gradient_filter=False):
26     """Returns an ordered list of dictionaries for each scale of Frangi info.
27
28

```

```

29 beta, gamma, and c can all be vectors as long as scales or constants
30 if c is None it will be set.
31
32 Each element in the output contains the following info:
33     {'sigma': sigma,
34      'beta': beta,
35      'gamma': gamma,
36      'H': hesh,
37      'F': targets,
38      'k1': k1,
39      'k2': k2,
40      't1': t1, # if find_principal_directions
41      't2': t2 # if find_principal_directions
42  }
43
44 is it necessary to lug all this shit around?
45 """
46
47 # store results of each scale (create as empty list)
48 multiscale = list()
49
50 img = ma.masked_array(img_as_float(img), mask=img.mask)
51
52 vectorize = lambda x: np.repeat(x, len(scales)) if (x is None or np.isscalar(x)) else x
53
54 # vectorize any scalar inputs here
55 beta = vectorize(beta)
56 gamma = vectorize(gamma)
57 c = vectorize(c)
58 print('finding multiscale targets ', end='')
59 for i, (sigma, b, g, cx) in enumerate(zip(scales, beta, gamma, c)):
60
61     print(' ', end=' ')
62
63     if dilate_per_scale:
64         if sigma > 20:
65             radius = int(2*sigma)
66         elif sigma < 3:
67             radius = 12
68         else:
69             radius = int(4*sigma)
70     else:
71         radius = None
72
73     targets, this_scale = frangi_from_image(img, sigma, beta=b, gamma=g,
74                                             c=cx, dark_bg=dark_bg,
75                                             dilation_radius=radius,
76                                             kernel=kernel,
77                                             signed_frangi=signed_frangi,
78                                             return_debug_info=True,
79                                             rescale_frangi=rescale_frangi,
80                                             gradient_filter=gradient_filter)
81
82     if find_principal_directions:
83         # principal directions should only be computed for critical regions
84         # this mask is where PD's will *NOT* be calculated
85         # is targets a masked array?
86         cutoff = nz_percentile(targets, 80)
87         pd_mask = np.bitwise_or(targets < cutoff, img.mask).filled(1)
88         percent_calculated = (pd_mask.size - pd_mask.sum()) / pd_mask.size
89
90     if verbose:
91         print(f"finding PD's for {percent_calculated:.2%} of image"
92               f"anything above vesselness score {cutoff:.6f}")

```

```

93
94         )
95         t1, t2 = principal_directions(img, sigma=sigma, H=this_scale['H'],
96                                         mask=pd_mask)
97
98         # add them to this scale's output
99         this_scale['t1'] = t1
100        this_scale['t2'] = t2
101
102    else:
103        if verbose:
104            print('skipping principal direction calculation')
105
106    # store results as a list of dictionaries
107    multiscale.append(this_scale)
108
109    print()
110
111 def extract_pcsvn(img, filename, scales, beta=0.5, gamma=0.5, c=None,
112                     dark_bg=True, dilate_per_scale=True, verbose=True,
113                     generate_json=True, output_dir=None, kernel=None,
114                     signed_frangi=False, rescale_frangi=False,
115                     gradient_filter=False):
116     """Run PCSVN extraction on the sample given in the file.
117
118     Despite the name, this simply returns the Frangi filter responses at
119     each provided scale without explicitly making any decisions about what
120     is or is not part of the PCSVN.
121
122     As a matter of fact, this function currently just is a wrapper for
123     make_multiscale that logs some output
124     The original main use of this function has kind of bled into
125     extract_NCS_pcsvn.py. that needs fixing. You should load the image
126     outside of this function, do post processing there, pass it inside here
127     with a dictionary of things to add to the json file
128
129     """
130
131     # Multiscale Frangi Filter#####
132
133     # output is a dictionary of relevant info at each scale
134     multiscale = make_multiscale(img, scales, beta=beta, gamma=gamma, c=None,
135                                   find_principal_directions=False,
136                                   dilate_per_scale=dilate_per_scale,
137                                   kernel=kernel, signed_frangi=signed_frangi,
138                                   dark_bg=dark_bg, verbose=verbose,
139                                   rescale_frangi=rescale_frangi,
140                                   gradient_filter=gradient_filter)
141
142     # extract these for logging
143     c = [scale['c'] for scale in multiscale]
144     border_radii = [scale['border_radius'] for scale in multiscale]
145
146     # ignore targets too close to edge of plate
147     # wait are we doing this twice?
148     if dilate_per_scale:
149         if verbose:
150             print('trimming collars of plates (per scale)')
151
152         for i in range(len(multiscale)):
153             f = multiscale[i]['F']
154             # twice the buffer (be conservative!)
155             radius = int(multiscale[i]['sigma'] * 2)

```

```

157     if verbose:
158         print('dilating plate for radius={}'.format(radius))
159         f = dilate_boundary(f, radius=radius, mask=img.mask)
160         # get rid of mask
161         multiscale[i]['F'] = f.filled(0)
162     else:
163         for i in range(len(multiscale)):
164             # get rid of mask
165             multiscale[i]['F'] = multiscale[i]['F'].filled(0)
166     # Make Composite#####
167
168     # get a M x N x n_scales array of Frangi targets at each level
169     F_all = np.dstack([scale['F'] for scale in multiscale])
170
171 if generate_json:
172
173     time_of_run = datetime.datetime.now()
174     timestamp = time_of_run.strftime("%y%m%d_%H%M")
175
176     # numpy arrays have to be turned into lists first
177     vectorize = lambda x: x if x is None or np.isscalar(x) else list(x)
178
179     logdata = {'time': timestamp,
180                'filename': filename,
181                'betas': vectorize(beta),
182                'gammas': vectorize(gamma),
183                'c': vectorize(c),
184                'sigmas': list(scales)}
185
186
187     if dilate_per_scale:
188         logdata['border_radii'] = border_radii
189
190     if output_dir is None:
191         output_dir = 'output'
192
193     base = os.path.basename(filename)
194     *base, suffix = base.split('.')
195     dumpfile = os.path.join(output_dir,
196                            ''.join(base) + '_' + str(timestamp)
197                            + '.json')
198
199     with open(dumpfile, 'w') as f:
200         json.dump(logdata, f, indent=True)
201
202     return F_all, dumpfile
203
204
205 def get_outname_lambda(filename, output_dir=None, timestamp=None):
206     """
207     return a lambda function which can build output filenames
208     """
209
210     if output_dir is None:
211         output_dir = 'output'
212
213     base = os.path.basename(filename)
214     *base, suffix = base.split('.')
215
216     if timestamp is None:
217         time_of_run = datetime.datetime.now()
218         timestamp = time_of_run.strftime("%y%m%d_%H%M")
219
220     outputstub = ''.join(base) + '_' + timestamp + '_{}.' + suffix

```

```

221     return lambda s: os.path.join(output_dir, outputstub.format(s))
222
223
224 def _build_scale_colormap(N_scales, base_colormap, basecolor=(0,0,0,1)):
225     """
226     returns a mpl.colors.ListedColormap with N samples,
227     based on the colormap named "default_colormap" (a string)
228
229     the N colors are given by the default colormap, and
230     basecolor (default black) is added to map to 0.
231     (you could change this, for example, to (1,1,1,1) for white)
232
233     reversed colormaps often work better if the basecolor is black
234     you should make sure there's good contrast between the basecolor
235     and the first color in the colormap
236     """
237
238     map_range = np.linspace(0, 1, num=N_scales)
239
240     colormap = plt.get_cmap(base_colormap)
241
242     colorlist = colormap(map_range)
243
244     # add basecolor as the first entry
245     colorlist = np.vstack((basecolor, colorlist))
246
247     return mpl.colors.ListedColormap(colorlist)
248
249
250 def scale_label_figure(whereis, scales, savefilename=None,
251                       crop=None, show_only=False, image_only=False,
252                       base_cmap='viridis_r', save_colorbar_separate=False,
253                       savecolorbarfile=None, output_dir=None):
254     """
255     crop is a slice object.
256     if show_only, then just plt.show (interactive).
257     if image_only, then this will *not* be printed with the colorbar
258
259     if save_colormap_separate, then the colormap will be saved as a separate
260     file
261     """
262     if crop is not None:
263         whereis = whereis[crop]
264
265     fig, ax = plt.subplots() # not sure about figsize
266     N = len(scales) # number of scales / labels
267
268     tabemap = _build_scale_colormap(N, base_cmap)
269
270     if image_only:
271         plt.imsave(savefilename, whereis, cmap=tabemap, vmin=0, vmax=N)
272         plt.close()
273     else:
274         imgplot = ax.imshow(whereis, cmap=tabemap, vmin=0, vmax=N)
275         # discrete colorbar
276         cbar = plt.colorbar(imgplot)
277
278         # this is apparently hackish, beats me
279         tick_locs = (np.arange(N+1) + 0.5)*(N-1)/N
280
281         cbar.set_ticks(tick_locs)
282         # label each tick with the sigma value
283         scalelabels = [r"\sigma = {:.2f}{}".format(s) for s in scales]
284         scalelabels.insert(0, "(no match)")

```

```

285     # label with their sigma value
286     cbar.set_ticklabels(scalelabels)
287     # ax.set_title(r"Scale ($\sigma$) of maximum vesselness ")
288     plt.tight_layout()
289     # plt.savefig(outname('labeled'), dpi=300)
290     if show_only or (savefilename is None):
291         plt.show()
292     else:
293         plt.savefig(savefilename, dpi=300)
294
295     plt.close()
296
297 if save_colorbar_separate:
298     if savecolorbarfile is None:
299         savecolorbarfile = os.path.join(output_dir, "scale_colorbar.png")
300     fig = plt.figure(figsize=(1, 8))
301     ax1 = fig.add_axes([0.05, 0.05, 0.15, 0.9])
302     tick_locs = (np.arange(N+1) + 0.5)*(N-1)/N
303     scalelabels = [r"$\sigma = {:.2f}$".format(s) for s in scales]
304     scalelabels.insert(0, "n/a")
305     cbar = mpl.colorbar.ColorbarBase(ax1, cmap=tabemap,
306                                     norm=mpl.colors.Normalize(vmin=0,
307                                     vmax=N),
308                                     orientation='vertical',
309                                     ticks=tick_locs)
310     cbar.set_ticklabels(scalelabels)
311     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 from numpy.ma import is_masked
23 from skimage.color import gray2rgb
24 import matplotlib.pyplot as plt
25
26
27 def open_typefile(filename, filetype, sample_dir=None, mode=None):
28     """
29     filetype is either 'mask' or 'trace'
30     mask -> 'L' mode
31     trace -> 'RGB' mode
32     use mode keyword to override this behavior (for example if you

```

```

33     want a binary trace)
34
35     typefiles that aren't the above will be treated as 'L'
36     """
37     # try to open what the mask *should* be named
38     # this should be done less hackishly
39     # for example, if filename is 'ncs.1029.jpg' then
40     # this would set the maskfile as 'ncs.1029.mask.jpg'
41
42     #if filetype not in ("mask", "trace"):
43     #    raise NotImplementedError("Can only deal with mask or trace files.")
44
45     # get the base of filename and build the type filename
46     *base, suffix = filename.split('.')
47     base = ''.join(base)
48     typefile = '.'.join((base, filetype, suffix))
49
50     if sample_dir is None:
51         sample_dir = 'samples'
52
53     typefile = os.path.join(sample_dir, typefile)
54
55     if mode is not None:
56         if filetype == 'mask':
57             mode = 'L'
58         elif filetype in ('ctrace', 'veins', 'arteries'):
59             mode = 'RGB'
60         else:
61             # handle this if you need to?
62             mode = 'L'
63     try:
64         img = imread(typefile, mode=mode)
65
66     except FileNotFoundError:
67         print('Could not find file', typefile)
68         return None
69
70     return img
71
72
73 def open_tracefile(base_filename, as_binary=True,
74                     sample_dir=None):
75     """
76
77     ###width parsing is no longer done here. instead, this function
78     should handle the venous/arterial difference.
79
80     this currently only serves to open the RGB traces as binary
81     files instead of RGB, which is processed later
82
83     #TODO: expand this later to handle arterial traces and venous traces
84     INPUT:
85         base_filename: the name of the base file, not the tracefile itself
86         as_binary: if True
87     """
88
89     if as_binary:
90         mode = 'L'
91     else:
92         mode = 'RGB'
93
94     T = open_typefile(base_filename, 'trace', sample_dir=sample_dir, mode=mode)
95
96

```

```

97     if as_binary:
98         return np.invert(T != 0)
99
100    else:
101        return T
102
103
104 def mimg_as_float(mimg):
105
106    if not ma.is_masked(mimg):
107
108        return img_as_float(mimg)
109
110    else:
111        return ma.masked_array(img_as_float(mimg.data.filled(0)),
112                               mask=mimg.mask)
113
114 def get_named_placenta(filename, sample_dir=None, masked=True,
115                         maskfile=None, mode='L'):
116
117    """This function is to be replaced by a more ingenious/natural
118    way of accessing a database of unregistered and/or registered
119    placental samples.
120
121    Parameters
122    -----
123
124    filename: name of file (including suffix?) but NOT directory
125    masked: return it masked.
126    maskfile: if supplied, this use the file will use a supplied 1-channel
127        mask (where 1 represents an invalid/masked pixel, and 0
128        represents a valid/unmasked pixel. the supplied image must be
129        the same shape as the image. if not provided, the mask is
130        calculated (unless masked=False)
131        the file must be located within the sample directory
132
133    If maskfile is 'None' then this function will look for
134    a default maskname with the following pattern:
135
136        test.jpg -> test.mask.jpg
137        ncs.1029.jpg -> ncs.1029.mask.jpg
138
139    sample_directory: Relative path where sample (and mask file) is located.
140                    defaults to './samples'
141
142    if masked is true (default), this returns a masked array.
143
144    NOTE: A previous logical incongruity has been corrected. Masks should have
145    1 as the invalid/background/mask value (to mask), and 0 as the
146    valid/plate/foreground value (to not mask)
147
148    if sample_dir is None:
149        sample_dir = 'samples'
150
151    full_filename = os.path.join(sample_dir, filename)
152
153    if mode.lower() in ('g', 'green'):
154        # first channel of RGBA (or RGBAI)
155        raw_img = imread(full_filename)[...,1]
156
157    else:
158        raw_img = imread(full_filename, mode=mode)
159
160    if maskfile is None:

```

```

161     # try to open what the mask *should* be named
162     # this should be done less hackishly
163     # for example, if filename is 'ncs.1029.jpg' then
164     # this would set the maskfile as 'ncs.1029.mask.jpg'
165     *base, suffix = filename.split('.')
166     test_maskfile = ''.join(base) + '.mask.' + suffix
167     test_maskfile = os.path.join(sample_dir, test_maskfile)
168     try:
169         mask = imread(test_maskfile, mode='L')
170     except FileNotFoundError:
171         print('Could not find maskfile', test_maskfile)
172         print('Please supply a maskfile. Autogeneration of mask',
173             'files is slow and buggy and therefore not supported.')
174         raise
175     #return mask_background(raw_img)
176 else:
177     # set maskfile name relative to path
178     maskfile = os.path.join(sample_dir, maskfile)
179     mask = imread(maskfile, mode='L')
180
181     return ma.masked_array(raw_img, mask=mask)
182
183
184 def list_by_quality(quality=0, N=None, json_file=None, return_empty=False):
185     """
186     returns a list of filenames that are of quality 'quality'
187     quality is either "good" or 0
188         "OK" or 1
189         "fair" or 2
190         "poor" or 3
191
192     N is the number of placentas to return (will return # of placentas
193     of that quality or N, whichever is smaller)
194
195     if json_name is not None just use that filename directly
196
197     if return_empty then silently failing is OK
198     """
199
200     quality_keys = ('good', 'okay', 'fair', 'poor')
201
202     if quality in quality_keys:
203         pass
204     elif quality in (0, 1, 2, 3):
205         quality = quality_keys[quality]
206     else:
207         try:
208             quality = quality.lower()
209         except AttributeError:
210             if return_empty:
211                 return []
212             else:
213                 print(f'unknown quality {quality}')
214                 raise
215         else:
216             # if no json file is provided, and quality is a string,
217             # just assume it follows a template format
218             if json_file is None:
219                 json_file = f'{quality}-mccs.json'
220
221
222     # if it's still not provided in the main file, it's in the main file
223     if json_file is None:
224         json_file = 'sample-qualities.json'

```

```

225
226     try:
227         with open(json_file, 'r') as f:
228             D = json.load(f)
229     except FileNotFoundError:
230         if return_empty:
231             return list()
232         else:
233             print('cannot find', json_file)
234             raise FileNotFoundError
235
236     if json_file == 'sample-qualities.json':
237         # go one level deep
238         placentas = [k for k in D[quality].keys()]
239     else:
240         placentas = [k for k in D.keys()]
241
242     if N is not None:
243         return placentas[:N]
244     else:
245         return placentas
246
247
248 def check_filetype(filename, assert_png=True, assert_standard=False):
249     """
250     'T-BN8333878.raw.png' returns 'raw'
251     'T-BN8333878.mask.png' returns 'mask'
252     'T-BN8333878.png' returns 'base'
253
254     if assert_png is True, then raise assertion error if the file
255     is not of type png
256
257     if assert_standard, then assert the filetype is
258     mask, base, trace, or raw.
259
260     etc.
261     """
262     basename, ext = os.path.splitext(filename)
263
264     if ext != '.png':
265         if assert_png:
266             assert ext == '.png'
267
268     sample_name, typestub = os.path.splitext(basename)
269
270     if typestub == '':
271         # it's just something like 'T-BN8333878.png'
272         return 'base'
273     elif typestub in ('.mask', '.trace', '.raw', '.ctrace', '.arteries', '.veins', '.ucip'):
274         # return 'mask' or 'trace' or 'raw'
275         return typestub.strip('.')
276     else:
277         print('unknown filetype:', typestub)
278         print('is it a weird filename?')
279
280         print('warning: lookup failed, unknown filetype:' + typestub)
281
282     return typestub
283
284 def list_placentas(label=None, sample_dir=None):
285     """
286     label is the specifier, basically just ''.startswith()
287     only real use is to find all the T-BN* files

```

```

289
290     this is hackish, if you ever decide to use a file other than
291     png then this needs to change
292     """
293
294     if sample_dir is None:
295         sample_dir = 'samples'
296
297     if label is None:
298         label = '' # str.startswith('') is always True
299
300     placentas = list()
301
302     for f in os.listdir(sample_dir):
303
304         if f.startswith(label):
305             # oh man they gotta be png files
306             if check_filetype(f) == 'base':
307                 placentas.append(f)
308
309     return sorted(placentas)
310
311
312 def show_mask(img, mask=None, interactive=False, mask_color=None):
313     """
314     rename this color_mask since showing the mask is just a secondary feature
315     show a masked grayscale image with a dark blue masked region
316
317     custom version of imshow that shows grayscale images with the right
318     colormap and, if they're masked arrays, sets makes the mask a dark blue) a
319     better function might make the grayscale value dark blue (so there's no
320     confusion)
321
322     if interactive, this operates like "plt.imshow"
323     if interactive==False, return the RGB matrix
324
325     if mask provided, add it to the image. (pass img.data instead if you don't
326     want to use the original mask)
327     """
328
329     if mask_color is None:
330         mask_color = (0, 0, 60)
331
332     # if there's no mask at all
333     if (mask is None) and (not is_masked(img)):
334         if interactive:
335             plt.imshow(img, cmap=plt.cm.gray)
336             return # we're done
337         else:
338             # return as an rgb image so output is uniform
339             return gray2rgb(img)
340
341     elif not is_masked(img):
342         # add mask to the image / add to existing mask
343         # if i just rewrite img will it change outside this function?
344         new_img = ma.masked_array(img, mask=mask)
345     else:
346         new_img = img.copy()
347
348     # otherwise, get an RGB array, black where the mask is
349     mimg = gray2rgb(new_img.filled(0))
350
351     # fill masked regions with the mask color
352     mimg[new_img.mask, :] = mask_color

```

```

353
354     if interactive:
355         plt.imshow(mimg)
356     else:
357         return mimg
358
359
360 def _cropped_bounds(img, mask=None):
361
362     if mask is not None:
363
364         img = ma.masked_array(img, mask=mask)
365
366         X, Y = (np.argwhere(np.invert(img.mask)).any(axis=k)).squeeze()
367             for k in (0, 1)
368         )
369
370     if X.size == 0:
371         X = [None, None] # these will slice correctly
372     if Y.size == 0:
373         Y = [None, None]
374
375     return Y[0], Y[-1], X[0], X[-1]
376
377
378 def cropped_args(img, mask=None):
379     """
380     get a slice that would crop image
381     i.e. img[cropped_args(img)] would be a cropped view
382     """
383
384     x0, x1, y0, y1 = _cropped_bounds(img, mask=None)
385
386     return np.s_[x0:x1, y0:y1]
387
388
389 def cropped_view(img, mask=None):
390     """
391     removes entire masked rows and columns from the borders of a masked array.
392     will return a masked array of smaller size
393
394     don't ask me about data
395
396     the name sucks too
397     """
398
399     # find first and last row with content
400     x0, x1, y0, y1 = _cropped_bounds(img, mask=mask)
401
402     return img[x0:x1, y0:y1]
403
404
405 CYAN = [0, 255, 255]
406 YELLOW = [255, 255, 0]
407
408
409 def measure_ncs_markings(ucip_img=None, filename=None, verbose=True):
410     """
411     find location of ucip and resolution of image based on input
412     (similar to perimeter layer in original NCS data set
413
414     Parameters
415     -----
416

```

```

417 ucip_img: an RGB ndarray or None
418     The perimeter layer of an NCS sample (colorations according to the
419     tracing protocol). if None, filename must be included. Default is None.
420 filename:
421     the filename of the SAMPLE (not the ucip image file itself)
422
423 Returns
424 -----
425 m : tuple of ints
426     the coordinates of (the center of) of the umbilical cord point
427     (depicted as a yellow dot) in the original image.
428 resolution: a float
429     measured distance between the two cyan dots
430 """
431
432 if ucip_img is None:
433     ucip_img = open_typefile(filename, 'ucip')
434
435 if ucip_img is None:
436     # if it's still none (no file), return None
437     return None, None
438
439 # just in case it's got an alpha channel, remove it
440 img = ucip_img[:, :, 0:3]
441
442 # given the image img (make sure no alpha channel)
443 # find all cyan pixels (there are two boxes of 3 pixels each and we
444 # just want to extract the middle of each
445 if verbose:
446     print('the image size is {}x{}'.format(img.shape[0], img.shape[1]))
447
448 rulemarks = np.all(img == CYAN, axis=-1)
449
450 # turn into two pixels (these should each be shape (18,))
451 X, Y = np.where(rulemarks)
452
453 assert X.shape == Y.shape
454
455 # if they followed the protocol correctly...
456 if X.size == 18:
457     # get the two pixels at the center of each box
458     A, B = (X[4], Y[4]), (X[13], Y[13])
459 else:
460     # dots are a nonstandard size for some reason. this works too.
461     thinned = morphology.thin(rulemarks)
462     X, Y = np.where(thinned)
463     assert(thinned.sum() == 2) # there should be just two pixels now.
464     A, B = (X[0], Y[0]), (X[1], Y[1])
465
466 ruler_distance = np.sqrt((A[0] - B[0])**2 + (A[1] - B[1])**2)
467 if verbose:
468     print(f'one cm equals {ruler_distance} pixels')
469
470 # the umbilical cord insertion point (UCIP) is a yellow circle, radius 19
471 ucipmarks = np.all(img == YELLOW, axis=-1)
472 X, Y = np.where(ucipmarks)
473
474 # find midpoint of the x & y coordinates
475 assert X.max() - X.min() == Y.max() - Y.min()
476 radius = (X.max() - X.min()) // 2
477
478 mid = (X.min() + radius, Y.min() + radius)
479
480 if verbose:

```

```

481     print('the middle of the UCIP location is', mid)
482     print('the radius outward is', radius)
483     print('the total measurable diameter is', radius*2 + 1)
484
485     return mid, ruler_distance
486
487
488 def add_ucip_to_mask(m, radius=100, mask=None, size_like=None):
489     """
490     - m is a tuple (2x1) representing the (coordinate) midpoint of the UCIP
491     - radius around which to dilate the UCIP is the dilation radius as it
492     works in morphology--this is passed directly to skimage.morphology.disk.
493     thus a circle centered at point m with diameter 2*radius + 1
494     - if no mask is supplied, dilate the point in an array of zeros the shape
495     of 'size_like' (would be the same as passing mask=np.zeros_like(size_like))
496
497     Note: this behaves much faster than binary dilation on the point
498     """
499     if mask is None:
500         if size_like is not None:
501             mask = np.zeros_like(size_like)
502         else:
503             raise ValueError("No mask info supplied!")
504
505     # an empty mask (since we need to merge--we don't want to copy the
506     # zeros of the dilated UCIP -- just the ones!)
507     to_add = np.zeros_like(mask)
508
509     # this is way faster than dilating the point in the matrix,
510     # just set this at the centered point
511
512     # doesn't check for out of bounds stuff. use at your own peril
513     D = morphology.disk(radius)
514     to_add[m[0]-radius:m[0]+radius+1, m[1]-radius:m[1]+radius+1] = D
515
516     # merge with supplied mask
517     return mask | to_add
518
519
520 if __name__ == "__main__":
521     """test that this works on an easy image."""
522
523     test_filename = 'barium1.png'
524
525     img = get_named_placenta(test_filename, maskfile=None)
526
527     print('showing the mask of', test_filename)
528     print('run plt.show() to see masked output')
529
530     show_mask(img, interactive=True)

```

listings/plate_morphology.py

```

1 #!/usr/bin/env python3
2
3 from skimage.morphology import (disk, binary_erosion, binary_dilation,
4                                 convex_hull_image, thin)
5 from skimage.segmentation import find_boundaries, watershed
6
7 from placenta import open_typefile, get_named_placenta
8

```

```

9 import numpy as np
10 import numpy.ma as ma
11
12 def dilate_boundary(img, radius=10, mask=None):
13     """
14         grows the mask by a specified radius of a masked 2D array
15         Manually remove (erode) the outside boundary of a plate.
16         The goal is remove any influence of the zeroed background
17         on reporting derivative information.
18
19     There is varying functionality here (maybe should be multiple functions
20     instead?)
21
22     If img is a masked array and mask=None, the mask will be dilated and a
23     masked array is outputted.
24
25     If img is any 2D array (masked or unmasked), if mask is specified, then
26     the mask will be dilated and the original image will be returned as a
27     masked array with a new mask.
28
29     If the img is None, then the specified mask will be dilated and returned
30     as a regular 2D array.
31
32     """
33
34     if mask is None:
35         # grab the mask from input image
36         # if img is None this will break too but not handled
37         try:
38             mask = img.mask
39         except AttributeError:
40             raise('Need to supply mask information')
41
42     perimeter = find_boundaries(mask, mode='inner')
43
44     maskpad = np.zeros_like(perimeter)
45
46     M,N = maskpad.shape
47     for i,j in np.argwhere(perimeter):
48         # just make a cross shape on each of those points
49         # these will silently fail if slice is OOB thus ranges are limited.
50         maskpad[max(i-radius,0):min(i+radius,M),j] = 1
51         maskpad[i,max(j-radius,0):min(j+radius,N)] = 1
52
53     new_mask = np.bitwise_or(maskpad, mask)
54
55     if img is None:
56         return new_mask # return a 2D array
57     else:
58         # replace the original mask or create a new masked array
59         return ma.masked_array(img, mask=new_mask)
60
61
62 def l2_dist(p,q):
63     return int(np.round(np.sqrt((p[0]-q[0])**2 + (p[1]-q[1])**2)))
64
65
66 def mask_cuts_simple(img, ucip, mask_only=False, in_place=False,
67                      return_success=False):
68     """
69     this covers up the cut with a disc originating at the perimeter of
70     significant radius
71     """
72

```

```

73 cutmarks = np.all(ucip==(0,0,255), axis=-1)
74 B = np.all(ucip==(0,0,255), axis=-1)
75 dilcut = img.copy()
76
77 if not np.any(cutmarks):
78     #print("no cutmarks found on image")
79
80     if return_success:
81         return img, False
82     else:
83         return img
84 else:
85     #print("found a cutmark!")
86     pass
87
88 cutmarks = np.nonzero(cutmarks)
89 # get the first pixel of it (we don't need to be too precise here)
90 G = np.all(ucip==(0,255,0), axis=-1) # perimeter elements
91 cutmarks = np.nonzero(thin(B))
92 perimeter = np.nonzero(G)
93
94 cutinds = np.stack(cutmarks).T
95
96 for P in cutinds:
97
98     # consider larger and larger window sizes
99     for W in [100,200,300]:
100         # consider all perimeter elements within these bounds
101
102         rmin, rmax = max(0, P[0]-W), min(img.shape[0], P[0]+W)
103         cmin, cmax = max(0, P[1]-W), min(img.shape[1], P[1]+W)
104         #window = np.s_[rmin:rmax, cmin:cmax]
105
106         # perimeter indices within the window
107         pinds = [(x,y) for x, y in zip(*perimeter)]
108         if x > rmin and x < rmax and y > cmin and y < cmax
109             ]
110
111         if pinds:
112             break #otherwise increase the size of the window
113 if pinds:
114
115     # max distance to boundary point in the window
116     # we really only need to keep the largest; deque?
117     dists = sorted([(pp, l2_dist(P,pp)) for pp in pinds],
118                   key=lambda t: t[1])
119     r = 2*int(dists[0][1]) + 1 # get largest radius but closest point
120     P = dists[0][0]
121     B = np.zeros_like(img.mask)
122
123     B[cutmarks] = True
124
125     # center a disk of found radius there
126     D = disk(r)
127     winx = max(P[0]-r,0), min(P[0]+r+1,B.shape[0])
128     winy = max(P[1]-r,0), min(P[1]+r+1,B.shape[1])
129
130     try:
131         B[winx[0]:winx[1], winy[0]:winy[1]] = D
132     except ValueError:
133         # they're out of bounds so it's a size mismatch. fix it
134         # by starting/ending D index with opposite sign of the initial
135         # p +/- radius that was out of bounds
136         # for example P[0]-r was -9 and everything else was fine
137         # so you just need to set left side to D[9:,:]
```

```

137     # but you should wrap this up in a function so the three times
138     # you do it here and the one time in ucip all gets the same
139     # code
140     print("too close to the boundary or size mismatch?")
141     success = False
142 else:
143     dilcut[B] = ma.masked
144     success = True
145 else:
146     print("we completely failed to mask the cut. too close to the",
147           "boundary to fit an unmodified disk in. fix this")
148     success = False
149
150 return dilcut, success
151
152
153 def mask_cuts_watershed(img, ucip, mask_only=False, in_place=False,
154                         return_success=False):
155     """
156
157     this doesn't handle any image, io. just provide the ucip img and the
158     base (masked) image and we'll fix the mask
159
160     ucip is the actual RGB array, not the file. do io elsewhere.
161
162     if mask_only, this will simply return the new mask as a 2D boolean array.
163     Otherwise, it returns a masked_array.
164     The cut region will be added to the img's mask. If you really want just the
165     difference, you'll have to run
166     >>>(cut_mark & ~img.mask) yourself.
167
168     yourself.
169
170     If in_place, this changes the mask of the image directly (but still returns
171     a masked array. If mask_only is True, in_place will automatically be set to
172     False to prevent hideous side effects
173
174     if return_success, this function returns True if there was a cutmark found,
175     otherwise false as a second output
176     """
177
178     # get indices where the blue square indicating center of a cut appears
179     cutmarks = np.all(ucip==(0,0,255), axis=-1)
180
181     if not np.any(cutmarks):
182         #print("no cutmarks found on image")
183
184         if return_success:
185             return img, False
186         else:
187             return img
188     else:
189         #print("found a cutmark!")
190         pass
191
192     cutmarks = np.nonzero(cutmarks)
193     # get the first pixel of it (we don't need to be too precise here)
194     X, Y = cutmarks[0][0], cutmarks[1][0]
195
196     # get a value somewhat lower than the value of bg in the cut
197     # (this should be a high number before we take 85%)
198     # sometimes this is in a shadowy region which fucks everything up though
199     #threshold = max(img[cutmarks].mean() * .85, 175)
200     # get the brightest value in a smallish window around the cut * .85

```

```

201 threshold = np.max(img[X-10:X+10, Y-10:Y+10])
202
203 rmin, rmax = max(0, X-100), min(img.shape[0], X+100)
204 cmin, cmax = max(0, Y-100), min(img.shape[1], Y+100)
205 cutregion = np.s_[rmin:rmax, cmin:cmax] # get a window around the mark
206
207 # mark inside of the placenta with label 2, original mask and cutmarks with
208 # label 1, and the rest with 0 (i dunno)
209 markers = np.zeros(img.shape, dtype='int32')
210 markers[img.filled(255) < threshold] = 2
211 markers[img.mask] = 1
212 markers[cutmarks] = 1
213
214 # perform watershedding on the thresholded image to fill in the cut with
215 # label 1
216 cutfix = watershed(img.filled(255) < threshold, markers=markers)
217
218 # this is a waste considering the in_place, but eh
219 new_mask = img.mask.copy()
220
221 new_mask[cutregion] = (cutfix[cutregion] == 1)
222
223 if mask_only:
224     out = new_mask
225
226 elif not in_place:
227     out = ma.masked_array(img, mask=new_mask)
228
229 else:
230     # will this work?
231     img[new_mask] = ma.masked
232
233     out = img
234
235     # now return succeed if asked to
236     if return_success:
237         return out, True
238
239     else:
240         return out
241
242
243 if __name__ == "__main__":
244
245     # DEMO FOR SHOWING OFF DILATE_BOUNDARY EFFECT
246
247     from placenta import get_named_placenta
248     from frangi import frangi_from_image
249     import matplotlib.pyplot as plt
250
251     import os.path
252
253     dest_dir = 'demo_output'
254     img = get_named_placenta('T-BN0164923.png')
255
256     sigma = 3
257     radius = 25
258
259     inset = np.s_[800:1000, 500:890]
260
261     D = dilate_boundary(img, radius=radius)
262
263
264

```

```

265 Fimg = frangi_from_image(img, sigma, dark_bg=False, dilation_radius=None)
266 FD = frangi_from_image(D, sigma, dark_bg=False)
267 FDinv = frangi_from_image(D, sigma, dark_bg=True)
268 Finv = frangi_from_image(img, sigma, dark_bg=True, dilation_radius=None)
269
270 fig, axes = plt.subplots(ncols=2, nrows=3)
271
272 axes[0,0].imshow(img[inset].filled(0), cmap=plt.cm.gray)
273 axes[0,1].imshow(D[inset].filled(0), cmap=plt.cm.gray)
274 axes[1,0].imshow(Fimg[inset].filled(0), cmap=plt.cm.nipy_spectral)
275 axes[1,1].imshow(FD[inset].filled(0), cmap=plt.cm.nipy_spectral)
276 axes[2,0].imshow(Finv[inset].filled(0), cmap=plt.cm.nipy_spectral)
277 axes[2,1].imshow(FDinv[inset].filled(0), cmap=plt.cm.nipy_spectral)
278
279 for a in axes.ravel():
280     # get rid of all the labels
281     plt.setp(a.get_xticklabels(), visible=False)
282     plt.setp(a.get_yticklabels(), visible=False)
283
284 # lol matlab
285 for i in range(5):
286     fig.tight_layout()
287
288 plt.savefig(os.path.join(dest_dir, "boundary_dilation_demo.png"), dpi=300)

```

listings/postprocessing.py

```

1 #!/usr/bin/env python3
2 """
3 doing things to the Frangi targets, i.e. feeding them into other algorithms
4 """
5
6 from skimage.filters import sobel
7 from skimage.morphology import remove_small_holes, remove_small_objects, thin
8 from frangi import frangi_from_image
9 from merging import apply_threshold, nz_percentile
10 from plate_morphology import dilate_boundary
11 from skimage.segmentation import random_walker
12 import numpy as np
13
14 def random_walk_fill(img, Fmax, high_thresh, low_thresh, dark_bg):
15     """
16         there are a lot of decisions to make here but it's just a demo
17         so who cares
18     """
19
20     s = sobel(img)
21     s = dilate_boundary(s, mask=img.mask, radius=20)
22
23     finv = frangi_from_image(img, sigma=0.8, beta=0.5, dark_bg=(not dark_bg),
24                             dilation_radius=20)
25
26     finv_thresh = (finv > nz_percentile(finv, 50)).filled(0)
27     margins = remove_small_objects(finv_thresh, min_size=32)
28
29     markers = np.zeros(img.shape, dtype=np.int32)
30     markers[Fmax < low_thresh] = 1
31
32     margins_added = (margins | (Fmax > high_thresh))
33     #margins_added = remove_small_holes(margins_added, area_threshold=50)
34

```

```

35     markers[Fmax < low_thresh] = 1
36
37     markers[margins_added] = 2
38
39     rw = random_walker(1-Fmax, markers, beta=1000)
40
41     approx_rw = (rw == 2)
42
43     return approx_rw, markers, margins_added
44
45 def random_walk_scalewise(F, high_thresh, return_labels=False):
46
47     print('doing scalewise random walk', end=' ')
48     V = np.transpose(F, axes=(2, 0, 1))
49     W = np.zeros(V.shape, np.bool)
50     for n, v in enumerate(V):
51         print(' ', end='', flush=True)
52         markers = np.zeros(v.shape, np.int32)
53         markers[v == 0] = 1
54         # this could be a vector too
55         markers[v > high_thresh] = 2
56         W[n] = (random_walker(1-v, markers) == 2)
57     print()
58     if return_labels:
59         return W.any(axis=0), W.argmax(axis=0)
60     else:
61         return W.any(axis=0)
62

```

listings/preprocessing.py

```

1 #!/usr/bin/env python3
2
3 # TODO: refactor this so inpaint_glare is the main function that takes
4 #       a keyword argument strategy='hybrid' or whatever then you can run
5 #       >>>for s in ['mean_window', 'median_boundary', 'biharmonic', 'hybrid']:
6 #           timeit.timeit('inpaint_glare(img, strategy=s)', globals=globals())
7 #
8 #       ... but it's annoying since you'll need a way to pass args to the
9 #       particular strategy
10
11 from skimage.morphology import binary_dilation, disk, remove_small_objects
12 from skimage.restoration import inpaint_biharmonic
13 import numpy as np
14 import numpy.ma as ma
15 from scipy.ndimage import label
16 from skimage.util import img_as_float
17 from skimage.segmentation import find_boundaries
18 from plate_morphology import dilate_boundary
19
20 def inpaint_glare(img, threshold=175, window_size=15, mask=None):
21     """
22         img is a masked array type uint [0,255]
23     """
24
25     # bool array, true where glare
26     if mask is None:
27         glared = mask_glare(img, threshold=threshold, mask_only=True)
28     else:
29         glared = mask
30
31
```

```

32     B = ma.masked_array(img, mask=glared) # masked background *and* glare
33     new_img = img.copy() # copy values of original image (will rewrite)
34     d = int(window_size)
35
36     for j, k in zip(*np.where(glared)):
37         # rewrite all glared pixels with the mean of nonmasked elements
38         # in a window_size window. (this doesn't check OoB, be careful!)
39         new_img[j, k] = B[j-d:j+d, k-d:k+d].compressed().mean()
40
41     return new_img
42
43
44 def inpaint_with_boundary_median(img, threshold=175, mask=None):
45     """
46     mask glare pixels, then replace by the median value on the mask's boundary
47     """
48     if mask is None:
49         glared = mask_glare(img, threshold=threshold, mask_only=True)
50     else:
51         glared = mask
52
53     B = ma.masked_array(img, mask=glared)
54
55     new_img = img.copy() # copy values of original image (will rewrite)
56     bounds = find_boundaries(glared)
57     lb, _ = label(bounds)
58     fill_vals = np.zeros_like(img.data)
59
60     # for each boundary of masked region, find the median value of the img
61     for lab in range(1, lb.max()+1):
62         inds = np.where(lb == lab)
63         fill_vals[inds] = nz_median(B[inds])
64
65     # label masked regions together with their boundaries (they'll be
66     # connected)
67     lm, _ = label(np.logical_or(glared, lb != 0))
68
69     # fill the masked areas with the corresponding fill value
70     for lab in range(1, lm.max()+1):
71         inds = np.where(lm == lab)
72         # find locations of filled values corresponding to this label
73         # median in case there's overlapped regions? (sloppy)
74         replace_value = nz_median(fill_vals[inds])
75
76         if replace_value == 0:
77             raise
78
79         fill_vals[inds] = replace_value
80
81     # now fill in the values
82     new_img[glared] = fill_vals[glared]
83
84     return new_img
85
86 def nz_median(A):
87
88     if ma.is_masked(A):
89         relevant = A[A > 0].compressed()
90     else:
91         relevant = A[A > 0]
92
93     return np.median(relevant)
94
95

```

```

96 def inpaint_hybrid(img, threshold=175, min_size=64, boundary_radius=10):
97     """
98     use biharmonic inpainting in larger, inner areas (important stuff)
99     and median inpainting in smaller areas and along boundary
100    """
101
102    glare = mask_glare(img, threshold=threshold, mask_only=True)
103
104    glare_inside = dilate_boundary(glare, mask=img.mask,
105                                    radius=boundary_radius).filled(0)
106
107    large_glare = remove_small_objects(glare_inside, min_size=min_size,
108                                       connectivity=2)
109    small_glare = np.logical_and(glare, np.invert(large_glare))
110
111    # inpaint smaller and less important values with less expensive method
112    inpainted = inpaint_with_boundary_median(img, mask=small_glare)
113    hybrid = img_as_float(inpainted) # scale 0 to 1
114
115    # inpaint larger regions with biharmonic inpainting
116    large_inpainted = inpaint_biharmonic(img.filled(0), mask=large_glare)
117
118    # now overwrite with these values
119    hybrid[large_glare] = large_inpainted[large_glare]
120
121    # put on old image mask
122    return ma.masked_array(hybrid, mask=img.mask)
123
124 def inpaint_with_biharmonic(img, threshold=175):
125     """
126     use biharmonic inpainting *all* glare
127     """
128     glare = mask_glare(img, threshold=threshold, mask_only=True)
129     inpainted = inpaint_biharmonic(img_as_float(img.filled(0)), mask=glare)
130
131     if ma.is_masked(img):
132         return ma.masked_array(inpainted, mask=img.mask)
133     else:
134         return inpainted
135
136 def mask_glare(img, threshold=175, mask_only=False):
137     """
138     for demoing purposes, with placenta.show_mask
139
140     if mask_only, just return the mask. Otherwise return a copy of img with
141     that added to the mask. If you want the original mask to be ignored,
142     just pass img.filled(0) ya doofus
143
144     threshold is expected to be of the same dtype as img *unless# it assumes
145     its default value, in which case the threshold will be converted to a float
146     """
147
148     # if img.dtype is floating but threshold value is still the default
149     # this could be generalized
150     if np.issubdtype(img.dtype, np.floating) and (threshold == 175):
151         threshold = 175 / 255
152     # region to inpaint
153     inp = (img > threshold)
154
155     # get a larger area around the specks
156     inp = binary_dilation(inp, selem=disk(2))
157
158     # remove anything large
159     #inp = white_tophat(inp, selem=disk(3))

```

```

160
161 if mask_only:
162     return inp
163 else:
164     # both the original background *and* these new glared regions
165     # are masked
166     return ma.masked_array(img, mask=inp)
167
168
169 DARK_RED = np.array([103, 15, 23]) / 255.
170
171 # test it on a particularly bad sample
172 if __name__ == "__main__":
173
174     from placenta import get_named_placenta, show_mask
175     import matplotlib.pyplot as plt
176
177     filename = 'T-BN0204423.png' # a particularly glary sample
178     img = get_named_placenta(filename)
179
180     img = ma.masked_array(img_as_float(img), mask=img.mask)
181     crop = np.s_[150:500, 150:800] # indices to zoom in on the region
182     zoom = np.s_[300:380, 300:380] # even smaller region
183
184     inset = zoom # which view to use
185
186     masked = mask_glare(img) # for viewing
187     inpainted = inpaint_glare(img)
188     minpainted = inpaint_with_boundary_median(img)
189     hinpainted = inpaint_hybrid(img)
190     binpainted = inpaint_with_biharmonic(img)
191
192     # view the closeup like this
193     minpainted_view = show_mask(minpainted, interactive=False,
194                                 mask_color=DARK_RED)
195     inpainted_view = show_mask(inpainted, interactive=False,
196                                mask_color=DARK_RED)
197     masked_view = show_mask(masked, interactive=False,
198                                mask_color=DARK_RED)
199     img_view = show_mask(img, interactive=False,
200                                mask_color=DARK_RED)
201     hinpainted_view = show_mask(hinpainted, interactive=False,
202                                mask_color=DARK_RED)
203     binpainted_view = show_mask(binpainted, interactive=False,
204                                mask_color=DARK_RED)
205
206     # view them all next to each other
207
208     fig, axes = plt.subplots(ncols=3, nrows=2)
209
210     axes[0,0].imshow(img_view[inset])
211     axes[0,1].imshow(masked_view[inset])
212     axes[0,2].imshow(inpainted_view[inset])
213     axes[1,0].imshow(minpainted_view[inset])
214     axes[1,1].imshow(binpainted_view[inset])
215     axes[1,2].imshow(hinpainted_view[inset])
216
217     for a in axes.ravel():
218         # get rid of all the labels
219         plt.setp(a.get_xticklabels(), visible=False)
220         plt.setp(a.get_yticklabels(), visible=False)
221
222     # lol matlab
223     for i in range(5):

```

```

224     fig.tight_layout()
225
226     IMGS = np.vstack((
227         np.hstack((img_view, masked_view, inpainted_view)),
228         np.hstack((minpainted_view, binpainted_view, hinpainted_view))))
229
230     # THEN IMSAVE
231
232     # plt.imsave('preprocessing_comparison_cropped.png', IMGS)
233     # plt.imsave('preprocessing_comparison_zoomed.png', IMGS)
234
235     # if it's zoomed, then rescale the output in GIMP to 4x

```

listings/process_NCS_xcfs.py

```

1 #!/usr/bin/env python
2
3 """
4 This should be a plugin to take images from the folder NCS_vessel_GIMP_xcf
5 and create trace, mask, and backgrounded images from each xcf file.
6
7 to use:
8 chmod +x and then copy or link to ~/gimp-2.x/plug-ins/
9 """
10
11 from gimpfu import *
12 import os.path
13 from functools import partial
14
15 #basefile, ext = os.path.splitext(xcffile)
16
17 def _outname(base, s=None):
18
19     #base = base.split("_", maxsplit=1)[0]
20     if s is None:
21         stubs = (base, 'png')
22     else:
23         stubs = (base, s, 'png')
24     file
25     filename = '.'.join(stubs)
26
27     return os.path.join(os.getcwd(), filename)
28
29 # get active image
30 def process_NCS_xcf(timg, tdrawable):
31     img = timg
32     basename, _ = os.path.splitext(img.name) # split off extension .xcf
33     basename = basename.split("_")[0] # only get T-BN-kjlksf part
34     print "*" * 80
35     print '\n\n'
36
37     print "Processing " , img.name
38     # generate output names easier
39     outname = partial(_outname, base=basename)
40
41     # get coordinates of the center
42     cx, cy = img.height // 2 , img.width // 2
43
44     # disable the undo buffer
45     img.disable_undo()
46
47     #perimeter = pdb.gimp_image_get_layer_by_name(img, 'perimeter')

```

```

48
49 for layer in img.layers:
50     if layer.name.lower() in ('perimeter', 'perimeters'):
51         # .copy() has optional arg of "add_alpha_channel"
52         mask = layer.copy()
53         break
54 else:
55     print "Could not find a perimeter layer."
56     print "Layers of this image are:"
57     for n,layer in enumerate(img.layers):
58         print "\t", n, ":", layer.name
59     print "Skipping this file."
60
61 return
62
63 for layer in img.layers:
64     layer.visible = False
65
66 mask.name = "mask" # name the new layer
67 img.add_layer(mask,0) # add in position 0 (top)
68
69 pdb.gimp_layer_flatten(mask) # Remove Alpha Channel.
70
71 # save the annotated perimeter file (for calculations later)
72 pdb.gimp_file_save(img,mask, outname(s="ucip"), '')
73
74 # remove unneeded annotations from mask layer
75 # color exchange yellow & blue to black
76 pdb.plug_in_exchange(img,mask,255,255,0,0,0,0,1,1,1)
77 pdb.plug_in_exchange(img,mask,0,0,255,0,0,0,1,1,1)
78
79 # set FG color to black (for tools, not of image)
80 gimp.set_foreground(0,0,0)
81
82 # Bucket Fill Inside black (center pixel is hopefully fine,
83 # do rest manually
84 pdb.gimp_edit_bucket_fill(mask,0,0,100,0,0,cx,cy)
85
86 # Color Exchange Green to White.
87 pdb.plug_in_exchange(img,mask,0,255,0,255,255,255,1,1,1)
88
89 # Color Exchange Cyan (00ffff) to White.
90 pdb.plug_in_exchange(img,mask,0,255,255,255,255,255,1,1,1)
91
92 # Export Layer as Image called "f".mask.png
93 pdb.gimp_file_save(img,mask, outname(s="mask"), '')
94
95 # invert (so exterior is now black)
96 pdb.gimp_invert(mask)
97 mask.mode = DARKEN_ONLY_MODE # the constant 9
98
99 # set bottom layer (placenta) to visible
100 raw = img.layers[-1]
101 raw.visible = True
102
103 # now make a new layer called 'raw_img' from visible
104 base = pdb.gimp_layer_new_from_visible(img,img,'base')
105 img.add_layer(base,0)
106 pdb.gimp_file_save(img , base, outname(s=None) , '')
107
108 # now get rid of mask and save the raw image
109 mask.visible = False
110 pdb.gimp_file_save(img, base, outname(s='raw') , '')
111

```

```

112
113 # now make the other one visible (this is dumb)
114 for layer in img.layers:
115     if layer.name.lower() in ("arteries", "veins"):
116         layer.visible = True
117     else:
118         layer.visible = False
119 # now with these two visible, merge them and add layer
120 trace = pdb.gimp_layer_new_from_visible(img, img, 'trace')
121 img.add_layer(trace, 0)
122
123 pdb.gimp_layer_flatten(trace) # remove alpha channel
124
125 # don't turn binary anymore
126 #pdb.gimp_desaturate(trace) # turn to grayscale
127 #pdb.gimp_threshold(trace,255,255) # anything not 255 turns black
128
129 pdb.gimp_file_save(img, trace, outname(s='ctrace'), '')
130
131 # now extract an each type individually.
132 found = 0
133 for subtype in ("arteries", "veins"):
134     for layer in img.layers:
135         if layer.name.lower() == subtype:
136             layer.visible = True
137             pdb.gimp_layer_flatten(layer) # remove alpha channel
138             pdb.gimp_file_save(img, layer, outname(s=subtype), '')
139             layer.mode = 9 # set to darken only (for merging)
140             found += 1
141         else:
142             layer.visible = False
143 if found < 2:
144     print "WARNING! Could not find appropriate artery/vein layers."
145
146
147 print "Saved."
148
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/scaleddecay.py

```

1 # coding: utf-8
2 from placenta import get_named_placenta, list_by_quality
3 list_by_quality(0)
4 filename = _[-2]
5 img = get_named_placenta(filename)
6 import matplotlib.pyplot as plt
7 import numpy as np

```

```

8 plt.imshow(img)
9 pls.show()
10 plt.show()
11 filename = list_by_quality(0)[0]
12 img = get_named_placenta(filename)
13 from placenta import cropped_args
14 crop = cropped_args(img)
15 img[crop]
16 plt.imshow(img[crop])
17 plt.show()
18 from hfft import fft_gaussian
19 get_ipython().run_line_magic('pinfo', 'fft_gaussian')
20 C = fft_gaussian(img, 32, 'discrete')
21 B = fft_gaussian(img, 5, 'discrete')
22 A = fft_gaussian(img, .12, 'discrete')
23 plt.imshow(A)
24 plt.show()
25 A = fft_gaussian(img, .25, 'discrete')
26 plt.imshow(A)
27 plt.show()
28 plt.imshow(B)
29 plt.show()
30 plt.imshow(C)
31 plt.show()
32 gA = np.gradient(A)
33 gA
34 gB = np.gradient(B)
35 gC = np.gradient(C)
36 gA.shape
37 gA[0].shape
38 aa = lambda g: np.sqrt((1+g[0]*g[0] + g[0]*g[0]))
39 plt.imshow(aa(gA))
40 plt.show()
41 from plate_morphology import dilate_boundary
42 from functools import partial
43 dilate = partial(dilate_boundary, mask=img.mask)
44 dilate(aa(gA), 20)
45 plt.imshow(_)
46 plt.show()
47 dilate(aa(gA), 20).filled(0)
48 plt.show()
49 plt.imshow(_)
50 plt.show()
51 Aaa = dilate(aa(gA), 20).filled(0)
52 Aaa.max()
53 Aaa.min()
54 Aaa[~img.mask].min()
55 Aaa[img.mask].min()
56 Aaa[img.mask].max()
57 Aaa[~img.mask].max()
58 Baa = dilate(aa(gB), 20).filled(0)
59 Caa = dilate(aa(gC), 20).filled(0)
60 plt.imshow(Baa)
61 plt.show()
62 plt.imshow(Caa)
63 plt.show()
64 Caa.min()
65 Caa[~img.mask].min()
66 Caa == 0
67 plt.imshow(_)
68 plt.show()
69 Caa[~img.mask].max()
70 Caa[~img.mask].min()
71 Caa[~img.mask].argmin()

```

```

72 dilate_boundary(img.mask, 20)
73 dilate_boundary(img.mask, 20, mask_only=True)
74 get_ipython().run_line_magic('pinfo', 'dilate_boundary')
75 dilate_boundary(img, radius=20)
76 dil = _.mask.copy()
77 dil
78 plt.imshow(dil)
79 plt.show()
80 Caa[~dil].argmin()
81 Caa[~dil]
82 Caa[~dil].min()
83 Caa[~dil].max()
84 Baa[~dil].max()
85 Baa[~dil].min()
86 Aaa[~dil].min()
87 Aaa[~dil].max()
88 plt.imshow(Caa)
89 plt.show()
90 plt.imshow(Aaa)
91 plt.show()
92 #for sigma in np.logspace(-3, 8, base=2, num=20):
93 #    D = fft_gaussian(img, sigma, 'discrete')
94 #    gD = np.gradient(D)
95 #    Daa = dilate(aa(dG), 20).filled(1)
96 #    print(Daa[~dil].min(), Daa[~dil].max())
97 aas = list()
98 for sigma in np.logspace(-3, 8, base=2, num=20):
99     D = fft_gaussian(img, sigma, 'discrete')
100    gD = np.gradient(D)
101    Daa = dilate(aa(dG), 20).filled(1)
102    aas.append(Daa)
103    print(f"sigma={sigma:.3f}", "min: {:.6f}, max:{:.6f}".format(
104        Daa[~dil].min(), Daa[~dil].max()))
105
106 for sigma in np.logspace(-3, 8, base=2, num=20):
107    D = fft_gaussian(img, sigma, 'discrete')
108    gD = np.gradient(D)
109    Daa = dilate(aa(gD), 20).filled(1)
110    aas.append(Daa)
111    print(f"sigma={sigma:.3f}", "min: {:.6f}, max:{:.6f}".format(
112        Daa[~dil].min(), Daa[~dil].max()))
113
114
115 for sigma in np.logspace(-4, 8, base=2, num=50):
116    D = fft_gaussian(img, sigma, 'discrete')
117    gD = np.gradient(D)
118    Daa = dilate(aa(gD), 20).filled(1)
119    aas.append(Daa)
120    print(f"sigma={sigma:.3f}", "min: {:.6f}, max:{:.6f}".format(
121        Daa[~dil].min(), Daa[~dil].max()))
122
123
124 bb = lambda g: np.linalg.norm(np.array([
125     [[1+g[1]**2, -g[0]*g[1]],
126     [-g[0]*g[1], 1 + g[0]**2]]))
127 bb = lambda g: np.linalg.norm(np.array([
128     [[1+g[1]**2, -g[0]*g[1]],
129     [-g[0]*g[1], 1 + g[0]**2]]))
130 bb(gA)
131 get_ipython().set_next_input('man np.linalg.norm');get_ipython().run_line_magic('pinfo', 'np.linalg.norm')
132 ginv = lambda g: np.array([
133     [[1+g[1]**2, -g[0]*g[1]],
134     [-g[0]*g[1], 1 + g[0]**2]])
135

```

```

136 ginv(gA)
137 _.shape
138 G[:, :, 34, 289]
139 ginvA = _101
140 ginvA[:, :, 340, 289]
141 bb = lambda g: np.sqrt((1+g[1]**2)**2 + 2*(g[0]*g[1])**2 + (1+g[0]**2)**2)
142 bb(gA)
143 _.shape
144 plt.imshow(bb)
145 _.shape
146 bb(gA)
147 plt.imshow(_)
148 plt.show()
149 bb(gA) / aa(gA)
150 plt.imshow(_)
151 plt.show()
152 dilate(bb(gA) / aa(gA))
153 plt.imshow(dilate(bb(gA) / aa(gA)).filled(0))
154 plt.show()
155 bb = lambda g: np.sqrt((1+g[1]**2)**2 + 2*(g[0]*g[1])**2 + (1+g[0]**2)**2)
156 plt.imshow(dilate(bb(gA), 20))
157 plt.show()
158 bb(gA)[~dil].min()
159 bb(gA)[~dil].max()
160 (bb(gA) / aa(gA))[~dil].min()
161 (bb(gA) / aa(gA))[~dil].max()
162 (bb(gA) / aa(gA))
163 plt.imshow(dilate(_, 20).filled(1.414))
164 plt.show()
165 for sigma in np.logspace(-4, 8, base=2, num=50):
166     D = fft_gaussian(img, sigma, 'discrete')
167     gD = np.gradient(D)
168     Daa, Dbb = aa(gD), bb(gD)
169     Dcc = Daa / Dbb
170     aas = Daa[~dil].min(), Daa[~dil].max()
171     bbs = Dbb[~dil].min(), Dbb[~dil].max()
172     ccs = Dcc[~dil].min(), Dcc[~dil].max()
173     print(f"sigma={sigma:.3f}",
174           "\tmin: {:.6f},\tmax:{:.6f}".format(aas),
175           "\tmin: {:.6f},\tmax:{:.6f}".format(bbs),
176           "\tmin: {:.6f},\tmax:{:.6f}".format(ccs), sep='\n')
177
178 for sigma in np.logspace(-4, 8, base=2, num=50):
179     D = fft_gaussian(img, sigma, 'discrete')
180     gD = np.gradient(D)
181     Daa, Dbb = aa(gD), bb(gD)
182     Dcc = Daa / Dbb
183     aas = Daa[~dil].min(), Daa[~dil].max()
184     bbs = Dbb[~dil].min(), Dbb[~dil].max()
185     ccs = Dcc[~dil].min(), Dcc[~dil].max()
186     print(f"sigma={sigma:.3f}",
187           "\tmin: {:.6f},\tmax:{:.6f}".format(*aas),
188           "\tmin: {:.6f},\tmax:{:.6f}".format(*bbs),
189           "\tmin: {:.6f},\tmax:{:.6f}".format(*ccs), sep='\n')
190
191
192
193 for sigma in np.logspace(-4, 8, base=2, num=50):
194     D = fft_gaussian(img, sigma, 'discrete')
195     gD = np.gradient(D)
196     Daa, Dbb = aa(gD), bb(gD)
197     Dcc = Dbb / Daa
198     aas = Daa[~dil].min(), Daa[~dil].max()

```

```

200 bbs = Dbb[~dil].min(), Dbb[~dil].max()
201 ccs = Dcc[~dil].min(), Dcc[~dil].max()
202 print(f"sigma={sigma:.3f}",
203     "\tmin: {:.6f},\tmax:{:.6f}" .format(*aas),
204     "\tmin: {:.6f},\tmax:{:.6f}" .format(*bbs),
205     "\tmin: {:.6f},\tmax:{:.6f}" .format(*ccs), sep='\n')
206
207 data = []
208 data = list()
209 for sigma in np.logspace(-4, 8, base=2, num=50):
210     D = fft_gaussian(img, sigma, 'discrete')
211     gD = np.gradient(D)
212     Daa, Dbb = aa(gD), bb(gD)
213     Dcc = Dbb / Daa
214     aas = Daa[~dil].min(), Daa[~dil].max()
215     bbs = Dbb[~dil].min(), Dbb[~dil].max()
216     ccs = Dcc[~dil].min(), Dcc[~dil].max()
217     print(f"sigma={sigma:.3f}",
218         "\tmin: {:.6f},\tmax:{:.6f}" .format(*aas),
219         "\tmin: {:.6f},\tmax:{:.6f}" .format(*bbs),
220         "\tmin: {:.6f},\tmax:{:.6f}" .format(*ccs), sep='\n')
221     data.append(
222         [sigma, *aas, *bbs, *ccs])
223
224 import pandas
225 table = pandas.DataFrame(data)
226 print(table)
227 from hfft import fft_hessian
228 get_ipython().run_line_magic('pinfo', 'fft_hessian')
229 helems = fft_hessian(img, sigma=1, kernel='discrete')
230 np.sqrt(helems[0]**2 + 2*helems[1]**2 + helems[2]**2)
231 _.shape
232 plt.imshow(np.sqrt(helems[0]**2 + 2*helems[1]**2 + helems[2]**2))
233 plt.show()
234 data = list()
235 for sigma in np.logspace(-4, 8, base=2, num=50):
236     D = fft_gaussian(img, sigma, 'discrete')
237     gD = np.gradient(D)
238     Daa, Dbb = aa(gD), bb(gD)
239     Dcc = Dbb / Daa
240     h = fft_hessian(img, sigma, 'discrete')
241     hnorm = np.sqrt(h[0]**2 + 2*h[1]**2 + h[2]**2)
242     Lnorm = hnorm*Dcc
243     aas = Daa[~dil].min(), Daa[~dil].max()
244     bbs = Dbb[~dil].min(), Dbb[~dil].max()
245     ccs = Dcc[~dil].min(), Dcc[~dil].max()
246     dds = hnorm[~dil].min(), hnorm[~dil].max()
247     lls = Lnorm[~dil].min(), Lnorm[~dil].max()
248     print(f"sigma={sigma:.3f}",
249         "\tmin: {:.6f},\tmax:{:.6f}" .format(*aas),
250         "\tmin: {:.6f},\tmax:{:.6f}" .format(*bbs),
251         "\tmin: {:.6f},\tmax:{:.6f}" .format(*ccs),
252         "\tmin: {:.6f},\tmax:{:.6f}" .format(*dds),
253         "\tmin: {:.6f},\tmax:{:.6f}" .format(*lls), sep='\n')
254     data.append(
255         [sigma, *aas, *bbs, *ccs, *dds, *lls])
256
257 data
258 table = pandas.DataFrame(data)
259 table
260 table.columns
261 help(table.columns)

```

```

263 plt.imshow(Lnorm)
264 plt.show()
265 Ls = list()
266 for sigma in np.logspace(-4, 8, base=2, num=50):
267     D = fft_gaussian(img, sigma, 'discrete')
268     gD = np.gradient(D)
269     Daa, Dbb = aa(gD), bb(gD)
270     Dcc = Dbb / Daa
271     h = fft_hessian(img, sigma, 'discrete')
272     hnorm = np.sqrt(h[0]**2 + 2*h[1]**2 + h[2]**2)
273     Lnorm = hnorm*Dcc
274     Ls.append(Lnorm)
275     aas = Daa[~dil].min(), Daa[~dil].max()
276     bbs = Dbb[~dil].min(), Dbb[~dil].max()
277     ccs = Dcc[~dil].min(), Dcc[~dil].max()
278     dds = hnorm[~dil].min(), hnorm[~dil].max()
279     lls = Lnorm[~dil].min(), Lnorm[~dil].max()
280     print(f"sigma={sigma:.3f}",
281           "\tmin: {:.6f},\tmax:{:.6f}".format(*aas),
282           "\tmin: {:.6f},\tmax:{:.6f}".format(*bbs),
283           "\tmin: {:.6f},\tmax:{:.6f}".format(*ccs),
284           "\tmin: {:.6f},\tmax:{:.6f}".format(*dds),
285           "\tmin: {:.6f},\tmax:{:.6f}".format(*lls), sep='\n')
286 #data.append(
287 #    [sigma, *aas, *bbs, *ccs, *dds, *lls])
288
289 L[0]
290 Ls[0]
291 plt.imshow(_)
292 plt.show()
293 for Lnorm, sigma in zip(Ls, np.logspace(-4,8, base=2,num=50)):
294     plt.imshow(Lnorm[crop], cmap='nipy_spectral')
295     mng = plt.get_current_fig_manager()
296     mng.window.showMaximized()
297     plt.colorbar()
298     plt.title(r'Lnorm $\sigma={:.3f}$'.format(sigma))
299     plt.axis('off')
300     plt.tight_layout()
301     plt.show()
302     plt.close('all')
303
304 for Lnorm, sigma in zip(Ls, np.logspace(-4,8, base=2,num=50)):
305     L = dilate(Lnorm, min(20,int(sigma))).filled(0)
306     plt.imshow(Lnorm[crop], cmap='nipy_spectral')
307     mng = plt.get_current_fig_manager()
308     mng.window.showMaximized()
309     plt.colorbar()
310     plt.title(r'Lnorm $\sigma={:.3f}$'.format(sigma))
311     plt.axis('off')
312     plt.tight_layout()
313     plt.show()
314     plt.close('all')
315
316 for Lnorm, sigma in zip(Ls, np.logspace(-4,8, base=2,num=50)):
317     L = dilate(Lnorm, min(20,int(sigma))).filled(0)
318     plt.imshow(L[crop], cmap='nipy_spectral')
319     mng = plt.get_current_fig_manager()
320     mng.window.showMaximized()
321     plt.colorbar()
322     plt.title(r'Lnorm $\sigma={:.3f}$'.format(sigma))
323     plt.axis('off')
324     plt.tight_layout()
325     plt.show()
326     plt.close('all')

```

```

327
328 for Lnorm, sigma in zip(Ls, np.logspace(-4,8, base=2, num=50)):
329     L = dilate(Lnorm, max(20,int(sigma))).filled(0)
330     plt.imshow(L[crop], cmap='nipy_spectral')
331     mng = plt.get_current_fig_manager()
332     mng.window.showMaximized()
333     plt.colorbar()
334     plt.title(r'Lnorm $\sigma={:.3f}$'.format(sigma))
335     plt.axis('off')
336     plt.tight_layout()
337     plt.show()
338     plt.close('all')
339
340 for Lnorm, sigma in zip(Ls, np.logspace(-4,8, base=2, num=50)):
341     L = dilate(Lnorm, max(20,int(sigma))).filled(0)
342     plt.imshow(L[crop], cmap='nipy_spectral')
343     mng = plt.get_current_fig_manager()
344     mng.window.showMaximized()
345     plt.colorbar()
346     plt.title(r'Lnorm $\sigma={:.3f}$'.format(sigma))
347     plt.axis('off')
348     plt.tight_layout()
349     plt.show()
350     plt.close('all')
351
352 for Lnorm, sigma in zip(Ls, np.logspace(-4,8, base=2, num=50)):
353     L = dilate(Lnorm, max(20,int(2*sigma))).filled(0)
354     plt.imshow(L[crop], cmap='nipy_spectral')
355     mng = plt.get_current_fig_manager()
356     mng.window.showMaximized()
357     plt.colorbar()
358     plt.title(r'Lnorm $\sigma={:.3f}$'.format(sigma))
359     plt.axis('off')
360     plt.tight_layout()
361     plt.show()
362     plt.close('all')
363
364 print(table)
365 table
366 table + 2
367 table[:,0]
368 table[0]
369 table / table[0]
370 T = np.array(table)
371 T
372 T.shape
373 T.dtype
374 T / T[:,0]
375 T / T[...,:,0]
376 T[..., -1] / T[...,0]
377 T[..., -1] / np.sqrt(T[...,0])
378 T[..., -1] * np.sqrt(T[...,0])
379 T[..., -1] * T[...,0]
380 T[..., -1] * T[...,0]**2
381 table
382 T[..., -3] * T[...,0]**2
383 T[..., -3] * T[...,0]
384 T[..., -4] * T[...,0]
385 T[..., -4] / T[...,0]
386 T[..., -4] / T[...,0] > .01
387 which =
388 T[...,0][which]
389 (T[...,-4] / T[...,0]) > .001
390 scales

```

```

391 scales = np.logspace(-4,8,num=50,base=2)
392 scales
393 (T[...,-3] / T[...,0]) > .001
394 (T[...,-3] / T[...,0]) > .005
395 scales[_]
396 (T[...,-3] / T[...,0]) > .001
397 scales
398 scales
399 (T[...,-3] / T[...,0]) > .001
400 scales[_]
401 (T[...,-3] / T[...,0]) > .05
402 T[...,-3]
403 (T[...,-3] / T[...,0])
404 (T[...,-3] / T[...,0]**2)
405 (T[...,-3] *np.sqrt(1/T[...,0]))
406 (T[...,-4] / T[...,0]) > .05
407 table
408 from frangi import frangi_from_image
409 g[0]*g[1]
410 g[0]**g[1]
411 gA[0]**gA[1]
412 plt.imshow(_)
413 plt.show()

```

listings/scale_sweep_demo.py

```

1 #!/usr/bin/env python3
2
3 from placenta import (get_named_placenta, list_placentas, _cropped_bounds,
4                        cropped_view, cropped_args, show_mask)
5 from frangi import frangi_from_image
6
7 import numpy as np
8 import numpy.ma as ma
9
10 from plate_morphology import dilate_boundary
11
12 import os.path
13 import matplotlib.pyplot as plt
14 import matplotlib as mpl
15 from itertools import product
16
17 # pick two samples and two insets (should be the same size)
18 demo1 = 'BN2315363', np.s_[370:670, 530:930]
19 demo2 = 'BN5280796', np.s_[150:450, 530:930]
20
21 make_individual = False
22
23 for sample_name, inset_slice in (demo1, demo2):
24     img = get_named_placenta(f'T-{sample_name}.png')
25
26     crop = cropped_args(img)
27
28     F, fi = list(), list() # make some empty lists to store for inspection
29
30     scales = [0.2, 0.8, 1.0, 2.0, 4.0, 6.0, 8.0, 16.0]
31     CMAP = plt.cm.nipy_spectral
32     cmin, cmax = (0, 0.4)
33
34     for n, sigma in enumerate(scales):
35         R = max(int(sigma**3), 10) # only really necessary for signed
36         target = frangi_from_image(img, sigma, dark_bg=False,

```

```

37                                     signed_frangi=False, dilation_radius=R)
38 plate = target[crop].filled(0)
39 inset = target[inset_slice].filled(0)
40 F.append(plate)
41 fi.append(inset)
42
43 if not make_individual:
44     continue
45
46 # else this might be nice for the actual thesis defense
47 for label in ['plate', 'inset']:
48     if label == 'inset':
49         printable = inset
50     else:
51         printable = plate
52
53 plt.imshow(printable, cmap=CMAP)
54 plt.title(r'$\sigma={:.2f}$'.format(sigma))
55 plt.tight_layout()
56 c = plt.colorbar()
57 c.set_ticks = np.linspace(cmin, cmax, num=len(scales)+1)
58 plt.clim(cmin, cmax)
59 plt.axis('off')
60 outname = f'demo_output/scale_sweep_{sample_name}_{label}_{n}.png'
61 plt.savefig(outname, dpi=300, bbox_inches='tight')
62 print('saved', outname)
63 plt.close()
64 # now make a stitched together version
65 for label in ['plate', 'inset']:
66     if label == 'inset':
67         L = fi
68         imgview = img[inset_slice].filled(0)
69         figsize = (12, 6)
70
71     else:
72         L = F
73         imgview = img[crop].filled(0)
74         figsize = (12, 9)
75 #adjust this manually depending on how many scales you end up using!
76
77 nrows, ncols = 2, 4
78 fig, axes = plt.subplots(nrows=nrows, ncols=ncols, figsize=figsize)
79
80 for n, (i, j) in enumerate(product(range(nrows), range(ncols))):
81
82     if n == 0:
83         axes[i,j].imshow(imgview, cmap=plt.cm.gray)
84         axes[i,j].set_title('raw')
85     else:
86         im = axes[i,j].imshow(L[n], cmap=CMAP, vmin=cmin, vmax=cmax)
87         axes[i,j].set_title(r'$\sigma={:.2f}$'.format(scales[n]))
88
89     plt.setp(axes[i,j].get_xticklabels(), visible=False)
90     plt.setp(axes[i,j].get_yticklabels(), visible=False)
91
92     fig.subplots_adjust(top=0.954, bottom=0.025, left=0.010,
93                         right=0.989, hspace=0.0, wspace=0.0)
94
95 plt.savefig(f'demo_output/scale_sweep_stitch_{sample_name}_{label}.png',
96             dpi=300)
97
98 cfile = 'demo_output/scale_sweep_colorbar.png'
99 if os.path.isfile(cfile):
100     continue # no need to make another

```

```

101
102     fig = plt.figure(figsize=(figsize[0],2))
103     ax1 = fig.add_axes([0.15, 0.25, 0.75, 0.5])
104     cbar = mpl.colorbar.ColorbarBase(ax1, cmap=CMAP,
105                                     norm=mpl.colors.Normalize(cmin,cmax),
106                                     orientation='horizontal')
107     plt.savefig(cfile) # don't set dpi maybe it won't be so small and weird

```

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 import itertools
8 from collections import deque
9
10 def rgb_to_widths(T):
11     """
12         this will take an RGB trace image (MxNx3) and return a 2D (MxN)
13         "labeled" trace corresponding to the traced pixel length.
14         there is no distinguishing between arteries and vessels
15
16         it's preferable to do this in real-time so only one tracefile
17         needs to be stored (making the sample folder less cluttered)
18         although obviously at the expense of storing a larger image
19         which is only needed for visualization purposes.
20
21     Input:
22         T: a MxNx3 RGB (uint8) array, where the colorations are
23             assumed as described in NOTES below.
24
25     Output:
26         widthtrace: a MxN array whose inputs describe the width of the
27             vessel (in pixels), see NOTES.
28
29     Notes:
30
31         The correspondence is as follows:
32         3 pixels: "#ff006f", # magenta
33         5 pixels: "#a80000", # dark red
34         7 pixels: "#a800ff", # purple
35         9 pixels: "#ff00ff", # light pink
36         11 pixels: "#008aff", # blue
37         13 pixels: "#8aff00", # green
38         15 pixels: "#ffc800", # dark yellow
39         17 pixels: "#ff8a00", # orange
40         19 pixels: "#ff0015" # bright red
41
42         According to the original tracing protocol, the traced vessels are
43         binned into these 9 sizes. Vessels with a diameter smaller than 3px
44         are not traced (unless they're binned into 3px).
45
46         Note: this does *not* deal with collisions. If you pass anything
47         with addition (blended colors) as the ctraces are, you will have
48         trouble, as those will not be registered as any of the colors above
49         and will thus be ignored. If you want to handle data from both
50         arterial *and* venous layers, you should do so outside of this
51         function.
52         """

```

```

53
54     # a 2D picture to fix in with the pixel widths
55     W = np.zeros_like(T[:, :, 0])
56
57     for pix, color in TRACE_COLORS.items():
58
59         #ignore pixelwidths outside the specified range
60         # get the 2D indices that are that color
61         idx = np.where(np.all(T == color, axis=-1))
62         W[idx] = pix
63
64
65     return W
66
67 def merge_widths_from_traces(A_trace, V_trace, strategy='minimum'):
68     """
69     combine the widths from two RGB-traces A_trace and V_trace
70     and return one width matrix according to 'strategy'
71
72     Parameters
73     -----
74     A_trace: ndarray
75         an MxNx3 matrix, where each pixel (along the
76         last dimension) is an RGB triplet (i.e. each entry
77         is an integer between [0,256]). The colors each
78         correspond to those in TRACE_COLORS, and (255,255,255)
79         signifies "no vessel". This will normally correspond to
80         the sample's arterial trace.
81     V_trace: ndarray
82         an MxNx3 matrix the same shape and other
83         requirements as A_trace (see above). This will normally
84         correspond to the sample's venous trace.
85     strategy: keyword string
86         when A_trace and V_trace coincide at some entry,
87         this is the merging strategy. It should be a keyword
88         of one of the following choices:
89
90         "minimum": take the minimum width of the two traces
91             (default). this is the sensible option if you
92             are filtering out larger widths.
93         "maximum": take the maximum width of the two traces
94         "artery" or "A" or "top": take the width from A_trace
95         "vein" or "V" or "bottom": take the width from V_trace
96
97     Returns
98     -----
99     W : ndarray
100    a width-matrix where each entry is a number 0 (no vessel), 3,5,7,...19
101
102 Notes
103 -----
104 Since arteries grow over the veins on the PCSVN and are generally easier
105 to extract, it might be preferable to indicate "arteries". In reality,
106 each strategy is a compromise, and only by keeping track of both would
107 you get the complete picture.
108
109 No filtering out widths is done here.
110 """
111 assert A_trace.shape == V_trace.shape
112
113 A = rgb_to_widths(A_trace)
114 V = rgb_to_widths(V_trace)
115
116 # collisions (where are widths both reported)

```

```

117 c = (A!=0)& (V!=0)
118
119 W = np.maximum(A,V) # get the nonzero value
120 if strategy == 'maximum':
121     pass # already done, else rewrite the collisions
122 elif strategy in ('arteries', 'A', 'top'):
123     W[c] = A[c]
124 elif strategy in ('veins', 'V', 'bottom'):
125     W[c] = V[c]
126 else:
127     if strategy != 'minimum':
128         print(f"Warning: unknown merge strategy: {strategy}")
129         print("Defaulting to minimum strategy")
130
131 W[c] = np.minimum(A[c], V[c])
132
133 return W
134
135 def filter_widths(W, widths=None, min_width=3, max_width=19):
136 """
137 Filter a width matrix, removing widths according to rules.
138
139 This function will take a 2D matrix of vessel widths and
140 remove any widths outside a particular range (or alternatively,
141 that are not included in a particular list)
142
143 Should be roughly as easy as doing it by hand, except that you
144 won't have to rewrite the code each time.
145
146 Inputs:
147
148 W: a width matrix (2D matrix with elements 0,3,5,7,...19
149
150 min_width: widths below this will be excluded (default is
151     3, the min recorded width). assuming these
152     are ints
153
154 max_width: widths above this will be excluded (default is
155     19, the max recorded width)
156
157 widths: an explicit list of widths that should be returned.
158     in this case the above min & max are ignored.
159     this way you could include widths = [3, 17, 19] only
160 """
161
162 Wout = W.copy()
163 if widths is None:
164     Wout[W < min_width] = 0
165     Wout[W > max_width] = 0
166
167 else:
168     # use numpy.isin(T, widths) but that's only in version 1.13 and up
169     # of numpy this is basically the code for that though
170     to_keep = np.in1d(W, widths, assume_unique=True).reshape(W.shape)
171     Wout[~to_keep] = 0
172
173 return Wout
174
175 TRACE_COLORS = {
176     3: (255, 0, 111),
177     5: (168, 0, 0),
178     7: (168, 0, 255),
179     9: (255, 0, 255),
180     11: (0, 138, 255),

```

```

181     13: (138, 255, 0),
182     15: (255, 200, 0),
183     17: (255, 138, 0),
184     19: (255, 0, 21)
185 }
186
187
188 def widths_to_rgb(w, show_non_matches=False):
189     """Convert width matrix back to RGB values.
190
191     For display purposes/convenience. Return an RGB matrix
192     converting back from [3,5,7, ..., 19] -> TRACE_COLORS
193
194     this doesn't do any rounding (i.e. it ignores anything outside of
195     the default widths), but maybe you'd want to?
196     """
197     B = np.zeros((w.shape[0], w.shape[1], 3))
198
199     for px, rgb_triplet in TRACE_COLORS.items():
200         B[w == px, :] = rgb_triplet
201
202     if show_non_matches:
203         # everything in w not found in TRACE_COLORS will be black
204         B[w == 0, :] = (255, 255, 255)
205     else:
206         non_filled = (B == 0).all(axis=-1)
207
208         B[non_filled,:] = (255,255,255) # make everything white
209
210     # matplotlib likes the colors as [0,1], so....
211     return B / 255.
212
213
214 def _hex_to_rgb(hexstring):
215     """
216     there's a function that does this in matplotlib.colors
217     but its scaled between 0 and 1 but not even as an
218     array so this is just as much work
219
220     ##TODO rewrite everything so this is useful if it's not been
221     rewritten already.
222     """
223     triple = hexstring.strip("#")
224     return tuple(int(x, 16) for x in (triple[:2], triple[2:4], triple[4:]))
225
226
227 def skeletonize_trace(T, T2=None):
228     """
229     if T is a boolean matrix representing a trace, then thin it
230
231     if T is an RGB trace, then register it according to the
232     tracing protocol then thin it
233
234     if T2 is provided, do the same thing to T2 and then merge the two
235     """
236     if T.ndim == 3:
237         trace = (rgb_to_widths(T) > 0) # booleanize it
238     else:
239         trace = T.astype('bool')
240
241     thinned = thin(trace)
242
243     if T2 is None:
244         return thinned

```

```

245
246     else:
247         # do the same thing to second trace and merge it
248         if T2.ndim == 3:
249             trace_2 = (rgb_to_widths(T2) > 0) # booleanize it
250             thinned_2 = thin(trace_2)
251
252         return np.logical_or(thinned, thinned_2)
253
254
255 def confusion(test, truth, bg_mask=None, colordict=None, tint_mask=True):
256     """
257     distinct coloration of false positives and negatives.
258
259     colors output matrix with
260         true_pos if test[-] == truth[-] == 1
261         true_neg if test[-] == truth[-] == 0
262         false_neg if test[-] == 0 and truth[-] == 1
263         false_pos if test[-] == 1 and truth[-] == 0
264
265     if colordict is supplied: you supply a dictionary of how to
266     color the four cases. Spec given by the default below:
267
268     if tint mask, then the mask is overlaid on the image, not replacing totally
269     colordict = {
270         'TN': (247, 247, 247), # true negative
271         'TP': (0, 0, 0) # true positive
272         'FN': (241, 163, 64), # false negative
273         'FP': (153, 142, 195), # false positive
274         'mask': (247, 200, 200) # mask color (not used in MCC calculation)
275     }
276
277
278     if colordict is None:
279         colordict = {
280             'TN': (247, 247, 247), # true negative# 'f7f7f7'
281             'TP': (0, 0, 0), # true positive # '000000'
282             'FN': (241, 163, 64), # false negative # 'f1a340' orange
283             'FP': (153, 142, 195), # false positive # '998ec4' purple
284             'mask': (247, 200, 200) # mask color (not used in MCC calculation)
285         }
286
287         colordict = {
288             'TN': (49,49,49), # true negative# 'f7f7f7'
289             'TP': (0, 0, 0), # true positive # '000000'
290             'FN': (201,53,108), # false negative # 'f1a340' orange
291             'FP': (0,112,163), # false positive # '998ec4' purple
292             'mask': (247, 200, 200) # mask color (not used in MCC calculation)
293         }
294     #TODO: else check if mask is specified and add it as color of TN otherwise
295
296     true_neg_color = np.array(colordict['TN'], dtype='f')/255
297     true_pos_color = np.array(colordict['TP'], dtype='f')/255
298     false_neg_color = np.array(colordict['FN'], dtype='f') /255
299     false_pos_color = np.array(colordict['FP'], dtype='f')/255
300     mask_color = np.array(colordict['mask'], dtype='f') /255
301
302     assert test.shape == truth.shape
303
304     # convert to bool
305     test, truth = test.astype('bool'), truth.astype('bool')
306
307     # RGB array size of test and truth for output
308     output = np.zeros((test.shape[0], test.shape[1], 3), dtype='f')

```

```

309
310     # truth conditions
311     true_pos = (test==truth & truth)
312     true_neg = (test==truth & ~truth)
313     false_neg = (truth & ~test)
314     false_pos = (test & ~truth)
315
316     output[true_pos,:] = true_pos_color
317     output[true_neg,:] = true_neg_color
318     output[false_pos,:] = false_pos_color
319     output[false_neg,:] = false_neg_color
320
321     # try to find a mask
322     if bg_mask is None:
323         try:
324             bg_mask = test.mask
325         except AttributeError:
326             # no mask is specified, we're done.
327             return output
328
329     # color the mask
330     if tint_mask:
331         output[bg_mask,:] += mask_color
332         output[bg_mask,:] /= 2
333     else:
334         output[bg_mask,:] = mask_color
335
336     return output
337
338
339 def compare_trace(approx, trace=None, filename=None,
340                   sample_dir=None, colordict=None):
341     """
342     compare approx matrix to trace matrix and output a confusion matrix.
343     if trace is not supplied, open the image from the tracefile.
344     if tracefile is not supplied, filename must be supplied, and
345     tracefile will be opened according to the standard pattern
346
347     colordict are parameters to pass to confusion()
348
349     returns a matrix
350     """
351
352     # load the tracefile if not supplied
353     if trace is None:
354         if filename is not None:
355             try:
356                 trace = open_typefile(filename, 'trace')
357             except FileNotFoundError:
358                 print("No trace file found matching ", filename)
359                 print("no trace found. generating dummy trace.")
360                 trace = np.zeros_like(approx)
361             else:
362                 print("no trace supplied/found. generating dummy trace.")
363                 trace = np.zeros_like(approx)
364
365     C = confusion(approx, trace, colordict=colordict)
366
367     return C
368
369
370 def mcc(test, truth, bg_mask=None, score_bg=False, return_counts=False):
371     """
372     Matthews correlation coefficient

```

```

373     returns a float between -1 and 1
374     -1 is total disagreement between test & truth
375     0 is "no better than random guessing"
376     1 is perfect prediction
377
378     bg_mask is a mask of pixels to ignore from the statistics
379     for example, things outside the placental plate will be counted
380     as "TRUE NEGATIVES" when there wasn't any chance of them not being
381     scored as negative. therefore, it's not really a measure of the
382     test's accuracy, but instead artificially pads the score higher.
383
384     setting bg_mask to None when test and truth are not masked
385     arrays should give you this artificially inflated score.
386     Passing score_bg=True makes this decision explicit, i.e.
387     any masks (even if supplied) will be ignored, and your count of
388     false positives will be inflated.
389
390     """
391
392     true_pos = ((test == truth) & truth)
393     true_neg = ((test == truth) & ~truth)
394     false_neg = (truth & ~test)
395     false_pos = (test & ~truth)
396
397     if score_bg:
398         # take the classifications above as they are (nothing is masked)
399         pass
400     else:
401         # if no specified mask, check the test array itself?
402         if bg_mask is None:
403             try:
404                 bg_mask = test.mask
405             except AttributeError:
406                 # no mask is specified, we're done.
407                 bg_mask = np.zeros_like(test)
408
409         # only get stats in the plate
410         true_pos[bg_mask] = 0
411         true_neg[bg_mask] = 0
412         false_pos[bg_mask] = 0
413         false_neg[bg_mask] = 0
414
415     # now tally
416     TP = true_pos.sum()
417     TN = true_neg.sum()
418     FP = false_pos.sum()
419     FN = false_neg.sum()
420
421     if not score_bg:
422         total = np.sum(~bg_mask)
423     else:
424         total = test.size
425
426     #print('TP: {} \t TN: {} \nFP: {} \tFN: {}'.format(TP, TN, FP, FN))
427     #print('TP+TN+FN+FP={} \n\ttotal pixels={}'.format(TP+TN+FP+TN, total))
428     # prevent potential overflow
429     denom = np.sqrt(TP+FP)*np.sqrt(TP+FN)*np.sqrt(TN+FP)*np.sqrt(TN+FN)
430
431     if denom == 0:
432         # set MCC to zero if any are zero
433         m_score = 0
434     else:
435         m_score = ((TP*TN) - (FP*FN)) / denom
436

```

```

437     if return_counts:
438         return m_score, (TP,TN,FP,FN)
439     else:
440         return m_score
441
442 def mean_squared_error(A,B):
443     """
444     get mean squared error between two matrices of the same size
445
446     input:
447         A, B : two ndarrays of the same size.
448
449     output:
450
451         mse:    a single number.
452     """
453
454
455     try:
456         mse = ((A-B)**2).sum() / A.size
457
458     except ValueError:
459         print("inputs must be of the same size")
460         raise
461
462     return mse
463
464 def chain_lengths(iterable):
465
466     pos, s = 0, 0
467
468     for b, g in itertools.groupby(iterable):
469
470         if not b:
471             # alternative if the bottom doesn't work or something
472             #d = deque(enumerate(g,1), maxlen=1)
473             #pos += d[0][0] if d else 0
474
475             pos += sum((1 for i in g if not i))
476
477         else:
478
479             s = sum(g)
480
481             yield pos, s
482
483             pos += s
484
485     if not s:
486         # so it will return something even if iterable is empty
487         yield 0, 0
488
489
490 def _longest_chain_1d(iterable):
491     """ will return a tuple of ind, length
492     where ind is the position in the iterable the chain starts and length is the
493     length of the chain
494     """
495     return max(chain_lengths(iterable), key=lambda x: x[1])
496
497
498 def longest_chain(arr, axis):
499     """Find where the longest chain of boolean values and occurs across an array
500     and also return its length

```

```

501 """
502
503 C = np.apply_along_axis(_longest_chain_1d, axis, arr.astype('bool'))
504 start_inds, chain_lens = np.split(C, 2, axis)
505
506 return np.squeeze(start_inds), np.squeeze(chain_lens)
507
508
509 if __name__ == "__main__":
510
511 import matplotlib.pyplot as plt
512 from skimage.data import binary_blobs
513
514 A = binary_blobs()
515 B = binary_blobs()
516
517 true_neg_color = np.array([247, 247, 247], dtype='f') # 'f7f7f7'
518 true_pos_color = np.array([0, 0, 0], dtype='f') # '000000'
519 false_neg_color = np.array([241, 163, 64], dtype='f')# 'f1a340'
520 false_pos_color = np.array([153, 142, 195], dtype='f') # '998ec4'
521
522 C = confusion(A,B)
523
524 fig, (ax0, ax1, ax2) = plt.subplots(nrows=1,
525                                         ncols=3,
526                                         figsize=(8, 2.5),
527                                         sharex=True,
528                                         sharey=True)
529
530
531 ax0.imshow(A, cmap='gray')
532 ax0.set_title('A')
533 ax0.axis('off')
534 ax0.set_adjustable('box-forced')
535
536 ax1.imshow(B, cmap='gray')
537 ax1.set_title('B')
538 ax1.axis('off')
539 ax1.set_adjustable('box-forced')
540
541 ax2.imshow(C)
542 ax2.set_title('confusion matrix of A and B')
543 ax2.axis('off')
544 ax2.set_adjustable('box-forced')
545
546 fig.tight_layout()

```

listings/signed_sweep_demo.py

```

1#!/usr/bin/env python3
2
3 from placenta import (get_named_placenta, list_placentas, _cropped_bounds,
4                         cropped_view, cropped_args, show_mask)
5 from frangi import frangi_from_image
6
7 import numpy as np
8 import numpy.ma as ma
9
10 from plate_morphology import dilate_boundary
11
12 import os.path
13 import matplotlib.pyplot as plt

```

```

14 import matplotlib as mpl
15 from itertools import product
16
17 # pick two samples and two insets (should be the same size)
18 demo1 = 'BN2315363', np.s_[370:670, 530:930]
19 demo2 = 'BN5280796', np.s_[150:450, 530:930]
20
21 for sample_name, inset_slice in (demo1, demo2):
22     img = get_named_placenta(f'T-{sample_name}.png')
23
24     crop = cropped_args(img)
25
26     F, fi = list(), list() # make some empty lists to store for inspection
27
28     #scales = np.logspace(-3, 4, num=8, base=2)
29     scales = [0.2, 0.8, 1.0, 2.0, 4.0, 6.0, 8.0, 16.0]
30     CMAP = plt.cm.Spectral
31     cmin, cmax = (-0.4, 0.4)
32
33     for n, sigma in enumerate(scales):
34         R = max(int(sigma**3), 10)
35         target = frangi_from_image(img, sigma, dark_bg=False,
36                                     signed_frangi=True, dilation_radius=R)
37         plate = target[crop].filled(0)
38         inset = target[inset_slice].filled(0)
39         F.append(plate)
40         fi.append(inset)
41         #for label in ['plate', 'inset']:
42         #    if label == 'inset':
43         #        printable = inset
44         #    else:
45         #        printable = plate
46
47         # plt.imshow(printable, cmap=CMAP)
48         # plt.title(r'$\sigma$={:.2f}'.format(sigma))
49         # plt.tight_layout()
50         # c = plt.colorbar()
51         # c.set_ticks = np.linspace(cmin, cmax, num=len(scales)+1)
52         # plt.clim(cmin, cmax)
53         # plt.axis('off')
54         # outname = f'demo_output/signsweep_{sample_name}_{label}_{n}.png'
55         # plt.savefig(outname, dpi=300, bbox_inches='tight')
56         # print('saved', outname)
57         # plt.close()
58
59     # now make a stitched together version
60     for label in ['plate', 'inset']:
61         if label == 'inset':
62             L = fi
63             imgview = img[inset_slice].filled(0)
64             figsize = (12, 6)
65
66         else:
67             L = F
68             imgview = img[crop].filled(0)
69             figsize = (12, 9)
70         #adjust this manually depending on how many scales you end up using!
71
72         nrows, ncols = 2, 4
73         fig, axes = plt.subplots(nrows=nrows, ncols=ncols, figsize=figsize)
74
75         for n, (i, j) in enumerate(product(range(nrows), range(ncols))):
76             if n == 0:

```

```

78         axes[i,j].imshow(imgview, cmap=plt.cm.gray)
79         axes[i,j].set_title('raw')
80     else:
81         im = axes[i,j].imshow(L[n], cmap=CMAP, vmin=cmin, vmax=cmax)
82         axes[i,j].set_title(r'$\sigma={:.2f}$'.format(scales[n]))
83
84     plt.setp(axes[i,j].get_xticklabels(), visible=False)
85     plt.setp(axes[i,j].get_yticklabels(), visible=False)
86
87 #for i in range(5):
88 #    fig.tight_layout()
89
90 #fig.subplots_adjust(right=0.8)
91 #cax = fig.add_axes([.85,.15,.05,.7])
92 #c = fig.colorbar(im, ax=cax)
93
94 fig.subplots_adjust(top=0.954, bottom=0.025, left=0.010,
95                     right=0.989, hspace=0.0, wspace=0.0)
96
97 plt.savefig(f'demo_output/signsweep_stitch_{sample_name}_{label}.png',
98             dpi=300)
99
100 cfile = 'demo_output/signsweep_colorbar.png'
101 if os.path.isfile(cfile):
102     continue # no need to make another
103
104 fig = plt.figure(figsize=(figsize[0],2))
105 ax1 = fig.add_axes([0.15, 0.25, 0.75, 0.5])
106 cbar = mpl.colorbar.ColorbarBase(ax1, cmap=CMAP,
107                                 norm=matplotlib.colors.Normalize(cmin,cmax),
108                                 orientation='horizontal')
109 plt.savefig(cfile, dpi=300)
110 #top = np.concatenate(L[:4],axis=1)
111 #bottom = np.concatenate(L[4:],axis=1)
112 #stitched = np.concatenate((top,bottom),axis=0)
113 #imga = plt.imshow(stitched, cmap=CMAP)
114 #plt.imsave(f'demo_output/signsweep_stitch_{sample_name}_{label}.png',
115 #            stitched, cmap=CMAP, vmin=cmin, vmax=cmax)
116
117 # also save the original pic
118 #plt.imsave(f'demo_output/signsweep_{sample_name}_{label}_raw',
119 #            imgview, cmap=plt.cm.gray)

```

APPENDIX B
3D VISUALIZATION OF THE FRANGI FILTER

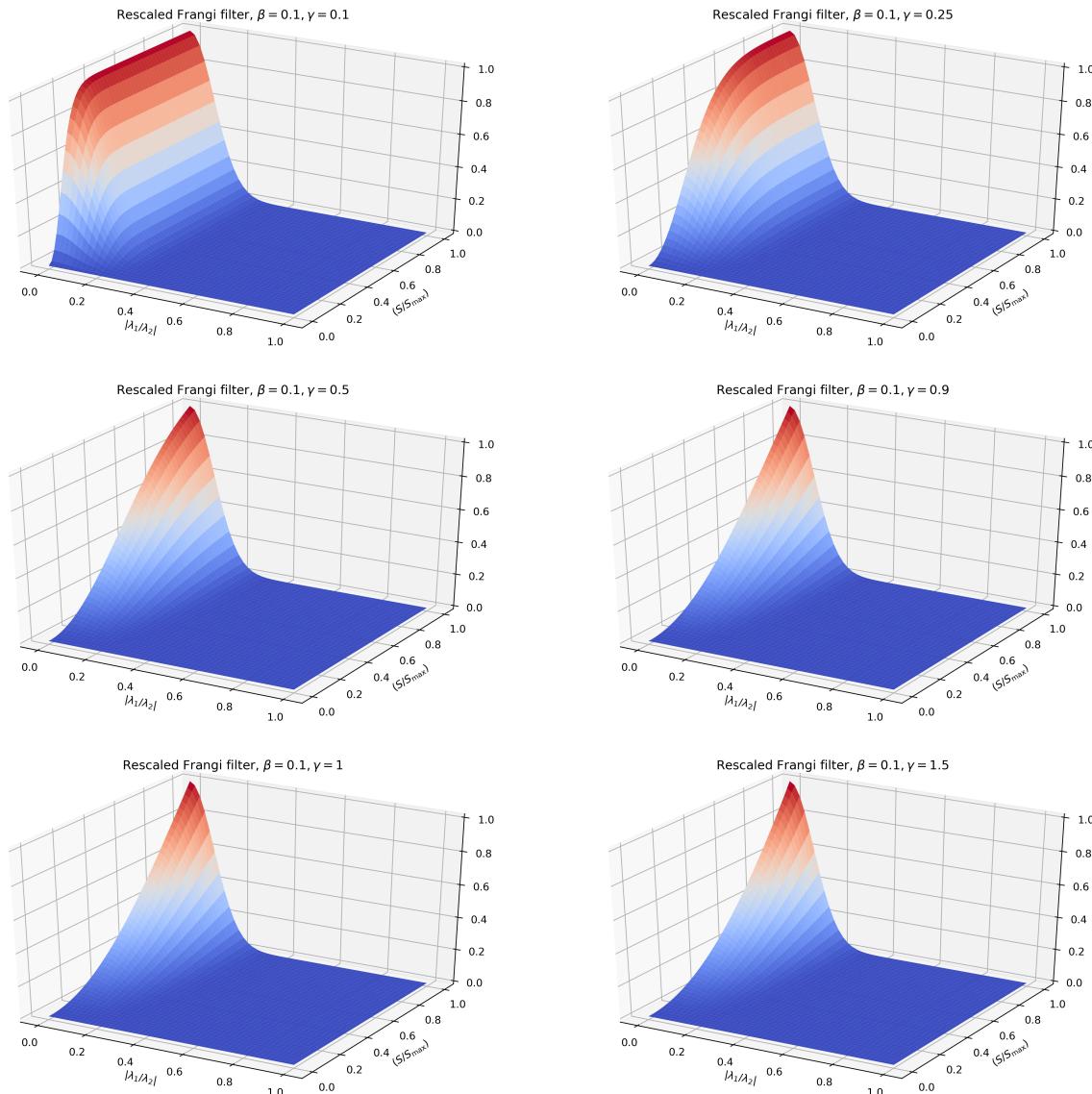


FIGURE 29: 3D graph of the Frangi Vesselness Measure, variable γ , $\beta = 0.1$

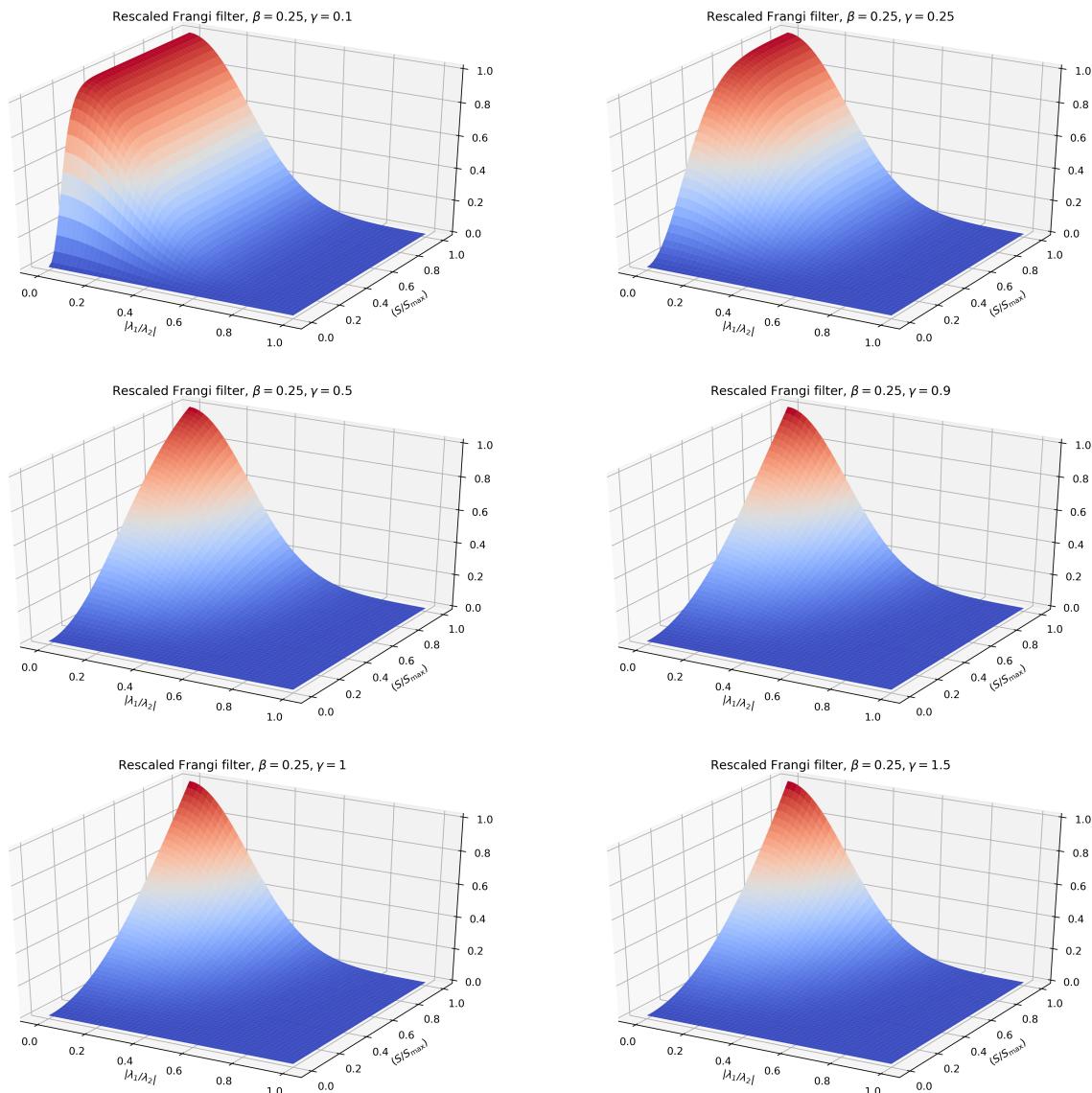


FIGURE 30: 3D graph of the Frangi Vesselness Measure, variable γ , $\beta = 0.25$

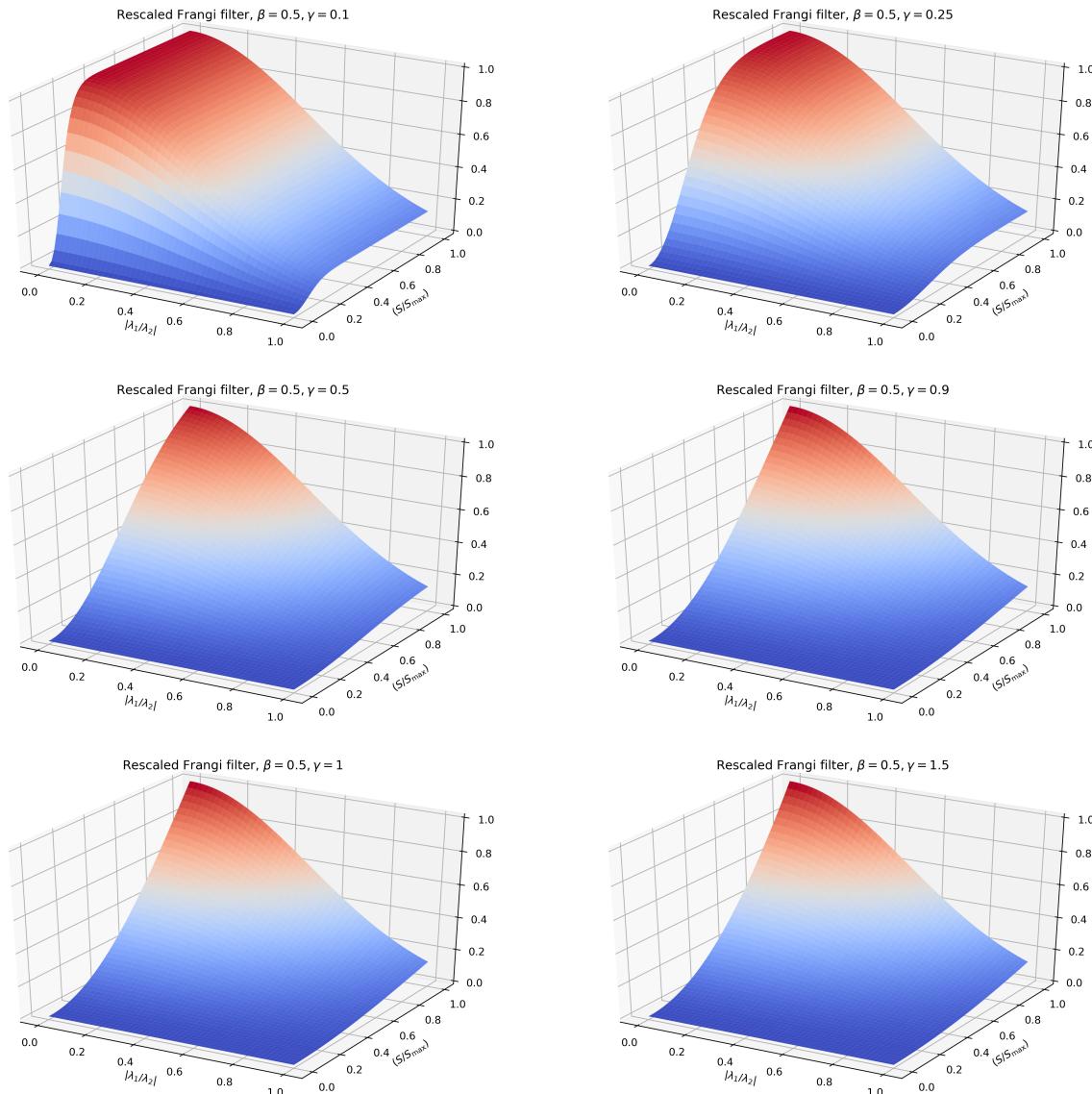


FIGURE 31: 3D graph of the Frangi Vesselness Measure, variable γ , $\beta = 0.5$

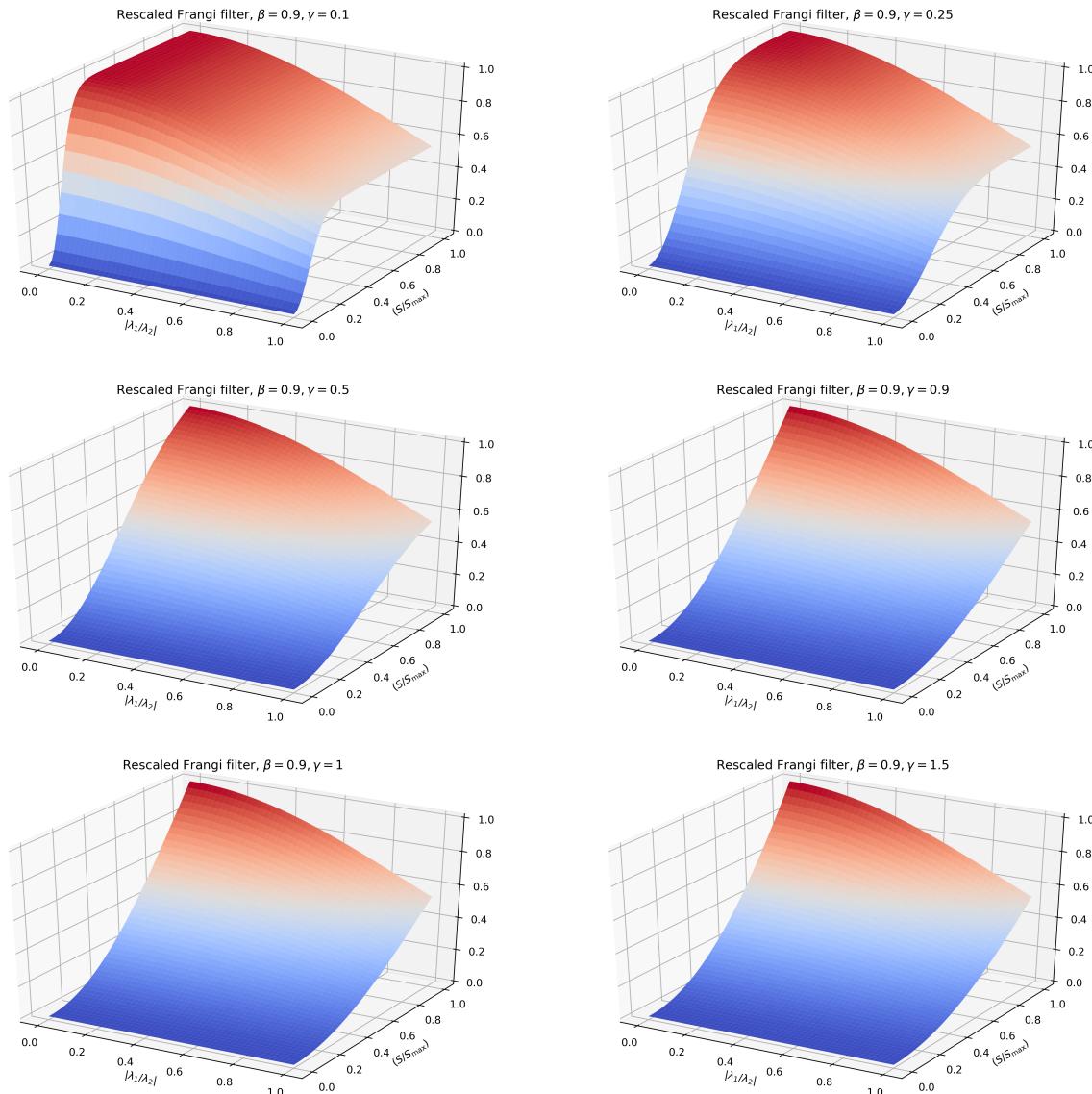


FIGURE 32: 3D graph of the Frangi Vesselness Measure, variable γ , $\beta = 0.9$

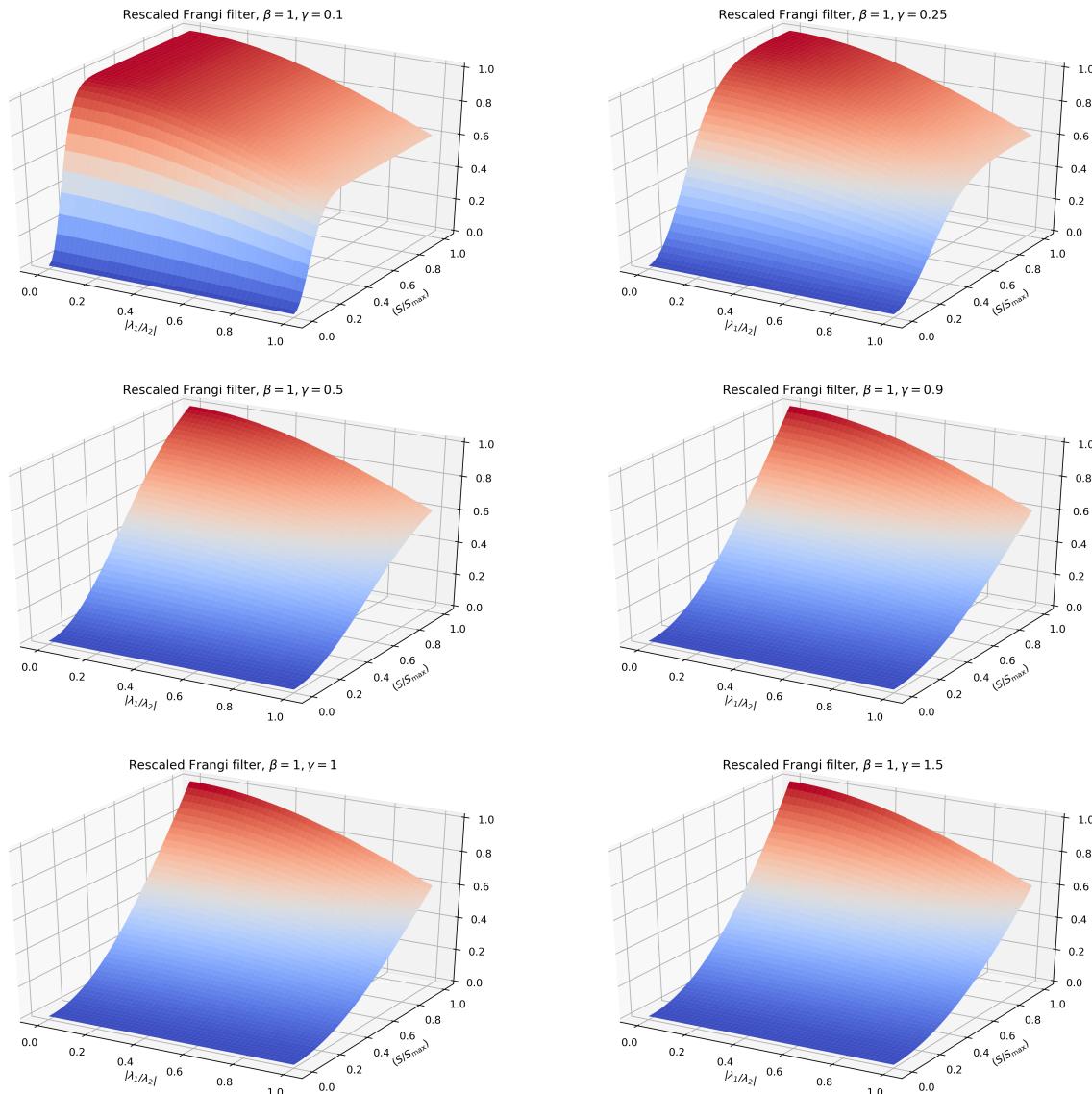


FIGURE 33: 3D graph of the Frangi Vesselness Measure, variable γ , $\beta = 1$

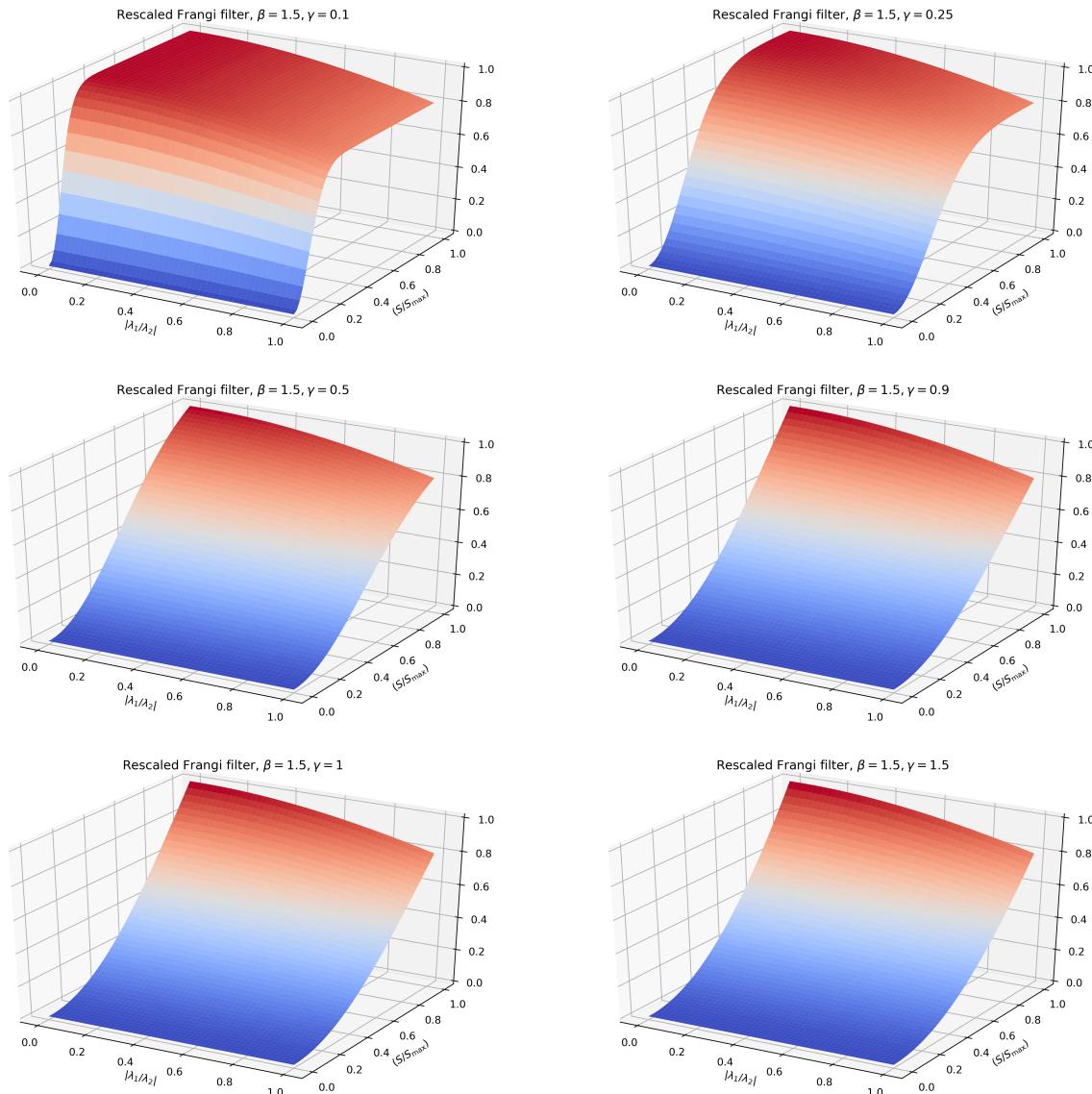


FIGURE 34: 3D graph of the Frangi Vesselness Measure, variable γ , $\beta = 1.5$

BIBLIOGRAPHY

BIBLIOGRAPHY

- [1] J.-M. Chang, H. Zeng, R. Han, Y.-M. Chang, R. Shah, C. M. Salafia, C. Newschaffer, R. K. Miller, P. Katzman, J. Moye, *et al.*, BMC medical informatics and decision making **17**, 162 (2017).
- [2] N. Huynh, Ph.D. thesis, California State University, Long Beach, 2013.
- [3] Y.-M. Chang, R. Han, H. Zeng, R. Shah, C. Newschaffer, R. Miller, P. Katzman, J. Moye, C. Salafia, *et al.*, Placenta **45**, 75 (2016).
- [4] N. Almoussa, B. Dutra, B. Lampe, P. Getreuer, T. Wittman, C. Salafia, and L. Vese, in *Medical Imaging 2011: Image Processing*, International Society for Optics and Photonics (PUBLISHER, ADDRESS, 2011), Vol. 7962, p. 79621L.
- [5] K. Y. Djima, C. Salafia, R. K. Miller, R. Wood, P. Katzman, C. Stodgell, and J.-M. Chang, Placenta **57**, 292 (2017).
- [6] R. C. Gonzalez and R. E. Woods, Upper Saddle River, NJ (2002).
- [7] W. Kühnel, B. Hunt, and A. M. Society, *Differential Geometry: Curves - Surfaces - Manifolds, Student mathematical library* (American Mathematical Society, ADDRESS, 2006).
- [8] *The Algebraic Eigenvalue Problem*, edited by J. H. Wilkinson (Oxford University Press, Inc., New York, NY, USA, 1988).
- [9] X. Jiao and H. Zha, in *Proceedings of the 2008 ACM symposium on Solid and physical modeling*, ACM (PUBLISHER, ADDRESS, 2008), pp. 159–170.
- [10] R. Burden and J. Faires, *Numerical Analysis*, 9 ed. (Brooks/Cole, ADDRESS, 2011).
- [11] A. F. Frangi, W. J. Niessen, K. L. Vincken, and M. A. Viergever, in *International Conference on Medical Image Computing and Computer-Assisted Intervention*, Springer (PUBLISHER, ADDRESS, 1998), pp. 130–137.
- [12] Y. Sato, S. Nakajima, N. Shiraga, H. Atsumi, S. Yoshida, T. Koller, G. Gerig, and R. Kikinis, Medical image analysis **2**, 143 (1998).
- [13] C. Lorenz, I. C. Carlsen, T. M. Buzug, C. Fassnacht, and J. Weese, in *CVRMed-MRCAS'97*, edited by J. Troccaz, E. Grimson, and R. Mösges (Springer Berlin Heidelberg, Berlin, Heidelberg, 1997), pp. 233–242.

- [14] S. D. Olabarriaga, M. Breeuwer, and W. Niessen, in *International Congress Series*, Elsevier (PUBLISHER, ADDRESS, 2003), Vol. 1256, pp. 1191–1196.
- [15] J. J. Koenderink, *Biological Cybernetics* **50**, 363 (1984).
- [16] J. Sporring, in *Gaussian Scale-Space Theory*, edited by L. Florack, M. Nielsen, and P. Johansen (Kluwer Academic Publishers, Norwell, MA, USA, 1997).
- [17] E. Hille and R. Phillips, *Functional Analysis and Semi-groups*, American Mathematical Society: Colloquium publications (American Mathematical Society, ADDRESS, 1957).
- [18] J. Babaud, M. Baudin, R. O. Duda, and A. P. Witkin, *IEEE Transactions on Pattern Analysis & Machine Intelligence* **8**, 26 (1986).
- [19] T. Lindeberg, *IEEE transactions on pattern analysis and machine intelligence* **12**, 234 (1990).
- [20] T. Lindeberg, *On the construction of a scale-space for discrete images* (KTH Royal Institute of Technology, ADDRESS, 1988).
- [21] M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, ninth dover printing, tenth gpo printing ed. (Dover, New York, 1964).
- [22] T. Lindeberg, *Journal of Mathematical Imaging and Vision* **3**, 349 (1993).
- [23] T. Lindeberg, *International journal of computer vision* **30**, 79 (1998).
- [24] B. Fornberg, *Mathematics of computation* **51**, 699 (1988).
- [25] A. Morar, F. Moldoveanu, and E. Gröller, in *2012 IEEE 8th International Conference on Intelligent Computer Communication and Processing*, IEEE (PUBLISHER, ADDRESS, 2012), pp. 213–220.
- [26] E. Jones, T. Oliphant, P. Peterson, *et al.*, SciPy: Open source scientific tools for Python, 2001–, [Online; accessed *today*].
- [27] S. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, T. Yu, and the scikit-image contributors, *PeerJ* **2**, e453 (2014).
- [28] S. Damelin and N. Hoang, *International Journal of Mathematics and Mathematical Sciences* **2018**, (2018).
- [29] H. Lange, in *Medical Imaging 2005: Image Processing*, International Society for Optics and Photonics (PUBLISHER, ADDRESS, 2005), Vol. 5747, pp. 2183–2193.
- [30] L. Grady, *IEEE Trans. Pattern Anal. Mach. Intell.* **28**, 1768 (2006).

- [31] B. Matthews, *Biochimica et Biophysica Acta (BBA) - Protein Structure* **405**, 442 (1975).
- [32] I. Laptev, H. Mayer, T. Lindeberg, W. Eckstein, C. Steger, and A. Baumgartner, *Machine Vision and Applications* **12**, 23 (2000).