

ABSTRACT

OPTIMIZED STRICT MULTISCALE FRANGI PREFILTERING FOR SEGMENTATION: TOWARDS AN AUTOMATED PLACENTAL CHORIONIC SURFACE VASCULAR NETWORK EXTRACTION

By

Lucas Allen Wukmer

May 2019

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 development on later conditions, but even perhaps provide a real-time diagnostic for neonatal risk factors.

Much work has been to develop reliable vascular network segmentation 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 techniques to study placental images, but the approach has been historically hindered by the comparative irregularity of the placental surface as a whole, which introduces significant noise into filtered result. Previous work has either involved additional filtering [2] or other techniques that are often time-consuming and resource intensive [3].

Here we provide an in-depth mathematical background of the multiscale Frangi filter. Informed by this theory, we are able to identify stricter parameters that allow us to greatly improve our result. We also reimplement the Frangi filter in frequency space (using a fast fourier transform), which allows us to quickly probe many scales.

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 compare several approaches of merging the multiscale result into an approximation of the PCSVN and compare them to manual tracings of the network. Finally, we develop the notion of the *signed* Frangi filter, upon which we describe a novel-yet-straightforward segmentation method called "trough filling".

**OPTIMIZED STRICT MULTISCALE FRANGI PREFILTERING FOR
SEGMENTATION: TOWARDS AN AUTOMATED 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
May 2019

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

**OPTIMIZED STRICT MULTISCALE FRANGI PREFILTERING FOR
SEGMENTATION: TOWARDS AN AUTOMATED 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

May 2019

ACKNOWLEDGEMENTS

Thank you to the committee for their patience. Thank you to my family and friends for their support. Thank you to you, the reader, for reading this. This work is dedicated to my mother, who always supported me and encouraged me to finish my thesis and get it over with already.

TABLE OF CONTENTS

	Page
ACKNOWLEDGEMENTS	iii
LIST OF TABLES	vii
LIST OF FIGURES	viii
CHAPTER	
1. INTRODUCTION	1
2. MATHEMATICAL METHODS: DIFFERENTIAL GEOMETRY AND SURFACE CURVATURE	4
Problem Setup in Image Processing	4
Differential Geometry	5
Preliminaries of Differential Geometry	5
Curvature of a surface and its calculation	8
Principal Curvatures and Principal Directions	15
The Weingarten map and Principal Curvatures of a Cylindrical Ridge	21
3. THE UNISCALE FRANGI FILTER	26
The Frangi Filter: Uniscale	26
Anisotropy Measure	27
Structurenness measure	28
The Frangi vesselness measure	29
Choosing parameters β and c	30
4. LINEAR SCALE SPACE THEORY	35
Properties/Axioms of Linear Scale Space Theory	36
Sufficiency of the Gaussian Kernel	37
Scale Spaces over Discrete Structures	38
5. THE MULTISCALE FRANGI FILTER AND SIGNED FRANGI FILTER	41
Rudimentary Thresholding	41

APPENDIX	Page
The Signed Frangi Filter	44
6. FFT-BASED DISCRETE DERIVATIVES	45
Convolution Speedup via FFT	45
Fourier Transform of a continuous 1D signal.....	45
Fourier Transform of a Discrete 1D signal	46
2D DFT Convolution Theorem	47
FFT	49
Calculating the Hessian via FFT: A demonstration.....	53
7. MORPHOLOGICAL IMAGE PROCESSING	59
8. RESEARCH PROTOCOL	60
Samples / Image Domain	60
A representative sample	60
Knowns and Unknowns.....	63
Data Cleaning and Preprocessing	64
Boundary Dilation.....	66
Deglaring	68
Multiscale Setup	72
The Research Protocol	73
9. RESULTS AND ANALYSIS	78
Sample visual output	78
Variations in the Data Set and Imperfections of the Ground Truth	79
Results.....	83
10. SEGMENTATION	93
Binary Classifications and the confusion matrix	93
Postprocessing Techniques	95
Method A: Fixed Threshold	95
Method B: Percentile Based Merging	97
Method C: Scale-Based Random Walker	97
Method D: Scale Based Sieving	99
Method E: Trough Filling	102
Nonfrangi Segmentation Methods	105
Segmentation Results	106
11. PRIMITIVE NETWORK COMPLETION	109

	Page
APPENDIX	
12. CONCLUSION	115
Future research directions	115
APPENDICES	116
A. CODE LISTINGS	117
B. 3D VISUALIZATION OF THE FRANGI FILTER	245
BIBLIOGRAPHY	252

LIST OF TABLES

	Page
TABLE	
1 MSE of Gaussian blurs ($\sigma = 0.3$)	55
2 MSE of Frangi scores $\sigma = 0.3$	55
3 MSE of Gaussian blurs of an image ($\sigma = 5$)	57
4 MSE of Frangi scores $\sigma = 5$	57
5 Vessel width color code for manual tracing protocol.....	62

LIST OF FIGURES

FIGURE	Page
1 Tangent plane of a graph	9
2 The graph of a cylindrical ridge of radius r	22
3 The principal eigenvectors at a ridge like structure	27
4 Dependence of the Anisotropy Factor on its Parameter	32
5 Dependence of the Structureness Factor on its Parameter.....	33
6 Nonzero-percentile thresholding of \mathcal{V}_{\max} (95th and 98th percentile)	43
7 Compatibility of Gaussian convolution strategies	54
8 Iterative Gaussian blur	56
9 Image cross-section of Gaussian blurred images	56
10 Image cross-section of Frangi targets images.....	56
11 Image cross-section of Gaussian blurred images $\sigma = 5$	57
12 Image cross-section of Frangi targets images $\sigma = 5$	57
13 Time required.....	58
14 A representative placental sample and tracing.....	61
15 Preprocessed files from an NCS sample.....	65
16 Effect of boundary dilation on Frangi responses	67
17 Deglaring a sample using a hybrid inpainting method	69
18 Comparison of glare inpainting methods (detail)	70
19 Example scalewise Frangi output (plate and inset) (Example 1)	76
20 Example scalewise Frangi output (plate and inset) (Example 2)	77
21 Vesselness score, percentile thresholds, and simple thresholds, Example 1	80
22 Vesselness score, percentile thresholds, and simple thresholds, Example 2	81
23 Issues with the ground truth manifesting in Frangi vesselness scores	82
24 Bad samples	84
25 \mathcal{V}_{\max} and CVR for varying multiscale Frangi parametrizations (Example 1)	86
26 \mathcal{V}_{\max} and CVR for varying multiscale Frangi parametrizations (Example 2)	87
27 CVR scores of 25 samples under varying parametrizations.....	88
28 Scale of maximum Frangi score for true positives and false negatives	89
29 Scale of maximum Frangi score for true positives only (95th-percentile filtering)	91

FIGURE		Page
30	Pixel Width of Ground Truth vs. Scale Length for True Positives ...	92
31	Sample confusion matrix	94
32	Nonzero-percentile thresholding of \mathcal{V}_{\max}	98
33	Scale-wise random walker segmentation (select scales)	100
34	Random walker segmentation (result and sample).....	101
35	Signed Frangi output (plate and inset) (Example 1)	103
36	Signed Frangi output (plate and inset) (Example 1)	104
37	Confusion matrices of various segmentation methods (Example 1)	106
38	Confusion matrices of various segmentation methods (Example 2)	107
39	Confusion matrices of various segmentation methods (Example 3)	108
40	Endpoints labels based on adjacent neighbor location.....	110
41	All lines between endpoints with nonzero \mathcal{V}_{\max}	111
42	Partially completed network.....	112
43	Trough Dilation and Network Completion (Example 1).....	113
44	Trough Dilation and Network Completion (Example 2).....	114
45	3D graph of the Frangi Vesselness Measure, variable $\gamma, \beta = 0.1$	246
46	3D graph of the Frangi Vesselness Measure, variable $\gamma, \beta = 0.25$	247
47	3D graph of the Frangi Vesselness Measure, variable $\gamma, \beta = 0.5$	248
48	3D graph of the Frangi Vesselness Measure, variable $\gamma, \beta = 0.9$	249
49	3D graph of the Frangi Vesselness Measure, variable $\gamma, \beta = 1$	250
50	3D graph of the Frangi Vesselness Measure, variable $\gamma, \beta = 1.5$	251

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 [4] of correlation between this risk and placental health factors.. Most ASD cases are not diagnosed until the child reaches three or four, so the benefit of an early detection technique would be massive, as the brain may be more receptive to treatment at a young age. In particular, it was shown in [4] 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 user intervention to generate each trace. The present work follows several other efforts towards an automated extraction technique, such as [5], [2], and most recently [3]. Automating this procedure would not only allow more data acquisition to study the potential connection between ASD and placental health, but could even potentially serve as the basis of 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 common one in image processing. There have been many techniques adapted to extracting vascular networks. However, the placenta in particular poses as a particularly difficult image domain to work with. It is a surface network with much irregularity: there are frequent gaps and crossings, and the "background" has a great degree of variation in color

intensity and structure itself, causing many naïve approaches at segmentation to fail completely.

Reliable vascular extraction methods do exist for well-known image domains (such as retinal MRA images) using Hessian-based filters, namely the (multiscale) Frangi filter. It is desirable to extend this 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. Previous attempts to implement the Frangi filter in this context [2] dealt with this noise by passing the Frangi filtered result to an additional local curvilinear filter (a directional filter bank) in an effort to retrieve a partial segmentation. Other more recent efforts [3] are very promising but are considerably more resource-intensive.

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. A more throughout treatment of these underlying theories allows us to choose parameters much more meaningfully than previous efforts. In particular, a more thoughtful selection of the Frangi filter's two parameters reduces much of the noise previously encountered within this image domain. 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 the usefulness of this filter by then demonstrating several approaches to segmentation of the PCSVN. These will range from naïve (i.e direct thresholding) to

more intricate and specialized (including a novel yet straightforward method called "trough-filling"). We compare each of these results to the ground truth manual tracings of the network, quantifying the success of various segmentation methods using Matthew's correlation constant (MCC) and precision scoring.

CHAPTER 2

MATHEMATICAL METHODS: DIFFERENTIAL GEOMETRY AND SURFACE CURVATURE

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 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 8.1), 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 chapter 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.

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 simplified

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

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 (that is, for surfaces f that are not graphs). We refer to [7] for the general proof. Our present situation is much less delicate, as we're only concerned for cases when f is a graph. In this case, 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

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

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

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

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

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 , definition 2.11 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

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

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

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

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

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

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

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

principal directions and curvatures in a simple example.

Theorem 2.6 (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 calcuate 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 for analyzing critical points alone. 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

much response anyway when the gradient is large.

From [9], we can bound the perturbation of eigenvalues from \widehat{L} to $\text{Hess}(h)$ by

Theorem 2.8. *Let A be a $n \times n$ normal matrix with eigenvalues $\lambda_1, \dots, \lambda_n$ and E an $n \times n$ arbitrary matrix. If $\hat{\lambda}$ is an eigenvalue of $A + E$, then there is an eigenvalue λ_i of A for which $|\hat{\lambda} - \lambda_i| \leq \|E\|_2$*

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 [10] 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

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

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

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

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



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

The gauss map is given by

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

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

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

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

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

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

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

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 the Hessian matrix of h 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, [11]:

Lemma 2.9 (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 $v_1, v_2 \neq 0$.

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

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

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

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

CHAPTER 3

THE UNISCALE FRANGI FILTER

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 [12] is a widely used 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 [13] and [14], as well as a comparison given in [15]. 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:

$$\mathcal{V}_o(x_0, y_0) = \begin{cases} 0 & \text{if } \lambda_2 > 0 \\ \exp\left\{-\frac{A^2}{2\beta^2}\right\} \left(1 - \exp\left\{-\frac{S^2}{2c^2}\right\}\right) & \text{otherwise} \end{cases} \quad (3.1)$$

where

$$A := |\lambda_1/\lambda_2| \quad \text{and} \quad S := \sqrt{\lambda_1^2 + \lambda_2^2} \quad (3.2)$$

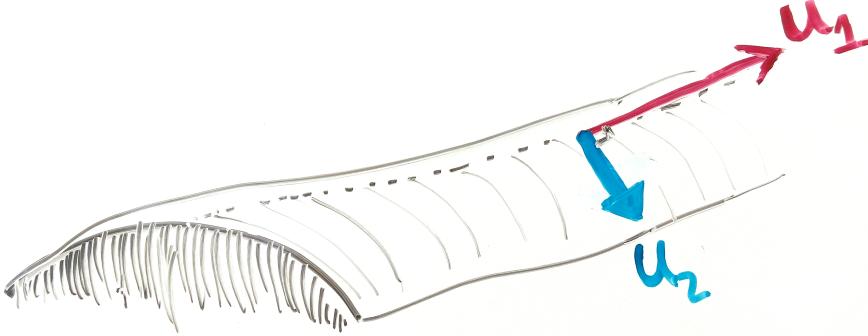


FIGURE 3: The principal eigenvectors at a ridge like structure

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. (3.1), and in particular, the ratios defined in eq. (3.2). A and S are known as the anisotropy measure and structureness measure, respectively. Consequently, we'll refer to the two factors in eq. (3.1) 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 (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.

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.

Similarly, the vessel 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 8.1), 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 than in the transverse direction.

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 this issue specifically by casting this as a multiscale problem in ??.

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 12.1.

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 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. (3.1) 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. (3.1), 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 $\lambda_2 < 0$.

Choosing parameters β and c

The parameters β and c are meant to scale so that the peaks of the anisotropy factor $\exp\left\{\frac{-A^2}{2\beta^2}\right\}$ and the structureness factor $(1 - \exp\left\{\frac{-S^2}{2c^2}\right\})$ 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,

$$\begin{aligned} \max\{\mathcal{V}_\sigma\} &\leq \max\left(\exp\left\{\frac{-A^2}{2\beta^2}\right\}\right) \max\left(\left(1 - \exp\left\{\frac{-S^2}{2(\gamma S_{\max})^2}\right\}\right)\right) \\ &\leq \max\left\{\left(1 - \exp\left\{\frac{-S^2}{2(\gamma S_{\max})^2}\right\}\right)\right\} \\ &= \left(1 - \exp\left\{\frac{-(S_{\max})^2}{2(\gamma S_{\max})^2}\right\}\right) = \left(1 - \exp\left\{\frac{-1}{2\gamma^2}\right\}\right) \end{aligned} \tag{3.3}$$

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 at any scale is $\max\{\mathcal{V}_\sigma\} \leq 1 - \exp\{-1\} \approx .8647$. 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 a explicit function of γ that rescales \mathcal{V}_σ so that the structureness factor has a maximum output score of 1 regardless of choice of γ . Our final Frangi vesselness measure is thus

$$\mathcal{V}_\sigma(x_0, y_0) = \begin{cases} 0 & \text{if } \lambda_2 > 0 \\ a_\gamma \exp\left\{\frac{-A^2}{2\beta^2}\right\} \left(1 - \exp\left\{\frac{-S^2}{2(\gamma S_{\max})^2}\right\}\right) & \text{otherwise} \end{cases} \quad (3.4)$$

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

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

For β , Frangi suggested 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. We shall show empirically that contexts in which more ‘bloblike’ structures are known to be present than that for which the Frangi filter was originally designed, we will benefit from a smaller choice of β .

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

We can theoretically choose any values $0 < \beta, \gamma < \infty$ for the two parameters. Two particular limits are of theoretical interest: as $\beta \rightarrow \infty$, the anisotropy factor tends to 1, and as $\gamma \rightarrow 0$, the structureness factor tends to 1, making the that measure entirely irrelevant. (Of course, in the limit that $\beta \rightarrow 0$ or $\gamma \rightarrow \infty$, their respective factors become 0, making the entire filter irrelevant). We will

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



FIGURE 4: Dependence of the Anisotropy Factor on its Parameter

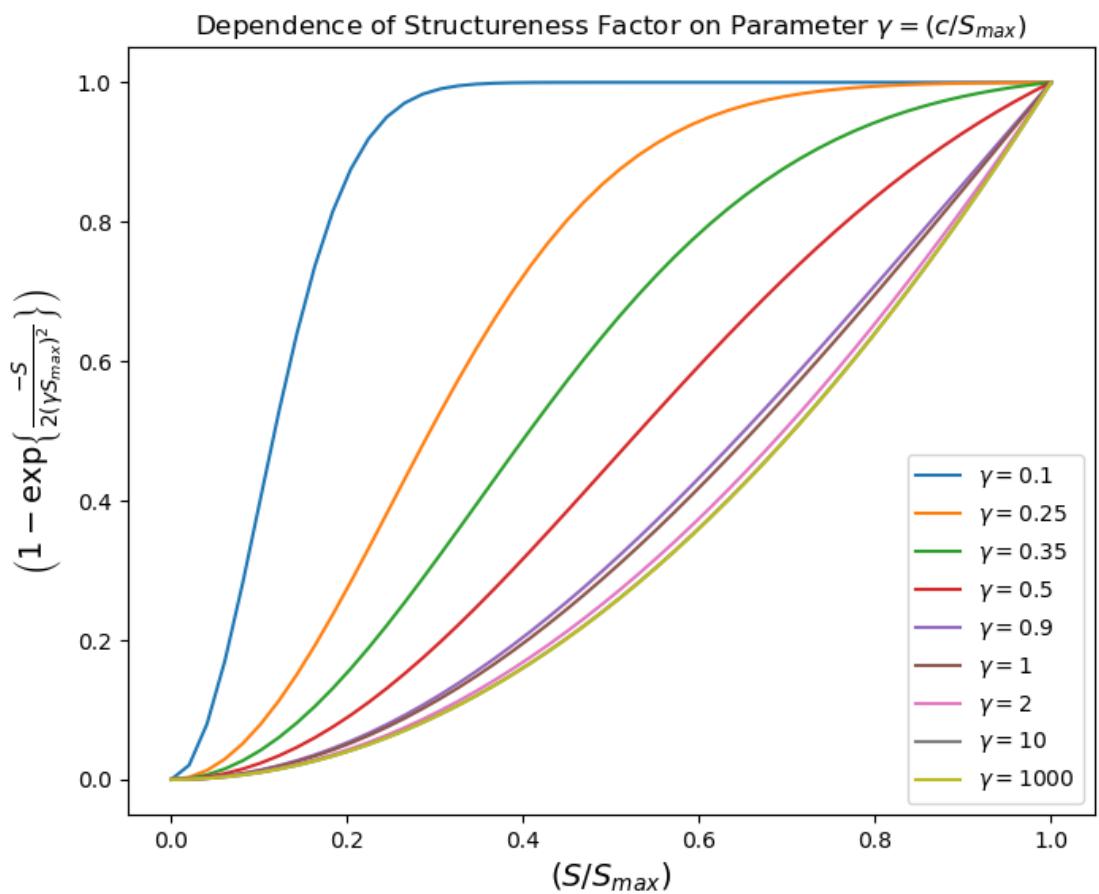


FIGURE 5: Dependence of the Structureness Factor on its Parameter

The ultimate choice of these parameters β and γ have the overall effect of tuning the selectivity of the Frangi filter itself. In the figures in appendix B, we plot the Frangi vesselness measure itself as a function of S and A while sweeping through different choices of β and γ .

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

CHAPTER 4

LINEAR SCALE SPACE THEORY

Although the ideas presented above require differentiation of continuous surfaces, our image is in fact composed of discrete pixels. 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. Divided differences can serve as an analogue of differentiation in discrete contexts, but our "derivative" and any use of it is then completely dependent on 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. Ideally, we would like statements we make about higher order phenomena to be stable enough at least that we would arrive at the same conclusions from analyzing a higher resolution image of the same surface. Koenderink [16] argued that "any image can be embedded in a one-parameter family of derived images (with resolution as the parameter) in essentially only one unique way" given a few of the so-called scale space axioms. He (and others) showed that a small set of intuitive axioms imply that any such family of images $\{K_\sigma\}$ must satisfy the heat equation

$$\begin{cases} \Delta K(x, y; \sigma) = \frac{\partial K}{\partial \sigma}(x, y; \sigma) \text{ for } \sigma \geq 0 \\ K(x, y, 0) = u_0(x, y) \end{cases} \quad (4.1)$$

where $u_0 : \mathbb{R}^2 \rightarrow \mathbb{R}$ is the original image (viewed as a continuous surface), σ is the resolution parameter, and $K : \mathbb{R}^2 \rightarrow \mathbb{R}$ for each fixed σ .

This result is intuitive and desirable—we would anticipate a lower-resolution version of our original image to represent a diffusion or “blurring” of the initial scales information. Much work has been done to formalize this approach [17]. This has resulted in various formulations of a minimal set of axioms from which all other desirable properties of the scale space can be derived, culminating eventually in the necessity and sufficiency of eq. (4.1) itself. We will refrain from such an exhaustive development in the present text, but rather list some desirable properties and outline the approach.

Properties/Axioms of Linear Scale Space Theory

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

$$\{ K(x, y; \sigma) = T_\sigma u_0 \mid \sigma \geq 0, K(x, y; 0) = u_0 \} \quad (4.2)$$

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

Axiom 4.1 (Linear-shift and Rotational Invariance). *We require 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, and cropping or rotating our initial image should not affect resolution of the content.*

Axiom 4.2 (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.*

$$T_\sigma u = T_{\sigma_1 + \sigma_2} u \quad (4.3)$$

Axiom 4.3 (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 any nonzero real number. Moreover, we require that the operator behaves continuously with respect to the scale parameter.

The following requirement has great implication, and is also very successful in encoding our intuitive sense of resolution.

Axiom 4.4 (Causality Condition). *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.*

To make this more precise, if $K(x_0, y_0; \sigma_0)$ (that is, a point (x_0, y_0) at some particular resolution σ_0) is a local maximum at that resolution, then an increase in scale cannot make this maximum more prominent, i.e.

$$\begin{cases} \nabla K(x_0, y_0; \sigma_0) = 0 \\ \Delta K(x_0, y_0; \sigma_0) < 0 \end{cases} \implies K(x_0, y_0; \sigma_1) \leq K(x_0, y_0; \sigma_0) \forall \sigma_1 \geq \sigma_0 \quad (4.4)$$

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.

$$\begin{cases} \nabla K(x_0, y_0; \sigma_0) = 0 \\ \Delta K(x_0, y_0; \sigma_0) > 0 \end{cases} \implies K(x_0, y_0; \sigma_1) \geq K(x_0, y_0; \sigma_0) \forall \sigma_1 \geq \sigma_0 \quad (4.5)$$

This encodes our intuition that no image feature should be sharpened by a decrease in resolution. The only result is a (non-strictly) monotonic blurring of the image as scale parameter σ tends to infinity.

Sufficiency of the Gaussian Kernel

Although we will omit it here, 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. (4.1) must hold. This has been shown in various ways, both by

Koenderink [16], Babaud [18], as well as Lindeberg in [17]. Of course, once eq. (4.1) has been established, it is straightforward to show that

$$K(x, y; \sigma) = T_\sigma u_0 = G_\sigma \star u_0 \quad \text{where} \quad G_\sigma := \frac{1}{2\pi\sigma^2} e^{(-|x|^2/(2\sigma^2))} \quad (4.6)$$

is a solution. That is, the family can be generated by convolution with a Gaussian kernel. Lindeberg and others furthered this by arguing that fT_σ 's continuity as σ approaches zero ultimately implies that convolution by a Gaussian is the *unique* operator that generates this scale space.

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 heat equation. However, viewing our actual image definition 2.1 as a sample of a continuous surface definition 2.2, we might 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 4.1 (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

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

In [19] and in particular [20], Lindeberg demonstrated that the sampled Gaussian kernel violates not only semigroup property (axiom 4.2), but—much less forgivably—the causality property (axiom 4.2). 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 4.4 and axiom 4.2:

Definition 4.2 (Discrete Gaussian Kernel). *The discrete Gaussian kernel, which can be shown to be a suitable generator for scale space, is given by*

$$T(n; \sigma) = e^{-\alpha\sigma} I_n(\alpha\sigma), \quad I_n(\sigma) = I_{-n}(\sigma) = (-1)^n J_n(i\sigma) \quad n \geq 0, \sigma, \alpha > 0 \quad (4.7)$$

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

$$K(x, \sigma) = \sum_{n=-\infty}^{\infty} T(n; \sigma) f(x-n) \quad \text{for } x \in \mathbb{Z}, \sigma > 0 \quad (4.8)$$

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 local central difference. Lastly, the 2D version of the family given in eq. (4.8) can be obtained by independent convolution of its dimensions (i.e. it is separable).

CHAPTER 5

THE MULTISCALE FRANGI FILTER AND SIGNED FRANGI FILTER

With the ideas of scale established, we may return to our discussion of the Frangi filter. 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 the dependence of the Frangi filter's response on choice of scale (as demonstrated in section 3.1), we wish to probe at multiple scales regions that would receive a high vesselness score at any range and somehow merge the result. Frangi [12] approached this problem by simply taking the maximum vesselness measure over all scales. Thus the multiscale Frangi vesselness score at the pixel (x_0, y_0) would be

$$\mathcal{V}_{\max}(x_0, y_0) = \max_{\sigma \in \Sigma} \{\mathcal{V}_\sigma(x_0, y_0)\} \quad (5.1)$$

where $\Sigma := \{\sigma_0, \sigma_1, \dots, \sigma_N\}$ is the set of scales at which to probe, and \mathcal{V}_σ is the Frangi vesselness measure at scale σ for the pixel (x_0, y_0) . The set of scales Σ should be chosen to be representative enough of all scales where meaningful content is expected to be found.

Rudimentary Thresholding

After the maximization in eq. (5.1), we are left with a matrix with as many pixels as the original image, all with a vesselness measure between 0 and 1 for each pixel in the image.

At this point, Frangi [12] refrained from explicitly interpreting the score assigned by eq. (5.1); that is—whether a particular pixel (x_0, y_0) in the image definitely represents a vessel or not based on its Frangi score. Instead, he cautioned that the result should not

be used as a segmentation method alone, and moreover that the width of the vasculature cannot be determined rigorously from the Frangi filter (as discussed in chapter 3).

Nonetheless, we wish to demonstrate the usefulness of the Frangi filter within our image domain towards segmentation. We should at least expect that a well-tuned Frangi filter (on an appropriately registered and denoised sample) should assign its highest scores to vessel pixels. We can select these strong Frangi responses and use them as seeds for some subsequent algorithm. A straightforward enough approach would be to simply threshold at some fixed value α . Such thresholding was used in [2].

$$\mathcal{V}_{\Sigma\alpha}(x_0, y_0) = \begin{cases} 1 & \text{if } \mathcal{V}_{\max}(x_0, y_0) \geq \alpha \\ 0 & \text{else} \end{cases}, \quad \alpha > 0 \text{ for } \alpha \text{ fixed.} \quad (5.2)$$

If we insist on such a performing such a thresholding, the “correct” choice of α unfortunately seems to depend on the image domain, so user intervention when dealing with the problem domain seems to be the best strategy. We would hope that some normalization of our data set would permit a single choice of α across all samples, but unfortunately we cannot guarantee this. Without prior knowledge of an appropriate choice of α , we may have to simply select α by trial and error.

A good alternative method of thresholding would be to simply select the highest scores from each responses: we calculate a high percentile score and threshold at that value. Due to the large number of zeros outputted by the filter, we opt instead to take the q th percentile of only nonzero values of V_{Σ} . We briefly demonstrate this in fig. 32 on a particularly well-behaved sample. The top left value image is the base image, the top right is \mathcal{V}_{\max} , the bottom left is \mathcal{V}_{\max} thresholded at the 95th percentile, and the bottom left is thresholded at the 98th percentile.

Of course, the downside of this method is that we do not know in general the size of the network, and we may miss entire branches of the vascular network if there is a

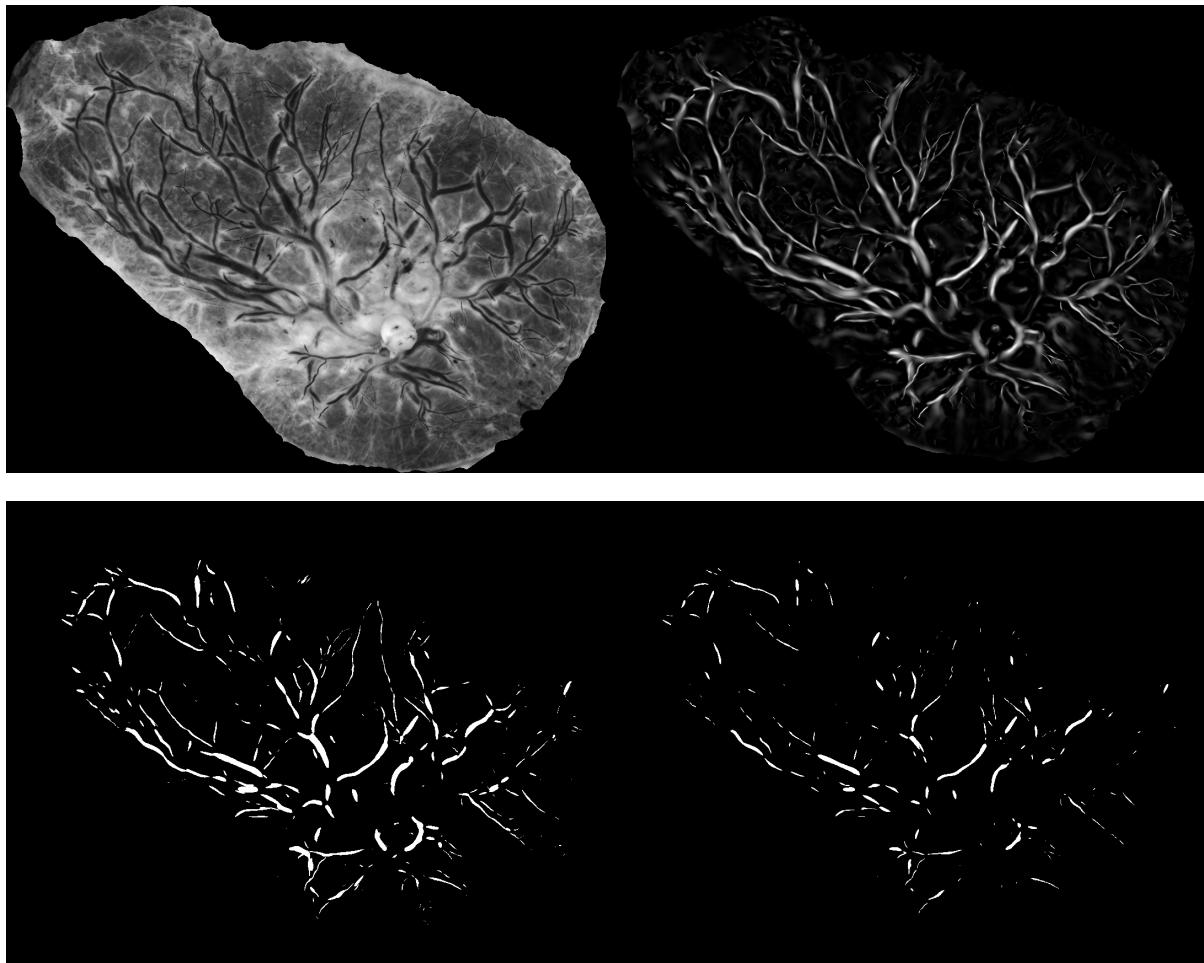


FIGURE 6: Nonzero-percentile thresholding of \mathcal{V}_{\max} (95th and 98th percentile)

large amount of curvilinear content elsewhere in the image.

We will discuss alternatives methods of aggregating results from our multiscale method, as well as optimal values for parameters and scales in chapter 10. As a final note, we admit that any future extensions of this work (as will be discussed in chapter 12) need not proceed from either of these thresholdings; analyzing the raw vesselness score \mathcal{V}_{\max} , or even the un-merged scale-wise scores may prove far more rewarding.

The Signed Frangi Filter

We finally introduce the novel (yet straightforward) notion of the signed Frangi filter. As will be shown in chapter 10, we can benefit from simultaneously calculate for a dark background and a light background. Since the Frangi filter normally throws away any response where $\lambda_2 < 0$ (if dark curvilinear features are targeted) or $\lambda_2 > 0$ (if light curvilinear features are targeted), we lose no computation time at all (although we must store more results). After computing the multiscale result, we can easily separate these into a positive and a negative strain, which we will denote $\mathcal{V}_{\max}^{(+)}$ and $\mathcal{V}_{\max}^{(-)}$. Our $\mathcal{V}_{\max}^{(+)}$ is the same as our \mathcal{V}_{\max} before, and $\mathcal{V}_{\max}^{(-)}$ is the same result as if we had taken the Frangi filter while only looking for the opposite type (light/dark) curvilinear feature. Plotting \mathcal{V}_{\max} over a scale of $[-1, 1]$ demonstrates an interesting effect. Whereas the Frangi filter generally is not reliable in terms of accurately predicting widths of through-like (or ridge-like) curvilinear features, by somehow combining the results of our positive and negative strains. We *can* get a sense of the width by looking at where there is a relatively strong response of opposite sign.

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.

CHAPTER 6

FFT-BASED DISCRETE DERIVATIVES

According to section 4.0.3, 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].

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 ??.

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:

$$f(t) = \sum_{-\infty}^{\infty} c_n e^{i \frac{2\pi n}{T} t}, \quad c_n = \frac{1}{T} \int_{-T/2}^{T/2} f(t) e^{-i \frac{2\pi n}{T} t} dt \quad (6.1)$$

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

$$F(\mu) := \mathcal{F}\{f(t)\} = \int_{-\infty}^{\infty} f(t)e^{j2\pi\mu t} dt \quad (6.2)$$

An inverse transform will then recover our original signal:

$$f(t) = \mathcal{F}^{-1}\{F(\mu)\} = \int_{-\infty}^{\infty} F(\mu)e^{j2\pi\mu t} dt \quad (6.3)$$

Together, eq. (6.2) and eq. (6.3) 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 :

$$s_{\Delta T}(t) := \sum_{n=-\infty}^{\infty} \delta[t - n\Delta T], \quad \delta[t] = \begin{cases} 1, & t = 0 \\ 0, & t \neq 0 \end{cases} \quad (6.4)$$

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

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

$$\tilde{f}(t) = f(t)s_{\Delta T}(t) \quad (6.5)$$

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.

$$\tilde{F}(\mu) = \sum_{n=-\infty}^{\infty} f_n e^{-i2\pi\mu n \Delta T}, \quad f_n = \tilde{f}(n) = f(n\Delta T) \quad (6.6)$$

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

2D DFT Convolution Theorem .

Theorem 6.1 (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:*

$$F(u, v) := \mathcal{D}\{f(x, y)\} = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) e^{-2\pi i (\frac{ux}{M} + \frac{vy}{N})} \quad (6.7)$$

$$H(u, v) := \mathcal{D}\{h(x, y)\} = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} h(x, y) e^{-2\pi i (\frac{ux}{M} + \frac{vy}{N})} \quad (6.8)$$

and given the convolution of the two functions

$$(f \star h)(x, y) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m, n) h(x - m, y - n) \quad (6.9)$$

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

$$(f \star h)(x, y) = \mathcal{D}^{-1}\{MN \cdot F(u, v)H(u, v)\} \quad (6.10)$$

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

Proof.

$$(f \star h)(x, y) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m, n)h(x-m, y-n) \quad (6.11)$$

$$= \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \left(\sum_{p=0}^{M-1} \sum_{q=0}^{N-1} F(p, q) e^{2\pi i (\frac{mp}{M} + \frac{nq}{N})} \right) \left(\sum_{u=0}^{M-1} \sum_{v=0}^{N-1} H(u, v) e^{2\pi i (\frac{u(x-m)}{M} + \frac{v(y-n)}{N})} \right) \quad (6.12)$$

$$= \left(\sum_{u=0}^{M-1} \sum_{v=0}^{N-1} H(u, v) e^{2\pi i (\frac{ux}{M} + \frac{vy}{N})} \right) \left(\sum_{p=0}^{M-1} \sum_{q=0}^{N-1} F(p, q) \left(\sum_{m=0}^{M-1} e^{2\pi i (\frac{m(p-u)}{M})} \right) \left(\sum_{n=0}^{N-1} e^{2\pi i (\frac{n(q-v)}{N})} \right) \right) \quad (6.13)$$

$$= \left(\sum_{u=0}^{M-1} \sum_{v=0}^{N-1} H(u, v) e^{2\pi i (\frac{ux}{M} + \frac{vy}{N})} \right) \left(\sum_{p=0}^{M-1} \sum_{q=0}^{N-1} F(p, q) (M \cdot \hat{\delta}_M(p-u)) (N \cdot \hat{\delta}_M(q-v)) \right) \quad (6.14)$$

$$= \left(\sum_{u=0}^{M-1} \sum_{v=0}^{N-1} H(u, v) e^{2\pi i (\frac{ux}{M} + \frac{vy}{N})} \right) \cdot MNF(u, v) \quad (6.15)$$

$$= MN \cdot \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u, v) H(u, v) e^{2\pi i (\frac{ux}{M} + \frac{vy}{N})} \quad (6.16)$$

$$= MN \cdot \mathcal{D}^{-1} \{ FH \} \quad (6.17)$$

where

$$\hat{\delta}_N(k) = \begin{cases} 1 & \text{when } k = 0 \pmod{N} \\ 0 & \text{else} \end{cases} \quad (6.18)$$

□

Above, we make use of the following lemma

Lemma 6.2. Let j and k be integers and let N be a positive integer. Then

$$\sum_{n=0}^{N-1} e^{2\pi i \left(\frac{n(j-k)}{N} \right)} = N \cdot \hat{\delta}_N(j-k) \quad (6.19)$$

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

$$z^N - 1 = (z - 1)(z^{n-1} + \dots + z + 1) = (z - 1) \sum_{n=0}^{N-1} z^n. \quad (6.20)$$

thus giving us

$$0 = \left(e^{2\pi i(j-k)/N} - 1\right) \sum_{n=0}^{N-1} e^{2\pi i n(j-k)/N} \quad (6.21)$$

To prove the claim in eq. (6.19), 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. (6.19) 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. (6.21). 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. (6.18), 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:

$$F(\mu, \nu) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) e^{-i2\pi(\mu x/M + \nu y/N)} \quad (6.22)$$

$$= \sum_{x=0}^{M-1} e^{-i2\pi\mu x/M} \left[\sum_{y=0}^{N-1} f(x, y) e^{-i2\pi\nu y/N} \right] \quad (6.23)$$

$$= \sum_{x=0}^{M-1} e^{-i2\pi\mu x/M} \mathcal{F}_x\{f(x, y)\} \quad (6.24)$$

$$= \mathcal{F}_y\{\mathcal{F}_x\{f(x, y)\}\} \quad (6.25)$$

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

$$F(u) = \sum_{x=0}^{M-1} f(x) W_M^{ux}, \quad \text{where} \quad W_m := e^{-i2\pi/m} \quad (6.26)$$

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

$$F(u) = \sum_{x=0}^{2K-1} f(x) W_{2K}^{ux} \quad (6.27)$$

$$= \sum_{x=0}^{K-1} f(2x) W_{2K}^{u(2x)} + \sum_{x=0}^{K-1} f(2x+1) W_{2K}^{u(2x+1)} \quad (6.28)$$

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

$$W_{(2m)}^{(2n)} = e^{\frac{-i2\pi(2m)}{2m}} = e^{\frac{-i2\pi m}{n}} = W_m^n \quad (6.29)$$

$$W_m^{(u+m)x} = e^{\frac{-i2\pi(u+m)x}{m}} = e^{\frac{-i2\pi unx}{m}} e^{\frac{-i2\pi mx}{m}} = e^{\frac{-i2\pi ux}{m}} (1) = W_m^{ux} \quad (6.30)$$

$$W_{2m}^{(u+m)} = e^{\frac{-i2\pi(u+m)}{2m}} = e^{\frac{-i2\pi ux}{2m}} e^{-i\pi} = W_{2m}^u e^{-i\pi} = -W_{2m}^u \quad (6.31)$$

Thus we can rewrite eq. (6.28) as

$$F(u) = \sum_{x=0}^{K-1} f(2x) W_{2K}^{2ux} + \sum_{x=0}^{K-1} f(2x+1) W_{2K}^{2ux} W_{2K}^u \quad (6.32)$$

$$\implies F(u) = \left(\sum_{x=0}^{K-1} f(2x) W_K^{ux} \right) + \left(\sum_{x=0}^{K-1} f(2x+1) W_K^{ux} \right) W_{2K}^u \quad (6.33)$$

The major advance comes via using the identities eq. (6.29) to consider the

Fourier transform K frequencies later :

$$F(u+K) = \left(\sum_{x=0}^{K-1} f(2x) W_K^{(u+K)x} \right) + \left(\sum_{x=0}^{K-1} f(2x+1) W_K^{(u+K)x} \right) W_{2K}^{(u+K)} \quad (6.34)$$

$$\implies F(u+K) = \left(\sum_{x=0}^{K-1} f(2x) W_K^{ux} \right) - \left(\sum_{x=0}^{K-1} f(2x+1) W_K^{ux} \right) W_K^u \quad (6.35)$$

Comparing eq. (6.33) and eq. (6.35), 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:

$$\mathcal{D}_u\{f_{\text{even}}(t)\} := \sum_{x=0}^{K-1} f(2x) W_K^{ux} \quad (6.36)$$

$$\mathcal{D}_u\{f_{\text{odd}}(t)\} := \sum_{x=0}^{K-1} f(2x+1) W_K^{ux} \quad (6.37)$$

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. (6.36) 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. (6.36) 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. (6.22).

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.

Calculating the Hessian via FFT: A demonstration

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. 7, 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. 8, 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

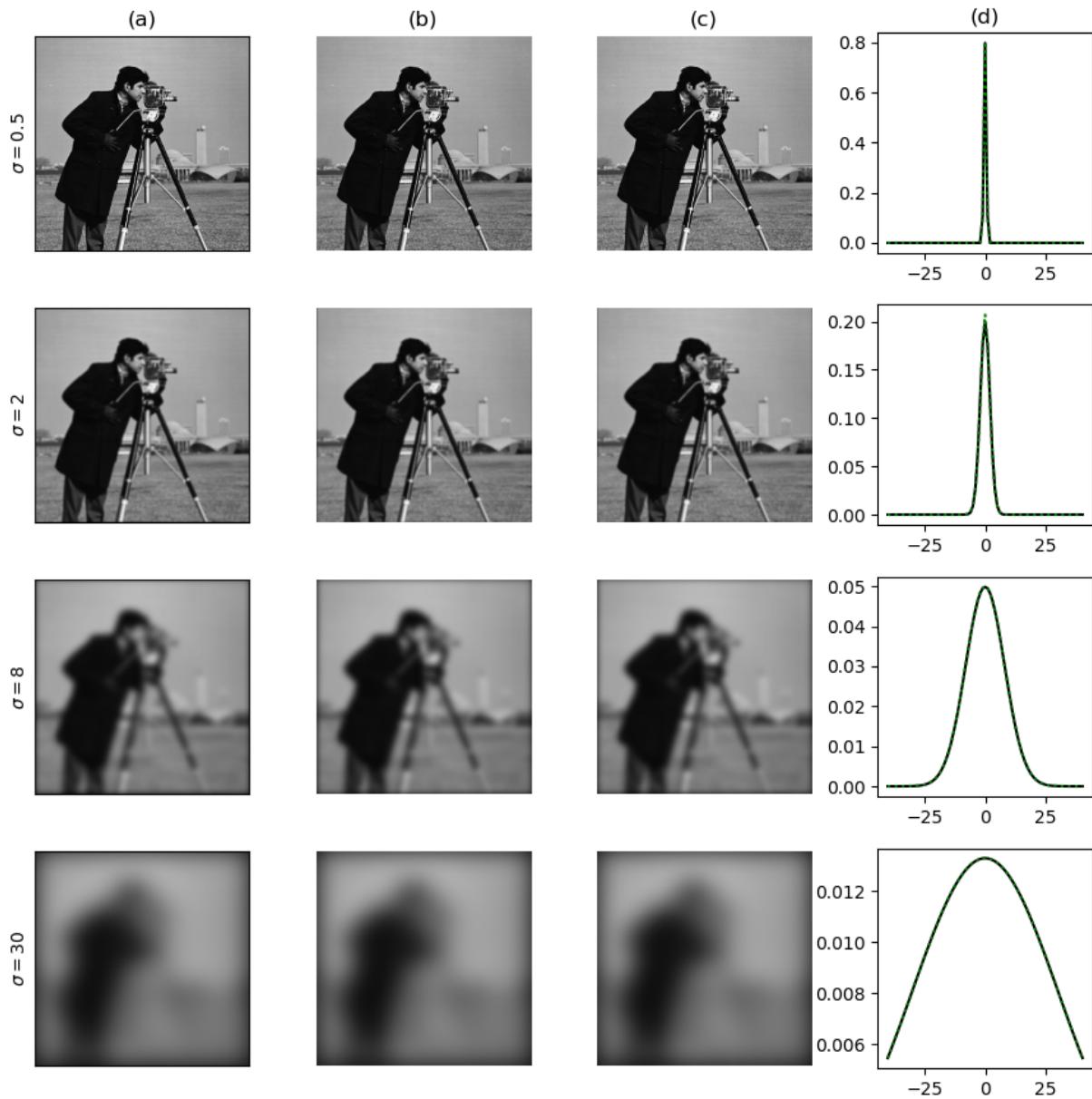


FIGURE 7: Compatibility of Gaussian convolution strategies

blurring method			MSE	MAE
spatial convolution, sampled kernel (A)			0.00054426	0.02015643
FFT convolution, sampled kernel (B)			0.00055205	0.02029916
FFT convolution, discrete kernel (C)			0.00054406	0.02015336

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

TABLE 1: MSE of Gaussian blurs ($\sigma = 0.3$) **TABLE 2:** MSE of Frangi scores $\sigma = 0.3$

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

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 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), which is an honest concern in image processing.

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 1, table 2, table 3 and table 4 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. 9, fig. 10, fig. 11, fig. 12, 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.

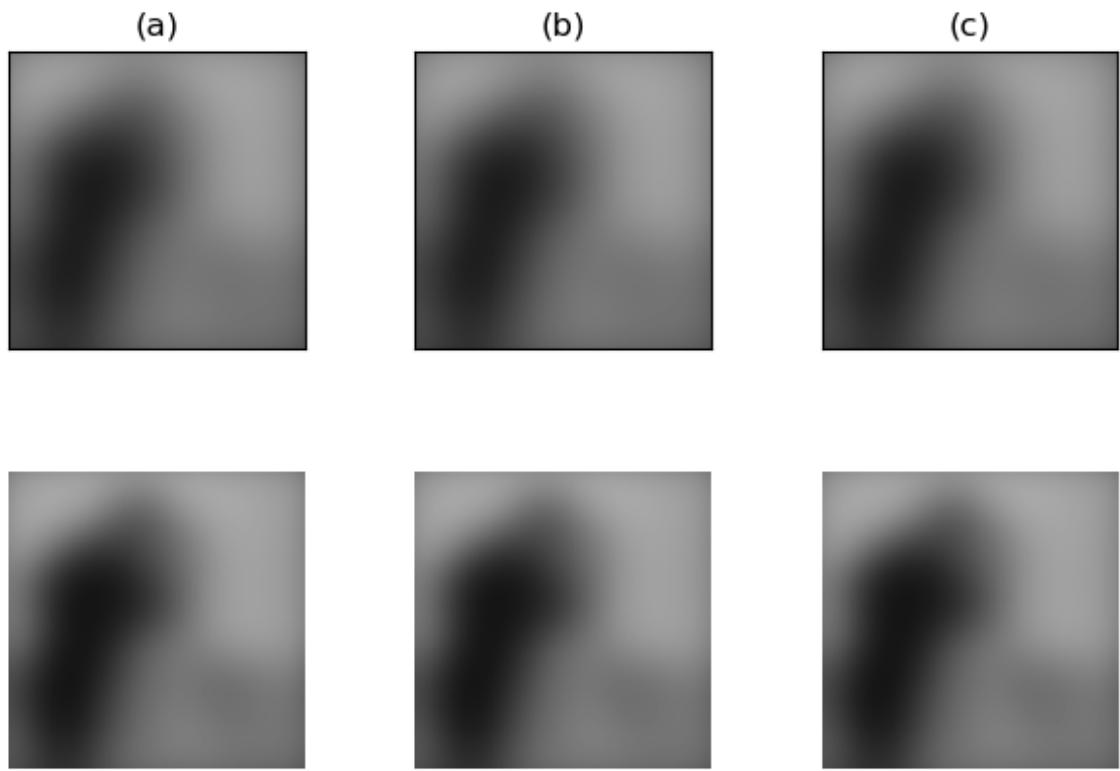


FIGURE 8: Iterative Gaussian blur

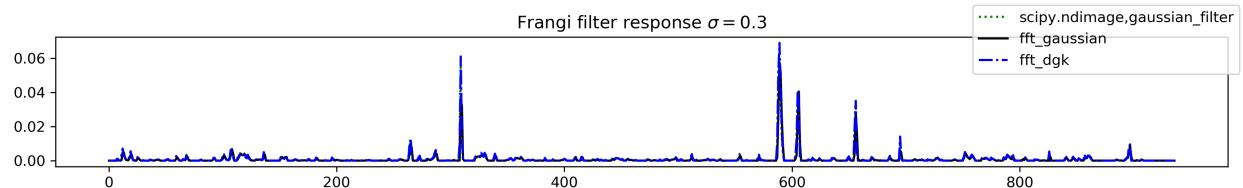


FIGURE 9: Image cross-section of Gaussian blurred images

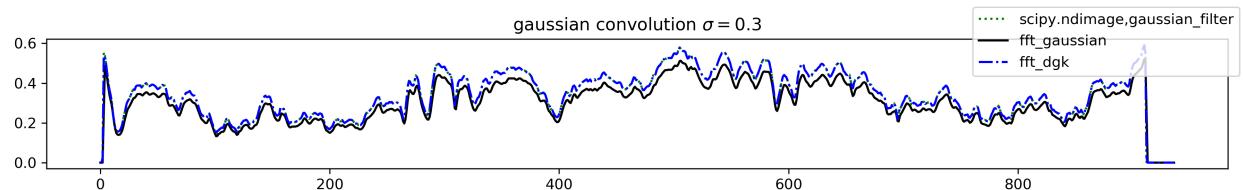


FIGURE 10: Image cross-section of Frangi targets images

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

TABLE 3: 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 4: MSE of Frangi scores $\sigma = 5$

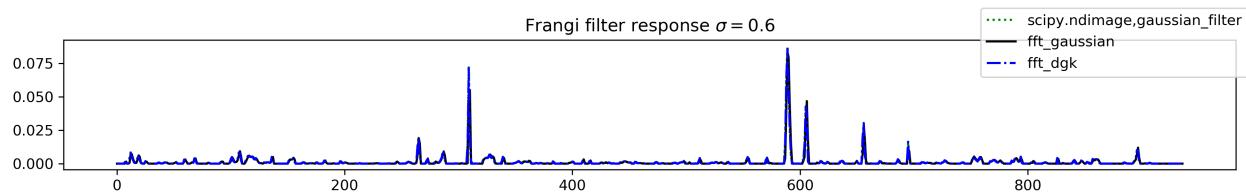


FIGURE 11: Image cross-section of Gaussian blurred images $\sigma = 5$

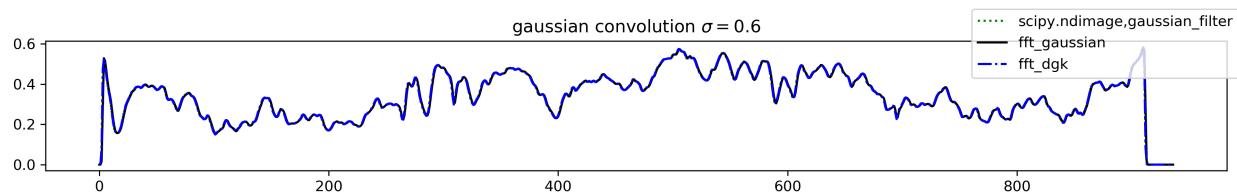


FIGURE 12: Image cross-section of Frangi targets images $\sigma = 5$

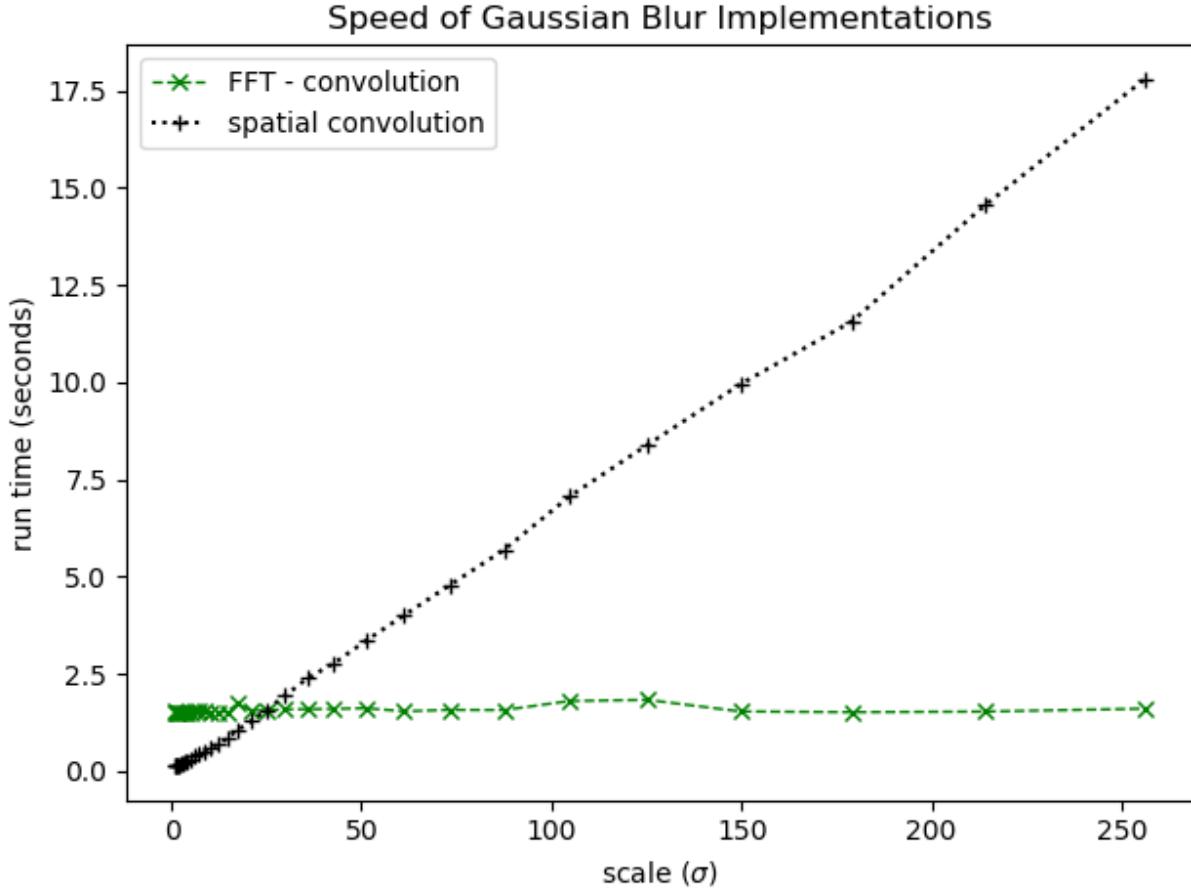


FIGURE 13: Time required

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 (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.

CHAPTER 7

MORPHOLOGICAL IMAGE PROCESSING

We describe some of the morphological methods in a really basic way and then briefly mention the thinning algorithm.

CHAPTER 8

RESEARCH PROTOCOL

Samples / Image Domain

We ultimately perform a multiscale Frangi prefilter on a subset of 201 color images of placental samples 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 [5]. 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. 14.

Each layer is roughly 1954x1200 pixels (with some occasional variation). In fig. 14, we see these four layers of a characteristic sample. fig. 14a is the base image. A cleaned, fixed placenta is placed on a table with a camera a fixed distance away, and a ruler and penny (presumably for redundancy) are placed nearby to aid registration and calibration of the resolution. The resolution of each sample is roughly 46 pixels per centimeter. fig. 14b 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. 14c and fig. 14d are each 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. 14e. The

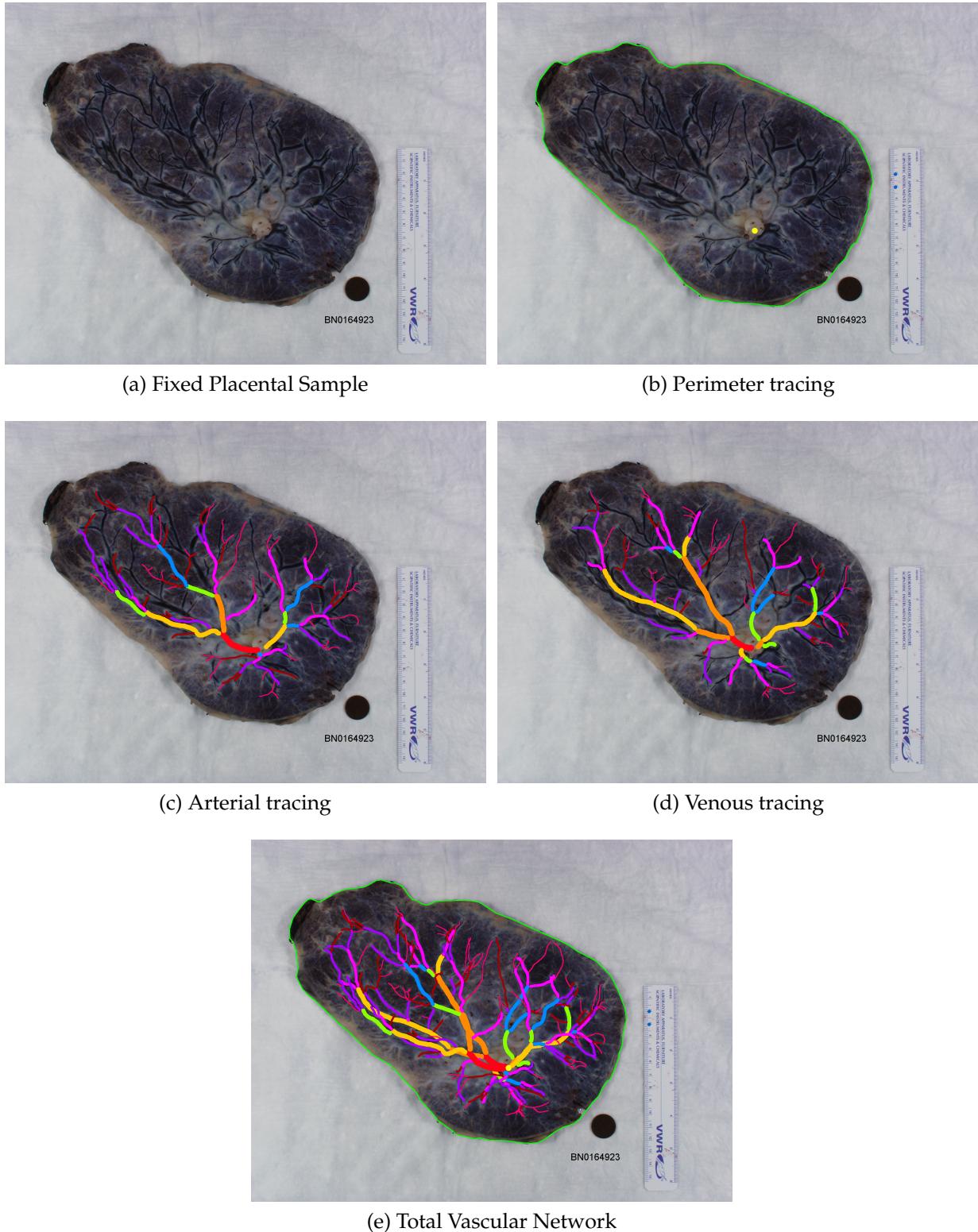


FIGURE 14: A representative placental sample and tracing

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 5: Vessel width color code for manual tracing protocol

coloration is meant to indicate the diameter of each vessel. The diameters are binned into 9 discrete widths, odd integers from 3 to 19 pixels. Vessels of smaller diameter are either binned to three or (quite frequently) left untraced. The correspondence between pencil color and (binned) vessel width used in the tracing protocol is given in table 5.

As stated in the introduction, the task of creating these samples, in particular the tracing and estimation involved in creating fig. 14c and fig. 14d is very 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 fig. 23, where we will demonstrate how the Frangi filter reacts to these problem areas. Our efforts at the eventual task of network completion must deal with these shortcomings and, in some circumstances, make subjective decisions like the manual tracer did.

Knowns and Unknowns

Since our final goal is a fully automated procedure, 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. 14e or fig. 15d without any user intervention.

For our purposes however, we will concede and provide a limited amount of information from the tracings, namely the provided placental perimeter (shown in green in fig. 14). 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. In fact, we have implemented this later algorithm, but unfortunately we achieve a subpar result on several images, and therefore we currently use it to improve upon the perimeter by removing “cuts”, which were previously reported as being inside the plate and sometimes led to large false positives.

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

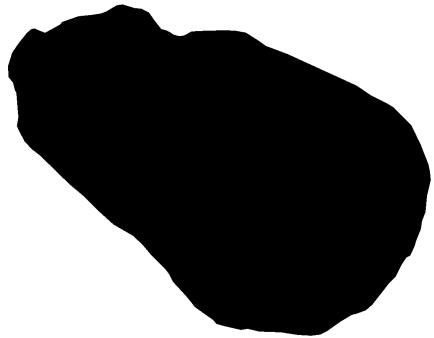
Building a sample suitable for use in our algorithm from fig. 14 is relatively simple. We zero outside the boundary of the plate (so as to not waste computational time calculating the differential geometry of a ruler, say), and also generate a binary mask to identify the plate. Finally, our vessel layers are combined and given as a binary trace, which we will use later for scoring. An example of the preprocessed samples used by the algorithm are given in section 8.2.

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. 15c is not actually produced directly by the extractor plug-in, but created when the 3 channel RGB image fig. 15b is imported at the start of the algorithm. This grayscale conversion is simply done for ease of analysis on the sample: although the Frangi filter is designed for arbitrary N-dimensional input [12], 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:

$$L = \frac{299}{1000} R + \frac{587}{1000} G + \frac{114}{1000} B \quad (8.1)$$

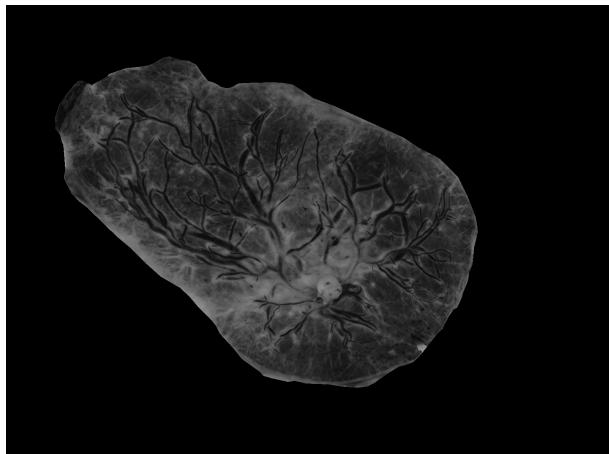
It should be noted that this choice is not a given—several other attempts have used the green channel unmodified, as in [5] and [2]. Preliminary and periodical rechecking has not indicated that such a conversion has any benefit over the luminance transformation for our image domain, although other placental samples with a different preparation method might benefit from it.



(a) Background Mask (in white)



(b) Sample with BG removed



(c) Grayscale

FIGURE 15: Preprocessed files from an NCS sample



(d) Trace / “Ground Truth”

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 perform 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 affected areas from consideration. This exclusion could be achieved by simply dilating the mask, but we opt to achieve it in a much more resource efficient manner: we iterate through an array of indices for the image where the boundary occurs and simply extend the mask R pixels in each direction (like a giant plus sign). Since the boundary of the placental plate forms a closed loop, the effect is very similar to convolving with a disk of radius R , but is much faster.

fig. 16 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. 16 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

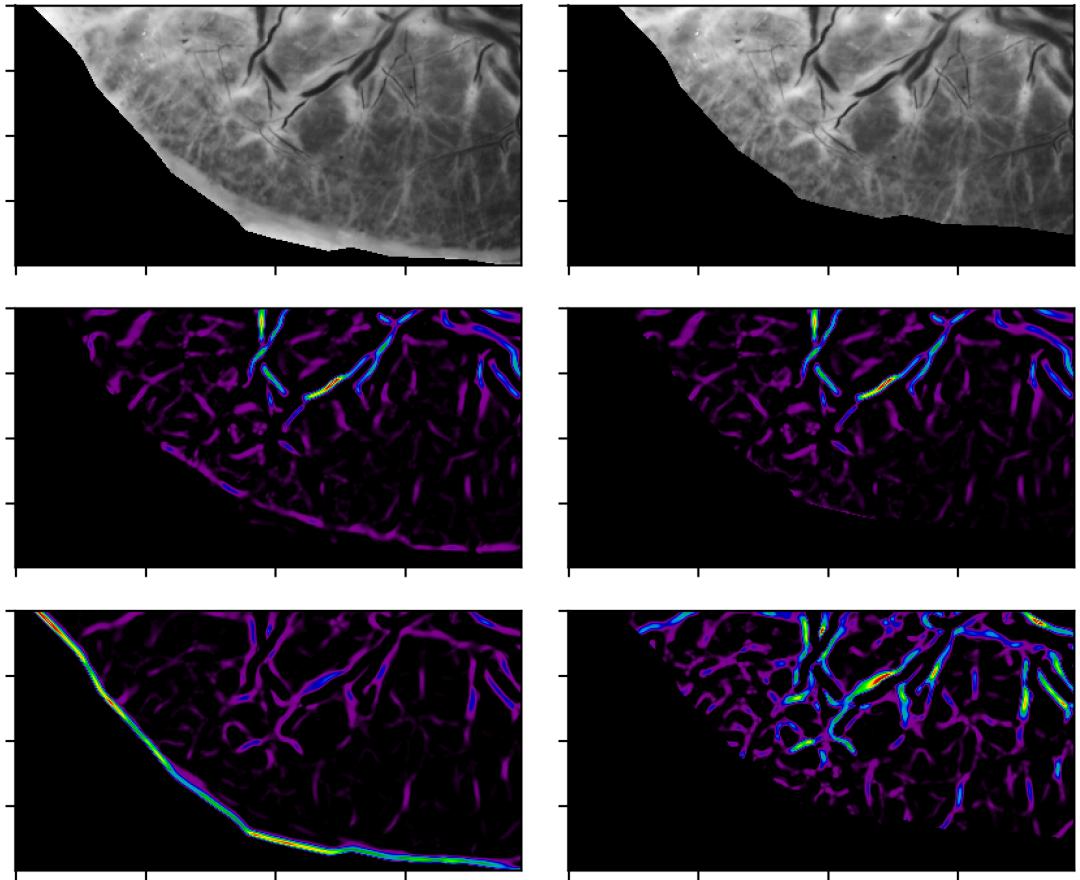


FIGURE 16: Effect of boundary dilation on Frangi responses

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], [5]). 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, and does not occur from a noise phenomenon.

The code for generating fig. 16 is found in the within the “`if __name__ == __main__`” block of the file `plate_morphology.py`, (so the figure will be generated when running `plate_morphology.py` as a top-level script from the command line). See appendix.

Deglaring

Despite best efforts when harvesting samples, a select number of the placental samples exhibit substantial glare, which leads to inaccuracies in identifying curvilinear content. Our protocol for deglaring is analogous to that performed in [5] 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, albeit at the expense of processing time.

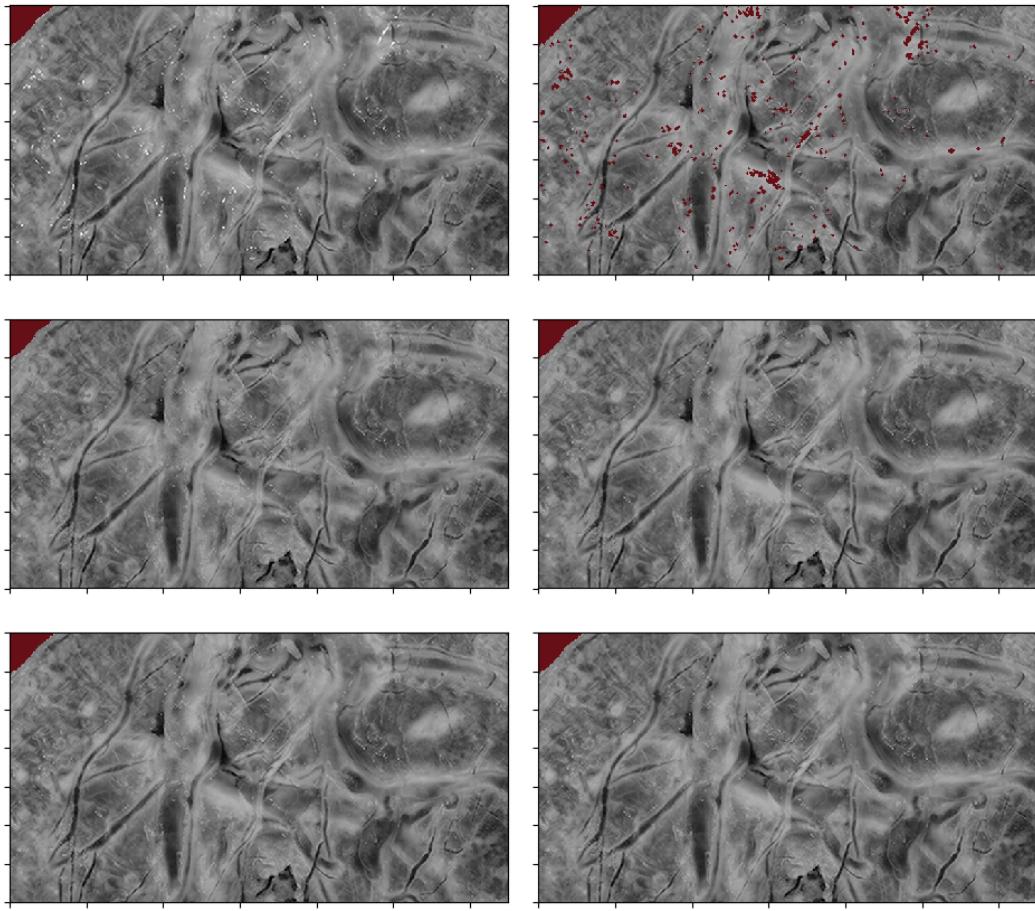


FIGURE 17: Deglaring a sample using a hybrid inpainting method

The function `inpaint_biharmonic` is based on [27], 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 [5], 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 unfortunately 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

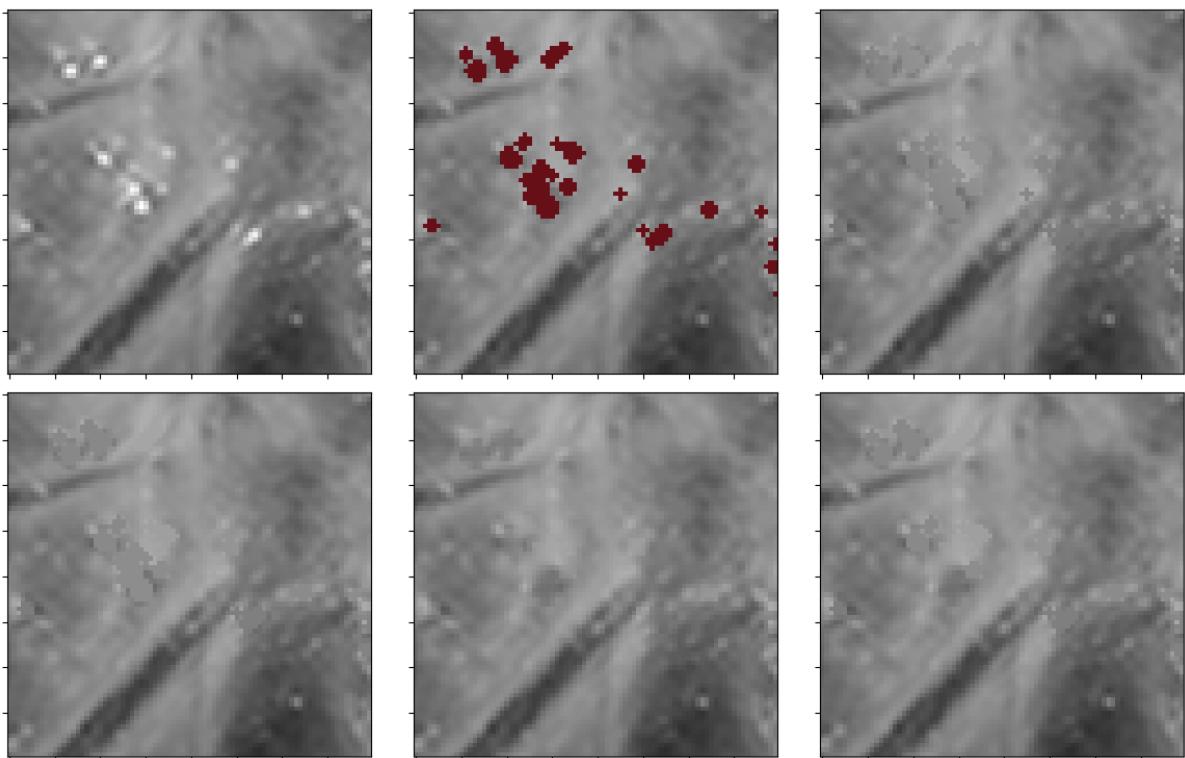


FIGURE 18: Comparison of glare inpainting methods (detail)

regions of glare. can be found in `preprocessing.py`. The first, called `inpaint_glare()` 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. 17, and a zoomed in portion is shown in fig. 18. 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 [28] that are 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 set of scales at which to probe will be denoted as $\Sigma := \{\sigma_1, \sigma_2, \dots, \sigma_N\}$, where each scale is indexed by increasing order.

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 [12], 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 [16], it is reasonable and natural to select these logarithmically; that is, for some selected inputs $m < M$ we have

$$\sigma_1 = 2^m, \sigma_j = 2^{(m + \frac{M-m}{N-1}j)}, \sigma_N = 2^M \quad (8.2)$$

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 $\sigma = 2$, we would not expect a substantial difference in response between, say, $\sigma = 46$ and $\sigma = 47$. Historically, there was another benefit of using a logarithmic scale space: 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, although we still obviously wish to avoid frivolous, redundant calculations no matter the speed of the implementation.

The optimal choice of scale sizes to probe at is intuitively dependent on the resolution of the image. If there is no particular care taken in selecting a minimum and maximum scale at which to probe, then we must assure that our Frangi filter is “normalized” in such a way that there is a decay in response past certain values. That is, probing at a unreasonably large (or small) scale (say $\sigma = 1000$ or $\sigma = .0001$) should result in an almost null response throughout the image. We will approach this issue in our discussion of “variable thresholding.”

The Research Protocol

Once we have chosen this set of scales 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 calculate the eigenvalues of each (2x2) Hessian matrix and then compute the Frangi filter according to section 3.1 and ???. 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.

```
% DECISIONTREE  
For each sample:
```

- A) Preprocessing
 - 1) RGB to single channel via Luminance Transform
 - 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, Hyy
 - 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.

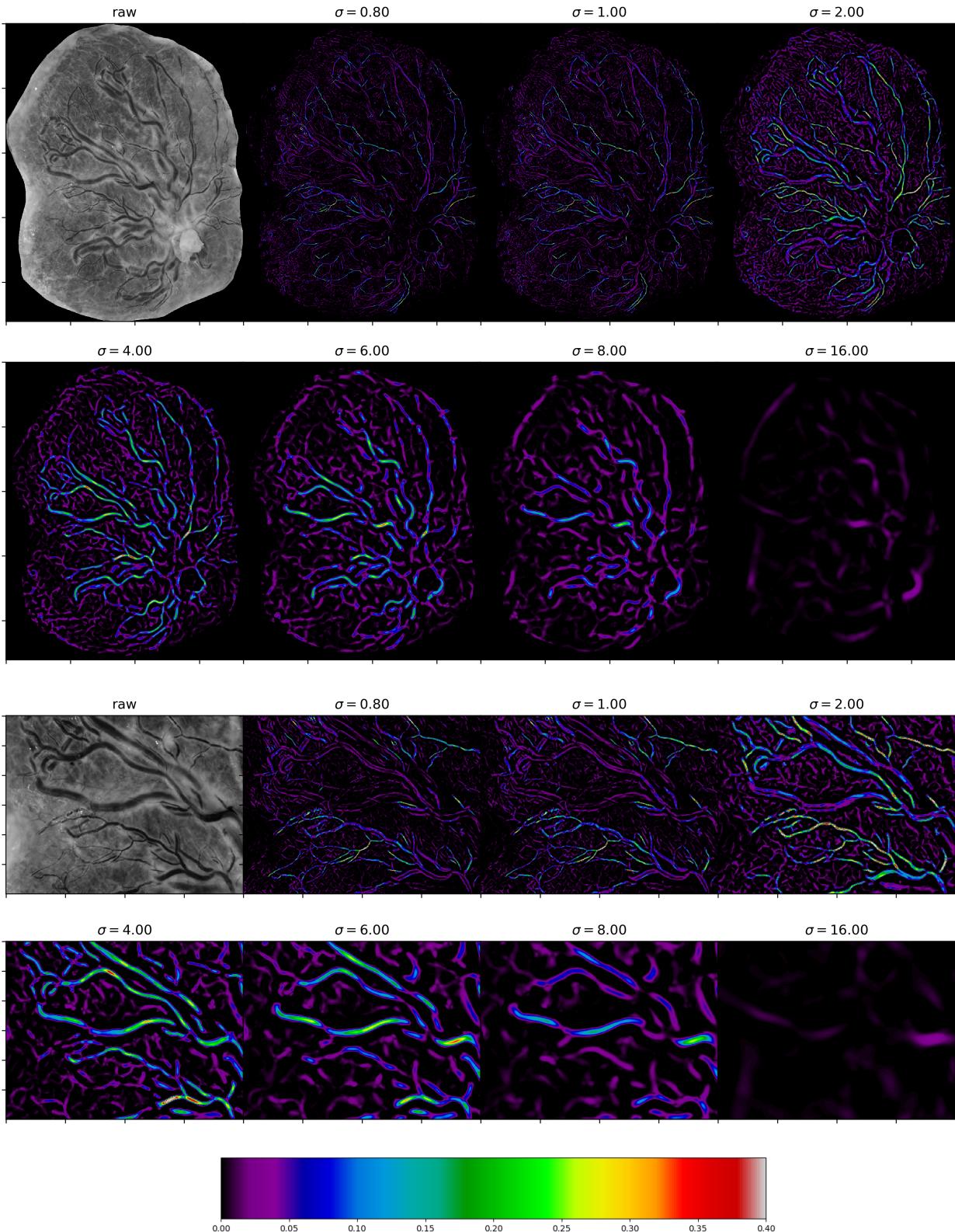


FIGURE 19: Example scalewise Frangi output (plate and inset) (Example 1)

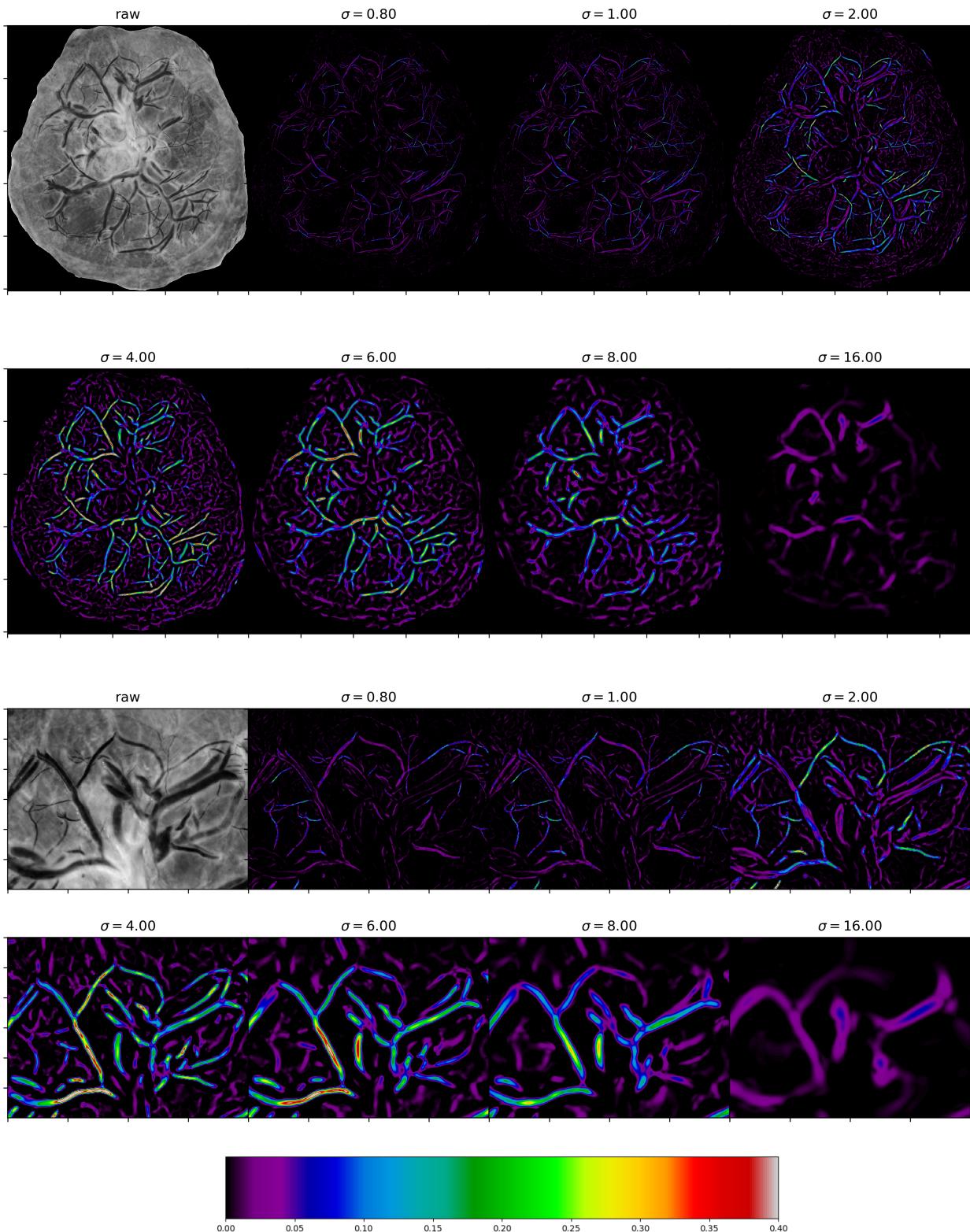


FIGURE 20: Example scalewise Frangi output (plate and inset) (Example 2)

CHAPTER 9

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 parameter $\beta = 0.35$ and standard structureness parameter $\gamma = 0.5$, 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. Our goal in this section is to provide a close up look at the Frangi filter on two samples, and then provide some measures of the Frangi output's correspondence with the "ground truth" network tracings across all 201 images, without explicitly performing segmentation. However, for visual demonstration, we will employ both simple thresholding techniques (arbitrary fixed and nonzero-percentile) described in chapter 5. In chapter 10 we will develop and analyze more sophisticated Frangi-based segmentation techniques and compare their performance to these rudimentary thresholding techniques across the entire dataset. For this chapter we will content ourselves with analysis of the raw Frangi vesselness measure within our image domain, and use our simple thresholded results for visual demonstration alone.

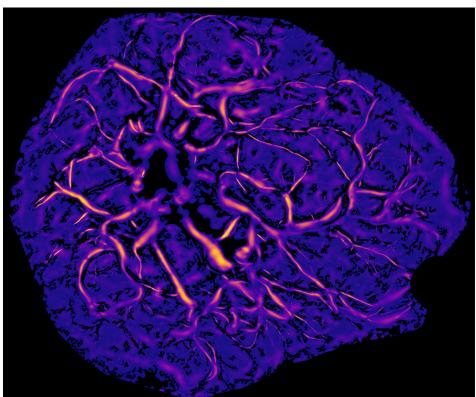
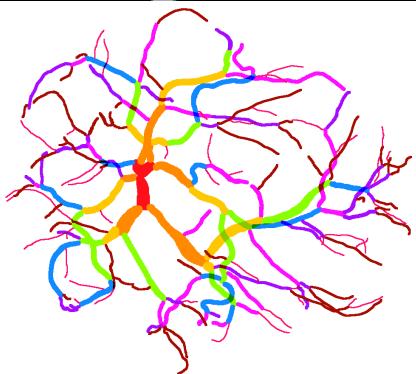
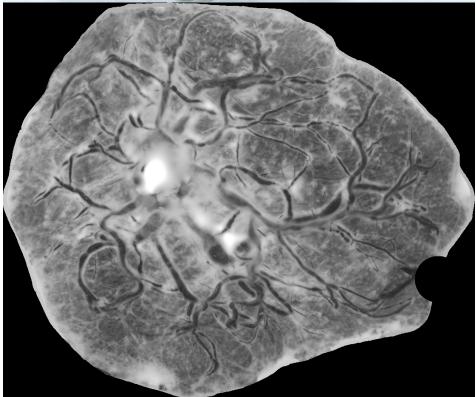
Sample visual output

In fig. 21 and fig. 22 we take a look at the Frangi output (\mathcal{V}_{\max}) for a well-behaved sample. In the left column we have (from top to bottom) the un-preprocessed placental sample, the preprocessed sample, the color trace(ground truth), and \mathcal{V}_{\max} . In the top-right, we have for each $n = 0, \dots, 19$ scales, we have the size of the scale σ_n , the 95th percentile score at that scale α_p where $p = 95$, and the maximum vesselness score at that

scale ($\max(V_\sigma)$). The bottom-right set of images are two rudimentary thresholds of the Frangi result. The top is the scalewise nonzero percentile threshold (95th percentile) and the bottom is a strict threshold of 0.4. left and right images are two rudimentary thresholds of \mathcal{V}_{\max} . The bottom left is nonzero percentile filtering (thresholded at the 95th percentile, and the right image is a simple threshold at $\alpha = 0.4$. For each pixel that passes the threshold, each of these graphs is color coded to indicate from which scale the maximum value at that pixel occurs, colored according to the scale on the left. These two thresholds are included mostly to further show the dependence of scale on which vessels are identified.

Variations in the Data Set and Imperfections of the Ground Truth

We must first qualify binary classification. There are limitations intrinsic to the image domain and tracing protocol that make our ground truth tracing somewhat inaccurate. In fig. 23 we demonstrate a few common issues with the samples. The four figures show (top left) the original colored raw sample, (top right) the ground truth tracing, (bottom left) \mathcal{V}_{\max} , and (bottom right) the confusion matrix after some segmentation strategy. The green arrow points to the umbilical stump. You can see there is circular noise around this point, and in general perfusion around this point is very low—many of these vessels have been estimated by the tracer. The orange arrows represent points where perfusion is very low, perhaps caused by a clot in the blood vessel. The pink arrows point to vessels that were not traced, although they are clearly visible. The blue arrows point to where the shape of the vessel and the trace clearly do not agree. There are many more examples of these in the frame, and many more across all samples. The red arrow points to an issue unrelated to the ground truth, but an issue that arises when selecting scales in our multiscale methodology—this is a point of dark curvature that represents noise at larger scales. As you can see from the \mathcal{V}_{\max} of this inset, there is a positive response in this dead space between vessels, and there is another



n	σ_n	α_p	$\max(V_\sigma)$
0	0.353	0.054	0.986
1	0.424	0.059	0.979
2	0.509	0.065	0.970
3	0.611	0.076	0.973
4	0.733	0.089	0.988
5	0.880	0.096	0.991
6	1.056	0.108	0.991
7	1.267	0.130	0.970
8	1.521	0.166	0.973
9	1.825	0.223	0.978
10	2.190	0.292	0.984
11	2.629	0.319	0.968
12	3.155	0.326	0.994
13	3.786	0.355	0.998
14	4.544	0.405	0.999
15	5.454	0.376	0.963
16	6.545	0.318	0.950
17	7.855	0.304	0.958
18	9.427	0.328	0.916
19	11.313	0.352	0.916

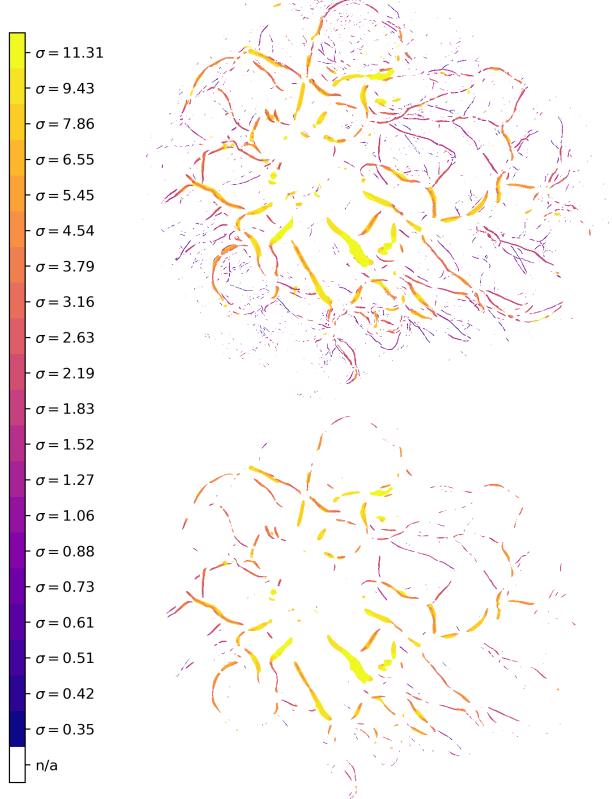


FIGURE 21: Vesselness score, percentile thresholds, and simple thresholds, Example 1



n	σ_n	α_p	$\max(V_\sigma)$
0	0.353	0.054	0.999
1	0.424	0.059	0.999
2	0.509	0.065	0.999
3	0.611	0.076	0.999
4	0.733	0.089	0.999
5	0.880	0.096	0.981
6	1.056	0.108	0.916
7	1.267	0.130	0.889
8	1.521	0.166	0.917
9	1.825	0.223	0.938
10	2.190	0.292	0.935
11	2.629	0.319	0.961
12	3.155	0.326	0.987
13	3.786	0.355	0.987
14	4.544	0.405	0.971
15	5.454	0.376	0.991
16	6.545	0.318	0.888
17	7.855	0.304	0.876
18	9.427	0.328	0.930
19	11.313	0.352	0.900

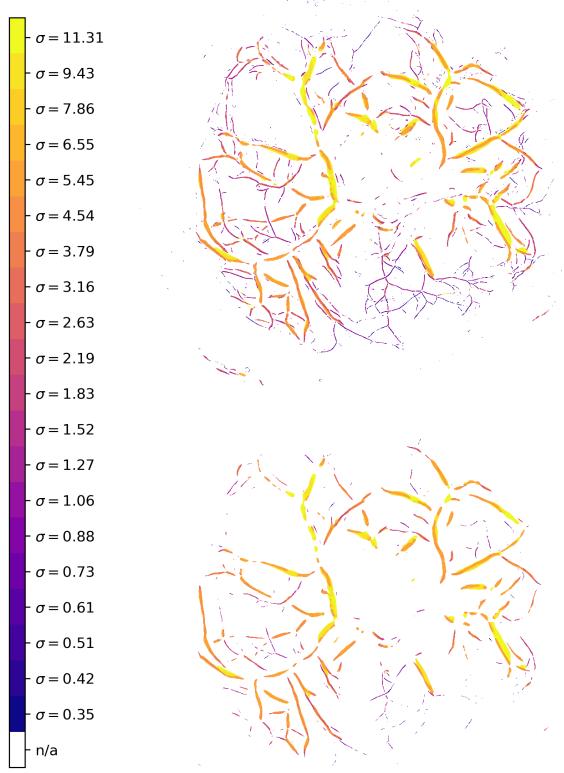
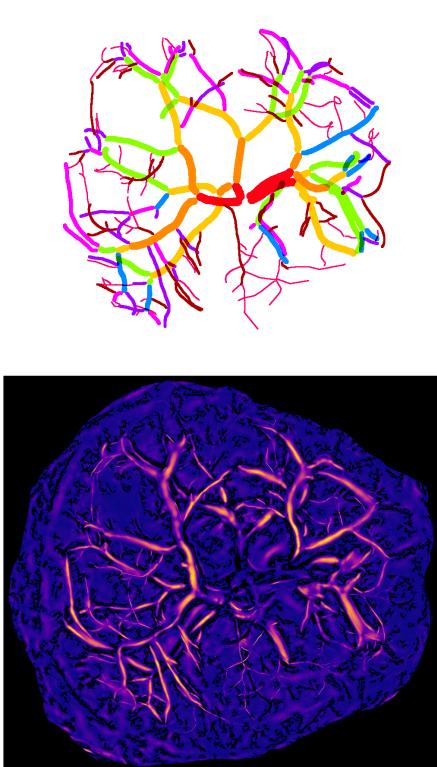


FIGURE 22: Vesselness score, percentile thresholds, and simple thresholds, Example 2

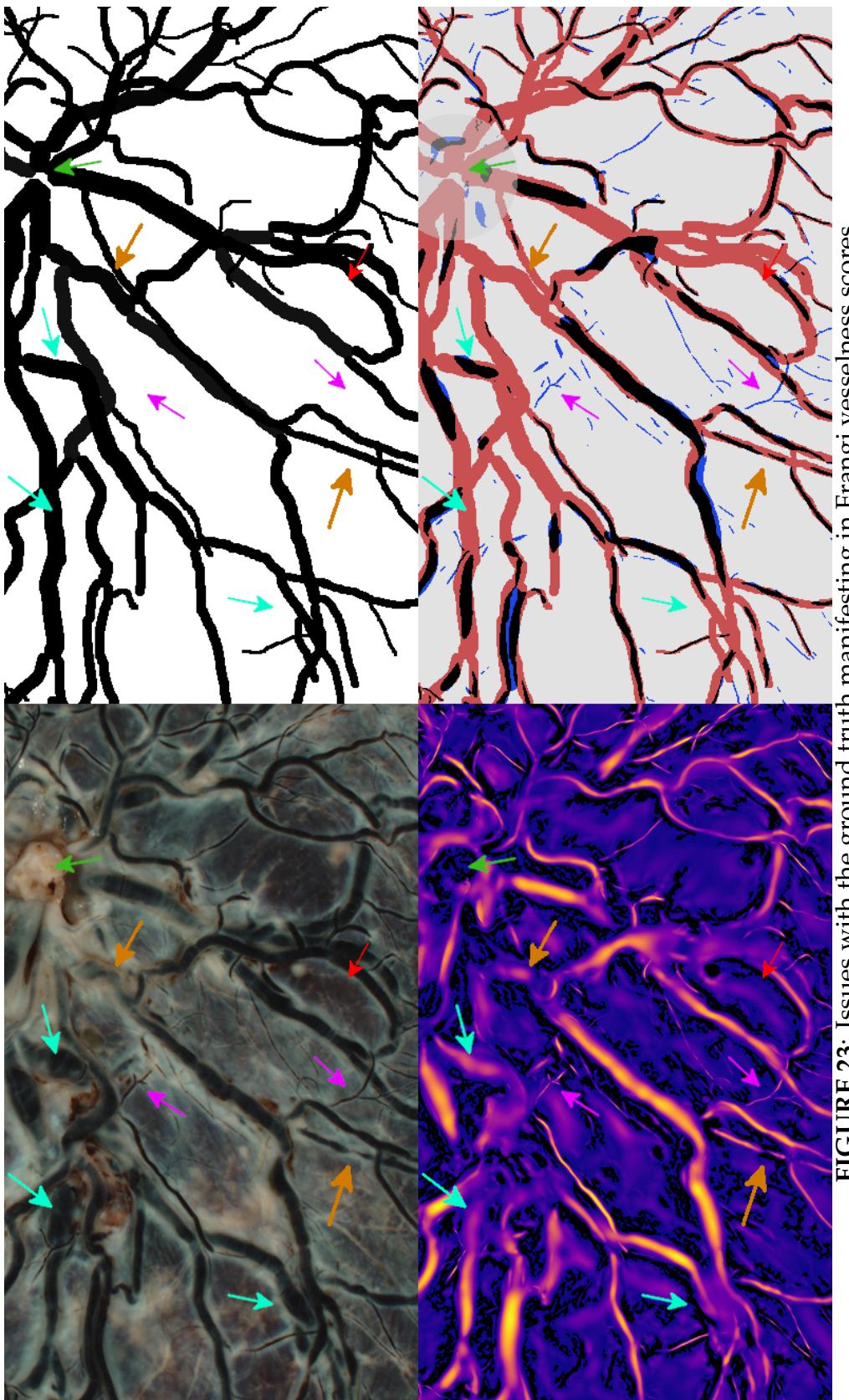


FIGURE 23: Issues with the ground truth manifesting in Frangi vesselness scores

one that appears in the bottom left between two close vessels of similar size.

The sample we examined was a relatively well-behaved sample compared to others. Conversely, in fig. 24, we give examples of some sampled that did not respond well to the Frangi filter, nor any of our Frangi-based segmentation methods. As can be seen from this gallery, poor perfusion of the blood vessels and large amounts of high contrast background is the common thread. One particular sample has very large arteries that were not reported. Increasing the scale size might fix the issue for this particular sample.

We have empirically identified a few reasons for issues. Although we have in many cases removed border areas from consideration where they would cause a large false response (see section 8.2.1 for our discussion on boundary dilation), some samples still demonstrate noise around the “collar” of each plate. This can marginally be seen in fig. 22—there is a large Frangi response in the bottom left that was not removed, and thus shows up in each simple segmentation.

Results

In fig. 25, we demonstrate the usefulness of stricter parameters for the Frangi filter. Since the Frangi vesselness measure is a "probability-like" score, we should be interested—without doing any actual segmentation yet—to what extent this score aligns with the ground truth in a cumulative sense. We should hope at least that larger values of our vesselness measure should occur along vessels and not within the background. We define the cumulative vesselness ratio by “integrating” the max vesselness score over pixels in the ground truth and over the entire image and considering the ratio:

Definition 9.1. *The cumulative vesselness ratio for a particular parametrization of the multiscale Frangi filter is given by*

$$CVR(\mathcal{V}_{\max}) := \frac{\sum_{G \subset I} \mathcal{V}_{\max}(x_0, y_0)}{\sum_I \mathcal{V}_{\max}(x_0, y_0)} \quad (9.1)$$

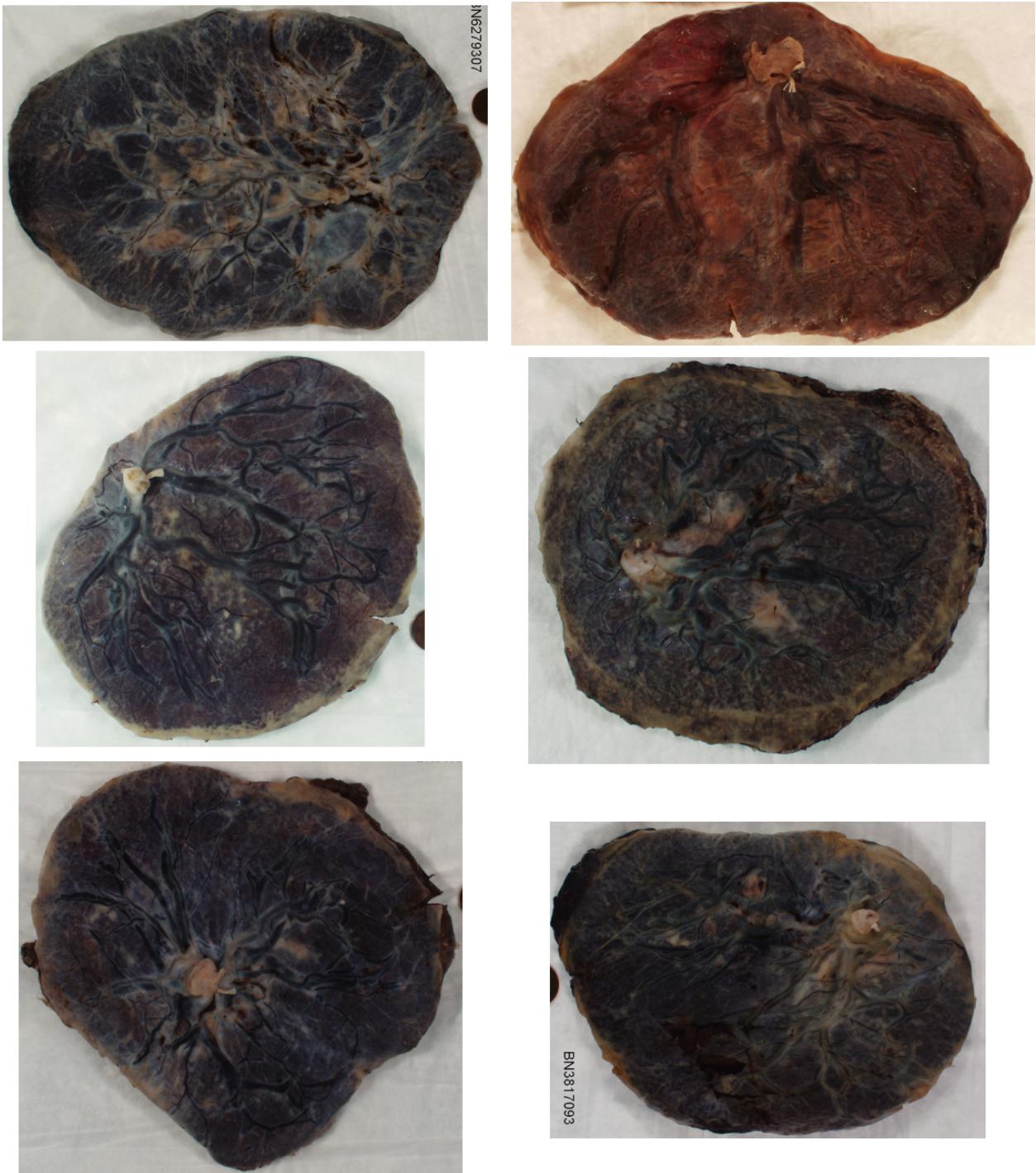


FIGURE 24: Bad samples

where the sums on top and bottom are being carried out for all pixels (x_0, y_0) in the "ground truth" subset G in the image I and over the entire image I itself, respectively.

fig. 25 and fig. 26 show \mathcal{V}_{\max} for two well-behaved samples with reported CVR for 9 various combinations of β and γ . Two combinations in particular, labeled "Anisotropy Factor" and "Structureness Factor," have parameter choices so that the structureness and anisotropy factors (respectively) are exactly one throughout the entire image. Comparing all parameter combinations, we see in both cases that stricter parameters (smaller β and larger γ) correspond to an increased CVR. That is, the more selective our Frangi filter is, the better it aligns with the vascular content within the image. This suggests not only that the Frangi filter is well suited to this image domain, but also that choosing stricter parameters for the Frangi filter concentrates the Frangi response within vascular features more and reduces background noise. Each of these sample was run over the same scales (logarithmically spaced from $2^{-1.5}$ to $2^{3.5}$ with $N = 20$ and the color output of each \mathcal{V}_{\max} is independently normalized between 0 and 1.

We wish to see if the observations in fig. 25 and fig. 26 are common to our entire sample set. In fig. 27 we show how the CVR is affected for 25 of the "best" samples, that is, those that performed better in segmentation techniques relative to other samples, regardless of Frangi parametrization or particular segmentation method. This shows that the results of fig. 25 and fig. 26 hold in general for all "well-behaved" samples in our image domain.

fig. 28 depicts, for pixels identified as vascular within the ground truth of a particular sample, the relationship between the scale at which \mathcal{V}_{\max} occurs for a particular pixel and the width of the vessel according the ground truth, and the bottom figure is a coloration of the ground truth with this same information. We can see that in general, larger scales correspond with larger reported vessel widths, and smaller scales correspond with smaller reported vessel widths. In fig. 29 we repeat these same

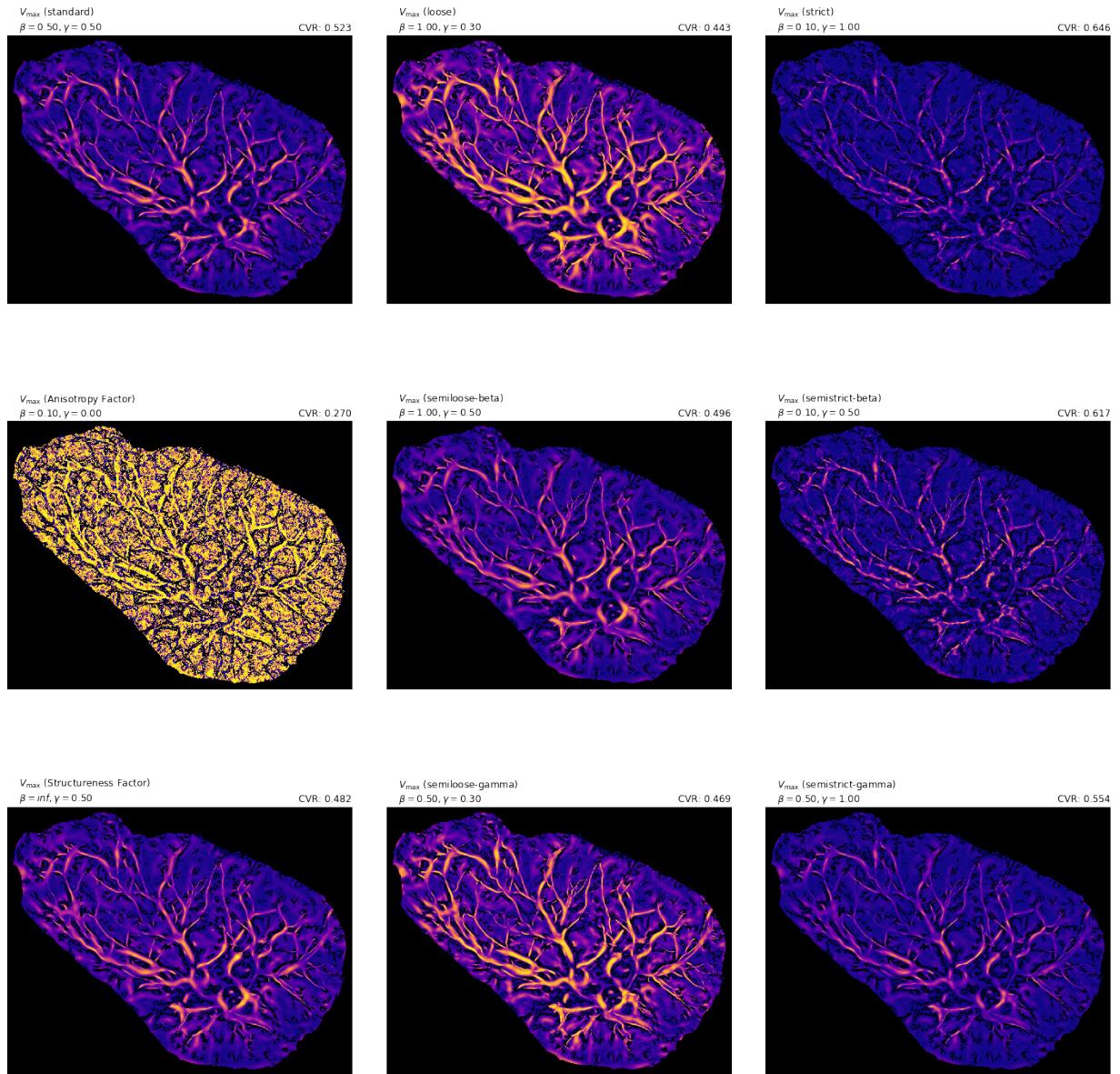


FIGURE 25: V_{\max} and CVR for varying multiscale Frangi parametrizations (Example 1)

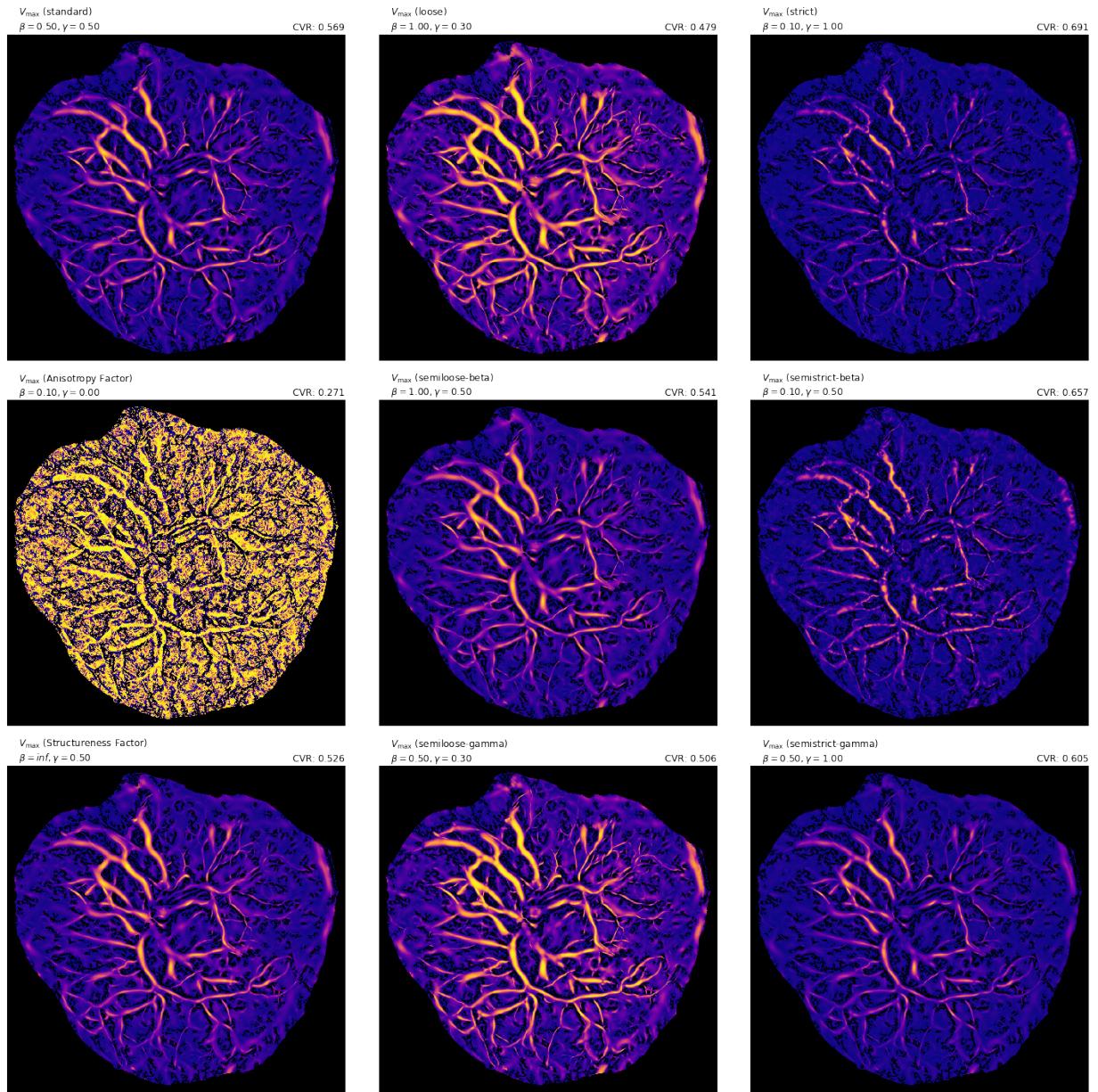


FIGURE 26: V_{\max} and CVR for varying multiscale Frangi parametrizations (Example 2)

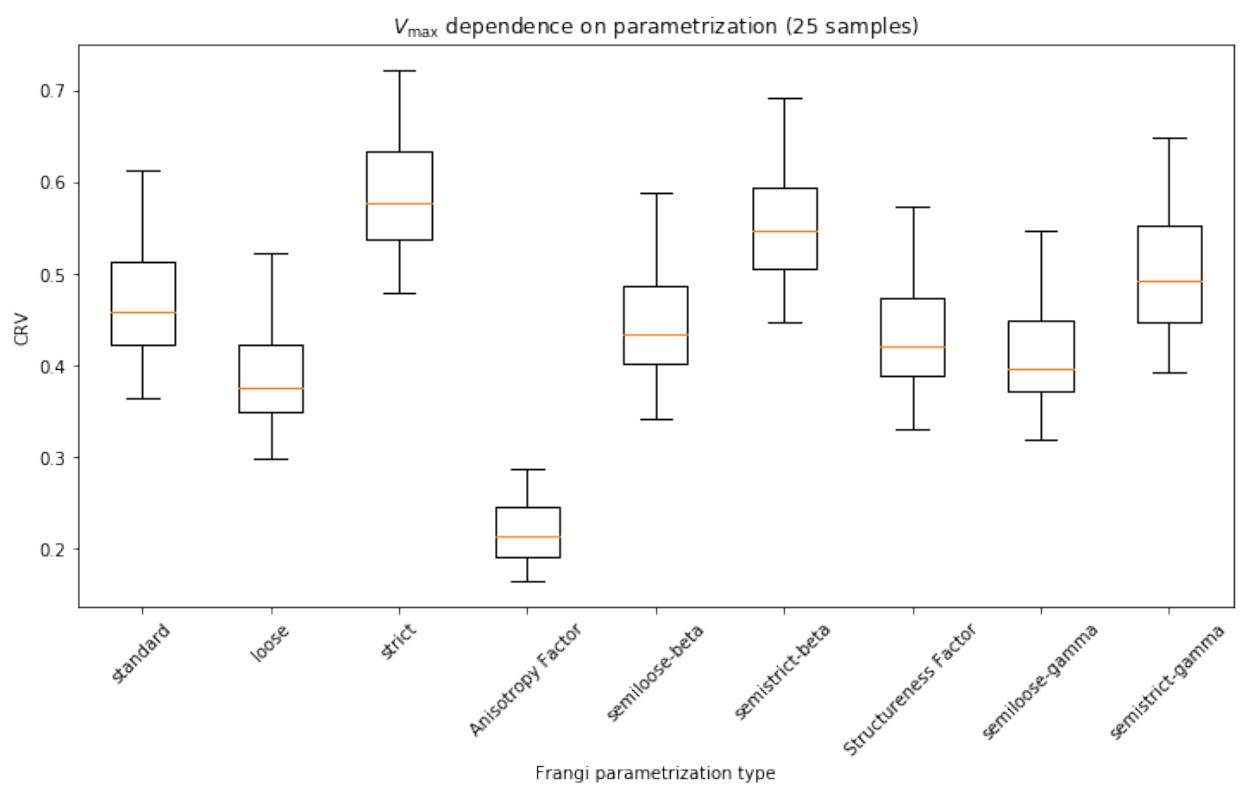


FIGURE 27: CVR scores of 25 samples under varying parametrizations

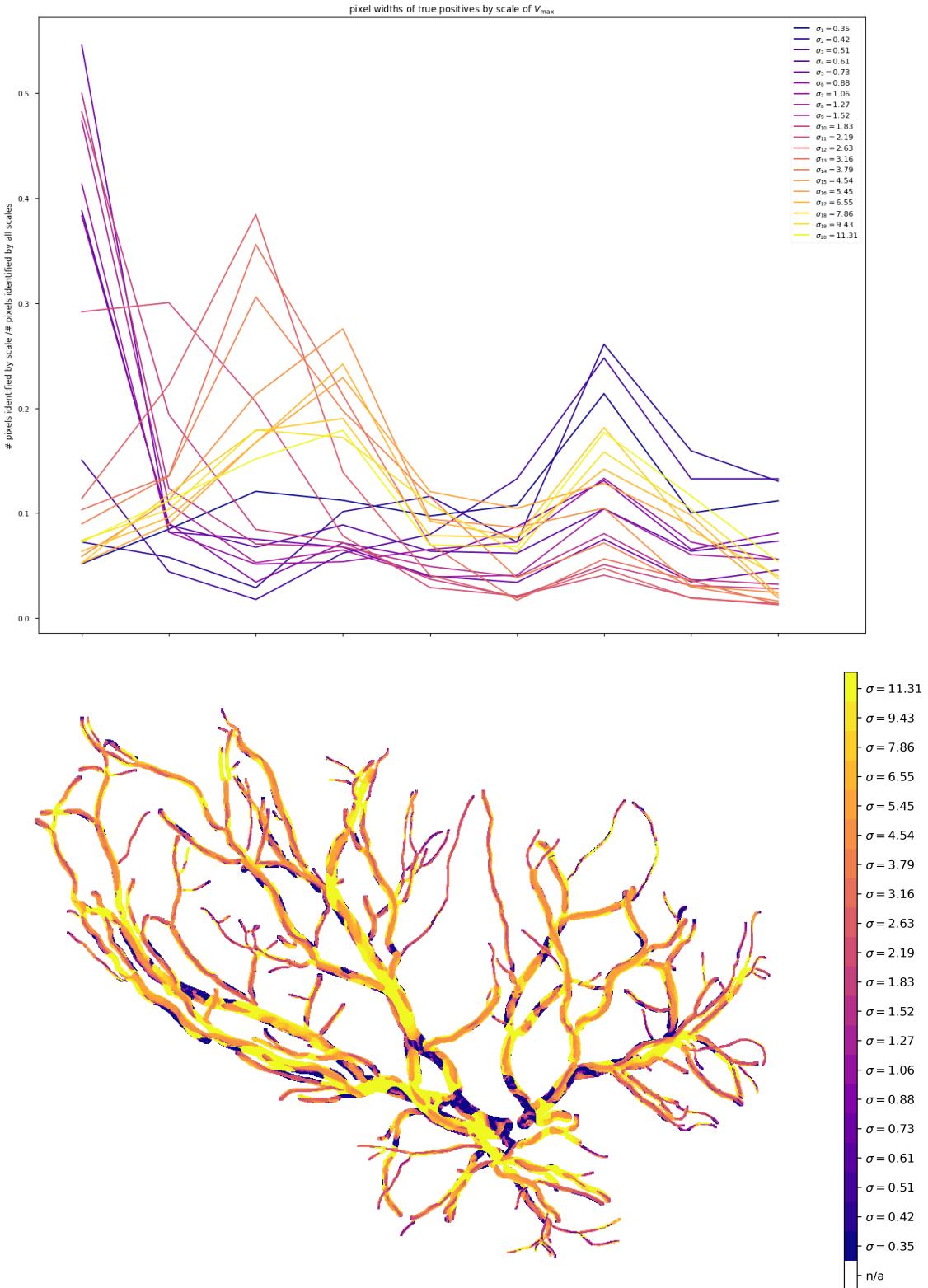


FIGURE 28: Scale of maximum Frangi score for true positives and false negatives

observations but limit ourselves only to those pixels identified in the ground truth for which some \mathcal{V}_o occurred above some scale's 95th percentile (as will be explained in chapter 10, these pixels are the “true positives” of the nz-percentile segmentation method). The y-axis of each of the charts is the ratio of pixels at which \mathcal{V}_{\max} is attained at the given width over all scales for that particular reported width.

fig. 30 is a recontextualization of fig. 29, where the total number of pixels (rather than a ratio) is considered for the same sample. We see that many more vascular pixels are identified as having width 7, and that in general we cannot provide a completely faithful mapping of reported vascular width to scale of largest response, but the trend of larger widths having larger scale responses and small widths having smaller scale responses remains true.

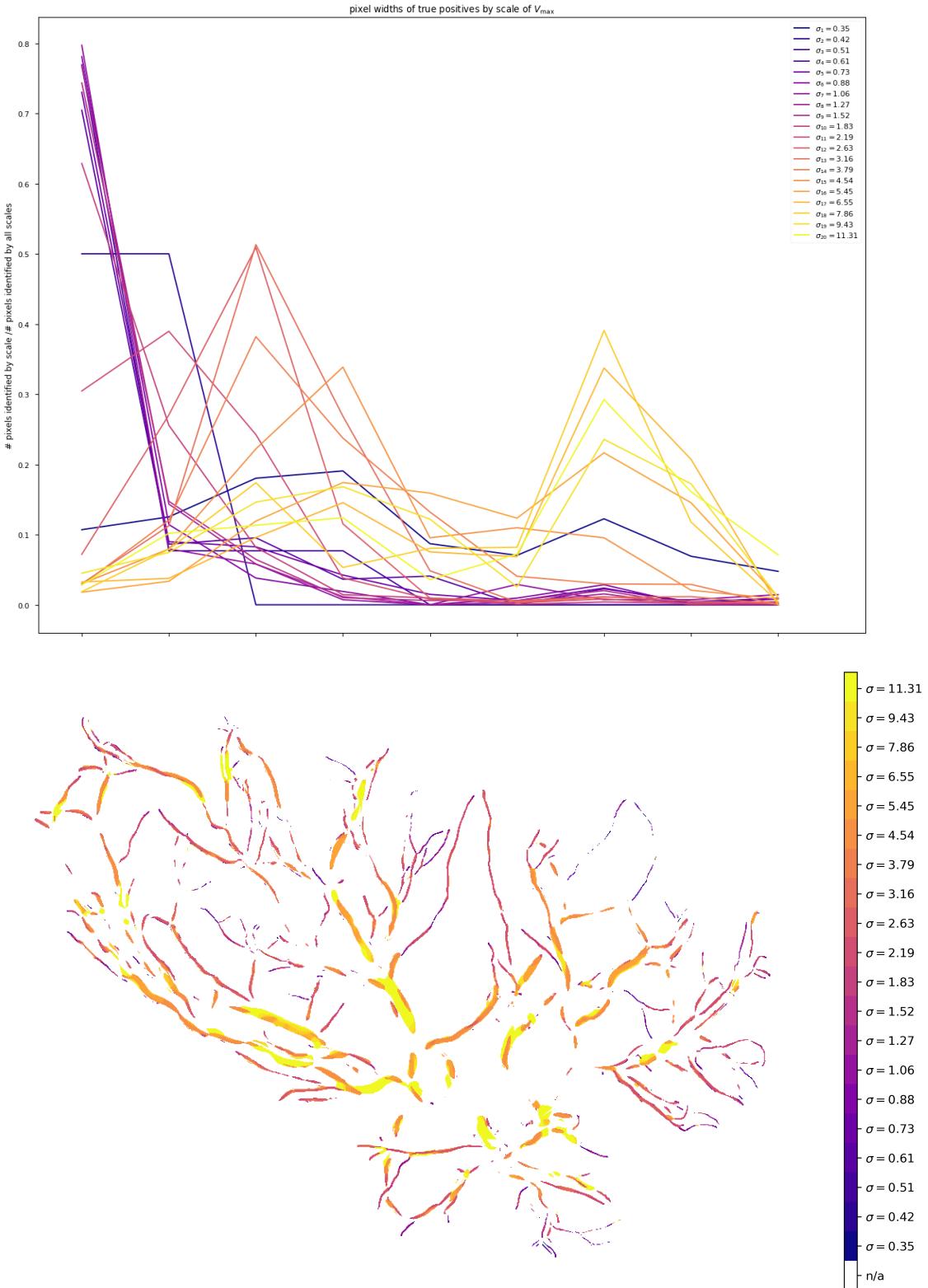


FIGURE 29: Scale of maximum Frangi score for true positives only (95th-percentile filtering)

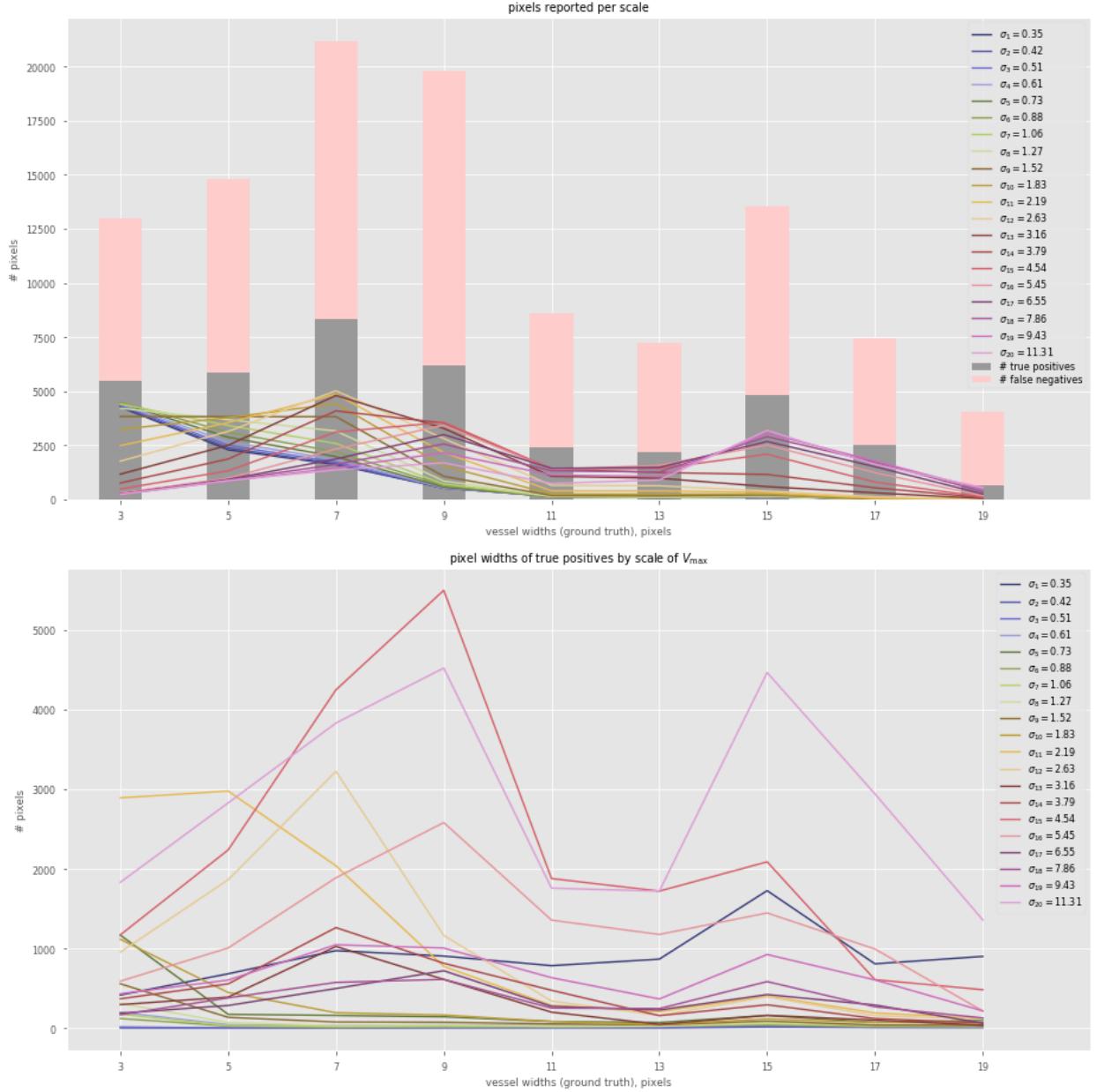


FIGURE 30: Pixel Width of Ground Truth vs. Scale Length for True Positives

CHAPTER 10

SEGMENTATION

We discuss how we use the Frangi as a prefilter and discuss several different segmentation strategies based upon our filtered result. These are compared to a ISODATA threshold. First, we define some standard quantitative measures of success for segmentation methods.

Binary Classifications and the confusion matrix

We wish to come up with a means of gauging the success of an arbitrary segmentation and will adopt a binary classification model to do so. We end up with a boolean matrix the size of the image, and compare it to the ground truth, the trace, and then compare them to get the number of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). We demonstrate these four labels visually with a confusion matrix, as in fig. 31. The lighter gray represents true negatives (TN), red represents false negatives (FN), blue represents false positives (FP), and black represents true positives. The darkest gray area is the background mask (pixels are not considered at all in gauging the success of segmentation).

Although there are many measures to gauge the success of binary classification, we will focus on two that we find particularly illustrative. The first is precision, given by

$$\text{precision} = \frac{TP}{TP + FP} \tag{10.1}$$

and the second is the Matthews Correlation Constant (MCC), given by

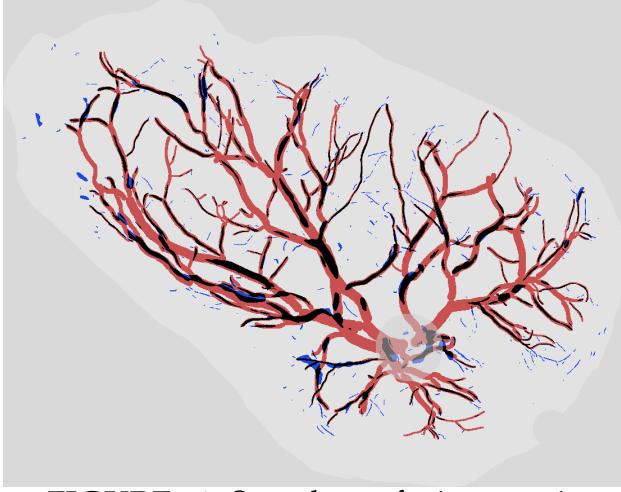


FIGURE 31: Sample confusion matrix

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \quad (10.2)$$

where precision is a ratio between 0 % and 100% and the MCC is a measure between -1 and 1. Precision (or positive predictive power) is of course a ratio between how many pixels were labeled correctly and all pixels labeled positive (correctly or incorrectly). This is a useful score for us—if we are using Frangi as a prefiltering for a more robust technique, we would not want to provide any wrong information or seeds to that algorithm. Precision therefore does an okay job of representing that scenario: we are not penalized for what we do not label as true, as long as our reports of true are correct.

Of course, we cannot rely on precision as our sole quantitative factor alone—we could simply return everything negative and receive a perfect score of 100% precision. Therefore we will gauge that measure with that of the MCC [29]. The main advantage of the MCC is that it is well balanced no matter what the size of the two classes are, and will only be high if the approximation scores well against both labels. A score of 1.0 means the approximation is 100% correct, a score of -1.0 means that everything is completely incorrect, and a score of 0 means that the test performs only as well as random guessing.

In our analysis, we will consider both the MCC and precision of a particular segmentation simultaneously. We would like an MCC score as high as possible, but will contextually settle for a lower score as long as the approximation is still *precise*.

One final point about these measures is that we have decided to report their scores only within the placental plate, rather than the entire rectangular image. Since the area outside of plate is masked from consideration, those points will be true negatives no matter what, and we don't want this artificial padding to influence our score. That being said, we do currently concede one part right now: we will also mask an area around the umbilical cord insertion point, as the large amount of noise here will mean that our scores are artificially low. We would like to remove these areas, but for now we will simply not score them.

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 [12] refrained from any explicit analysis of the Frangi score apart from taking the maximum across all scales. Still, we wish to demonstrate some several immediate methods of postprocessing these samples in order to illustrate the usefulness of this optimized Frangi filter.

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.

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.

Nonetheless, we wish to demonstrate the usefulness of the Frangi filter within our image domain towards the segmentation problem, and will therefore, like [2], consider a thresholded \mathcal{V}_{\max} as a crude segmentation.

$$\mathcal{V}_{\Sigma\alpha}(x_0, y_0) = \begin{cases} 1 & \text{if } \mathcal{V}_{\max}(x, y) \geq \alpha \\ 0 & \text{else} \end{cases}, \quad \alpha > 0 \text{ for } \alpha \text{ fixed.} \quad (10.3)$$

If we insist on such a performing such a thresholding, the “correct” choice of α unfortunately seems to depend on the image domain, so user intervention when dealing with the problem domain seems to be the best strategy (referred to in the code as `nz-percentile`). For lack of any better strategy, we could calculate a high percentile of the data and threshold there. Due to the large number of zeros outputted by the filter, we opt instead to take the q th percentile of only nonzero values of V_{Σ} .

A more sincere use of thresholding might be to select a very high threshold, and then use those as seeds for a further technique. We will discuss alternatives methods of aggregating results from our multiscale method, as well as optimal values for parameters and scales. As a final note, we admit that any future extensions of this work (as will be discussed in chapter 12) should not hold too much stock in this thresholded result, and analyzing the raw vesselness score eq. (5.1), or even the un-merged scale-wise scores, would be far more rewarding.

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} .

We briefly demonstrate this in fig. 32 on a particularly well-behaved sample. The top left value image is the original (grayscale image), the top right is V_{\max} , the bottom left is the 95th percentile of nonzero scores of each scale, and the bottom right is the 98th percentile of nonzero scores at each scale.

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 fig. 33 and the result (along with the original sample for comparison) is shown in fig. 34. In fig. 33, the first column is

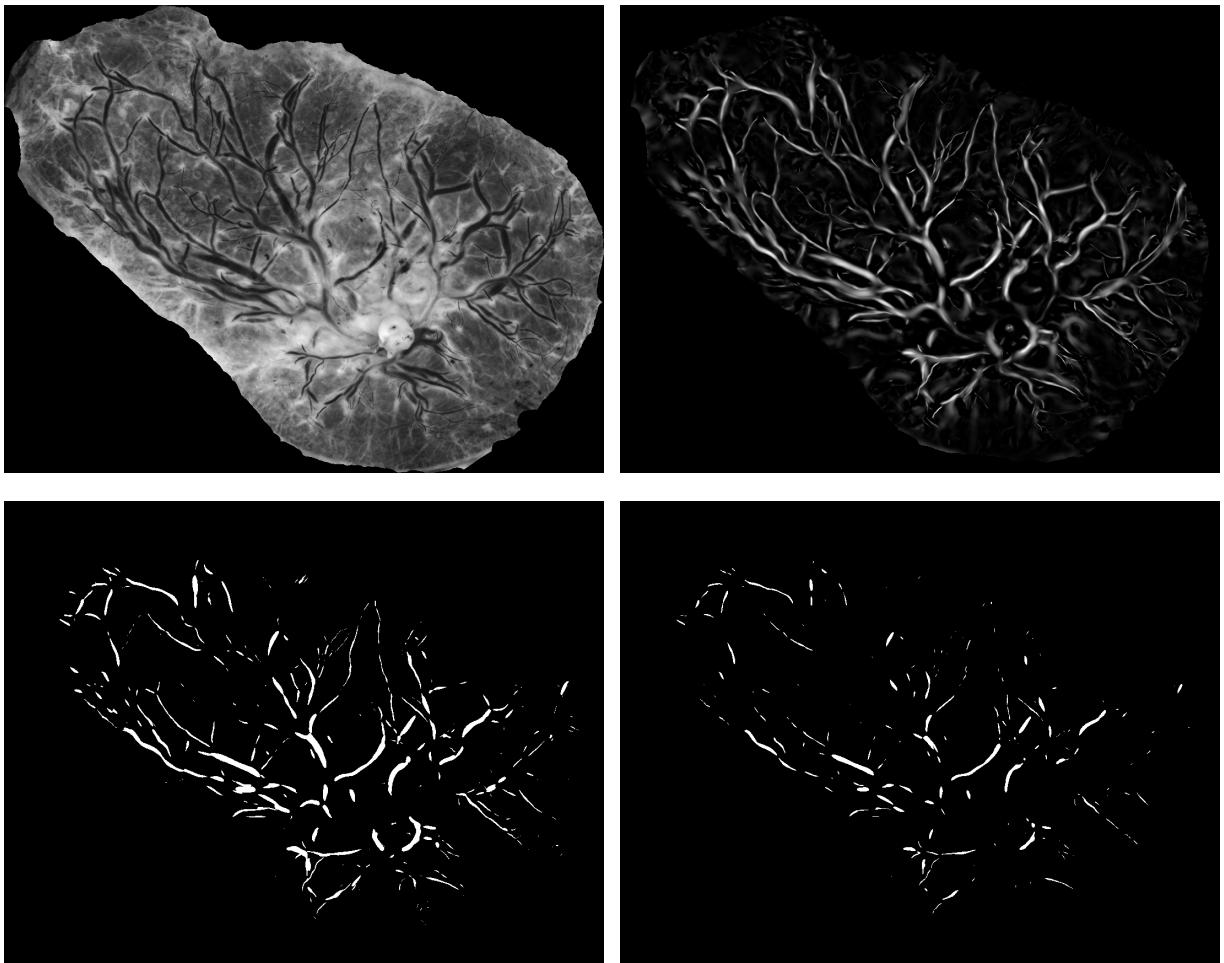


FIGURE 32: Nonzero-percentile thresholding of \mathcal{V}_{\max}

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. 34 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 9, 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 parameter for the random walker algorithm (unfortunately 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.

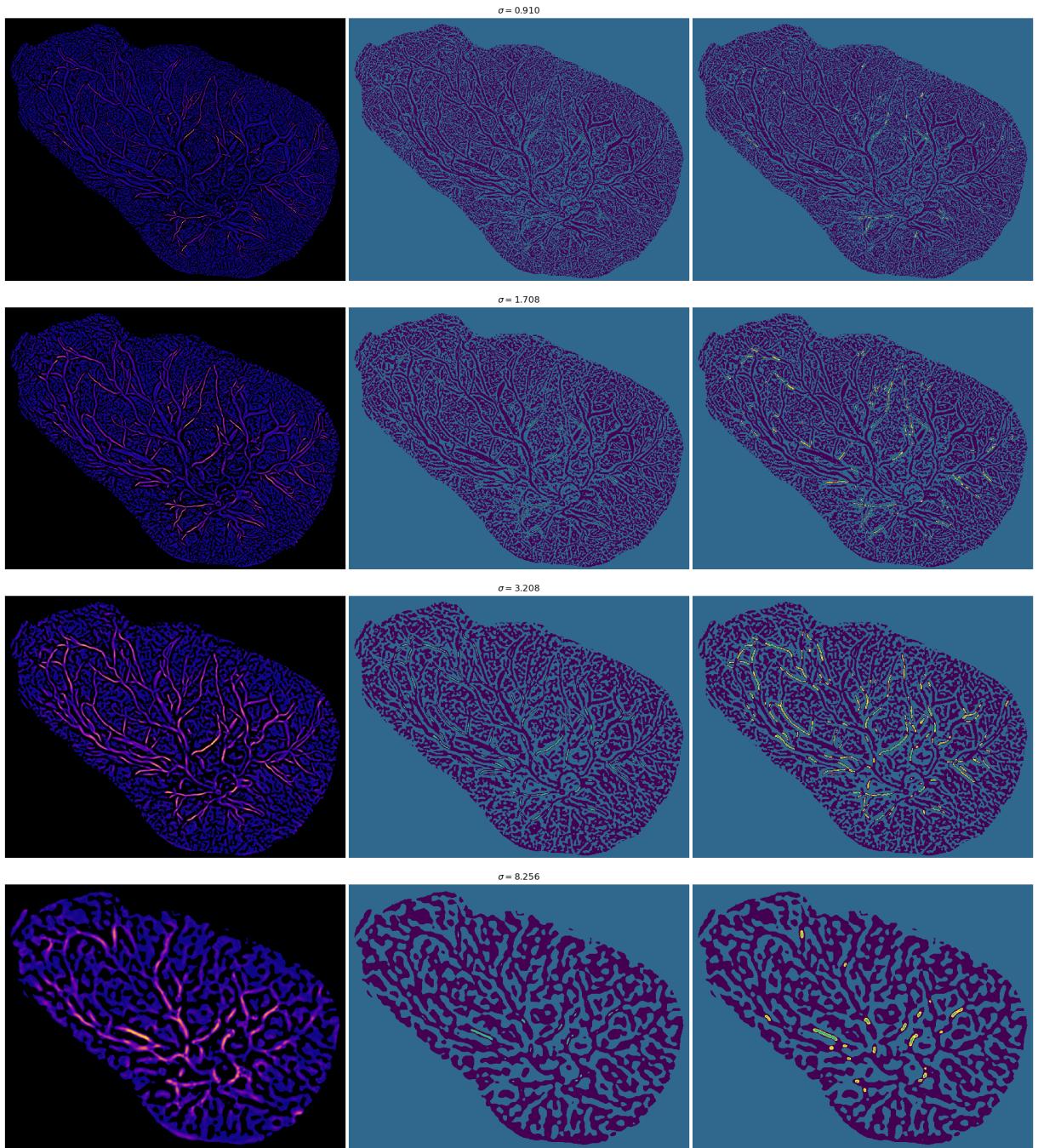


FIGURE 33: Scale-wise random walker segmentation (select scales)

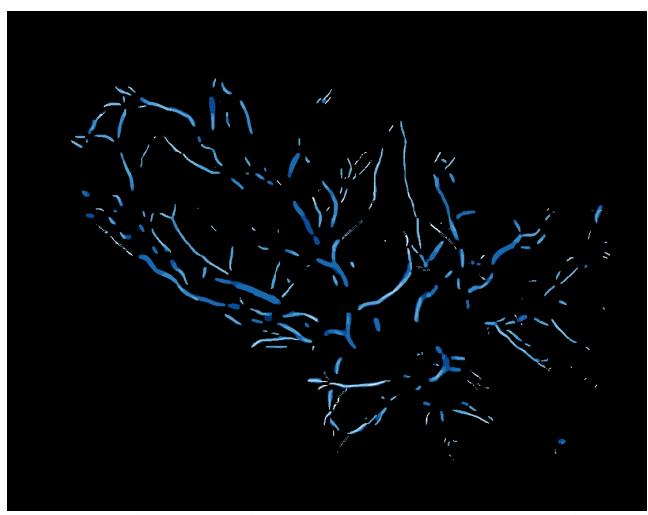


FIGURE 34: Random walker segmentation (result and sample)

Method E: Trough Filling

As we discussed in section 5.1, we can simultaneously calculate the Frangi filter for light and dark curvilinear structures without any added computation time. We shown the signed result of the Frangi filter at different scales two examples in fig. 35 and fig. 36, we can simultaneously calculate for a dark background and a light background. Since the Frangi filter normally throws away any response where $\lambda_2 < 0$ (if dark curvilinear features are targeted) or $\lambda_2 > 0$ (if light curvilinear features are targeted), we lose no computation time at all by simply keeping both, although we must store more results. After computing the multiscale result, we can easily separate these into a positive and a negative strain, which we will denote $\mathcal{V}_{\max}^{(+)}$ and $\mathcal{V}_{\max}^{(-)}$. Our $\mathcal{V}_{\max}^{(+)}$ is the same as our \mathcal{V}_{\max} in a conventional Frangi filter, and $\mathcal{V}_{\max}^{(-)}$ is the same result as if we had taken the Frangi filter while only looking for the opposite type (light/dark) curvilinear feature. Plotting \mathcal{V}_{\max} over a scale of $[-1, 1]$ demonstrates an interesting effect. Whereas the Frangi filter generally is not reliable in terms of accurately predicting widths, we *can* get a sense of the width by looking at where there is a relatively strong response of opposite sign.

That is, at the "foot" of every trough on either side, we can see a bordering curvilinear structure of opposite sign. We perform strict Frangi filtering and separate the positive and negative components. We then perform a different threshold for each signed portion of the Frangi response—a strict one (say $\alpha^{(+)} > .4$) for the conventional \mathcal{V}_{\max} , and a much looser one for our opposite signed $\mathcal{V}_{\max}^{(-)} > \alpha^{(-)} = .05$. We also truncate the scales we consider for calculating $\mathcal{V}_{\max}^{(-)}$, say only the first 12 of 20, since we empirically notice that the bordering curvilinear features are consistently narrower than the vessels themselves.

We then perform a morphological thinning operation on our positive (or standard) Frangi response. We will refer to this thinned approximation as the "spine"

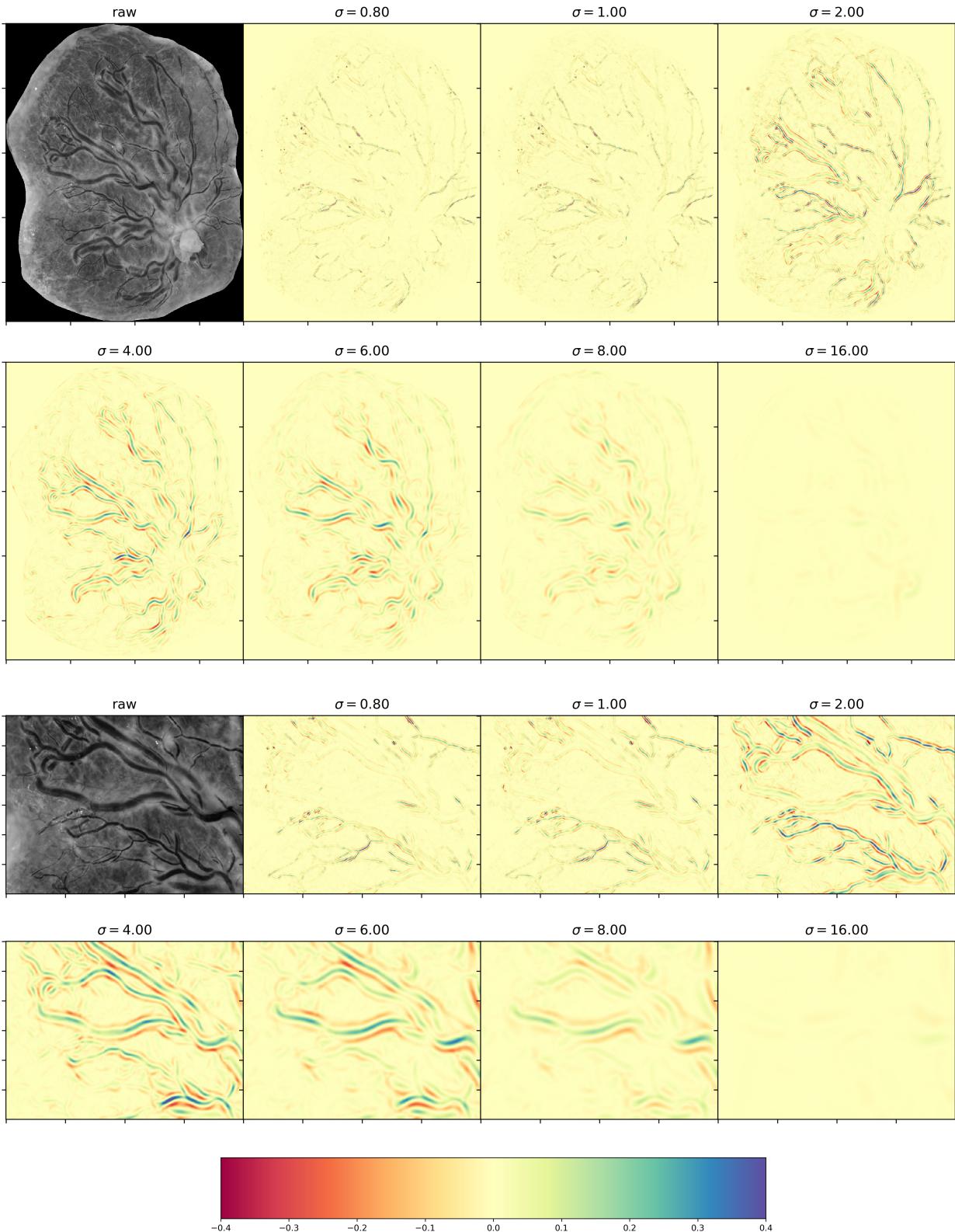


FIGURE 35: Signed Frangi output (plate and inset) (Example 1)

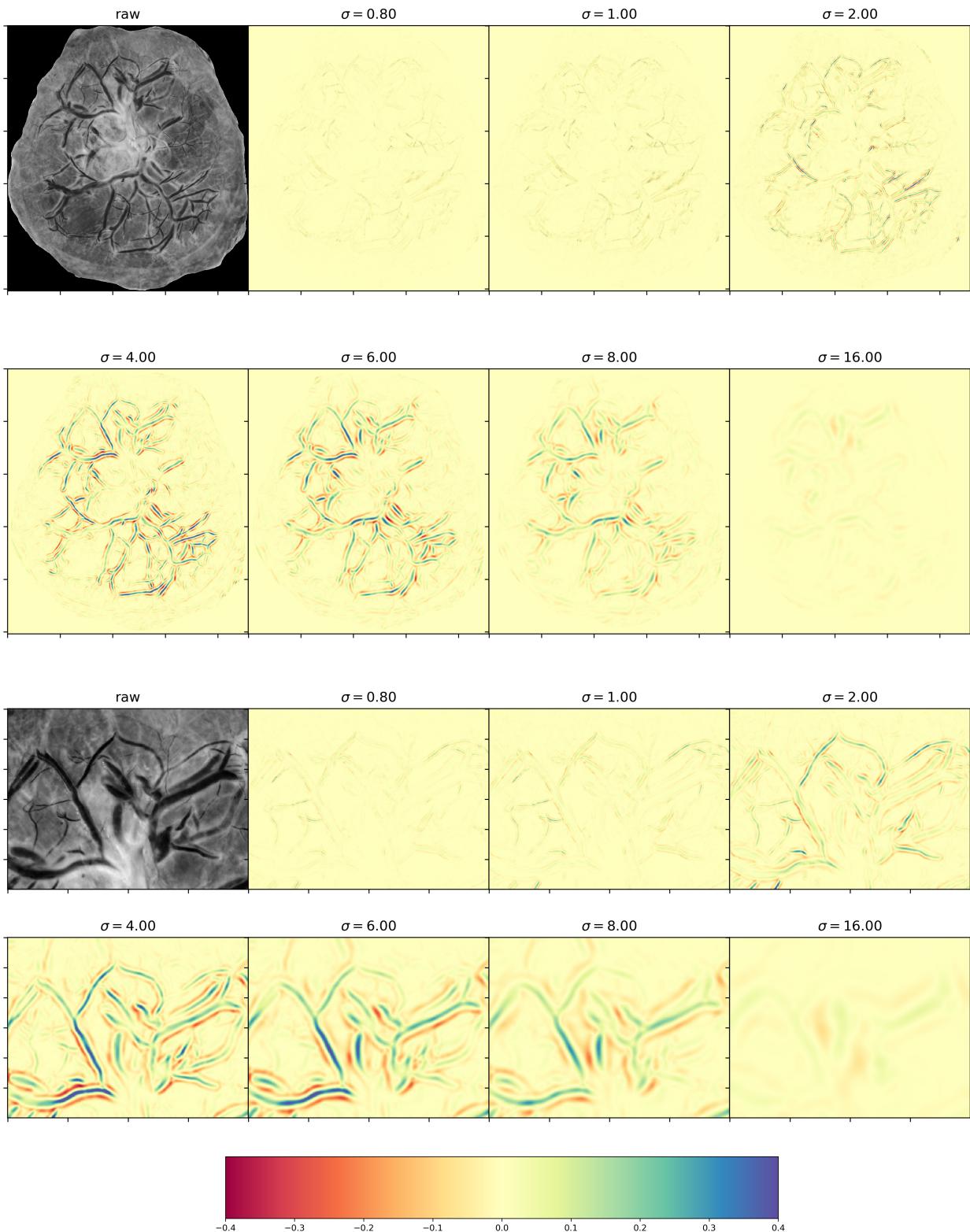


FIGURE 36: Signed Frangi output (plate and inset) (Example 1)

and the (un-thinned) opposite signed component as the “margin.” For each pixel on the thinned spine, we iterate over disks of integer radius and dilate the pixel by a disk of that radius if that disk coincides with a margin point.

This allows us to extend the Frangi output from the point where it’s strongest all the way to the base of the curvilinear structure, providing a much better indication of the true width of the curvilinear structure.

Nonfrangi Segmentation Methods

Of course, there are many over viable methods of segmentation. We refer to [31] for several high-performing (albeit considerably resource intensive) methods, namely involving shearlets, laplacian eigenmaps, and a conditional generative adversarial network.

We will be content in the present paper to simply compare our results to a relatively fast technique that is not based on differential geometry or morphology, but instead that of a global threshold on the grayscale image. Although many viable thresholding algorithms exist, we opt for an intermeans threshold, or the Ridler-Calvert method, as described in [32], which is implemented in `scikit-image` as `filters.threshold_isodata`. This functions uses an iterative process to find the smallest threshold which is midway between the mean intensities of the high pixels and the low pixels. In other words, the optimal α^* satisfies

$$\alpha^* = \arg \min_{\alpha} \left(\frac{1}{2} [\text{mean}\{\mathbf{I}(x, y) \mid \mathbf{I}(x, y) \leq \alpha\} + \text{mean}\{\mathbf{I}(x, y) \mid \mathbf{I}(x, y) > \alpha\}] \right) \quad (10.4)$$

Since the vascular structure in our image domain is darker than the background, we select pixels where $\mathbf{I}(x, y) < \alpha^*$.

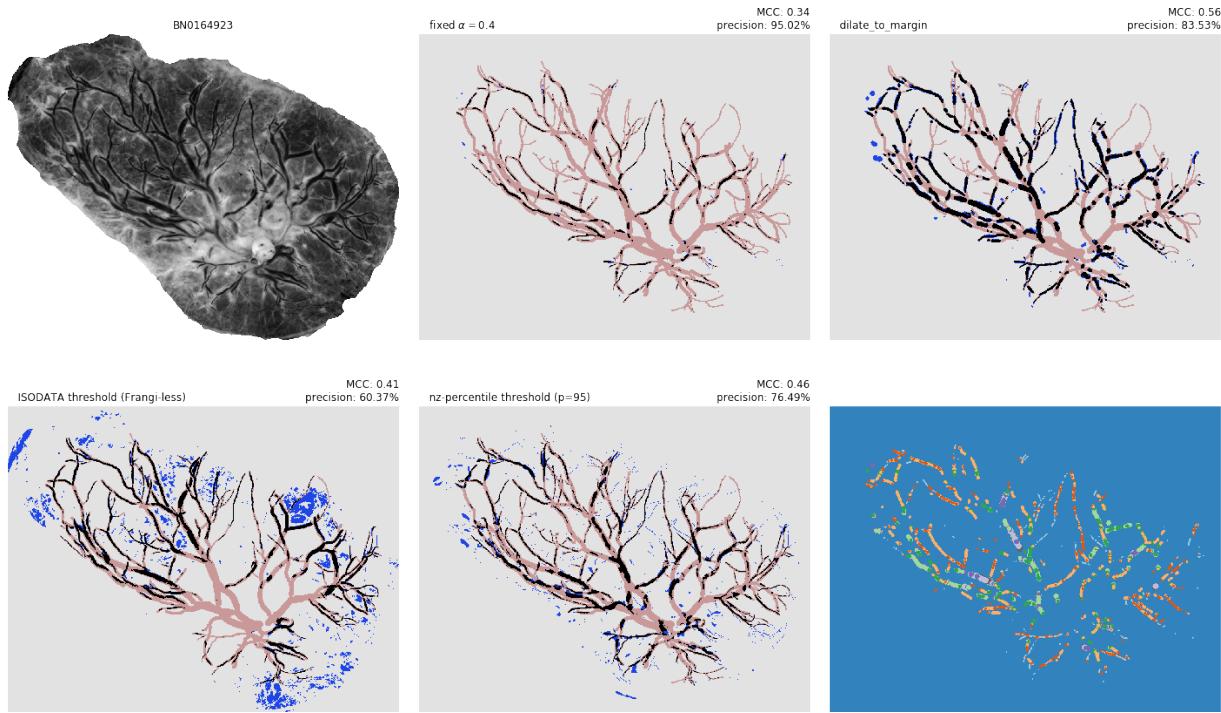


FIGURE 37: Confusion matrices of various segmentation methods (Example 1)

Segmentation Results

We first look at the results of our various segmentation procedures for a few well-behaved samples.

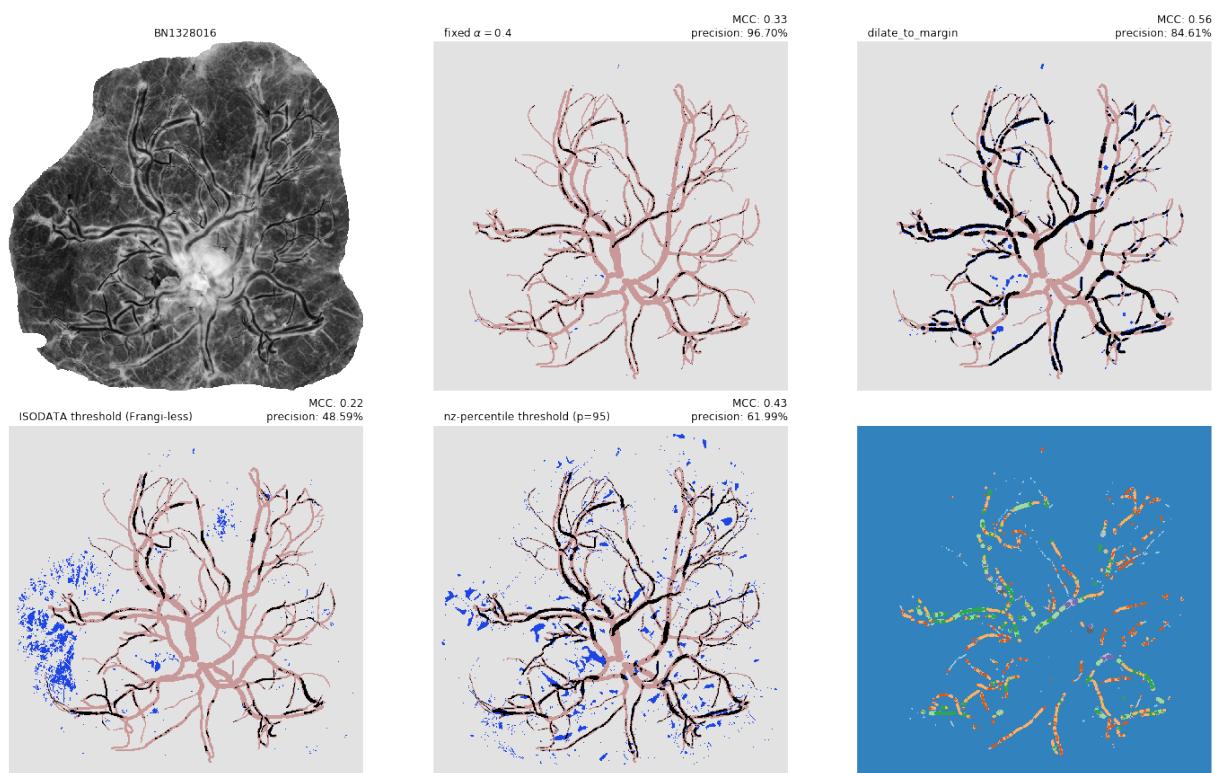


FIGURE 38: Confusion matrices of various segmentation methods (Example 2)

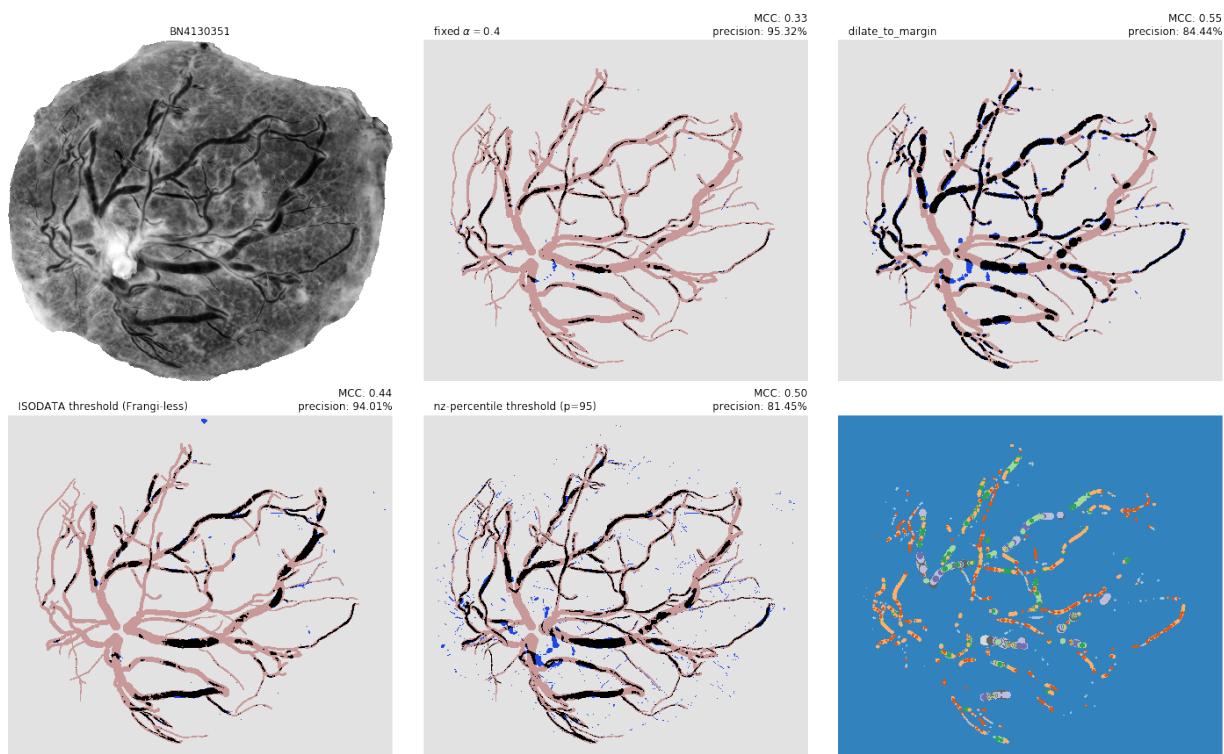


FIGURE 39: Confusion matrices of various segmentation methods (Example 3)

CHAPTER 11

PRIMITIVE NETWORK COMPLETION

In this chapter we discuss the idea of network completion and demonstrate a relatively brute-force method of using a segmentation result to solve some of the network completion problem.

One of the biggest shortcomings of our demonstrated segmentation methods is that they frequently result in gaps in parts of the network. We demonstrate a way to rectify this problem in certain situations, and then discuss how to extend these arguments to fill larger, more uncertain gaps.

First, once we've produced a segmentation, we thin it down with [33] and look for endpoints of an otherwise connected point. Using a 3×3 structuring element, we iterate over each pixel and identify how many local neighbors it has. If a pixel has zero or one local neighbors, we identify it as an endpoint of the partial network. After identifying these endpoints, we assign each a label (i, j) depending on where the neighboring pixel is located, as according to fig. 40. We deem two endpoints potentially connectible only if they're not connected on the same side. That is, their labels (i, j) and (i', j') must have $i \neq i'$ or $j \neq j'$ (unless i or j is 1). For example, if an endpoint is connected to the partial network on its top side, any endpoint that connects to it cannot also be connected to a network on its top. If a pixel has no connections at all, with label $(1, 1)$, we do not restrict its connections at all. To save time (though we don't anticipate it will affect the result much), we also restrict pairs from being more than a set distance away (in this case, 100 pixels in Euclidean length).

After we limit the connections, we consider each pair of endpoints and draw a

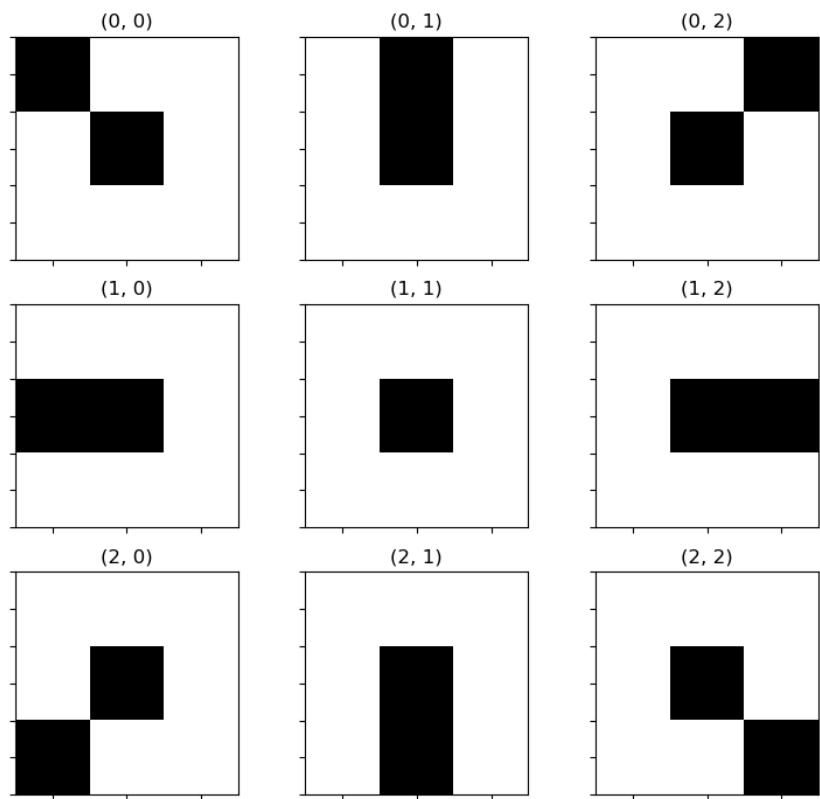


FIGURE 40: Endpoints labels based on adjacent neighbor location

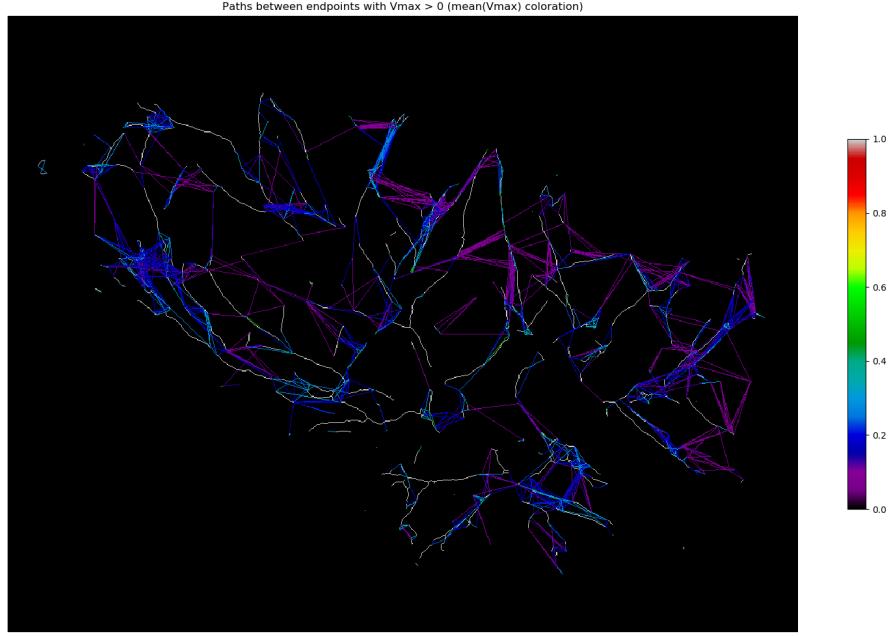


FIGURE 41: All lines between endpoints with nonzero \mathcal{V}_{\max}

straight line segment between the two. If that passes through a point where \mathcal{V}_{\max} is 0, we disallow that pair as well. We also disallow any line which crosses any part of the network which is known to exist. Finally, from the list of all remaining pairs of endpoints, we simply select the path along which the maximum mean value of \mathcal{V}_{\max} is achieved. fig. 41 shows non-violating paths between end-point pairs, and ?? shows a partially completed network (in yellow) overlaid on \mathcal{V}_{\max} .

We show the complete process for comparison on two different samples.

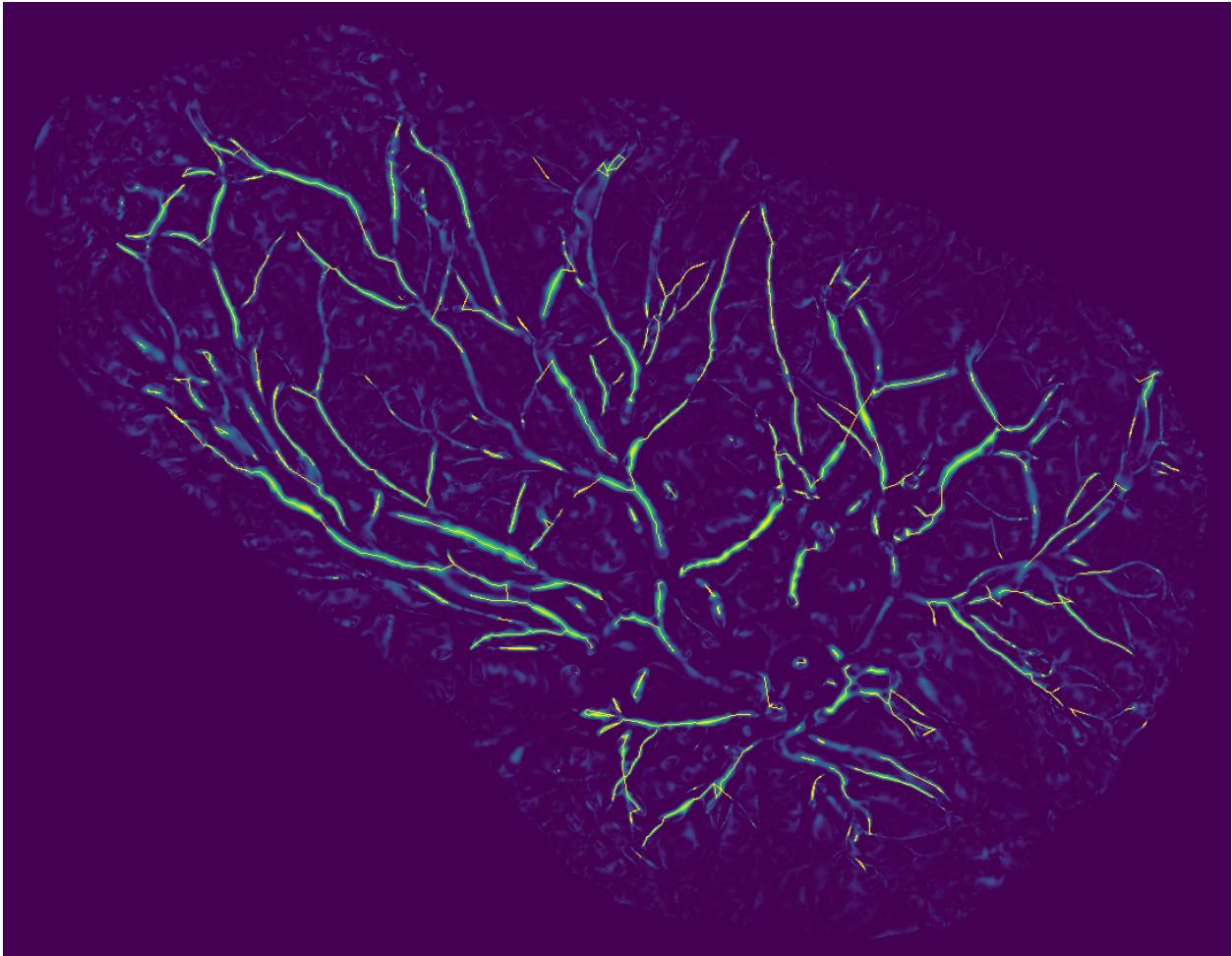


FIGURE 42: Partially completed network

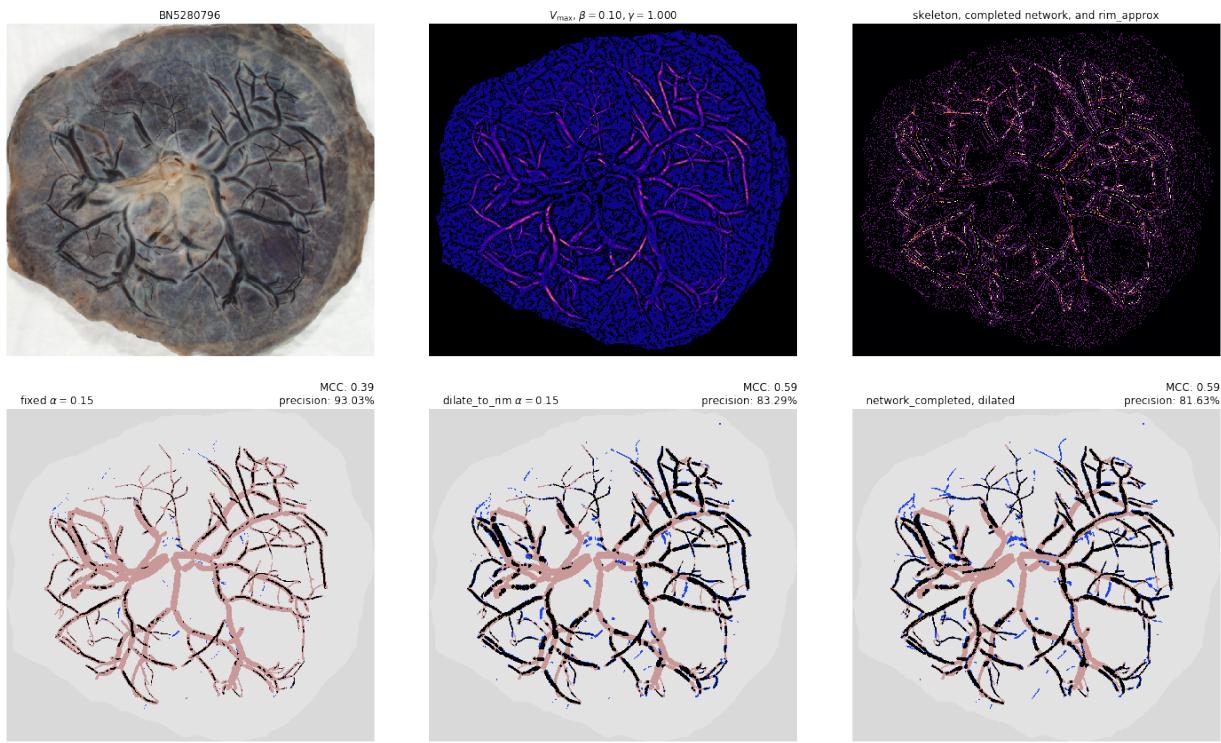


FIGURE 43: Trough Dilation and Network Completion (Example 1)

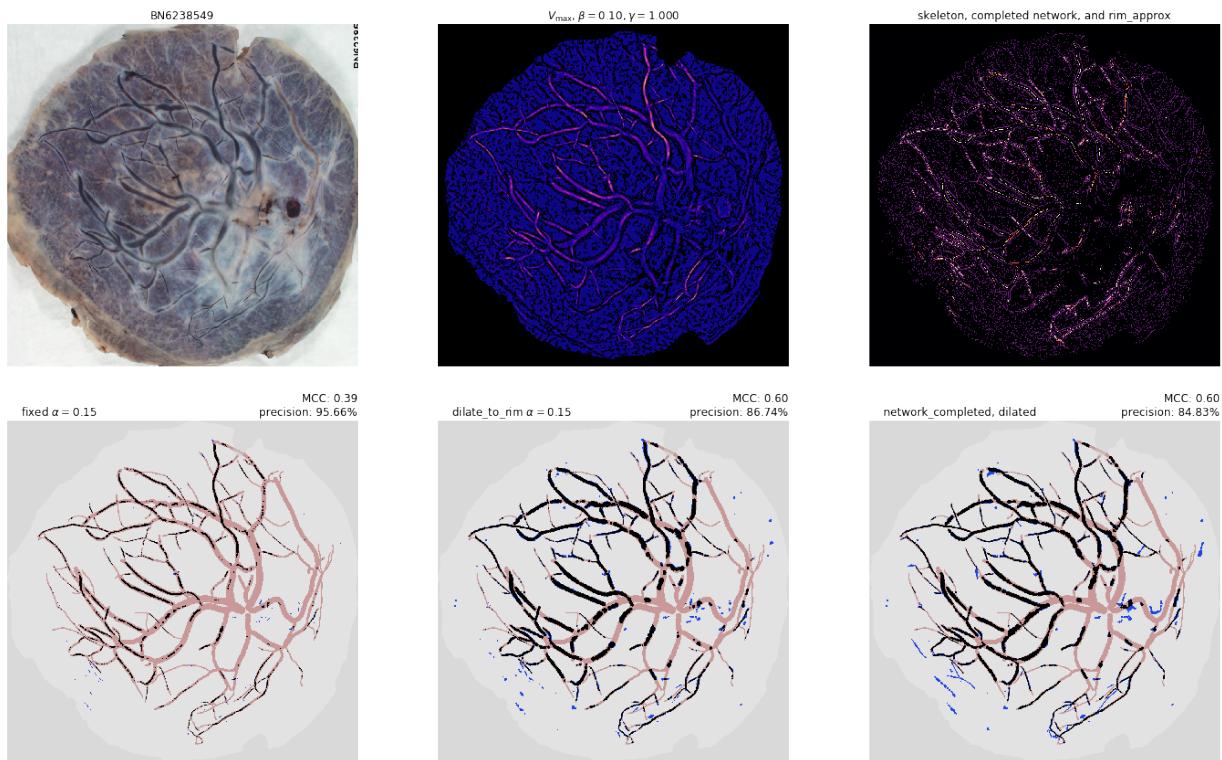


FIGURE 44: Trough Dilation and Network Completion (Example 2)

CHAPTER 12

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.

- Perfect stump removal and watershedding to not rely on use of maxes/improve vesselness measure.
- Try something like [34] 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} .
- Use this as preprocessing for a Neural Network. (cite kara's work, katalinas work)
- Apply to more image domains (STARE, other placental domains).

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/compare_parameters_alt.py

```
1 #!/usr/bin/env python3
2
3 import numpy as np
4 import matplotlib.pyplot as plt
5 import matplotlib as mpl
6 from skimage.util import img_as_float
7 from skimage.io import imread
8 from placenta import (get_named_placenta, list_by_quality, cropped_args,
9                         img_as_float)
10
11 from frangi import frangi_from_image
12 import numpy.ma as ma
13 from hfft import fft_gradient, fft_hessian, fft_gaussian
14 from merging import nz_percentile
15 from plate_morphology import dilate_boundary
16 import os.path, os
17 from skimage.morphology import thin
18 from postprocessing import dilate_to_rim
19 from scoring import confusion, mcc, integrate_score
20 from placenta import open_tracefile, open_typefile
21 from preprocessing import inpaint_hybrid
22 from placenta import measure_ncs_markings, add_ucip_to_mask
23
24
25
26
27 STRICT = {'label': 'strict', 'beta': 0.10, 'gamma': 1.0}
28 SEMISTRRICT_BETA = {'label': 'semistrict-beta', 'beta': 0.10, 'gamma': 0.5}
29 SEMISTRRICT_GAMMA = {'label': 'semistrict-gamma', 'beta': 0.5, 'gamma': 1.0}
30
31 STANDARD = {'label': 'standard', 'beta': 0.5, 'gamma': 0.5}
32
33 SEMILOOSE_BETA = {'label': 'semiloose-beta', 'beta': 1.0, 'gamma': 0.5}
34 SEMILOOSE_GAMMA = {'label': 'semiloose-gamma', 'beta': 0.5, 'gamma': 0.30}
35 LOOSE = {'label': 'loose', 'beta': 1.0, 'gamma': 0.30}
36 STRUCTURENESS = {'label': 'Structureness Factor', 'beta': np.inf, 'gamma': 0.5}
37 ANISOTROPY = {'label': 'Anisotropy Factor', 'beta': 0.10, 'gamma': 0}
38 cm = mpl.cm.plasma
39 #cmscales = mpl.cm.magma
40 cm.set_bad('k', 1) # masked areas are black, not white
41 #cmscales.set_bad('w', 1)
42
43 scales = np.logspace(-1.5, 3.5, num=12, base=2)
44
45 integral_scores = list()
46
47 for filename in list_by_quality():
48
49     cimg = open_typefile(filename, 'raw')
50     ctrace = open_typefile(filename, 'ctrace')
51     trace = open_tracefile(filename)
52     img = get_named_placenta(filename)
53     crop = cropped_args(img)
54     ucip = open_typefile(filename, 'ucip')
55     img = inpaint_hybrid(img)
56
57     # make the size of figures more consistent
58     if img[crop].shape[0] > img[crop].shape[1]:
59         # and rotating it would be fix all this automatically
60         cimg = np.rot90(cimg)
61         ctrace = np.rot90(ctrace)
62         trace = np.rot90(trace)
```

```

63     img = np.rot90(img)
64     ucip = np.rot90(ucip)
65     crop = cropped_args(img)
66
67     ucip_midpoint, resolution = measure_ncs_markings(ucip)
68     ucip_mask = add_ucip_to_mask(ucip_midpoint, radius=60, mask=img.mask)
69
70     name_stub = filename.rstrip('.png').strip('T-')
71
72
73     F_demos = list()
74     integrals = list()
75     PARAMS = [STANDARD, LOOSE, STRICT,
76                ANISOTROPY, SEMILOOSE_BETA, SEMISTRRICT_BETA,
77                STRUCTURENESS, SEMILOOSE_GAMMA, SEMISTRRICT_GAMMA]
78
79     for params in PARAMS:
80         print(f"running {params['label']} Frangi on {name_stub}")
81
82         if params['gamma'] == 0:
83             rescale = False
84         else:
85             rescale = True
86
87         F_demo = np.stack([frangi_from_image(img, sigma, beta=params['beta'],
88                                gamma=params['gamma'], dark_bg=False,
89                                dilation_radius=20,
90                                rescale_frangi=rescale)
91                                for sigma in scales])
92
93         F_max = F_demo.max(axis=0)
94
95         integral = integrate_score(F_max, trace, mask=img.mask)
96
97         F_demos.append(F_max)
98         integrals.append(integral)
99
100        del F_demo
101
102
103    fig, ax = plt.subplots(nrows=3, ncols=3, figsize=(20,20))
104    A = ax.ravel()
105
106    #A[0].imshow(cimg[crop])
107    #A[0].set_title(name_stub)
108
109    #A[1].imshow(ctrace[crop])
110    #A[1].set_title('ground truth')
111
112    for i, (Fmax, integral, params) in enumerate(zip(F_demos, integrals,
113                                                    PARAMS)):
114        beta = params['beta']
115        gamma = params['gamma']
116        label = params['label']
117        A[i].imshow(ma.masked_where(Fmax==0,Fmax)[crop], cmap=cm, vmin=0, vmax=1)
118        A[i].set_title(rf' Vmax ({label})' + '\n' +
119                      rf' β = beta:.2f, γ = gamma:.2f', loc='left')
120        A[i].set_title(rf' CVR: {integral:.3f}', loc='right')
121
122    [a.axis('off') for a in A]
123    fig.tight_layout()
124    fig.subplots_adjust(wspace=0.1, hspace=0.1)
125    plt.show()
126

```

127 integral_scores.append((name_stub, integrals))

listings/compare_parameters.py

```
1 #!/usr/bin/env python3
2
3 import numpy as np
4 import matplotlib.pyplot as plt
5 import matplotlib as mpl
6 from skimage.util import img_as_float
7 from skimage.io import imread
8 from placenta import (get_named_placenta, list_by_quality, cropped_args,
9                         img_as_float)
10
11 from frangi import frangi_from_image
12 import numpy.ma as ma
13 from hfft import fft_gradient, fft_hessian, fft_gaussian
14 from merging import nz_percentile
15 from plate_morphology import dilate_boundary
16 import os.path, os
17 from skimage.morphology import thin
18 from postprocessing import dilate_to_rim
19 from scoring import confusion, mcc, integrate_score
20 from placenta import open_tracefile, open_typefile
21 from preprocessing import inpaint_hybrid
22 from placenta import measure_ncs_markings, add_ucip_to_mask
23
24
25 STRICT = {'label': 'strict', 'beta': 0.10, 'gamma': 1.0, 'alpha': .15 }
26
27 STANDARD = {'label': 'standard', 'beta': 0.5, 'gamma': 0.5, 'alpha': .4}
28
29 SEMISTRRICT = {'label': 'semistrict', 'beta': 0.35, 'gamma': 0.5, 'alpha': .4}
30
31 LOOSE = {'label': 'loose', 'beta': 1.0, 'gamma': 0.5, 'alpha': .8}
32
33 cm = mpl.cm.plasma
34 #cmscales = mpl.cm.magma
35 cm.set_bad('k', 1) # masked areas are black, not white
36 #cmscales.set_bad('w', 1)
37
38 scales = np.logspace(-1.5, 3.5, num=12, base=2)
39
40 integral_scores = list()
41
42 for filename in list_by_quality():
43
44     cimg = open_typefile(filename, 'raw')
45     ctrace = open_typefile(filename, 'ctrace')
46     trace = open_tracefile(filename)
47     img = get_named_placenta(filename)
48     crop = cropped_args(img)
49     ucip = open_typefile(filename, 'ucip')
50     img = inpaint_hybrid(img)
51
52     # make the size of figures more consistent
53     if img[crop].shape[0] > img[crop].shape[1]:
54         # and rotating it would be fix all this automatically
55         cimg = np.rot90(cimg)
56         ctrace = np.rot90(ctrace)
57         trace = np.rot90(trace)
58         img = np.rot90(img)
```

```

59         ucip = np.rot90(ucip)
60         crop = cropped_args(img)
61
62         ucip_midpoint, resolution = measure_ncs_markings(ucip)
63         ucip_mask = add_ucip_to_mask(ucip_midpoint, radius=60, mask=img.mask)
64
65         name_stub = filename.rstrip('.png').strip('T-')
66
67
68         F_demos = list()
69         integrals = list()
70         PARAMS = [LOOSE, SEMISTRRICT, STANDARD, STRICT]
71
72         for params in PARAMS:
73             print(f"running {params['label']} Frangi on {name_stub}")
74             F_demo = np.stack([frangi_from_image(img, sigma, beta=params['beta'],
75                                         gamma=params['gamma'], dark_bg=False,
76                                         dilation_radius=20, rescale_frangi=True)
77                                         for sigma in scales])
78
79             F_max = F_demo.max(axis=0)
80
81             integral = integrate_score(F_max, trace, mask=img.mask)
82
83             F_demos.append(F_max)
84             integrals.append(integral)
85
86             del F_demo
87
88
89         fig, ax = plt.subplots(nrows=2, ncols=3, figsize=(20,12))
90         A = ax.T.ravel()
91
92         A[0].imshow(cimg[crop])
93         A[0].set_title(name_stub)
94
95         A[1].imshow(ctrace[crop])
96         A[1].set_title('ground truth')
97
98         for i, (Fmax, integral, params) in enumerate(zip(F_demos, integrals, PARAMS), 2):
99             beta = params['beta']
100            gamma = params['gamma']
101            label = params['label']
102            A[i].imshow(ma.masked_where(Fmax==0,Fmax)[crop], cmap=cm, vmin=0, vmax=1)
103            A[i].set_title(rf' Vmax ({label})'+'\n'+
104                           rf' β={beta:.2f}, γ={gamma:.3f}', loc='left')
105            A[i].set_title(rf'CVR: {integral:.3f}', loc='right')
106
107        [a.axis('off') for a in A]
108        fig.tight_layout()
109        plt.show()
110
111        integral_scores.append((name_stub, integrals))

```

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

```

```

10
11 def principal_curvatures(img, sigma=1.0, H=None):
12     """Calculate the approximated principal curvatures of an image
13
14     Return the (approximated) principal curvatures {κ1, κ2} of an image,
15     that is, the eigenvalues of the Hessian at each point (x,y). The output
16     is arranged such that |κ1| <= |κ2|. Note that the Hessian of the image,
17     if not provided, is computed using skimage.feature.hessian_matrix, which
18     can be very slow for large sigmas.
19
20 Parameters
21 -----
22 img: array or ma.MaskedArray
23
24     An ndarray representing a 2D or multichannel image. If the image is
25     multichannel (e.g. RGB), then each channel will be processed
26     individually. Additionally, the input image may be a masked array-- in
27     which case the output will preserve this mask identically.
28
29 sigma: float, optional
30     Standard deviation of the Gaussian (used to calculate the hessian
31     matrix).
32 H: list of array, optional
33     The hessian itself (Hxx,Hxy,Hyy) whose eigenvalues will be calculated.
34     Use this option if you're going to calculate the Hessian using faster
35     means, e.g. via FFT.
36
37 Returns
38 -----
39 (K1, K2): tuple of arrays
40     K1, K2 each are the exact dimension of the input image, ordered in
41     magnitude such that |κ1| <= |κ2| in all locations.
42
43 Examples
44 -----
45 >>> K1, K2 = principal_curvatures(img)
46 >>> K1.shape == img.shape
47 True
48 >>> (K1 <= K2).all()
49 True
50 >>> K1.mask == img.mask
51 True
52 """
53
54 # determine if multichannel
55 multichannel = (img.ndim == 3)
56
57 if not multichannel:
58     # add a trivial dimension
59     img = img[:, :, np.newaxis]
60
61 K1 = np.zeros_like(img, dtype='float64')
62 K2 = np.zeros_like(img, dtype='float64')
63
64 for ic in range(img.shape[2]):
65
66     channel = img[:, :, ic]
67
68     # returns the tuple (Hxx, Hxy, Hyy)
69     if H is None:
70         H = hessian_matrix(channel, sigma=sigma)
71
72     # returns tuple (l1,l2) where l1 >= l2 but this *includes sign*
73     L = hessian_matrix_eigvals(H)

```

```

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

```

```

138 >>>(np.abs(K1) <= np.abs(K2)).all()
139 False
140 >>>K1r, K2r = reorder_eigs(K1,K2)
141 >>>(K1r <= K2r).all()
142 False
143 >>>(np.abs(K1r) <= np.abs(K2r)).all()
144 True
145
146 TODO
147 -----
148 Support out= keyword
149 """
150 # this will do nothing if L is already an array but will make it an array
151 # if it's a tuple/list/iterable
152 L = np.stack(L)
153 mag = np.argsort(np.abs(L), axis=0)
154
155 # now L2 is larger in absolute value, as consistent with Frangi paper
156 return np.take_along_axis(L, mag, axis=0)
157
158
159 def principal_directions(img, sigma, H=None, mask=None):
160     """Calculate principal directions of
161     will ignore calculation of principal directions of masked areas
162
163     mask should be positive where the PD's should *NOT* be calculated
164     this function actually returns the theta corresponding to
165     leading and trailing principal directions, i.e. angle w / x axis
166     """
167
168     if H is None:
169         H = fft_hessian(img, sigma)
170
171     Hxx, Hxy, Hyy = H
172
173
174     # determine if there was a supplied mask or use images if it exists
175     if mask is None:
176         try:
177             mask = img.mask
178         except AttributeError:
179             masked = False
180         else:
181             masked = True
182     else:
183         masked = True
184
185     dims = img.shape
186
187     # where to store
188     trailing_thetas = np.zeros_like(img, dtype='float64')
189     leading_thetas = np.zeros_like(img, dtype='float64')
190
191     # maybe implement a small angle correction
192     for i, (xx, xy, yy) in enumerate(np.nditer([Hxx, Hxy, Hyy])):
193
194         # grab the (x,y) coordinate of the hxx, hxy, hyy you're using
195         subs = np.unravel_index(i, dims)
196
197         # ignore masked areas (if masked array)
198         if masked and mask[sub]:
199             continue
200
201         h = np.array([[xx, xy], [xy, yy]]) # per-pixel hessian

```

```

202     l, v = eig(h) # eigenvectors as columns
203
204     # reorder eigenvectors by (increasing) magnitude of eigenvalues
205     v = v[:, np.argsort(np.abs(l))]
206
207     # angle between each eigenvector and positive x-axis
208     # arccos of first element (dot product with (1,0) and eigvec is already
209     # normalized)
210     trailing_thetas[subs] = np.arccos(v[0, 0]) # first component of each
211     leading_thetas[subs] = np.arccos(v[0, 1]) # first component of each
212
213 if masked:
214     leading_thetas = ma.masked_array(leading_thetas, mask)
215     trailing_thetas = ma.masked_array(trailing_thetas, mask)
216
217 return trailing_thetas, leading_thetas
218
219
220 if __name__ == "__main__":
221     pass
222
223
224 #from get_base import get_preprocessed
225 #import matplotlib.pyplot as plt
226 #from functools import partial
227 #from fpd import get_targets
228 #b = partial(plt.imshow, cmap=plt.cm.Blues)
229 #sp = partial(plt.imshow, cmap=plt.cm.spectral)
230 #s = plt.show
231
232 #import time
233
234 #img = get_preprocessed(mode='G')
235
236 #for sigma in [0.5, 1, 2, 3, 5, 10]:
237
238 #    print('-'*80)
239 #    print('σ=', sigma)
240 #    print('calculating hessian H')
241
242 #    tic = time.time()
243 #    H = hessian_matrix(img, sigma=sigma)
244
245 #    toc = time.time()
246 #    print('time elapsed: ', toc - tic)
247 #    tic = time.time()
248 #    print('calculating hessian via FFT (F)')
249 #    h = fft_hessian(img, sigma)
250
251 #    toc = time.time()
252 #    print('time elapsed: ', toc - tic)
253 #    tic = time.time()
254 #    print('calculating principal curvatures for σ={}'.format(sigma))
255 #    K1,K2 = principal_curvatures(img, sigma=sigma, H=H)
256 #    toc = time.time()
257 #    print('time elapsed: ', toc - tic)
258 #    tic = time.time()
259 #    print('calculating principal curvatures for σ={} (fast)'.format(sigma))
260 #    k1,k2 = principal_curvatures(img, sigma=sigma, H=h)
261
262 #    toc = time.time()
263 #    print('time elapsed: ', toc - tic)
264 #    tic = time.time()

```

```

266     # #####
267
268     # print('calculating targets for σ={}'.format(sigma))
269     # T = get_targets(K1,K2, threshold=False)
270
271     # toc = time.time()
272     # print('time elapsed: ', toc - tic)
273     # tic = time.time()
274
275     # print('calculating targets for σ={} (fast)'.format(sigma))
276     # t = get_targets(k1,k2, threshold=False)
277
278     # toc = time.time()
279     # print('time elapsed: ', toc - tic)
280
281     # #####
282
283     # print('extending masks')
284
285     # extend mask over nontargets items
286     # img1 = ma.masked_where( T < T.mean(), img)
287     # img2 = ma.masked_where( t < t.mean(), img)
288
289     # tic = time.time()
290     # print('calculating principal directions for σ={}'.format(sigma))
291     # T1,T2 = principal_directions(img1, sigma=sigma, H=H)
292     # toc = time.time()
293     # print('time elapsed: ', toc - tic)
294     # tic = time.time()
295
296     # print('calculating principal directions for σ={} (fast)'.format(sigma))
297     # t1,t2 = principal_directions(img2, sigma=sigma, H=h)
298     # toc = time.time()
299     # print('time elapsed: ', toc - tic)

```

listings/extract_NCS_pcsvn.py

```

1 #!/usr/bin/env python3
2 """
3 This is the main program. It approximates the PCCSVN of a list of samples.
4 It does not do network completion.
5 """
6
7
8
9 from placenta import (get_named_placenta, cropped_args, cropped_view,
10                      list_placentas, list_by_quality, open_typefile,
11                      open_tracefile, add_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 # placentas = list_placentas('T-BN')
46 # load placentas from a certain quality category 0=good, 1=okay, 2=fair, 3=poor
47
48 #placentas = list_by_quality(2)
49 #placentas.extend(list_by_quality(3))
50
51 placentas = list_by_quality(0, N=1)
52 # load from a file (sample names are keys of the json file)
53 # placentas = list_by_quality(json_file='manual_batch.json')
54
55 # for a single named sample, use a 1 element list.
56 # placentas = ['T-BN0204423.png']
57
58 #placentas = ['barium1.png',]
59 # RUNTIME OPTIONS -----
60 # Where to save and whether or not to use old targets.
61
62 MAKE_NPZ_FILES = False # pickle frangi targets if you can
63 USE_NPZ_FILES = False # use old npz files if you can
64 NPZ_DIR = 'output/181204-test' # where to look for npz files
65 OUTPUT_DIR = 'output/181204-test' # where to save outputs
66
67 # add in a meta switch for verbosity (or levels)
68 #VERBOSE = False
69
70 # FRANGI / EXTRACT_PCSVN OPTIONS -----
71
72 # Find bright curvilinear structure against a dark background -> True
73 # Find dark curvilinear structure against a bright background -> False
74 # DARK_BG -> ignore and return signed Frangi scores
75 DARK_BG = False
76
77 # Along with the above, this will return "opposite" signed frangi scores.
78 # if this is True, then DARK_BG controls the "polarity" of the filter.
79 # See frangi.get_frangi_targets for details.
80 SIGNED_FRANGI = True
81
82 # Do not calculate hessian scores close to the boundary (this is important
83 # mainly in terms of ensuring that the hessian is very large on the edge of
84 # the plate (which would influence gamma calculation)
85 DILATE_PER_SCALE = True
86
87 # Attempt to remove glare from sample (some are OK, some are bad)
88 FLATTEN_MODE = 'L' # 'G' or 'L'
89 REMOVE_GLARE = True
90

```

```

91 # Which scales to use
92 SCALE_RANGE = (-1.5, 3.2); SCALE_TYPE = 'logarithmic'
93 #SCALE_RANGE = (.2, 12); SCALE_TYPE = 'linear'
94 N_SCALES = 20
95
96 # use this if you want to use a custom argument (comment out the above)
97 SCALES = None
98 #SCALE_RANGE = None, SCALE_TYPE == 'custom'
99
100
101 # Explicit Frangi Parameters (pass a scalar, array as long as scales)
102 BETAS = 0.35
103 GAMMAS = 0.5
104 CS = None # pass scalar, array, or None
105 ALPHAS = None # set custom alphas or calculate later
106 FIXED_ALPHA = .4
107
108 RESCALE_FRANGI = True
109 GRADIENT_FILTER = False
110
111
112 # Scoring Decisions (don't need to touch these)
113 UCIP_RADIUS = 60 # area around the umbilical cord insertion point to ignore
114
115
116
117
118 # CODE BEGINS HERE -----
119
120 if SCALES is None:
121     if SCALE_TYPE == 'linear':
122         scales = np.linspace(*SCALE_RANGE, num=N_SCALES)
123     elif SCALE_TYPE == 'logarithmic':
124         scales = np.logspace(*SCALE_RANGE, num=N_SCALES, base=2)
125 else:
126     scales = SCALES
127     SCALE_TYPE = 'custom' # this and the next three lines are just for logging
128     N_SCALES = len(SCALES)
129     SCALES = (min(SCALES), max(SCALES))
130
131 mccs = dict() # empty dict to store MCC's of each sample
132 pncs = dict() # empty dict to store percent network covered for each sample
133 precisions = dict()
134
135 n_samples = len(placentas)
136
137 if not os.path.exists(OUTPUT_DIR):
138     os.makedirs(OUTPUT_DIR)
139
140 print(n_samples, "samples total!")
141
142 for i, filename in enumerate(placentas):
143
144     print('*'*80)
145     print(f'extracting PCSVN of {filename}\t({i} of {n_samples})')
146
147     # --- Setup, Preprocessing, Frangi Filter (it's mixed up) -----
148
149     raw_img = get_named_placenta(filename, mode=FLATTEN_MODE)
150     cimg = open_typefile(filename, 'raw')
151
152     ucip = open_typefile(filename, 'ucip')
153
154     if REMOVE_CUTS:

```

```

155     #img, has_cut = mask_cuts_simple(raw_img, ucip, return_success=True)
156     #img.data[img.mask] = 0 # actually zero out that area
157     print("removing cuts doesn't do anything anymore")
158     pass
159 else:
160     img = raw_img.copy()
161
162 if REMOVE_GLARE:
163     img = inpaint_hybrid(img)
164
165
166 if USE_NPZ_FILES:
167     # find the first npz file with the sample name in it in the
168     # specified directory.
169     stub = filename.rstrip('.png')
170     for f in os.scandir(NPZ_DIR):
171         if f.name.endswith('npz') and f.name.startswith(stub):
172             npz_filename = os.path.join(NPZ_DIR, f.name)
173             print(f'using the npz file {npz_filename}')
174             break # we'll just use the first one we can find.
175     else:
176         print(f'no npz file found for {filename}.')
177     npz_filename = None
178 else:
179     npz_filename = None
180
181 # set a lambda function to make output file names
182 outname = get_outname_lambda(filename, output_dir=OUTPUT_DIR)
183
184 if npz_filename is not None:
185
186     F = np.load(npz_filename)['F']
187
188     # in case preprocessing happens inside extract_pcsvn, do it out here
189
190     print('successfully loaded the frangi targets!')
191
192 else:
193     print('finding multiscale frangi targets')
194
195     # F is an array of frangi scores of shape (*img.shape, N_SCALES)
196     F, jfile = extract_pcsvn(img, filename, dark_bg=DARK_BG, beta=BETAS,
197                               scales=scales, gamma=GAMMAS, c=CS,
198                               kernel='discrete', dilate_per_scale=True,
199                               verbose=False, signed_frangi=SIGNED_FRANGI,
200                               generate_json=True, output_dir=OUTPUT_DIR,
201                               rescale_frangi=RESCALE_FRANGI,
202                               gradient_filter=GRADIENT_FILTER)
203
204     if MAKE_NPZ_FILES:
205         npzfile = ".".join((outname("F").rsplit('.', maxsplit=1)[0], 'npz'))
206         print("saving frangi targets to ", npzfile)
207         np.savez_compressed(npzfile, F=F)
208
209 # - Multiscale Analysis, etc -----
210
211 # This is the maximum frangi response over all scales at each location
212 Fmax = F.max(axis=-1)
213
214 print("...making outputs")
215
216 if ALPHAS is None:
217     print("thresholding ALPHAS with top 5% scores at each scale")

```

```

219     ALPHAS = np.array([nz_percentile(F[..., k], 95.0)
220                         for k in range(N_SCALES)])
221                     )
222 # the maximum value of the entire image at each scale
223 scale_maxes = np.array([F[...,i].max() for i in range(F.shape[-1])])
224
225 table = pandas.DataFrame(np.dstack((scales, ALPHAS, scale_maxes)).squeeze(),
226                           columns=('σ', 'α_p', 'max(F_σ)'))
227
228 print(table)
229 # threshold the responses at each of these values and get labels of max
230
231 # --- Segmentation Postprocessing -----
232
233 # get the main (boolean) tracefile and the RGB tracefiles
234 trace = open_tracefile(filename, as_binary=True)
235 A_trace = open_typefile(filename, 'arteries')
236 if A_trace is None:
237     # there are no special trace files for this sample
238     skeltrace = skeletonize_trace(trace)
239 else:
240     V_trace = open_typefile(filename, 'veins')
241     skeltrace = skeletonize_trace(A_trace, V_trace)
242
243     # get a matrix of pixel widths in the trace
244     widths = merge_widths_from_traces(A_trace, V_trace, strategy='arteries')
245
246 # find cord insertion point and resolution of the image
247 ucip_midpoint, resolution = measure_ncs_markings(ucip)
248 # if verbose:
249 #     print(f"The umbilical cord insertion point is at {ucip_midpoint}")
250 #     print(f"The resolution of the image is {resolution} pixels per cm.")
251
252 if ucip_midpoint is None:
253     ucip_mask = img.mask
254 # mask anywhere close to the UCIP
255 else:
256     ucip_mask = add_ucip_to_mask(ucip_midpoint,
257                                   radius=int(UCIP_RADIUS), mask=img.mask)
258
259 # The following are examples of things you can do:
260
261 # matrix of widths of traced image
262 # min_widths = merge_widths_from_traces(A_trace, V_trace,
263 #                                         strategy='minimum')
264
265 # trace ignoring largest vessels (19 pixels wide)
266 # trace_smaller_only = filter_widths(min_widths, min_width=3, max_width=17)
267 # trace_smaller_only != 0
268
269 # -- Segmentation Strategies -----
270
271 # strawman
272 from skimage.filters import threshold_mean
273 from functools import partial
274
275 confusion_matrix = partial(confusion, truth=trace, bg_mask=ucip_mask)
276 mcc_with_counts = partial(mcc, truth=trace, bg_mask=ucip_mask,
277                           return_counts=True)
278 percent_network_coverage = lambda a: np.sum(skeltrace&a)/np.sum(skeltrace)
279 precision = lambda t: int(t[0]) / int(t[0] + t[2])
280 V = np.transpose(F, axes=(2, 0, 1))
281
282

```

```

283
284 approx_sm = threshold_mean(img.filled(img.compressed().mean()))
285 mcc_sm, counts_sm = mcc_with_counts(approx_sm)
286 prec_sm = precision(counts_sm)
287 confuse_sm = confusion_matrix(approx_sm)
288 pnc_sm = percent_network_coverage(approx_sm)
289
290 approx_PF, labs_PF = apply_threshold(F, ALPHAS, return_labels=True)
291 mcc_PF, counts_PF = mcc_with_counts(approx_PF)
292 confuse_PF = confusion_matrix(approx_PF)
293 pnc_PF = percent_network_coverage(approx_PF)
294 prec_PF = precision(counts_PF)
295
296 approx_FA, labs_FA = apply_threshold(F, FIXED_ALPHA)
297 mcc_FA, counts_FA = mcc(approx_FA, trace, ucip_mask, return_counts=True)
298 confuse_FA = confusion_matrix(approx_FA)
299 pnc_FA = percent_network_coverage(approx_FA)
300 prec_FA = precision(counts_FA)
301
302 approx_RW, labs_RW = random_walk_scalewise(F, FIXED_ALPHA, return_labels=True)
303 confuse_RW = confusion_matrix(approx_RW)
304 mcc_RW, counts_RW = mcc_with_counts(approx_RW)
305 pnc_RW = percent_network_coverage(approx_RW)
306 prec_RW = precision(counts_RW)
307
308 sieved = sieve_scales(V, 98, 95)
309 approx_S, labs_S = (sieved != 0), sieved
310 confuse_S = confusion_matrix(approx_S)
311 mcc_S, counts_S = mcc_with_counts(approx_S)
312 pnc_S = percent_network_coverage(approx_S)
313 prec_S = precision(counts_S)
314
315 # PUT MARGIN ADD IN HERE
316 #
317 #
318 #
319 #
320
321 mccs[filename] = (mcc_PF, mcc_FA, mcc_RW, mcc_S, mcc_sm)
322 pncs[filename] = (pnc_PF, pnc_FA, pnc_RW, pnc_S, pnc_sm)
323 precisions[filename] = (prec_PF, prec_FA, prec_RW, prec_S, prec_sm)
324
325 scoretable = pandas.DataFrame(np.vstack((mccs[filename], pncs[filename],
326                                         precisions[filename])),
327                                         columns=('PF', 'FA', 'RW', 'PS', 'SM'),
328                                         index=('MCC', 'skel coverage', 'precision'))
329
330 print(scoretable)
331 print('\n\n')
332 print(scoretable.to_latex())
333
334
335 # use only some scales
336 #approx_L0, labs_L0 = apply_threshold(F[:, :, L0_offset:], ALPHAS[L0_offset:])
337 # fix labels to incorporate offset
338 #labs_L0 = (labs_L0 != 0)*(labs_L0 + L0_offset)
339 #confuse_L0 = confusion(approx_L0, trace, bg_mask=ucip_mask)
340
341 #TP, TN, FP, FN = counts
342
343 # this all just verifies that the 4 categories were added up
344 # correctly and match the total number of pixels in the reported
345 # placental plate.
346 #total = np.sum(~ucip_mask)

```

```

347 #print(f'TP: {TP}\t TN: {TN}\nFP: {FP}\tFN: {FN}')
348 # just a sanity check
349 #print(f'TP+TN+FP+FN={TP+TN+FP+FN}\ttotal pixels={total}')
350
351
352 #approx_rw, markers, margins_added = random_walk_fill(img, Fmax, .3, .01,
353 #                                         DARK_BG)
354
355
356
357
358 #view_slices(F[crop], axis=-1, scales=scales)
359
360 # --- Generating Visual Outputs-----
361
362 # cmap and the "set bad" argument / mask color
363 SCALE_CMAP = ('plasma', (1,1,1,1))
364
365 crop = cropped_args(img) # these indices crop out the mask significantly
366
367 fmax_colors = plt.cm.plasma
368 fmax_colors.set_bad('k', 1)
369
370 # save the raw, unaltered image
371 plt.imsave(outname('0_raw'), cimg[crop])
372
373 # save the preprocessed image
374 plt.imsave(outname('1_img'), img[crop].filled(0), cmap=plt.cm.gray)
375
376 # save the maximum frangi output over all scales
377 plt.imsave(outname('2_fmax'), ma.masked_where(Fmax==0,Fmax)[crop], vmin=0,
378 vmax=1.0, cmap=fmax_colors)
379
380 # only save the colorbar the first time
381 #save_colorbar = (i==0)
382 #scale_label_figure(labs, scales, crop=crop,
383 #                     savefilename=outname('3_labeled'), image_only=True,
384 #                     save_colorbar_separate=save_colorbar,
385 #                     basecolor=SCALE_CMAP[1], base_cmap=SCALE_CMAP[0],
386 #                     output_dir=OUTPUT_DIR)
387
388 #plt.imsave(outname('4_confusion'), confuse[crop])
389
390 #scale_label_figure(labs_rw, scales, crop=crop,
391 #                     savefilename=outname('A_labeled_rw'), image_only=True,
392 #                     save_colorbar_separate=save_colorbar,
393 #                     basecolor=SCALE_CMAP[1], base_cmap=SCALE_CMAP[0],
394 #                     output_dir=OUTPUT_DIR)
395
396
397 #plt.imsave(outname('7_confusion_FA'), confuse_FA[crop])
398 #plt.imsave(outname('B_confusion_rw'), confuse_rw[crop])
399 ##plt.imsave(outname('A_markers_rw'), markers[crop])
400 ##plt.imsave(outname('9_margin_for_rw'), confuse_margins[crop])
401
402
403
404 st_colors = {
405     'TN': (79, 79, 79), # true negative# 'f7f7f7'
406     'TP': (0, 0, 0), # true positive # '000000'
407     'FN': (201, 53, 108), # false negative # 'f1a340' orange
408     'FP': (92, 92, 92), # false positive
409     'mask': (247, 200, 200) # mask color (not used in MCC calculation)
410 }

```

```

411
412 #plt.imsave(outname('5_coverage'), confusion(approx, skeltrace,
413 #                                         colordict=st_colors)[crop])
414 #plt.imsave(outname('8_coverage_FA'), confusion(approx_FA, skeltrace,
415 #                                         colordict=st_colors)[crop])
416 #plt.imsave(outname('C_coverage_rw'), confusion(approx_rw, skeltrace,
417 #                                         colordict=st_colors)[crop])
418
419 # make the graph that shows what scale the max was pulled from
420
421 #scale_label_figure(labs_FA, scales, crop=crop,
422 #                     savefilename=outname('6_labeled_FA'), image_only=True,
423 #                     basecolor=SCALE_CMAP[1], base_cmap=SCALE_CMAP[0],
424 #                     save_colorbar_separate=False, output_dir=OUTPUT_DIR)
425
426
427 #scale_label_figure(labs_S, scales, crop=crop,
428 #                     savefilename=outname('D_labeled_S'), image_only=True,
429 #                     basecolor=SCALE_CMAP[1], base_cmap=SCALE_CMAP[0],
430 #                     save_colorbar_separate=False, output_dir=OUTPUT_DIR)
431
432 #plt.imsave(outname('E_confusion_S'), confuse_S[crop])
433
434
435
436
437 ##### THIS IS ALL A HORRIBLE MESS. FIX IT
438 # why don't you just return the dict instead
439 with open(jfile, 'r') as f:
440     slog = json.load(f)
441
442 c2d = lambda t: dict(zip(('TP', 'TN', 'FP', 'FN'), [int(c) for c in t]))
443
444 slog['counts'] = c2d(counts)
445 slog['counts_FA'] = c2d(counts_FA)
446 slog['counts_rw'] = c2d(counts_rw)
447 slog['counts_S'] = c2d(counts_S)
448 slog['pnc'] = pnccs[filename]
449 slog['mcc'] = mccs[filename]
450 slog['scale_maxes'] = list(scale_maxes)
451 slog['ALPHAS'] = list(ALPHAS)
452 slog['precision'] = precisions[filename]
453
454
455 with open(jfile, 'w') as f:
456     json.dump(slog, f)
457
458 plt.close('all')
459
460 # Post-run Meta-Output and Logging -----
461
462 timestamp = datetime.datetime.now()
463 timestamp = timestamp.strftime("%y%m%d_%H%M")
464
465 mccfile = os.path.join(OUTPUT_DIR, f"runlog_{timestamp}.json")
466
467 runlog = {
468     'time': timestamp,
469     'DARK_BG': DARK_BG,
470     'DILATE_PER_SCALE': DILATE_PER_SCALE,
471     'SCALE_RANGE': SCALE_RANGE,
472     'SCALE_TYPE': SCALE_TYPE,
473     'N_SCALES': N_SCALES,
474     'scales': list(scales),

```

```

475     'ALPHAS': list(ALPHAS),
476     'BETAS': None,
477     'use_npz_files': False,
478     'remove_glare': REMOVE_GLARE,
479     'files': list(placentas),
480     'MCCS': mccs,
481     'PNCS': pncts,
482     'precisions': precisions
483 }
484
485 # save to a json file
486 with open(mccfile, 'w') as f:
487     json.dump(runlog, f, indent=True)

```

listings/find_plate.py

```

1 # coding: utf-8
2 #filename = 'T-BN4981652.png'
3 #img = get_named_placenta(filename)
4 #crop = cropped_args(img)
5 #plt.imshow(img)
6 #s = plt.show
7 #s()
8 #plt.imshow(img > .9*img.mask)
9 #plt.show()
10 #plt.imshow(img > .9*img.max())
11 #plt.show()
12 #plt.imshow(img > .8*img.max())
13 #plt.show()
14 #plt.imshow(img > .75*img.max())
15 #plt.show()
16 #from skimage.filters import sobel
17 #sobel(img)
18 #plt.imshow(_)
19 #plt.show()
20 #from skimage import filters
21 #filters.laplace(img)
22 #plt.imshow(_)
23 #plt.show()
24 #dilate_boundary(img, 20)
25 #plt.imshow(_)
26 #plt.show()
27 #filters.laplace(dilate_boundary(img, 30))
28 #plt.imshow(_)
29 #plt.show()
30 #filters.laplace(dilate_boundary(img, 30) == 0)
31 #plt.imshow(_)
32 #s()
33 #filters.laplace(dilate_boundary(img, 30))==0
34 #plt.imshow(_)
35 #plt.show()
36 #from skimage import morphology as mph
37 #from frangi import frangi_from_image
38 #fft_gradient(img, sigma=20)
39 #plt.imshow(_)
40 #plt.show()
41 #fft_gradient(img, sigma=15)
42 #plt.imshow(_)
43 #plt.show()
44 #raw_img
45 #rimg
46 from placenta import open_typefile
47 open_typefile(filename, 'raw')

```

```

48 plt.imshow(_)
49 plt.show()
50 raw = open_typefile(filename, 'raw')
51 plt.imshow(raw[...,:1])
52 plt.show()
53 plt.imshow(fft_gradient(raw[...,:1], sigma=15) )
54 plt.show()
55 plt.imshow(fft_gradient(raw[...,:1], sigma=.01))
56 plt.show()
57 plt.imshow(fft_gradient(raw[...,:1], sigma=.001))
58 plt.show()
59 from skimage.segmentation import watershed
60 get_ipython().run_line_magic('pinfo', 'watershed')
61 marks = np.zeros(img.shape, np.int32)
62 marks[0,0] = 1
63 marks[img.shape//2]
64 marks[img.shape[0]//2, img.shape[1]//2]
65 marks[img.shape[0]//2, img.shape[1]//2] = 2
66 watershed(fft_gradient(raw[...,:1], sigma=.01))
67 watershed(fft_gradient(raw[...,:1], sigma=.01), marks)
68 plt.imshow(_)
69 plt.show()
70 watershed(fft_gradient(raw[...,:1], sigma=.001), marks)
71 plt.show()
72 plt.imshow(_)
73 plt.show()
74 g = fft_gradient(raw[...,:1], sigma=.01)
75 plt.imshow(g)
76 plt.show()
77 g = fft_gradient(raw[...,:1], sigma=.001)
78 plt.imshow(g)
79 plt.show()
80 from skimage.segmentation import find_boundaries
81 find_boundaries(g)
82 plt.imshow(_)
83 plt.show()
84 plt.imshow(g>g.mean())
85 plt.show()
86 marks[g>g.mean()] = 2
87 watershed(g, marks)
88 plt.imshow(_)
89 plt.show()
90 from scipy.ndimage import find_objects
91 find_objects(watershed(g,marks))
92 help(find_objects)
93 from scipy.ndimage import label
94 get_ipython().run_line_magic('pinfo', 'label')
95 label(watershed(g,marks))
96 help(label)
97 plt.imshow(watershed(g,marks))
98 plt.show()
99 plt.imshow(watershed(g,marks))
100 plt.show()
101 from skimage.morphology import binary_erosion
102 plt.imshow(watershed(g,marks), vmin=0, vmax=2)
103 plt.show()
104 w = watershed(g,marks)
105 plt.imshow(w)
106 plt.show()
107 from skimage.morphology import binary_erosion, disk
108 binary_erosion(w, disk(10))
109 plt.imshow(_)
110 plt.show()
111 plt.imshow(w-1)

```

```

112 plt.show()
113 plt.imshow(binary_erosion(w-1, disk(10)))
114 plt.show()
115 # oops need to make watershed thing binary. w-1 is hacky but works
116 eroded = binary_erosion(w-1, disk(10))
117 find_boundaries(eroded)
118 plt.imshow(_)
119 plt.show()
120 import scipy.ndimage as ndi
121 ndi.label(eroded)
122 plt.imshow(_[0])
123 plt.show()
124 labs, nlabs = ndi.label(eroded)
125 for i in range(nlabs):
126     print(i, np.sum(labs==i))
127
128 labs==1
129 plt.imshow(_)
130 plt.show()
131 plt.imshow(np.logical_xor(labs==1, ~img.mask))
132 plt.show()
133 sum(~img.mask)
134 np.sum(~img.mask)
135 np.sum(labs==1)

```

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
18 def r(y, beta):
19     """normalized anisotropy factor
20     y is the ratio |k1 / k2|, so y=0 corresponds to perfectly isotropic
21     and y->1 corresponds to highly anisotropic
22     """
23     return np.exp(-y**2 / (2*beta**2))
24
25 #plt.rc('text', usetex=True)
26 #plt.rc('font', family='serif')
27
28
29 dom = np.linspace(0, 1)
30
31 plt.close('all')
32
33 # show dependence of structureness factor on its parameter
34 for gamma in [0.1, 0.25, 0.35, 0.5, 0.9, 1, 2, 10, 1000]:
35     plt.plot(dom, s(dom, gamma), label=r'$\gamma = ' + .format(gamma))
```

```

36
37 plt.ylabel(r' $\left(1 - \exp\left\{\frac{-S}{2(\gamma S_{max})^2}\right\}\right)$ ' ,
38         fontsize=14)
39 plt.xlabel(r' $(S/S_{max})$ ', fontsize=14)
40 plt.title(r'Dependence of Structureness Factor on Parameter  $\gamma = (c/S_{max})$ ')
41 plt.legend()
42
43 #plt.show()
44 plt.close('all')
45
46 for beta in [0.1, 0.25, 0.35, 0.5, 0.9, 1, 2, 10, 1000]:
47     plt.plot(dom, r(dom, beta), label=r' $\beta = .format(beta)')

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     ax.set_title(r'Rescaled Frangi filter, "
73                 r" $\beta = , \gamma = .format(beta, gamma))$ ")
74
75     #fig.colorbar(surf, shrink=0.5, aspect=5)
76     fig.tight_layout()
77
78     plt.savefig(f'demo_output/frangi3d/{n}.png', dpi=300)
79
80     plt.close()
81
82 imgs = [imread(f'demo_output/frangi3d/{n}.png') for n in range(36*1)]
83 imgs = np.stack(imgs)
84
85 for n in range(6):
86     plt.imsave(f'demo_output/frangi3d/frangi3dpart{n}.png',
87                 montage(imgs[(n*6):((n+1)*6)], multichannel=True,
88                         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
       Frangi score will only be 0.86 at its maximum. Larger values of gamma

```

```

68 will only dampen the frangi filter more. Smaller values toward 0 will
69 result in a "looser" filter. For example, if gamma = .25, then the
70 maximum score is (1-exp{-8}) around .999 (it may be desirable that the
71 franginess score should be able to achieve a score of 1).
72
73 This function will accept 0 an input, and the structureness factor will
74 be set to 1 everywhere (the limiting case as gamma -> 0)
75
76 Frangi structureness factor is (1 - exp((-S**2)/(2*c**2)))
77 """
78 hesh = fft_hessian(img, sigma, kernel=kernel) # the triple (Hxx,Hxy,Hyy)
79 # calculate principal curvatures with |k1| <= |k2|
80
81
82 k1, k2 = principal_curvatures(img, sigma, H=hesh)
83
84 if dilation_radius is not None:
85     # pass None to just get the mask back
86     collar = dilate_boundary(None, radius=dilation_radius, mask=img.mask)
87
88     # get rid of "bad" K values before you calculate gamma and Frangi
89     k1[collar] = 0
90     k2[collar] = 0
91     hesh[0][collar] = 0
92     hesh[1][collar] = 0
93     hesh[2][collar] = 0
94 else:
95     collar = img.mask.copy()
96
97
98 # no need to set gamma or c anymore. will be set inside get_frangi_targets
99 #if c is None:
100 # Frangi suggested 'half the max Hessian norm' as an empirical
101 # half the max spectral radius is easier to calculate so do that
102 # shouldn't be affected by mask data but should make sure the
103 # mask is *well* far away from perimeter
104 # we actually calculate half of max hessian norm
105 # using frob norm = sqrt(trace(AA^T))
106 # alternatively you could use gamma = .5 * np.abs(k2).max()
107 #hnorm = hessian_norm(hesh, mask=collar)
108 #print(f'\sigma={sigma:.2f}')
109 #gamma0 = .5*hessian_norm(hesh).max()
110 #print(f'\t{gamma0:.5f} = frob-norm \gamma pre-dilation')
111
112 #gamma1 = .5*hessian_norm(hesh, mask=collar).max()
113 #print(f'\t{gamma1:.5f} = frob-norm \gamma post-collar dilation {dilation_radius}')
114 #l2gamma = .5*np.max(np.abs(k2))
115 #print(f'\t{l2gamma:.5f} = from L2-norm \gamma (K2 with collar)')
116
117 #hdilation = int(max(np.ceil(sigma),10))
118 #hcollar = dilate_boundary(None, radius=hdilation, mask=img.mask)
119 #gamma = .5 * max_hessian_norm(hesh, mask=hcollar)
120
121 #print(f'\t{gamma:.5f} = \gamma post-hdilation (radius {hdilation}) (old \gamma)')
122
123 #print('changing \gamma to L2-norm with collar')
124 #gamma = max(gamma1, l2gamma, gamma, gamma0)
125
126 # wish this scaled a little better
127
128 # a very large gamma here will make the Frangi score zero
129 # a very small gamma means that we are artificially inflating the
130 # structureness measure
131

```

```

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()})')
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()})')
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={beta} and gamma={c:.2}')
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': hess,
184                 'F': targets,
185                 'k1': k1,
186                 'k2': k2,
187                 'border_radius': dilation_radius
188             }
189
190     return targets, scale_dict
191
192 def get_frangi_targets(K1, K2, beta=0.5, gamma=0.5, c=None,
193                         dark_bg=True, signed=False, rescale_frangi=False):
194     """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         if gamma==0:
259             # prevent division by zero

```

```

260             max_theoretical = 1
261     else:
262         max_theoretical = (1 - np.exp(-1/(2*gamma**2)))
263     F = F / max_theoretical
264
265 # now just filter/ change sign as appropriate.
266 if not signed:
267     # calculate the regular frangi filter
268     if dark_bg is None:
269         #keep F the way it is
270         pass
271     elif dark_bg:
272         # zero responses from positive curvatures
273         F = (K2 < 0)*F
274     else:
275         # zero responses from negative curvatures
276         F = (K2 > 0)*F
277 else:
278     if dark_bg is None:
279         # output is already signed
280         pass
281     elif dark_bg:
282         # positive curvature spots will be made negative
283         F[K2 > 0] = -1 * F[K2 > 0]
284     else:
285         # negative curvature spots will be made positive
286         F[K2 < 0] = -1 * F[K2 < 0]
287
288 # finally, reapply the mask if the inputs came with one
289 if numpy.ma.is_masked(K1):
290     F = numpy.ma.masked_array(F, mask=K1.mask)
291     # zero out masked region (there can be issues here if the output is
292     # signed, just makes stacking easier
293     F[K1.mask] = 0
294
295 return F
296
297
298 def hessian_norm(hesh, mask=None):
299     """Calculate Frobenius norm of Hessian.
300
301     Calculates the maximal value (over all pixels of the image) of the
302     Frobenius norm of the Hessian. This should be the same as the square root
303     of unscaled structurenness.
304
305     Parameters
306     -----
307     hesh: a tuple of ndarrays
308         The tuple hxx,hxy,hyy which are all the same shape. The hessian at
309         the point (m,n) is then [[hxx[m,n], hxy[m,n]],
310                               [hxy[m,n], hyy[m,n]]]
311
312     Returns
313     -----
314     float
315     """
316
317     hxx, hxy, hyy = hesh
318
319     # frob norm is just sqrt(trace(AA^T)) which is easy for a 2x2
320     hnrm = (hxx**2 + 2*hxy**2 + hyy**2)
321
322     if mask is not None:
323         hnrm[mask] = 0

```

```

324
325     hnorm = np.sqrt(hnorm)
326     return hnorm
327
328
329 def anisotropy(K1,K2, beta=0.5):
330     """Convenience function for the exponential argument in the Frangi
331     anisotropy factor.
332
333     According to Frangi (1998) this is technically (A**2) / (2*beta**2)
334     unless beta is None, in which case just A**2 is returned
335
336     The frangi vesselness factor is formally (np.exp(-R))
337     where R is what's returned by this function
338     """
339
340     A = (K1 / K2) ** 2
341     #print(f'inside anisotropy, \beta={beta}')
342     if beta == 0:
343         return np.full(A.shape, np.inf) # the limiting case as beta -> 0
344
345     elif beta == 'inf' or np.isinf(beta):
346         return np.zeros_like(A) # the limiting case as beta -> inf
347
348     elif beta is None:
349         return A # just return the A**2 part (why though)
350     else:
351         return A / (2*beta**2)
352
353
354 def structureness(K1, K2, gamma=0.5, c=None):
355     """Convenience function for Structureness measure.
356     According to Frangi (1998) this is technically S**2
357     """
358     S = K1**2 + K2**2
359
360
361     # is c is not provided, calculate it
362     if c is None:
363         c = gamma * np.sqrt(S).max() # the max Frob norm of the Hessian
364
365     #print(f'inside structureness, \gamma={gamma}, c={c}')
366
367     if c == 0:
368         return np.full(S.shape, np.inf)
369
370     elif c == 'inf' or np.isinf(c):
371         return np.zeros_like(S)
372
373     elif c is None:
374         return S
375
376     else:
377         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 mimg_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  $\sigma={\sigma:.2f}$ ', 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  $\sigma={\sigma:.2f}$ ')
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
5 BOILERPLATE
6
7 show that gaussian blur of hfft is accurate, except potentially around the
8 boundary proportional to sigma.
9
10 or if they're off by a scaling factor, show that the derivates
11 (taken the same way) are proportional.
12
13 pseudocode
14
15 A = gaussian_blur(image, sigma, method='conventional')
16 B = gaussian_blur(image, sigma, method='fourier')
17
18 zero_order_accurate = isclose(A, B, tol)
19
20 J_A = get_jacobian(A)
21 J_B = get_jacobian(B)
22
23 first_order_accurate = isclose(J_A, J_B, tol)
24
25 A_eroded = zero_around_plate(A, sigma)
26 B_eroded = zero_around_plate(B, sigma)
27
28 J_A_eroded = zero_around_plate(J_A, sigma)
29 J_B_eroded = zero_around_plate(J_B, sigma)
30
31 zero_order_accurate_no_boundary = isclose(A_eroded, B_eroded, tol)
32 first_order_accurate = isclose(J_A_eroded, J_B_eroded, tol)
33
34 """
35
36
37 from placenta import get_named_placenta, cropped_args
38 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[it_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={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  $\sigma = \text{.format}(\sigma)$ ')
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  $\sigma = \text{.format}(\sigma)$ ')
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 image based in frequency space (rather than direct convolution with a gaussian
19 as is standard in scipy, for example).
20
21 TODO: PROVIDE MAIN USAGE NOTES

```

```

23 """
24 def fft_gaussian(img, sigma, kernel=None):
25     """
26     https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.fftconvolve.html
27     in particular the example in which a gaussian blur is implemented.
28     along with the comment:
29     "Gaussian blur implemented using FFT convolution. Notice the dark borders
30     around the image, due to the zero-padding beyond its boundaries. The
31     convolve2d function allows for other types of image boundaries, but is far
32     slower"
33     (i.e. doesn't use FFT).
34
35     note that here, you actually take the FFT of a gaussian (rather than
36     build it in frequency space). there are ~6 ways to do this.
37     """
38     #create a 2D gaussian kernel to take the FFT of
39     # output of signal.gaussian is normalized to 1 so you need to scale
40     # it back to work
41     #A = 1 / (2*np.pi*sigma**2) # scale factor for 2D
42
43     if kernel in ('discrete', None):
44         kern_x = discrete_gaussian_kernel(img.shape[0], sigma)
45         kern_y = discrete_gaussian_kernel(img.shape[1], sigma)
46     elif kernel == 'sampled':
47         # scaling factor for 1d gaussian (use it twice)
48         A = 1 / (np.sqrt((2*np.pi)*sigma**2))
49         kern_x = A*signal.gaussian(img.shape[0], sigma)
50         kern_y = A*signal.gaussian(img.shape[1], sigma)
51     else:
52         raise ValueError("Key must be 'discrete' or 'sampled'")
53
54     kernel = np.outer(kern_x, kern_y)
55
56     return signal.fftconvolve(img, kernel, mode='same')
57
58 def discrete_gaussian_kernel(n_samples, sigma):
59     """
60     sigma is the scale, n_samples is the number of samples to compute
61     will return a window centered a zero
62     i.e. arange(-n_samples//2, n_samples//2+1)
63
64     not sure how to center it on zero though, since integers only
65
66     note! to make this work similarly to fft_gaussian, this uses
67     sigma = np.sqrt(t). Usually you'll find this in terms of t
68
69     by using scipy.special.ive instead we prevent blowups
70     """
71     dom = np.arange(-(n_samples//2), (n_samples//2) + 1)
72     #there should be a scaling parameter alpha but whatever
73     #return np.exp(-t) * iv(dom,t)
74     if (n_samples % 2) == 0:
75         dom = dom[1:]
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.

```

```

87 """
88 m,n = img.shape
89 # i don't know if this will suck if there are odd dimensions
90 kernel = np.outer(discrete_gaussian_kernel(m,sigma**2),
91                     discrete_gaussian_kernel(n,sigma**2))
92
93     return signal.fftconvolve(img, kernel, mode='same')
94
95 def fft_fdgk(img,sigma):
96     """
97     convolve with discrete gaussian kernel in freq. space
98     """
99     # this would be a lot better since you wouldn't have to deal
100    # with an arbitrary cutoff of size of the discrete kernel
101    # since the freq. space version is just
102    #  $\exp\{\alpha^*t(\cos\theta - 1)\}$ 
103    # see formula 22 of lindeberg discrete paper
104
105    pass
106
107 def fft_hessian(image, sigma=1., kernel=None):
108     """
109     a reworking of skimage.feature.hessian_matrix that uses
110     FFT to compute gaussian, which results in a considerable speedup
111
112     INPUT:
113         image - a 2D image (which type?)
114         sigma - coefficient for gaussian blur
115         kernel - input to fft_gaussian
116         gradient - if you've already computed this
117
118     OUTPUT:
119         (Lxx, Lxy, Lyy) - a triple containing three arrays
120             each of size image.shape containing the xx, xy, yy derivatives
121             respectively at each pixel. That is, for the pixel value given
122             by image[j][k] has a calculated 2x2 hessian of
123             [ [Lxx[j][k], Lxy[j][k]],  

124               [Lxy[j][k], Lyy[j][k]] ]
125     """
126
127     gaussian_filtered = fft_gaussian(image, sigma=sigma, kernel=kernel)
128
129     gradients = np.gradient(gaussian_filtered)
130
131     axes = range(image.ndim)
132
133     H_elems = [np.gradient(gradients[ax0], axis=ax1)
134                for ax0, ax1 in combinations_with_replacement(axes, 2)]
135
136     return H_elems
137
138
139 def fft_gradient(image, sigma=1.):
140     """ returns gradient norm """
141
142     gaussian_filtered = fft_gaussian(image, sigma=sigma)
143
144     Lx, Ly = np.gradient(gaussian_filtered)
145
146     return np.sqrt(Lx**2 + Ly**2)
147
148
149 def demo(img=None):
150     """

```

```

151 old main function for testing.
152
153 This simply tests fft_gaussian on a test image,
154 """
155
156 if img is None:
157     img = img_as_float(camera())
158 else:
159     img = img_as_float(img)
160
161 sample_sigmas = (.5, 2, 8, 30)
162 #sample_sigmas = (.2, 2)
163
164 # build the graphs here side by side
165 # show regular blur, sampled blur, discrete blur, 1d plot of signals
166 # so a 4 by 4 grid
167
168 fig, axes = plt.subplots(nrows=len(sample_sigmas), ncols=4,
169                         figsize=(10, 10))
170
171 for cax, sigma in enumerate(sample_sigmas):
172
173     # convolve the image with a gaussian kernel, one of three ways
174     fft_dgk = fft_gaussian(img, sigma, kernel='discrete')
175     fft_sampled = fft_gaussian(img, sigma, kernel='sampled')
176     xy_sampled = gaussian_filter(img, sigma, mode='constant', cval=0)
177
178     # make the fancy sample
179     N = 80
180     dom = np.arange(-(N//2), N//2 + 1)
181     dgk = discrete_gaussian_kernel(N, sigma)
182     A = np.sqrt(2*np.pi*sigma**2)
183     A = 1 / A
184     sgk = A * signal.gaussian(N+1, sigma)
185
186     axes[cax, 0].imshow(xy_sampled, cmap='gray', vmin=0, vmax=1)
187     axes[cax, 0].set_ylabel(r'$\sigma$ = ' + format(sigma))
188     #axes[cax, 0].set_title(f'ndi.gaussian_filter, $\sigma$={sigma}')
189     #axes[cax, 0].axis('off')
190     axes[cax, 0].set_xticks([])
191     axes[cax, 0].set_yticks([])
192
193     axes[cax, 1].imshow(fft_sampled, cmap='gray', vmin=0, vmax=1)
194     #axes[cax, 1].imshow(fft_sampled, cmap='gray')
195     #axes[cax, 1].set_title(f'fft sampled kernel, $\sigma$={sigma}')
196     axes[cax, 1].axis('off')
197
198     axes[cax, 2].imshow(fft_dgk, cmap='gray', vmin=0, vmax=1)
199     #axes[cax, 2].set_title(f'fft discrete kernel, $\sigma$={sigma}')
200     axes[cax, 2].axis('off')
201
202
203     axes[cax, 3].plot(dom, sgk, 'k', dom, dgk, 'g:')
204     #axes[cax, 3].set_title(f'discrete vs. sampled kernel $\sigma$={sigma}')
205     #axes[cax, 3].axes.set_aspect('equal')
206
207     # set titles for the first column
208     axes[0,0].set_title('(a)')
209     axes[0,1].set_title('(b)')
210     axes[0,2].set_title('(c)')
211     axes[0,3].set_title('(d)')
212
213 plt.tight_layout()
214 plt.show()

```

```

215 def compare_mae(arr1, arr2):
216     assert arr1.shape == arr2.shape
217     return np.abs(arr1 - arr2).sum() / arr1.size
218
219
220
221 def semigroup_demo(img=None):
222     """
223         the step ones don't look anywhere near as blurred as the initial image
224         in any case! don't use this till it's good!
225     """
226
227     if img is None:
228         img = img_as_float(camera())
229     else:
230         img = img_as_float(img)
231
232     sigma = 45.
233     n_steps = 2
234     sigmas = (10, 35)
235
236     fft_discrete = fft_gaussian(img, sigma, kernel='discrete')
237     fft_sampled = fft_gaussian(img, sigma, kernel='sampled')
238     xy_sampled = gaussian_filter(img, sigma, mode='constant', cval=0)
239
240     step_discrete = img.copy()
241     step_fft_sampled = img.copy()
242     step_xy_sampled = img.copy()
243
244     #sigma_n = sigma / n_steps
245     #sigma_n = np.power(sigma, 1/n_steps)
246     counter = 0
247     for sigma_n in sigmas:
248         step_discrete = fft_gaussian(step_discrete, sigma_n, kernel='discrete')
249         step_fft_sampled = fft_gaussian(step_fft_sampled, sigma_n,
250                                         kernel='sampled')
251         step_xy_sampled = gaussian_filter(step_xy_sampled, sigma_n,
252                                           mode='constant', cval=0)
253         counter += sigma_n
254         #print(counter, end=' ')
255
256     print()
257     er = int(sigma)
258     crop = np.s_[er:-er, er:-er]
259
260     fft_discrete = fft_discrete[crop]
261     fft_sampled = fft_sampled[crop]
262     xy_sampled = xy_sampled[crop]
263     step_discrete = step_discrete[crop]
264     step_fft_sampled = step_fft_sampled[crop]
265     step_xy_sampled = step_xy_sampled[crop]
266
267     fig, axes = plt.subplots(ncols=3, nrows=2)
268     axes[0,0].imshow(xy_sampled, vmin=0, vmax=1, cmap='gray')
269     axes[0,0].set_title('(a)')
270     axes[0,0].set_xticks([]), axes[0,0].set_yticks([])
271
272     axes[0,1].imshow(fft_sampled, vmin=0, vmax=1, cmap='gray')
273     axes[0,1].set_title('(b)')
274     axes[0,1].set_xticks([]), axes[0,1].set_yticks([])
275
276     axes[0,2].imshow(fft_discrete, vmin=0, vmax=1, cmap='gray')
277     axes[0,2].set_title('(c)')
278     axes[0,2].set_xticks([]), axes[0,2].set_yticks([])

```

```

279
280     axes[1,0].imshow(step_xy_sampled, vmin=0, vmax=1, cmap='gray')
281     axes[1,0].axis('off')
282     axes[1,1].imshow(step_fft_sampled, vmin=0, vmax=1, cmap='gray')
283     axes[1,1].axis('off')
284     axes[1,2].imshow(step_discrete, vmin=0, vmax=1, cmap='gray')
285     axes[1,2].axis('off')
286
287     MSE_sampled = compare_mse(xy_sampled, step_xy_sampled)
288     MSE_fft_sampled = compare_mse(fft_sampled, step_fft_sampled)
289     MSE_discrete = compare_mse(fft_discrete, step_discrete)
290
291     print(f'MSE sampled:{MSE_sampled}')
292     print(f'MSE fft_sampled:{MSE_fft_sampled}')
293     print(f'MSE discrete:{MSE_discrete}')
294     print()
295     #NRMSE_sampled = compare_nrmse(xy_sampled, step_xy_sampled)
296     #NRMSE_fft_sampled = compare_nrmse(fft_sampled, step_fft_sampled)
297     #NRMSE_discrete = compare_nrmse(fft_discrete, step_discrete)
298
299     #print(f'NRMSE sampled:{NRMSE_sampled}')
300     #print(f'MSE fft_sampled:{NRMSE_fft_sampled}')
301     #print(f'NRMSE discrete:{NRMSE_discrete}')
302     #for ax, title in zip(axes.ravel(), ['(a)', '(b)', '(c)', '(d)']):
303     #    ax.axis('off')
304     #    ax.set_title(title)
305
306
307     MAE_sampled = compare_mae(xy_sampled, step_xy_sampled)
308     MAE_fft_sampled = compare_mae(fft_sampled, step_fft_sampled)
309     MAE_discrete = compare_mae(fft_discrete, step_discrete)
310
311     print(f'MAE sampled:{MAE_sampled}')
312     print(f'MAE fft_sampled:{MAE_fft_sampled}')
313     print(f'MAE discrete:{MAE_discrete}')
314
315     plt.show()
316
317 if __name__ == "__main__":
318     from skimage.io import imread
319     from placenta import list_by_quality, get_named_placenta
320     #A = list_by_quality(0)[0]
321     #A = get_named_placenta(A)
322     #A = imread('samples/5.3.02.tifff')
323     A = None
324     demo(A)
325     semigroup_demo(A)

```

listings/make_sample_qualities.py

```

1 # coding: utf-8
2 import numpy as np
3 import json
4
5 # what runlog to use
6 jname = 'output/181117-declared/runlog_181118_0610.json'
7
8 with open(jname) as f:
9     runlog = json.load(f)
10
11 M = {k:np.round(v,decimals=3) for k,v in runlog['MCCS'].items()}
12 avg = [m.mean() for m in M.values()]
13

```

```

14 # get divisions for the four sections
15 # good and bad will be half as big as the other two
16 p1 = np.percentile(avgs, 12.5)
17 p2 = np.percentile(avgs, 50.0)
18 p3 = np.percentile(avgs, 87.5)
19
20 print(f'thresholds calculated at {p1:.3%}, {p2:.3%}, {p3:.3%}')
21 qualities = dict()
22 poor = {k:m.mean() for k,m in M.items() if m.mean() <= p1}
23 fair = {k:m.mean() for k,m in M.items() if m.mean() > p1 and m.mean() <= p2}
24 okay = {k:m.mean() for k,m in M.items() if m.mean() > p2 and m.mean() <= p3}
25 good = {k:m.mean() for k,m in M.items() if m.mean() > p3}
26
27 qualities['good'] = good
28 qualities['okay'] = okay
29 qualities['fair'] = fair
30 qualities['poor'] = poor
31
32 print("""
33     run:
34         with open('sample-qualities.json', 'w') as f:
35             json.dump(qualities, f, indent=True)
36
37     ::; to save this. won't do it for you!
38
39     """
)

```

listings/margin_add_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, img_as_int
6 from skimage.io import imread
7 from placenta import (get_named_placenta, list_by_quality, cropped_args,
8                         img_as_float, open_typefile, open_tracefile)
9
10 from frangi import frangi_from_image
11 import numpy.ma as ma
12 from hfft import fft_gradient, fft_hessian, fft_gaussian
13 from merging import nz_percentile, apply_threshold
14 from plate_morphology import dilate_boundary
15 import os.path, os
16
17 from scipy.ndimage import distance_transform_edt as edt
18 from skimage.filters.rank import enhance_contrast_percentile as ecp
19 from skimage.morphology import disk, binary_dilation, thin
20 from scoring import confusion, mcc
21
22 from skimage.filters import threshold_isodata
23 from skimage.exposure import rescale_intensity
24 from postprocessing import dilate_to_rim
25 from preprocessing import inpaint_hybrid
26 import json
27
28 placentas = list_by_quality(0)
29 OUTPUT_DIR = 'output/190130-margin_add_demo'
30
31 if not os.path.exists(OUTPUT_DIR):
32     os.makedirs(OUTPUT_DIR)
33

```

```

34 beta = 0.15
35 gamma = .5
36 THRESHOLD = .4
37 scales = np.logspace(-1.5, 3.5, base=2, num=20)
38
39 mccs = list()
40 precs = list()
41 for filename in placentas:
42     basename = filename.rstrip('.png')
43     basename = basename.strip('T-')
44
45     print(filename, '*'*30)
46     img = get_named_placenta(filename)
47     img = inpaint_hybrid(img)
48     crop = cropped_args(img)
49     trace = open_tracefile(filename)
50
51     # threshold according to ISODATA threshold for a strawman
52     straw = img.filled(0) < threshold_isodata(img.filled(0))
53     straw[img.mask] = False
54
55     F = np.stack([frangi_from_image(img, sigma, beta, gamma,
56                                     dark_bg=False, dilation_radius=20,
57                                     rescale_frangi=True, signed_frangi=True).filled(0)
58                                     for sigma in scales])
59
60     f = F[:-4].max(axis=0)
61     nf = ((-F*(F<0))[:12]).max(axis=0)
62
63     nf = dilate_boundary(nf, mask=img.mask, radius=20).filled(0)
64     spine = dilate_boundary(f, mask=img.mask, radius=20).filled(0)
65     spine = rescale_intensity(spine, in_range=(0,1), out_range='uint8')
66     spine = spine.astype('uint8')
67     bspine = ecp(spine, disk(3)) > (THRESHOLD*255)
68
69     approx, radii = dilate_to_rim(bspine, nf > .05, thin_spine=True,
70                                    return_radii=True)
71
72     approx2, radii2 = dilate_to_rim(spine > (THRESHOLD*255), nf > .05,
73                                     thin_spine=True, return_radii=True)
74
75     approx_FA = (spine > (THRESHOLD*255))
76     approx_PFA = bspine
77
78     ALPHAS = np.array([nz_percentile(F[k], 95.0)
79                         for k in range(len(scales))])
80     approx_PF = apply_threshold(np.transpose(F,(1,2,0)), ALPHAS,
81                                 return_labels=False)
82
83     m_st, counts_st = mcc(straw, trace, bg_mask=img.mask, return_counts=True)
84     m, counts = mcc(approx, trace, bg_mask=img.mask, return_counts=True)
85     m2, counts2 = mcc(approx2, trace, bg_mask=img.mask, return_counts=True)
86     m_PFA, counts_PFA = mcc(approx_PFA, trace, bg_mask=img.mask,
87                               return_counts=True)
88     m_FA, counts_FA = mcc(approx_FA, trace, bg_mask=img.mask,
89                           return_counts=True)
90
91     m_PF, counts_PF = mcc(approx_PF, trace, bg_mask=img.mask,
92                           return_counts=True)
93
94     precision_score = lambda t: int(t[0]) / int(t[0] + t[2])
95
96     p = precision_score(counts)
97     p_st = precision_score(counts_st)

```

```

98 p2 = precision_score(counts2)
99 p_PFA = precision_score(counts_PFA)
100 p_PF = precision_score(counts_PF)
101 p_FA = precision_score(counts_FA)
102
103 fig, ax = plt.subplots(nrows=2, ncols=3, figsize=(25,15))
104
105 ax[0,0].imshow(img[crop], cmap=plt.cm.gray)
106 ax[0,0].set_title(basename)
107
108 ax[1,0].imshow(confusion(straw,trace)[crop])
109 ax[1,0].set_title(rf' ISODATA threshold (Frangi-less)', loc='left')
110 ax[1,0].set_title(f'MCC: {m_st:.2f}\n' f'precision: {p_st:.2%}', loc='right')
111
112 ax[0,1].imshow(confusion(approx_FA, trace)[crop])
113 ax[0,1].set_title(rf' fixed  $\alpha$ =THRESHOLD', loc='left')
114 ax[0,1].set_title(f'MCC: {m_FA:.2f}\n' f'precision: {p_FA:.2%}', loc='right')
115
116 #ax[1,1].imshow(confusion(bspine,trace)[crop])
117 #ax[1,1].set_title(rf' local percentile  $\alpha$ =THRESHOLD', loc='left')
118 #ax[1,1].set_title(f'MCC: {m_PFA:.2f}\n' f'precision: {p_PFA:.2%}', loc='right')
119
120 ax[1,1].imshow(confusion(approx_PF,trace)[crop])
121 ax[1,1].set_title(rf' nz-percentile threshold (p=95)', loc='left')
122 ax[1,1].set_title(f'MCC: {m_PF:.2f}\n' f'precision: {p_PF:.2%}', loc='right')
123
124 ax[0,2].imshow(confusion(approx2,trace)[crop])
125 ax[0,2].set_title(rf' dilate_to_margin', loc='left')
126 ax[0,2].set_title(f'MCC: {m2:.2f}\n' f'precision: {p2:.2%}', loc='right')
127 im = ax[1,2].imshow(2*radii2[crop], cmap='tab20c')
128
129 #fig.subplots_adjust(right=0.9, wspace=0.05, hspace=0.1)
130 #cbar_ax = fig.add_axes([0.9, 0.15, 0.05, 0.7])
131 #fig.colorbar(im, cax=cbar_ax, shrink=0.5)
132
133 [a.axis('off') for a in ax.ravel()]
134 fig.subplots_adjust(wspace=0.05, hspace=0.1)
135 plt.savefig(os.path.join(OUTPUT_DIR, ''.join((fig-, basename, .png)))))
136 plt.show()
137
138 mccs.append((filename, m_FA, m_PF, m_PFA, m, m2, m_st))
139 precs.append((filename, p_FA, p_PF, p_PFA, p, p2, p_st))
140
141 runlog = { 'mccs': mccs, 'precs': precs}
142 with open(os.path.join(OUTPUT_DIR, 'runlog.json'), 'w') as f:
143     json.dump(runlog, f, indent=True)

```

listings/margin_add.py

```

1 # coding: utf-8
2 plt.imshow(basic.py)
3 plt.imshow(img.filled(0))
4 plt.show()
5 from skimage.filters import scharr
6 scharr(img)
7 plt.imshow(_)
8 plt.show()
9 scharr(img, mask=img.mask)

```

```

10 plt.imshow(_)
11 plt.show()
12 scharr(img, mask='img.mask')
13 scharr(img, mask=~img.mask)
14 plt.imshow(_)
15 plt.show()
16 from skimage.filters.rank import enhance_contrast_percentile as ecp
17 from skimage.morphology import selem
18 scales = np.logspace(-1.5, 3.5, base=2, num=20)
19 beta = 0.35
20 gamma = 0.5
21 F = np.stack([frangi_from_image(img, sigma, beta, gamma, dark_bg=False, dilation_radius=20
22 plt.imshow(F.max(axis=0))
23 plt.show()
24 plt.imshow(F[:-2].max(axis=0))
25 plt.show()
26 plt.imshow(F[:-2].max(axis=-1))
27 plt.imshow(F[:-2].max(axis=0))
28 plt.show()
29 plt.imshow(F[:-2].max(axis=0))
30 ecp(F[:-2].max(axis=0), selem=disk(2))
31 from skimage.morphology import disk
32 ecp(F[:-2].max(axis=0), selem=disk(2))
33 plt.imshow(_)
34 plt.show()
35 ecp(F[:-2].max(axis=0), selem=disk(3))
36 plt.show()
37 plt.imshow(_)
38 plt.show()
39 ecp(F[:-2].max(axis=0), selem=disk(3))
40 ef = -
41 plt.imshow(ef > .4)
42 plt.show()
43 ef.max()
44 ef.min()
45 .4*255
46 plt.imshow(ef > 102)
47 plt.show()
48 plt.imshow(ef > 102)
49 from placenta import open_typefile
50 open_typefile(filename, 'trace')
51 from placenta import open_tracefile
52 open_tracefile(filename)
53 trace = -
54 from scoring import confusion
55 confusion(ef > 102, trace)
56 plt.show()
57 confusion(ef > 102, trace)
58 plt.imshow(_)
59 plt.show()
60 confusion(ef > 102, trace)
61 plt.show()
62 plt.imshow(_)
63 plt.show()
64 mcc
65 confusion(F.max(axis=0) > .4, trace)
66 plt.show()
67 plt.imshow(_)
68 plt.show()
69 fig, ax = plt.subplots(ncols=2, nrows=1)
70 ax[0].imshow(confusion(F.max(axis=0) > .4, trace)[crop])
71 ax[1].imshow(confusion(ef > 102, trace)[crop])
72 plt.show()
73 fig, ax = plt.subplots(ncols=2, nrows=1)

```

```

74 ax[0].imshow(confusion(F.max(axis=0) > .4, trace)[crop])
75 ax[1].imshow(confusion(ef > 102, trace)[crop])
76 [a.axis('off') for a in ax]
77 plt.show()
78 fig, ax = plt.subplots(ncols=2, nrows=1)
79 ax[0].imshow(confusion(F[:-2].max(axis=0) > .4, trace)[crop])
80 ax[1].imshow(confusion(ef > 102, trace)[crop])
81 [a.axis('off') for a in ax]
82 plt.show()
83 F = np.stack([frangi_from_image(img, sigma, beta, gamma, dark_bg=False, dilation_radius=20
84 plt.imshow(np.abs(F).max(axis=0)[crop])
85 plt.show()
86 absF = np.abs(F).max(axis=0)
87 eps(absF, selem=disk(3))
88 epc(absF, selem=disk(3))
89 ecp(absF, selem=disk(3))
90 plt.imshow(_)
91 plt.show()
92 plt.imshow(epc > .3)
93 ecp(absF, selem=disk(3))
94 eaf = _ / 255
95 eaf
96 plt.imshow(eaf / 255)
97 plt.show()
98 plt.imshow(eaf > .4)
99 plt.show()
100 plt.imshow(eaf > .5)
101 plt.show()
102 plt.imshow(eaf > .5)
103 plt.show()
104 F[F < 0].max(axis=0)
105 F[F < 0].max(axis=0)
106 F[F < 0].max(axis=0)
107 plt.show()
108 plt.imshow(_)
109 -F[F < 0].max(axis=0)
110 (-F[F < 0])
111 plt.imshow(_)
112 (-F*(F<0)).max(axis=0)
113 plt.imshow(_)
114 plt.show()
115 (-F[:12]*(F<0)).max(axis=0)
116 (-F[:]* (F<0))[:12].max(axis=0)
117 (-F*(F<0))[:12].max(axis=0)
118 plt.imshow(_)
119 plt.show()
120 (-F*(F<0))[:10].max(axis=0)
121 plt.show()
122 plt.imshow(_)
123 plt.show()
124 nf = (-F*(F<0))[:10].max(axis=0)
125 plt.imshow(nf)
126 plt.show()
127 plt.imshow(nf+Fmax)
128 plt.imshow(nf+F[:-2].max(axis=0))
129 plt.show()
130 plt.imshow(nf+F[:-4].max(axis=0))
131 plt.show()
132 ecp(nf+F[:-4].max(axis=0), selem=disk(3))
133 ecp(max(nf, F[:-4].max(axis=0)), selem=disk(3))
134 help(np.max)
135 get_ipython().run_line_magic('pinfo', 'np.choose')
136 get_ipython().run_line_magic('pinfo', 'np.take')
137 get_ipython().run_line_magic('pinfo', 'np.maximum')

```

```

138 ecp(np.maximum(nf,F[:-4].max(axis=0)), selem=disk(3))
139 plt.imshow(_)
140 plt.show()
141 ecp(np.maximum(nf,F[:-4].max(axis=0))[crop], selem=disk(3))
142 plt.show()
143 plt.imshow(_)
144 plt.show()
145 np.maximum(nf,F[:-4].max(axis=0))[crop]
146 plt.imshow(_)
147 plt.show()
148 np.maximum(nf,F[:-4].max(axis=0))[crop]
149 plt.imshow(_)
150 plt.show()
151 spine = F[:-4].max(axis=0)
152 margins = nf
153 np.maximum(spine,margins)
154 np.maximum(spine,margins)*~img.mask
155 c = ~
156 plt.imshow(c)
157 plt.show()
158 np.maximum(spine,margins)
159 plt.imshow(_)
160 plt.show()
161 plt.imshow()
162 plt.imshow(c)
163 plt.show()
164 c = ma.masked_array(np.maximum(spine,margins), mask=img.mask)
165 plt.imshow(c)
166 plt.show()
167 dilate_boundary(c, radius=20)
168 plt.imshow(_)
169 plt.show()
170 c = ma.masked_array(np.maximum(spine,margins), mask=img.mask)
171 dilate_boundary(c, 20).filled(0)
172 c =
173 plt.imshow(c)
174 plt.show()
175 epc(c, disk(4))
176 ecp(c, disk(4))
177 plt.imshow(_)
178 plt.show()
179 spine
180 spine*img.mask
181 spine*~img.mask
182 plt.imshow(_)
183 plt.show()
184 dilate_boundary(spine, mask=img.mask, radius=20)
185 plt.imshow(_)
186 plt.show()
187 spine = dilate_boundary(spine, mask=img.mask, radius=20).filled(0)
188 plt.imshow(spine[crop])
189 plt.show()
190 from skimage.segmentation import thin
191 from skimage.morphology import thin
192 thin(spine)
193 plt.imshow(_)
194 plt.show()
195 ecp(spine, disk=3)
196 ecp(spine, disk(3))
197 plt.imshow(_)
198 plt.show()
199 ecp(spine, disk(3) > 102)
200 plt.imshow(_)
201 plt.show()

```

```

202 ecp(spine, disk(3)) > 102
203 plt.imshow(_)
204 plt.show()
205 bspine = ecp(spine, disk(3)) > 102
206 plt.imshow(thin(bspine))
207 plt.show()
208 plt.imshow(np.maximum(margins, bspine))
209 plt.show()
210 plt.imshow(np.maximum(margins, bspine))
211 plt.show()
212 plt.imshow(np.maximum(margins*255, bspine))
213 plt.show(_)
214 plt.imshow(np.maximum(margins*255, bspine))
215 plt.show()
216 plt.imshow(np.maximum(margins, bspine))
217 plt.show()
218 plt.imshow(np.maximum(margins, thin(bspine)))
219 plt.show()
220 margins
221 plt.imshow(margins)
222 plt.show()
223 margins > .2
224 plt.imshow(_)
225 plt.show()
226 margins > .15
227 plt.imshow()
228 plt.imshow()
229 plt.imshow(margins > .1)
230 plt.show()
231 plt.imshow(margins > .05)
232 plt.show()
233 D = np.zeros(margins, np.bool)
234 D = np.zeros(margins.shape, np.bool)
235 from scipy.ndimage import distance_transform_edt
236 from scipy.ndimage import distance_transform_edt as edt
237 D[thin(bspine)] = 1
238 D[:] = 1
239 D[margins>.05] = 0
240 plt.imshow(_)
241 plt.imshow(D)
242 plt.imshow(D)
243 plt.show()
244 edt(D)
245 spine_dists = D.copy()
246 spine_dists[~thin(bspine)] = 0
247 plt.imshow(spine_dists)
248 plt.show()
249 spine_dists = edt(D)
250 spine_dists[~thin(bspine)] = 0
251 plt.imshow(spine_dists)
252 plt.show()
253 spine_dists<=10
254 plt.imshow(_)
255 plt.show()
256 (spine_dists<=10) & (spine_dists > 0)
257 plt.show()
258 plt.imshow()
259 plt.imshow(_255)
260 plt.show()
261 (spine_dists<=6) & (spine_dists > 0)
262 plt.imshow(_)
263 plt.show()
264 from skimage.morphology import binary_dilation
265 help(np.round)

```

```

266 help(np.around)
267 help(np.fix)
268 spine_radii = np.round(spine_dists).astype('int')
269 plt.imshow(spine_radii)
270 plt.show()
271 plt.show()
272 \plt.imshow(spine_radii)
273 np.stack([binary_dilation(spine_radii==r, selem=disk(r)) for r in range(1,10)])
274 dilstack =
275 plt.imshow(dilstack.max(axis=0))
276 plt.show()
277 confusion(dilstack.max(axis=0), trace)
278 plt.show()
279 plt.imshow(_)
280 plt.show()
281 np.stack([binary_dilation(spine_radii==r, selem=disk(r)) for r in range(1,12)])
282 plt.imshow(_)
283 dilstack = np.stack([binary_dilation(spine_radii==r, selem=disk(r)) for r in range(1,12)])
284 plt.imshow(dilstack.any(axis=0))
285 plt.show()
286 confusion(dilstack.any(axis=0), approx)
287 confusion(dilstack.any(axis=0), trace)
288 plt.show()
289 plt.imshow(_)
290 plt.show()
291 dilstack = np.stack([binary_dilation(spine_radii==r, selem=disk(r)) for r in range(1,15)])
292 plt.imshow(confusion(dilstack.any(axis=0), trace))
293 plt.show()
294 from scoring import mcc
295 approx = dilstack.any(axis=0)
296 mcc(approx, trace, bg_mask=img.mask, return_counts=True)
297 counts =_[1]
298 TP, TN, FP, FN = counts
299 precision = TP / (TP+FN)
300 precision
301 precision = TP / (TP+FP)
302 precision

```

listings/make_output_montage.py

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

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
123 def sieve_scales(multiscale, high_percentile, low_percentile, min_size=None,
124                  axis=0):
125     """
126     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('σ', 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, crop=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     assert multiscale.ndim == 3
187
188     if axis in (-1, 2):
189         # this won't change the input, just creates a view
190         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 crop is None:
206             viewable = v
207         else:
208             viewable = v[crop]

```

```

209     if outname is None:
210         plt.imshow(viewable, cmap=cmap, vmin=vmin, vmax=vmax)
211
212         mng = plt.get_current_fig_manager()
213         #mng.window.showMaximized()
214         mng.window.showFullScreen()
215         plt.tight_layout()
216
217         if sigma is not None:
218             plt.title(r' $\sigma$ ={:2f}'.format(sigma))
219             plt.axis('off')
220
221         if show_colorbar:
222             plt.colorbar()
223
224         plt.tight_layout()
225         plt.show()
226         plt.close()
227
228     else:
# save them non interactively with imsave
        plt.imsave(outname, viewable, cmap=cmap, vmin=vmin, vmax=vmax)

```

listings/network_completion.py

```

1 #!/usr/bin/env python3
2
3 from skimage.filters.rank import sum as local_count
4 from skimage.draw import line
5 import numpy as np
6 from itertools import combinations
7
8 def find_endpoints(arr):
9     """Find endpoints of a boolean array (like a thinned approximation)
10
11     Will return any pixels that have between 1 and 2 adjacent neighbors using
12     2-connectivity. 2 neighbors means it's directly connected to one other
13     pixel, and 1 pixel means it's not connected to anything else.
14     """
15     # 2-connectivity
16     selem = np.ones((3,3), np.bool)
17
18     a = arr.astype('bool')
19
20     neighbors = local_count(a.astype('uint8'), selem)
21
22     return arr & ((neighbors==1)|(neighbors==2))
23
24 def categorize_endpoints(arr):
25     """Find endpoints, as in find_endpoints, but label each with the *kind*
26     of endpoint it is describing its orientation. Is it 2 connected by:
27
28     - - - <--- top (0)
29     - . - <--- middle (1)
30     - - - <--- bottom (2)
31
32     | | | ----- right (0)
33     | | | ----- middle (1)
34     | | | ----- left (2)
35
36     these are the same as the row/column indices into the 3x3 square. so
37     if a pixel's only neighboring pixel is to its immediate right, then
38     the label for the pixel is (1,0). If the endpoint is isolated (i.e. it
39     has no neighbors) then its label is (1,1) (since 1 is the more forgivable

```

```

40 connection condition).
41
42 The idea is that
43 - a left-connected pixel should not connect with another left-connected
44   pixel
45 - a right-connected pixel should not connect with another right-connected
46   pixel.
47 - a bottom-connected pixel should not be connected with another
48   bottom-connected pixel
49 - a top-connected pixel should not be connected with another
50   top-connected pixel
51
52 There are other ideas (that a connection shouldn't be made through a point
53 where the frangi score is 0; that a connection shouldn't be made that
54 crosses an established part of the skeleton; that connections shouldn't
55 be made between two endpoints whose distance exceeds some specified amount)
56 that cannot be cast in terms of these labels, but this does reduce the
57 number of possible connections substantially.
58
59 this returns two lists of tuples, where the first contains the indices of
60 each endpoint, and the corresponding place in the second list contains
61 its label (also a tuple)
62 """
63
64 # still get a list of endpoints because this is faster.
65 selem = np.ones((3,3), np.bool)
66 a = arr.astype('bool')
67 endpoint_list = list()
68 label_list = list()
69 bd = np.array([[1,1,1],[1,0,1],[1,1,1]], dtype='bool')
70 for px, py in zip(*np.where(a)):
71     local = a[px-1:px+2, py-1:py+2]
72
73     if local.sum() == 1:
74         endpoint_list.append((px,py))
75         label_list.append((1,1))
76     elif local.sum() == 2:
77         endpoint_list.append((px,py))
78         label_list.append(tuple(map(int, np.where(local&bd))))
79     else:
80         pass # this isn't an endpoint
81
82 return endpoint_list, label_list
83
84
85 def mean_colored_connections(arr, scores=None, double_connect=True):
86     """Lines that connect nearest endpoints, colored by the mean value of
87     that line through the matrix scores
88
89     if scores is None, just return a boolean array
90     """
91
92     # list of endpoints as ordered pairs
93     a = arr.astype('bool') # cast to bool first to prevent weirdness
94     neighbors = local_count(a.astype('uint8'), np.ones((3,3), np.bool))
95     endpoints = a & ((neighbors==1)|(neighbors==2))
96     endpoint_list = list(zip(*np.where(endpoints)))
97
98     lines = np.zeros(endpoints.shape) # but type float
99     for p0 in endpoint_list:
100         if (neighbors[p0] == 2) or not double_connect:
101             nearest = endpoint_list[np.argmin([
102                 (p0[0]-p[0])**2 + (p0[1] - p[1])**2
103                 if p0!=p else 10000 for p in endpoint_list
104             ])]

```

```

104     #print(p0, nearest)
105     line_to_add = line(*p0, *nearest)
106
107     if scores is None:
108         lines[line_to_add] = 1
109     else:
110         lines[line_to_add] = scores[line_to_add].mean()
111         #print(scores[line_to_add].mean())
112
113     else:
114         # same thing but get the two nearest
115         nearest, second, __ = np.argpartition([
116             (p0[0]-p[0])**2 + (p0[1] - p[1])**2
117             if p0!=p else 10000 for p in endpoint_list], 2)
118         nearest = endpoint_list[nearest]
119         second = endpoint_list[second]
120         print(p0, nearest, second, '<-----')
121         line_to_add = line(*p0, *nearest)
122         line2_to_add = line(*p0, *second)
123
124         if scores is None:
125             lines[line_to_add] = 1
126             lines[line2_to_add] = 1
127         else:
128             lines[line_to_add] = scores[line_to_add].mean()
129             lines[line2_to_add] = scores[line2_to_add].mean()
130
131     return lines
132
133
134 def connect_iterative(arr, scores=None, max_iterations=10):
135     """Lines that connect nearest endpoints, colored by the mean value of
136         that line through the matrix scores
137
138     if scores is None, just return a boolean array
139     """
140     # list of endpoints as ordered pairs
141     a = arr.astype('bool') # cast to bool first to prevent weirdness
142     bad_pairs = list()
143     lines = a.copy() # but type float
144     for i in range(max_iterations):
145         neighbors = local_count(a.astype('uint8'), np.ones((3,3), np.bool))
146         endpoints = a & ((neighbors==1)|(neighbors==2))
147         endpoint_list = list(zip(*np.where(endpoints)))
148         print(f"at start of {i}th iteration, {len(bad_pairs)} many bad pairs")
149         old_network_size = lines.sum()
150         for k, p0 in enumerate(endpoint_list):
151             candidates = [p for p in endpoint_list[k:]
152                           if (p,p0) not in bad_pairs and
153                           (p0,p) not in bad_pairs]
154             if not candidates:
155                 continue
156             nearest = candidates[np.argmin([
157                 (p0[0]-p[0])**2 + (p0[1] - p[1])**2
158                 if p0!=p else 10000 for p in candidates]
159             )]
160             line_to_add = line(*p0, *nearest)
161
162             if scores is None:
163                 lines[line_to_add] = 1
164             else:
165                 mean_score = scores[line_to_add].mean()
166                 if (mean_score < .3) or (scores[line_to_add]==0).any():
167                     bad_pairs.append((p0,nearest))

```

```

168         continue
169     else:
170         lines[line_to_add] = 1
171         #print(scores[line_to_add].mean())
172     new_network_size = lines.sum()
173     size_diff = new_network_size - old_network_size
174     print(f'after {i}th iteration, ', size_diff, 'pixels were added')
175     if size_diff == 0:
176         break
177     else:
178         print(f'returned after {max_iterations} iterations')
179
180     return lines
181
182 def _euclidean_distance(p0,p1):
183     return np.sqrt((p0[0]-p1[0])**2 + (p0[1]-p1[1])**2)
184
185
186 def connect_iterative_by_label(arr, scores, max_iterations=10,
187                               max_dist=None):
188     """Lines that connect nearest endpoints, colored by the mean value of
189     that line through the matrix scores
190
191     if scores is None, just return a boolean array
192     """
193
194     # list of endpoints as ordered pairs
195     a = arr.astype('bool') # cast to bool first to prevent weirdness
196     matched = list()
197
198
199     endlist, endlabs = categorize_endpoints(arr)
200     N_ends = len(endlist)
201     # handshake matrix, matchable[j,k]==0 if endlist[j] and endlist[k]
202     # cannot be connected, otherwise 1. this is upper triangular
203     # on the first diagonal, so make sure k > j
204     matchable = np.triu(np.ones((len(endlist), len(endlist))), k=1)
205
206     # distances between these two points with lazy filling. could be overloaded
207     # into matchable but let's keep it separate
208     dists = np.zeros(matchable.shape, dtype=np.float64)
209     print(f'there are {len(endlist)} endpoints')
210
211     print('checking compatibility of endpoints by labels and calculating distances')
212
213     for j, jlab in enumerate(endlabs):
214
215         # still calculate the distances below
216         #if jlab == (1,1):
217             #    continue # this can connect with anything
218
219         for k, klab in enumerate(endlabs[j+1:], j+1):
220             if jlab[0] != 1:
221                 if jlab[0] == klab[0]:
222                     matchable[j,k] = 0
223                     continue
224             if jlab[1] != 1:
225                 if jlab[1] == klab[1]:
226                     matchable[j,k] = 0
227                     continue
228             # if they are matchable, let's find the distance between them
229             dist = _euclidean_distance(endlist[j], endlist[k])
230             if max_dist is not None:
231                 if dist > max_dist:

```

```

232         matchable[j,k] = 0
233     else:
234         dists[j,k] = dist
235     else:
236         dists[j,k] = dist
237
238 lines = a.copy() # but type float
239
240 print(f"removed {N_ends*(N_ends-1)/2 - matchable.sum():n} "
241       "pairs of endpoints from consideration "
242       f'(out of {N_ends*(N_ends-1) / 2:n} possible pairs')
243
244 size_before = matchable.sum()
245 for j, pj in enumerate(endlist):
246     for k, pk in enumerate(endlist[j+1:], j+1):
247         if not matchable[j,k]:
248             continue
249         l = line(*pj, *pk)
250         # lines will have the two endpoints so sum is at least 2
251         if (lines[l].sum() > 2) or (scores[l]==0).any():
252             matchable[j,k] = 0
253             dists[j,k] = 0
254
255 size_after = matchable.sum()
256
257 print(f"removed {size_before-size_after:n} more pairs",
258       "by avoiding crossings/zero score paths")
259
260 for j, pj in enumerate(endlist):
261     mean_score = lambda p: scores[line(*pj,*p)].mean()
262     point_scores = [(k, pk, mean_score(pk))
263                     for k,pk in enumerate(endlist[j+1:], j+1)
264                     if matchable[j,k]]
265
266     if not point_scores:
267         continue
268     s = sorted(point_scores, key=lambda x: x[2])
269     p_best = s[-1][1]
270     lines[line(*pj, *p_best)] = 1
271
272 return lines
273 #for i in range(max_iterations):
274 #    old_network_size = lines.sum()
275
276 #    print(f"at start of {i}th iteration:")
277 #    print(f"\t{matchable.any(axis=1).sum()} unmatched endpoints")
278 #    print(f"\t{old_network_size} is the size of the network")
279
280 #    for j, p0 in enumerate(endlist):
281
282 #        if not matchable[k,:].any():
283 #            continue # we can't connect this endpoint or it's already
284 #                    # been connected to
285
286 #        for k, p in enumerate(endlist[j+1:], j+1):
287 #            if not matchable[j,k]:
288 #                continue
289
290 #            if not candidates:
291 #                continue # this endpoint can't connect to anything anymore
292
293 #            for candidate in candidate:
294 #                pass
295 #                #nearest = candidates[np.argmin([

```

```

296     #             #
297     #             #
$$(p0[0]-p[0])^2 + (p0[1] - p[1])^2$$

298     #             if p0!=p else 10000 for p in candidates]
299     #         ]
300     #line_to_add = line(*p0, *nearest)
301
302     #if scores is None:
303     #    lines[line_to_add] = 1
304     #else:
305     #    mean_score = scores[line_to_add].mean()
306     #    if (mean_score < .3) or (scores[line_to_add]==0).any():
307     #        bad_pairs.append((p0,nearest))
308     #        continue
309     #    else:
310     #        lines[line_to_add] = 1
311     #        #print(scores[line_to_add].mean())
312     #new_network_size = lines.sum()
313     #size_diff = new_network_size - old_network_size
314     #print(f'after {i}th iteration,', size_diff, 'pixels were added')
315     #if size_diff == 0:
316     #    break
317     #else:
318     #    old_network_size = new_network_size
319     #else:
320     #    print(f'returned after {max_iterations} iterations')
321
322     #return lines
323
324 def colored_connections_any_nonzero(arr, scores):
325     """Lines that connect nearest endpoints, colored by the mean value of
326     that line through the matrix scores
327
328     WOULD BE BETTER IF YOU USED SCORES FROM THE APPROPRIATE SCALE
329     if scores is None, just return a boolean array
330     """
331     # list of endpoints as ordered pairs
332     a = arr.astype('bool') # cast to bool first to prevent weirdness
333     neighbors = local_count(a.astype('uint8'), np.ones((3,3), np.bool))
334     endpoints = a & ((neighbors==1)|(neighbors==2))
335     endpoint_list = list(zip(*np.where(endpoints)))
336
337     lines = np.zeros(endpoints.shape) # but type float
338     #good_lines = list()
339     for i, p0 in enumerate(endpoint_list):
340         for j, p1 in enumerate(endpoint_list[i+1:], i+1):
341             #nearest = endpoint_list[np.argmax([
342             #            (p0[0]-p[0])^2 + (p0[1] - p[1])^2
343             #            if p0!=p else 10000 for p in endpoint_list]
344             #            ])
345
346             candidate = line(*p0, *p1)
347
348             if (scores[candidate] > 0).all():
349                 lines[candidate] = scores[candidate].mean()
350
351     return lines
352
353
354 def colored_connections_max_path(arr, scores):
355     """Lines that connect nearest endpoints, colored by the mean value of
356     that line through the matrix scores
357
358     WOULD BE BETTER IF YOU USED SCORES FROM THE APPROPRIATE SCALE

```

```

360     if scores is None, just return a boolean array
361     """
362     # list of endpoints as ordered pairs
363     a = arr.astype('bool') # cast to bool first to prevent weirdness
364     neighbors = local_count(a.astype('uint8'), np.ones((3,3), np.bool))
365     endpoints = a & ((neighbors==1)|(neighbors==2))
366     endpoint_list = list(zip(*np.where(endpoints)))
367
368     lines = np.zeros(endpoints.shape) # but type float
369     #good_lines = list()
370     for i, p0 in enumerate(endpoint_list):
371         mean_score = lambda p: scores[line(*p0,*p)].mean() if ((p0!=p) and np.all(scores[line(*p0,*p)]!=0)) else 0
372         point_scores = [(p, mean_score(p)) for p in endpoint_list[i+1:]]
373         point_scores = [(p, score) for p, score in point_scores if score!=0]
374         s = sorted(point_scores, key=lambda x: x[1])
375         if i<5:
376             print(f'{i}:', s)
377             for p, score in s:
378                 path = line(*p0, *p)
379                 lines[path] = score
380
381     return lines
382
383 if __name__ == "__main__":
384
385
386
387     import matplotlib.pyplot as plt
388     import matplotlib as mpl
389     from skimage.util import img_as_float
390     from skimage.io import imread
391     from placenta import (get_named_placenta, list_by_quality, cropped_args,
392                           img_as_float)
393
394     from frangi import frangi_from_image
395     import numpy.ma as ma
396     from hfft import fft_gradient, fft_hessian, fft_gaussian
397     from merging import nz_percentile
398     from plate_morphology import dilate_boundary
399     import os.path, os
400     from skimage.morphology import thin
401     from postprocessing import dilate_to_rim
402     from scoring import confusion, mcc
403     from placenta import open_tracefile, open_typefile
404     from preprocessing import inpaint_hybrid
405     from placenta import measure_ncs_markings, add_ucip_to_mask
406
407     import json
408
409     mccs = list()
410     precs = list()
411
412     scales = np.logspace(-1.5, 3.5, num=20, base=2)
413     threshold = .15
414     neg_threshold = 0.001
415     max_pos_scale = -6
416     max_neg_scale = 2
417     beta = 0.10
418     gamma = 1.0
419
420     cm = mpl.cm.plasma
421     #cmscales = mpl.cm.magma
422     cm.set_bad('k', 1) # masked areas are black, not white
423     #cmscales.set_bad('w', 1)

```

```

424
425 QUALITY = 3
426 for filename in list_by_quality(QUALITY):
427
428     cimg = open_typefile(filename, 'raw')
429     ctrace = open_typefile(filename, 'ctrace')
430     trace = open_tracefile(filename)
431     img = get_named_placenta(filename)
432     crop = cropped_args(img)
433     ucip = open_typefile(filename, 'ucip')
434     img = inpaint_hybrid(img)
435
436     # make the size of figures more consistent
437     if img[crop].shape[0] > img[crop].shape[1]:
438         # and rotating it would be fix all this automatically
439         cimg = np.rot90(cimg)
440         ctrace = np.rot90(ctrace)
441         trace = np.rot90(trace)
442         img = np.rot90(img)
443         ucip = np.rot90(ucip)
444         crop = cropped_args(img)
445
446     ucip_midpoint, resolution = measure_ncs_markings(ucip)
447     ucip_mask = add_ucip_to_mask(ucip_midpoint, radius=60, mask=img.mask)
448
449     name_stub = filename.rstrip('.png').strip('T-')
450
451
452     F = np.stack([frangi_from_image(img, sigma=sigma, beta=beta, gamma=gamma, dark_bg=False,
453                                         signed_frangi=True, dilation_radius=20,
454                                         rescale_frangi=True)
455                                         for sigma in scales])
456     # need to fix this in the signed_frangi logic
457     F = F * ~dilate_boundary(None, mask=img.mask, radius=20)
458
459     Fmax = (F * (F > 0))[:max_pos_scale].max(axis=0)
460     Fneg = -(F * (F < 0))[:max_neg_scale].min(axis=0)
461
462     approx = Fmax > threshold
463     rim_approx = (Fneg > neg_threshold)
464     skel = thin(approx)
465     completed = connect_iterative_by_label(skel, Fmax, max_dist=100)
466     completed_dilated = dilate_to_rim(completed, rim_approx, max_radius=10)
467     approx_dilated = dilate_to_rim(approx, rim_approx, max_radius=10)
468
469     network = np.maximum(skel * 3., (completed & ~skel) * 2)
470     network = np.maximum(network, rim_approx * 1.)
471
472
473     precision = lambda t: int(t[0]) / int(t[0] + t[2])
474
475     mcc_FA, counts_FA = mcc(approx, trace, bg_mask=img.mask, return_counts=True)
476     mcc_FAD, counts_FAD = mcc(approx_dilated, trace, bg_mask=img.mask, return_counts=True)
477     mcc_NCD, counts_NCD = mcc(completed_dilated, trace, bg_mask=img.mask, return_counts=True)
478
479     prec_FA = precision(counts_FA)
480     prec_FAD = precision(counts_FAD)
481     prec_NCD = precision(counts_NCD)
482
483     mccs.append((name_stub, mcc_FA, mcc_FAD, mcc_NCD))
484     precs.append((name_stub, prec_FA, prec_FAD, prec_NCD))
485
486     fig, ax = plt.subplots(nrows=2, ncols=3, figsize=(20, 12))
487     A = ax.ravel()

```

```

488
489     A[0].imshow(cimg[crop])
490     A[0].set_title(name_stub)
491
492     A[1].imshow(ma.masked_where(Fmax==0,Fmax)[crop], cmap=cm, vmin=0, vmax=1)
493     A[1].set_title(rf'Vmax, $\beta = \text{beta} : .2f, \gamma = \text{gamma} : .3f'$ )
494
495     A[2].imshow(network[crop], cmap=plt.cm.magma)
496     A[2].set_title('skeleton, completed network, and rim_approx')
497
498     A[3].imshow(confusion(approx, trace, bg_mask=img.mask)[crop])
499     A[3].set_title(fr'fixed  $\alpha = \text{threshold} : .2f$ ', loc='left')
500     A[3].set_title(f'MCC: {mcc_FA:.2f}\n' f'precision: {prec_FA:.2%}', loc='right')
501
502     A[4].imshow(confusion(approx_dilated, trace, bg_mask=img.mask)[crop])
503     A[4].set_title(fr'dilate_to_rim  $\alpha = \text{threshold} : .2f$ ', loc='left')
504     A[4].set_title(f'MCC: {mcc_FAD:.2f}\n' f'precision: {prec_FAD:.2%}', loc='right')
505
506     A[5].imshow(confusion(completed_dilated, trace, bg_mask=img.mask)[crop])
507     A[5].set_title(fr'network_completed, dilated', loc='left')
508     A[5].set_title(f'MCC: {mcc_NCD:.2f}\n' f'precision: {prec_NCD:.2%}', loc='right')
509
510
511 [a.axis('off') for a in A]
512 fig.tight_layout()
513 plt.show()
514
515
516
517
518 runlog = { 'mccs':mccs,
519             'precs':precs,
520             'scales':list(scales),
521             'beta':beta,
522             'gamma':gamma,
523             'max_pos_scale':max_pos_scale,
524             'max_neg_scale':max_neg_scale,
525             'threshold':threshold,
526             'neg_threshold':neg_threshold
527         }
528
529
530
531 with open(f'output/network_completion_{QUALITY}.json', 'w') as f:
532     json.dump(runlog, f, indent=True)

```

listings/pcsvn.py

```

1 #!/usr/bin/env python3
2 """
3 the contents of this file should be split up between frangi (multiscale)
4 and scoring/output
5 """
6
7 from placenta import get_named_placenta
8 from diffgeo import principal_directions
9 from frangi import frangi_from_image
10 from skimage.util import img_as_float
11 import numpy as np
12 from preprocessing import inpaint_hybrid
13
14 from merging import nz_percentile

```

```

15
16 from plate_morphology import dilate_boundary
17
18 import matplotlib.pyplot as plt
19 import matplotlib as mpl
20 import numpy.ma as ma
21
22 import os.path
23 import json
24 import datetime
25
26
27 def make_multiscale(img, scales, beta=0.5, gamma=0.5, c=None, dark_bg=True,
28                     find_principal_directions=False, dilate_per_scale=True,
29                     signed_frangi=False, kernel=None, verbose=True,
30                     rescale_frangi=False, gradient_filter=False):
31     """Returns an ordered list of dictionaries for each scale of Frangi info.
32
33     beta, gamma, and c can all be vectors as long as scales or constants
34     if c is None it will be set.
35
36     Each element in the output contains the following info:
37     {'sigma': sigma,
38      'beta': beta,
39      'gamma': gamma,
40      'H': hesh,
41      'F': targets,
42      'k1': k1,
43      'k2': k2,
44      't1': t1, # if find_principal_directions
45      't2': t2 # if find_principal_directions
46    }
47
48     is it necessary to lug all this shit around?
49     """
50
51     # store results of each scale (create as empty list)
52     multiscale = list()
53
54     img = ma.masked_array(img_as_float(img), mask=img.mask)
55
56     vectorize = lambda x: np.repeat(x, len(scales)) if (x is None or np.isscalar(x)) else x
57
58     # vectorize any scalar inputs here
59     beta = vectorize(beta)
60     gamma = vectorize(gamma)
61     c = vectorize(c)
62     print('finding multiscale targets ', end='')
63     for i, (sigma, b, g, cx) in enumerate(zip(scales, beta, gamma, c)):
64         print('σ', end=' ', flush=True)
65
66         if dilate_per_scale:
67             if sigma > 20:
68                 radius = int(2*sigma)
69             elif sigma < 3:
70                 radius = 12
71             else:
72                 radius = int(4*sigma)
73         else:
74             radius = None
75
76         targets, this_scale = frangi_from_image(img, sigma, beta=b, gamma=g,
77                                               c=cx, dark_bg=dark_bg,

```

```

79
80
81
82
83
84
85         dilation_radius=radius,
86         kernel=kernel,
87         signed_frangi=signed_frangi,
88         return_debug_info=True,
89         rescale_frangi=rescale_frangi,
90         gradient_filter=gradient_filter)
91
92     if find_principal_directions:
93         # principal directions should only be computed for critical regions
94         # this mask is where PD's will *NOT* be calculated
95         # is targets a masked array?
96         cutoff = nz_percentile(targets, 80)
97         pd_mask = np.bitwise_or(targets < cutoff, img.mask).filled(1)
98         percent_calculated = (pd_mask.size - pd_mask.sum()) / pd_mask.size
99
100    if verbose:
101        print(f"finding PD's for {percent_calculated:.2%} of image"
102              f"anything above vesselness score {cutoff:.6f}")
103    t1, t2 = principal_directions(img, sigma=sigma, H=this_scale['H'],
104                                  mask=pd_mask)
105
106    # add them to this scale's output
107    this_scale['t1'] = t1
108    this_scale['t2'] = t2
109
110 else:
111     if verbose:
112         print('skipping principal direction calculation')
113
114     # store results as a list of dictionaries
115     multiscale.append(this_scale)
116
117 def extract_pcsvn(img, filename, scales, beta=0.5, gamma=0.5, c=None,
118                    dark_bg=True, dilate_per_scale=True, verbose=True,
119                    generate_json=True, output_dir=None, kernel=None,
120                    signed_frangi=False, rescale_frangi=False,
121                    gradient_filter=False):
122     """Run PCCSVN extraction on the sample given in the file.
123
124     Despite the name, this simply returns the Frangi filter responses at
125     each provided scale without explicitly making any decisions about what
126     is or is not part of the PCCSVN.
127
128     As a matter of fact, this function currently just is a wrapper for
129     make_multiscale that logs some output
130     The original main use of this function has kind of bled into
131     extract_NCS_pcsvn.py. that needs fixing. You should load the image
132     outside of this function, do post processing there, pass it inside here
133     with a dictionary of things to add to the json file
134
135     """
136
137     # Multiscale Frangi Filter#####
138     # output is a dictionary of relevant info at each scale
139     multiscale = make_multiscale(img, scales, beta=beta, gamma=gamma, c=None,
140                                  find_principal_directions=False,
141                                  dilate_per_scale=dilate_per_scale,
142                                  kernel=kernel, signed_frangi=signed_frangi,

```

```

143         dark_bg=dark_bg, verbose=verbose,
144         rescale_frangi=rescale_frangi,
145         gradient_filter=gradient_filter)
146
147     # extract these for logging
148     c = [scale['c'] for scale in multiscale]
149     border_radii = [scale['border_radius'] for scale in multiscale]
150
151     # ignore targets too close to edge of plate
152     # wait are we doing this twice?
153     if dilate_per_scale:
154         if verbose:
155             print('trimming collars of plates (per scale)')
156
157         for i in range(len(multiscale)):
158             f = multiscale[i]['F']
159             # twice the buffer (be conservative!)
160             radius = int(multiscale[i]['sigma']**2)
161             if verbose:
162                 print('dilating plate for radius={}'.format(radius))
163             f = dilate_boundary(f, radius=radius, mask=img.mask)
164             # get rid of mask
165             multiscale[i]['F'] = f.filled(0)
166     else:
167         for i in range(len(multiscale)):
168             # get rid of mask
169             multiscale[i]['F'] = multiscale[i]['F'].filled(0)
170     # Make Composite#####
171
172     # get a M x N x n_scales array of Frangi targets at each level
173     F_all = np.dstack([scale['F'] for scale in multiscale])
174
175     if generate_json:
176
177         time_of_run = datetime.datetime.now()
178         timestring = time_of_run.strftime("%y%m%d_%H%M")
179
180         # numpy arrays have to be turned into lists first
181         vectorize = lambda x: x if x is None or np.isscalar(x) else list(x)
182
183         logdata = {'time': timestring,
184                    'filename': filename,
185                    'betas': vectorize(beta),
186                    'gammas': vectorize(gamma),
187                    'c': vectorize(c),
188                    'sigmas': list(scales)}
189
190
191     if dilate_per_scale:
192         logdata['border_radii'] = border_radii
193
194     if output_dir is None:
195         output_dir = 'output'
196
197     base = os.path.basename(filename)
198     *base, suffix = base.split('.')
199     dumpfile = os.path.join(output_dir,
200                            ''.join(base) + '_' + str(timestring)
201                            + '.json')
202
203     with open(dumpfile, 'w') as f:
204         json.dump(logdata, f, indent=True)
205
206     return F_all, dumpfile

```

```

207
208 def get_outname_lambda(filename, output_dir=None, timestring=None):
209 """
210     return a lambda function which can build output filenames
211 """
212
213 if output_dir is None:
214     output_dir = 'output'
215
216 base = os.path.basename(filename)
217 *base, suffix = base.split('.')
218
219 if timestring is None:
220     time_of_run = datetime.datetime.now()
221     timestring = time_of_run.strftime("%y%m%d_%H%M")
222
223 outputstub = ''.join(base) + '_' + timestring + '_{}.' + suffix
224 return lambda s: os.path.join(output_dir, outputstub.format(s))
225
226
227 def _build_scale_colormap(N_scales, base_colormap, basecolor=(0,0,0,1)):
228 """
229     returns a mpl.colors.ListedColormap with N samples,
230     based on the colormap named "default_colormap" (a string)
231
232     the N colors are given by the default colormap, and
233     basecolor (default black) is added to map to 0.
234     (you could change this, for example, to (1,1,1,1) for white)
235
236     reversed colormaps often work better if the basecolor is black
237     you should make sure there's good contrast between the basecolor
238     and the first color in the colormap
239 """
240
241 map_range = np.linspace(0, 1, num=N_scales)
242 colormap = plt.get_cmap(base_colormap)
243 colorlist = colormap(map_range)
244
245 # add basecolor as the first entry
246 colorlist = np.vstack((basecolor, colorlist))
247
248 return mpl.colors.ListedColormap(colorlist)
249
250
251 def scale_label_figure(whereis, scales, savefilename=None,
252                         crop=None, show_only=False, image_only=False,
253                         base_cmap='viridis_r', save_colorbar_separate=False,
254                         basecolor=(0, 0, 0, 1), savecolorbarfile=None,
255                         output_dir=None):
256 """
257     crop is a slice object.
258     if show_only, then just plt.show (interactive).
259     if image_only, then this will *not* be printed with the colorbar
260
261     if save_colormap_separate, then the colormap will be saved as a separate
262     file
263
264     if crop is not None:
265         whereis = whereis[crop]
266
267 fig, ax = plt.subplots() # not sure about figsize

```

```

271 N = len(scales) # number of scales / labels
272 tabemap = _build_scale_colormap(N, base_cmap, basecolor)
273
274 if image_only:
275     plt.imsave(savefilename, wheres, cmap=tabemap, vmin=0, vmax=N)
276     plt.close()
277 else:
278     imgplot = ax.imshow(wheres, cmap=tabemap, vmin=0, vmax=N)
279     # discrete colorbar
280     cbar = plt.colorbar(imgplot)
281
282     # this is apparently hackish, beats me
283     tick_locs = (np.arange(N+1) + 0.5)*(N-1)/N
284
285     cbar.set_ticks(tick_locs)
286     # label each tick with the sigma value
287     scalelabels = [r" $\sigma$ ={:2f}".format(s) for s in scales]
288     scalelabels.insert(0, "(no match)")
289     # label with their sigma value
290     cbar.set_ticklabels(scalelabels)
291     # ax.set_title(r"Scale ( $\sigma$ ) of maximum vesselness ")
292     plt.tight_layout()
293     # plt.savefig(outname('labeled'), dpi=300)
294     if show_only or (savefilename is None):
295         plt.show()
296     else:
297         plt.savefig(savefilename, dpi=300)
298
299 plt.close()
300
301 if save_colorbar_separate:
302     if savecolorbarfile is None:
303         savecolorbarfile = os.path.join(output_dir, "scale_colorbar.png")
304     fig = plt.figure(figsize=(1, 8))
305     ax1 = fig.add_axes([0.05, 0.05, 0.15, 0.9])
306     tick_locs = (np.arange(N+1) + 0.5)*(N-1)/N
307     scalelabels = [r" $\sigma$ ={:2f}".format(s) for s in scales]
308     scalelabels.insert(0, "n/a")
309     cbar = mpl.colorbar.ColorbarBase(ax1, cmap=tabemap,
310                                     norm=mpl.colors.Normalize(vmin=0,
311                                                               vmax=N),
312                                     orientation='vertical',
313                                     ticks=tick_locs)
314     cbar.set_ticklabels(scalelabels)
315     plt.savefig(savecolorbarfile, dpi=300)
316

```

listings/pd_demo_uniscale.py

```

1 #!/usr/bin/env python3
2
3 import numpy as np
4 import numpy.ma as ma
5
6 import matplotlib.pyplot as plt
7 import matplotlib as mpl
8
9 from skimage.io import imread
10 from skimage.util import img_as_float
11
12 from placenta import (get_named_placenta, list_by_quality, cropped_args,
13                         img_as_float)

```

```

14
15 from frangi import frangi_from_image
16 from hfft import fft_gradient, fft_hessian, fft_gaussian
17 from merging import nz_percentile
18 from plate_morphology import dilate_boundary
19 import os.path, os
20
21 from diffgeo import principal_curvatures, principal_directions
22
23
24 filename = list_by_quality(N=1)[0]
25 img = get_named_placenta(filename)
26 crop = cropped_args(img)
27
28 sigma = 1.5
29 img = mimg_as_float(img)
30
31 print('calculating frangi filter')
32
33 f = frangi_from_image(img, sigma=1.5, dark_bg=False, dilation_radius=20,
34                         beta=0.35)
35
36 print('calculating hessian again (oops)')
37 H = fft_hessian(img, sigma=1.5)
38 print('calculating pd where f > .05')
39 v1, v2 = principal_directions(img, 1.5, H=H, mask=(f < 0.05))
40 print('done')
41 vm = ma.masked_array(v2, mask=f<.05)
42
43 # this colormap doesn't have any black in it!
44 cmap = mpl.cm.hsv
45 # so set the mask to black
46
47 cmap.set_bad(color=(0,0,0), alpha=1)
48
49 fig, ax = plt.subplots()
50 cax = ax.imshow(vm[crop], cmap=cmap, vmin=0, vmax=np.pi)
51 ax.axis('off')
52 cbar = fig.colorbar(cax, ticks=[0, np.pi/3, np.pi/2, 2*np.pi/3, np.pi])
53 cbar.ax.set_yticklabels(['0', r'\frac{\pi}{3}', r'\frac{\pi}{2}', r'\frac{2\pi}{3}', r'\pi'])
54
55
56 ax.set_title(r'leading (local) principal direction, \sigma=1.5')
57 fig.tight_layout()
58
59 plt.show() # save manually with the name pd_demo_uniscale.png

```

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 import os.path
17 import os
18 import json
19 from scipy.ndimage import imread
20 from skimage import morphology
21
22 from numpy.ma import is_masked
23 from skimage.color import gray2rgb
24 from skimage.util import img_as_float
25 import matplotlib.pyplot as plt
26 from hfft import fft_gradient
27 from skimage.segmentation import watershed
28 from skimage.morphology import binary_erosion, disk
29 import scipy.ndimage as ndi
30
31
32 def open_typefile(filename, filetype, sample_dir=None, mode=None):
33     """
34         filetype is either 'mask' or 'trace'
35         mask -> 'L' mode
36         trace -> 'RGB' mode
37         use mode keyword to override this behavior (for example if you
38         want a binary trace)
39
40     typefiles that aren't the above will be treated as 'L'
41     """
42     # try to open what the mask *should* be named
43     # this should be done less hackishly
44     # for example, if filename is 'ncs.1029.jpg' then
45     # this would set the maskfile as 'ncs.1029.mask.jpg'
46
47     #if filetype not in ("mask", "trace"):
48     #    raise NotImplementedError("Can only deal with mask or trace files.")
49
50     # get the base of filename and build the type filename
51     *base, suffix = filename.split('.')
52     base = ''.join(base)
53     typefile = '.'.join((base, filetype, suffix))
54
55     if sample_dir is None:
56         sample_dir = 'samples'
57
58     typefile = os.path.join(sample_dir, typefile)
59
60     if mode is not None:
61         if filetype == 'mask':
62             mode = 'L'
63         elif filetype in ('ctrace', 'veins', 'arteries'):
64             mode = 'RGB'
65         else:
66             # handle this if you need to?
67             mode = 'L'
68     try:
69         img = imread(typefile, mode=mode)
70
71     except FileNotFoundError:
72         print('Could not find file', typefile)
73         return None
74
75     return img
76
77

```

```

78 def open_tracefile(base_filename, as_binary=True,
79                     sample_dir=None):
80     """
81
82     ###width parsing is no longer done here. instead, this function
83     should handle the venous/arterial difference.
84
85     this currently only serves to open the RGB traces as binary
86     files instead of RGB, which is processed later
87
88     #TODO: expand this later to handle arterial traces and venous traces
89     INPUT:
90         base_filename: the name of the base file, not the tracefile itself
91         as_binary: if True
92     """
93
94
95     if as_binary:
96         mode = 'L'
97     else:
98         mode = 'RGB'
99
100    T = open_typefile(base_filename, 'trace', sample_dir=sample_dir, mode=mode)
101
102    if as_binary:
103
104        return np.invert(T != 0)
105
106    else:
107        return T
108
109 def mimg_as_float(mimg):
110
111     if not ma.is_masked(mimg):
112
113         return img_as_float(mimg)
114
115     else:
116         return ma.masked_array(img_as_float(mimg.data),
117                               mask=mimg.mask)
118
119 def get_named_placenta(filename, sample_dir=None, masked=True,
120                         maskfile=None, mode='L'):
121     """
122     This function is to be replaced by a more ingenious/natural
123     way of accessing a database of unregistered and/or registered
124     placental samples.
125
126     Parameters
127     -----
128
129     filename: name of file (including suffix?) but NOT directory
130     masked: return it masked.
131     maskfile: if supplied, this use the file will use a supplied 1-channel
132                 mask (where 1 represents an invalid/masked pixel, and 0
133                 represents a valid/unmasked pixel. the supplied image must be
134                 the same shape as the image. if not provided, the mask is
135                 calculated (unless masked=False)
136                 the file must be located within the sample directory
137
138             If maskfile is 'None' then this function will look for
139             a default maskname with the following pattern:
140
141                 test.jpg -> test.mask.jpg

```

```

142             ncs.1029.jpg -> ncs.1029.mask.jpg
143
144 sample_directory: Relative path where sample (and mask file) is located.
145     defaults to './samples'
146
147 if masked is true (default), this returns a masked array.
148
149 NOTE: A previous logical incongruity has been corrected. Masks should have
150 1 as the invalid/background/mask value (to mask), and 0 as the
151 valid/plate/foreground value (to not mask)
152 """
153 if sample_dir is None:
154     sample_dir = 'samples'
155
156 full_filename = os.path.join(sample_dir, filename)
157
158 if mode.lower() in ('g', 'green'):
159     # first channel of RGBA (or RGBAl)
160     raw_img = imread(full_filename)[...,1]
161
162 else:
163     raw_img = imread(full_filename, mode=mode)
164
165 if maskfile is None:
166     # try to open what the mask *should* be named
167     # this should be done less hackishly
168     # for example, if filename is 'ncs.1029.jpg' then
169     # this would set the maskfile as 'ncs.1029.mask.jpg'
170     *base, suffix = filename.split('.')
171     test_maskfile = ''.join(base) + '.mask.' + suffix
172     test_maskfile = os.path.join(sample_dir, test_maskfile)
173
174     try:
175         mask = imread(test_maskfile, mode='L')
176     except FileNotFoundError:
177         print('Could not find maskfile', test_maskfile)
178         print('Please supply a maskfile. Autogeneration of mask',
179               'files is slow and buggy and therefore not supported.')
180         raise
181     #return mask_background(raw_img)
182 else:
183     # set maskfile name relative to path
184     maskfile = os.path.join(sample_dir, maskfile)
185     mask = imread(maskfile, mode='L')
186
187 if filename.startswith('T-BN'):
188     if filename not in FAILS:
189         #print('tightening the mask')
190         raw = open_typefile(filename, 'raw')
191         plate = find_plate_in_raw(raw)
192
193         stuff = ma.masked_array(raw_img, mask=mask)
194         return ma.masked_array(stuff, mask=plate)
195     else:
196         #print('leaving this mask as it is')
197         pass
198
199 return ma.masked_array(raw_img, mask=mask)
200
201 def list_by_quality(quality=0, N=None, json_file=None, return_empty=False):
202 """
203     returns a list of filenames that are of quality ``quality``
204     quality is either "good" or 0

```

```

206             "OK" or 1
207             "fair" or 2
208             "poor" or 3
209
210     N is the number of placentas to return (will return # of placentas
211     of that quality or N, whichever is smaller)
212
213     if json_name is not None just use that filename directly
214
215     if return_empty then silently failing is OK
216     """
217
218     quality_keys = ('good', 'okay', 'fair', 'poor')
219
220     if quality in quality_keys:
221         pass
222     elif quality in (0, 1, 2, 3):
223         quality = quality_keys[quality]
224     else:
225         try:
226             quality = quality.lower()
227         except AttributeError:
228             if return_empty:
229                 return list()
230             else:
231                 print(f'unknown quality {quality}')
232                 raise
233         else:
234             # if no json file is provided, and quality is a string,
235             # just assume it follows a template format
236             if json_file is None:
237                 json_file = f"{quality}-mccs.json"
238
239     # if it's still not provided in the main file, it's in the main file
240     if json_file is None:
241         json_file = 'sample-qualities.json'
242
243     try:
244         with open(json_file, 'r') as f:
245             D = json.load(f)
246     except FileNotFoundError:
247         if return_empty:
248             return list()
249         else:
250             print('cannot find', json_file)
251             raise FileNotFoundError
252
253     if json_file == 'sample-qualities.json':
254         # go one level deep
255         placentas = [k for k in D[quality].keys()]
256     else:
257         placentas = [k for k in D.keys()]
258
259     if N is not None:
260         return placentas[:N]
261     else:
262         return placentas
263
264 def check_filetype(filename, assert_png=True, assert_standard=False):
265     """
266     'T-BN8333878.raw.png' returns 'raw'
267     'T-BN8333878.mask.png' returns 'mask'
268     'T-BN8333878.png' returns 'base'
269

```

```

270
271     if assert_png is True, then raise assertion error if the file
272     is not of type png
273
274     if assert_standard, then assert the filetype is
275     mask, base, trace, or raw.
276
277     etc.
278     """
279     basename, ext = os.path.splitext(filename)
280
281     if ext != '.png':
282         if assert_png:
283             assert ext == '.png'
284
285     sample_name, typestub = os.path.splitext(basename)
286
287     if typestub == '':
288         # it's just something like 'T-BN8333878.png'
289         return 'base'
290     elif typestub in ('.mask', '.trace', '.raw', '.ctrace', '.arteries', '.veins', '.ucip'):
291         # return 'mask' or 'trace' or 'raw'
292         return typestub.strip('.')
293     else:
294         print('unknown filetype:', typestub)
295         print('is it a weird filename?')
296
297         print('warning: lookup failed, unknown filetype:' + typestub)
298
299     return typestub
300
301 def list_placentas(label='T-BN', sample_dir=None):
302     """
303     label is the specifier, basically just ''.startswith()
304
305     only real use is to find all the T-BN* files
306
307     this is hackish, if you ever decide to use a file other than
308     png then this needs to change
309     """
310
311     if sample_dir is None:
312         sample_dir = 'samples'
313
314     if label is None:
315         label = '' # str.startswith('') is always True
316
317     placentas = list()
318
319     for f in os.listdir(sample_dir):
320
321         if f.startswith(label):
322             # oh man they gotta be png files
323             if check_filetype(f) == 'base':
324                 placentas.append(f)
325
326     return sorted(placentas)
327
328
329 def show_mask(img, mask=None, interactive=False, mask_color=None):
330     """
331     rename this color_mask since showing the mask is just a secondary feature
332     show a masked grayscale image with a dark blue masked region
333

```

```

334     custom version of imshow that shows grayscale images with the right
335     colormap and, if they're masked arrays, sets makes the mask a dark blue) a
336     better function might make the grayscale value dark blue (so there's no
337     confusion)
338
339     if interactive, this operates like "plt.imshow"
340     if interactive==False, return the RGB matrix
341
342     if mask provided, add it to the image. (pass img.data instead if you don't
343     want to use the original mask)
344     """
345
346     if mask_color is None:
347         mask_color = (0, 0, 60)
348
349     # if there's no mask at all
350     if (mask is None) and (not is_masked(img)):
351         if interactive:
352             plt.imshow(img, cmap=plt.cm.gray)
353             return # we're done
354         else:
355             # return as an rgb image so output is uniform
356             return gray2rgb(img)
357
358     elif not is_masked(img):
359         # add mask to the image / add to existing mask
360         # if i just rewrite img will it change outside this function?
361         new_img = ma.masked_array(img, mask=mask)
362     else:
363         new_img = img.copy()
364
365     # otherwise, get an RGB array, black where the mask is
366     mimg = gray2rgb(new_img.filled(0))
367
368     # fill masked regions with the mask color
369     mimg[new_img.mask, :] = mask_color
370
371     if interactive:
372         plt.imshow(mimg)
373     else:
374         return mimg
375
376
377 def _cropped_bounds(img, mask=None):
378
379     if mask is not None:
380
381         img = ma.masked_array(img, mask=mask)
382
383         X, Y = (np.argwhere(np.invert(img.mask)).any(axis=k)).squeeze()
384             for k in (0, 1)
385             )
386
387     if X.size == 0:
388         X = [None, None] # these will slice correctly
389     if Y.size == 0:
390         Y = [None, None]
391
392     return Y[0], Y[-1], X[0], X[-1]
393
394
395 def cropped_args(img, mask=None):
396     """
397     get a slice that would crop image

```

```

398     i.e. img[cropped_args(img)] would be a cropped view
399
400
401     x0, x1, y0, y1 = _cropped_bounds(img, mask=None)
402
403     return np.s_[x0:x1, y0:y1]
404
405
406 def cropped_view(img, mask=None):
407     """
408     removes entire masked rows and columns from the borders of a masked array.
409     will return a masked array of smaller size
410
411     don't ask me about data
412
413     the name sucks too
414     """
415
416     # find first and last row with content
417     x0, x1, y0, y1 = _cropped_bounds(img, mask=mask)
418
419     return img[x0:x1, y0:y1]
420
421
422 CYAN = [0, 255, 255]
423 YELLOW = [255, 255, 0]
424
425 def find_plate_in_raw(raw, sigma=.01):
426     g = fft_gradient(raw[...,1], sigma=.01)
427     marks=np.zeros(g.shape, np.int32)
428     marks[0,0] = 1
429     marks[g > g.mean()] = 2
430     #marks[g > np.percentile(g,25)] = 2
431     w = watershed(g,marks)
432
433     eroded = binary_erosion(w==2, disk(15))
434
435     labeled, n_labs = ndi.label(eroded)
436
437     # get largest object (0 is gonna be background)
438     # sort labels by decreasing magnitude
439     labs_by_size = sorted(list(range(1,n_labs+1)),
440                           key=lambda l: np.sum(labeled==l), reverse=True)
441
442     # unless something went horribly wrong
443     plate_index = labs_by_size[0]
444
445     return ~(labeled == plate_index)
446
447
448 FAILS = [
449     "T-BN0687730.png",
450     "T-BN1629357.png",
451     "T-BN2050224.png",
452     "T-BN6381701.png",
453     "T-BN7476220.png",
454     "T-BN7644170.png",
455     "T-BN7767693.png",
456 ]
457
458 def measure_ncs_markings(ucip_img=None, filename=None, verbose=False):
459     """
460     find location of ucip and resolution of image based on input
461     (similar to perimeter layer in original NCS data set

```

```

462
463 Parameters
464 -----
465
466 ucip_img: an RGB ndarray or None
467     The perimeter layer of an NCS sample (colorations according to the
468     tracing protocol). if None, filename must be included. Default is None.
469 filename:
470     the filename of the SAMPLE (not the ucip image file itself)
471
472 Returns
473 -----
474 m : tuple of ints
475     the coordinates of (the center of) of the umbilical cord point
476     (depicted as a yellow dot) in the original image.
477 resolution: a float
478     measured distance between the two cyan dots
479 """
480
481 if ucip_img is None:
482     ucip_img = open_typefile(filename, 'ucip')
483
484 if ucip_img is None:
485     # if it's still none (no file), return None
486     return None, None
487
488 # just in case it's got an alpha channel, remove it
489 img = ucip_img[:, :, 0:3]
490
491 # given the image img (make sure no alpha channel)
492 # find all cyan pixels (there are two boxes of 3 pixels each and we
493 # just want to extract the middle of each
494 if verbose:
495     print('the image size is {}x{}'.format(img.shape[0], img.shape[1]))
496
497 rulemarks = np.all(img == CYAN, axis=-1)
498
499 # turn into two pixels (these should each be shape (18,))
500 X, Y = np.where(rulemarks)
501
502 assert X.shape == Y.shape
503
504 # if they followed the protocol correctly...
505 if X.size == 18:
506     # get the two pixels at the center of each box
507     A, B = (X[4], Y[4]), (X[13], Y[13])
508 else:
509     # dots are a nonstandard size for some reason. this works too.
510     thinned = morphology.thin(rulemarks)
511     X, Y = np.where(thinned)
512     assert(thinned.sum() == 2) # there should be just two pixels now.
513     A, B = (X[0], Y[0]), (X[1], Y[1])
514
515 ruler_distance = np.sqrt((A[0] - B[0])**2 + (A[1] - B[1])**2)
516 if verbose:
517     print(f'one cm equals {ruler_distance} pixels')
518
519 # the umbilical cord insertion point (UCIP) is a yellow circle, radius 19
520 ucipmarks = np.all(img == YELLOW, axis=-1)
521 X, Y = np.where(ucipmarks)
522
523 # find midpoint of the x & y coordinates
524 assert X.max() - X.min() == Y.max() - Y.min()
525 radius = (X.max() - X.min()) // 2

```

```

526     mid = (X.min() + radius, Y.min() + radius)
527
528     if verbose:
529         print('the middle of the UCIP location is', mid)
530         print('the radius outward is', radius)
531         print('the total measurable diameter is', radius*2 + 1)
532
533     return mid, ruler_distance
534
535
536
537 def add_ucip_to_mask(m, radius=100, mask=None, size_like=None):
538     """
539     - m is a tuple (2x1) representing the (coordinate) midpoint of the UCIP
540     - radius around which to dilate the UCIP is the dilation radius as it
541     works in morphology--this is passed directly to skimage.morphology.disk.
542     thus a circle centered at point m with diameter 2*radius + 1
543     - if no mask is supplied, dilate the point in an array of zeros the shape
544     of 'size_like' (would be the same as passing mask=np.zeros_like(size_like))
545
546     Note: this behaves much faster than binary dilation on the point
547     """
548     if mask is None:
549         if size_like is not None:
550             mask = np.zeros_like(size_like)
551         else:
552             raise ValueError("No mask info supplied!")
553
554     # an empty mask (since we need to merge--we don't want to copy the
555     # zeros of the dilated UCIP -- just the ones!)
556     to_add = np.zeros_like(mask)
557
558     # this is way faster than dilating the point in the matrix,
559     # just set this at the centered point
560
561     # doesn't check for out of bounds stuff. use at your own peril
562     D = morphology.disk(radius)
563     to_add[m[0]-radius:m[0]+radius+1, m[1]-radius:m[1]+radius+1] = D
564
565     # merge with supplied mask
566     return mask | to_add
567
568
569 if __name__ == "__main__":
570     """test that this works on an easy image."""
571
572     test_filename = 'barium1.png'
573
574     img = get_named_placenta(test_filename, maskfile=None)
575
576     print('showing the mask of', test_filename)
577     print('run plt.show() to see masked output')
578
579     show_mask(img, interactive=True)

```

listings/plate_test.py

```

1 #!/usr/bin/env python3
2
3 from placenta import (get_named_placenta, list_placentas, open_typefile,
4                       show_mask)
5 import numpy as np

```

```

6 import matplotlib.pyplot as plt
7 from preprocessing import mask_stump
8 import numpy.ma as ma
9 import sys
10 from skimage.exposure import equalize_adapthist
11 from skimage.util import img_as_int
12 from hfft import fft_gradient
13
14 from skimage.segmentation import watershed
15 import scipy.ndimage as ndi
16 from skimage.morphology import binary_erosion, disk
17
18 def find_plate_in_raw(raw, sigma=.01):
19     g = fft_gradient(raw[...,1], sigma=.01)
20     marks=np.zeros(img.shape, np.int32)
21     marks[0,0] = 1
22     marks[g > g.mean()] = 2
23     #marks[g > np.percentile(g,25)] = 2
24     w = watershed(g,marks)
25
26     eroded = binary_erosion(w==2, disk(15))
27
28     labeled, n_labs = ndi.label(eroded)
29
30     # get largest object (0 is gonna be background)
31     # sort labels by decreasing magnitude
32     labs_by_size = sorted(list(range(1,n_labs+1)),
33                           key=lambda l: np.sum(labeled==l), reverse=True)
34
35     # unless something went horribly wrong
36     plate_index = labs_by_size[0]
37
38     return ~(labeled == plate_index)
39
40 FAILS = [
41     "T-BN0687730.png",
42     "T-BN1629357.png",
43     "T-BN2050224.png",
44     "T-BN6381701.png",
45     "T-BN7476220.png",
46     "T-BN7644170.png",
47     "T-BN7767693.png",
48 ]
49
50 for n, filename in enumerate(FAILS):
51     print(filename)
52
53     raw = open_typefile(filename, 'raw')
54     img = get_named_placenta(filename)
55
56     fig, ax = plt.subplots(ncols=3, nrows=1, figsize=(30,12))
57
58     ax[0].imshow(raw)
59     ax[0].set_title(filename)
60
61     plate = find_plate_in_raw(raw)
62     newmask = show_mask(ma.masked_array(img, mask=plate))
63     ax[1].imshow(newmask)
64     ax[2].imshow(plate*1. + img.mask*2, vmin=0, vmax=3)
65
66     plt.show()
67
68     if input('make more?') == 'n':

```

70 sys.exit(0)

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
79         #print("no cutmarks found on image")
80
81         if return_success:
82             return img, False
83         else:
84             return img
85     else:
86         #print("found a cutmark!")
87         pass
88
89     cutmarks = np.nonzero(cutmarks)
90     # get the first pixel of it (we don't need to be too precise here)
91     G = np.all(ucip==(0,255,0), axis=-1) # perimeter elements
92     cutmarks = np.nonzero(thin(B))
93     perimeter = np.nonzero(G)
94
95     cutinds = np.stack(cutmarks).T
96
97     for P in cutinds:
98
99         # consider larger and larger window sizes
100        for W in [100,200,300]:
101            # consider all perimeter elements within these bounds
102
103            rmin, rmax = max(0, P[0]-W), min(img.shape[0], P[0]+W)
104            cmin, cmax = max(0, P[1]-W), min(img.shape[1], P[1]+W)
105            #window = np.s_[rmin:rmax, cmin:cmax]
106
107            # perimeter indices within the window
108            pinds = [(x,y) for x, y in zip(*perimeter)]
109            if x > rmin and x < rmax and y > cmin and y < cmax
110                ]
111
112            if pinds:
113                break #otherwise increase the size of the window
114
115        if pinds:
116
117            # max distance to boundary point in the window
118            # we really only need to keep the largest; deque?
119            dists = sorted([(pp, l2_dist(P,pp)) for pp in pinds],
120                           key=lambda t: t[1])
121            r = 2*int(dists[0][1]) + 1 # get largest radius but closest point
122            P = dists[0][0]
123            B = np.zeros_like(img.mask)
124
125            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     try:
130         B[winx[0]:winx[1] , winy[0]:winy[1]] = D
131     except ValueError:
132         # they're out of bounds so it's a size mismatch. fix it
133         # by starting/ending D index with opposite sign of the initial
134         # p +/- radius that was out of bounds
135         # for example P[0]-r was -9 and everything else was fine
136         # 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 uclip 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 def mask_cuts_watershed(img, uclip, mask_only=False, in_place=False,
153                         return_success=False):
154     """
155
156     this doesn't handle any image, io. just provide the uclip img and the
157     base (masked) image and we'll fix the mask
158
159     uclip is the actual RGB array, not the file. do io elsewhere.
160
161     if mask_only, this will simply return the new mask as a 2D boolean array.
162     Otherwise, it returns a masked_array.
163     The cut region will be added to the img's mask. If you really want just the
164     difference, you'll have to run
165     >>>(cut_mask & ~img.mask) yourself.
166
167     yourself.
168
169     If in_place, this changes the mask of the image directly (but still returns
170     a masked array. If mask_only is True, in_place will automatically be set to
171     False to prevent hideous side effects
172
173     if return_success, this function returns True if there was a cutmark found,
174     otherwise false as a second output
175     """
176
177     # get indices where the blue square indicating center of a cut appears
178     cutmarks = np.all(uclip==(0,0,255), axis=-1)
179
180     if not np.any(cutmarks):
181
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
225         out = new_mask
226
227     elif not in_place:
228
229         out = ma.masked_array(img, mask=new_mask)
230
231     else:
232
233         # will this work?
234         img[new_mask] = ma.masked
235
236         out = img
237
238     # now return succeed if asked to
239     if return_success:
240         return out, True
241
242     else:
243         return out
244
245
246 if __name__ == "__main__":
247
248     # DEMO FOR SHOWING OFF DILATE_BOUNDARY EFFECT
249
250     from placenta import get_named_placenta
251     from frangi import frangi_from_image

```

```

252 import matplotlib.pyplot as plt
253
254 import os.path
255
256 dest_dir = 'demo_output'
257 img = get_named_placenta('T-BN0164923.png')
258
259 sigma = 3
260 radius = 25
261
262 inset = np.s_[800:1000, 500:890]
263
264 D = dilate_boundary(img, radius=radius)
265
266 Fimg = frangi_from_image(img, sigma, dark_bg=False, dilation_radius=None)
267 FD = frangi_from_image(D, sigma, dark_bg=False)
268 FDinv = frangi_from_image(D, sigma, dark_bg=True)
269 Finv = frangi_from_image(img, sigma, dark_bg=True, dilation_radius=None)
270
271 fig, axes = plt.subplots(ncols=2, nrows=3)
272
273 axes[0,0].imshow(img[inset].filled(0), cmap=plt.cm.gray)
274 axes[0,1].imshow(D[inset].filled(0), cmap=plt.cm.gray)
275 axes[1,0].imshow(Fimg[inset].filled(0), cmap=plt.cm.nipy_spectral)
276 axes[1,1].imshow(FD[inset].filled(0), cmap=plt.cm.nipy_spectral)
277 axes[2,0].imshow(Finv[inset].filled(0), cmap=plt.cm.nipy_spectral)
278 axes[2,1].imshow(FDinv[inset].filled(0), cmap=plt.cm.nipy_spectral)
279
280 for a in axes.ravel():
281     # get rid of all the labels
282     plt.setp(a.get_xticklabels(), visible=False)
283     plt.setp(a.get_yticklabels(), visible=False)
284
285 # lol matlab
286 for i in range(5):
287     fig.tight_layout()
288
289 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
7 from skimage.filters import sobel
8 from skimage.morphology import (remove_small_objects, thin, disk,
9                                 binary_dilation)
10
11 from frangi import frangi_from_image
12 from merging import apply_threshold, nz_percentile
13 from plate_morphology import dilate_boundary
14 from skimage.segmentation import random_walker
15 import numpy as np
16
17 from skimage.filters.rank import enhance_contrast_percentile as ecp
18 from scipy.ndimage import distance_transform_edt as edt
19
20
21 def random_walk_fill(img, Fmax, high_thresh, low_thresh, dark_bg):

```

```

22 """
23 # this is deprecated, it's trash
24 """
25
26 s = sobel(img)
27 s = dilate_boundary(s, mask=img.mask, radius=20)
28
29 finv = frangi_from_image(img, sigma=0.8, beta=0.5, dark_bg=(not dark_bg),
30                           dilation_radius=20)
31
32 finv_thresh = (finv > nz_percentile(finv, 50)).filled(0)
33 margins = remove_small_objects(finv_thresh, min_size=32)
34
35 markers = np.zeros(img.shape, dtype=np.int32)
36 markers[Fmax < low_thresh] = 1
37
38 margins_added = (margins | (Fmax > high_thresh))
39 #margins_added = remove_small_holes(margins_added, area_threshold=50)
40
41 markers[Fmax < low_thresh] = 1
42
43 markers[margins_added] = 2
44
45 rw = random_walker(1-Fmax, markers, beta=1000)
46
47 approx_rw = (rw == 2)
48
49 return approx_rw, markers, margins_added
50
51
52 def random_walk_scalewise(F, high_thresh=0.4, rw_beta=130,
53                           return_labels=False):
54     """Random walker on each a multiscale Frangi result"""
55     print('doing scalewise random walk', end=' ')
56     V = np.transpose(F, axes=(2, 0, 1))
57     W = np.zeros(V.shape, np.bool)
58     for n, v in enumerate(V):
59         print('σ', end=' ', flush=True)
60         markers = np.zeros(v.shape, np.int32)
61         markers[v == 0] = 1
62         # this could be a vector too
63         markers[v > high_thresh] = 2
64         # or 1-v
65         W[n] = (random_walker(v, markers, rw_beta) == 2)
66     print()
67     if not return_labels:
68         return W.any(axis=0)
69     else:
70         # argmax grabs the first scale where it was satisfied
71         # so this will grab the lowest scale that matches
72         return W.any(axis=0), W.argmax(axis=0)
73
74
75 def dilate_to_rim(spine, rim, thin_spine=False, max_radius=15,
76                   return_radii=False):
77     """Grow spine region by dilating towards closest rim point of the trough.
78
79     Parameters:
80     -----
81     spine, rim: 2D ndarray type bool
82         spine are points, dilate to rim
83     thin_spine: bool, optional
84         Thin the spine before performing dilation. This makes it much faster
85         at the risk of giving an incomplete fill if the spine isn't in the

```

```

86     middle of the margins, also makes returning the labels more meaningful
87
88     Returns
89     -----
90     ...
91
92     """
93
94     D = np.ones(spine.shape, np.bool)
95     D[rim] = 0
96
97     spine_dists = edt(D)
98
99     # either thin first or don't, i dunno
100    if thin_spine:
101        spine_dists[~thin(spine)] = 0
102    else:
103        spine_dists[~spine] = 0
104
105    spine_radii = np.round(spine_dists).astype('int')
106
107    dilation_stack = np.stack([binary_dilation(spine_radii==r, selem=disk(r))
108                               for r in range(1,max_radius+1)])
109
110    approx = dilation_stack.any(axis=0)
111
112    if not return_radii:
113        return approx
114    else:
115        return approx, dilation_stack.argmax(axis=0)

```

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                                 convex_hull_object)
13 from skimage.restoration import inpaint_biharmonic
14 import numpy as np
15 import numpy.ma as ma
16 from scipy.ndimage import label
17 from skimage.util import img_as_float
18 from skimage.segmentation import find_boundaries
19 from plate_morphology import dilate_boundary
20
21 import scipy.ndimage as ndi
22 import matplotlib.pyplot as plt
23
24 def inpaint_glare(img, threshold=175, window_size=15, mask=None):
25     """
26     img is a masked array type uint [0,255]
27     """
28
29     # bool array, true where glare

```

```

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

```

```

94     relevant = A[A > 0]
95
96     return np.median(relevant)
97
98
99 def inpaint_hybrid(img, threshold=175, min_size=64, boundary_radius=10):
100    """
101        use biharmonic inpainting in larger, inner areas (important stuff)
102        and median inpainting in smaller areas and along boundary
103    """
104
105    glare = mask_glare(img, threshold=threshold, mask_only=True)
106
107    glare_inside = dilate_boundary(glare, mask=img.mask,
108                                    radius=boundary_radius).filled(0)
109
110    large_glare = remove_small_objects(glare_inside, min_size=min_size,
111                                       connectivity=2)
112    small_glare = np.logical_and(glare, np.invert(large_glare))
113
114    # inpaint smaller and less important values with less expensive method
115    inpainted = inpaint_with_boundary_median(img, mask=small_glare)
116    hybrid = img_as_float(inpainted) # scale 0 to 1
117
118    # inpaint larger regions with biharmonic inpainting
119    large_inpainted = inpaint_biharmonic(img.filled(0), mask=large_glare)
120
121    # now overwrite with these values
122    hybrid[large_glare] = large_inpainted[large_glare]
123
124    # put on old image mask
125    return ma.masked_array(hybrid, mask=img.mask)
126
127 def inpaint_with_biharmonic(img, threshold=175):
128    """
129        use biharmonic inpainting *all* glare
130    """
131    glare = mask_glare(img, threshold=threshold, mask_only=True)
132    inpainted = inpaint_biharmonic(img_as_float(img.filled(0)), mask=glare)
133
134    if ma.is_masked(img):
135        return ma.masked_array(inpainted, mask=img.mask)
136    else:
137        return inpainted
138
139 def mask_glare(img, threshold=175, mask_only=False):
140    """
141        for demoing purposes, with placenta.show_mask
142
143        if mask_only, just return the mask. Otherwise return a copy of img with
144        that added to the mask. If you want the original mask to be ignored,
145        just pass img.filled(0) ya doofus
146
147        threshold is expected to be of the same dtype as img *unless# it assumes
148        its default value, in which case the threshold will be converted to a float
149    """
150
151    # if img.dtype is floating but threshold value is still the default
152    # this could be generalized
153    if np.issubdtype(img.dtype, np.floating) and (threshold == 175):
154        threshold = 175 / 255
155    # region to inpaint
156    inp = (img > threshold)
157

```

```

158     # get a larger area around the specks
159     inp = binary_dilation(inp, selem=disk(2))
160
161     # remove anything large
162     #inp = white_tophat(inp, selem=disk(3))
163
164     if mask_only:
165         return inp
166     else:
167         # both the original background *and* these new glared regions
168         # are masked
169         return ma.masked_array(img, mask=inp)
170
171
172 def mask_stump(img, mask=None, mask_only=True):
173
174     if img.ndim < 3:
175         print('better to pass the raw image')
176         channel = img
177     else:
178         channel = img[...,0]
179
180     C = channel.copy()
181     if mask is not None:
182         C[mask] = 0
183     elif ma.is_masked(img):
184         C[img.mask] = 0
185         mask = img.mask
186     else:
187         mask = np.zeros(img.shape, np.bool)
188
189     thresh = (170/255)*C.max()
190     b = ndi.white_tophat(C > thresh, 90)
191     b = remove_small_objects(b, 1000)
192     #b = convex_hull_object(b)
193     b[mask] = 0
194
195     # sort by size of object (largest first)
196
197     #incl_count = 0 #objects used
198     #mags = sorted(list(range(1,n_labs+1)), key=lambda lb: np.sum(labs==lb),
199     #               reverse=True)
200     #print(mags)
201     #for l in mags:
202     #    # big things get weird
203     #    if np.sum(b==l) > 5000:
204     #        print('skipping very large object')
205     #        # get rid of it, whatever
206     #        plt.imshow(b==l)
207     #        b[b==l] = 0
208     #    else:
209     #        incl_count += 1
210     #
211     #    if incl_count > 4:
212     #        print('only removing a few things here')
213     #        b[b==l] = 0
214     #
215     #print('b after')
216     #plt.imshow(b)
217     #b = ndi.binary_dilation(b, disk(15))
218     #
219     if mask_only:
220         return b
221

```

```

222     else:
223         # will add to existing mask
224         return ma.masked_array(img, mask=b)
225
226
227
228 DARK_RED = np.array([103, 15, 23]) / 255.
229
230 # test it on a particularly bad sample
231 if __name__ == "__main__":
232
233     from placenta import get_named_placenta, show_mask
234     import matplotlib.pyplot as plt
235
236     filename = 'T-BN0204423.png' # a particularly glary sample
237     img = get_named_placenta(filename)
238
239     img = ma.masked_array(img_as_float(img), mask=img.mask)
240     crop = np.s_[150:500, 150:800] # indices to zoom in on the region
241     zoom = np.s_[300:380, 300:380] # even smaller region
242
243     inset = zoom # which view to use
244
245     masked = mask_glare(img) # for viewing
246     inpainted = inpaint_glare(img)
247     minpainted = inpaint_with_boundary_median(img)
248     hinpainted = inpaint_hybrid(img)
249     binpainted = inpaint_with_biharmonic(img)
250
251     # view the closeup like this
252     minpainted_view = show_mask(minpainted, interactive=False,
253                                  mask_color=DARK_RED)
254     inpainted_view = show_mask(inpainted, interactive=False,
255                                mask_color=DARK_RED)
256     masked_view = show_mask(masked, interactive=False,
257                             mask_color=DARK_RED)
258     img_view = show_mask(img, interactive=False,
259                           mask_color=DARK_RED)
260     hinpainted_view = show_mask(hinpainted, interactive=False,
261                                mask_color=DARK_RED)
262     binpainted_view = show_mask(binpainted, interactive=False,
263                                mask_color=DARK_RED)
264
265     # view them all next to each other
266
267     fig, axes = plt.subplots(ncols=3, nrows=2)
268
269     axes[0,0].imshow(img_view[inset])
270     axes[0,1].imshow(masked_view[inset])
271     axes[0,2].imshow(inpainted_view[inset])
272     axes[1,0].imshow(minpainted_view[inset])
273     axes[1,1].imshow(binpainted_view[inset])
274     axes[1,2].imshow(hinpainted_view[inset])
275
276     for a in axes.ravel():
277         # get rid of all the labels
278         plt.setp(a.get_xticklabels(), visible=False)
279         plt.setp(a.get_yticklabels(), visible=False)
280
281     # lol matlab
282     for i in range(5):
283         fig.tight_layout()
284
285     IMGS = np.vstack((

```

```

286     np.hstack((img_view, masked_view, inpainted_view)),
287     np.hstack((minpainted_view, binpainted_view, hinpainted_view)))
288
289 # THEN IMSAVE
290
291 # plt.imsave('preprocessing_comparison_cropped.png', IMGS)
292 # plt.imsave('preprocessing_comparison_zoomed.png', IMGS)
293
294 # 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 # now make the other one visible (this is dumb)
113 for layer in img.layers:
114     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     print "Saved."
147
148
149 register(
150     "process_NCS_xcf",
151     "Create base image + trace + mask from an NCS xcf file",
152     "Create base image + trace + mask from an NCS xcf file",
153     "Luke Wukmer",
154     "Luke Wukmer",
155     "2018",
156     "<Image>/Image/Process_NCS_xcf...",
157     "RGB*", "GRAY*",
158     [],
159     [],
160     process_NCS_xcf)
161
162 main()

```

listings/rw_demo.py

```

1 #!/usr/bin/env python3
2
3 import numpy as np
4 import numpy.ma as ma
5
6 import matplotlib.pyplot as plt
7 from skimage.util import img_as_float
8 from skimage.io import imread
9 from placenta import (get_named_placenta, list_by_quality, cropped_args,
10                         img_as_float, open_typefile, open_tracefile,
11                         measure_ncs_markings, add_ucip_to_mask)
12

```

```

13 from frangi import frangi_from_image
14 from hfft import fft_gradient, fft_hessian, fft_gaussian
15 from merging import nz_percentile
16 from plate_morphology import dilate_boundary
17 import os.path, os
18 from scoring import confusion, mcc
19 from preprocessing import inpaint_hybrid
20
21 import matplotlib as mpl
22
23 from skimage.segmentation import random_walker
24
25 import json
26
27
28 def rw_demo(filename, rw_beta, threshold, output_dir=None):
29
30     # ideally this would be a class with all of these
31     cimg = open_typefile(filename, 'raw')
32    ctrace = open_typefile(filename, 'ctrace')
33     trace = open_tracefile(filename)
34     img = get_named_placenta(filename)
35     crop = cropped_args(img)
36     ucip = open_typefile(filename, 'ucip')
37     img = inpaint_hybrid(img)
38
39     # make the size of figures more consistent
40     if img[crop].shape[0] > img[crop].shape[1]:
41         # and rotating it would be fix all this automatically
42         cimg = np.rot90(cimg)
43         ctrace = np.rot90(ctrace)
44         trace = np.rot90(trace)
45         img = np.rot90(img)
46         ucip = np.rot90(ucip)
47         crop = cropped_args(img)
48
49     ucip_midpoint, _ = measure_ncs_markings(ucip)
50     ucip_mask = add_ucip_to_mask(ucip_midpoint, radius=60, mask=img.mask)
51
52     plt.close('all')
53
54     cm = mpl.cm.plasma
55     cmscales = mpl.cm.magma
56     cm.set_bad('k', 1) # masked areas are black, not white
57     cmscales.set_bad('w', 1)
58
59     scales = np.logspace(-1.5, 3.5, num=12, base=2)
60
61     W = np.zeros((len(scales), *img.shape), dtype=np.bool)
62     WL = np.zeros((len(scales), *img.shape), dtype=np.bool)
63     V = np.zeros((len(scales), *img.shape))
64
65     #fig, ax = plt.subplots(ncols=3, nrows=len(scales), figsize=(10,10))
66
67     for n, sigma in enumerate(scales):
68
69         f = frangi_from_image(img, sigma, dark_bg=False, dilation_radius=20,
70                               beta=0.35)
71         V[n] = f
72
73         f[f == 0] = ma.masked
74
75         fig, ax = plt.subplots(ncols=4, nrows=1, figsize=(20,6))

```

```

77
78     ax[0].imshow(f[crop], cmap=cm)
79     ax[0].axis('off')
80     ax[0].set_title(rf'Frangi score, ( $\sigma_n = \text{sigma} : .3f$ )')
81
82     markers = np.zeros(img.shape, np.int32)
83     markers[f.mask] = 1
84     markers[f > THRESHOLD] = 2
85
86     ax[1].imshow(markers[crop], cmap=plt.cm.viridis, vmin=0, vmax=3)
87     ax[1].axis('off')
88     ax[1].set_title('markers')
89
90     rw = random_walker(1-f.filled(0), markers, beta=RW_BETA)
91     rw_L = random_walker(f.filled(0) > 0, markers, beta=RW_BETA)
92     W[n] = (rw == 2)
93     WL[n] = (rw_L == 2)
94
95     # set the new stuff to a higher number so you can see what was added
96     show_added = rw.copy()
97     show_added_L = rw_L.copy()
98     show_added[~(markers == 2) & (rw==2)] = 3
99     show_added_L[~(markers == 2) & (rw_L==2)] = 3
100    # set the zero stuff back to 0 so you can tell what wasn't filled
101    show_added[(rw == 1) & (markers == 0)] = 0
102    show_added_L[(rw_L == 1) & (markers == 0)] = 0
103
104    ax[2].imshow(show_added[crop], vmin=0, vmax=3)
105    ax[2].axis('off')
106    ax[2].set_title('rw')
107    ax[3].imshow(show_added_L[crop], vmin=0, vmax=3)
108    ax[3].axis('off')
109    ax[3].set_title('rw-loose')
110    fig.tight_layout()
111    fig.subplots_adjust(hspace=0.00, wspace=0.01)
112
113    if output_dir is not None:
114        fig.savefig(f'./output/{output_dir}/{basename}_{n:02}.png')
115    if INTERACTIVE:
116        plt.show()
117
118    Vmax, Vargmax = V.max(axis=0), V.argmax(axis=0)
119    Vmax = ma.masked_where(Vmax==0, Vmax)
120    #Vargmax = ma.masked_where(~trace, Vargmax)
121
122    labs_FA = Vargmax*(Vmax > THRESHOLD).filled(0)
123    approx_FA = labs_FA!=0
124    confuse_FA = confusion(approx_FA, trace, bg_mask=ucip_mask)
125    # get the smallest label that matched
126
127    labs = np.argmax(W, axis=0) # returns the first index of boolean
128    labs =ma.masked_where(labs==0, labs)
129    approx = labs.filled(0)!=0
130    confuse = confusion(approx, trace, bg_mask=ucip_mask)
131
132
133    labs_L = np.argmax(WL, axis=0) # returns the first index of boolean
134    labs_L =ma.masked_where(labs_L==0, labs_L)
135    approx_L = labs_L.filled(0)!=0
136    confuse_L = confusion(approx_L, trace, bg_mask=ucip_mask)
137
138    fig, ax = plt.subplots(ncols=4, nrows=2, figsize=(20,12))
139
140    ax[0,0].imshow(cimg[crop])

```

```

141 ax[0,0].axis('off')
142 ax[0,0].set_title(basename)
143
144 ax[1,0].imshow(ctrace[crop])
145 ax[1,0].axis('off')
146 ax[1,0].set_title('ground truth')
147
148 ax[0,1].imshow(Vmax[crop], cmap=cm)
149 ax[0,1].axis('off')
150 ax[0,1].set_title('max( $V_\sigma$ )')
151
152
153 ax[0,2].imshow(labs[crop], cmap='magma')
154 ax[0,2].axis('off')
155 ax[0,2].set_title('segmentation (rw)')
156
157 ax[0,3].imshow(labs_L[crop], cmap='magma')
158 ax[0,3].axis('off')
159 ax[0,3].set_title('segmentation (rw-loose)')
160
161 precision_score = lambda t: int(t[0]) / int(t[0] + t[2])
162
163 m, counts = mcc(approx, trace, bg_mask=ucip_mask, return_counts=True)
164 m_L, counts_L = mcc(approx_L, trace, bg_mask=ucip_mask, return_counts=True)
165 m_FA, counts_FA = mcc(approx_FA, trace, bg_mask=ucip_mask,
166                         return_counts=True)
167
168 p = precision_score(counts)
169 p_L = precision_score(counts_L)
170 p_FA = precision_score(counts_FA)
171
172 ax[1,1].imshow(confuse_FA[crop])
173 ax[1,1].axis('off')
174 ax[1,1].set_title(f'fixed  $\alpha=THRESHOLD$ ', loc='left')
175 ax[1,1].set_title(f'MCC: {m_FA:.2f}\n' +
176                   f'precision: {p_FA:.2%}', loc='right')
177 ax[1,2].imshow(confuse[crop])
178 ax[1,2].axis('off')
179 ax[1,2].set_title('scalewise-RW', loc='left')
180 ax[1,2].set_title(f'MCC: {m:.2f}\n' +
181                   f'precision: {p:.2%}', loc='right')
182
183 ax[1,3].imshow(confuse_L[crop])
184 ax[1,3].axis('off')
185 ax[1,3].set_title('scalewise-RW (loose)', loc='left')
186 ax[1,3].set_title(f'MCC: {m_L:.2f}\n' +
187                   f'precision: {p_L:.2%}', loc='right')
188
189 fig.tight_layout()
190 fig.subplots_adjust(hspace=0.05, wspace=0.01)
191
192 if output_dir is not None:
193     fig.savefig(f'./output/{output_dir}/{basename}_m.png')
194
195 return (filename, m_FA, p_FA, m, p, m_L, p_L)
196
197
198 INTERACTIVE = True
199 filenames = list_by_quality(1)
200 filenames.extend(list_by_quality(2))
201 filenames.extend(list_by_quality(3))
202 filenames.extend(list_by_quality(0))
203 RW_BETA = 10

```

```

205 THRESHOLD = .4
206
207 run_data = list()
208
209 for N, filename in enumerate(filenames):
210
211     basename = filename.strip('T-').rstrip('.png')
212     print('running rw_demo on', basename, f'{N+1} of {len(filenames)}')
213     row = rw_demo(filename, RW_BETA, THRESHOLD, None)
214     run_data.append(row)
215
216 if INTERACTIVE:
217
218     plt.show()
219
220 #print(row)
221
222 # do this incrementally; i'm afraid
223 if (N % 25 == 0) and (N > 0):
224     print('backing up data!')
225     with open(f'rw_demo_scores_1206{N//25}.json', 'w') as f:
226         json.dump(run_data, f, indent=True)
227
228 with open(f'rw_demo_scores_all_1206.json', 'w') as f:
229     json.dump(run_data, f, indent=True)

```

listings/rwpf_demo.py

```

1 #!/usr/bin/env python3
2
3 import numpy as np
4 import numpy.ma as ma
5
6 import matplotlib.pyplot as plt
7 from skimage.util import img_as_float
8 from skimage.io import imread
9 from placenta import (get_named_placenta, list_by_quality, cropped_args,
10                      img_as_float, open_typefile, open_tracefile,
11                      measure_ncs_markings, add_ucip_to_mask)
12
13 from frangi import frangi_from_image
14 from hfft import fft_gradient, fft_hessian, fft_gaussian
15 from merging import nz_percentile, apply_threshold
16 from plate_morphology import dilate_boundary
17 import os.path, os
18 from scoring import confusion, mcc
19
20 import matplotlib as mpl
21
22 from skimage.segmentation import random_walker
23
24 import json
25
26
27
28 INTERACTIVE = True
29 filenames = list_by_quality(1)
30 filenames.extend(list_by_quality(2))
31 filenames.extend(list_by_quality(3))
32 filenames.extend(list_by_quality(0))
33 RW_BETA = 10

```

```

34 P_THRESHOLD = 95
35
36 run_data = list()
37
38 for N, filename in enumerate(filenames):
39     basename = filename.strip('T-').rstrip('.png')
40     print('running rwpf_demo on', basename, f'({N+1} of {len(filenames)})')
41
42 # ideally this would be a class with all of these
43 cimg = open_typefile(filename, 'raw')
44 ctrace = open_typefile(filename, 'ctrace')
45 trace = open_tracefile(filename)
46 img = get_named_placenta(filename)
47 crop = cropped_args(img)
48 ucip = open_typefile(filename, 'ucip')
49
50 # make the size of figures more consistent
51 if img[crop].shape[0] > img[crop].shape[1]:
52     # and rotating it would be fix all this automatically
53     cimg = np.rot90(cimg)
54     ctrace = np.rot90(ctrace)
55     trace = np.rot90(trace)
56     img = np.rot90(img)
57     ucip = np.rot90(ucip)
58     crop = cropped_args(img)
59
60 ucip_midpoint, _ = measure_ncs_markings(ucip)
61 ucip_mask = add_ucip_to_mask(ucip_midpoint, radius=60, mask=img.mask)
62
63 plt.close('all')
64
65 cm = mpl.cm.plasma
66 cmscales = mpl.cm.magma
67 cm.set_bad('k', 1) # masked areas are black, not white
68 cmscales.set_bad('w', 1)
69
70 scales = np.logspace(-1.5, 3.5, num=12, base=2)
71
72 W = np.zeros((len(scales), *img.shape), dtype=np.bool)
73 WL = np.zeros((len(scales), *img.shape), dtype=np.bool)
74 V = np.zeros((len(scales), *img.shape))
75 p_alphas = np.zeros_like(scales)
76
77 #fig, ax = plt.subplots(ncols=3, nrows=len(scales), figsize=(10,10))
78
79 for n, sigma in enumerate(scales):
80
81     f = frangi_from_image(img, sigma, dark_bg=False, dilation_radius=20,
82                           beta=0.35)
83     V[n] = f
84
85     f[f == 0] = ma.masked
86
87     fig, ax = plt.subplots(ncols=4, nrows=1, figsize=(20,6))
88
89     ax[0].imshow(f[crop], cmap=cm)
90     ax[0].axis('off')
91     ax[0].set_title(rf'Frangi score, ( $\sigma_n = \text{sigma} : 3f$ )')
92     p = nz_percentile(f, P_THRESHOLD)
93     p_alphas[n] = p
94     hi_marks = (f > p).filled(0)
95     markers = np.zeros(img.shape, np.int32)
96     markers[f.mask] = 1
97     markers[hi_marks] = 2

```

```

98
99     ax[1].imshow(markers[crop], cmap=plt.cm.viridis, vmin=0, vmax=3)
100    ax[1].axis('off')
101    ax[1].set_title('markers')
102
103    rw = random_walker(1-f.filled(0), markers, beta=RW_BETA)
104    rw_L = random_walker(f.filled(0) > 0, markers, beta=RW_BETA)
105    W[n] = (rw == 2)
106    WL[n] = (rw_L == 2)
107
108    # set the new stuff to a higher number so you can see what was added
109    show_added = rw.copy()
110    show_added_L = rw_L.copy()
111    show_added[~(markers == 2) & (rw==2)] = 3
112    show_added_L[~(markers == 2) & (rw_L==2)] = 3
113    # set the zero stuff back to 0 so you can tell what wasn't filled
114    show_added[(rw == 1) & (markers == 0)] = 0
115    show_added_L[(rw_L == 1) & (markers == 0)] = 0
116
117    ax[2].imshow(show_added[crop], vmin=0, vmax=3)
118    ax[2].axis('off')
119    ax[2].set_title('rw')
120    ax[3].imshow(show_added_L[crop], vmin=0, vmax=3)
121    ax[3].axis('off')
122    ax[3].set_title('rw-loose')
123    fig.tight_layout()
124    fig.subplots_adjust(hspace=0.00, wspace=0.01)
125
126    fig.savefig(f'./output/RWPFDemo/{basename}_{n:02}.png')
127    if INTERACTIVE:
128        plt.show()
129
130    Vmax, Vargmax = V.max(axis=0), V.argmax(axis=0)
131    Vmax = ma.masked_where(Vmax==0, Vmax)
132    #Vargmax = ma.masked_where(~trace, Vargmax)
133
134    approx_FA, labs_FA = apply_threshold(np.transpose(V, axes=(1,2,0)), p_alphas)
135    #labs_FA = Vargmax*(hi_marks)
136    #approx_FA = labs_FA!=0
137    confuse_FA = confusion(approx_FA, trace, bg_mask=ucip_mask)
138    # get the smallest label that matched
139
140    labs = np.argmax(W, axis=0) # returns the first index of boolean
141    labs = ma.masked_where(labs==0, labs)
142    approx = labs.filled(0)!=0
143    confuse = confusion(approx, trace, bg_mask=ucip_mask)
144
145
146    labs_L = np.argmax(WL, axis=0) # returns the first index of boolean
147    labs_L = ma.masked_where(labs_L==0, labs_L)
148    approx_L = labs_L.filled(0)!=0
149    confuse_L = confusion(approx_L, trace, bg_mask=ucip_mask)
150
151    fig, ax = plt.subplots(ncols=4, nrows=2, figsize=(20,12))
152
153    ax[0,0].imshow(cimg[crop])
154    ax[0,0].axis('off')
155    ax[0,0].set_title(basename)
156
157    ax[1,0].imshow(ctrace[crop])
158    ax[1,0].axis('off')
159    ax[1,0].set_title('ground truth')
160
161    ax[0,1].imshow(Vmax[crop], cmap=cm)

```

```

162 ax[0,1].axis('off')
163 ax[0,1].set_title('max( $V_\sigma$ )')
164
165
166 ax[0,2].imshow(labs[crop], cmap='magma')
167 ax[0,2].axis('off')
168 ax[0,2].set_title('segmentation (rw)')
169
170 ax[0,3].imshow(labs_L[crop], cmap='magma')
171 ax[0,3].axis('off')
172 ax[0,3].set_title('segmentation (rw-loose)')
173
174 precision_score = lambda t: int(t[0]) / int(t[0] + t[2])
175
176 m, counts = mcc(approx, trace, bg_mask=ucip_mask, return_counts=True)
177 m_L, counts_L = mcc(approx_L, trace, bg_mask=ucip_mask, return_counts=True)
178 m_FA, counts_FA = mcc(approx_FA, trace, bg_mask=ucip_mask,
179                         return_counts=True)
180
181 p = precision_score(counts)
182 p_L = precision_score(counts_L)
183 p_FA = precision_score(counts_FA)
184
185 ax[1,1].imshow(confuse_FA[crop])
186 ax[1,1].axis('off')
187 ax[1,1].set_title(rf' fixed  $p=P_T\text{THRESHOLD}$ ', loc='left')
188 ax[1,1].set_title(f'MCC: {m_FA:.2f}\n' +
189                   f'precision: {p_FA:.2%}', loc='right')
190 ax[1,2].imshow(confuse[crop])
191 ax[1,2].axis('off')
192 ax[1,2].set_title(' scalewise-RW', loc='left')
193 ax[1,2].set_title(f'MCC: {m:.2f}\n' +
194                   f'precision: {p:.2%}', loc='right')
195
196 ax[1,3].imshow(confuse_L[crop])
197 ax[1,3].axis('off')
198 ax[1,3].set_title(' scalewise-RW (loose)', loc='left')
199 ax[1,3].set_title(f'MCC: {m_L:.2f}\n' +
200                   f'precision: {p_L:.2%}', loc='right')
201
202 fig.tight_layout()
203 fig.subplots_adjust(hspace=0.05, wspace=0.01)
204
205 fig.savefig(f'./output/RWPFDemo/{basename}_m.png')
206
207 row = (basename, m_FA, p_FA, m, p, m_L, p_L)
208
209 run_data.append(row)
210 if INTERACTIVE:
211     plt.show()
212
213 print(row)
214
215 # do this incrementally; i'm afraid
216 if (N % 25 == 0) and (N > 0):
217     print('backing up data!')
218     with open(f'rwpf_demo_scores_{N//25}.json', 'w') as f:
219         json.dump(run_data, f)
220
221 with open(f'rwpf_demo_scores_all.json', 'w') as f:
222     json.dump(run_data, f)

```

listings/sampled_gauss_problem.py

```
1 # coding: utf-8
2 from hfft import discrete_gaussian_kernel
3 help(discrete_gaussian_kernel)
4 discrete_gaussian_kernel(100,1)
5 import matplotlib.pyplot as plt
6 np.arange(-25,25)
7 import numpy as np
8 np.linspace(-1,1,n=50)
9 np.linspace(-1,1,num=50)
10 dom =
11 plt.plot(dom, discrete_gaussian_kernel(len(dom), 1))
12 plt.plot(dom, discrete_gaussian_kernel(len(dom), 1))
13 len(dom)
14 dgk = discrete_gaussian_kernel
15 dgk(50)
16 dgk(50,1)
17 _.shape
18 dgk(49,1)
19 _.shape
20 dgk(25,1)
21 dgk(25,1)
22 dom = np.linspace(-1,1,num=25)
23 plt.plot(dom,dgk(25,1))
24 plt.show()
25 plt.plot(dom,dgk(25,.5))
26 plt.show()
27 plt.plot(dom,dgk(25,.1))
28 plt.show()
29 plt.plot(dom,dgk(25,.05))
30 plt.show()
31 plt.plot(dom,dgk(25,.01))
32 plt.show()
33 plt.plot(dom,dgk(25,.001))
34 plt.show()
35 plt.plot(dom,dgk(25,.0001))
36 plt.show()
37 plt.plot(dom,dgk(25,0))
38 plt.show()
39 from scipy.signal import gaussian
40 get_ipython().run_line_magic('pinfo', 'gaussian')
41 gaussian(25,.1)
42 gaussian(25,.1)
43 plt.plot(dom, gaussian(25,.1))
44 plt.show()
45 plt.plot(dom, gaussian(25,.1), dom, dgk(25,.1))
46 plt.show()
47 plt.plot(dom, gaussian(25,10), dom, dgk(25,10))
48 plt.show()
49 plt.plot(dom, gaussian(25,10), dom, dgk(25,10))
50 plt.plot(dom, (1/np.sqrt(2*np.pi*10**2))*gaussian(25,10), dom, dgk(25,10))
51 plt.show()
52 plt.plot(dom, (1/np.sqrt(2*np.pi*10**2))*gaussian(25,10), dom, dgk(25,10**2))
53 plt.show()
54 plt.plot(dom, np.sqrt((1/np.sqrt(2*np.pi*10**2)))*gaussian(25,10), dom, dgk(25,10**2))
55 plt.show()
56 plt.plot(dom, np.sqrt((1/np.sqrt(2*np.pi*10**2)))*gaussian(100,10), dom, dgk(100,10**2))
57 dom = np.linspace(-1,1,num=100)
58 plt.plot(dom, np.sqrt((1/np.sqrt(2*np.pi*100**2)))*gaussian(100,10), dom, dgk(100,10**2))
59 plt.plot(dom, np.sqrt((1/np.sqrt(2*np.pi*10**2)))*gaussian(101,10), dom, dgk(101,10**2))
60 dom = np.linspace(-1,1,num=101)
61 plt.plot(dom, np.sqrt((1/np.sqrt(2*np.pi*10**2)))*gaussian(101,10), dom, dgk(101,10**2))
62 plt.show()
```

```

63 plt.plot(dom, np.sqrt((1/np.sqrt(2*np.pi*10**2)))*gaussian(101,10), dom, dgk(101,10**2))
64 plt.show()
65 plt.plot(dom, np.sqrt((1/np.sqrt(2*np.pi*10**2)))*gaussian(101,10), dom, dgk(101,10))
66 plt.show()
67 plt.plot(dom, np.sqrt((1/np.sqrt(2*np.pi*10**2)))*gaussian(101,10), dom, dgk(101,10**2))
68 plt.show()
69 plt.show()
70 plt.plot(dom, np.sqrt((1/np.sqrt(2*np.pi*10**2)))*gaussian(101,10), dom, dgk(101,10))
71 plt.show()
72 dgk(5,1)
73 dgk(15,1)
74 dom = np.arange(15)
75 plt.imshow(dom, dgk(len(dom),1))
76 plt.imshow(dom, dgk(dom.size,1))
77 plt.imshow(dom, dgk(dom.size,1))
78 plt.imshow(dom, dgk(dom.size,1))
79 plt.imshow(dom, dgk(dom.size,1))
80 plt.imshow(dom, dgk(dom.size,1.))
81 dom = np.linspace(15)
82 dom = np.linspace(0,1,15)
83 dom
84 plt.imshow(dom, dgk(dom.size,1.))
85 plt.imshow(dom, dgk(15,1.))
86 get_ipython().run_line_magic('clear', '')
87 plt.imshow(dom, dgk(15,1.))
88 plt.plot(dom, dgk(15,1.))
89 plt.show()
90 plt.plot(dom, dgk(15,1.))
91 plt.show()
92 plt.plot(dom, dgk(15,.05))
93 plt.show()
94 plt.plot(dom, dgk(15,0))
95 plt.show()
96 plt.plot(dgk(100,0))
97 plt.show()
98 plt.scatter(dgk(100,0))
99 get_ipython().run_line_magic('pinfo', 'plt.scatter')
100 plt.plot(dgk(100,0), 'o')
101 plt.show()
102 plt.plot(dgk(100,0), 'o', markersize=1)
103 plt.show()
104 plt.show()
105 plt.plot(dgk(100,1), 'o', markersize=1)
106 plt.show()
107 plt.plot(dgk(100,15), 'o', markersize=1)
108 plt.show()
109 gaussian(100,15)
110 gaussian(100,15) / (15*np.sqrt(2*np.pi))
111 plot(gaussian(100,15) / (15*np.sqrt(2*np.pi)), 'or', markersize=1)
112 plt.plot(gaussian(100,15) / (15*np.sqrt(2*np.pi)), 'or', markersize=1)
113 plt.plot(dgk(100,15), 'o', markersize=1)
114 plt.show()
115 plt.plot(gaussian(105,15) / (15*np.sqrt(2*np.pi)), 'or', markersize=1)
116 plt.plot(dgk(105,15), 'o', markersize=1)
117 plt.show()
118 plt.plot(dgk(105,5), 'o-', markersize=1)
119 plt.plot(gaussian(105,5) / (5*np.sqrt(2*np.pi)), 'or', markersize=1)
120 plt.show()
121 plt.plot(gaussian(105,2) / (2*np.sqrt(2*np.pi)), 'or', markersize=1)
122 plt.plot(dgk(105,2), 'o-', markersize=1)
123 plt.show()
124 plt.plot(dgk(105,1.2), 'o-', markersize=1)
125 plt.show()
126 sigma = 2; plt.plot(gaussian(105,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=1); p

```

```

127 plt.show()
128 sigma = 2; plt.plot(gaussian(105,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2); p
129 plt.show()
130 sigma = 1.2; plt.plot(gaussian(105,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);
131 plt.show()
132 sigma = 1.2; plt.plot(gaussian(105,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);
133 plt.show()
134 sigma = .8; plt.plot(gaussian(105,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);
135 plt.show()
136 sigma = .3; plt.plot(gaussian(105,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);
137 plt.show()
138 plt.show()
139 sigma = .01; plt.plot(gaussian(105,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);
140 plt.show()
141 sigma = .01; plt.plot(gaussian(105,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);
142 plt.show()
143 sigma = .10; plt.plot(gaussian(105,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);
144 plt.show()
145 sigma = .10; plt.plot(gaussian(100,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);
146 plt.show()
147 sigma = .20; plt.plot(gaussian(100,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);
148 plt.show()
149 sigma = .20; plt.plot(gaussian(100,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);
150 plt.show()
151 sigma = .20; plt.plot(gaussian(100,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);
152 plt.show()
153 np.arange(-(n_samples//2), (n_samples//2) + 1)
154 N=100; np.arange(-(N//2), (N//2) + 1)
155 N=100; np.arange(-N//2, (N//2) + 1)
156 N=100; np.arange(-(N)//2, (N//2) + 1)
157 N=100; np.arange(-(N+1)//2, (N//2) + 1)
158 N=100; np.arange(-(N-1)//2, (N//2) + 1)
159 _.shape
160 N=100; np.arange(-(N-1)//2, (N//2))
161 N=100; np.arange(-(N+1)//2, (N//2))
162 N=100; np.arange(-(N)//2-1, (N//2))
163 N=100; np.arange(-((N)//2-1), (N//2))
164 N=100; np.arange(-((N)//2-1), (N//2)+1)
165 _.shape
166 N=101; np.arange(-((N)//2-1), (N//2)+1)
167 _.shape
168 from importlib import reload
169 reload(hfft)
170 import hfft
171 reload(hfft)
172 from hfft import discrete_gaussian_kernel as dgk
173 N=101; np.arange(-((N)//2-1), (N//2)+1)
174 dgk(50,.1)
175 _.shape
176 sigma = .20; plt.plot(gaussian(100,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);
177 plt.show()
178 dgk(100,sigma).shape
179 gaussian(100,sigma).shape
180 dgk(100,sigma)
181 _.max()
182 gaussian(100,sigma).max()
183 gaussian(100,sigma) / (sigma*np.sqrt(2*np.pi))
184 _.max()
185 sigma = .20; plt.plot(gaussian(100,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);
186 plt.show()
187 sigma = .20; plt.plot(gaussian(100,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);
188 plt.show()
189 sigma = .20; plt.plot(gaussian(100,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);
190 plt.show()

```

```
191 sigma = .20; plt.plot(gaussian(100,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);  
192 plt.show()
```

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 plt.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
107 for sigma in np.logspace(-3, 8, base=2, num=20):
108    D = fft_gaussian(img, sigma, 'discrete')
109    gD = np.gradient(D)
110    Daa = dilate(aa(gD), 20).filled(1)
111    aas.append(Daa)
112    print(f"sigma={sigma:.3f}", "min: {:.6f}, max:{:.6f}".format(
113        Daa[~dil].min(), Daa[~dil].max()))
114    )
115
116 for sigma in np.logspace(-4, 8, base=2, num=50):
117    D = fft_gaussian(img, sigma, 'discrete')
118    gD = np.gradient(D)
119    Daa = dilate(aa(gD), 20).filled(1)
120    aas.append(Daa)
121    print(f"sigma={sigma:.3f}", "min: {:.6f}, max:{:.6f}".format(
122        Daa[~dil].min(), Daa[~dil].max()))

```

```

123
124
125 bb = lambda g: np.linalg.norm(np.array(
126     [[1+g[1]**2, -g[0]*g[1]],
127      [-g[0]*g[1], 1 + g[0]**2]]))
128 bb = lambda g: np.linalg.norm(np.array(
129     [[1+g[1]**2, -g[0]*g[1]],
130      [-g[0]*g[1], 1 + g[0]**2]]))
131 bb(gA)
132 get_ipython().set_next_input('man np.linalg.norm');get_ipython().run_line_magic('pinfo', 'np.linalg.norm')
133 ginv = lambda g: np.array(
134     [[1+g[1]**2, -g[0]*g[1]],
135      [-g[0]*g[1], 1 + g[0]**2]])
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
179 for sigma in np.logspace(-4, 8, base=2, num=50):
180     D = fft_gaussian(img, sigma, 'discrete')
181     gD = np.gradient(D)
182     Daa, Dbb = aa(gD), bb(gD)
183     Dcc = Daa / Dbb
184     aas = Daa[~dil].min(), Daa[~dil].max()
185     bbs = Dbb[~dil].min(), Dbb[~dil].max()
186     ccs = Dcc[~dil].min(), Dcc[~dil].max()

```

```

187 print(f"sigma={sigma:.3f}",
188     "\tmin: {:.6f},\tmax:{:.6f}" .format(*aas),
189     "\tmin: {:.6f},\tmax:{:.6f}" .format(*bbs),
190     "\tmin: {:.6f},\tmax:{:.6f}" .format(*ccs), sep='\n')
191
192
193
194 for sigma in np.logspace(-4, 8, base=2, num=50):
195     D = fft_gaussian(img, sigma, 'discrete')
196     gD = np.gradient(D)
197     Daa, Dbb = aa(gD), bb(gD)
198     Dcc = Dbb / Daa
199     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
258 data
259 table = pandas.DataFrame(data)
260 table
261 table.columns
262 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 σ=: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 σ=: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')

```

```

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/scalesweep_stitch_{sample_name}_{label}.png',
96                 dpi=300)
97
98     cfile = 'demo_output/scalesweep_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 matplotlib as mpl
8 import matplotlib.pyplot as plt
9
10 import itertools
11 from collections import deque
12
13 def rgb_to_widths(T):
14     """
15     this will take an RGB trace image (MxNx3) and return a 2D (MxN)
16     "labeled" trace corresponding to the traced pixel length.
17     there is no distinguishing between arteries and vessels
18
19     it's preferable to do this in real-time so only one tracefile
20     needs to be stored (making the sample folder less cluttered)
21     although obviously at the expense of storing a larger image
22     which is only needed for visualization purposes.
23
24     Input:
25         T: a MxNx3 RGB (uint8) array, where the colorations are
26             assumed as described in NOTES below.
27
28     Output:
29         widthtrace: a MxN array whose inputs describe the width of the
30         vessel (in pixels), see NOTES.
31
32     Notes:
33
34         The correspondence is as follows:
35         3 pixels: "#ff006f", # magenta
36         5 pixels: "#a80000", # dark red
37         7 pixels: "#a800ff", # purple
38         9 pixels: "#ff00ff", # light pink
39         11 pixels: "#008aff", # blue
40         13 pixels: "#8aff00", # green
41         15 pixels: "#ffc800", # dark yellow
42         17 pixels: "#ff8a00", # orange

```

```

43     19 pixels: "#ff0015"    # bright red
44
45 According to the original tracing protocol, the traced vessels are
46 binned into these 9 sizes. Vessels with a diameter smaller than 3px
47 are not traced (unless they're binned into 3px).
48
49 Note: this does *not* deal with collisions. If you pass anything
50 with addition (blended colors) as the ctraces are, you will have
51 trouble, as those will not be registered as any of the colors above
52 and will thus be ignored. If you want to handle data from both
53 arterial *and* venous layers, you should do so outside of this
54 function.
55 """
56
57 # a 2D picture to fix in with the pixel widths
58 W = np.zeros_like(T[:, :, 0])
59
60 for pix, color in TRACE_COLORS.items():
61
62     #ignore pixelwidths outside the specified range
63     # get the 2D indices that are that color
64     idx = np.where(np.all(T == color, axis=-1))
65     W[idx] = pix
66
67
68 return W
69
70 def merge_widths_from_traces(A_trace, V_trace, strategy='minimum'):
71 """
72 combine the widths from two RGB-traces A_trace and V_trace
73 and return one width matrix according to 'strategy'
74
75 Parameters
76 -----
77 A_trace: ndarray
78     an MxNx3 matrix, where each pixel (along the
79     last dimension) is an RGB triplet (i.e. each entry
80     is an integer between [0,256]. The colors each
81     correspond to those in TRACE_COLORS, and (255,255,255)
82     signifies "no vessel". This will normally correspond to
83     the sample's arterial trace.
84 V_trace: ndarray
85     an MxNx3 matrix the same shape and other
86     requirements as A_trace (see above). This will normally
87     correspond to the sample's venous trace.
88 strategy: keyword string
89     when A_trace and V_trace coincide at some entry,
90     this is the merging strategy. It should be a keyword
91     of one of the following choices:
92
93     "minimum": take the minimum width of the two traces
94         (default). this is the sensible option if you
95         are filtering out larger widths.
96     "maximum": take the maximum width of the two traces
97     "artery" or "A" or "top": take the width from A_trace
98     "vein" or "V" or "bottom": take the width from V_trace
99
100 Returns
101 -----
102     W : ndarray
103     a width-matrix where each entry is a number 0 (no vessel), 3,5,7,...19
104
105 Notes
106 -----

```

```

107 Since arteries grow over the veins on the PCSVN and are generally easier
108 to extract, it might be preferable to indicate "arteries". In reality,
109 each strategy is a compromise, and only by keeping track of both would
110 you get the complete picture.
111
112 No filtering out widths is done here.
113 """
114 assert A_trace.shape == V_trace.shape
115
116 A = rgb_to_widths(A_trace)
117 V = rgb_to_widths(V_trace)
118
119 # collisions (where are widths both reported)
120 c = (A!=0)& (V!=0)
121
122 W = np.maximum(A,V) # get the nonzero value
123 if strategy == 'maximum':
124     pass # already done, else rewrite the collisions
125 elif strategy in ('arteries', 'A', 'top'):
126     W[c] = A[c]
127 elif strategy in ('veins', 'V', 'bottom'):
128     W[c] = V[c]
129 else:
130     if strategy != 'minimum':
131         print(f"Warning: unknown merge strategy: {strategy}")
132         print("Defaulting to minimum strategy")
133
134     W[c] = np.minimum(A[c], V[c])
135
136 return W
137
138 def filter_widths(W, widths=None, min_width=3, max_width=19):
139 """
140 Filter a width matrix, removing widths according to rules.
141
142 This function will take a 2D matrix of vessel widths and
143 remove any widths outside a particular range (or alternatively,
144 that are not included in a particular list)
145
146 Should be roughly as easy as doing it by hand, except that you
147 won't have to rewrite the code each time.
148
149 Inputs:
150
151 W: a width matrix (2D matrix with elements 0,3,5,7,...19
152
153 min_width: widths below this will be excluded (default is
154     3, the min recorded width). assuming these
155     are ints
156
157 max_width: widths above this will be excluded (default is
158     19, the max recorded width)
159
160 widths: an explicit list of widths that should be returned.
161     in this case the above min & max are ignored.
162     this way you could include widths = [3, 17, 19] only
163 """
164
165 Wout = W.copy()
166 if widths is None:
167     Wout[W < min_width] = 0
168     Wout[W > max_width] = 0
169
170 else:

```

```

171     # use numpy.isin(T, widths) but that's only in version 1.13 and up
172     # of numpy this is basically the code for that though
173     to_keep = np.in1d(W, widths, assume_unique=True).reshape(W.shape)
174     Wout[~to_keep] = 0
175     return Wout
176
177
178 TRACE_COLORS = {
179     3: (255, 0, 111),
180     5: (168, 0, 0),
181     7: (168, 0, 255),
182     9: (255, 0, 255),
183     11: (0, 138, 255),
184     13: (138, 255, 0),
185     15: (255, 200, 0),
186     17: (255, 138, 0),
187     19: (255, 0, 21)
188 }
189
190
191 def widths_to_rgb(w, show_non_matches=False):
192     """Convert width matrix back to RGB values.
193
194     For display purposes/convenience. Return an RGB matrix
195     converting back from [3,5,7, ..., 19] -> TRACE_COLORS
196
197     this doesn't do any rounding (i.e. it ignores anything outside of
198     the default widths), but maybe you'd want to?
199     """
200     B = np.zeros((w.shape[0], w.shape[1], 3))
201
202     for px, rgb_triplet in TRACE_COLORS.items():
203         B[w == px, :] = rgb_triplet
204
205     if show_non_matches:
206         # everything in w not found in TRACE_COLORS will be black
207         B[w == 0, :] = (255, 255, 255)
208     else:
209         non_filled = (B == 0).all(axis=-1)
210
211         B[non_filled, :] = (255, 255, 255) # make everything white
212
213     # matplotlib likes the colors as [0,1], so....
214     return B / 255.
215
216
217 def _hex_to_rgb(hexstring):
218     """
219     there's a function that does this in matplotlib.colors
220     but its scaled between 0 and 1 but not even as an
221     array so this is just as much work
222
223     ##TODO rewrite everything so this is useful if it's not been
224     rewritten already.
225     """
226     triple = hexstring.strip("#")
227     return tuple(int(x, 16) for x in (triple[:2], triple[2:4], triple[4:]))
228
229
230 def skeletonize_trace(T, T2=None):
231     """
232     if T is a boolean matrix representing a trace, then thin it
233     if T is an RGB trace, then register it according to the

```

```

235     tracing protocol then thin it
236
237     if T2 is provided, do the same thing to T2 and then merge the two
238     """
239     if T.ndim == 3:
240         trace = (rgb_to_widths(T) > 0) # booleanize it
241     else:
242         trace = T.astype('bool')
243
244     thinned = thin(trace)
245
246     if T2 is None:
247         return thinned
248
249     else:
250         # do the same thing to second trace and merge it
251         if T2.ndim == 3:
252             trace_2 = (rgb_to_widths(T2) > 0) # booleanize it
253             thinned_2 = thin(trace_2)
254
255         return np.logical_or(thinned, thinned_2)
256
257 def precision(counts):
258
259     return int(t[0]) / int(t[0] + t[2])
260
261
262 def integrate_score(score, truth, mask=None):
263     """Integrate/sum a probability-like score over a ground truth subset.
264     Truth is a binary matrix
265     """
266
267     if mask is None:
268         if ma.is_masked(score):
269             plate = score.filled(0)
270         else:
271             plate = score
272
273         ground_truth = truth
274
275     else:
276         plate = score*(~mask)
277         ground_truth = truth*(~mask)
278
279     subset_sum = score[truth].sum()
280     total_sum = score.sum()
281
282     return subset_sum / total_sum
283
284
285 def confusion(test, truth=None, bg_mask=None, colordict=None, tint_mask=True):
286     """
287     distinct coloration of false positives and negatives.
288
289     colors output matrix with
290         true_pos if test[-] == truth[-] == 1
291         true_neg if test[-] == truth[-] == 0
292         false_neg if test[-] == 0 and truth[-] == 1
293         false_pos if test[-] == 1 and truth[-] == 0
294
295     if colordict is supplied: you supply a dictionary of how to
296     color the four cases. Spec given by the default below:
297
298     if tint mask, then the mask is overlaid on the image, not replacing totally

```

```

299     colordict = {
300         'TN': (247, 247, 247), # true negative
301         'TP': (0, 0, 0) # true positive
302         'FN': (241, 163, 64), # false negative
303         'FP': (153, 142, 195), # false positive
304         'mask': (247, 200, 200) # mask color (not used in MCC calculation)
305     }
306     """
307
308     if colordict is None:
309         #
310         colordict = {
311             'TN': (247, 247, 247), # true negative# 'f7f7f7'
312             'TP': (0, 0, 0), # true positive # '000000'
313             'FN': (241, 163, 64), # false negative # 'f1a340' orange
314             'FP': (153, 142, 195), # false positive # '998ec4' purple
315             'mask': (247, 200, 200) # mask color (not used in MCC calculation)
316         #
317         #colordict = {
318             '#        'TN': (49,49,49), # true negative# 'f7f7f7'
319             '#        'TP': (0, 0, 0), # true positive # '000000'
320             '#        'FN': (201,53,108), # false negative # 'f1a340' orange
321             '#        'FP': (0,112,163), # false positive # '998ec4' purple
322             '#        'mask': (247, 200, 200) # mask color (not used in MCC calculation)
323         #
324         colordict = {
325             'TP': (0,0,0),
326             'TN': (226,226,226),
327             'FN': (201,152,152),
328             'FP': (30,69,230),
329             'mask': (209,209,209)
330         }
331
332     #TODO: else check if mask is specified and add it as color of TN otherwise
333     if truth is None:
334         truth = np.zeros_like(test)
335
336     true_neg_color = np.array(colordict['TN'], dtype='f')/255
337     true_pos_color = np.array(colordict['TP'], dtype='f')/255
338     false_neg_color = np.array(colordict['FN'], dtype='f') /255
339     false_pos_color = np.array(colordict['FP'], dtype='f')/255
340     mask_color = np.array(colordict['mask'], dtype='f') /255
341
342     assert test.shape == truth.shape
343
344     # convert to bool
345     test, truth = test.astype('bool'), truth.astype('bool')
346
347     # RGB array size of test and truth for output
348     output = np.zeros((test.shape[0], test.shape[1], 3), dtype='f')
349
350     # truth conditions
351     true_pos = (test==truth & truth)
352     true_neg = (test==truth & ~truth)
353     false_neg = (truth & ~test)
354     false_pos = (test & ~truth)
355
356     output[true_pos,:,:] = true_pos_color
357     output[true_neg,:,:] = true_neg_color
358     output[false_pos,:,:] = false_pos_color
359     output[false_neg,:,:] = false_neg_color
360
361     # try to find a mask
362     if bg_mask is None:

```

```

363     try:
364         bg_mask = test.mask
365     except AttributeError:
366         # no mask is specified, we're done.
367         return output
368
369     # color the mask
370     if tint_mask:
371         output[bg_mask,:] += mask_color
372         output[bg_mask,:] /= 2
373     else:
374         output[bg_mask,:] = mask_color
375
376     return output
377
378 def binary_counts(test, truth, bg_mask=None, score_bg=False):
379     """returns TP,TN,FP,FN"""
380
381     true_pos = ((test == truth) & truth)
382     true_neg = ((test == truth) & ~truth)
383     false_neg = (truth & ~test)
384     false_pos = (test & ~truth)
385
386     if score_bg:
387         # take the classifications above as they are (nothing is masked)
388         pass
389     else:
390         # if no specified mask, check the test array itself?
391         if bg_mask is None:
392             try:
393                 bg_mask = test.mask
394             except AttributeError:
395                 # no mask is specified, we're done.
396                 bg_mask = np.zeros_like(test)
397
398         # only get stats in the plate
399         true_pos[bg_mask] = 0
400         true_neg[bg_mask] = 0
401         false_pos[bg_mask] = 0
402         false_neg[bg_mask] = 0
403
404     # now tally
405     TP = true_pos.sum()
406     TN = true_neg.sum()
407     FP = false_pos.sum()
408     FN = false_neg.sum()
409
410     return TP, TN, FP, FN
411
412 def compare_trace(approx, trace=None, filename=None,
413                   sample_dir=None, colordict=None):
414     """
415     compare approx matrix to trace matrix and output a confusion matrix.
416     if trace is not supplied, open the image from the tracefile.
417     if tracefile is not supplied, filename must be supplied, and
418     tracefile will be opened according to the standard pattern
419
420     colordict are parameters to pass to confusion()
421
422     returns a matrix
423     """
424
425     # load the tracefile if not supplied
426     if trace is None:

```

```

427     if filename is not None:
428         try:
429             trace = open_typefile(filename, 'trace')
430         except FileNotFoundError:
431             print("No trace file found matching ", filename)
432             print("no trace found. generating dummy trace.")
433             trace = np.zeros_like(approx)
434         else:
435             print("no trace supplied/found. generating dummy trace.")
436             trace = np.zeros_like(approx)
437
438 C = confusion(approx, trace, colordict=colordict)
439
440 return C
441
442
443 def mcc(test, truth=None, bg_mask=None, score_bg=False, return_counts=False):
444     """
445     Matthews correlation coefficient
446     returns a float between -1 and 1
447     -1 is total disagreement between test & truth
448     0 is "no better than random guessing"
449     1 is perfect prediction
450
451     bg_mask is a mask of pixels to ignore from the statistics
452     for example, things outside the placental plate will be counted
453     as "TRUE NEGATIVES" when there wasn't any chance of them not being
454     scored as negative. therefore, it's not really a measure of the
455     test's accuracy, but instead artificially pads the score higher.
456
457     setting bg_mask to None when test and truth are not masked
458     arrays should give you this artificially inflated score.
459     Passing score_bg=True makes this decision explicit, i.e.
460     any masks (even if supplied) will be ignored, and your count of
461     false positives will be inflated.
462
463     """
464     if truth is None:
465         truth = np.zeros_like(test)
466
467     TP, TN, FP, FN = binary_counts(test, truth, bg_mask=bg_mask,
468                                     score_bg=score_bg)
469
470
471     denom = np.sqrt(TP+FP)*np.sqrt(TP+FN)*np.sqrt(TN+FP)*np.sqrt(TN+FN)
472
473     if denom == 0:
474         # set MCC to zero if any are zero
475         m_score = 0
476     else:
477         m_score = ((TP*TN) - (FP*FN)) / denom
478
479     if return_counts:
480         return m_score, (TP, TN, FP, FN)
481     else:
482         return m_score
483
484
485 def mean_squared_error(A,B):
486     """
487     get mean squared error between two matrices of the same size
488
489     input:
490         A, B : two ndarrays of the same size.

```

```

491     output:
492
493     mse:    a single number.
494
495
496
497     try:
498         mse = ((A-B)**2).sum() / A.size
499
500     except ValueError:
501         print("inputs must be of the same size")
502         raise
503
504     return mse
505
506 def chain_lengths(iterable):
507
508     pos, s = 0, 0
509
510     for b, g in itertools.groupby(iterable):
511
512         if not b:
513             # alternative if the bottom doesn't work or something
514             #d = deque(enumerate(g,1), maxlen=1)
515             #pos += d[0][0] if d else 0
516
517             pos += sum((1 for i in g if not i))
518
519         else:
520
521             s = sum(g)
522
523             yield pos, s
524
525             pos += s
526
527     if not s:
528         # so it will return something even if iterable is empty
529         yield 0, 0
530
531
532 def _longest_chain_1d(iterable):
533     """ will return a tuple of ind, length
534     where ind is the position in the iterable the chain starts and length is the
535     length of the chain
536 """
537     return max(chain_lengths(iterable), key=lambda x: x[1])
538
539
540 def longest_chain(arr, axis):
541     """Find where the longest chain of boolean values and occurs across an array
542     and also return its length
543 """
544
545     C = np.apply_along_axis(_longest_chain_1d, axis, arr.astype('bool'))
546
547     start_inds, chain_lens = np.split(C, 2, axis)
548
549     return np.squeeze(start_inds), np.squeeze(chain_lens)
550
551
552 def _bunch_hists(H, bunches):
553
554     return np.stack((np.sum(np.atleast_2d(H[b,:]), axis=0) for b in bunches))

```

```

555
556
557 def scale_to_width_plots(multiscale_approx, max_labels, widths, scales,
558                         bunches=None, cmap=None, approx_method=None,
559                         figsize=(13,14), style='seaborn', bunch_until=None):
560     """
561     multiscale_approx is a 3d boolean array whose first dimension is scale
562     max_labels is a 2d array of integers that say where the max value of
563     F occured. you can get max_labels by running V.argmax(axis=0)
564
565     in widths, each pixel has a unique width
566
567     bunches.flatten() should be the same as arange(scales)
568     but can be something like
569     ( (0,1,2,3), (4,5), 6, 7, 8, (9,10,11) )
570
571     or even
572
573     (2,3,4,5,(0,1,6,7))
574
575     this is to prevent similar scales from clogging, you can just bin them
576     all together.
577
578     approx method is a label to use in the fig titles
579     """
580
581     if bunches is None:
582         if bunch_until is not None:
583             indices = list(range(len(scales)))
584             bunches = [indices[:bunch_until],] + indices[bunch_until:]
585
586     plt.style.use(style)
587
588     fig, ax = plt.subplots(nrows=2, ncols=1, figsize=figsize)
589
590     A = multiscale_approx # easier to work with
591
592     wbins = np.arange(3,20,2) # bins of widths in ground truth
593
594     max_hists = [[np.sum((max_labels == s) & (widths==w)) for w in wbins]
595                  for s in range(len(scales))]
596
597     hists = np.array([[np.sum((widths==w) & A[n]) for w in wbins]
598                      for n in range(len(scales))])
599
600
601     if cmap is None:
602         # this will just use the default cycle of colors
603         colors = np.repeat(None, len(scales))
604     else:
605         if not isinstance(cmap, mpl.colors.LinearSegmentedColormap):
606             # try this
607             cmap = plt.get_cmap(cmap)
608
609         colors = cmap(np.linspace(0,1,len(scales)))
610
611     labels = [rf'$\sigma_k={sigma:.2f}$'
612               for k, sigma in enumerate(scales,1)]
613
614
615     # number of true positives, false negatives for each width
616     tp_hists = [np.sum((widths==w) & A.any(axis=0)) for w in wbins]
617     fn_hists = [np.sum((widths==w) & ~A.any(axis=0)) for w in wbins]
618

```

```

619 if bunches is not None:
620     hists = _bunch_hists(hists, bunches)
621
622     # just return \sigma_{1,2,3} or something rather than listing
623     bunch_label = lambda b: r"\sigma".format(','.join((str(x+1)
624                                                 for x in b)
625                                         ))
626
627     labels = [labels[b] if np.isscalar(b) else bunch_label(b)
628               for b in bunches]
629
630     if cmap is None:
631         # just make it the appropriate length
632         cmap = np.repeat(None, len(bunches))
633     else:
634         colors = [colors[b] if np.isscalar(b) else colors[b[0]]
635                   for b in bunches]
636
637     ax[0].bar(wbins, tp_hists, color=(0.6,0.6,0.6),
638               label='# true positives')
639     ax[0].bar(wbins, fn_hists, bottom=tp_hists, color=(1,.8,.8),
640               label='# false negatives')
641
642     for h, mh, label, color in zip(hists, max_hists, labels, colors):
643         ax[0].plot(wbins, h, label=label, color=color)
644         ax[1].plot(wbins, mh, label=label, color=color)
645
646
647     ax[0].set_xticks(wbins)
648     ax[0].set_xlabel('vessel widths (ground truth), pixels')
649     ax[0].set_ylabel('# pixels')
650     ax[0].set_xlim(2,21)
651
652     ax[1].set_xticks(wbins)
653     ax[1].set_xlabel('vessel widths (ground truth), pixels')
654     ax[1].set_ylabel('# pixels')
655     ax[1].set_xlim(2,21)
656
657     title = 'pixels reported per scale'
658     max_title = r'pixel widths of true positives by scale of Vmax'
659     if approx_method is not None:
660         title += f'({approx_method})'
661         max_title += f'({approx_method})'
662
663     ax[0].set_title(title)
664     ax[1].set_title(max_title)
665     ax[0].legend(loc='best', labelspacing=0.2)
666     ax[1].legend(loc='best', labelspacing=0.2)
667
668     fig.tight_layout()
669     return fig, ax
670
671
672 def scale_to_argmax_plot(max_labels, widths, scales, normalize=False,
673                          bunches=None, cmap=None, figsize=(13,10),
674                          style='seaborn-paper', bunch_until=None):
675     """
676     if normalize, normalize each scale over columns (i.e. all widths)
677     multiscale_approx is a 3d boolean array whose first dimension is scale
678     max_labels is a 2d array of integers that say where the max value of
679     F occurred. you can get max_labels by running V.argmax(axis=0)
680
681     in widths, each pixel has a unique width
682

```

```

683 bunches.flatten() should be the same as arange(scales)
684 but can be something like
685 ( (0,1,2,3), (4,5), 6, 7, 8, (9,10,11) )
686
687 or even
688
689 (2,3,4,5,(0,1,6,7))
690
691 this is to prevent similar scales from clogging, you can just bin them
692 all together.
693
694 approx method is a label to use in the fig titles
695 """
696
697 if bunches is None:
698     if bunch_until is not None:
699         indices = list(range(len(scales)))
700         bunches = [indices[:bunch_until],] + indices[bunch_until:]
701
702 plt.style.use(style)
703
704 fig, ax = plt.subplots(figsize=figsize)
705
706 wbins = np.arange(3,20,2) # bins of widths in ground truth
707
708 max_hists = np.array([[np.sum((max_labels == s) & (widths==w)) for w in wbins]
709                         for s in range(len(scales))])
710
711 if normalize:
712     max_hists = max_hists / max_hists.sum(axis=1, keepdims=True)
713
714 if cmap is None:
715     # this will just use the default cycle of colors
716     colors = np.repeat(None, len(scales))
717 else:
718     if not isinstance(cmap, mpl.colors.LinearSegmentedColormap):
719         # try this
720         cmap = plt.get_cmap(cmap)
721
722     colors = cmap(np.linspace(0,1,len(scales)))
723
724 labels = [rf'\sigma_k=\sigma{:.2f}'
725             for k, sigma in enumerate(scales,1)]
726
727 # number of true positives, false negatives for each width
728
729 if bunches is not None:
730
731     # just return \sigma_{1,2,3} or something rather than listing
732     bunch_label = lambda b: r"\sigma".format(',').join(( str(x+1)
733                                         for x in b
734                                         ))
735
736     labels = [labels[b] if np.isscalar(b) else bunch_label(b)
737               for b in bunches]
738
739     if cmap is None:
740         # just make it the appropriate length
741         cmap = np.repeat(None, len(bunches))
742     else:
743         colors = [colors[b] if np.isscalar(b) else colors[b[0]]
744                   for b in bunches]
745
746 #ax.bar(wbins, tp_hists, color=(0.6,0.6,0.6),
747 #        label='# true positives')

```

```

747 #ax.bar(wbins, fn_hists, bottom=tp_hists, color=(1,.8,.8),
748 #           label='# false negatives')
749
750 for mh, label, color in zip(max_hists, labels, colors):
751     ax.plot(wbins, mh, label=label, color=color)
752
753
754
755 ax.set_xticks(wbins)
756 ax.set_xlabel('vessel widths (ground truth), pixels')
757 if normalize:
758     ax.set_ylabel('# pixels identified by scale /'
759                 '# pixels identified by all scales')
760 else:
761     ax.set_ylabel('# pixels')
762
763 ax.set_xlim(2,21)
764
765 max_title = r'pixel widths of true positives by scale of  $V_{\max}$ '
766
767 ax.set_title(max_title)
768 ax.legend(loc='best', labelspacing=0.2)
769
770 fig.tight_layout()
771 return fig, ax
772
773
774 if __name__ == "__main__":
775
776     import matplotlib.pyplot as plt
777     from skimage.data import binary_blobs
778
779     A = binary_blobs()
780     B = binary_blobs()
781
782     true_neg_color = np.array([247, 247, 247], dtype='f') # 'f7f7f7',
783     true_pos_color = np.array([0, 0, 0], dtype='f') # '000000',
784     false_neg_color = np.array([241, 163, 64], dtype='f')# 'f1a340',
785     false_pos_color = np.array([153, 142, 195], dtype='f') # '998ec4',
786
787     C = confusion(A,B)
788
789     fig, (ax0, ax1, ax2) = plt.subplots(nrows=1,
790                                         ncols=3,
791                                         figsize=(8, 2.5),
792                                         sharex=True,
793                                         sharey=True)
794
795     ax0.imshow(A, cmap='gray')
796     ax0.set_title('A')
797     ax0.axis('off')
798     ax0.set_adjustable('box-forced')
799
800     ax1.imshow(B, cmap='gray')
801     ax1.set_title('B')
802     ax1.axis('off')
803     ax1.set_adjustable('box-forced')
804
805     ax2.imshow(C)
806     ax2.set_title('confusion matrix of A and B')
807     ax2.axis('off')
808     ax2.set_adjustable('box-forced')
809
810 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
77         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)

```

listings/stump_test.py

```

1 #!/usr/bin/env python3
2
3 from placenta import (get_named_placenta, list_placentas, open_typefile,

```

```

4                                     show_mask, measure_ncs_markings)
5 import numpy as np
6 import matplotlib.pyplot as plt
7 from preprocessing import mask_stump
8 import numpy.ma as ma
9 import sys
10 from skimage.exposure import equalize_adapthist
11 from scipy.ndimage import label
12 from skimage.morphology import thin, binary_opening, disk, convex_hull_image
13
14
15 for n, filename in enumerate(list_placentas('T-BN')):
16     print(filename)
17
18     raw = open_typefile(filename, 'raw')
19     #raw = (equalize_adapthist(raw))
20     img = get_named_placenta(filename)
21
22     stump = mask_stump(raw, mask=img.mask, mask_only=True)
23     if not stump.any():
24         print('no stump found at all, continuing')
25         plt.imshow(raw)
26         plt.show()
27         continue
28
29     stump_labs = label(stump)
30
31     ucip_mid, _ = measure_ncs_markings(filename=filename)
32
33     labeled, n_labs = label(stump)
34     dists = list()
35     for lab in range(1, n_labs+1):
36         X, Y = np.where(labeled==lab)
37         dists.append((lab, min([(x-ucip_mid[0])**2 + (y-ucip_mid[1])**2
38                               for x,y in zip(X,Y)])))
39
40     closest_lab = sorted(dists, key=lambda v: v[1])[0][0]
41
42     closest_stump = (labeled==closest_lab)
43     closest_opening = binary_opening(closest_stump, disk(10))
44     final_stump = convex_hull_image(closest_opening)
45
46     fig, ax = plt.subplots(ncols=3, nrows=2, figsize=(30,12))
47
48     ax[0,0].imshow(raw)
49     ax[0,0].set_title(filename)
50
51     newmask = show_mask(ma.masked_array(img, mask=stump))
52     ax[0,1].imshow(newmask)
53     ax[0,2].imshow(stump*1. + img.mask*2, vmin=0, vmax=3)
54     ax[1,0].imshow(closest_stump)
55     ax[1,1].imshow(closest_opening)
56     ax[1,2].imshow(final_stump)
57
58     plt.show()
59
60     if input('make more?') == 'n':
61         sys.exit(0)

```

listings/threshold_demo.py

```

1 #!/usr/bin/env python3
2
3 import numpy as np
4 import matplotlib.pyplot as plt
5 import matplotlib as mpl
6 from skimage.util import img_as_float
7 from skimage.io import imread
8 from placenta import (get_named_placenta, list_placentas, cropped_args)
9
10 import numpy.ma as ma
11 from plate_morphology import dilate_boundary
12 import os.path, os
13 from postprocessing import dilate_to_rim
14 from scoring import confusion, mcc
15 from placenta import open_tracefile, open_typefile
16 from preprocessing import inpaint_hybrid
17 from placenta import measure_ncs_markings, add_ucip_to_mask
18
19 import json
20
21 mcss = list()
22 precs = list()
23
24 scales = np.logspace(-1.5, 3.5, num=20, base=2)
25 threshold = .15
26 neg_threshold = 0.001
27 max_pos_scale = -6
28 max_neg_scale = 2
29 beta = 0.10
30 gamma = 1.0
31
32 cm = mpl.cm.plasma
33 #cmscales = mpl.cm.magma
34 cm.set_bad('k', 1) # masked areas are black, not white
35 #cmscales.set_bad('w', 1)
36
37 QUALITY = 3
38 for filename in list_by_quality(QUALITY):
39
40     cimg = open_typefile(filename, 'raw')
41     ctrace = open_typefile(filename, 'ctrace')
42     trace = open_tracefile(filename)
43     img = get_named_placenta(filename)
44     crop = cropped_args(img)
45     ucip = open_typefile(filename, 'ucip')
46     img = inpaint_hybrid(img)
47
48     # make the size of figures more consistent
49     if img[crop].shape[0] > img[crop].shape[1]:
50         # and rotating it would be fix all this automatically
51         cimg = np.rot90(cimg)
52         ctrace = np.rot90(ctrace)
53         trace = np.rot90(trace)
54         img = np.rot90(img)
55         ucip = np.rot90(ucip)
56         crop = cropped_args(img)
57
58     ucip_midpoint, resolution = measure_ncs_markings(ucip)
59     ucip_mask = add_ucip_to_mask(ucip_midpoint, radius=60, mask=img.mask)
60
61     name_stub = filename.rstrip('.png').strip('T-')
62
63     F = np.stack([frangi_from_image(img, sigma, beta=beta, gamma=gamma, dark_bg=False,

```

```

65                     signed_frangi=True, dilation_radius=20,
66                     rescale_frangi=True)
67                     for sigma in scales])
68 # need to fix this in the signed_frangi logic
69 F = F*~dilate_boundary(None, mask=img.mask, radius=20)
70
71 Fmax = (F*(F>0))[:max_pos_scale].max(axis=0)
72 Fneg = -(F*(F<0))[:max_neg_scale].min(axis=0)
73
74 approx = Fmax > threshold
75 rim_approx = (Fneg > neg_threshold)
76 skel = thin(approx)
77 completed = connect_iterative_by_label(skel, Fmax, max_dist=100)
78 completed_dilated = dilate_to_rim(completed, rim_approx, max_radius=10)
79 approx_dilated = dilate_to_rim(approx, rim_approx, max_radius=10)
80
81 network = np.maximum(skel*3., (completed & ~skel)*2)
82 network = np.maximum(network, rim_approx*1.)
83
84
85 precision = lambda t: int(t[0]) / int(t[0] + t[2])
86
87 mcc_FA, counts_FA = mcc(approx, trace, bg_mask=img.mask, return_counts=True)
88 mcc_FAD, counts_FAD = mcc(approx_dilated, trace, bg_mask=img.mask, return_counts=True)
89 mcc_NCD, counts_NCD = mcc(completed_dilated, trace, bg_mask=img.mask, return_counts=True)
90
91 prec_FA = precision(counts_FA)
92 prec_FAD = precision(counts_FAD)
93 prec_NCD = precision(counts_NCD)
94
95 mccs.append((name_stub, mcc_FA, mcc_FAD, mcc_NCD))
96 precs.append((name_stub, prec_FA, prec_FAD, prec_NCD))
97
98 fig, ax = plt.subplots(nrows=2, ncols=3, figsize=(20,12))
99 A = ax.ravel()
100
101 A[0].imshow(cimg[crop])
102 A[0].set_title(name_stub)
103
104 A[1].imshow(ma.masked_where(Fmax==0, Fmax)[crop], cmap=cm, vmin=0, vmax=1)
105 A[1].set_title(rf'Vmax,  $\beta = \text{beta}: .2f, \gamma = \text{gamma}: .3f'$ ')
106
107 A[2].imshow(network[crop], cmap=plt.cm.magma)
108 A[2].set_title('skeleton, completed network, and rim_approx')
109
110 A[3].imshow(confusion(approx, trace, bg_mask=img.mask)[crop])
111 A[3].set_title(fr'fixed  $\alpha = \text{threshold}: .2f$ ', loc='left')
112 A[3].set_title(f'MCC: {mcc_FA:.2f}\n' f'precision: {prec_FA:.2%}', loc='right')
113
114 A[4].imshow(confusion(approx_dilated, trace, bg_mask=img.mask)[crop])
115 A[4].set_title(fr'dilate_to_rim  $\alpha = \text{threshold}: .2f$ ', loc='left')
116 A[4].set_title(f'MCC: {mcc_FAD:.2f}\n' f'precision: {prec_FAD:.2%}', loc='right')
117
118 A[5].imshow(confusion(completed_dilated, trace, bg_mask=img.mask)[crop])
119 A[5].set_title(fr'network_completed, dilated', loc='left')
120 A[5].set_title(f'MCC: {mcc_NCD:.2f}\n' f'precision: {prec_NCD:.2%}', loc='right')
121
122 [a.axis('off') for a in A]
123 fig.tight_layout()
124 plt.show()
125
126
127
128

```

```
129
130
131 runlog = { 'mccs':mccs,
132     'precs':precs,
133     'scales':list(scales),
134     'beta':beta,
135     'gamma':gamma,
136     'max_pos_scale':max_pos_scale,
137     'max_neg_scale':max_neg_scale,
138     'threshold':threshold,
139     'neg_threshold':neg_threshold
140 }
141
142
143 with open(f'output/network_completion_{QUALITY}.json', 'w') as f:
144     json.dump(runlog, f, indent=True)
```

APPENDIX B
3D VISUALIZATION OF THE FRANGI FILTER

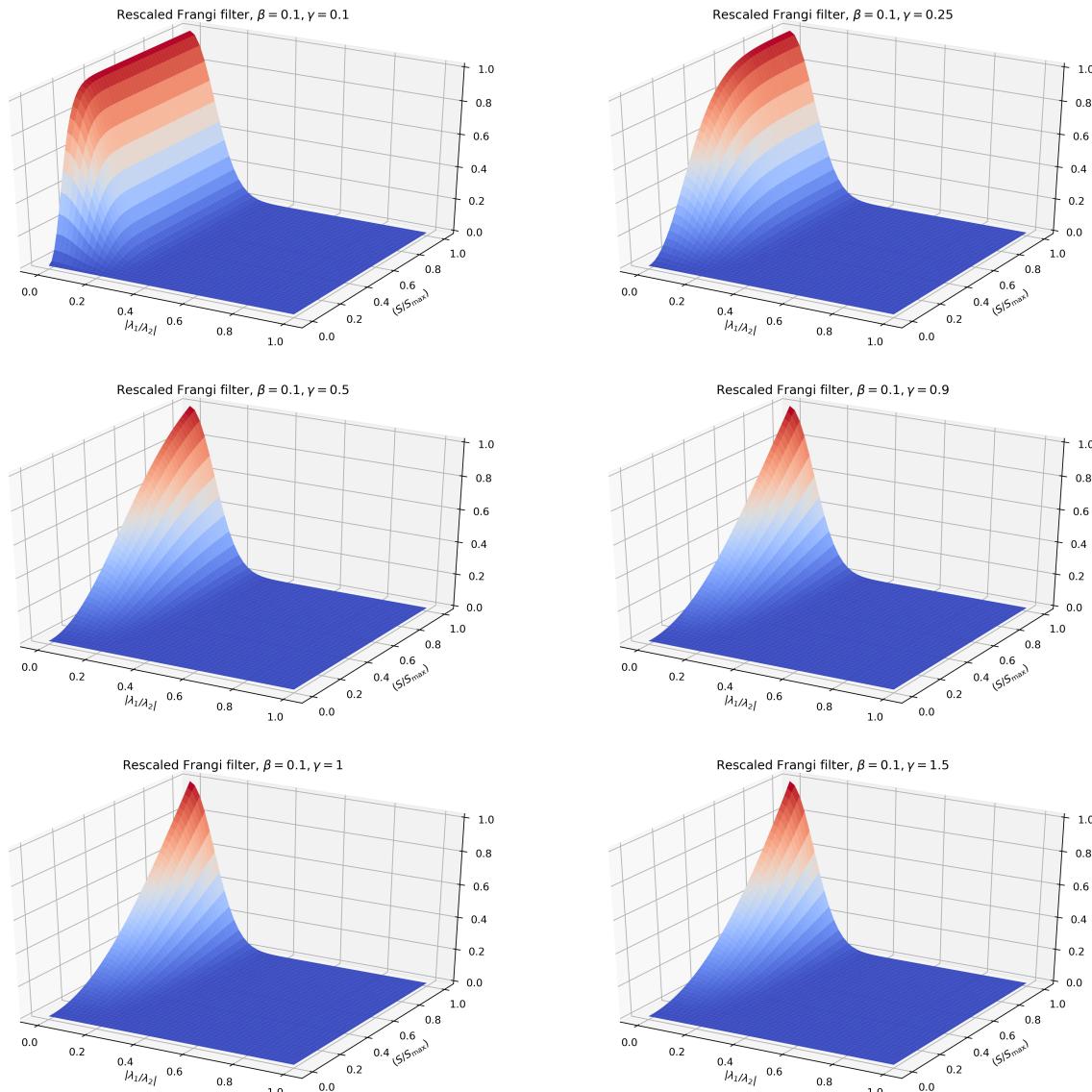


FIGURE 45: 3D graph of the Frangi Vesselness Measure, variable γ , $\beta = 0.1$

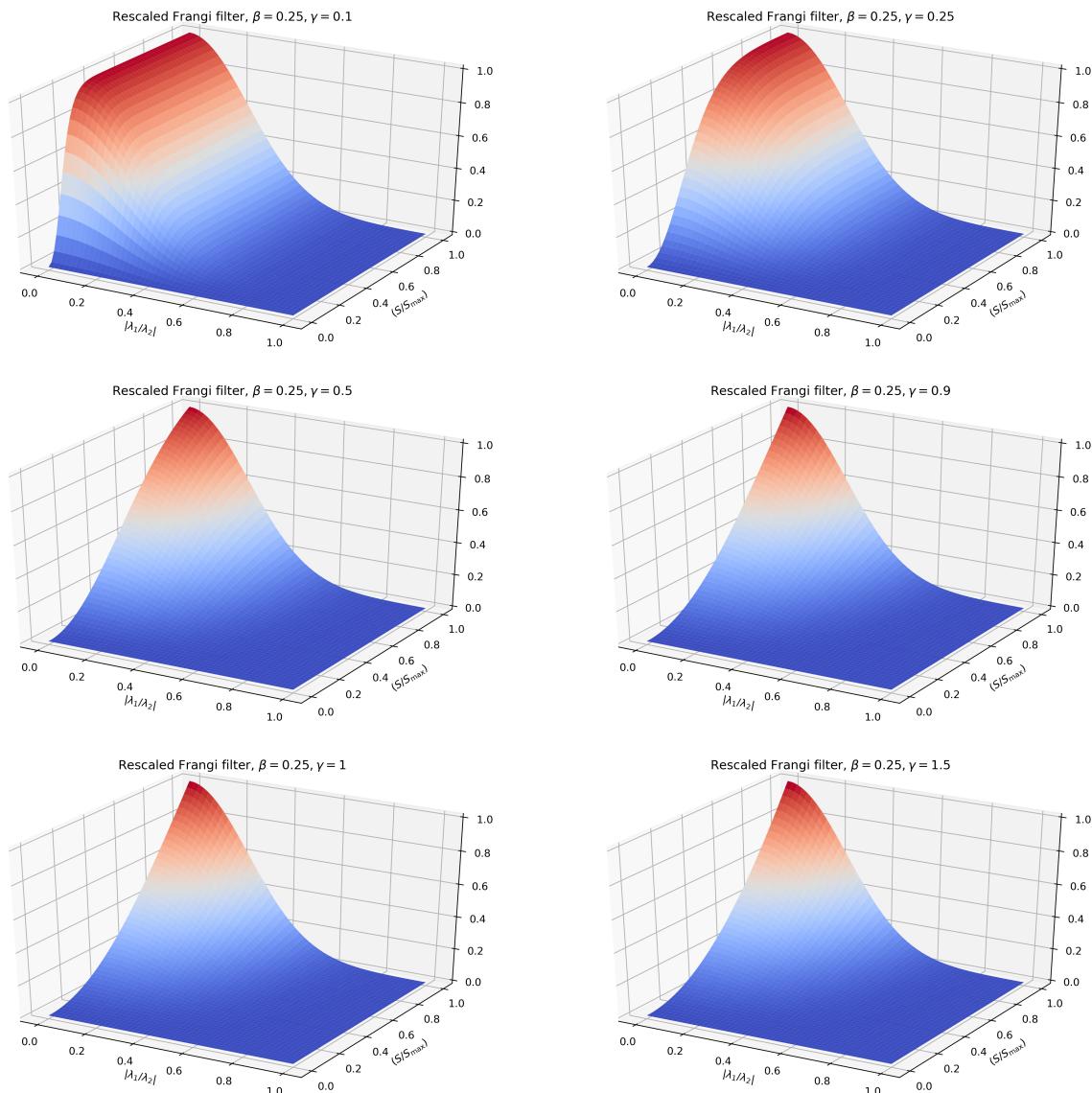


FIGURE 46: 3D graph of the Frangi Vesselness Measure, variable γ , $\beta = 0.25$

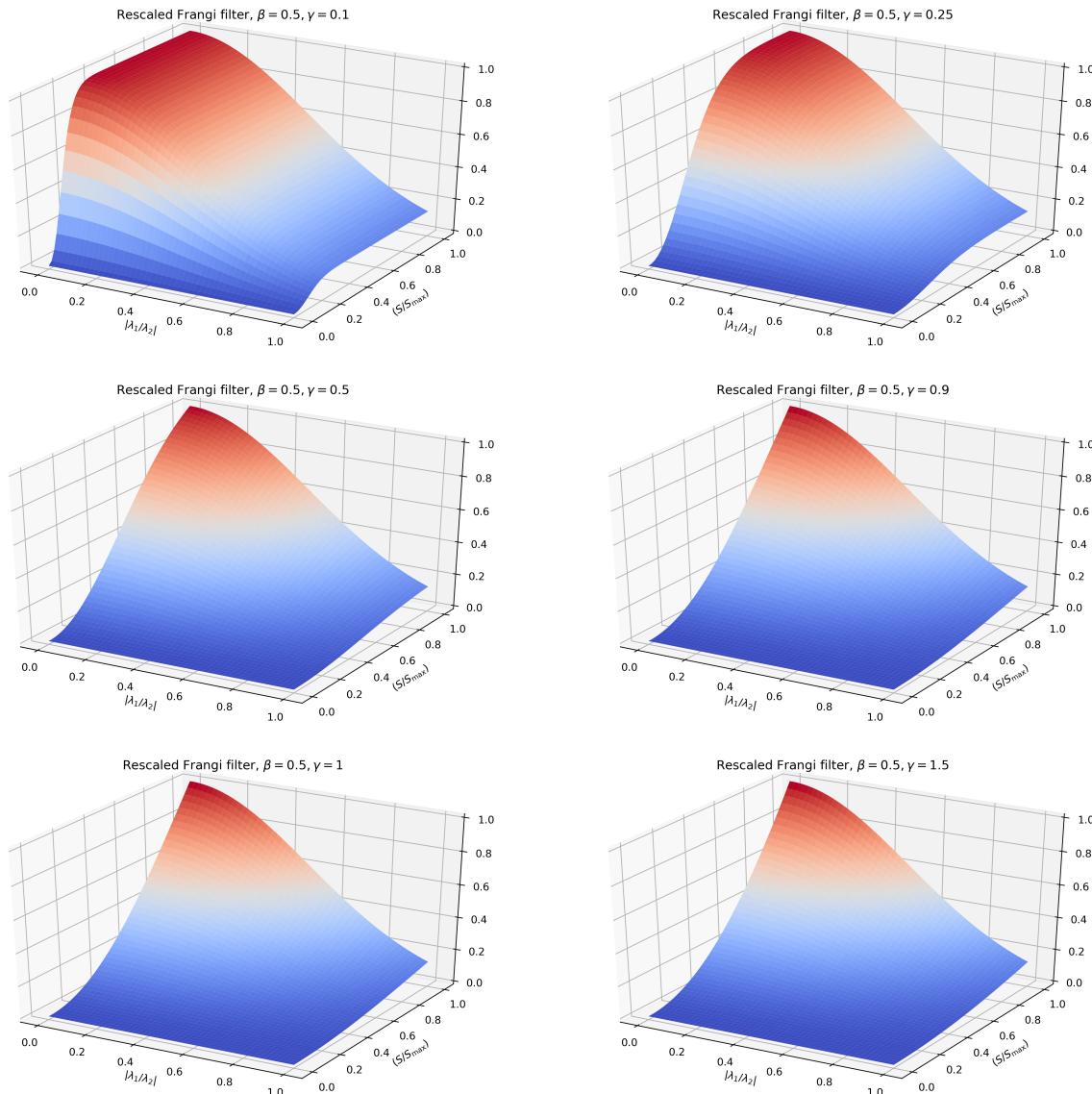


FIGURE 47: 3D graph of the Frangi Vesselness Measure, variable γ , $\beta = 0.5$

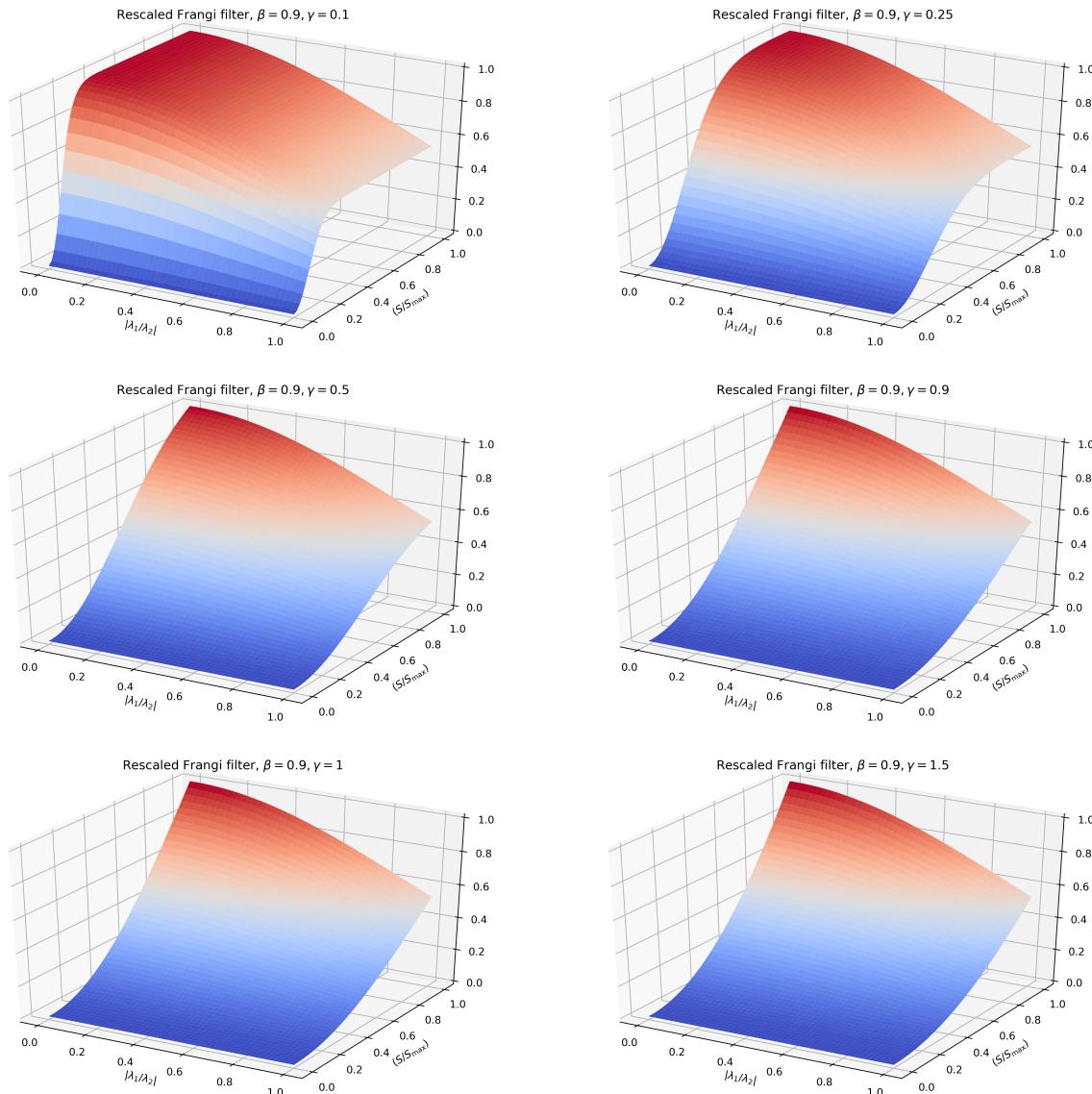


FIGURE 48: 3D graph of the Frangi Vesselness Measure, variable γ , $\beta = 0.9$

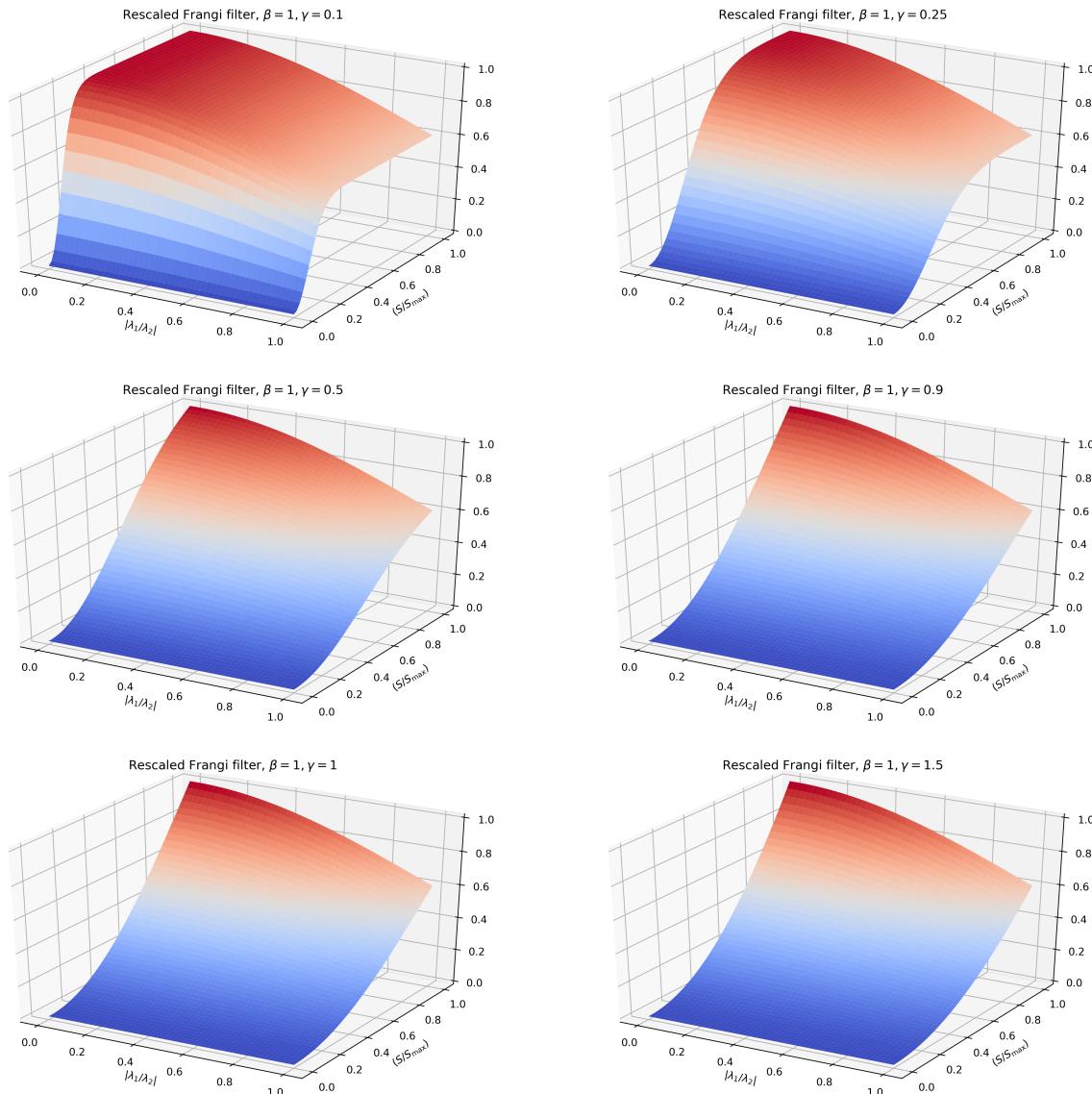


FIGURE 49: 3D graph of the Frangi Vesselness Measure, variable γ , $\beta = 1$

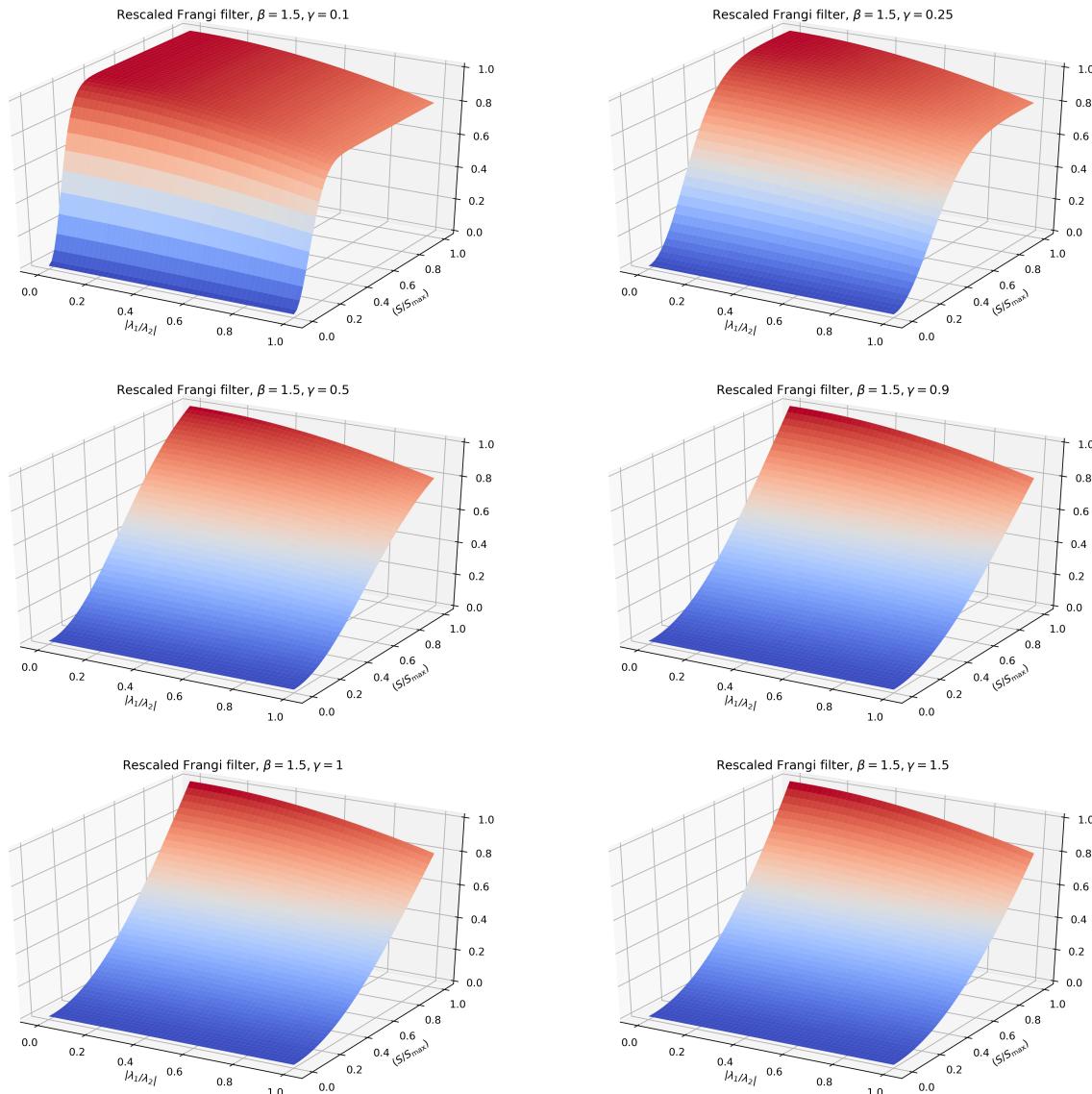


FIGURE 50: 3D graph of the Frangi Vesselness Measure, variable γ , $\beta = 1.5$

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] K. Y. Djima, C. Salafia, R. K. Miller, R. Wood, P. Katzman, C. Stodgell, and J.-M. Chang, Placenta **57**, 292 (2017).
- [4] 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).
- [5] 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.
- [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] R. Horn and C. Johnson, *Matrix Analysis, Matrix Analysis* (Cambridge University Press, ADDRESS, 2012).
- [10] X. Jiao and H. Zha, in *Proceedings of the 2008 ACM symposium on Solid and physical modeling*, ACM (PUBLISHER, ADDRESS, 2008), pp. 159–170.
- [11] R. Burden and J. Faires, *Numerical Analysis*, 9 ed. (Brooks/Cole, ADDRESS, 2011).
- [12] 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.
- [13] Y. Sato, S. Nakajima, N. Shiraga, H. Atsumi, S. Yoshida, T. Koller, G. Gerig, and R. Kikinis, Medical image analysis **2**, 143 (1998).

- [14] 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.
- [15] S. D. Olabarriaga, M. Breeuwer, and W. Niessen, in *International Congress Series*, Elsevier (PUBLISHER, ADDRESS, 2003), Vol. 1256, pp. 1191–1196.
- [16] J. J. Koenderink, *Biological Cybernetics* **50**, 363 (1984).
- [17] J. Sporrings, in *Gaussian Scale-Space Theory*, edited by L. Florack, M. Nielsen, and P. Johansen (Kluwer Academic Publishers, Norwell, MA, USA, 1997).
- [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. Damelin and N. Hoang, *International Journal of Mathematics and Mathematical Sciences* **2018**, (2018).
- [28] H. Lange, in *Medical Imaging 2005: Image Processing*, International Society for Optics and Photonics (PUBLISHER, ADDRESS, 2005), Vol. 5747, pp. 2183–2193.
- [29] B. Matthews, *Biochimica et Biophysica Acta (BBA) - Protein Structure* **405**, 442 (1975).
- [30] L. Grady, *IEEE Trans. Pattern Anal. Mach. Intell.* **28**, 1768 (2006).

- [31] C. Anghel, K. Archer, J.-M. Chang, A. Cochran, A. Radulescu, C. M. Salafia, R. Turner, K. Y. Djima, and L. Zhong, *Understanding Complex Biological Systems with Mathematics* (Springer, ADDRESS, 2018), pp. 171–196.
- [32] T. Ridler, S. Calvard, *et al.*, IEEE trans syst Man Cybern **8**, 630 (1978).
- [33] Z. Guo and R. W. Hall, Communications of the ACM **32**, 359 (1989).
- [34] I. Laptev, H. Mayer, T. Lindeberg, W. Eckstein, C. Steger, and A. Baumgartner, Machine Vision and Applications **12**, 23 (2000).