

## **ABSTRACT**

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

**By**

**Lucas Allen Wukmer**

**December 2018**

Recent statistical analysis of placental features has suggested the usefulness of studying key features of the placental chorionic surface vascular network (PCSVN) as a measure of overall neonatal health [1]. A recent study has suggested that reliable reporting of these features may be useful in identifying risks of certain neurodevelopmental disorders at birth. The necessary features can be extracted from an accurate tracing of the surface vascular network, but such tracings must still be done manually, with significant user intervention. Automating this procedure would not only allow more data acquisition to study the potential effects of placental health on later conditions, but may ideally serve as a real-time diagnostic for neonatal risk factors as well.

Much work has been to develop reliable vascular extraction methods for well-known image domains (such as retinal MRA images) using Hessian-based filters, namely the (multiscale) Frangi filter. It is desirable to extend these technique to study placental images, but this approach is greatly hindered by the comparative irregularity of the placental surface as a whole, which introduces significant noise into the image domain. Prior work [2] has made to apply an additional local curvilinear filter to the

Frangi result in an effort to remove some noise from the final extraction.

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

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

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

A THESIS

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

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

Committee Members:

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

College Designee:  
Tangan Gao, Ph.D.

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

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

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

By

Lucas Allen Wukmer

**COMMITTEE MEMBERS**

---

Jen-Mei Chang, Ph.D. (Chair) Mathematics and Statistics

---

James von Brecht, Ph.D. Mathematics and Statistics

---

William Ziemer, Ph.D. Mathematics and Statistics

**ACCEPTED AND APPROVED ON BEHALF OF THE UNIVERSITY**

---

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

California State University, Long Beach

December 2018

## **ACKNOWLEDGEMENTS**

Thank you to people for things.

Thank you for reading this.

## TABLE OF CONTENTS

	Page
ACKNOWLEDGEMENTS .....	iii
LIST OF TABLES .....	vi
LIST OF FIGURES .....	vii
 CHAPTER	
1. INTRODUCTION .....	1
2. MATHEMATICAL METHODS .....	3
Problem Setup in Image Processing .....	3
Differential Geometry .....	4
Preliminaries of Differential Geometry .....	4
Curvature of a surface and its calculation .....	7
Principal Curvatures and Principal Directions .....	14
The Weingarten map and Principal Curvatures of a Cylindrical Ridge .....	20
The Frangi Filter: Uniscale .....	24
Anisotropy Measure .....	25
Structureness measure .....	27
The Frangi vesselness measure .....	28
The Frangi vesselness filter: Choosing parameters $\beta$ and $c$ .	28
Linear Scale Space Theory .....	30
Axioms .....	33
Uniqueness of the Gaussian Kernel.....	35
Scale Spaces over Discrete Structures.....	39
The Frangi Filter: A multiscale approach .....	41
Thresholding .....	42
Calculating the 2D Hessian .....	43
Convolution Speedup via FFT .....	43
Fourier Transform of a continuous 1D signal.....	43
Fourier Transform of a Discrete 1D signal .....	44
2D DFT Convolution Theorem .....	45
FFT .....	47

APPENDIX	Page
3. RESEARCH PROTOCOL .....	52
Samples / Image Domain .....	52
A representative sample .....	52
Knowns and Unknowns .....	54
Data Cleaning and Preprocessing .....	55
Boundary Dilation .....	57
Cut Removal .....	59
Deglaring .....	60
Multiscale Setup .....	64
4. IMPLEMENTATIONS .....	69
Calculating the Hessian via FFT .....	69
Postprocessing Techniques .....	74
Method A: Fixed Threshold .....	75
Method B: Percentile Based Merging .....	75
Method C: Scale-Based Random Walker .....	76
Method D: Scale Based Sieving .....	76
5. RESULTS AND ANALYSIS .....	77
Sample visual output .....	77
Binary Classificaitons and the confusion matrix .....	77
A Source of “False Negatives” in the NCS data set .....	77
Variations in the Data Set and Imperfections of the Ground Truth .....	77
Results .....	82
Answer Research Questions .....	82
6. CONCLUSION .....	83
Future research directions .....	83
APPENDICES .....	86
A. CODE LISTINGS .....	87
B. 3D VISUALIZATION OF THE FRANGI FILTER .....	153
BIBLIOGRAPHY .....	160

## LIST OF TABLES

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

## LIST OF FIGURES

FIGURE	Page
1 Tangent plane of a graph .....	8
2 The graph of a cylindrical ridge of radius $r$ .....	20
3 The principal eigenvectors at a ridge like structure .....	26
4 Dependence of the Anisotropy Factor on its Parameter .....	31
5 Dependence of the Structureness Factor on its Parameter.....	32
6 A representative placental sample and tracing.....	53
7 Preprocessed files from an NCS sample.....	56
8 Effect of boundary dilation on Frangi responses.....	58
9 Removing a “cut” from the placental plate. ....	60
10 Deglaring a sample using a hybrid inpainting method .....	61
11 Comparison of glare inpainting methods (detail) .....	62
12 Signed Frangi output (plate and inset) (Example 1) .....	67
13 Signed Frangi output (plate and inset) (Example 1).....	68
14 Compatibility of Gaussian convolution strategies .....	70
15 Iterative Gaussian blur .....	72
16 Image cross-section of Gaussian blurred images .....	72
17 Image cross-section of Frangi targets images.....	72
18 Image cross-section of Gaussian blurred images $\sigma = 5$ .....	73
19 Image cross-section of Frangi targets images $\sigma = 5$ .....	73
20 Time required.....	74
21 Sample Multiscale Frangi output ( $\beta = 0.35$ ) with simple segmentation strategies (Example 1).....	78
22 Sample Multiscale Frangi output ( $\beta = 0.35$ ) with simple segmentation strategies (Example 2).....	79
23 Demonstration of postprocessing techniques .....	80
24 “True” false positives and “False” false positives.....	81
25 Signed Frangi output (plate and inset) (Example 1) .....	84
26 Signed Frangi output (plate and inset) (Example 1).....	85
27 3D graph of the Frangi Vesselness Measure, variable $\gamma$ , $\beta = 0.1$ .....	154
28 3D graph of the Frangi Vesselness Measure, variable $\gamma$ , $\beta = 0.25$ ....	155
29 3D graph of the Frangi Vesselness Measure, variable $\gamma$ , $\beta = 0.5$ .....	156
30 3D graph of the Frangi Vesselness Measure, variable $\gamma$ , $\beta = 0.9$ .....	157
31 3D graph of the Frangi Vesselness Measure, variable $\gamma$ , $\beta = 1$ .....	158
32 3D graph of the Frangi Vesselness Measure, variable $\gamma$ , $\beta = 1.5$ .....	159

## CHAPTER 1

### INTRODUCTION

From [1], it is useful to develop a neonatal test for high risk of Autism Spectrum Disorder. There is some evidence as in [3] that there is some correlation between risk and placental health. Most ASD cases are not diagnosed until the child reaches three or four, so the benefit of any neonatal testing would be very beneficial, as the brain may be more receptive to treatment at a young age. In particular, it was shown in [3] that measurements of the placental chorionic surface vascular network (PCSVN) may be useful in identifying such risk. [1] has provided a method of automatically calculating such features from an extracted vascular network, but does so with manual tracing of the PCSVN in order to make these measurements. These manual tracings are labor-intensive, requiring 4 to 8 hours of labor for each trace. There has been work to automate this procedure [4] [2] [5]. Automating this procedure would not only allow more data acquisition to study the potential effects of placental health on later conditions, but may ideally serve as a real-time diagnostic for neonatal risk factors as well. We continue the work of developing a procedure to automate extraction of the PCSVN.

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

Much work has been done to develop reliable vascular extraction methods for

well-known image domains (such as retinal MRA images) using Hessian-based filters, namely the (multiscale) Frangi filter. It is desirable to extend these technique to study placental images, but this approach is greatly hindered by the comparative irregularity of the placental surface as a whole, which introduces significant noise into the image domain. Prior work [2] solved this problem by provided an additional local curvilinear filter to the Frangi result in an effort to remove some noise from the final extraction.

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

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

## CHAPTER 2

### MATHEMATICAL METHODS

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

#### **Problem Setup in Image Processing**

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

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

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

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

**Definition 2.2** (Image as an interpolated surface).

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

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

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

It is of course necessary to admit that  $I$  is not really a perfect representation of the underlying “content” within the picture. Not only is information lost when  $I$  is stored as an integer, there are also elements of noise and anomalies of lighting that would constitute noise to the original signal. There are multiple treatments of image processing that do address this discrepancy in a pragmatic way [6], especially when the goal is noise reduction. However, we will be content to simply represent the pixels of  $I$  as the ultimate “cause” of the surface  $h$  in definition 2.2, and worry not about how faithfully that sampling corresponds to the real world. Moreover, though our samples in the image domain have been carefully prepared (as outlined in section 3.1), there are numerous shortcomings therein, and improvements to the veracity of our original signal could be made from many angles. Though we shall draw upon the notion of the pixel matrix  $I$  as a sampling again to motivate our development of scale space theory in section 2.4, we ultimately use these techniques because we find them successful to our problem.

## Differential Geometry

We wish to describe the structure of an image as a surface. To do this, we develop the notion of curvature of a surface in  $\mathbb{R}^3$  in a standard way, following [7] (although any undergraduate text in Differential Geometry should prove satisfactory).

### Preliminaries of Differential Geometry

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

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

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

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

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

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

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

**Definition 2.4** (Tangent space of  $U$  at  $u$ ).

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

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

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

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

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

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

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

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

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

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

**Definition 2.7** (Tangent plane of a graph).

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

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

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

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

Quite obviously, we're assuming  $(1,0), (0,1) \in U$ . If this is not the case, we pick

some  $\alpha$  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*

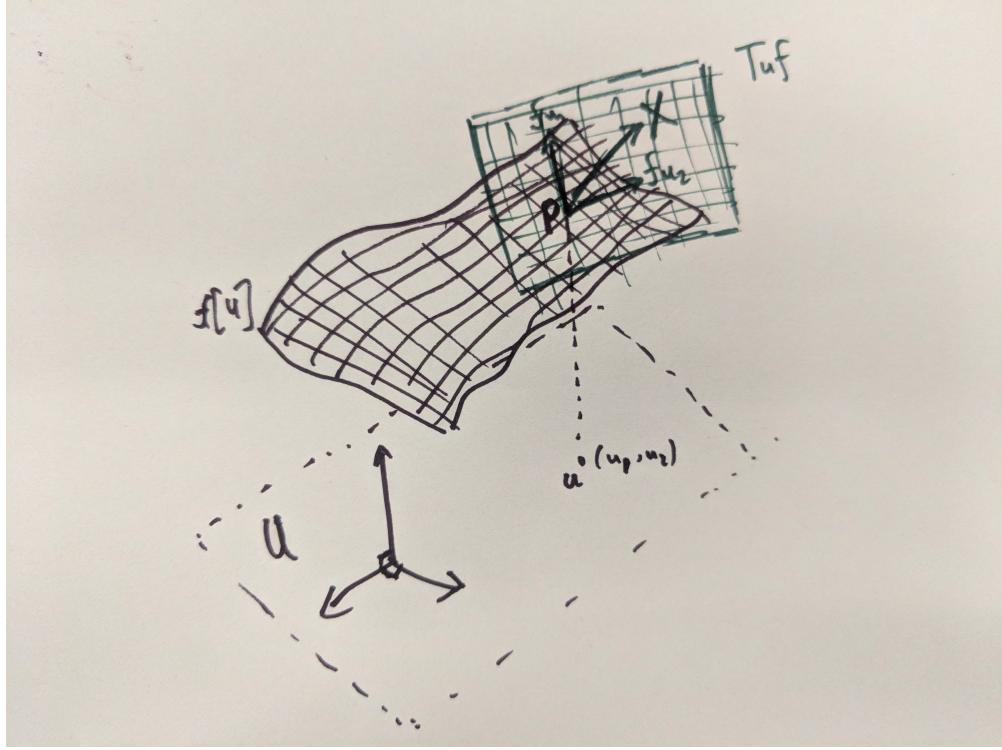


FIGURE 1: Tangent plane of a graph

the graph  $f$  ensures that we can define (for the given curve) an intermediary parametrization  $\theta_c$  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  $\nu$  are indeed orthogonal, as  $X \in \text{span}\left\{\frac{\partial f}{\partial u_i}\right\} = T_u f$ , and  $\nu \perp T_u f$ . We then have (at this fixed point  $u = \theta_c(p)$ )

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**Definition 2.10** (The Weingarten Map).

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

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

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

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

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

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

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

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

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

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

**Definition 2.11.** *The normal curvature of a surface, denoted  $\kappa_v$  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  $L(\frac{\partial f}{\partial u_i}) = -\frac{\partial v}{\partial u_i}$ .

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

The Weingarten map can be formally shown to be well-defined, invariant under coordinate transformation in the general case, which is certainly useful for surfaces  $f$  that are not graphs. We refer to [7] for the general proof. The situation is much less delicate if  $f$  is a graph—the linear transformation may be simply constructed, and we proceed by simply calculating its matrix representation.

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

To find a matrix representation for  $L$ , (which we will denote  $\widehat{L} \in R^{2 \times 2}$ ) we simply wish to find a linear transformation such that  $\widehat{L} \frac{\partial f}{\partial u_i} \Big|_{T_u f} = -\frac{\partial v}{\partial u_i} \Big|_{T_u f}$  for  $i = 1, 2$  where

$- X|_{T_u f}$  denotes that  $X \in T_u f$  is being represented in so-called 'local coordinates' for  $T_u f$ . (Strictly speaking, of course  $T_u f \subset \mathbb{R}^3$  and thus  $\frac{\partial f}{\partial u_i} \in \mathbb{R}^3$ . Thus when we say  $\frac{\partial f}{\partial u_i}|_{T_u f}$  we are referring to this 3-vector expanded with respect to the two-dimensional basis for  $T_u f$ ). In matrix form, we describe this situation as

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

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

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

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

We can simplify this greatly by defining

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

so that

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

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

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

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

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

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

### Principal Curvatures and Principal Directions

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

First, we need the following lemma:

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

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

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

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

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

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

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

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

□

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

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

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

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

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

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

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

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

which implies that  $X$  is an eigenvector of  $\widehat{\mathcal{L}}$  with corresponding eigenvalue  $\lambda$  ( $X \neq 0$  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  $\lambda$  is the Lagrangian multiplier itself.  $\square$

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

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

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

principal directions and curvatures in a simple example.

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

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

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

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

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

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

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

and we have the unit normal vector (Gauss map)

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

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

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

We then calculate each  $h_{ij}$  as

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

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

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

To calculate the second, we use

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

and thus

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

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

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

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

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

We can then rewrite eq. (2.35) as

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

We can see that as  $h_x, h_y$  are close to zero,  $[\delta]$  will be very close to the zero matrix (and the constant  $\frac{1}{\sqrt{1+h_x^2+h_y^2}}$  will be very close to 1 as well), and we should not expect the addition of a "close to 0" matrix to have much effect on the eigenvectors or eigenvalues. This intuition is confirmed by a result from Wilkinson [8], which we state without rigorous proof.

**Theorem 2.7.** *If  $A, B$  are matrices such that  $|A_{ij}| < 1, |B_{ij}| < 1$  (a condition that can be ignored with scaling) and  $\lambda$  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  $\epsilon$ . Thus  $\widehat{L} \approx \text{Hess}(h)$  for any point where the gradient  $\nabla h \approx 0$ . We shall see that we're only concerned with regions where  $h_x, h_y$  is small anyway, and we do not expect

In the event that we do wish to rigorously compute the Weingarten map should want to be rigorously computed "without approximation"—that is, without concern for

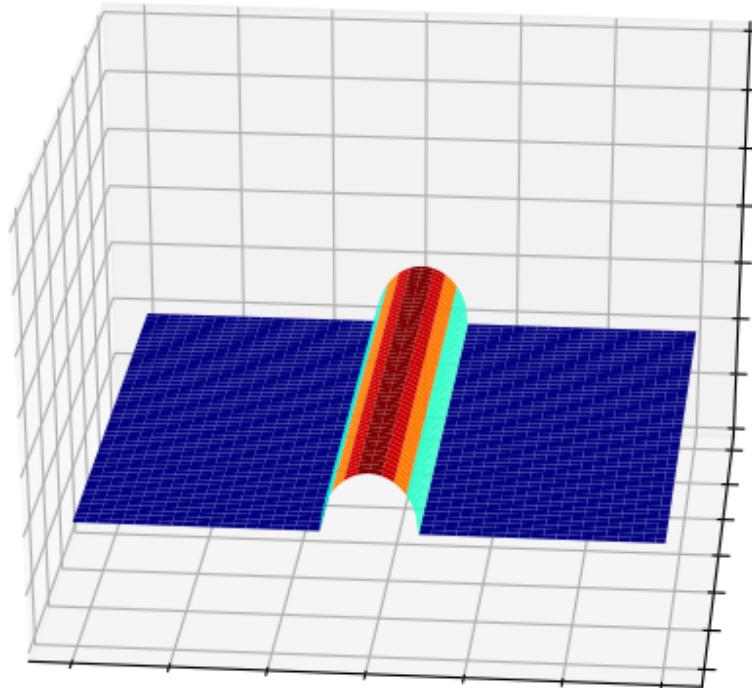
the magnitude of the gradient—we refer to [9] and survey papers mentioned therein.

To make the Weingarten map and its relationship to the Hessian more explicit, we will calculate the Weingarten map for a relatively simple graph.

### The Weingarten map and Principal Curvatures of a Cylindrical Ridge

Let  $f$  be the graph given by

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



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

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

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

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

The gauss map is given by

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

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

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

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

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

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

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

We see that  $u_2 = (0, 1)$  and  $u_1 = (1, 0)$  are eigenvectors for  $\widehat{L}$  with respective

eigenvalues  $\kappa_2 = -\frac{1}{r}$ ,  $\kappa_1 = 0$ . Given the theorem of Olinde Rodriguez suggests that  $u_2$  points in the direction of maximum curvature of the surface,  $-\frac{1}{r}$ , which is predictably in the direction directly perpendicular to the trough, whereas the direction of least curvature is along the trough and is 0. The theorem of Meusnier theorem 2.2 suggests that the normal curvature  $\kappa_2 = -\frac{1}{r}$  is reasonable—any curve on the trough perpendicular to the ridge should have the curvature of a circle (the negative simply indicates that we are on the “outside” of the surface). Finally, we note that at the ridge of the trough is exactly where  $\nabla f = 0$ , and the Weingarten map is exactly the Hessian matrix there.

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

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

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

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

where  $\kappa_i$  and  $u_i$  are known as the *principal curvatures* and *principal directions* of  $L(x, y)$ , respectively, and we label such that  $|\kappa_2| \geq |\kappa_1|$ . Notably,  $\text{Hess}(x, y)$  is a real, symmetric matrix (since  $L_{xy} = L_{yx}$  and  $L$  is a real function) and thus its eigenvalues are real and its eigenvectors are orthonormal to each other, as given by following basic result from linear algebra, [10]:

**Lemma 2.8** (Principal Axis Theorem?). *Let  $A$  be a real, symmetric matrix. The eigenvalues of  $A$  are real and its eigenvectors are orthonormal to each other.*

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

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

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

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

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

---

<sup>1</sup>To simplify notation, we simplify our argument to consider two explicit eigenvectors only, since we're only concerned with the  $2 \times 2$  matrix Hess anyway.

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

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

and is perpendicular to  $v_1$ , since

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

□

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

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

### The Frangi Filter: Uniscale

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

comparison given in [14]. These filters use information about the principal curvatures, approximated as eigenvalues of the Hessian) at each point in the image to identify regions of significant curvature within an image.

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

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

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

where

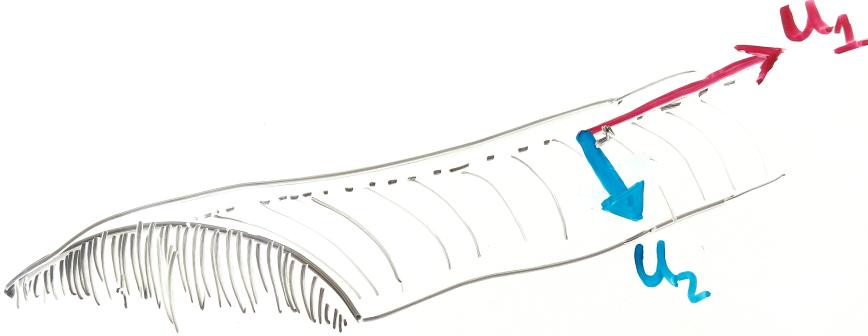
$$A := |\lambda_1/\lambda_2| \quad \text{and} \quad S := \sqrt{\lambda_1^2 + \lambda_2^2} \quad (2.58)$$

and  $\beta$  and  $c$  are tuning parameters. Before we discuss appropriate values for  $\beta$  and  $c$ , we first seek to highlight the significance of eq. (2.57), and in particular, the ratios defined in eq. (2.58).  $A$  and  $S$  are known as the anisotropy measure and structureness measure, respectively. Consequently, we'll refer to the two factors in eq. (2.57) as the anisotropy factor and structureness factor, respectively.

### Anisotropy Measure

The anisotropy (or directionality) measure  $A$  is simply the ratio of magnitudes of  $\lambda_1$  and  $\lambda_2$ . 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  $\lambda_1$  and  $\lambda_2$ . At such a ridgelike structure, we could predict the largest change in curvature to be straight down the ridge



**FIGURE 3:** The principal eigenvectors at a ridge like structure

(in the direction of  $u_2$ ), and the direction of least curvature to be directly along the ridge (in the direction of  $u_1$ ).  $\lambda_1 \approx 0$  and  $\lambda_2$  is large and negative Note that the length of these vectors in this picture is not meant to represent their magnitudes, as  $u_2$  should have a much larger relative magnitude by design!

Of course, if the the ridge is perfectly circular along its cross section (as was in section 2.2.4, it is of course apparent that  $\lambda_2$  would be the same value at any place along the ridge (not just at its crest ), and  $\lambda_1$  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,  $\lambda_2$  as a function of  $x$  would in fact be largest nearest to the bottom. This thought experiment should dispel a naïve misunderstanding of the power of a Frangi filter: a high anisotropy measure (and a large structureness measure ) will not in general identify the crests of a ridge-like structure—it only will highlight that such a pixel is on a ridge-like structure at all. Thus, the anisotropy measure will not necessarily be at a maximum at the crest of the ridge, but instead, somewhere along it.

Similiarly, the vessel we we wish to identify can not be reasonably expected to behave as perfectly as our toy example. There will likely be small aberrations in a ridgelike structure, such as small divots or depressions in an overall ridge-like structure.

Of importance in our data set later ( section 3.1), there will be points where we seem to “lose” our ridgelike structure, but this is simply due to an error in the sample.

Importantly, this formulation does not require  $\lambda_1$  to be approximately zero, just that the curvature in the downward direction is much more significant.

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

We will fix some of these issues by casting this as a multiscale problem in section 2.5.

Two other ideas that could fix some other discrepancies mentioned above is to identify these ridges on their own, or also where the ‘feet are’. We will discuss these ideas in section 6.1.

### **Structureness measure**

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

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

## The Frangi vesselness measure

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

The form Frangi arrived at in eq. (2.57) in which a factor of  $\exp\{\dots\}$  and  $(1 - \exp\{\dots\})$  are multiplied together are simply to ensure that the final vesselness measure  $V$  is largest when  $A$  is small and  $S$  is large enough, with rapid decay in other situations.

Frangi further strengthened the filter by adding an additional case to in eq. (2.57), ensuring that  $\lambda_2$  is not positive. If we are indeed at a curvilinear ridge, we need the second derivative of the surface in the maximal direction to be negative, which hasn't been accounted for as yet in our formulation of  $A$  and  $S$  – we wish (for our purposes) to only identify when we are finding crests.  $A$  will still be small and  $S$  will still be large however if we identify a “trough”.

The only perceivable difference is that the maximum normal curvature will be positive—we are at a local minimum in the direction of  $u_2$ . In situations where we wish to only identify ridges (as is the case here) we simply exclude any points where there is not a negative curvature in the maximal direction. Conversely, we could only seek to find valley, or local minima, as thus require  $\lambda_2 > 0$ , and set the vesselness measure to zero when  $\lambda_2 < 0$ .

## The Frangi vesselness filter: Choosing parameters $\beta$ and $c$

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

Frangi suggested for  $c$  that half of (the Frobenius norm of the) Hessian matrix is

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

$$\begin{aligned}\max\{V_\sigma\} &\leq \max\left(\exp\left\{\frac{-A^2}{2\beta^2}\right\}\right) \max\left(\left(1 - \exp\left\{\frac{-S^2}{2(\gamma S_{\max})^2}\right\}\right)\right) \\ &\leq \max\left\{\left(1 - \exp\left\{\frac{-S^2}{2(\gamma S_{\max})^2}\right\}\right)\right\} \\ &= \left(1 - \exp\left\{\frac{-(S_{\max})^2}{2(\gamma S_{\max})^2}\right\}\right) = \left(1 - \exp\left\{\frac{-1}{2\gamma^2}\right\}\right)\end{aligned}\tag{2.59}$$

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

$$V_\sigma(x_0, y_0) = \begin{cases} 0 & \text{if } \lambda_2 > 0 \\ a_\gamma \exp\left\{\frac{-A^2}{2\beta^2}\right\} \left(1 - \exp\left\{\frac{-S^2}{2(\gamma S_{\max})^2}\right\}\right) & \text{otherwise} \end{cases}\tag{2.60}$$

where, as before,

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

and

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

For  $\beta$  Frangi chose an innocuous intermediate point,  $\beta = 1/2$  (and thus  $2\beta^2 = 1/2$ ).

As we will show later, choosing the structureness parameter  $\gamma$  is rather important for the context especially if the background (non-ridgelike structure) is significant and noisy.  $\beta$  should be strengthened/relaxed depending on how “flat” the ridgelike structure is. If there is a lot of gain then  $\beta$  should be smaller. If this is not the case, a stronger filter can be created by requiring  $A$  to be much smaller.

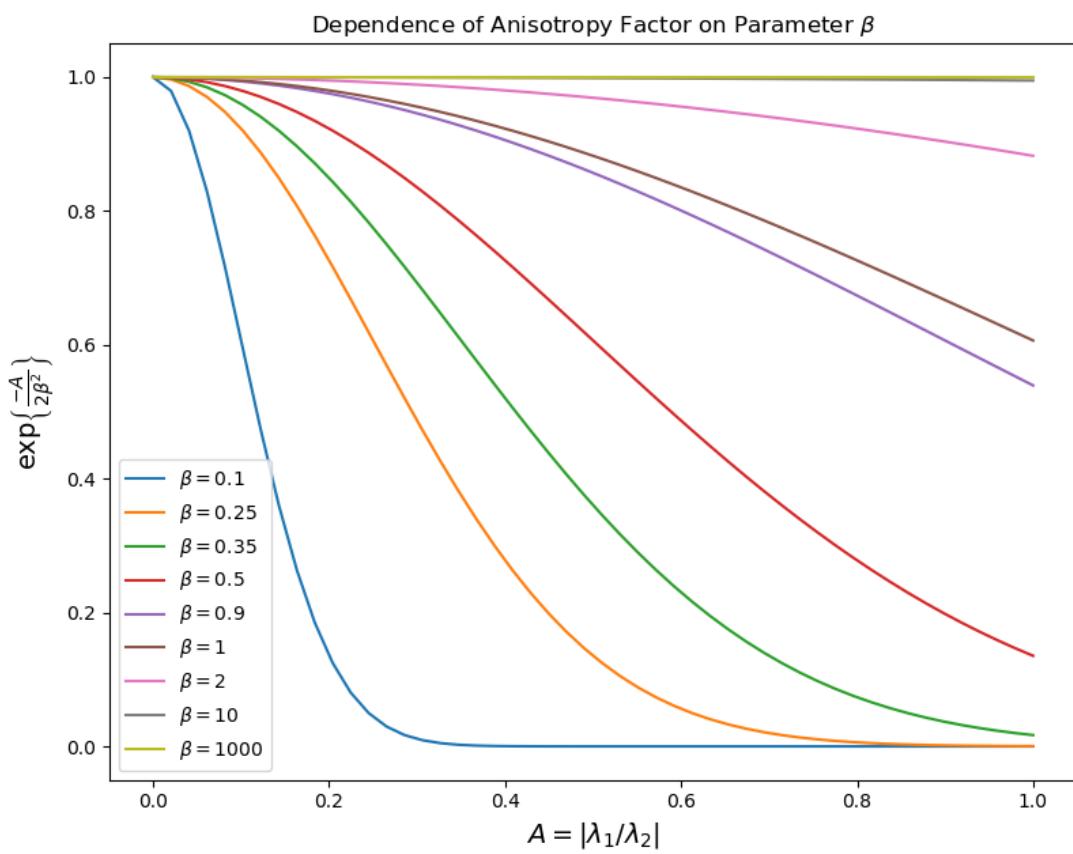
Considering as the anisotropy measure  $(\lambda_1/\lambda_2) \in [0, 1]$  (simply since  $|\lambda_1| \geq |\lambda_2|$ ), we can actually visualize how much the anisotropy factor varies depending on our choice of  $\beta$ , as seen in fig. 4.

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

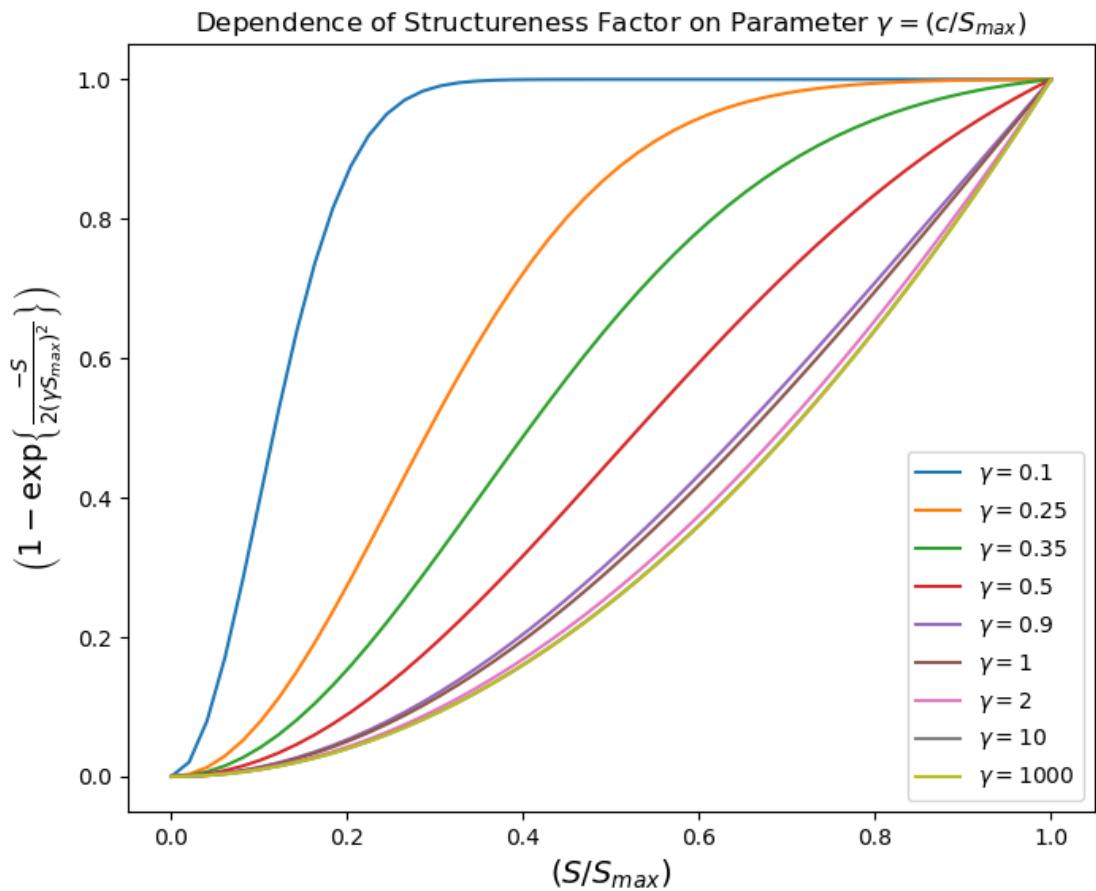
We now take a quick tangent from our description of the Frangi filter to develop and justify our “multiscale” approach.

### Linear Scale Space Theory

There is obviously a major disconnect in the ideas presented above. Although the ideas presented above require differentiation of continuous surfaces, our image is in fact a discrete pixel. That is, our previous discussions have been in terms of an image as the continuous surface in definition 2.2, rather than the more realistic discrete pixel matrix as in definition 2.1. The present section seeks to address this disconnect. In particular, we seek to mitigate the bias of our limited sampling of the “true” 3D surface. Our main goal is to counter against some of the bias of our particular sampling. In particular, we wish to not over-represent structures that are clear at our resolution without giving appropriate weight to larger structures as well. Koenderink [15] argued that “any image



**FIGURE 4:** Dependence of the Anisotropy Factor on its Parameter



**FIGURE 5:** Dependence of the Structureness Factor on its Parameter

can be embedded in a one-parameter family of derived images (with resolution as the parameter) in essentially only one unique way" given a few of the so-called scale space axioms. He (and others) showed that a small set of intuitive axioms imply require that any such family of images must satisfy the heat equation

$$\Delta K(x, y, \sigma) = K_\sigma(x, y, \sigma) \text{ for } \sigma \geq 0 \text{ such that } K(x, y, 0) = u_0(x, y). \quad (2.61)$$

where  $K : \mathbb{R}^3 \rightarrow \mathbb{R}$  and  $u_0 : \mathbb{R}^2 \rightarrow \mathbb{R}$  is the original image (viewed as a continuous surface) and  $\sigma$  is a resolution parameter. Much work has been done to formalize this approach [16]. There is a long list of desired properties—we will try to identify a minimal subset of axioms and show that other desired properties follow.

### Axioms

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

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

The following axioms are then requirements on what sort of operation  $T_\sigma$  should be.

**Axiom 2.1** (Linear-shift and Rotational Invariance). *Linear-shift (or translation) invariance means that no position in the original signal is favored. This is intuitive, as our operation should apply to any image fairly, regardless of where content is found in the image. Similarly, there should be not be favoritism toward any particular orientation of content within the image.*

**Axiom 2.2** (Continuity of Scale Parameter). *There is no reason for the scale parameter to be discrete; we may alter the resolution with whatever precision we desire. That is, we take the resolution parameter  $\sigma$  to be a nonzero real number (as opposed to an integer). Moreover, we require that the operator behaves continuously with respect to the scale parameter.*

What happens as  $\sigma \downarrow 0$  is not immediately clear though. An argument from functional analysis (see [17]) implies that there is a so-called “infinitesimal generator”  $A$  which is a limit case of our desired operator  $T$ ; that is

$$Au_0 = \lim_{\sigma \downarrow 0} \frac{T_\sigma u_0 - u_0}{\sigma} \quad (2.62)$$

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

$$\partial_\sigma K(x, y; \sigma) = \lim_{\sigma \downarrow 0} \frac{K(\cdot; \sigma + h) - K(\cdot; \sigma)}{h} = A(T_\sigma u) = A(K(\cdot, \sigma)) \quad (2.63)$$

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

**Axiom 2.3** (Semigroup property). *The semigroup property is simply that transforming the original image by some resolution  $\sigma$  should have the same overall effect of two successive transformations  $\sigma_1$  and  $\sigma_2$ , i.e.*

$$T_\sigma u = T_{\sigma_1 + \sigma_2} u \quad (2.64)$$

**Axiom 2.4** (Causality Condition). *The following requirement has great implication, and is also very successful in encoding our intuitive sense of “resolution”. The causality condition is the one that, as resolution decreases, no finer detail is introduced into the image. That is, as the scale increases, there will be no creation of local extrema that did not exist at a smaller scale.*

In other words, if  $K(x_0, y_0; \sigma_0)$  is a local maximum (at the point  $(x_0, y_0)$ , at this fixed

$\sigma_0$ ) i.e. then an increase in scale can only weaken this peak, i.e.

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

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

$$\begin{cases} \nabla K(x_0, y_0; \sigma_0) = 0 \\ \Delta K(x_0, y_0; \sigma_0) > 0 \end{cases} \implies K(x_0, y_0; \sigma_1) \geq K(x_0, y_0; \sigma_0) \forall \sigma_1 \geq \sigma_0 \quad (2.66)$$

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

### Uniqueness of the Gaussian Kernel

The above requirements are actually sufficient in proving not only that the operator  $T_\sigma$  is a convolution, but that the heat equation described in eq. (2.61) must hold. This has been shown in various ways, both by Koenderink [15], Babaud [18], as well as Lindeberg in [16]. In fact, it is shown that the Gaussian is the unique convolution kernel that works.

To this, show that:

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

That is,

$$K(x, y; \sigma) = T_\sigma u_0 = G_\sigma \star u_0 \quad \text{where} \quad G_\sigma := \frac{1}{2\pi\sigma^2} e^{(-|x|^2/(2\sigma^2))} \quad (2.67)$$

We can show that this solution solves the heat equation. Given  $u_0$  as a continuous image (unscaled), we construct PDE with this as a boundary condition.

$$u : \mathbb{R}^2 \supset \Omega \rightarrow \mathbb{R} \text{ with } u(\mathbf{x}, t) : \begin{cases} \frac{\partial u}{\partial t}(\mathbf{x}, t) = \Delta u(\mathbf{x}, t) & , t \geq 0 \\ u(\mathbf{x}, 0) = u_0(\mathbf{x}) \end{cases} \quad (2.68)$$

We show that

$$u(\mathbf{x}, t) = (G_{\sqrt{2t}} \star u_0)(\mathbf{x}) \quad (2.69)$$

solves (the above tagged equation), where

$s$

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

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

*Proof.* For a single variable,

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

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

$$= \int f(t) \left( \frac{\partial g}{\partial \alpha} \right) g(\alpha - t) dt \quad (2.72)$$

$$= f(\alpha) \star g'(\alpha) \quad (2.73)$$

By symmetry of convolution we can also conclude

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

If  $f$  and  $g$  are twice differentiable, we can compound this result to show a similar

statement holds for second derivatives, and then, given the additivity of convolution, we may conclude

$$\Delta(f \star g) = \Delta(f) \star g = f \star \Delta(g) \quad (2.74)$$

□

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

*Proof.* We focus on the particular kernel

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

Then

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

$$= \frac{\partial}{\partial t} (G_{\sqrt{2t}}(\mathbf{x}, t)) \star u_0(\mathbf{x}) \quad (2.76)$$

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

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

$$= -\frac{1}{4t^2} \left( e^{(-|\mathbf{x}|^2/(4t))} + |\mathbf{x}|^2 G_{\sqrt{2t}}(\mathbf{x}, t) \right) \star u_0(\mathbf{x}) \quad (2.79)$$

and from the previous lemma,

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

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

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

By symmetry of argument we also may conclude

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

and so

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

Then, given lemma 2.9, we conclude

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

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

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

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

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

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

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

$$= \frac{\partial}{\partial t} [G_{\sqrt{2t}}(x, y)] \star u_0(x, y) \quad (2.85)$$

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

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

$$= \left( G_{\sqrt{2t}}(x, y) \left[ \frac{x^2 + y^2}{4t^2} - \frac{1}{t} \right] \right) \star u_0(x, y) \quad (2.88)$$

Combining these results, we find that

$$\frac{\partial}{\partial t} [G_{\sqrt{2t}} \star u_0] = \Delta [G_{\sqrt{2t}} \star u_0] \quad (2.89)$$

as desired.  $\square$

## Scale Spaces over Discrete Structures

The above developments from scale space axioms have (since their first appearance) been recast in terms of discrete structures (rather than continuous surfaces) as in [19]. However, we've chosen to present the above in their original continuous surface for clarity of argument. The discrete case is not much different– we still have the same axioms, and it can be shown that the family of scaled images must simply satisfy a discrete version of the However, viewing our actual image definition 2.1 as a sample of a continuous surface definition 2.2, we might naively expect our convolution by the Gaussian to “commute” with our supposed sampling of the continuous signal, or even that we could simply convolve our discrete signal with a discretely sampled Gaussian

kernel. The latter in fact, seems to be an often implemented interpretation of scale space theory.

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

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

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

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

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

The reality of the matter is that a discretely sampled Gaussian is not an appropriate kernel for creating discrete scale space. In [19] and in particular [20], Lindeberg demonstrated that the sampled Gaussian kernel violates not only semigroup property (axiom 2.3), but—much less forgivably—the causality property (axiom 2.3). There is absolutely no guarantee that convolution with a sampled Gaussian kernel will not create “spurious” structures as resolution increases.

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

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

$$T(n; \sigma) = e^{-\alpha\sigma} I_n(\alpha\sigma), \quad I_n(\sigma) = I_{-n}(\sigma) = (-1)^n J_n(i\sigma) \quad n \geq 0, \sigma, \alpha > 0 \quad (2.90)$$

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  $\alpha$  above is simply an optional scaling parameter which is simply set to 1 hereforth.

The derived family of 1D signals is then given by

$$K(x, \sigma) = \sum_{n=-\infty}^{\infty} T(n; t) f(x-n) \quad \text{for } x \in \mathbb{Z}, t > 0 \quad (2.91)$$

The compatibility of scale space theory and derivatives on discrete structures and extension to two dimensions was also demonstrated by Lindeberg in [22] and [23]. In particular, we may take derivatives of the convolutions of our discrete images using, say, a central difference. Lastly, the 2D version of the family given in eq. (2.91) can be obtained by independent convolution of its dimensions (i.e. it is separable). We will make these ideas explicit in chapter 4 and the Appendix.

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

### **The Frangi Filter: A multiscale approach**

Our ideas of scale developed in the previous section imply that, if the ridgelike structures we wish to detect are more prominent at different scales, then a multiscale approach is the natural one. Considering our developments in section 2.3, we wish to probe at multiple scales regions that would receive a high vesselness score at any range, and consider them all together. Frangi [11] approached this problem by simply aggregating vesselness measure over all scales:

$$V(x_0, y_0) = \max_{\sigma \in \Sigma} V_\sigma(x_0, y_0) \quad (2.92)$$

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

### Thresholding

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

$$V_\Sigma := [V(x, y)]_{\substack{0 \leq x < M \\ 0 \leq y < N}} \quad (2.93)$$

Notably, Frangi [11] refrained from explicitly interpreting the probability assigned by eq. (2.92); that is—whether a particular point  $(x, y)$  in the image definitely a vessel or not. Instead, he cautioned that the result should not be used as a segmentation method alone, and that the size of the vasculature cannot be determined rigorously from the filter alone.

However, for the purposes of obtaining an intermediate result, we wish to be final about the whole matter and ultimately say whether or not a pixel does in fact corresponds to a curvilinear structure. A straightforward enough approach is to simply threshold at some fixed value. The resulting matrix can be given in terms of either eq. (2.92) or eq. (2.93)

$$V_{\Sigma, \alpha}(x, y) = \begin{cases} 1 & \text{if } V(x, y) \geq \alpha \\ 0 & \text{else} \end{cases}, \quad \alpha > 0 \text{ for } \alpha \text{ fixed.} \quad (2.94)$$

We will discuss alternatives methods of aggregating results from our multiscale

method, as well as optimal values for parameters and scales in chapter 4. As a final note, we admit that any future extensions of this work (as will be discussed in chapter 6) should not hold too much stock in this thresholded result, and analyzing the raw vesselness score eq. (2.93), or even the un-merged scale-wise scores, would be far more rewarding.

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

### Calculating the 2D Hessian

According to section 2.4.3, we may calculate derivatives of our structure by calculating a gradient on our convolved image. Our method of calculating the gradient of a matrix uses a second-order accurate central difference, as in [24]. Specific implementation will be discussed in chapter 4.

We note in passing that we may take the derivative of the Gaussian kernel and then convolve it, and the effect will be the same as if we had taken the derivative subsequently [6]. This could offer some computational speedup if we wish to run this procedure on many samples and fixed scale sizes, although we have implemented our scale spaces in the conventional way, as discussed in chapter 4.

### Convolution Speedup via FFT

In practice, the convolutions described above are very slow for large scales ( $\sigma$ ), as the size of the kernel is very large. Instead, we will perform a fast Fourier transform, which requires only  $\mathcal{O}(N \cdot \log_2 N)$  operations for a one dimension signal of length  $N$ , as compared to the  $N^2$  operations required of a conventional discrete Fourier transform [6]. We will briefly outline the theory of Fourier transforms.

#### Fourier Transform of a continuous 1D signal .

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

$$f(t) = \sum_{n=-\infty}^{\infty} c_n e^{j\frac{2\pi n}{T}t}, \quad c_n = \frac{1}{T} \int_{-T/2}^{T/2} f(t) e^{-i\frac{2\pi n}{T}t} dt \quad (2.95)$$

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

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

An inverse transform will then recover our original signal:

$$f(t) = \mathcal{F}^{-1}\{F(\mu)\} = \int_{-\infty}^{\infty} F(\mu) e^{j2\pi\mu t} dt \quad (2.97)$$

Together, eq. (2.96) and eq. (2.97) 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  $\Delta T$ :

$$s_{\Delta T}(t) := \sum_{n=-\infty}^{\infty} \delta[t - n\Delta T], \quad \delta[t] = \begin{cases} 1, & t = 0 \\ 0, & t \neq 0 \end{cases} \quad (2.98)$$

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

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

$$\tilde{f}(t) = f(t)s_{\Delta T}(t) \quad (2.99)$$

From this we can calculate  $\tilde{F}(t)$ . Given the discrete signal  $\tilde{f}$ , we construct the

transform  $\tilde{F}(\mu) = \mathcal{F}\{\tilde{f}(t)\}$ . by expanding  $\tilde{f}$  in the same infinite basis as the continuous case.

$$\tilde{F}(\mu) = \sum_{n=-\infty}^{\infty} f_n e^{-i2\pi\mu n \Delta T}, \quad f_n = \tilde{f}(n) = f(n\Delta T) \quad (2.100)$$

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

## 2D DFT Convolution Theorem .

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

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

$$H(u, v) := \mathcal{D}\{h(x, y)\} = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} h(x, y) e^{-2\pi i (\frac{ux}{M} + \frac{vy}{N})} \quad (2.102)$$

and given the convolution of the two functions

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

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

$$(f \star h)(x, y) = \mathcal{D}^{-1}\{MN \cdot F(u, v)H(u, v)\} \quad (2.104)$$

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

*Proof.*

$$(f \star h)(x, y) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m, n)h(x-m, y-n) \quad (2.105)$$

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

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

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

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

$$= MN \cdot \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u, v) H(u, v) e^{2\pi i (\frac{ux}{M} + \frac{vy}{N})} \quad (2.110)$$

$$= MN \cdot \mathcal{D}^{-1} \{ FH \} \quad (2.111)$$

where

$$\hat{\delta}_N(k) = \begin{cases} 1 & \text{when } k = 0 \pmod{N} \\ 0 & \text{else} \end{cases} \quad (2.112)$$

□

Above, we make use of the following lemma

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

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

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

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

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

$$z^N - 1 = (z - 1)(z^{n-1} + \dots + z + 1) = (z - 1) \sum_{n=0}^{N-1} z^n. \quad (2.114)$$

thus giving us

$$0 = \left(e^{2\pi i(j-k)/N} - 1\right) \sum_{n=0}^{N-1} e^{2\pi i n(j-k)/N} \quad (2.115)$$

To prove the claim in eq. (2.113), we consider two cases: First, if  $j - k$  is a multiple of  $N$ , we of course have  $e^{2\pi i n(j-k)/N} = \left(e^{2\pi i}\right)^{n(j-k)/N} = 1$  and thus the left side of eq. (2.113) reduces to

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

In the case that  $j - k$  is *not* a multiple of  $N$ , we refer to eq. (2.115). The first factor is not zero since,  $\left(e^{2\pi i(j-k)/N}\right) \neq 1$  (simply since  $(j - k)/N$  is not an integer), and thus it must be that the second factor is 0:

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

We can combine these two cases by invoking the definition of eq. (2.112), giving us the result.  $\square$

## FFT

As noted, the above result applies to the Discrete Fourier Transform. We actually achieve a convolution speedup using a Fast Fourier Transform (FFT) instead. We follow the developments of [6]. For clarity, we present the following theorems which allow a

framework to calculate a 2D Fourier transforms quickly.

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

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

$$= \sum_{x=0}^{M-1} e^{-i2\pi\mu x/M} \left[ \sum_{y=0}^{N-1} f(x, y) e^{-i2\pi\nu y/N} \right] \quad (2.117)$$

$$= \sum_{x=0}^{M-1} e^{-i2\pi\mu x/M} \mathcal{F}_x\{f(x, y)\} \quad (2.118)$$

$$= \mathcal{F}_y\{\mathcal{F}_x\{f(x, y)\}\} \quad (2.119)$$

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

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

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

To "simplify" our notation we will use a new notation for the Fourier

kernels/basis functions. Let the 1D Fourier transform be given by

$$F(u) = \sum_{x=0}^{M-1} f(x) W_M^{ux}, \quad \text{where} \quad W_m := e^{-i2\pi/m} \quad (2.120)$$

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

We use this to rewrite the series in eq. (2.120) and split it into odd and even entries in the summation

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

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

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

$$W_{(2m)}^{(2n)} = e^{\frac{-i2\pi(2m)}{2m}} = e^{\frac{-i2\pi m}{n}} = W_m^n \quad (2.123)$$

$$W_m^{(u+m)x} = e^{\frac{-i2\pi(u+m)x}{m}} = e^{\frac{-i2\pi unx}{m}} e^{\frac{-i2\pi mx}{m}} = e^{\frac{-i2\pi ux}{m}} (1) = W_m^{ux} \quad (2.124)$$

$$W_{2m}^{(u+m)} = e^{\frac{-i2\pi(u+m)}{2m}} = e^{\frac{-i2\pi ux}{2m}} e^{-i\pi} = W_{2m}^u e^{-i\pi} = -W_{2m}^u \quad (2.125)$$

Thus we can rewrite eq. (2.122) as

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

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

The major advance comes via using the identities eq. (2.123) to consider the

Fourier transform  $K$  frequencies later :

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

$$\implies F(u+K) = \left( \sum_{x=0}^{K-1} f(2x) W_K^{ux} \right) - \left( \sum_{x=0}^{K-1} f(2x+1) W_K^{ux} \right) W_K^u \quad (2.129)$$

Comparing eq. (2.127) and eq. (2.129), we see that the expressions within parentheses are identical. What's more, these parenetical expressions are functionally identical to discrete fourier transforms themselves. Let's notate them as follows:

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

$$\mathcal{D}_u\{f_{\text{odd}}(t)\} := \sum_{x=0}^{K-1} f(2x+1) W_K^{ux} \quad (2.131)$$

If we're calculating an  $M$  point transform (i.e. we're wishing to calculate  $F(1), \dots, F(M)$ ), once we've calculated the first  $K$  discrete frequencies (i.e.  $F(1), \dots, F(K)$ ) we may simply reuse the two values we've calculated in eq. (2.130) to calculate the next  $F(K+1), \dots, F(K+K) = F(M)$ . Since each expression in parentheses involves  $K$  complex multiplications and  $K-1$  complex additions, we are effectively saving  $K(2K-1)$  calculations in computing the entire spectrum  $F(1), \dots, F(M)$ . When  $M$  is large, the payoff is undeniable.

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

Of course, since eq. (2.130) are DFTs themselves, there's nothing stopping us from reiterating this procedure; if  $M$  is substantially large, we can just as easily repeat this process a few times.

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

of eq. (2.116).

The one caveat is that the above development was for transforming sequences whose lengths are perfect powers of 2. Since our inputs have no reason to be this, we need to adjust for this. The explanation is that you just do the part that's a power of 2 and then do the rest manually or pick a different power.

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

## CHAPTER 3

### RESEARCH PROTOCOL

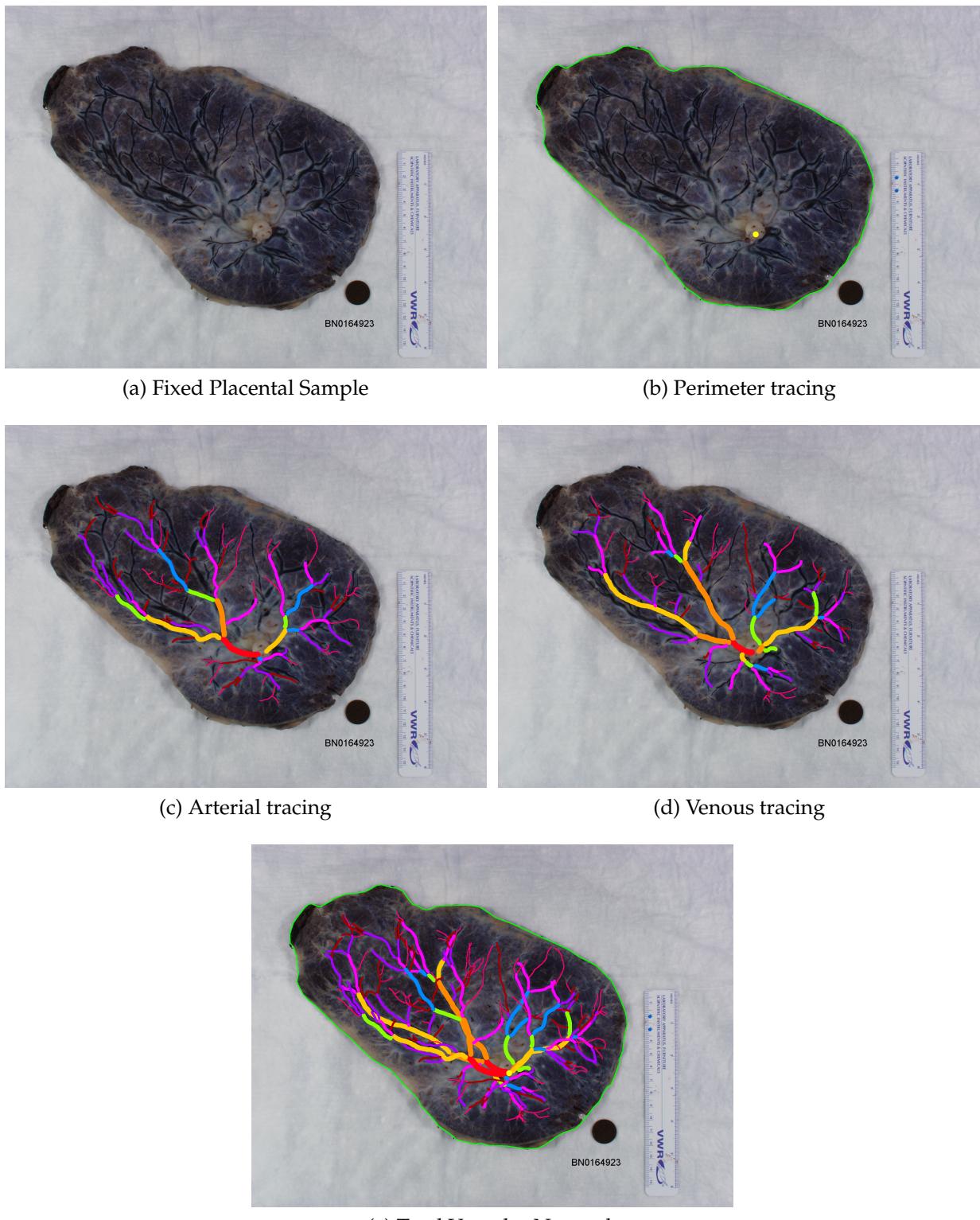
#### Samples / Image Domain

We ultimately perform a multiscale Frangi filter to estimate the PCSVN from a subset of 201 color placental images from a private database provided by the National Children’s Study, which had been prepared for a different study. A detailed description of the data set is given in [1], and a description of the cleaning and fixing procedure is given in [4]. The samples are provided as XCF files (the native project file for GIMP) and contain four major layers.

#### A representative sample

The layers together give a hand tracing of the vascular network and perimeter. A sample of overlaid layers in a representative sample (with ID number “BN0164923”) is given in fig. 6.

Each layer is roughly 1954x1200 pixels (with some occasional variation). In fig. 6a, a cleaned, fixed placenta is placed on a table with a camera a fixed distance away, and a ruler and penny (presumably for redundancy) to aid registration and calibration of the resolution. fig. 6b is a tracing (in green) of the perimeter of the placenta. The point of umbilical cord insertion is notated in yellow. Two cyan marks are placed on consecutive centimeter markings on the ruler (the dots are enlarged and shown as a darker blue here for clarity). fig. 6c and fig. 6d are both hand traces of the PCSVN, with a layer for each the arteries and veins. These layers are simultaneously overlain on the base image in fig. 6e. The coloration is meant to indicate the diameter of each vessel. The diameters are binned into 9 discrete widths, odd integers from 3 to 19 pixels. Vessels of smaller



**FIGURE 6:** A representative placental sample and tracing

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

TABLE 1: Vessel width color code

diameter are either binned to three or (quite frequently) left untraced. The correspondence between pencil color and (binned) vessel width is given in table 1.

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

### Knowns and Unknowns

Of course, we wish to simply operate on the placental sample itself, without any understanding of its provided tracing (except for judging the strength of our algorithm); our goal is to develop an algorithm that can produce a “ground truth” tracing similar to fig. 6e or fig. 7d without any user intervention.

For our purposes however, we will use a limited amount of information from the tracings, namely the provided placental perimeter (shown in green in fig. 6). In

developing a fully automated algorithm, it would be relatively straightforward to obtain this boundary ourselves using various techniques, such as an Active Contour Model [25] or, or even a simple edge finding algorithm followed by watershedding and largest object selection as in [2]. We leave that for future work. We do use the traced placental perimeter at our own peril, however, since often there are tears in the side of the plate or large amounts of non-vascular content with large changes in height that are not adequately accounted for in the perimeter tracing.

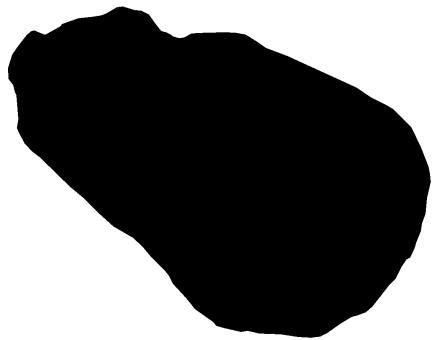
Finally, we will consider the location of the umbilical insertion point as a “known”, as the vessels around it are frequently impossible to see and we wish to exclude them from consideration. It is not unreasonable, however, to consider this to be a known—in future preparations of samples, we could simply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

### Data Cleaning and Preprocessing

Building a sample suitable for use in our algorithm from fig. 6 is relatively simple. We zero outside the boundary of the plate (so as to not waste computational time calculating the differential geometry of a ruler, say), and also generate a binary mask to identify the plate. Finally, our vessel layers are combined and given as a binary trace. Our preprocessed samples used by the algorithm are given in section 3.2.

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

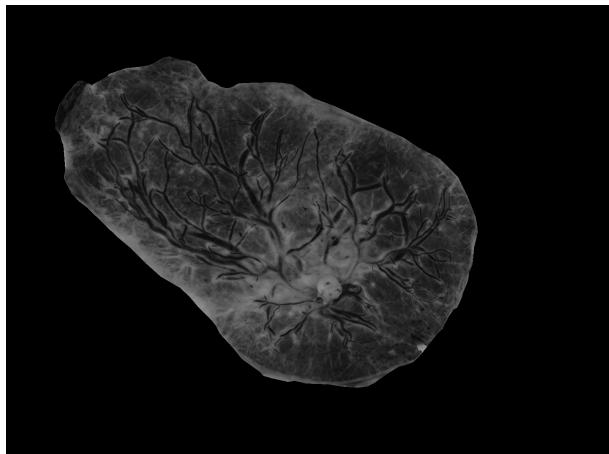
As a point of technicality, the grayscale image in fig. 7c is not actually produced directly by the extractor plug-in, but created when the 3 channel RGB image fig. 7b is



(a) Background Mask (in white)



(b) Sample with BG removed



(c) Grayscale



(d) Trace / “Ground Truth”

**FIGURE 7:** Preprocessed files from an NCS sample

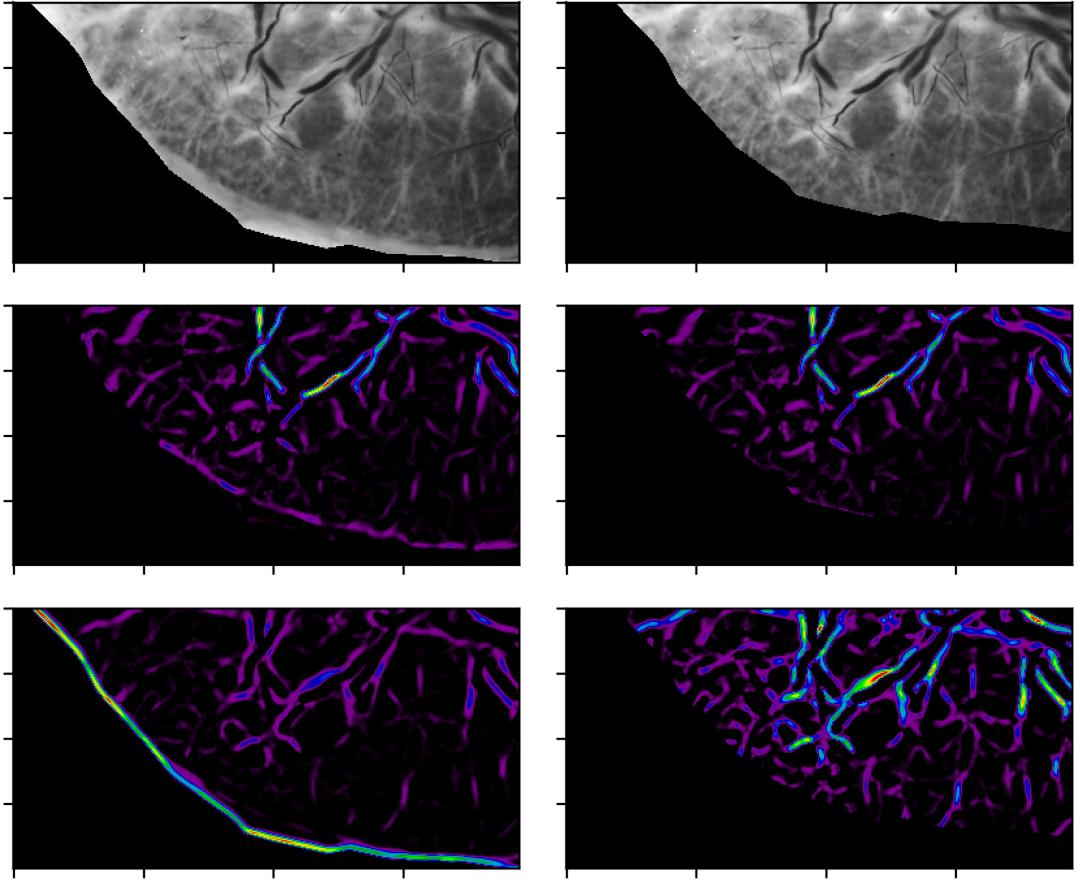
imported at the start of the algorithm. This grayscale conversion is simply done for ease of analysis on the sample: although the Frangi filter is designed for arbitrary N-dimensional input [11], an image with three color channels does not have 3 spatial dimensions. We therefore simply combine the information in three channels using the well-known and oft-implemented ITU-R 601-2 luma [26], or “luminance” transform:

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

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

### Boundary Dilation

All images are grayscale,  $M, N$  pixels as a masked array (of type `numpy.ma.MaskedArray`), where pixels outside of the placental region are masked so they will not be considered by the algorithm. However, some standard implementations of algorithms, namely `numpy.gradient` and `scipy.signal.convolve2d` are not designed to handle masked regions. Although it would be potentially useful to adapt such methods in a way to, say, calculate a gradient or performs a convolution by a “reflection” across an arbitrary closed boundary (as opposed to the edge of the image matrix), we opted instead to “zero out” unwanted background pixels and simply exclude adjacent areas from consideration. This excluding function, `plate_morphology.dilate_plate`, ultimately relies on two functions provided by the Python library `scikit-image` [27]. The first, `skimage.segmentation.find_boundaries()`, takes the mask input (such as fig. 7a) and calculates where differences in a morphological erosion and dilation occur (which should have the same effect as using the perimeter labeled in fig. 6b directly, though we’ve chosen to not include that in our sample). That boundary itself is then dilated by the desired factor. The second is a much quicker implementation of binary dilation that is particularly efficient for our problem: we iterate through an array of



**FIGURE 8:** Effect of boundary dilation on Frangi responses

indices for the image where the boundary occurs and simply extend the mask  $R$  pixels in each direction (like a giant plus scale). Since these pixels are all connected, the effect is very similar to convolving with a disk of radius  $R$ , but is much faster.

fig. 8 shows the effect of this so-called “boundary dilation.” In the image above,  $\sigma = 3$  and border radius is 25 to exaggerate the effect. The first row shows the unaltered boundary of the sample (left) and the sample after boundary dilation (with radius dilation of 25 pixels). The second row shows the Frangi vesselness measure at single scale ( $\sigma = 3$ ) where `DARK_BG=False` to target dark curvilinear structures performed on the altered sample (left) and the boundary dilated sample (right). Removing an unnecessary

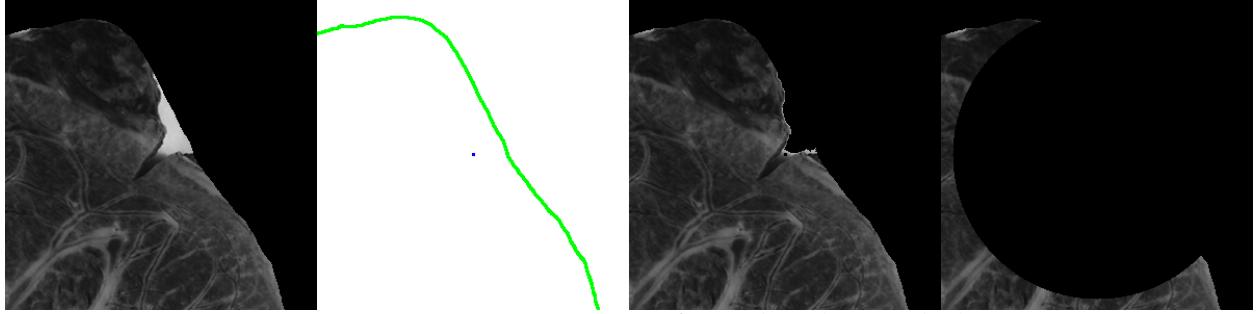
part of the placental plate prevents a small response to a non-vascular yet mildly curvilinear background feature from appearing. The third row of fig. 8 shows the Frangi vesselness measure at the same scale ( $\sigma = 3$ ) when we are probing for bright curvilinear structures (i.e. `DARK_BG=True`). Here, wherever the very edge of the placental plate is \*any\* brighter than adjacent interior, a very large Frangi response will occur, as seen on the left. Dilating the boundary completely avoids this issue, as seen by the figure on the right. Thus we prevent a visual artifact that is present in much prior work on this problem (see [2], [4]). It should be noted that, while the figure on the right shows a much larger interior response, this is simply because the intensity of the output in each of these images is being independently scaled between the minimum and maximum intensity in the image. However, we argue that this is an appropriate and desired depiction of the situation, as we will frequently consider only the relative maxima of Frangi response per scale in our analysis.

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

The code for generating fig. 8 is found in the within the “`if __name__ == __main__`” block of the file `plate_morphology.py`, (so the figure will be generated when running `plate_morphology.py` as a top-level script from the command line). See appendix.

## Cut Removal

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

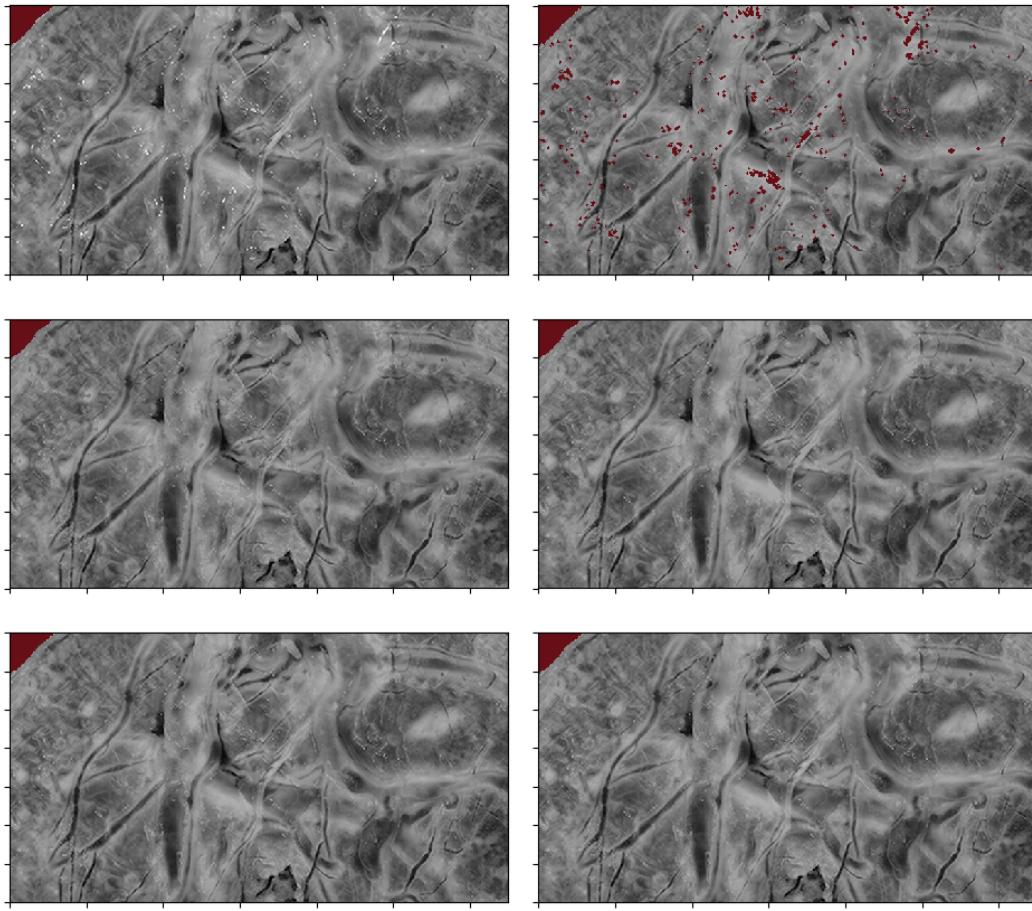


**FIGURE 9:** Removing a “cut” from the placental plate.

based on the background value of the blue dot; if this is incorrectly placed (or anything else weird happens), we can manually opt to simply remove a large area from consideration from the plate, as we do the umbilical cord insertion point. This is demonstrated in fig. 9. From left to right, we have the original masked placental sample, the cut mark noted on the perimeter layer, the improved mask after watershedding, and finally, the “scorched earth” approach in case the previous attempt failed. In few additional cases, neither approach is adequate. We again will stress that a fully automated algorithm would have no knowledge of our traced perimeter (as in fig. 6b or the second from left image in fig. 9), so we anticipate a fully automated method of handling that problem should also be able to correct for these “cuts” as well.

### Deglaring

Despite best efforts when harvesting samples, some placental images have substantial glare, which leads to inaccuracies in identifying curvilinear content. Our protocol for deglaring is analogous to that performed in [4] and [2]. Unfortunately, the method relied upon by those previous papers (MATLAB’s `imfill`, which relies on inpainting by solving the Dirichlet problem for masked regions) was not immediately available in a Python environment. Instead, we used an already implemented inpainting algorithm, `scikit-image`’s `inpaint_biharmonic()`, which should be expected to achieve similar results, at the expense of processing time.

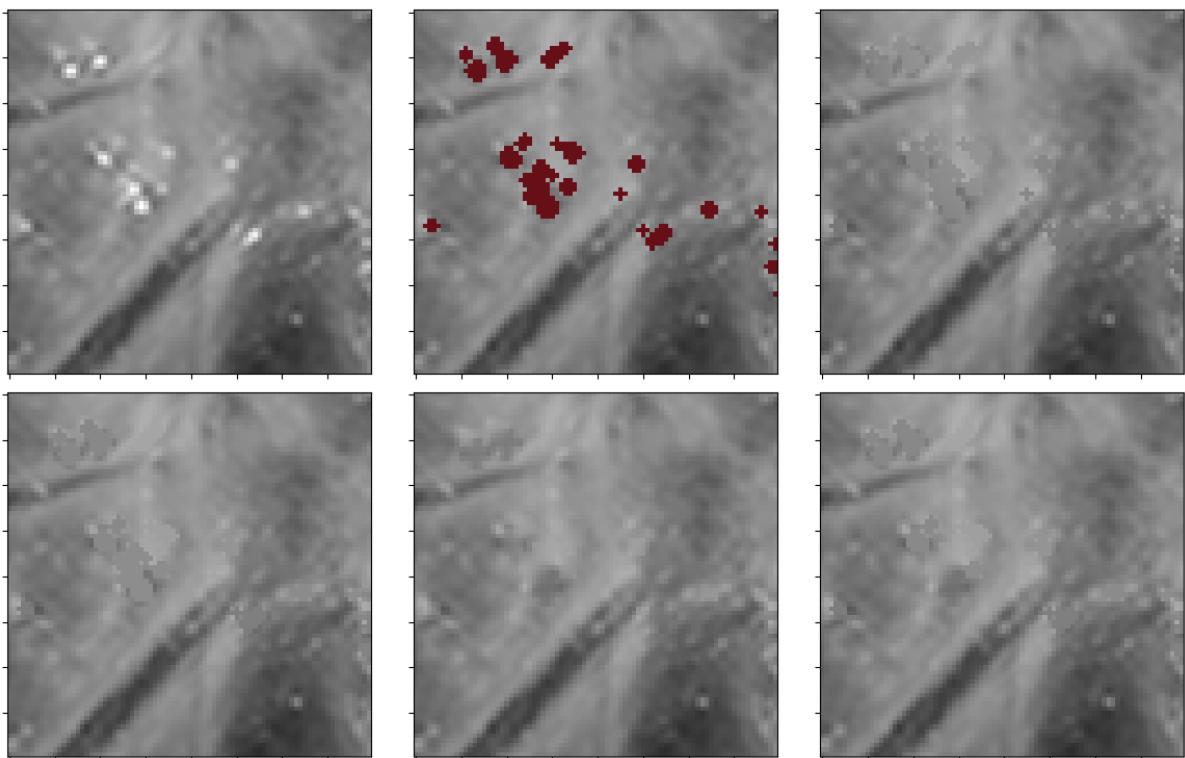


**FIGURE 10:** Deglaring a sample using a hybrid inpainting method

The function `inpaint_biharmonic` is based on [28], and relies on solving a biharmonic equation i.e.  $\nabla\nabla f = 0$  for the surface  $f$  subject to boundary conditions (as compared to `imfill`'s solving the Laplace equation  $\nabla f = 0$  in regions marked as glare).

The method for deciding what is considered glare is similar to [4], in which we consider any intensities close the maximum intensity in the image (Almoussa et al. used 80% of max intensity, and we use  $175/255 \approx 68\%$ ). This threshold is dependent on the image domain.

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



**FIGURE 11:** Comparison of glare inpainting methods (detail)

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

A comparison of these methods is shown in fig. 10, and a zoomed in portion is shown in fig. 11. In the top left, the glary image is shown. In the top middle, regions above the threshold intensity are masked (shown in dark red, along with the background). In the top right, the strategy is “mean window” with a window size of 15 pixels. The bottom left uses “boundary median” strategy. The middle is the more expensive “biharmonic inpainting” strategy, and the bottom right uses a “hybrid” strategy.

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

---

```

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

```

13 Out [6]: 0.011444460505513942  
14

---

We stress again that only a small subset our image domain exhibits disruptive amounts of glare. Future improvements in this direction should probably seek to implement more robust method such as [29] that is not dependent on an arbitrary global threshold for deciding what regions exhibit glare.

## Multiscale Setup

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

Although we cannot expect *a priori* that there is an direct proportionality between our scale size  $\sigma$  and (even some function of) the width of a particular vessel [11], we generally expect to isolate narrower curvilinear structures at smaller scales, and thicker curvilinear structures at larger scales. The smallest one should be an effective size where details are expected to be isolated, and the largest should be an effective size as well. In fact, following [15], it is reasonable and natural to select these logarithmically; that is, for some selected inputs  $m < M$  we have

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

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

feasible to collect so many large scale readings. This is much less of an issue with the present implementation.

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

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

```
% DECISION TREE
For each sample:
```

- A) Preprocessing
  - 1) RGB to single channel
    - + Luminance transform
    - . Isolate green channel only (Almoussa, Huynh)
  - 2) Cut removal
  - 3) Remove glare
    - a) Mask glare
      - + Threshold (\*175/255\*)
      - . Threshold at \*80%\* of max intensity (Almoussa)
      - . Lange (2005) (multistep procedure, done in RGB space actually)
    - b) Post-process mask
      - + Dilate with radius \*2\*
      - . Do nothing
    - c) Inpaint glare
      - + Hybrid inpainting, with size threshold \*32\*
      - . Biharmonic inpainting
      - . Mean value of boundary
      - . Median value of boundary

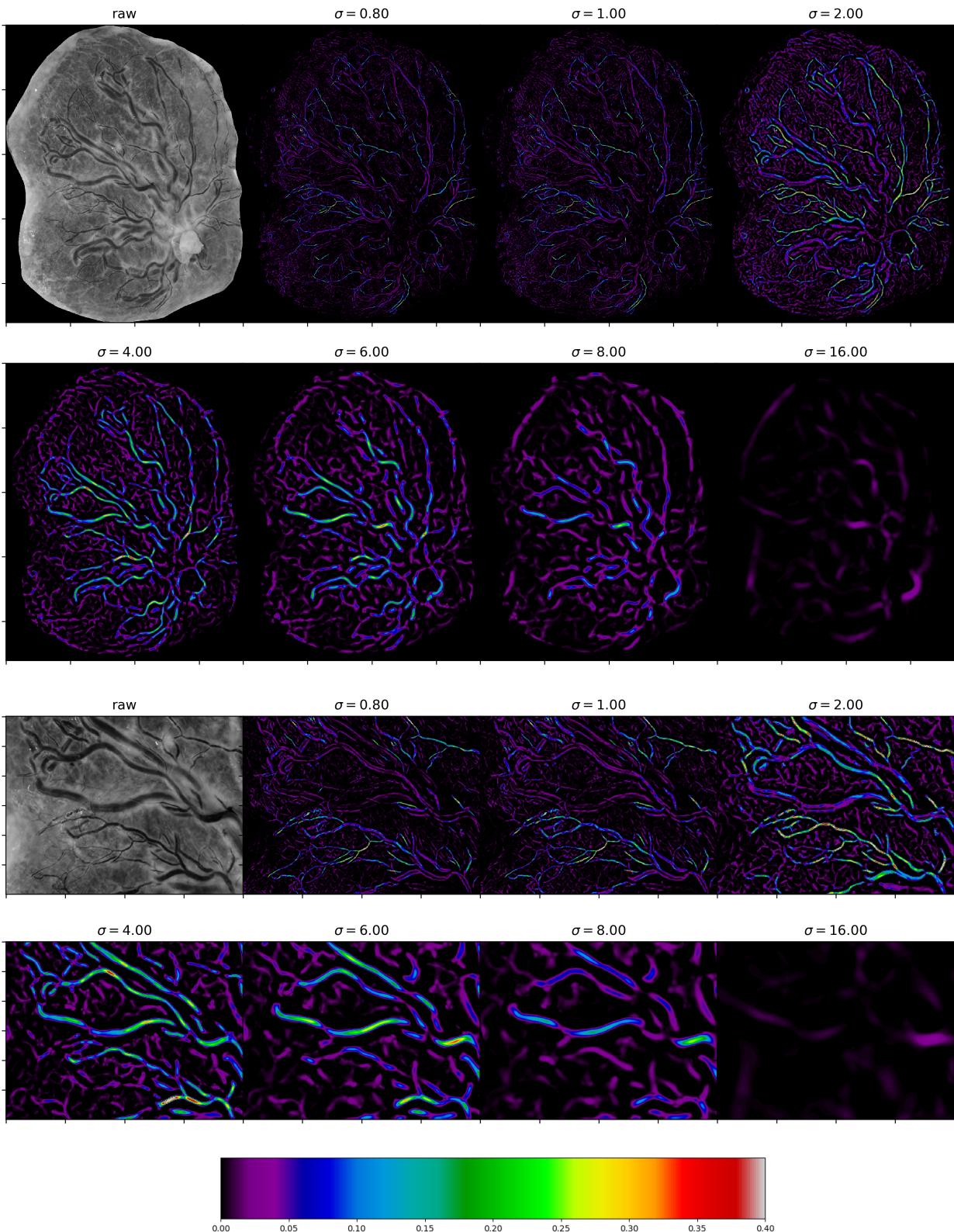
```

    . Windowed mean (radius: *15*)

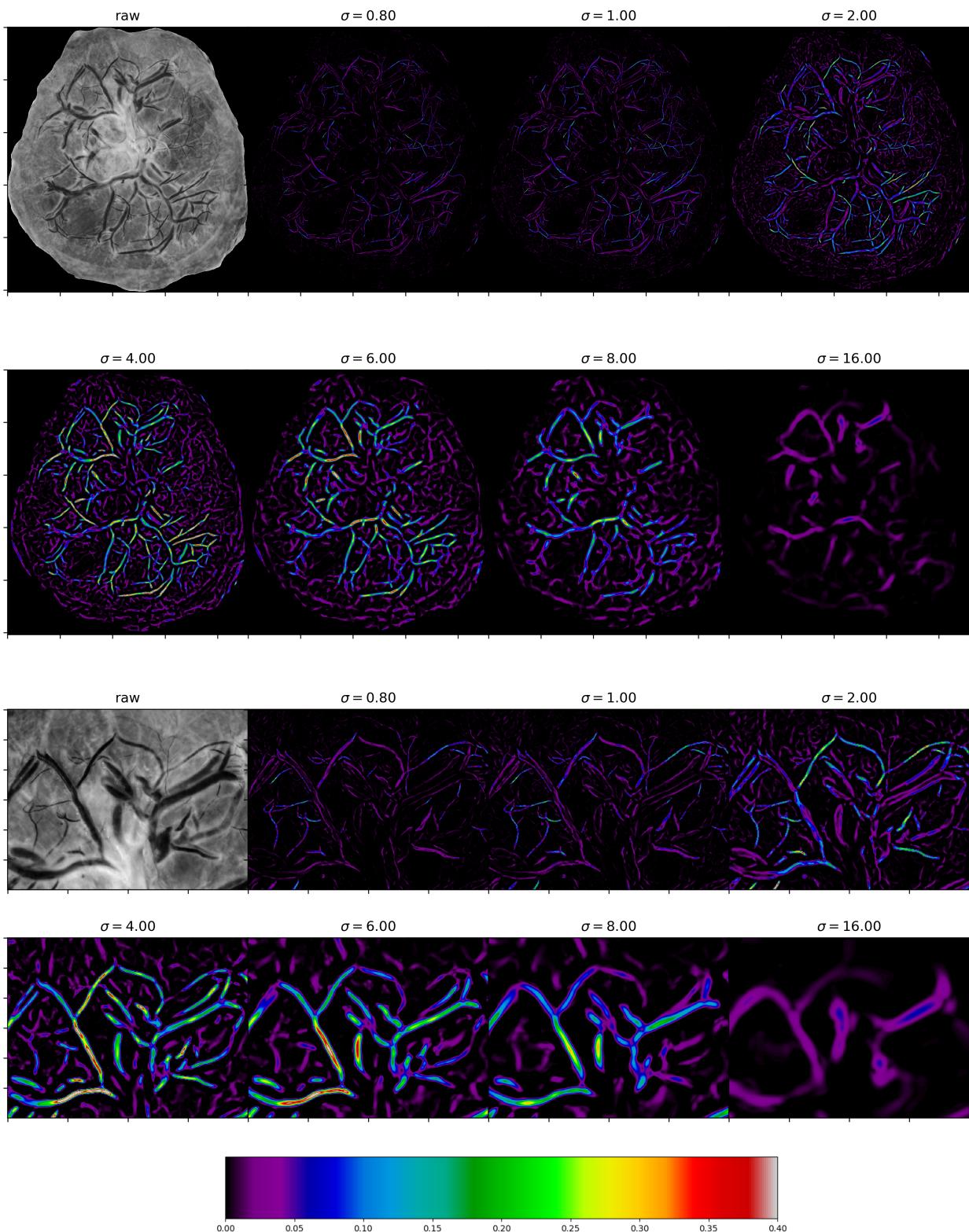
B) Multiscale Frangi filter
1) Define parameters
    a) Scales
        = n_scales (default: *40*)
        = scale_range (default *[-2, 3.5]*)
        = scale_type (*logarithmic base 2* or linear or custom)
        -> build scales
    b) Betas
        = *0.5* each scale or custom range
    c) Gammas
        = strategy: (half L2 hessian norm or *half hessian frobenius norm*)
                    or custom value each scale
        = redilate plate per scale (?)
    d) Dilate per scale
        + Custom function of scale
            (default *max{10, int(4sigma)} if (sigma < 20) else int(2*sigma)*)
        . No dilation
    e) Scale space convolution method
        + Discrete Gaussian kernel with FFT
        . Sampled gaussian kernel with FFT
        . Sample gaussian kernel, standard convolution
2) For each sigma: do Uniscale Frangi Filter
    a) gauss blur image with method from (1e)
    b) take gradient across each axis, take gradient across each axis
        of gradient to get Hxx, Hxy, Hy
    c) find eigenvalues of hessian at each point (using np.eig)
        and sort by magnitude
    d) zero out principal directions according to Dilate Per Scale
    e) zero out hessian according to max(ceil(sigma),10)
    f) Calculate Frangi Vesselness Measure
C) Estimate PCSVN
1) Approximate using strategy
    a) Calculate Fmax and Fmax.where -> Fmax
    b) Threshold at 95th percentile -> approx
    c) Threshold at fixed alpha
    d) Margin adding to one of the above
    e) Random walker after margin adding
2) Compare to Trace
3) Calculate Network Coverage and MCC score

```

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



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



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

## CHAPTER 4

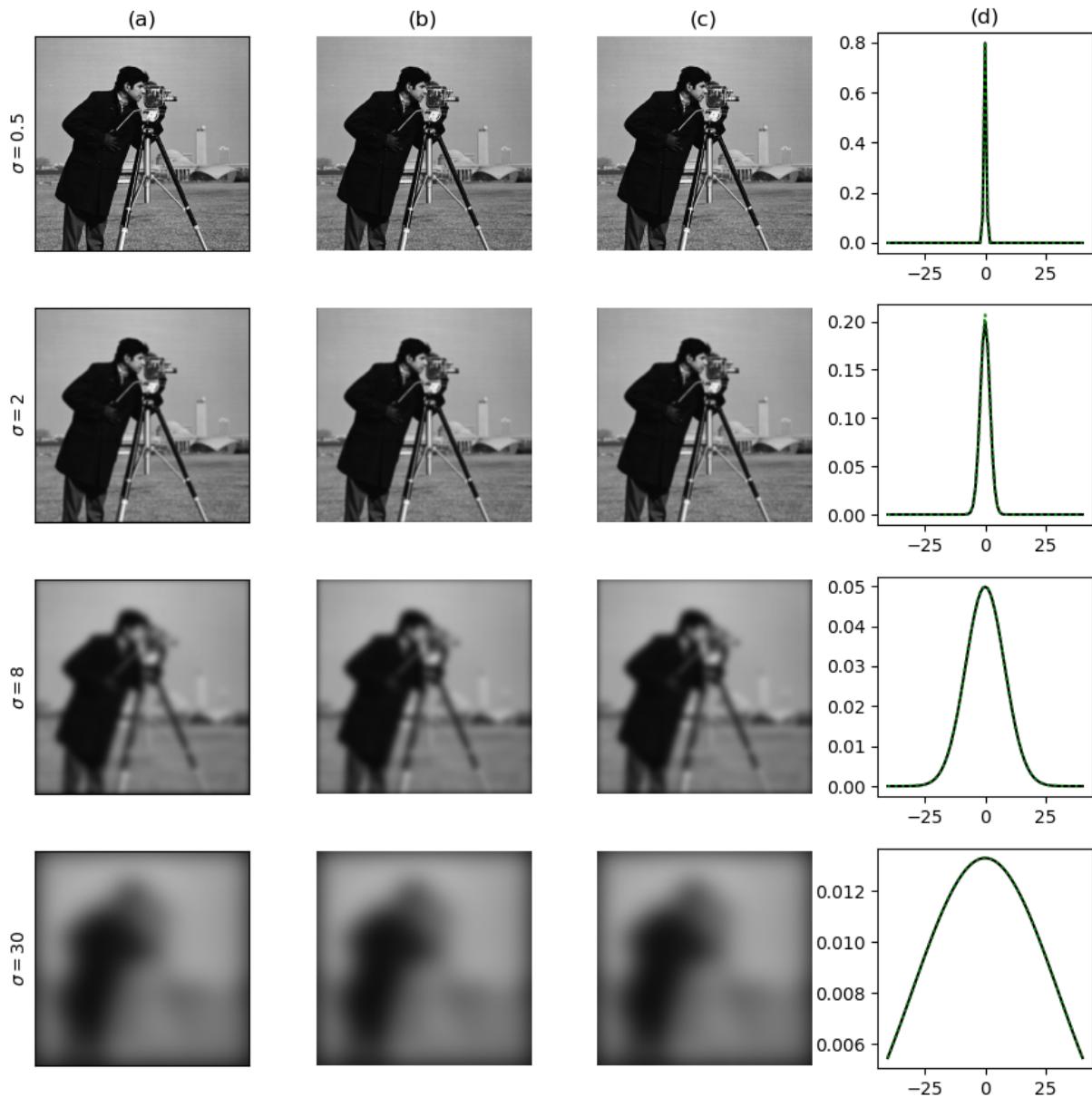
### IMPLEMENTATIONS

#### Calculating the Hessian via FFT

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

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

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



**FIGURE 14:** Compatibility of Gaussian convolution strategies

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

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

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

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

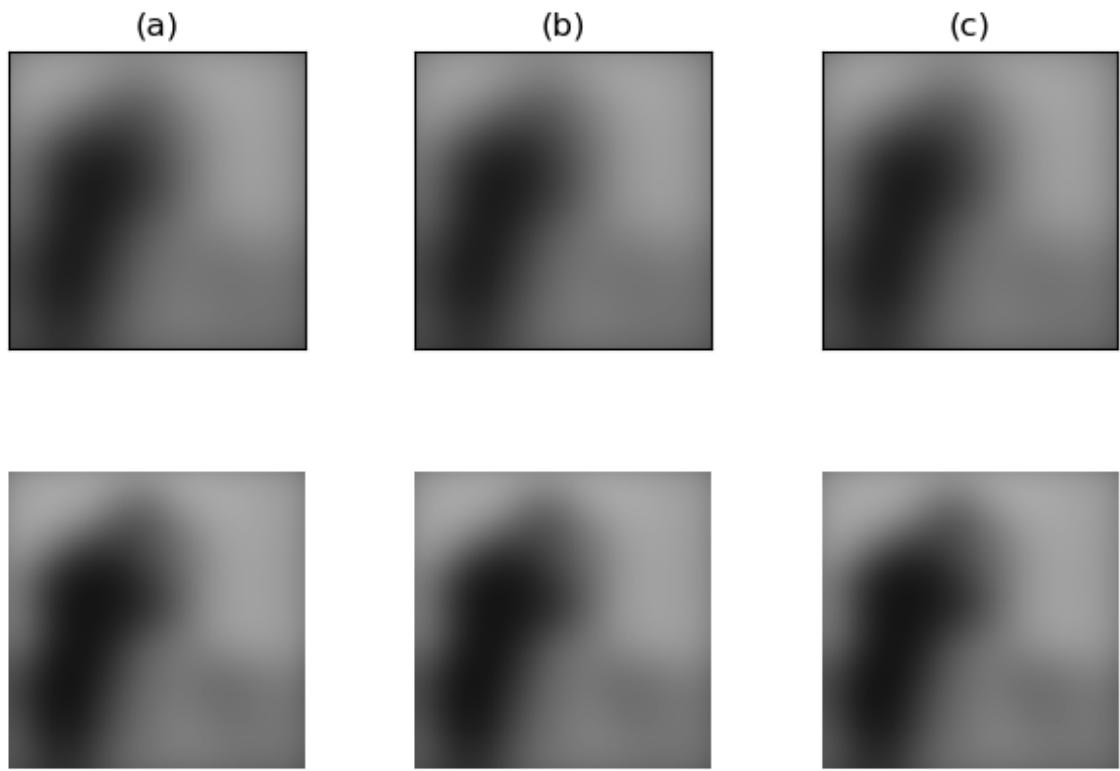
**TABLE 3:** MSE of Frangi scores  $\sigma = 0.3$

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

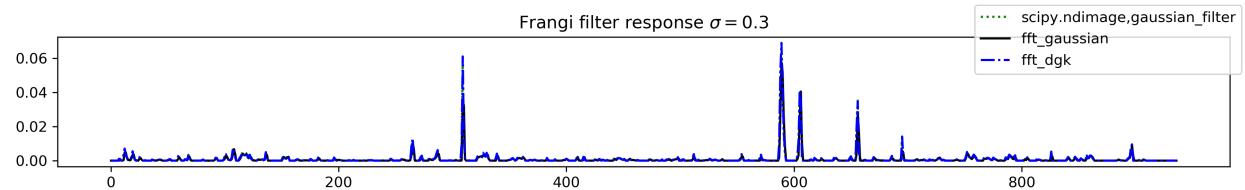
We could show this again with a zero border or maybe even just a 1D signal.

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

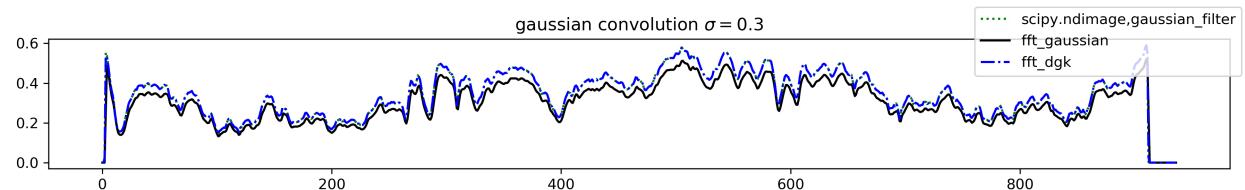
Finally, we wish to demonstrate the point of this comparison—that *FFT-based* convolution is much faster than spatial convolution. We took a much larger sample



**FIGURE 15:** Iterative Gaussian blur



**FIGURE 16:** Image cross-section of Gaussian blurred images



**FIGURE 17:** Image cross-section of Frangi targets images

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

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

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

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

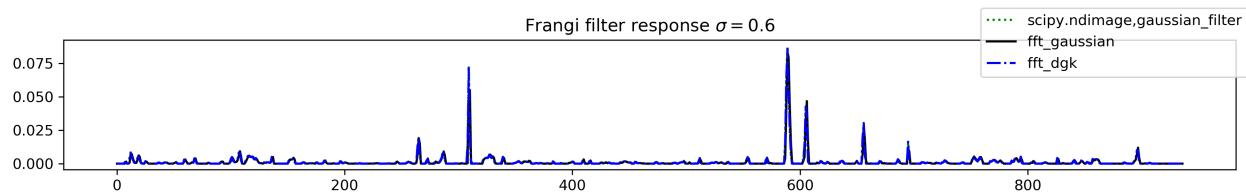


FIGURE 18: Image cross-section of Gaussian blurred images  $\sigma = 5$

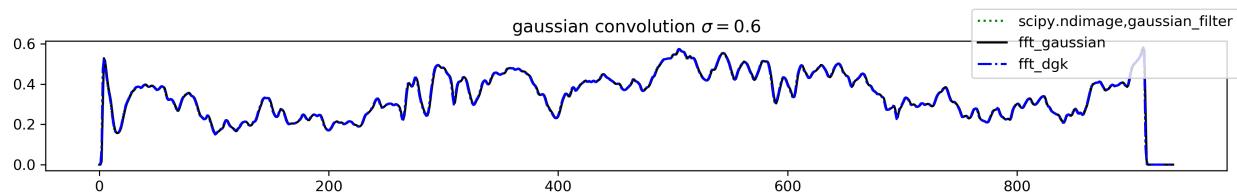
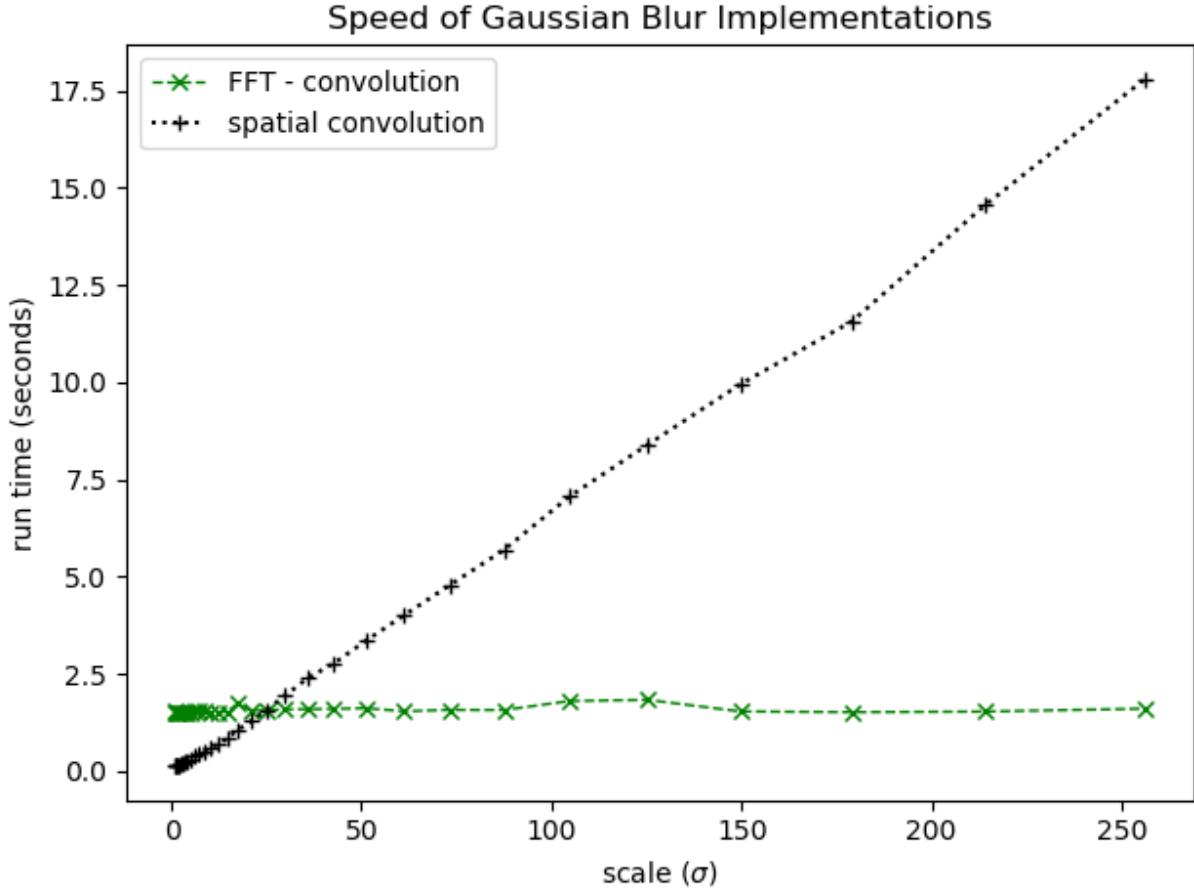


FIGURE 19: Image cross-section of Frangi targets images  $\sigma = 5$



**FIGURE 20:** Time required

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

### Postprocessing Techniques

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

analysis of the Frangi score apart from taking the maximum across all scales, as in eq. (2.92). Still, we wish to demonstrate some several immediate methods of postprocessing these samples in order to illustrate the usefulness of this optimized Frangi filter.

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

### **Method A: Fixed Threshold**

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

### **Method B: Percentile Based Merging**

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

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

### **Method C: Scale-Based Random Walker**

This method isn't working well and I should remove it, but the idea is to use a fill to the edge of the vessel and connect them.

### **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.

## CHAPTER 5

### RESULTS AND ANALYSIS

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

#### Sample visual output

In fig. 21 and fig. 22 we take a partial look at the Frangi output for two particularly well-behaved samples. In the top-left, the preprocessed placenta is shown. In the top-right, the maximum of the Frangi output over  $N$  scales. The bottom left and right images are simple segmentation strategies of merging the result.

#### Binary Classifications and the confusion matrix

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

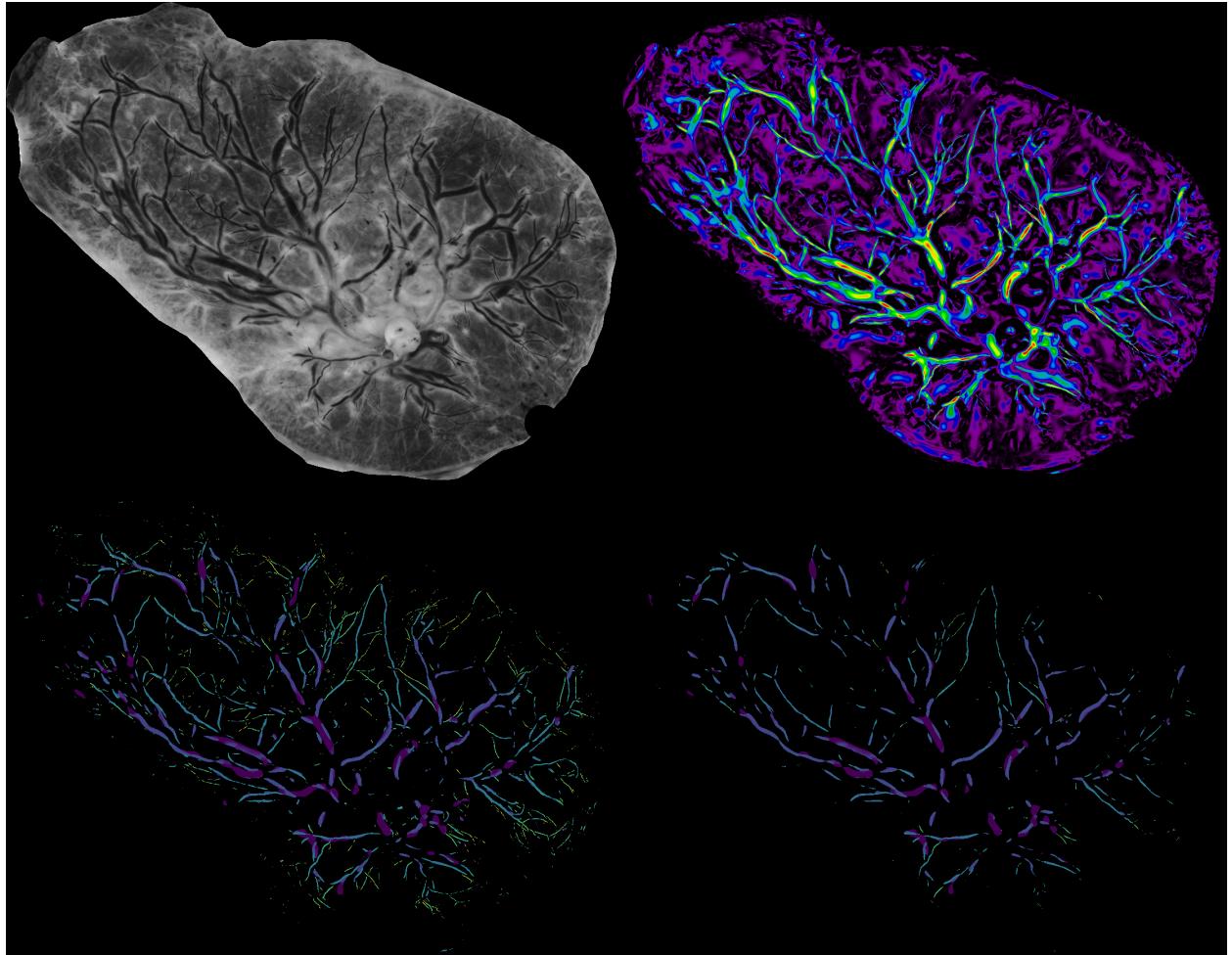
use MCC [30]

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

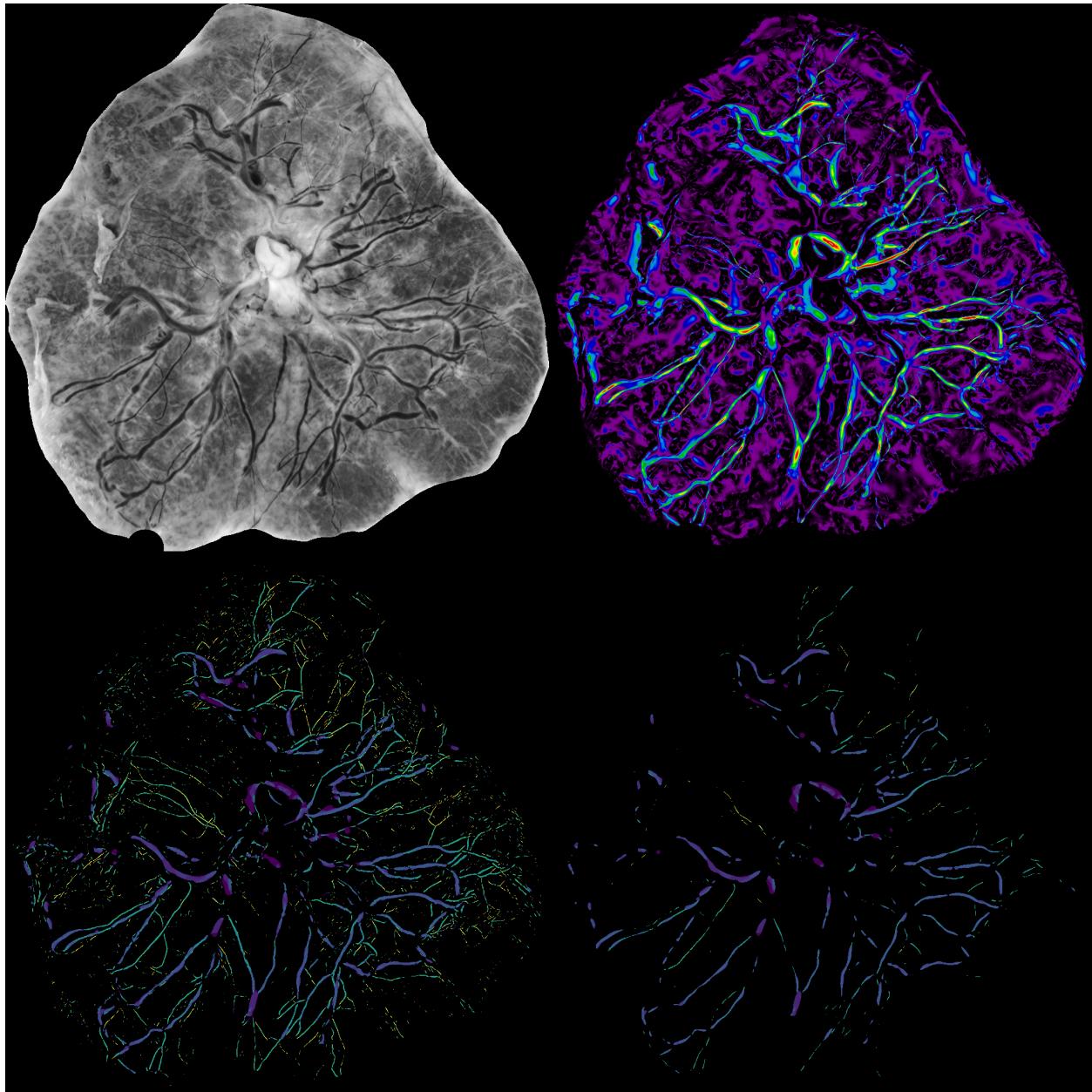
Sometimes the output doesn’t agree with the trace, i.e. “the ground truth” is not 100% correct. sometimes either there’s a false negative (reported) but something just wasn’t traced in the original 1602443.

#### Variations in the Data Set and Imperfections of the Ground Truth

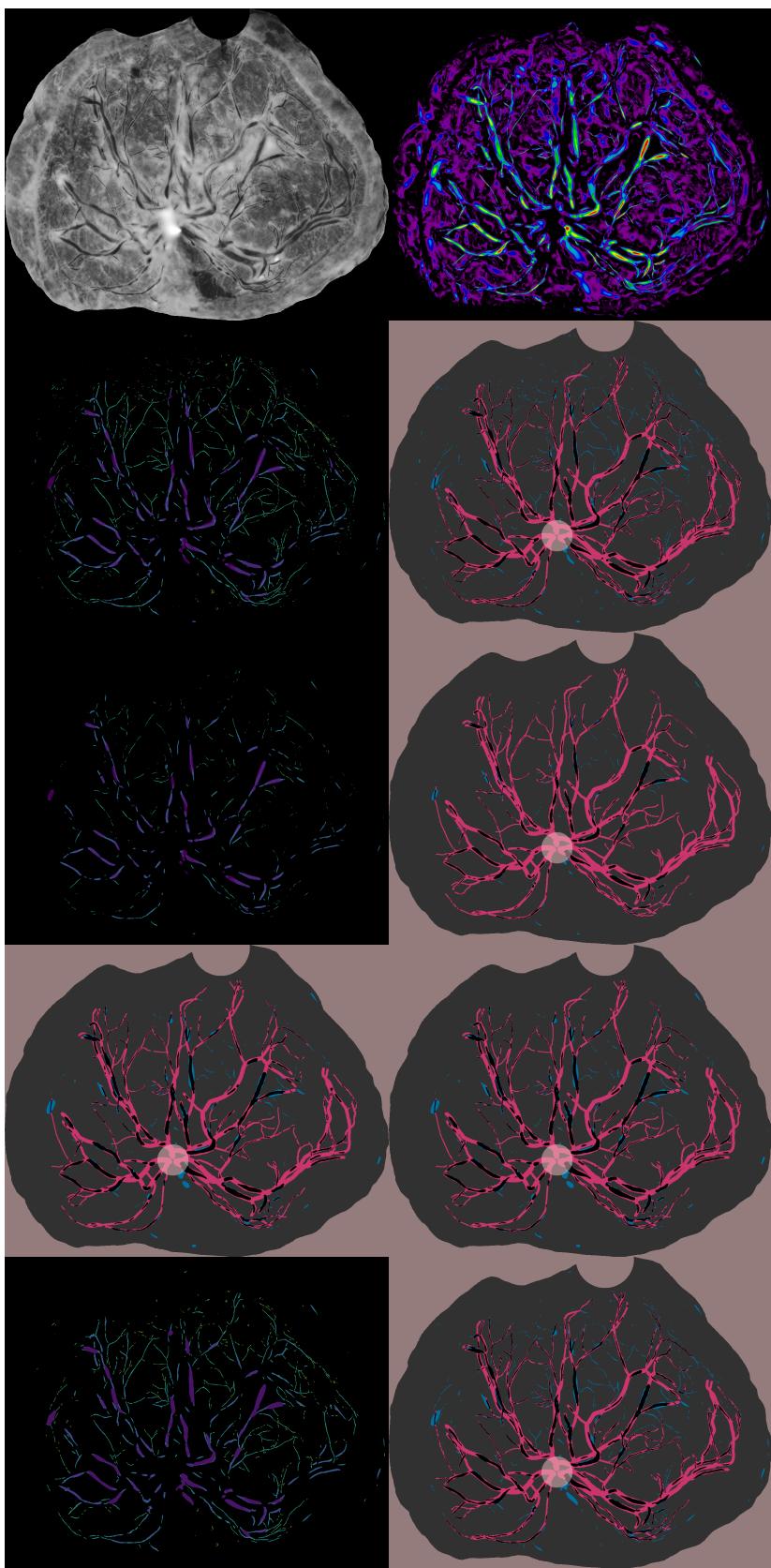
1. Collar is stupid and should really be considered like a error in marking the perimeter. Throw these away or edit. Maybe make a section called discarded



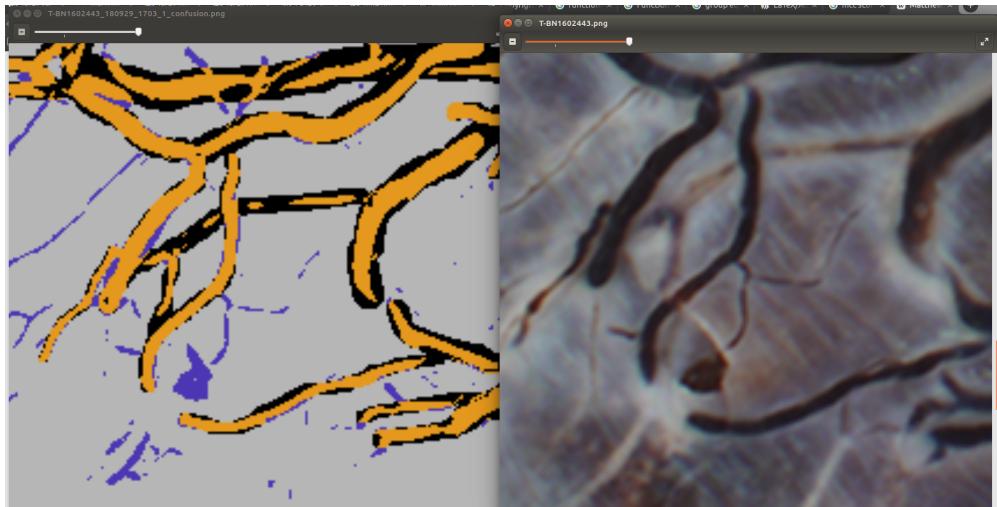
**FIGURE 21:** Sample Multiscale Frangi output ( $\beta = 0.35$ ) with simple segmentation strate-  
gies (Example 1)



**FIGURE 22:** Sample Multiscale Frangi output ( $\beta = 0.35$ ) with simple segmentation strategies (Example 2)



**FIGURE 23:** Demonstration of postprocessing techniques  
80



**FIGURE 24:** "True" false positives and "False" false positives

samples that's stupid but yeah.

2. Vessels suck sometimes. In the portion above, 1602443, there's a random blood clot which gets identified at large  $\sigma$ . But also the small forked shaped thing which is obviously a vessel doesn't get defined.
3. Too much blood (not enough?? no idea) is left in the vessels. leading to the weird white border around some vessels. you could identify these along with black center and combine them somehow. no idea. Also, holy shit, some of the white vessel "sleeves" ARE identified in the tracing, and some aren't. Find an example of this and whine about it.
4. Umbilical cord insertion point is stupid and obscures a lot. The tracer guesses but there's no real guiding principle AFAIK..
5. Small vessels aren't accounted for at all. Not sure how to coincide measurement in terms of scale space anymore, but should figure out how to cut off those values before running MCC metric.

## **Results**

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

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

## **Answer Research Questions**

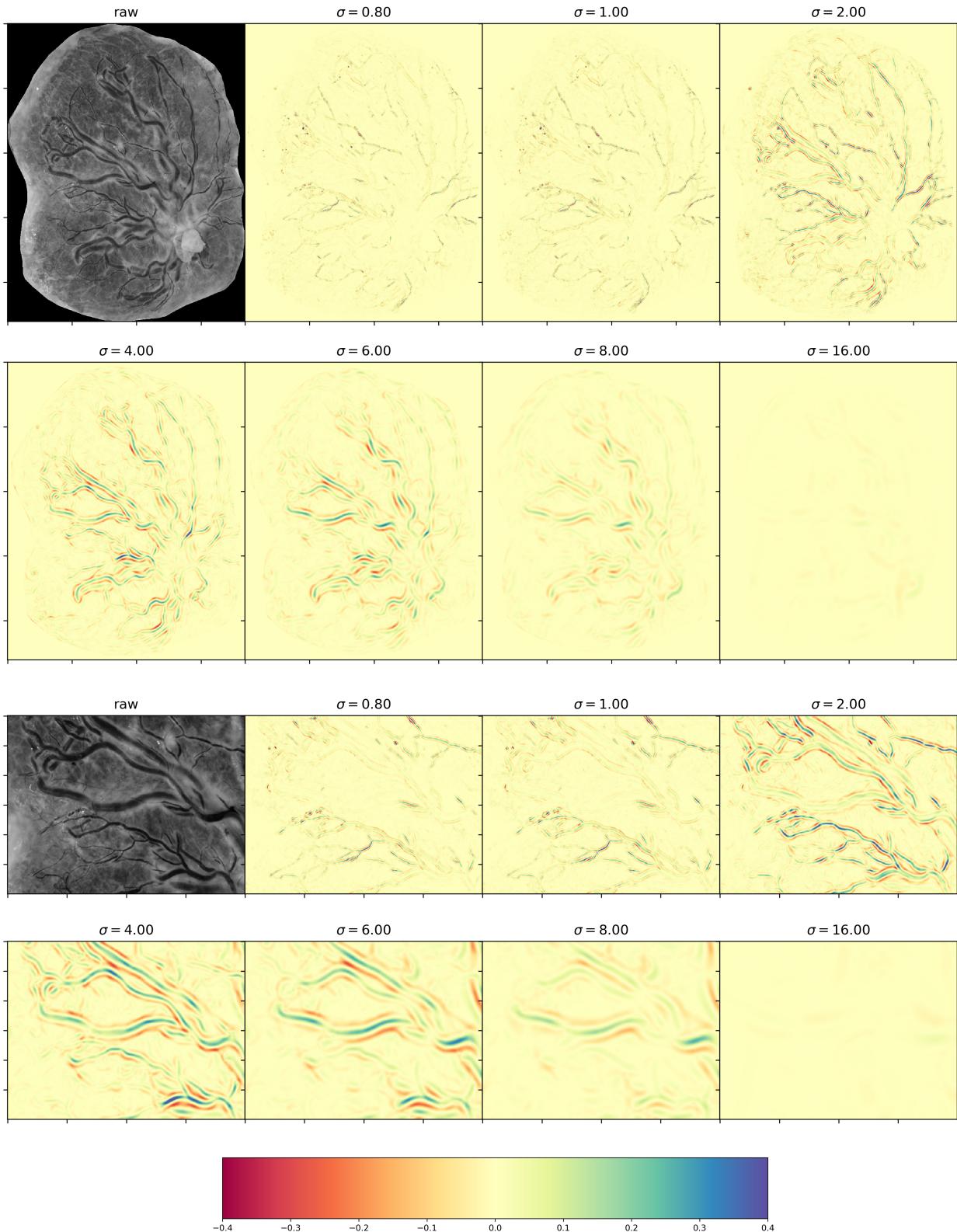
## CHAPTER 6

### CONCLUSION

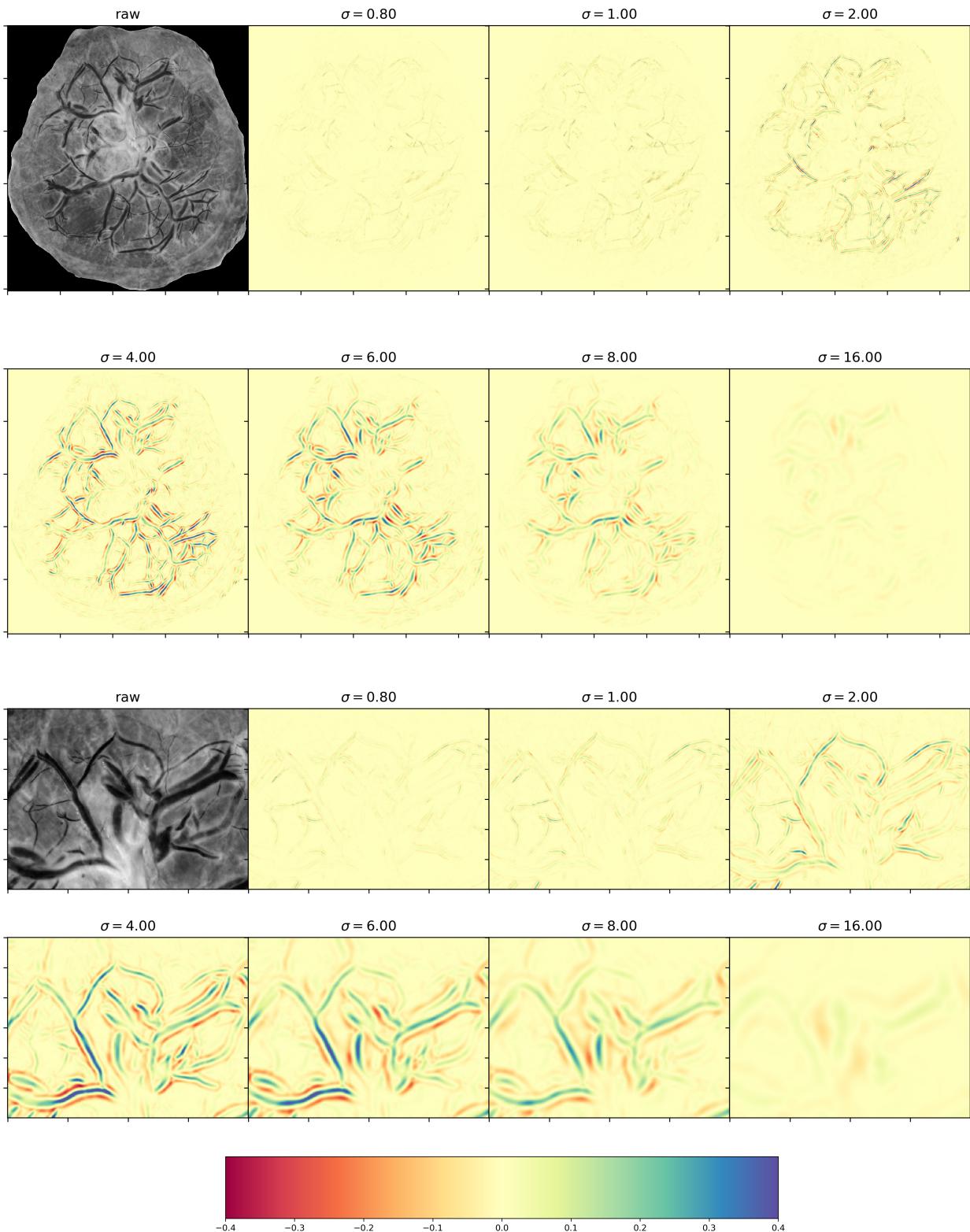
We justified the use of differential geometry in 2D discrete image processing, and vastly improved upon the implementation of the Frangi filter. Our improved implementation allowed us to take more steps in our multiscale method and thus choose stricter parameters for Frangi scale. We used our multiscale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

#### Future research directions

- Solve the Network Connection Problem (PICTURE OF GAPS) Try something like [31] 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  $\sigma_{\min}$  and  $\sigma_{\max}$ .
- Look into gradient prefiltering more.
- Look into using signed frangi arguments.
- Use this as preprocessing for a Neural Network. (cite kara's work, katalinas work)
- Apply to more image domains (STARE, other placental domains).



**FIGURE 25:** Signed Frangi output (plate and inset) (Example 1)



**FIGURE 26:** Signed Frangi output (plate and inset) (Example 1)

## APPENDICES

**APPENDIX A**  
**CODE LISTINGS**

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

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

Earlier versions of these packages may be compatible but are not guaranteed to be so.

The scripts listed in this appendix are also hosted at [github.com/wukm/pycake](https://github.com/wukm/pycake).

#### listings/add\_margins.py

---

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

---

## listings/diffgeo.py

---

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

```

63     for ic in range(img.shape[2]):
64         channel = img[:, :, ic]
65
66         # returns the tuple (Hxx, Hxy, Hyy)
67         if H is None:
68             H = hessian_matrix(channel, sigma=sigma)
69
70         # returns tuple (l1,l2) where l1 >= l2 but this *includes sign*
71         L = hessian_matrix_eigvals(H)
72         L = reorder_eigs(L)
73
74         # Make K2 larger in magnitude, as consistent with Frangi paper
75         K1[:, :, ic] = L[0, :, :]
76         K2[:, :, ic] = L[1, :, :]
77
78     try:
79         mask = img.mask # get mask to add to each if input was a masked array
80
81     except AttributeError:
82
83         pass # there's no mask, so do nothing
84
85     else:
86         K1 = ma.masked_array(K1, mask=mask)
87         K2 = ma.masked_array(K2, mask=mask)
88
89     # now undo the trivial dimension
90     if not multichannel:
91         K1 = np.squeeze(K1)
92         K2 = np.squeeze(K2)
93
94     return K1, K2
95
96
97
98 def reorder_eigs(L):
99     """reorder eigenvalues by decreasing magnitude.
100
101    Eigenvalues are outputted from hessian_matrix_eigvals so that L1 >= L2.
102    This reorders this so that |L1| >= |L2| instead (where L1,L2=L)
103    Parameters
104    -----
105    L: ndarray or iterable of ndarrays
106        As outputted by, say, hessian_matrix_eigs. If a single ndarray, it
107        should be the shape (N, *img.shape) where there are N eigenvalues to
108        reorder. You may also input a tuple like (L1,L2).
109    Returns
110    -----
111    eigs: ndarray
112        The eigenvalues in decreasing order of magnitude; that is
113        eigs[i,j,k] is the ith-largest eigenvalue at position (j, k).
114        Each of these is the same shape the original inputs, but
115        np.abs(L1r) >= np.abs(L2r) will be true. See warning below.
116
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 >>>(np.abs(K1) <= np.abs(K2)).all()
138 False
139 >>>K1r, K2r = reorder_eigs(K1,K2)
140 >>>(K1r <= K2r).all()
141 False
142 >>>(np.abs(K1r) <= np.abs(K2r)).all()
143 True
144
145 TODO
146 -----
147 Support out= keyword
148 """
149 # this will do nothing if L is already an array but will make it an array
150 # if it's a tuple/list/iterable
151 L = np.stack(L)
152 mag = np.argsort(np.abs(L), axis=0)
153
154 # now L2 is larger in absolute value, as consistent with Frangi paper
155 return np.take_along_axis(L, mag, axis=0)
156
157
158 def principal_directions(img, sigma, H=None, mask=None):
159     """Calculate principal directions of
160     will ignore calculation of principal directions of masked areas
161
162     mask should be positive where the PD's should *NOT* be calculated
163     this function actually returns the theta corresponding to
164     leading and trailing principal directions, i.e. angle w / x axis
165     """
166
167     if H is None:
168         H = hessian_matrix(img, sigma)
169
170     Hxx, Hxy, Hyx = H
171
172
173     # determine if there was a supplied mask or use images if it exists
174     if mask is None:
175         try:
176             mask = img.mask
177         except AttributeError:
178             masked = False
179         else:
180             masked = True
181     else:
182         masked = True
183
184     dims = img.shape
185
186     # where to store
187     trailing_thetas = np.zeros_like(img, dtype='float64')
188     leading_thetas = np.zeros_like(img, dtype='float64')
189
190     # maybe implement a small angle correction

```

```

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

```

```

255     # toc = time.time()
256     # print('time elapsed: ', toc - tic)
257     # tic = time.time()
258     # print('calculating principal curvatures for ={} (fast)'.format(sigma))
259     # k1,k2 = principal_curvatures(img, sigma=sigma, H=h)
260
261     # toc = time.time()
262     # print('time elapsed: ', toc - tic)
263     # tic = time.time()
264
265     ######
266
267     # print('calculating targets for ={}'.format(sigma))
268     # T = get_targets(K1,K2, threshold=False)
269
270     # toc = time.time()
271     # print('time elapsed: ', toc - tic)
272     # tic = time.time()
273
274     # print('calculating targets for ={} (fast)'.format(sigma))
275     # t = get_targets(k1,k2, threshold=False)
276
277     # toc = time.time()
278     # print('time elapsed: ', toc - tic)
279
280     #####
281
282     # print('extending masks')
283
284     # extend mask over nontargets items
285     # img1 = ma.masked_where( T < T.mean(), img)
286     # img2 = ma.masked_where( t < t.mean(), img)
287
288     # tic = time.time()
289     # print('calculating principal directions for ={}'.format(sigma))
290     # T1,T2 = principal_directions(img1, sigma=sigma, H=H)
291     # toc = time.time()
292     # print('time elapsed: ', toc - tic)
293     # tic = time.time()
294
295     # print('calculating principal directions for ={} (fast)'.format(sigma))
296     # t1,t2 = principal_directions(img2, sigma=sigma, H=h)
297     # toc = time.time()
298     # print('time elapsed: ', toc - tic)

```

---

### listings/extract\_NCS\_pcsvn.py

---

```

1 #!/usr/bin/env python3
2 """
3 This is the main program. It approximates the PCSVN of a list of samples.
4 It does not do network completion.
5 """
6
7 """
8 from placenta import (get_named_placenta, cropped_args, cropped_view,
9                         list_placentas, list_by_quality, open_typefile,
10                        open_tracefile, add_ucip_to_mask, measure_ncs_markings)
11
12 from merging import nz_percentile, apply_threshold
13 from scoring import (compare_trace, rgb_to_widths, merge_widths_from_traces,
14                      filter_widths, mcc, confusion, skeletonize_trace)
15
16

```

```

17 from pcsvn import extract_pcsvn, scale_label_figure, get_outname_lambda
18 from preprocessing import inpaint_hybrid
19
20 import numpy as np
21 import numpy.ma as ma
22
23 import matplotlib.pyplot as plt
24
25 import os.path
26 import os
27 import json
28 import datetime
29 import pandas
30
31 # for some post_processing, this needs to be moved elsewhere
32 from skimage.filters import sobel
33 from frangi import frangi_from_image
34 from plate_morphology import dilate_boundary
35 from skimage.morphology import remove_small_holes, remove_small_objects
36 from skimage.segmentation import random_walker
37
38
39 # INITIALIZE SAMPLES -----
40 #     There are several ways to initialize samples. Uncomment one.
41
42 # load all 201 samples
43 placentas = list_placentas('T-BN')
44 # load placentas from a certain quality category 0=good, 1=okay, 2=fair, 3=poor
45
46 #placentas = list_by_quality(0, N=2)
47 #placentas.extend(list_by_quality(1, N=1))
48 #placentas.extend(list_by_quality(2, N=1))
49 #placentas.extend(list_by_quality(3, N=1))
50
51 # load from a file (sample names are keys of the json file)
52 # placentas = list_by_quality(json_file='manual_batch.json')
53
54 # for a single named sample, use a 1 element list.
55 # placentas = ['T-BN0204423.png']
56
57 n_samples = len(placentas)
58
59 # RUNTIME OPTIONS -----
60 #     Where to save and whether or not to use old targets.
61
62 MAKE_NPZ_FILES = True # pickle frangi targets if you can
63 USE_NPZ_FILES = True # use old npz files if you can
64 NPZ_DIR = 'output/181122-bigrun' # where to look for npz files
65 OUTPUT_DIR = 'output/181122-bigrun' # 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 = False

```

```

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 REMOVE_GLARE = True
89
90 # What scales to use!
91 log_range = (-1, 3.5)
92 n_scales = 40
93
94 # when showing "large scales only", this is where to start
95 # (some index between 0 and n_scales)
96 L0_offset = 8
97
98 # Explicit Frangi Parameters (pass an array as long as scales or pass None)
99 betas = None # None -> use default parameters (0.5)
100 gammas = None # None -> use default parameters (calculate half of hessian norm)
101 alphas = None # none to set later
102 fixed_alpha = .2
103
104 # Scoring Decisions (don't need to touch these)
105 ucip_radius = 50 # area around the umbilical cord insertion point to ignore
106
107 # some other initializations, don't mind me
108
109
110
111
112
113 # CODE BEGINS HERE -----
114
115 n_samples = len(placentas)
116 scales = np.logspace(log_range[0], log_range[1], num=n_scales, base=2)
117 mccs = dict() # empty dict to store MCC's of each sample
118 pnccs = dict() # empty dict to store percent network covered for each sample
119
120 if not os.path.exists(OUTPUT_DIR):
121     os.makedirs(OUTPUT_DIR)
122
123 print(n_samples, "samples total!")
124
125 for i, filename in enumerate(placentas):
126
127     print('*'*80)
128     print(f'extracting PCSVN of {filename}\t({i} of {n_samples})')
129
130     # --- Setup, Preprocessing, Frangi Filter (it's mixed up) -----
131
132     raw_img = get_named_placenta(filename)
133
134     if REMOVE_GLARE:
135         img = inpaint_hybrid(raw_img)
136     else:
137         img = raw_img # in case preprocessing happens in extract_pcsvn
138
139     if USE_NPZ_FILES:
140         # find the first npz file with the sample name in it in the
141         # specified directory.
142         stub = filename.rstrip('.png')
143         for f in os.scandir(NPZ_DIR):
144             if f.name.endswith('npz') and f.name.startswith(stub):

```

```

145         npz_filename = os.path.join(NPZ_DIR, f.name)
146         print(f'using the npz file {npz_filename}')
147         break # we'll just use the first one we can find.
148     else:
149         print(f'no npz file found for {filename}.')
150         npz_filename = None
151     else:
152         npz_filename = None
153
154     # set a lambda function to make output file names
155     outname = get_outname_lambda(filename, output_dir=OUTPUT_DIR)
156
157 if npz_filename is not None:
158
159     F = np.load(npz_filename)[‘F’]
160
161     # in case preprocessing happens inside extract_pcsvn, do it out here
162
163     print(‘successfully loaded the frangi targets! ’)
164
165 else:
166     print(‘finding multiscale frangi targets’)
167
168     # F is an array of frangi scores of shape (*img.shape, n_scales)
169     F, jfile = extract_pcsvn(img, filename, dark_bg=DARK_BG, betas=betas,
170                             scales=scales, gammas=gammas,
171                             kernel=‘discrete’, dilate_per_scale=True,
172                             verbose=False, signed_frangi=SIGNED_FRANGI,
173                             generate_json=True, output_dir=OUTPUT_DIR)
174
175 if MAKE_NPZ_FILES:
176     npzfile = “.”.join((outname(“F”).rsplit(‘.’, maxsplit=1)[0], ‘npz’))
177
178     print(“saving frangi targets to ”, npzfile)
179     np.savez_compressed(npzfile, F=F)
180
181 # --- Merging & Postprocessing -----
182
183 # This is the maximum frangi response over all scales at each location
184 Fmax = F.max(axis=-1)
185
186 print(“...making outputs”)
187
188 if alphas is None:
189     print(“thresholding alphas with top 5% scores at each scale”)
190     alphas = np.array([nz_percentile(F[:, :, k], 95.0)
191                       for k in range(n_scales)])
192
193 scale_maxes = np.array([F[..., i].max() for i in range(F.shape[-1])])
194 #print(‘percentile alphas:’, alphas)
195 #print(‘max at each scale:’, scale_maxes)
196 table = pandas.DataFrame(np.dstack((scales, alphas, scale_maxes)).squeeze(),
197                           columns=(‘’, ‘_p’, ‘max(F_)’))
198
199 print(table)
200 # threshold the responses at each of these values and get labels of max
201 approx, labs = apply_threshold(F, alphas, return_labels=True)
202
203 # --- Scoring and Outputs -----
204
205 # get the main (boolean) tracefile and the RGB tracefiles
206 trace = open_tracefile(filename, as_binary=True)
207 A_trace = open_typefile(filename, ‘arteries’)
208 V_trace = open_typefile(filename, ‘veins’)

```

```

209 skeltrace = skeletonize_trace(A_trace, V_trace)
210
211 # get a matrix of pixel widths in the trace
212 widths = merge_widths_from_traces(A_trace, V_trace, strategy='arteries')
213
214 # find cord insertion point and resolution of the image
215 ucip_midpoint, resolution = measure_ncs_markings(filename=filename)
216
217 # if verbose:
218 # print(f"The umbilical cord insertion point is at {ucip_midpoint}")
219 # print(f"The resolution of the image is {resolution} pixels per cm.")
220
221 # mask anywhere close to the UCIP
222 ucip_mask = add_ucip_to_mask(ucip_midpoint, radius=int(ucip_radius),
223                             mask=img.mask)
224
225 # The following are examples of things you can do:
226
227 # matrix of widths of traced image
228 # min_widths = merge_widths_from_traces(A_trace, V_trace,
229 #                                         strategy='minimum')
230
231 # trace ignoring largest vessels (19 pixels wide)
232 # trace_smaller_only = filter_widths(min_widths, min_width=3, max_width=17)
233 # trace_smaller_only != 0
234
235 # use only some scales
236 #approx_L0, labs_L0 = apply_threshold(F[:, :, L0_offset:], alphas[L0_offset:])
237 approx_FA, labs_FA = apply_threshold(F, fixed_alpha)
238
239 # fix labels to incorporate offset
240 #labs_L0 = (labs_L0 != 0)*(labs_L0 + L0_offset)
241
242 # confusion matrix against default trace
243 confuse = confusion(approx, trace, bg_mask=ucip_mask)
244 #confuse_L0 = confusion(approx_L0, trace, bg_mask=ucip_mask)
245 confuse_FA = confusion(approx_FA, trace, bg_mask=ucip_mask)
246
247 m_score, counts = mcc(approx, trace, ucip_mask, return_counts=True)
248 m_score_FA, counts_FA = mcc(approx_FA, trace, ucip_mask,
249                             return_counts=True)
250
251 # this all just verifies that the 4 categories were added up
252 # correctly and match the total number of pixels in the reported
253 # placental plate.
254 TP, TN, FP, FN = counts # return these for more analysis?
255
256 total = np.invert(ucip_mask).sum()
257 print(f'TP: {TP}\t TN: {TN}\nFP: {FP}\tFN: {FN}')
258 # just a sanity check
259 print(f'TP+TN+FP+FN={TP+TN+FP+FN}\ttotal pixels={total}')
260
261 # MOVE THIS ELSEWHERE
262 s = sobel(img)
263 s = dilate_boundary(s, mask=img.mask, radius=20)
264 finv = frangi_from_image(s, sigma=0.8, dark_bg=True, dilation_radius=10)
265 finv_thresh = nz_percentile(finv, 80)
266 margins = remove_small_objects((finv > finv_thresh).filled(0), min_size=32)
267 margins_added = np.logical_or(margins, approx)
268 margins_added = remove_small_holes(margins_added, area_threshold=100,
269                                   connectivity=2)
270
271 confuse_margins = confusion(margins_added, trace, bg_mask=ucip_mask)
272
```

```

273 # random walker markers
274 markers = np.zeros(img.shape, dtype=np.uint8)
275 markers[Fmax < .1] = 1
276 markers[margins_added] = 2
277 rw = random_walker(img, markers, beta=1000)
278 approx_rw = (rw == 2)
279 confuse_rw = confusion(approx_rw, trace, bg_mask=ucip_mask)
280 m_score_rw, counts_rw = mcc(approx_rw, trace, ucip_mask,
281                             return_counts=True)
282 pnc_rw = np.logical_and(skeltrace, approx_rw).sum() / skeltrace.sum()
283
284 mccs[filename] = (m_score, m_score_FA, m_score_rw)
285
286 print(f'mcc score of {m_score:.3} for {filename}')
287 #print(f'mcc score of {m_score_L0:.3} with larger sigmas only')
288 print(f'mcc score of {m_score_rw:.3} after random walker')
289 # --- Generating Visual Outputs-----
290 crop = cropped_args(img) # these indices crop out the mask significantly
291
292 # save the raw, unaltered image
293 plt.imsave(outname('0_raw'), raw_img[crop].filled(0), cmap=plt.cm.gray)
294
295 # save the preprocessed image
296 plt.imsave(outname('1_img'), img[crop].filled(0), cmap=plt.cm.gray)
297
298 # save the maximum frangi output over all scales
299 plt.imsave(outname('2_fmax'), Fmax[crop], vmin=0, vmax=1.0,
300             cmap=plt.cm.nipy_spectral)
301
302 # only save the colorbar the first time
303 save_colorbar = (i==0)
304 scale_label_figure(labs, scales, crop=crop,
305                     savefilename=outname('3_labeled'), image_only=True,
306                     save_colorbar_separate=save_colorbar,
307                     output_dir=OUTPUT_DIR)
308
309 plt.imsave(outname('4_confusion'), confuse[crop])
310
311 #plt.imsave(outname('7_confusion_L0'), confuse_L0[crop])
312 plt.imsave(outname('7_confusion_FA'), confuse_FA[crop])
313 plt.imsave(outname('A_confusion_rw'), confuse_rw[crop])
314
315 plt.imsave(outname('9_margin_for_rw'), confuse_margins[crop])
316 percent_covered = np.logical_and(skeltrace, approx).sum() / skeltrace.sum()
317 percent_covered_FA = np.logical_and(skeltrace,
318                                     approx_FA).sum() / skeltrace.sum()
319
320 pncs[filename] = (percent_covered, percent_covered_FA, pnc_rw)
321
322
323 st_colors = {
324     'TN': (79, 79, 79), # true negative# 'f7f7f7'
325     'TP': (0, 0, 0), # true positive # '000000'
326     'FN': (201, 53, 108), # false negative # 'f1a340' orange
327     'FP': (92, 92, 92), # false positive
328     'mask': (247, 200, 200) # mask color (not used in MCC calculation)
329 }
330
331
332 print('percentage of skeltrace covered:', f'{percent_covered:.2%}')
333 print('percentage of skeltrace covered (larger sigmas only):',
334       f'{percent_covered_FA:.2%}')
335 print('percentage of skeltrace covered (random_walker):',
336       f'{pnc_rw:.2%}')

```

```

337 plt.imsave(outname('5_coverage'), confusion(approx, skeltrace,
338                                         colordict=st_colors)[crop])
339 #plt.imsave(outname('8_coverage_L0'), confusion(approx_L0, skeltrace)[crop])
340 plt.imsave(outname('8_coverage_FA'), confusion(approx_FA, skeltrace,
341                                         colordict=st_colors)[crop])
342 plt.imsave(outname('B_coverage_rw'), confusion(approx_rw, skeltrace,
343                                         colordict=st_colors)[crop])
344
345 # make the graph that shows what scale the max was pulled from
346
347 scale_label_figure(labs_FA, scales, crop=crop,
348                     savefilename=outname('6_labeled_FA'), image_only=True,
349                     save_colorbar_separate=False, output_dir=OUTPUT_DIR)
350 plt.close('all') # something's leaking :(
351
352
353 ### THIS IS ALL A HORRIBLE MESS. FIX IT
354
355 # why don't you just return the dict instead
356 with open(jfile, 'r') as f:
357     slog = json.load(f)
358
359 c2d = lambda t: dict(zip(('TP', 'TN', 'FP', 'FN'), [int(c) for c in t]))
360
361 slog['counts'] = c2d(counts)
362 slog['counts_FA'] = c2d(counts_FA)
363 slog['counts_rw'] = c2d(counts_rw)
364 slog['pnc'] = pncts[filename]
365 slog['mcc'] = mccs[filename]
366 slog['scale_maxes'] = list(scale_maxes)
367 slog['alphas'] = list(alphas)
368
369 with open(jfile, 'w') as f:
370     json.dump(slog, f)
371
372
373
374 # Post-run Meta-Output and Logging -----
375
376 timestamp = datetime.datetime.now()
377 timestamp = timestamp.strftime("%y%m%d_%H%M")
378
379 mccfile = os.path.join(OUTPUT_DIR, f"runlog_{timestamp}.json")
380
381 runlog = {
382     'time': timestamp,
383     'dark_bg': DARK_BG,
384     'dilate_per_scale': DILATE_PER_SCALE,
385     'log_range': log_range,
386     'n_scales': n_scales,
387     'scales': list(scales),
388     'alphas': list(alphas),
389     'betas': None,
390     'use_npz_files': False,
391     'remove_glare': REMOVE_GLARE,
392     'files': list(placentas),
393     'MCCS': mccs,
394     'PNC': pncts
395 }
396
397 # save to a json file
398 with open(mccfile, 'w') as f:
399     json.dump(runlog, f, indent=True)

```

## listings/frangi.py

---

```
1 import numpy as np
2 import numpy.ma
3 from hfft import fft_hessian
4 from diffgeo import principal_curvatures
5 from plate_morphology import dilate_boundary
6
7 def frangi_from_image(img, sigma, beta=0.5, gamma=None, dark_bg=True,
8                      dilation_radius=None, kernel=None,
9                      signed_frangi=False, return_debug_info=False,
10                     verbose=False):
11     """Calculate the (uniscale) Frangi vesselness measure on a grayscale image
12
13     Parameters
14     -----
15     img: ndarray or ma.MaskedArray
16         a one-channel image. If this is a masked array (preferred), ignore the
17         masked regions of the image
18     sigma: float
19         Standard deviation of the gaussian, used to calculate derivatives.
20     beta: float, optional
21         The anisotropy parameter of the Frangi filter (default is 0.5)
22     gamma: float or None, optional
23         The structureness parameter of the Frangi filter. If None, gamma
24         returns half of Frobenius norm of the calculated hessian (Default is
25         None).
26     dilation_radius: int or None
27         If dilation radius is supplied, then areas within that amount of pixels
28         will not be calculated. This is preferable in certain contexts,
29         especially when there is a dark background and dark_bg=True. This is
30         especially recommended for small sigmas and when gamma is not provided.
31         None to forgo this procedure (default). A mask must be supplied for
32         this to make sense.
33     dark_bg: boolean or None
34         if True, then frangi will select only for bright curvilinear
35         features; if False, then Frangi will select only for dark
36         curvilinear structures. if None instead of a bool, then curvilinear
37         structures of either type will be reported.
38     signed_frangi: bool, optional
39         if signed is True, the result will be the same as if dark_bg is set
40         to None, except that the sign will change to match the desired
41         features. See example below.
42     return_debug_info: bool, optional
43         will return a large dict consisting of several large matrices,
44         calculated hessian, etc.
45
46     scale_dict = {'sigma': sigma,
47                   'beta': beta,
48                   'gamma': gamma,
49                   'H': hesh,
50                   'F': targets,
51                   'k1': k1,
52                   'k2': k2,
53                   'border_radius': dilation_radius
54               }
55
56     """
57     hesh = fft_hessian(img, sigma, kernel=kernel) # the triple (Hxx,Hxy,Hyy)
58     # calculate principal curvatures with |k1| <= |k2|
59
60     k1, k2 = principal_curvatures(img, sigma, H=hesh)
61
62
```

```

63 if dilation_radius is not None:
64     # pass None to just get the mask back
65     collar = dilate_boundary(None, radius=dilation_radius, mask=img.mask)
66
67     # get rid of "bad" K values before you calculate gamma
68     k1[collar] = 0
69     k2[collar] = 0
70
71 # set default gamma value if not supplied
72 if gamma is None:
73     # Frangi suggested 'half the max Hessian norm' as an empirical
74     # half the max spectral radius is easier to calculate so do that
75     # shouldn't be affected by mask data but should make sure the
76     # mask is *well* far away from perimeter
77     # we actually calculate half of max hessian norm
78     # using frob norm = sqrt(trace(AA^T))
79     # alternatively you could use gamma = .5 * np.abs(k2).max()
80     print(f'  ={sigma:.2f}')
81     gamma0 = .5*max_hessian_norm(hesh)
82     print(f'\t{gamma0:.5f} = frob-norm    pre-dilation')
83
84     gamma1 = .5*max_hessian_norm(hesh, mask=collar)
85     print(f'\t{gamma1:.5f} = frob-norm    post-collar dilation {dilation_radius}')
86     l2gamma = .5*np.max(np.abs(k2))
87     print(f'\t{l2gamma:.5f} = from L2-norm    (K2 with collar)')
88
89     hdilation = int(max(np.ceil(sigma),10))
90     hcollar = dilate_boundary(None, radius=hdilation, mask=img.mask)
91     gamma = .5 * max_hessian_norm(hesh, mask=hcollar)
92     print(f'\t{gamma:.5f} = post-hdilation (radius {hdilation}) (old   )')
93
94     print('changing      to L2-norm with collar')
95     gamma = l2gamma
96     print('-'*80)
97
98 if verbose:
99     print(f"gamma (half of max hessian (frob) norm is {gamma})")
100
101 # make a better test?
102 if np.isclose(gamma, 0):
103     print("WARNING: gamma is close to 0. should skip this layer.")
104
105 if verbose:
106     print(f'finding Frangi targets with  ={beta} and  ={gamma:.2f}')
107
108 targets = get_frangi_targets(k1, k2, beta=beta, gamma=gamma,
109                             dark_bg=dark_bg, signed=signed_frangi)
110
111 if not return_debug_info:
112     return targets
113 else:
114     scale_dict = {'sigma': sigma,
115                  'beta': beta,
116                  'gamma': gamma,
117                  'H': hesh,
118                  'F': targets,
119                  'k1': k1,
120                  'k2': k2,
121                  'border_radius': dilation_radius
122                  }
123
124 return targets, scale_dict
125
126

```

```

127 def get_frangi_targets(K1, K2, beta=0.5, gamma=None, dark_bg=True,
128                           signed=False):
129     """Calculate the Frangi vesselness measure from eigenvalues.
130
131     Parameters
132     -----
133         K1, K2 : ndarray (each)
134             each is an ndarray of eigenvalues (approximated principal
135             curvatures) for some image.
136         beta: float
137             the anisotropy parameter (default is 0.5)
138         gamma: float or None
139             the structureness parameter. if gamma is None (default), use
140             half of L2 norm of hessian (calculated from K2). if you want to
141             use half of frobenius norm, calculate it outside here.
142         dark_bg: boolean or None
143             if True, then frangi will select only for bright curvilinear
144             features; if False, then Frangi will select only for dark
145             curvilinear structures. if None instead of a bool, then curvilinear
146             structures of either type will be reported.
147         signed: boolean
148             if signed is True, the result will be the same as if dark_bg is set
149             to None, except that the sign will change to match the desired
150             features. See example below.
151
152     Returns
153     -----
154         F: ndarray, same shape as K1
155             the Frangi vesselness measure.
156
157     Examples
158     -----
159     >>>f1 = get_frangi_targets(K1,K2, dark_bg=True, signed=True)
160     >>>f2 = get_frangi_targets(K1,K2, dark_bg=False, signed=True)
161     >>>f1 == -f2
162     True
163
164     """
165
166     if gamma is None:
167         # half of max hessian norm (using L2 norm)
168         gamma = .5 * np.abs(K2).max()
169         if np.isclose(gamma, 0):
170             print("warning! gamma is very close to zero."
171                  "maybe this layer isn't worth it...")
172             print("returning an empty array")
173
174             return np.zeros_like(K1)
175
176     R = anisotropy(K1, K2, beta=beta)
177     S = structureness(K1, K2, gamma=gamma)
178
179     F = np.exp(-R)
180     F *= 1 - np.exp(-S)
181
182     # now just filter/ change sign as appropriate.
183     if not signed:
184         # calculate the regular frangi filter
185         if dark_bg is None:
186             #keep F the way it is
187             pass
188         elif dark_bg:
189             # zero responses from positive curvatures
190             F = (K2 < 0)*F
191         else:

```

```

191     # zero responses from negative curvatures
192     F = (K2 > 0)*F
193 else:
194     if dark_bg is None:
195         # output is already signed
196         pass
197     elif dark_bg:
198         # positive curvature spots will be made negative
199         F[K2 > 0] = -1 * F[K2 > 0]
200     else:
201         # negative curvature spots will be made positive
202         F[K2 < 0] = -1 * F[K2 < 0]
203
204 # reapply the mask if the inputs came with one
205 if numpy.ma.is_masked(K1):
206     F = numpy.ma.masked_array(F, mask=K1.mask)
207
208 return F
209
210 def max_hessian_norm(hesh, mask=None):
211     """Calculate max Frobenius norm of Hessian.
212
213     Calculates the maximal value (over all pixels of the image) of the
214     Frobenius norm of the Hessian.
215
216     Parameters
217     -----
218     hesh: a tuple of ndarrays
219         The tuple hxx,hxy,hyy which are all the same shape. The hessian at
220         the point (m,n) is then [[hxx[m,n], hxy[m,n]],
221         [hxy[m,n], hyy[m,n]]]
222
223     Returns
224     -----
225     float
226     """
227
228     hxx, hxy, hyy = hesh
229
230     # frob norm is just sqrt(trace(AA^T)) which is easy for a 2x2
231     hnrm = (hxx**2 + 2*hxy**2 + hyy**2)
232
233     if mask is not None:
234         hnrm[mask] = 0
235
236     hnrm = np.sqrt(hnrm)
237     return hnrm.max()
238
239
240 def anisotropy(K1,K2, beta=None):
241     """Convenience function for Anisotropy measure.
242
243     According to Frangi (1998) this is technically A**2
244     """
245
246     A = (K1/K2) **2
247
248     if beta is None:
249         return A
250     else:
251         return A / (2*beta**2)
252
253 def structureness(K1,K2, gamma=None):
254     """Convenience function for Structureness measure.

```

```

255     According to Frangi (1998) this is technically S**2
256     """
257     S = K1**2 + K2**2
258
259     if gamma is None:
260         return S
261     else:
262         return S / (2*gamma**2)

```

---

### listings/hfft\_accuracy.py

---

```

1 #!/usr/bin/env python3
2 """
3 here you want to show the accuracy of hfft.py
4 BOILERPLATE
5 show that gaussian blur of hfft is accurate, except potentially around the
6 boundary proportional to sigma.
7 or if they're off by a scaling factor, show that the derivates
8 (taken the same way) are proportional.
9 pseudocode
10
11 A = gaussian_blur(image, sigma, method='conventional')
12 B = gaussian_blur(image, sigma, method='fourier')
13
14 zero_order_accurate = isclose(A, B, tol)
15
16 J_A= get_jacobian(A)
17 J_B = get_jacobian(B)
18
19 first_order_accurate = isclose(J_A, J_B, tol)
20
21 A_eroded = zero_around_plate(A, sigma)
22 B_eroded = zero_around_plate(B, sigma)
23
24 J_A_eroded = zero_around_plate(A, sigma)
25 J_B_eroded = zero_around_plate(B, sigma)
26
27 zero_order_accurate_no_boundary = isclose(A_eroded, B_eroded, tol)
28 first_order_accurate = isclose(J_A_eroded, J_B_eroded, tol)
29
30 """
31
32 from placenta import get_named_placenta
33
34 from itertools import combinations_with_replacement
35 from skimage.exposure import rescale_intensity
36
37 from hfft import fft_hessian, fft_gaussian, fft_dgk
38 from scipy.ndimage import gaussian_filter
39 import matplotlib.pyplot as plt
40 from placenta import show_mask, list_by_quality
41
42 from scoring import mean_squared_error
43 from itertools import combinations
44 import numpy as np
45 from scipy.ndimage import laplace
46 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):
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     fixed axis is 0, who cares. it's annoying to get the other
70     """
71     # hopefully the fixed axis is 0 or 1. this gets the other one
72     it_axis = 1 if fixed_axis==0 else 0
73
74     # if it's a tuple, make it an array, etc. etc.
75     arrs = np.stack(arrs)
76
77     # make sure we can iterate over it if there's just as single image
78     if arrs.ndim < 3:
79         arrs = np.expand_dims(arrs, 0)
80
81     if labels is None:
82         labels = [None for a in arrs]
83     if formats is None:
84         formats = ['' for a in arrs]
85
86     if fixed_index is None:
87         # find halfway point of the appropriate dimension from the first array
88         fixed_index = arrs[0].shape[fixed_axis] // 2
89
90     for a, lab, fmt in zip(arrs, labels, formats):
91         plt.plot(np.arange(a.shape[it_axis]),
92                  np.moveaxis(a, fixed_axis, 0)[fixed_index, :],
93                  fmt, label=lab)
94
95     # can this be at least a little object-oriented? :(
96     plt.legend()
97
98 def multiway_comparison(arrs, scorefunc):
99
100    scores = np.zeros((len(arrs), len(arrs)))
101
102    for j in range(len(arrs)):
103        for k in range(j+1, len(arrs)):
104            scores[j, k] = scorefunc(arrs[j], arrs[k])
105
106    return scores
107
108 filename = list_by_quality(0)[5]
109
110 img = get_named_placenta(filename)
111
112 # so that scipy.ndimage.gaussian_filter doesn't use uint8 precision (jesus)
113 img = ma.masked_array(img_as_float(img), mask=img.mask)
114
115 test_sigmas = [0.12, .3, .6, 1.0, 5.0, 15, 30, 60, 90]

```

```

116
117 for sigma in test_sigmas:
118
119     print('*'*80, '\n\n', f"  ={sigma}"))
120     print('applying standard gauss blur')
121
122     # this is exactly how it's passed to skimage.feature.hessian_matrix(...)
123     A = gaussian_filter(img.filled(0), sigma, mode='constant', cval=0)
124
125     print('applying fft gauss blur')
126     #B = fft_gaussian(img, sigma)
127     B = fft_gaussian(img, sigma)
128     C = fft_dgk(img, sigma)
129
130     # figure out what is supposed to happen here
131     #B_unnormalized = B.copy()
132     #B = B / (2*(sigma**2)*np.pi)
133
134     print('calculating first derivatives')
135     # zero the masks before calculating derivates if they're masked
136     Agrad = np.gradient(A)
137     Bgrad = np.gradient(B)
138     Cgrad = np.gradient(C)
139
140
141     axes = range(img.ndim)
142
143     print('calculating second derivatives')
144     # this is the same way it's done in skimage.feature.hessian_matrix(...)
145     H_A = [np.gradient(Agrad[ax0], axis=ax1)
146            for ax0, ax1 in combinations_with_replacement(axes, 2)]
147     H_B = [np.gradient(Bgrad[ax0], axis=ax1)
148            for ax0, ax1 in combinations_with_replacement(axes, 2)]
149     H_C = [np.gradient(Cgrad[ax0], axis=ax1)
150            for ax0, ax1 in combinations_with_replacement(axes, 2)]
151
152     print('calculating eigenvalues of hessian')
153     ak1, ak2 = principal_curvatures(img, sigma=sigma, H=H_A)
154     bk1, bk2 = principal_curvatures(img, sigma=sigma, H=H_B)
155     ck1, ck2 = principal_curvatures(img, sigma=sigma, H=H_C)
156
157
158     #RA = anisotropy(ak1,ak2)
159     #RB = anisotropy(bk1,bk2)
160     #RC = anisotropy(ck1,ck2)
161
162     #SA = structureness(ak1, ak2)
163     #SB = structureness(bk1, bk2)
164     #SC = structureness(ck1, ck2)
165
166     ## ugh, apply masks here. too large to be conservative?
167     ## otherwise structureness only shows up for small sizes
168     new_mask = dilate_boundary(None, radius=int(3*sigma), mask=img.mask)
169
170     ak1 = ma.masked_array(ak1,new_mask)
171     ak2 = ma.masked_array(ak2,new_mask)
172     bk1 = ma.masked_array(bk1,new_mask)
173     bk2 = ma.masked_array(bk2,new_mask)
174     ck1 = ma.masked_array(ck1,new_mask)
175     ck2 = ma.masked_array(ck2,new_mask)
176
177     FA = get_frangi_targets(ak1,ak2, dark_bg=False).filled(0)
178     FB = get_frangi_targets(bk1,bk2, dark_bg=False).filled(0)
179     FC = get_frangi_targets(ck1,ck2, dark_bg=False).filled(0)

```

```

180
181 # even without scaling (which occurs below) the second derivates should be
182 # close. normalize matrices using frobenius norm of the hessian?
183 # note: A & B are off but have the same shape
184
185
186 # rescale to [0,255] (actually should keep as 0,1? )
187 #A_unscaled = A.copy()
188 #B_unscaled = B.copy()
189
190 #Ascaled = (A-A.min())/(A.max()-A.min())
191 #Bscaled = (B-B.min())/(B.max()-B.min())
192
193 # the following shows a random vertical slice of A & B (when scaled)
194 # the results are even more fitting when you scale B to coincide with A's max
195 # (which obviously isn't feasible in practice)
196
197
198 # FIXEDISH AFTER SCALING!
199
200 Bs = rescale_intensity(B, out_range=(0, A.max()))
201 plot_image_slices((A,B,C), labels=('scipy.ndimage,gaussian_filter',
202                               'fft_gaussian', 'fft_dgk'))
203 plt.show()
204 plot_image_slices((FA,FB,FC), labels=('scipy.ndimage,gaussian_filter',
205                               'fft_gaussian', 'fft_dgk'))
206 plt.show()
207
208 print('comparing gaussians (mean squared error)')
209 print(multiway_comparison((A,B,C), mean_squared_error))
210 print('comparing frangi response (mean squared error)')
211 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
7 import matplotlib.pyplot as plt
8 from hfft import gauss_freq, blur, fft_gaussian, fft_hessian, fft_dgk
9 from scipy.ndimage import gaussian_filter
10
11 from scipy.linalg import norm
12 import timeit
13
14 #img = camera() / 255.
15 img = imread('samples/barium1.png', as_grey=True) / 255.
16 mask = imread('samples/barium1.mask.png', as_grey=True)
17
18 # compare computation speed over sigmas
19
20 # N logarithmically spaced scales between 1 and 2^m
21 N = 5
22 m = 8
23 sigmas = np.logspace(0,m, num=N, base=2)
24
25 fft_results = list()
26 std_results = list()
27

```

```

28 for sigma in sigmas:
29     # test statements to compare (fft-based gaussian vs convolution-based)
30     fft_test_statement = 'fft_gaussian(img, {})'.format(sigma)
31     std_test_statement = 'gaussian_filter(img, {})'.format(sigma)
32     # run each statement 1 times (with 2 runs in each trial)
33     # returns/appends the average of 3 runs
34     fft_results.append(timeit.timeit(fft_test_statement,
35                                     number=1, globals=globals()))
36     std_results.append(timeit.timeit(std_test_statement,
37                                     number=1, globals=globals()))
38
39     # now actually evaluate both to compare
40     f = eval(fft_test_statement)
41     s = eval(std_test_statement)
42
43     # normalize each matrix by frobenius norm and take difference
44     # ideally should try to zero out the "mask" area
45     diff = np.abs(f / norm(f) - s / norm(s))
46     raw_diff = np.abs(f - s)
47     # don't care if it's the background
48     diff[mask==1] = 0
49     raw_diff[mask==1] = 0
50
51     # should format this stuff better into a legible table
52     print(sigma, diff.max(), raw_diff.max())
53
54 lines = plt.plot(sigmas, fft_results, 'go', sigmas, std_results, 'bo')
55 plt.xlabel('sigma (gaussian blur parameter)')
56 plt.ylabel('run time (seconds)')
57 plt.legend(lines, ('fft-gaussian', 'conv-gaussian'))
58 plt.title('Comparision of Gaussian Blur Implementations')

```

---

### listings/hfft.py

```

1 #!/usr/bin/env python3
2
3 import numpy as np
4 from scipy import signal
5 import scipy.fftpack as fftpack
6 from scipy.special import iv, ive
7 from itertools import combinations_with_replacement
8 """
9
10 hfft.py is the implementation of calculating the hessian of a real
11
12 image based in frequency space (rather than direct convolution with a gaussian
13 as is standard in scipy, for example).
14
15 TODO: PROVIDE MAIN USAGE NOTES
16 """
17
18 def gauss_freq(shape, =1.):
19     """
20         DEPRECATED
21
22     NOTE:
23         this function is/should be? for illustrative purposes only--
24         we can actually build this much faster using the builtin
25         scipy.signal.gaussian rather than a roll-your-own
26
27     build a shape=(M,N) sized gaussian kernel in frequency space
28     with size

```

```

29
30     (due to the convolution theorem for fourier transforms, the function
31     created here may simply be "multiplied" against the signal.
32
33     """
34
35     M, N = shape
36     fgauss = np.fromfunction(lambda : ((+M+1)/2)**2 + ((+N+1)/2)**2,
37                             shape=shape)
38
39     # is this used?
40     coeff = (1 / (2*np.pi * **2))
41
42     return np.exp(-fgauss / (2* **2))
43
44 def blur(img, sigma):
45     """
46     DEPRECATED
47     a roll-your-own FFT-implemented gaussian blur.
48     fft_gaussian below is preferred (it is more efficient)
49
50     I = fftpack.fft2(img) # get 2D transform of the image
51
52     # do whatever
53
54     I *= gauss_freq(I.shape, sigma)
55
56
57     return fftpack.ifft2(I).real
58
59
60 def fft_gaussian(img, sigma, A=None):
61     """
62     https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.fftconvolve.html
63     in particular the example in which a gaussian blur is implemented.
64
65     along with the comment:
66     "Gaussian blur implemented using FFT convolution. Notice the dark borders
67     around the image, due to the zero-padding beyond its boundaries. The
68     convolve2d function allows for other types of image boundaries, but is far
69     slower"
70
71     (i.e. doesn't use FFT).
72
73     note that here, you actually take the FFT of a gaussian (rather than
74     build it in frequency space). there are ~6 ways to do this.
75     """
76
77     #create a 2D gaussian kernel to take the FFT of
78
79     A = 1 / (2*np.pi*sigma**2) # scale factor for 2D
80
81     kernel = A*np.outer(signal.gaussian(img.shape[0], sigma),
82                         signal.gaussian(img.shape[1], sigma))
83
84     return signal.fftconvolve(img, kernel, mode='same')
85
86
87 def discrete_gaussian_kernel(n_samples, t):
88     """
89     t is the scale, n_samples is the number of samples to compute
90     will return a window centered a zero
91     i.e. arange(-n_samples//2, n_samples//2+1)
92

```

```

93 note! to make this work similarly to fft_gaussian, you should pass
94 sigma**2 into t here. Figure out why?
95
96 by using scipy.special.iv instead we prevent blowups
97 """
98 dom = np.arange(-n_samples//2, n_samples // 2 + 1)
99 #there should be a scaling parameter alpha but whatever
100 #return np.exp(-t) * iv(dom,t)
101 return iv(dom,t)
102
103 def fft_dgk(img, sigma, order=0, A=None):
104 """
105 A is scaling factor.
106 This is the discrete gaussian kernel which is supposedly less crappy
107 than using a sampled gaussian.
108 """
109 m,n = img.shape
110 # i don't know if this will suck if there are odd dimensions
111 kernel = np.outer(discrete_gaussian_kernel(m,sigma**2),
112 discrete_gaussian_kernel(n,sigma**2))
113
114 return signal.fftconvolve(img, kernel, mode='same')
115
116 def fft_fdgk(img, sigma):
117 """
118 convolve with discrete gaussian kernel in freq. space
119 """
120 # this would be a lot better since you wouldn't have to deal
121 # with an arbitrary cutoff of size of the discrete kernel
122 # since the freq. space version is just
123 # exp{\alpha*t (cos\theta - 1)}
124 # see formula 22 of lindeberg discrete paper
125
126 pass
127
128 def fft_hessian(image, sigma=1., kernel=None):
129 """
130 a reworking of skimage.feature.hessian_matrix that uses
131 e FFT to compute gaussian, which results in a considerable speedup
132
133 INPUT:
134     image - a 2D image (which type?)
135     sigma - coefficient for gaussian blur
136
137 OUTPUT:
138     (Lxx, Lxy, Lyy) - a triple containing three arrays
139         each of size image.shape containing the xx, xy, yy derivatives
140         respectively at each pixel. That is, for the pixel value given
141         by image[j][k] has a calculated 2x2 hessian of
142             [ [Lxx[j][k], Lxy[j][k]], ,
143               [Lxy[j][k], Lyy[j][k]] ]
144 """
145 if kernel in (None, 'discrete'):
146     #print('using discrete kernel!')
147     gaussian_filtered = fft_dgk(image, sigma=sigma)
148 else:
149     #print('using sampled gauss kernel')
150     gaussian_filtered = fft_gaussian(image, sigma=sigma)
151
152 gradients = np.gradient(gaussian_filtered)
153
154 axes = range(image.ndim)
155 H_elems = [np.gradient(gradients[ax0], axis=ax1)

```

```

157         for ax0, ax1 in combinations_with_replacement(axes, 2)]
158
159     return H_elems
160
161
162 def fft_gradient(image, sigma=1.):
163     """ returns gradient norm """
164
165     gaussian_filtered = fft_gaussian(image, sigma=sigma)
166
167     Lx, Ly = np.gradient(gaussian_filtered)
168
169     return np.sqrt(Lx**2 + Ly**2)
170
171
172 def _old_test():
173     """
174     old main function for testing.
175
176     This simply tests fft_gaussian on a test image, exemplifying the speedup
177     compared to a traditional gaussian.
178     """
179     import matplotlib.pyplot as plt
180
181     from skimage.data import camera
182
183     img = camera() / 255.
184
185     sample_sigmas = (.2, 2, 10, 30)
186
187     outputs = (fft_gaussian(img, sample_sigmas[0]),
188                fft_gaussian(img, sample_sigmas[1]),
189                fft_gaussian(img, sample_sigmas[2]),
190                fft_gaussian(img, sample_sigmas[3]),
191                )
192
193
194     fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(10, 10))
195
196     axes[0, 0].imshow(outputs[0], cmap='gray')
197     axes[0, 0].set_title('fft_gaussian ={}'.format(sample_sigmas[0]))
198     axes[0, 0].axis('off')
199
200     axes[0, 1].imshow(outputs[1], cmap='gray')
201     axes[0, 1].set_title('fft_gaussian ={}'.format(sample_sigmas[1]))
202     axes[0, 1].axis('off')
203
204     axes[1, 0].imshow(outputs[2], cmap='gray')
205     axes[1, 0].set_title('fft_gaussian ={}'.format(sample_sigmas[2]))
206     axes[1, 0].axis('off')
207
208     axes[1, 1].imshow(outputs[3], cmap='gray')
209     axes[1, 1].set_title('fft_gaussian ={}'.format(sample_sigmas[3]))
210     axes[1, 1].axis('off')
211
212     plt.tight_layout()
213     plt.show()
214
215 if __name__ == "__main__":
216     pass

```

---

## listings/merging.py

---

```
1 #!/usr/bin/env python3
2
3 import numpy as np
4 import numpy.ma as ma
5
6
7 def nz_percentile(A, q, axis=None, interpolation='linear'):
8     """calculate np.percentile(...,q) on an array's nonzero elements only
9
10    Parameters
11    -----
12    A : ndarray
13        matrix from which percentiles will be calculated. Percentiles
14        are calculated on an elementwise basis, so the shape is not important
15    q : a float
16        Percentile to compute, between 0 and 100.0 (inclusive).
17
18    (other arguments): see numpy.percentile docstring
19    ...
20
21    Returns
22    -----
23    out: float
24
25    """
26
27    if ma.is_masked(A):
28        A = A.filled(0)
29
30    return np.percentile(A[A > 0], q, axis=axis, interpolation=interpolation)
31
32
33 def apply_threshold(targets, alphas, return_labels=True):
34     """Threshold targets at each scale, then return max target over all scales.
35
36     A unique alpha can be given for each scale (see below). Return a 2D boolean
37     array, and optionally another array representing what at what scale the max
38     filter response occurred.
39
40    Parameters
41    -----
42    targets : ndarray
43        a 3D array, where targets[:, :, k] is the result of the Frangi filter
44        at the kth scale.
45    alphas : float or array_like
46        a list / 1d array of length targets.shape[-1]. each alphas[k] is a
47        float which thresholds the Frangi response at the kth scale. Due to
48        broadcasting, this can also be a single float, which will be applied
49        to each scale.
50    return_labels : bool, optional
51        If True, return another ndarray representing the scale (see Notes
52        below). Default is True.
53
54    Returns
55    -----
56    out : ndarray, dtype=bool
57        if return labels is true, this will return both the final
58        threshold and the labels as two separate matrices. This is
59        a convenience, since you could easily find labels with
60    labels : ndarray, optional, dtype=uint8
61        The scale at which the largest filter response was found after
62        thresholding. Element is 0 if no scale passed the threshold,
```

```

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

```

---

### listings/pcsvn.py

---

```

1 #!/usr/bin/env python3
2
3 from placenta import get_named_placenta
4 from diffgeo import principal_directions

```

```

5 from frangi import frangi_from_image
6 from skimage.util import img_as_float
7 import numpy as np
8 from preprocessing import inpaint_hybrid
9
10 from merging import nz_percentile
11
12 from plate_morphology import dilate_boundary
13
14 import matplotlib.pyplot as plt
15 import matplotlib as mpl
16 import numpy.ma as ma
17
18 import os.path
19 import json
20 import datetime
21
22
23 def make_multiscale(img, scales, betas, gammas, dark_bg=True,
24                     find_principal_directions=False, dilate_per_scale=True,
25                     signed_frangi=False, kernel=None, verbose=True):
26     """Returns an ordered list of dictionaries for each scale of Frangi info.
27
28     Each element in the output contains the following info:
29         {'sigma': sigma,
30          'beta': beta,
31          'gamma': gamma,
32          'H': hesh,
33          'F': targets,
34          'k1': k1,
35          'k2': k2,
36          't1': t1, # if find_principal_directions
37          't2': t2 # if find_principal_directions
38      }
39
40     is it necessary to lug all this shit around?
41     """
42
43     # store results of each scale (create as empty list)
44     multiscale = list()
45
46     img = ma.masked_array(img_as_float(img), mask=img.mask)
47
48     for i, (sigma, beta, gamma) in enumerate(zip(scales, betas, gammas)):
49
50         if dilate_per_scale:
51             if sigma > 20:
52                 radius = int(2*sigma)
53             else:
54                 radius = int(4*sigma)
55         else:
56             radius = None
57
58         if verbose:
59             print(f'  ={sigma}\t, dilation radius ={radius}')
60
61         targets, this_scale = frangi_from_image(img, sigma, beta=beta,
62                                               gamma=gamma, dark_bg=dark_bg,
63                                               dilation_radius=radius,
64                                               kernel=kernel,
65                                               signed_frangi=signed_frangi,
66                                               return_debug_info=True)
67
68         if find_principal_directions:

```

```

69     # principal directions should only be computed for critical regions
70     # this mask is where PD's will *NOT* be calculated
71     # is targets a masked array?
72     cutoff = nz_percentile(targets, 80)
73     pd_mask = np.bitwise_or(targets < cutoff, img.mask).filled(1)
74     percent_calculated = (pd_mask.size - pd_mask.sum()) / pd_mask.size
75
76     if verbose:
77         print(f"finding PD's for {percent_calculated:.2%} of image"
78               f"anything above vesselness score {cutoff:.6f}")
79     )
80     t1, t2 = principal_directions(img, sigma=sigma, H=this_scale['H'],
81                                    mask=pd_mask)
82
83     # add them to this scale's output
84     this_scale['t1'] = t1
85     this_scale['t2'] = t2
86
87 else:
88     if verbose:
89         print('skipping principal direction calculation')
90
91     # store results as a list of dictionaries
92     multiscale.append(this_scale)
93
94 return multiscale
95
96
97 def extract_pcsvn(img, filename, scales, betas=None, gammas=None, dark_bg=True,
98                   dilate_per_scale=True, verbose=True, generate_json=True,
99                   output_dir=None, kernel=None, signed_frangi=False):
100    """Run PCSVN extraction on the sample given in the file.
101
102    Despite the name, this simply returns the Frangi filter responses at
103    each provided scale without explicitly making any decisions about what
104    is or is not part of the PCSVN.
105
106    As a matter of fact, this function currently just is a wrapper for
107    make_multiscale that logs some output
108    The original main use of this function has kind of bled into
109    extract_NCS_pcsvn.py. that needs fixing. You should load the image
110    outside of this function, do post processing there, pass it inside here
111    with a dictionary of things to add to the json file
112
113    """
114
115    # Multiscale & Frangi Parameters ##########
116
117    # set default betas if undeclared
118    if betas is None:
119        betas = [0.5 for s in scales] # anisotropy constant
120
121    # declare None here to calculate half of hessian's norm
122    if gammas is None:
123        gammas = [None for s in scales] # structureness parameter
124
125    # Multiscale Frangi Filter#####
126
127    # output is a dictionary of relevant info at each scale
128    multiscale = make_multiscale(img, scales, betas, gammas,
129                                 find_principal_directions=False,
130                                 dilate_per_scale=dilate_per_scale,
131                                 kernel=kernel, signed_frangi=signed_frangi,
132                                 dark_bg=dark_bg, verbose=verbose)

```

```

133
134 # extract these for logging
135 gammas = [scale['gamma'] for scale in multiscale]
136 border_radii = [scale['border_radius'] for scale in multiscale]
137
138 # removed another bordering round. i don't think it did anything but
139 # i should have checked :/
140
141 # ignore targets too close to edge of plate
142 # wait are we doing this twice?
143 if dilate_per_scale:
144     if verbose:
145         print('trimming collars of plates (per scale)')
146
147     for i in range(len(multiscale)):
148         f = multiscale[i]['F']
149         # twice the buffer (be conservative!)
150         radius = int(multiscale[i]['sigma']*2)
151         if verbose:
152             print('dilating plate for radius={}'.format(radius))
153         f = dilate_boundary(f, radius=radius, mask=img.mask)
154         # get rid of mask
155         multiscale[i]['F'] = f.filled(0)
156     else:
157         for i in range(len(multiscale)):
158             # get rid of mask
159             multiscale[i]['F'] = multiscale[i]['F'].filled(0)
160 # Make Composite#####
161
162 # get a M x N x n_scales array of Frangi targets at each level
163 F_all = np.dstack([scale['F'] for scale in multiscale])
164
165 if generate_json:
166
167     time_of_run = datetime.datetime.now()
168     timestamp = time_of_run.strftime("%y%m%d_%H%M")
169
170     logdata = {'time': timestamp,
171                'filename': filename,
172                'betas': list(betas),
173                'gammas': gammas,
174                'sigmas': list(scales),
175                }
176
177     if dilate_per_scale:
178         logdata['border_radii'] = border_radii
179
180     if output_dir is None:
181         output_dir = 'output'
182
183     base = os.path.basename(filename)
184     *base, suffix = base.split('.')
185     dumpfile = os.path.join(output_dir,
186                            ''.join(base) + '_' + str(timestamp)
187                            + '.json')
188
189     with open(dumpfile, 'w') as f:
190         json.dump(logdata, f, indent=True)
191
192     return F_all, dumpfile
193
194
195 def get_outname_lambda(filename, output_dir=None, timestamp=None):
196     """

```

```

197     return a lambda function which can build output filenames
198     """
199
200     if output_dir is None:
201         output_dir = 'output'
202
203     base = os.path.basename(filename)
204     *base, suffix = base.split('.')
205
206     if timestring is None:
207         time_of_run = datetime.datetime.now()
208         timestring = time_of_run.strftime("%Y%m%d_%H%M")
209
210     outputstub = ''.join(base) + '_' + timestring + '_{}.' + suffix
211     return lambda s: os.path.join(output_dir, outputstub.format(s))
212
213
214     def _build_scale_colormap(N_scales, base_colormap, basecolor=(0,0,0,1)):
215         """
216             returns a mpl.colors.ListedColormap with N samples,
217             based on the colormap named "default_colormap" (a string)
218
219             the N colors are given by the default colormap, and
220             basecolor (default black) is added to map to 0.
221             (you could change this, for example, to (1,1,1,1) for white)
222
223             reversed colormaps often work better if the basecolor is black
224             you should make sure there's good contrast between the basecolor
225             and the first color in the colormap
226         """
227
228     map_range = np.linspace(0, 1, num=N_scales)
229
230     colormap = plt.get_cmap(base_colormap)
231
232     colorlist = colormap(map_range)
233
234     # add basecolor as the first entry
235     colorlist = np.vstack((basecolor, colorlist))
236
237     return mpl.colors.ListedColormap(colorlist)
238
239
240     def scale_label_figure(whereis, scales, savefilename=None,
241                           crop=None, show_only=False, image_only=False,
242                           save_colorbar_separate=False, savecolorbarfile=None,
243                           output_dir=None):
244         """
245             crop is a slice object.
246             if show_only, then just plt.show (interactive).
247             if image_only, then this will *not* be printed with the colorbar
248
249             if save_colormap_separate, then the colormap will be saved as a separate
250             file
251         """
252         if crop is not None:
253             whereis = whereis[crop]
254
255         fig, ax = plt.subplots() # not sure about figsize
256         N = len(scales) # number of scales / labels
257
258         tabemap = _build_scale_colormap(N, 'viridis_r')
259
260         if image_only:

```

```

261     plt.imsave(savefilename, wheres, cmap=tabemap, vmin=0, vmax=N)
262     plt.close()
263 else:
264     imgplot = ax.imshow(wheres, cmap=tabemap, vmin=0, vmax=N)
265     # discrete colorbar
266     cbar = plt.colorbar(imgplot)
267
268     # this is apparently hackish, beats me
269     tick_locs = (np.arange(N+1) + 0.5)*(N-1)/N
270
271     cbar.set_ticks(tick_locs)
272     # label each tick with the sigma value
273     scalelabels = [r"\sigma = {:.2f}{}".format(s) for s in scales]
274     scalelabels.insert(0, "(no match)")
275     # label with their sigma value
276     cbar.set_ticklabels(scalelabels)
277     # ax.set_title(r"Scale ($\sigma$) of maximum vesselness ")
278     plt.tight_layout()
279
280     # plt.savefig(outname('labeled'), dpi=300)
281     if show_only or (savefilename is None):
282         plt.show()
283     else:
284         plt.savefig(savefilename, dpi=300)
285
286     plt.close()
287
288 if save_colorbar_separate:
289     if savecolorbarfile is None:
290         savecolorbarfile = os.path.join(output_dir, "scale_colorbar.png")
291     fig = plt.figure(figsize=(1, 8))
292     ax1 = fig.add_axes([0.05, 0.05, 0.15, 0.9])
293     tick_locs = (np.arange(N+1) + 0.5)*(N-1)/N
294     scalelabels = [r"\sigma = {:.2f}{}".format(s) for s in scales]
295     scalelabels.insert(0, "n/a")
296     cbar = mpl.colorbar.ColorbarBase(ax1, cmap=tabemap,
297                                     norm=mpl.colors.Normalize(vmin=0,
298                                     vmax=N),
299                                     orientation='vertical',
300                                     ticks=tick_locs)
301     cbar.set_ticklabels(scalelabels)
302     plt.savefig(savecolorbarfile, dpi=300)

```

---

## listings/placenta.py

---

```

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

```

```

18 import os
19 import json
20 from scipy.ndimage import imread
21
22 from numpy.ma import is_masked
23 from skimage.color import gray2rgb
24 import matplotlib.pyplot as plt
25
26
27 def open_typefile(filename, filetype, sample_dir=None, mode=None):
28     """
29     filetype is either 'mask' or 'trace'
30     mask -> 'L' mode
31     trace -> 'RGB' mode
32     use mode keyword to override this behavior (for example if you
33     want a binary trace)
34
35     typefiles that aren't the above will be treated as 'L'
36     """
37     # try to open what the mask *should* be named
38     # this should be done less hackishly
39     # for example, if filename is 'ncs.1029.jpg' then
40     # this would set the maskfile as 'ncs.1029.mask.jpg'
41
42     #if filetype not in ("mask", "trace"):
43     #    raise NotImplementedError("Can only deal with mask or trace files.")
44
45     # get the base of filename and build the type filename
46     *base, suffix = filename.split('.')
47     base = ''.join(base)
48     typefile = '.'.join((base, filetype, suffix))
49
50     if sample_dir is None:
51         sample_dir = 'samples'
52
53     typefile = os.path.join(sample_dir, typefile)
54
55     if mode is not None:
56         if filetype == 'mask':
57             mode = 'L'
58         elif filetype in ('ctrace', 'veins', 'arteries'):
59             mode = 'RGB'
60         else:
61             # handle this if you need to?
62             mode = 'L'
63     try:
64         img = imread(typefile, mode=mode)
65
66     except FileNotFoundError:
67         print('Could not find file', typefile)
68         raise
69
70     return img
71
72
73 def open_tracefile(base_filename, as_binary=True,
74                     sample_dir=None):
75     """
76
77     ###width parsing is no longer done here. instead, this function
78     should handle the venous/arterial difference.
79
80     this currently only serves to open the RGB traces as binary

```

```

82     files instead of RGB, which is processed later
83
84     #TODO: expand this later to handle arterial traces and venous traces
85     INPUT:
86         base_filename: the name of the base file, not the tracefile itself
87         as_binary: if True
88     """
89
90     if as_binary:
91         mode = 'L'
92     else:
93         mode = 'RGB'
94
95     T = open_typefile(base_filename, 'trace', sample_dir=sample_dir, mode=mode)
96
97     if as_binary:
98
99         return np.invert(T != 0)
100
101    else:
102        return T
103
104
105 def get_named_placenta(filename, sample_dir=None, masked=True,
106                         maskfile=None):
107     """
108     This function is to be replaced by a more ingenious/natural
109     way of accessing a database of unregistered and/or registered
110     placental samples.
111
112     Parameters
113     -----
114
115     filename: name of file (including suffix?) but NOT directory
116     masked: return it masked.
117     maskfile: if supplied, this use the file will use a supplied 1-channel
118         mask (where 1 represents an invalid/masked pixel, and 0
119         represents a valid/unmasked pixel. the supplied image must be
120         the same shape as the image. if not provided, the mask is
121         calculated (unless masked=False)
122         the file must be located within the sample directory
123
124     If maskfile is 'None' then this function will look for
125     a default maskname with the following pattern:
126
127         test.jpg -> test.mask.jpg
128         ncs.1029.jpg -> ncs.1029.mask.jpg
129
130     sample_directory: Relative path where sample (and mask file) is located.
131         defaults to './samples'
132
133     if masked is true (default), this returns a masked array.
134
135     NOTE: A previous logical incongruity has been corrected. Masks should have
136     1 as the invalid/background/mask value (to mask), and 0 as the
137     valid/plate/foreground value (to not mask)
138     """
139     if sample_dir is None:
140         sample_dir = 'samples'
141
142     full_filename = os.path.join(sample_dir, filename)
143
144     raw_img = imread(full_filename, mode='L')
145

```

```

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

```

```

210     json_file = 'sample-qualities.json'
211
212     try:
213         with open(json_file, 'r') as f:
214             D = json.load(f)
215     except FileNotFoundError:
216         if return_empty:
217             return list()
218         else:
219             print('cannot find', json_file)
220             raise FileNotFoundError
221
222     if json_file == 'sample-qualities.json':
223         # go one level deep
224         placentas = [k for k in D[quality].keys()]
225     else:
226         placentas = [k for k in D.keys()]
227
228     if N is not None:
229         return placentas[:N]
230     else:
231         return placentas
232
233
234 def check_filetype(filename, assert_png=True, assert_standard=False):
235     """
236     'T-BN8333878.raw.png' returns 'raw'
237     'T-BN8333878.mask.png' returns 'mask'
238     'T-BN8333878.png' returns 'base'
239
240     if assert_png is True, then raise assertion error if the file
241     is not of type png
242
243     if assert_standard, then assert the filetype is
244     mask, base, trace, or raw.
245
246     etc.
247     """
248     basename, ext = os.path.splitext(filename)
249
250     if ext != '.png':
251         if assert_png:
252             assert ext == '.png'
253
254     sample_name, typestub = os.path.splitext(basename)
255
256     if typestub == '':
257         # it's just something like 'T-BN8333878.png'
258         return 'base'
259     elif typestub in ('.mask', '.trace', '.raw', '.ctrace', '.arteries', '.veins', '.ucip'):
260         # return 'mask' or 'trace' or 'raw'
261         return typestub.strip('.')
262     else:
263         print('unknown filetype:', typestub)
264         print('is it a weird filename?')
265
266         print('warning: lookup failed, unknown filetype:' + typestub)
267
268     return typestub
269
270 def list_placentas(label=None, sample_dir=None):
271     """
272     label is the specifier, basically just '.startswith()'
273

```

```

274     only real use is to find all the T-BN* files
275
276     this is hackish, if you ever decide to use a file other than
277     png then this needs to change
278     """
279
280     if sample_dir is None:
281         sample_dir = 'samples'
282
283     if label is None:
284         label = '' # str.startswith('') is always True
285
286     placentas = list()
287
288     for f in os.listdir(sample_dir):
289
290         if f.startswith(label):
291             # oh man they gotta be png files
292             if check_filetype(f) == 'base':
293                 placentas.append(f)
294
295     return sorted(placentas)
296
297
298 def show_mask(img, mask=None, interactive=False, mask_color=None):
299     """
300         rename this color_mask since showing the mask is just a secondary feature
301         show a masked grayscale image with a dark blue masked region
302
303         custom version of imshow that shows grayscale images with the right
304         colormap and, if they're masked arrays, sets makes the mask a dark blue) a
305         better function might make the grayscale value dark blue (so there's no
306         confusion)
307
308         if interactive, this operates like "plt.imshow"
309         if interactive==False, return the RGB matrix
310
311         if mask provided, add it to the image. (pass img.data instead if you don't
312         want to use the original mask)
313     """
314
315     if mask_color is None:
316         mask_color = (0, 0, 60)
317
318     # if there's no mask at all
319     if (mask is None) and (not is_masked(img)):
320         if interactive:
321             plt.imshow(img, cmap=plt.cm.gray)
322             return # we're done
323         else:
324             # return as an rgb image so output is uniform
325             return gray2rgb(img)
326
327     elif not is_masked(img):
328         # add mask to the image / add to existing mask
329         # if i just rewrite img will it change outside this function?
330         new_img = ma.masked_array(img, mask=mask)
331     else:
332         new_img = img.copy()
333
334     # otherwise, get an RGB array, black where the mask is
335     mimg = gray2rgb(new_img.filled(0))
336
337     # fill masked regions with the mask color

```

```

338     mimg[new_img.mask, :] = mask_color
339
340     if interactive:
341         plt.imshow(mimg)
342     else:
343         return mimg
344
345
346 def _cropped_bounds(img, mask=None):
347
348     if mask is not None:
349
350         img = ma.masked_array(img, mask=mask)
351
352     X, Y = (np.argwhere(np.invert(img.mask)).any(axis=k)).squeeze()
353         for k in (0, 1)
354             )
355
356     if X.size == 0:
357         X = [None, None] # these will slice correctly
358     if Y.size == 0:
359         Y = [None, None]
360
361     return Y[0], Y[-1], X[0], X[-1]
362
363
364 def cropped_args(img, mask=None):
365     """
366     get a slice that would crop image
367     i.e. img[cropped_args(img)] would be a cropped view
368     """
369
370     x0, x1, y0, y1 = _cropped_bounds(img, mask=None)
371
372     return np.s_[x0:x1, y0:y1]
373
374
375 def cropped_view(img, mask=None):
376     """
377     removes entire masked rows and columns from the borders of a masked array.
378     will return a masked array of smaller size
379
380     don't ask me about data
381
382     the name sucks too
383     """
384
385     # find first and last row with content
386     x0, x1, y0, y1 = _cropped_bounds(img, mask=mask)
387
388     return img[x0:x1, y0:y1]
389
390
391 CYAN = [0, 255, 255]
392 YELLOW = [255, 255, 0]
393
394
395 def measure_ncs_markings(ucip_img=None, filename=None, verbose=True):
396     """
397     find location of ucip and resolution of image based on input
398     (similar to perimeter layer in original NCS data set
399
400     Parameters
401     -----

```

```

402
403     ucip_img: an RGB ndarray or None
404         The perimeter layer of an NCS sample (colorations according to the
405             tracing protocol). if None, filename must be included. Default is None.
406     filename:
407         the filename of the SAMPLE (not the ucip image file itself)
408
409     Returns
410     -----
411     m : tuple of ints
412         the coordinates of (the center of) of the umbilical cord point
413             (depicted as a yellow dot) in the original image.
414     resolution: a float
415         measured distance between the two cyan dots
416     """
417
418     if ucip_img is None:
419         ucip_img = open_typefile(filename, 'ucip')
420
421     # just in case it's got an alpha channel, remove it
422     img = ucip_img[:, :, 0:3]
423
424     # given the image img (make sure no alpha channel)
425     # find all cyan pixels (there are two boxes of 3 pixels each and we
426     # just want to extract the middle of each
427     if verbose:
428         print('the image size is {}x{}'.format(img.shape[0], img.shape[1]))
429
430     rulemarks = np.all(img == CYAN, axis=-1)
431
432     # turn into two pixels (these should each be shape (18,))
433     X, Y = np.where(rulemarks)
434
435     assert X.shape == Y.shape
436
437     # if they followed the protocol correctly...
438     if X.size == 18:
439         # get the two pixels at the center of each box
440         A, B = (X[4], Y[4]), (X[13], Y[13])
441     else:
442         # dots are a nonstandard size for some reason. this works too.
443         thinned = morphology.thin(rulemarks)
444         X, Y = np.where(thinned)
445         assert(thinned.sum() == 2) # there should be just two pixels now.
446         A, B = (X[0], Y[0]), (X[1], Y[1])
447
448     ruler_distance = np.sqrt((A[0] - B[0])**2 + (A[1] - B[1])**2)
449     if verbose:
450         print(f'one cm equals {ruler_distance} pixels')
451
452     # the umbilical cord insertion point (UCIP) is a yellow circle, radius 19
453     ucipmarks = np.all(img == YELLOW, axis=-1)
454     X, Y = np.where(ucipmarks)
455
456     # find midpoint of the x & y coordinates
457     assert X.max() - X.min() == Y.max() - Y.min()
458     radius = (X.max() - X.min()) // 2
459
460     mid = (X.min() + radius, Y.min() + radius)
461
462     if verbose:
463         print('the middle of the UCIP location is', mid)
464         print('the radius outward is', radius)
465         print('the total measurable diameter is', radius**2 + 1)

```

```

466     return mid, ruler_distance
467
468
469
470 def add_ucip_to_mask(m, radius=100, mask=None, size_like=None):
471     """
472     - m is a tuple (2x1) representing the (coordinate) midpoint of the UCIP
473     - radius around which to dilate the UCIP is the dilation radius as it
474     works in morphology--this is passed directly to skimage.morphology.disk.
475     thus a circle centered at point m with diameter 2*radius + 1
476     - if no mask is supplied, dilate the point in an array of zeros the shape
477     of 'size_like' (would be the same as passing mask=np.zeros_like(size_like))
478
479     Note: this behaves much faster than binary dilation on the point
480     """
481     if mask is None:
482         if size_like is not None:
483             mask = np.zeros_like(size_like)
484         else:
485             raise ValueError("No mask info supplied!")
486
487     # an empty mask (since we need to merge--we don't want to copy the
488     # zeros of the dilated UCIP -- just the ones!)
489     to_add = np.zeros_like(mask)
490
491     # this is way faster than dilating the point in the matrix,
492     # just set this at the centered point
493
494     # doesn't check for out of bounds stuff. use at your own peril
495     D = morphology.disk(radius)
496     to_add[m[0]-radius:m[0]+radius+1, m[1]-radius:m[1]+radius+1] = D
497
498     # merge with supplied mask
499     return np.logical_or(mask, to_add)
500
501
502 if __name__ == "__main__":
503     """test that this works on an easy image."""
504     test_filename = 'barium1.png'
505
506     img = get_named_placenta(test_filename, maskfile=None)
507
508     print('showing the mask of', test_filename)
509     print('run plt.show() to see masked output')
510
511     show_mask(img, interactive=True)

```

---

### listings/plate\_morphology.py

---

```

1 #!/usr/bin/env python3
2
3 from skimage.morphology import disk, binary_erosion, binary_dilation
4 from skimage.morphology import convex_hull_image
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 mask_cuts(img, ucip, mask_only=False, in_place=False, return_success=False):
63     """
64
65     this doesn't handle any image, io. just provide the ucip img and the
66     base (masked) image and we'll fix the mask
67
68     ucip is the actual RGB array, not the file. do io elsewhere.
69
70     if mask_only, this will simply return the new mask as a 2D boolean array.
71     Otherwise, it returns a masked_array.
72     The cut region will be added to the img's mask. If you really want just the
73     difference, you'll have to run
74     >>>np.logical_and(cut_mask, np.invert(img.mask)) yourself.
75

```

```

76 yourself.
77
78 If in_place, this changes the mask of the image directly (but still returns
79 a masked array. If mask_only is True, in_place will automatically be set to
80 False to prevent hideous side effects
81
82 if return_success, this function returns True if there was a cutmark found,
83 otherwise false as a second output
84 """
85 # get indices where the blue square indicating center of a cut appears
86 cutmarks = np.all(ucip==(0,0,255), axis=-1)
87
88 if not np.any(cutmarks):
89
90     #print("no cutmarks found on image")
91
92     if return_success:
93         return img, False
94     else:
95         return img
96
97 else:
98     #print("found a cutmark!")
99     pass
100
101 cutmarks = np.nonzero(cutmarks)
102 # get the first pixel of it (we don't need to be too precise here)
103 X, Y = cutmarks[0][0], cutmarks[1][0]
104
105 # get a value somewhat lower than the value of bg in the cut
106 # (this should be a high number before we take 85%)
107 # sometimes this is in a shadowy region which fucks everything up though
108 threshold = max(img[cutmarks].mean() * .85, 175)
109
110 rmin, rmax = max(0, X-100), min(img.shape[0], X+100)
111 cmin, cmax = max(0, Y-100), min(img.shape[1], Y+100)
112 cutregion = np.s_[rmin:rmax, cmin:cmax] # get a window around the mark
113
114 # mark inside of the placenta with label 2, original mask and cutmarks with
115 # label 1, and the rest with 0 (i dunno)
116 markers = np.zeros(img.shape, dtype='int32')
117 markers[img.filled(255) < threshold] = 2
118 markers[img.mask] = 1
119 markers[cutmarks] = 1
120
121 # perform watershedding on the thresholded image to fill in the cut with
122 # label 1
123 cutfix = watershed(img.filled(255) < threshold, markers=markers)
124
125 # this is a waste considering the in_place, but eh
126 new_mask = img.mask.copy()
127
128 new_mask[cutregion] = (cutfix[cutregion] == 1)
129
130 if mask_only:
131
132     out = new_mask
133
134 elif not in_place:
135
136     out = ma.masked_array(img, mask=new_mask)
137
138 else:
139
140     # will this work?

```

```

140         img[new_mask] = ma.masked
141
142     out = img
143
144     # now return succeed if asked to
145     if return_success:
146         return out, True
147
148     else:
149         return out
150
151
152 if __name__ == "__main__":
153
154     # DEMO FOR SHOWING OFF DILATE_BOUNDARY EFFECT
155
156     from placenta import get_named_placenta
157     from frangi import frangi_from_image
158     import matplotlib.pyplot as plt
159
160     import os.path
161
162     dest_dir = 'demo_output'
163     img = get_named_placenta('T-BN0164923.png')
164
165     sigma = 3
166     radius = 25
167
168     inset = np.s_[800:1000, 500:890]
169
170     D = dilate_boundary(img, radius=radius)
171
172     Fimg = frangi_from_image(img, sigma, dark_bg=False, dilation_radius=None)
173     FD = frangi_from_image(D, sigma, dark_bg=False)
174     FDinv = frangi_from_image(D, sigma, dark_bg=True)
175     Finv = frangi_from_image(img, sigma, dark_bg=True, dilation_radius=None)
176
177     fig, axes = plt.subplots(ncols=2, nrows=3)
178
179     axes[0,0].imshow(img[inset].filled(0), cmap=plt.cm.gray)
180     axes[0,1].imshow(D[inset].filled(0), cmap=plt.cm.gray)
181     axes[1,0].imshow(Fimg[inset].filled(0), cmap=plt.cm.nipy_spectral)
182     axes[1,1].imshow(FD[inset].filled(0), cmap=plt.cm.nipy_spectral)
183     axes[2,0].imshow(Finv[inset].filled(0), cmap=plt.cm.nipy_spectral)
184     axes[2,1].imshow(FDinv[inset].filled(0), cmap=plt.cm.nipy_spectral)
185
186     for a in axes.ravel():
187         # get rid of all the labels
188         plt.setp(a.get_xticklabels(), visible=False)
189         plt.setp(a.get_yticklabels(), visible=False)
190
191     # lol matlab
192     for i in range(5):
193         fig.tight_layout()
194
195     plt.savefig(os.path.join(dest_dir, "boundary_dilation_demo.png"), dpi=300)

```

---

### listings/preprocessing.py

---

```

1 #!/usr/bin/env python3
2
3 # TODO: refactor this so inpaint_glare is the main function that takes

```

```

4 #           a keyword argument strategy='hybrid' or whatever then you can run
5 #           >>>for s in ['mean_window', 'median_boundary', 'biharmonic', 'hybrid']:
6 #               timeit.timeit('inpaint_glare(img, strategy=s)', globals=globals())
7 #
8 #           ... but it's annoying since you'll need a way to pass args to the
9 #           particular strategy
10
11 from skimage.morphology import binary_dilation, disk, remove_small_objects
12 from skimage.restoration import inpaint_biharmonic
13 import numpy as np
14 import numpy.ma as ma
15 from scipy.ndimage import label
16 from skimage.util import img_as_float
17 from skimage.segmentation import find_boundaries
18 from plate_morphology import dilate_boundary
19
20
21 def inpaint_glare(img, threshold=175, window_size=15, mask=None):
22     """
23         img is a masked array type uint [0,255]
24     """
25
26     # bool array, true where glare
27     if mask is None:
28         glared = mask_glare(img, threshold=threshold, mask_only=True)
29     else:
30         glared = mask
31
32     B = ma.masked_array(img, mask=glared) # masked background *and* glare
33     new_img = img.copy() # copy values of original image (will rewrite)
34     d = int(window_size)
35
36     for j, k in zip(*np.where(glared)):
37         # rewrite all glared pixels with the mean of nonmasked elements
38         # in a window_size window. (this doesn't check OoB, be careful!)
39         new_img[j, k] = B[j-d:j+d, k-d:k+d].compressed().mean()
40
41     return new_img
42
43
44 def inpaint_with_boundary_median(img, threshold=175, mask=None):
45     """
46         mask glare pixels, then replace by the median value on the mask's boundary
47     """
48     if mask is None:
49         glared = mask_glare(img, threshold=threshold, mask_only=True)
50     else:
51         glared = mask
52
53     B = ma.masked_array(img, mask=glared)
54
55     new_img = img.copy() # copy values of original image (will rewrite)
56     bounds = find_boundaries(glared)
57     lb, _ = label(bounds)
58     fill_vals = np.zeros_like(img.data)
59
60     # for each boundary of masked region, find the median value of the img
61     for lab in range(1, lb.max()+1):
62         inds = np.where(lb == lab)
63         fill_vals[inds] = nz_median(B[inds])
64
65     # label masked regions together with their boundaries (they'll be
66     # connected)
67     lm, _ = label(np.logical_or(glared, lb != 0))

```

```

68
69 # fill the masked areas with the corresponding fill value
70 for lab in range(1, lm.max()+1):
71     inds = np.where(lm == lab)
72     # find locations of filled values corresponding to this label
73     # median in case there's overlapped regions? (sloppy)
74     replace_value = nz_median(fill_vals[inds])
75
76     if replace_value == 0:
77         raise
78
79     fill_vals[inds] = replace_value
80
81 # now fill in the values
82 new_img[glared] = fill_vals[glared]
83
84 return new_img
85
86 def nz_median(A):
87
88     if ma.is_masked(A):
89         relevant = A[A > 0].compressed()
90     else:
91         relevant = A[A > 0]
92
93     return np.median(relevant)
94
95
96 def inpaint_hybrid(img, threshold=175, min_size=64, boundary_radius=10):
97 """
98     use biharmonic inpainting in larger, inner areas (important stuff)
99     and median inpainting in smaller areas and along boundary
100 """
101
102     glare = mask_glare(img, threshold=threshold, mask_only=True)
103
104     glare_inside = dilate_boundary(glare, mask=img.mask,
105                                     radius=boundary_radius).filled(0)
106
107     large_glare = remove_small_objects(glare_inside, min_size=min_size,
108                                         connectivity=2)
109     small_glare = np.logical_and(glare, np.invert(large_glare))
110
111     # inpaint smaller and less important values with less expensive method
112     inpainted = inpaint_with_boundary_median(img, mask=small_glare)
113     hybrid = img_as_float(inpainted) # scale 0 to 1
114
115     # inpaint larger regions with biharmonic inpainting
116     large_inpainted = inpaint_biharmonic(img.filled(0), mask=large_glare)
117
118     # now overwrite with these values
119     hybrid[large_glare] = large_inpainted[large_glare]
120
121     # put on old image mask
122     return ma.masked_array(hybrid, mask=img.mask)
123
124 def inpaint_with_biharmonic(img, threshold=175):
125 """
126     use biharmonic inpainting *all* glare
127 """
128     glare = mask_glare(img, threshold=threshold, mask_only=True)
129     inpainted = inpaint_biharmonic(img_as_float(img.filled(0)), mask=glare)
130
131     if ma.is_masked(img):

```

```

132         return ma.masked_array(inpainted, mask=img.mask)
133     else:
134         return inpainted
135
136 def mask_glare(img, threshold=175, mask_only=False):
137     """
138     for demoing purposes, with placenta.show_mask
139
140     if mask_only, just return the mask. Otherwise return a copy of img with
141     that added to the mask. If you want the original mask to be ignored,
142     just pass img.filled(0) ya doofus
143
144     threshold is expected to be of the same dtype as img *unless# it assumes
145     its default value, in which case the threshold will be converted to a float
146
147     """
148     # if img.dtype is floating but threshold value is still the default
149     # this could be generalized
150     if np.issubdtype(img.dtype, np.floating) and (threshold == 175):
151         threshold = 175 / 255
152     # region to inpaint
153     inp = (img > threshold)
154
155     # get a larger area around the specks
156     inp = binary_dilation(inp, selem=disk(2))
157
158     # remove anything large
159     #inp = white_tophat(inp, selem=disk(3))
160
161     if mask_only:
162         return inp
163     else:
164         # both the original background *and* these new glared regions
165         # are masked
166         return ma.masked_array(img, mask=inp)
167
168
169 DARK_RED = np.array([103, 15, 23]) / 255.
170
171 # test it on a particularly bad sample
172 if __name__ == "__main__":
173
174     from placenta import get_named_placenta, show_mask
175     import matplotlib.pyplot as plt
176
177     filename = 'T-BN0204423.png' # a particularly glary sample
178     img = get_named_placenta(filename)
179
180     img = ma.masked_array(img_as_float(img), mask=img.mask)
181     crop = np.s_[150:500, 150:800] # indices to zoom in on the region
182     zoom = np.s_[300:380, 300:380] # even smaller region
183
184     inset = zoom # which view to use
185
186     masked = mask_glare(img) # for viewing
187     inpainted = inpaint_glare(img)
188     minpainted = inpaint_with_boundary_median(img)
189     hinpainted = inpaint_hybrid(img)
190     binpainted = inpaint_with_biharmonic(img)
191
192     # view the closeup like this
193     minpainted_view = show_mask(minpainted, interactive=False,
194                                 mask_color=DARK_RED)
195     inpainted_view = show_mask(inpainted, interactive=False,

```

```

196                         mask_color=DARK_RED)
197 masked_view = show_mask(masked, interactive=False,
198                         mask_color=DARK_RED)
199 img_view = show_mask(img, interactive=False,
200                         mask_color=DARK_RED)
201 inpainted_view = show_mask(inpainted, interactive=False,
202                         mask_color=DARK_RED)
203 minpainted_view = show_mask(minpainted, interactive=False,
204                         mask_color=DARK_RED)
205
206 # view them all next to each other
207
208 fig, axes = plt.subplots(ncols=3, nrows=2)
209
210 axes[0,0].imshow(img_view[inset])
211 axes[0,1].imshow(masked_view[inset])
212 axes[0,2].imshow(inpainted_view[inset])
213 axes[1,0].imshow(minpainted_view[inset])
214 axes[1,1].imshow(binpainted_view[inset])
215 axes[1,2].imshow(hinpainted_view[inset])
216
217 for a in axes.ravel():
218     # get rid of all the labels
219     plt.setp(a.get_xticklabels(), visible=False)
220     plt.setp(a.get_yticklabels(), visible=False)
221
222 # lol matlab
223 for i in range(5):
224     fig.tight_layout()
225
226 IMGS = np.vstack((
227     np.hstack((img_view, masked_view, inpainted_view)),
228     np.hstack((minpainted_view, binpainted_view, hinpainted_view))))
229
230 # THEN IMSAVE
231
232 # plt.imsave('preprocessing_comparison_cropped.png', IMGS)
233 # plt.imsave('preprocessing_comparison_zoomed.png', IMGS)
234
235 # if it's zoomed, then rescale the output in GIMP to 4x

```

---

### listings/process\_NCS\_xcfs.py

---

```

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

```

```

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

```

```

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

```

```

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

```

---

### listings/score.py

---

```

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

```

```

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

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

---

### listings/scoring.py

---

```

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

```

```

29 Notes:
30
31     The correspondence is as follows:
32     3 pixels: "#ff006f", # magenta
33     5 pixels: "#a80000", # dark red
34     7 pixels: "#a800ff", # purple
35     9 pixels: "#ff00ff", # light pink
36     11 pixels: "#008aff", # blue
37     13 pixels: "#8aff00", # green
38     15 pixels: "#ffc800", # dark yellow
39     17 pixels: "#ff8a00", # orange
40     19 pixels: "#ff0015" # bright red
41
42 According to the original tracing protocol, the traced vessels are
43 binned into these 9 sizes. Vessels with a diameter smaller than 3px
44 are not traced (unless they're binned into 3px).
45
46 Note: this does *not* deal with collisions. If you pass anything
47 with addition (blended colors) as the ctraces are, you will have
48 trouble, as those will not be registered as any of the colors above
49 and will thus be ignored. If you want to handle data from both
50 arterial *and* venous layers, you should do so outside of this
51 function.
52 """
53
54 # a 2D picture to fix in with the pixel widths
55 W = np.zeros_like(T[:, :, 0])
56
57 for pix, color in TRACE_COLORS.items():
58
59     #ignore pixelwidths outside the specified range
60     # get the 2D indices that are that color
61     idx = np.where(np.all(T == color, axis=-1))
62     W[idx] = pix
63
64
65 return W
66
67 def merge_widths_from_traces(A_trace, V_trace, strategy='minimum'):
68 """
69     combine the widths from two RGB-traces A_trace and V_trace
70     and return one width matrix according to 'strategy'
71
72 Parameters
73 -----
74 A_trace: ndarray
75     an MxNx3 matrix, where each pixel (along the
76     last dimension) is an RGB triplet (i.e. each entry
77     is an integer between [0,256]. The colors each
78     correspond to those in TRACE_COLORS, and (255,255,255)
79     signifies "no vessel". This will normally correspond to
80     the sample's arterial trace.
81 V_trace: ndarray
82     an MxNx3 matrix the same shape and other
83     requirements as A_trace (see above). This will normally
84     correspond to the sample's venous trace.
85 strategy: keyword string
86     when A_trace and V_trace coincide at some entry,
87     this is the merging strategy. It should be a keyword
88     of one of the following choices:
89
90     "minimum": take the minimum width of the two traces
91         (default). this is the sensible option if you
92         are filtering out larger widths.

```

```

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

```

```

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

```

```

221     rewritten already.
222 """
223     triple = hexstring.strip("#")
224     return tuple(int(x, 16) for x in (triple[:2], triple[2:4], triple[4:]))
225
226
227 def skeletonize_trace(T, T2=None):
228     """
229     if T is a boolean matrix representing a trace, then thin it
230
231     if T is an RGB trace, then register it according to the
232     tracing protocol then thin it
233
234     if T2 is provided, do the same thing to T2 and then merge the two
235     """
236     if T.ndim == 3:
237         trace = (rgb_to_widths(T) > 0) # booleanize it
238
239     thinned = thin(trace)
240
241     if T2 is None:
242         return thinned
243
244     else:
245         # do the same thing to second trace and merge it
246         if T2.ndim == 3:
247             trace_2 = (rgb_to_widths(T2) > 0) # booleanize it
248             thinned_2 = thin(trace_2)
249
250         return np.logical_or(thinned, thinned_2)
251
252
253 def confusion(test, truth, bg_mask=None, colordict=None, tint_mask=True):
254     """
255     distinct coloration of false positives and negatives.
256
257     colors output matrix with
258         true_pos if test[-] == truth[-] == 1
259         true_neg if test[-] == truth[-] == 0
260         false_neg if test[-] == 0 and truth[-] == 1
261         false_pos if test[-] == 1 and truth[-] == 0
262
263     if colordict is supplied: you supply a dictionary of how to
264     color the four cases. Spec given by the default below:
265
266     if tint mask, then the mask is overlaid on the image, not replacing totally
267     colordict = {
268         'TN': (247, 247, 247), # true negative
269         'TP': (0, 0, 0) # true positive
270         'FN': (241, 163, 64), # false negative
271         'FP': (153, 142, 195), # false positive
272         'mask': (247, 200, 200) # mask color (not used in MCC calculation)
273     }
274     """
275
276     if colordict is None:
277         colordict = {
278             'TN': (247, 247, 247), # true negative# 'f7f7f7'
279             'TP': (0, 0, 0), # true positive # '000000'
280             'FN': (241, 163, 64), # false negative # 'f1a340' orange
281             'FP': (153, 142, 195), # false positive # '998ec4' purple
282             'mask': (247, 200, 200) # mask color (not used in MCC calculation)
283         }
284

```

```

285     colordict = {
286         'TN': (49,49,49), # true negative# 'f7f7f7'
287         'TP': (0, 0, 0), # true positive # '000000'
288         'FN': (201,53,108), # false negative # 'f1a340' orange
289         'FP': (0,112,163), # false positive # '998ec4' purple
290         'mask': (247, 200, 200) # mask color (not used in MCC calculation)
291     }
292 #TODO: else check if mask is specified and add it as color of TN otherwise
293
294 true_neg_color = np.array(colordict['TN'], dtype='f')/255
295 true_pos_color = np.array(colordict['TP'], dtype='f')/255
296 false_neg_color = np.array(colordict['FN'], dtype='f') /255
297 false_pos_color = np.array(colordict['FP'], dtype='f')/255
298 mask_color = np.array(colordict['mask'], dtype='f') /255
299
300 assert test.shape == truth.shape
301
302 # convert to bool
303 test, truth = test.astype('bool'), truth.astype('bool')
304
305 # RGB array size of test and truth for output
306 output = np.zeros((test.shape[0], test.shape[1], 3), dtype='f')
307
308 # truth conditions
309 true_pos = np.bitwise_and(test==truth, truth)
310 true_neg = np.bitwise_and(test==truth, np.invert(truth))
311 false_neg = np.bitwise_and(truth, np.invert(test))
312 false_pos = np.bitwise_and(test, np.invert(truth))
313
314 output[true_pos,:,:] = true_pos_color
315 output[true_neg,:,:] = true_neg_color
316 output[false_pos,:,:] = false_pos_color
317 output[false_neg,:,:] = false_neg_color
318
319 # try to find a mask
320 if bg_mask is None:
321     try:
322         bg_mask = test.mask
323     except AttributeError:
324         # no mask is specified, we're done.
325         return output
326
327 # color the mask
328 if tint_mask:
329     output[bg_mask,:,:] += mask_color
330     output[bg_mask,:,:] /= 2
331 else:
332     output[bg_mask,:,:] = mask_color
333
334 return output
335
336
337 def compare_trace(approx, trace=None, filename=None,
338                   sample_dir=None, colordict=None):
339     """
340     compare approx matrix to trace matrix and output a confusion matrix.
341     if trace is not supplied, open the image from the tracefile.
342     if tracefile is not supplied, filename must be supplied, and
343     tracefile will be opened according to the standard pattern
344
345     colordict are parameters to pass to confusion()
346
347     returns a matrix
348     """

```

```

349
350     # load the tracefile if not supplied
351     if trace is None:
352         if filename is not None:
353             try:
354                 trace = open_typefile(filename, 'trace')
355             except FileNotFoundError:
356                 print("No trace file found matching ", filename)
357                 print("no trace found. generating dummy trace.")
358                 trace = np.zeros_like(approx)
359             else:
360                 print("no trace supplied/found. generating dummy trace.")
361                 trace = np.zeros_like(approx)
362
363     C = confusion(approx, trace, colordict=colordict)
364
365     return C
366
367
368 def mcc(test, truth, bg_mask=None, score_bg=False, return_counts=False):
369     """
370         Matthews correlation coefficient
371         returns a float between -1 and 1
372         -1 is total disagreement between test & truth
373         0 is "no better than random guessing"
374         1 is perfect prediction
375
376         bg_mask is a mask of pixels to ignore from the statistics
377         for example, things outside the placental plate will be counted
378         as "TRUE NEGATIVES" when there wasn't any chance of them not being
379         scored as negative. therefore, it's not really a measure of the
380         test's accuracy, but instead artificially pads the score higher.
381
382         setting bg_mask to None when test and truth are not masked
383         arrays should give you this artificially inflated score.
384         Passing score_bg=True makes this decision explicit, i.e.
385         any masks (even if supplied) will be ignored, and your count of
386         false positives will be inflated.
387
388     """
389     true_pos = np.logical_and(test==truth, truth)
390     true_neg = np.logical_and(test==truth, np.invert(truth))
391     false_neg = np.logical_and(truth, np.invert(test))
392     false_pos = np.logical_and(test, np.invert(truth))
393
394     if score_bg:
395         # take the classifications above as they are (nothing is masked)
396         pass
397     else:
398         # if no specified mask, check the test array itself?
399         if bg_mask is None:
400             try:
401                 bg_mask = test.mask
402             except AttributeError:
403                 # no mask is specified, we're done.
404                 bg_mask = np.zeros_like(test)
405
406         # only get stats in the plate
407         true_pos[bg_mask] = 0
408         true_neg[bg_mask] = 0
409         false_pos[bg_mask] = 0
410         false_neg[bg_mask] = 0
411
412     # now tally

```

```

413 TP = true_pos.sum()
414 TN = true_neg.sum()
415 FP = false_pos.sum()
416 FN = false_neg.sum()
417
418 if not score_bg:
419     total = np.invert(bg_mask).sum()
420 else:
421     total = test.size
422 #print('TP: {}\\t TN: {}\\tFN: {}'.format(TP,TN,FP,FN))
423 #print('TP+TN+FN+FP={}\\ntotal pixels={}'.format(TP+TN+FP+FN,total))
424 # prevent potential overflow
425 denom = np.sqrt(TP+FP)*np.sqrt(TP+FN)*np.sqrt(TN+FP)*np.sqrt(TN+FN)
426
427 if denom == 0:
428     # set MCC to zero if any are zero
429     m_score = 0
430 else:
431     m_score = ((TP*TN) - (FP*FN)) / denom
432
433 if return_counts:
434     return m_score, (TP,TN,FP,FN)
435 else:
436     return m_score
437
438
439 def mean_squared_error(A,B):
440     """
441     get mean squared error between two matrices of the same size
442
443     input:
444         A, B : two ndarrays of the same size.
445
446     output:
447
448         mse:    a single number.
449     """
450
451     try:
452         mse = ((A-B)**2).sum() / A.size
453
454     except ValueError:
455         print("inputs must be of the same size")
456         raise
457
458     return mse
459
460 def chain_lengths(iterable):
461     pos, s = 0, 0
462
463     for b, g in itertools.groupby(iterable):
464
465         if not b:
466             # alternative if the bottom doesn't work or something
467             #d = deque(enumerate(g,1), maxlen=1)
468             #pos += d[0][0] if d else 0
469
470             pos += sum((1 for i in g if not i))
471
472         else:
473
474             s = sum(g)
475
476

```

```

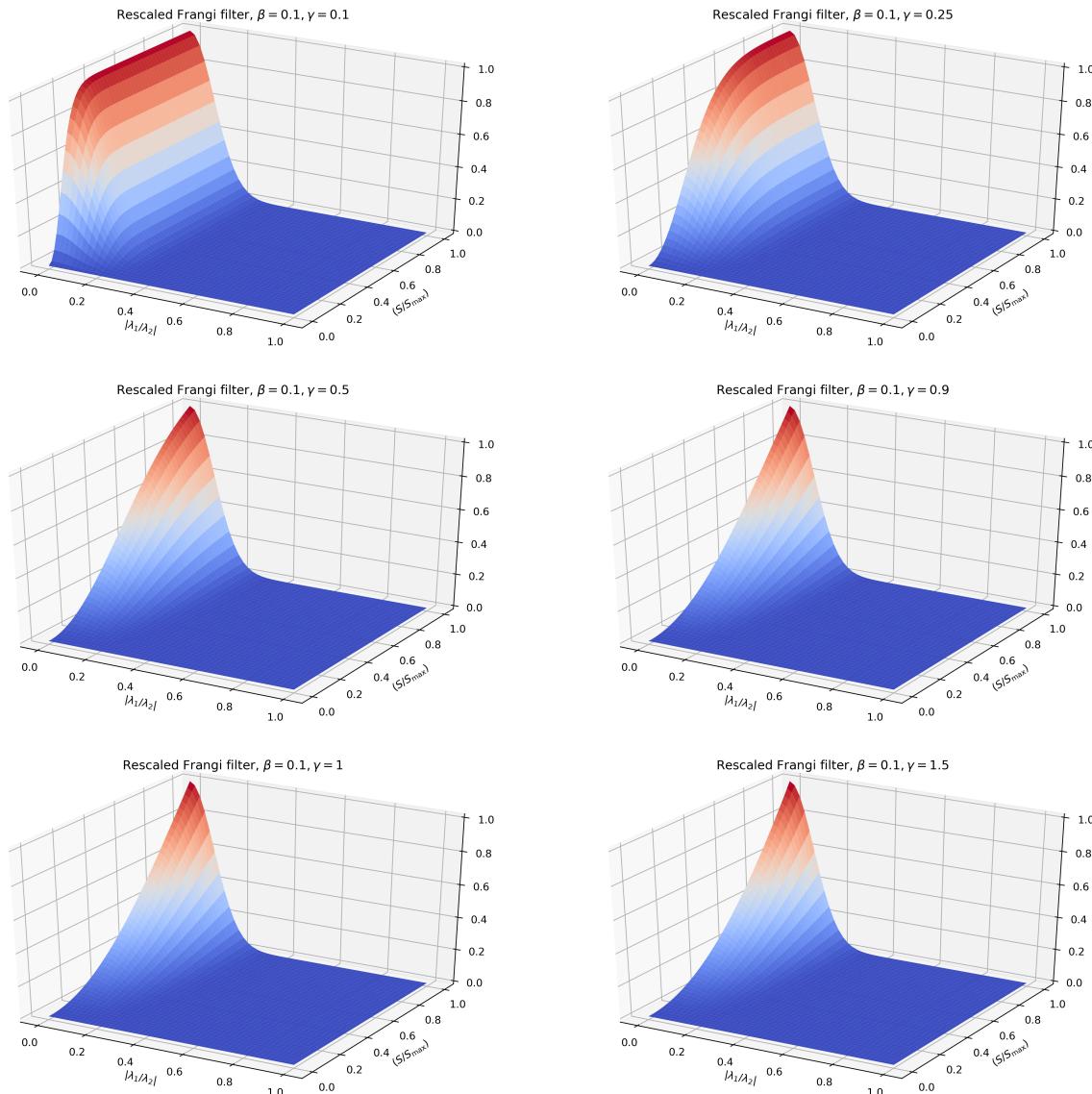
477     yield pos, s
478
479     pos += s
480
481 if not s:
482     # so it will return something even if iterable is empty
483     yield 0, 0
484
485
486 def _longest_chain_1d(iterable):
487     """ Will return a tuple of ind, length
488     where ind is the position in the iterable the chain starts and length is the
489     length of the chain
490     """
491     return max(chain_lengths(iterable), key=lambda x: x[1])
492
493
494 def longest_chain(arr, axis):
495     """Find where the longest chain of boolean values and occurs across an array
496     and also return its length
497     """
498
499     C = np.apply_along_axis(_longest_chain_1d, axis, arr.astype('bool'))
500
501     start_inds, chain_lens = np.split(C, 2, axis)
502
503     return np.squeeze(start_inds), np.squeeze(chain_lens)
504
505
506 if __name__ == "__main__":
507
508     import matplotlib.pyplot as plt
509     from skimage.data import binary_blobs
510
511     A = binary_blobs()
512     B = binary_blobs()
513
514     true_neg_color = np.array([247, 247, 247], dtype='f') # 'f7f7f7'
515     true_pos_color = np.array([0, 0, 0], dtype='f') # '000000'
516     false_neg_color = np.array([241, 163, 64], dtype='f')# 'f1a340'
517     false_pos_color = np.array([153, 142, 195], dtype='f') # '998ec4'
518
519     C = confusion(A, B)
520
521     fig, (ax0, ax1, ax2) = plt.subplots(nrows=1,
522                                         ncols=3,
523                                         figsize=(8, 2.5),
524                                         sharex=True,
525                                         sharey=True)
526
527     ax0.imshow(A, cmap='gray')
528     ax0.set_title('A')
529     ax0.axis('off')
530     ax0.set_adjustable('box-forced')
531
532     ax1.imshow(B, cmap='gray')
533     ax1.set_title('B')
534     ax1.axis('off')
535     ax1.set_adjustable('box-forced')
536
537     ax2.imshow(C)
538     ax2.set_title('confusion matrix of A and B')
539     ax2.axis('off')
540     ax2.set_adjustable('box-forced')

```

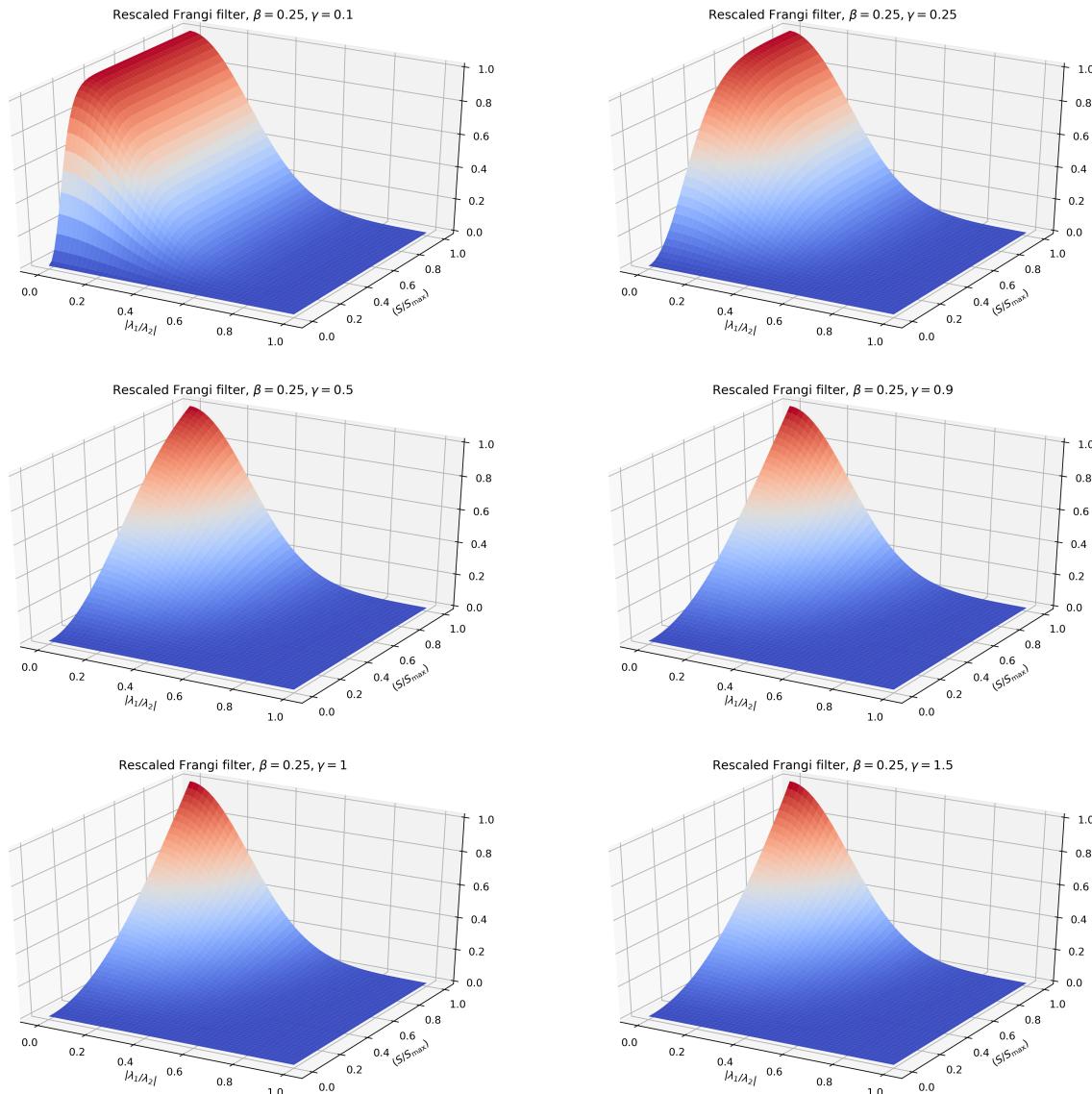
541  
542       **fig.tight\_layout()**

---

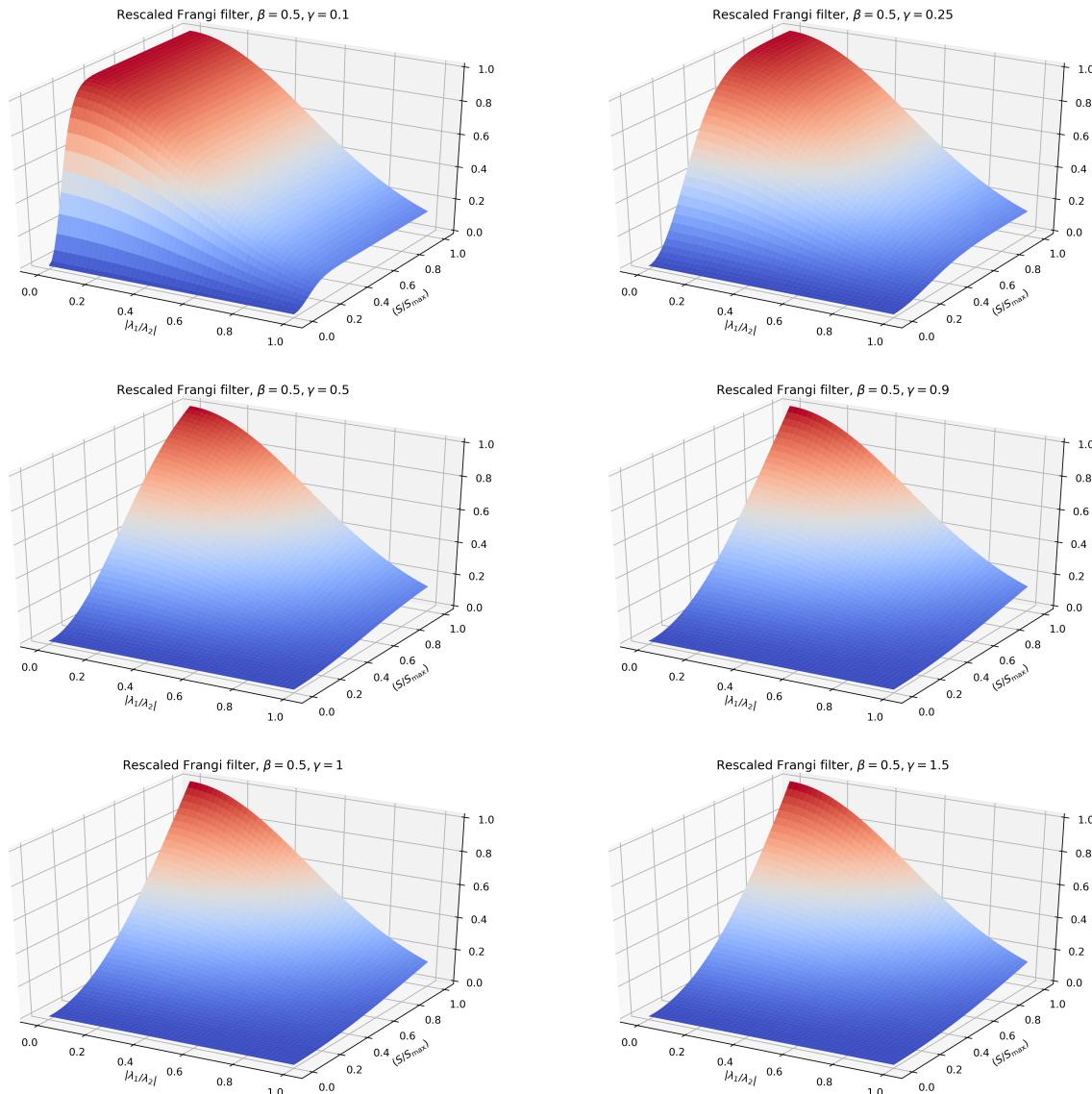
**APPENDIX B**  
**3D VISUALIZATION OF THE FRANGI FILTER**



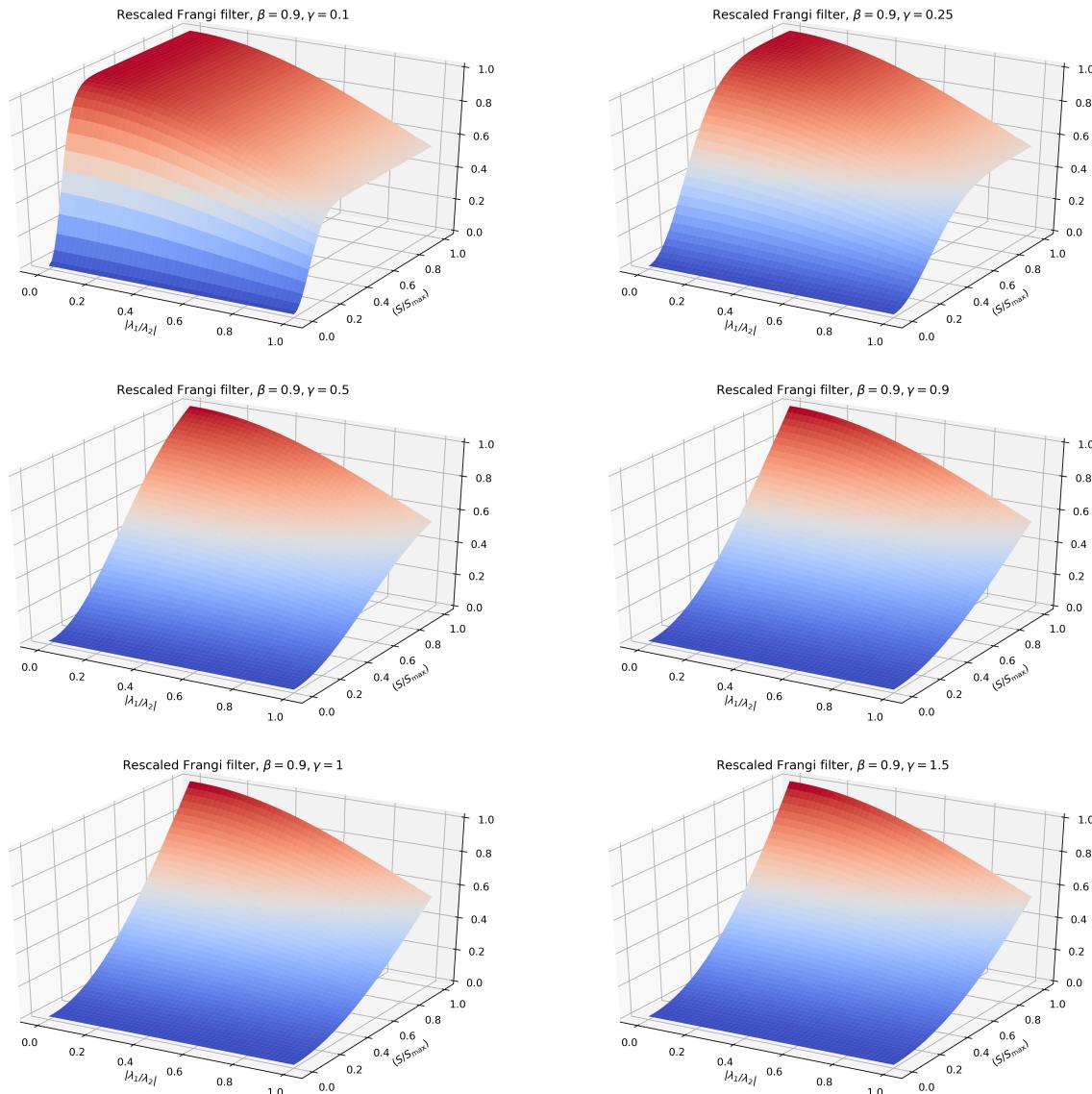
**FIGURE 27:** 3D graph of the Frangi Vesselness Measure, variable  $\gamma$ ,  $\beta = 0.1$



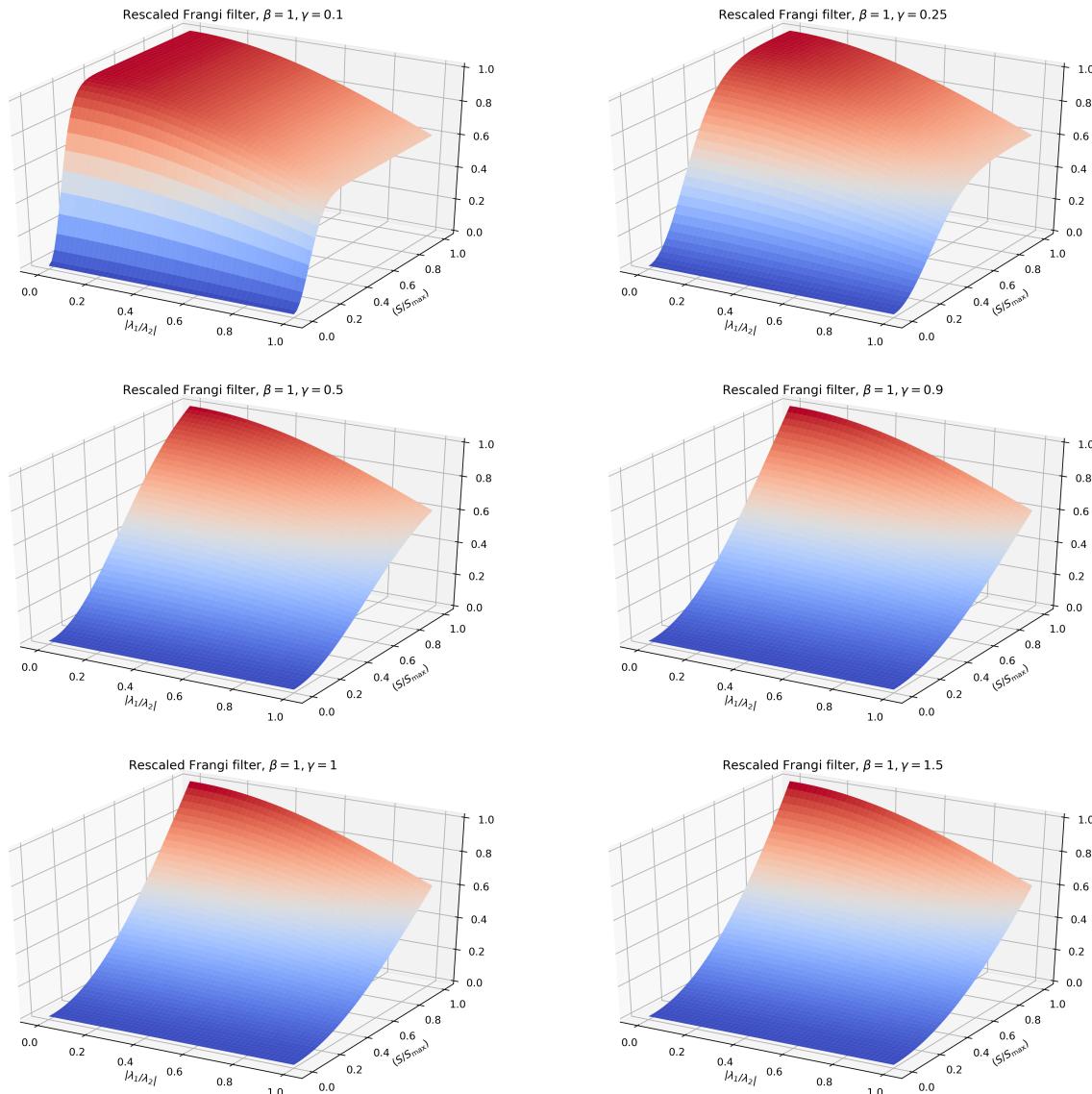
**FIGURE 28:** 3D graph of the Frangi Vesselness Measure, variable  $\gamma$ ,  $\beta = 0.25$



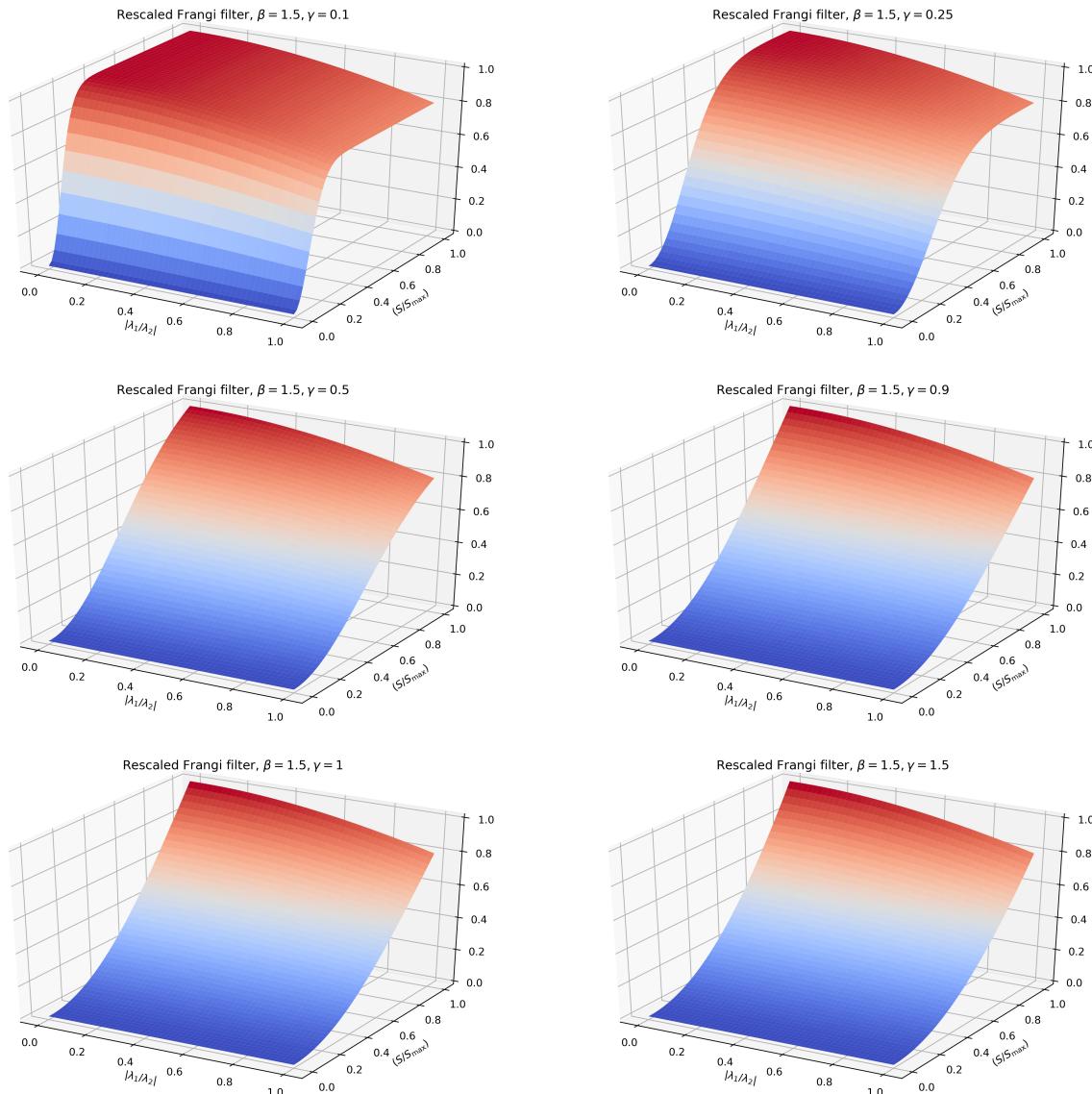
**FIGURE 29:** 3D graph of the Frangi Vesselness Measure, variable  $\gamma$ ,  $\beta = 0.5$



**FIGURE 30:** 3D graph of the Frangi Vesselness Measure, variable  $\gamma$ ,  $\beta = 0.9$



**FIGURE 31:** 3D graph of the Frangi Vesselness Measure, variable  $\gamma$ ,  $\beta = 1$



**FIGURE 32:** 3D graph of the Frangi Vesselness Measure, variable  $\gamma$ ,  $\beta = 1.5$

## BIBLIOGRAPHY

## BIBLIOGRAPHY

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

- [14] S. D. Olabarriaga, M. Breeuwer, and W. Niessen, in *International Congress Series*, Elsevier (PUBLISHER, ADDRESS, 2003), Vol. 1256, pp. 1191–1196.
- [15] J. J. Koenderink, *Biological Cybernetics* **50**, 363 (1984).
- [16] J. Sporring, in *Gaussian Scale-Space Theory*, edited by L. Florack, M. Nielsen, and P. Johansen (Kluwer Academic Publishers, Norwell, MA, USA, 1997).
- [17] E. Hille and R. Phillips, *Functional Analysis and Semi-groups*, American Mathematical Society: Colloquium publications (American Mathematical Society, ADDRESS, 1957).
- [18] J. Babaud, M. Baudin, R. O. Duda, and A. P. Witkin, *IEEE Transactions on Pattern Analysis & Machine Intelligence* **8**, 26 (1986).
- [19] T. Lindeberg, *IEEE transactions on pattern analysis and machine intelligence* **12**, 234 (1990).
- [20] T. Lindeberg, *On the construction of a scale-space for discrete images* (KTH Royal Institute of Technology, ADDRESS, 1988).
- [21] M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, ninth dover printing, tenth gpo printing ed. (Dover, New York, 1964).
- [22] T. Lindeberg, *Journal of Mathematical Imaging and Vision* **3**, 349 (1993).
- [23] T. Lindeberg, *International journal of computer vision* **30**, 79 (1998).
- [24] B. Fornberg, *Mathematics of computation* **51**, 699 (1988).
- [25] A. Morar, F. Moldoveanu, and E. Gröller, in *2012 IEEE 8th International Conference on Intelligent Computer Communication and Processing*, IEEE (PUBLISHER, ADDRESS, 2012), pp. 213–220.
- [26] E. Jones, T. Oliphant, P. Peterson, *et al.*, SciPy: Open source scientific tools for Python, 2001–, [Online; accessed *today*].
- [27] S. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, T. Yu, and the scikit-image contributors, *PeerJ* **2**, e453 (2014).
- [28] S. Damelin and N. Hoang, *International Journal of Mathematics and Mathematical Sciences* **2018**, (2018).
- [29] H. Lange, in *Medical Imaging 2005: Image Processing*, International Society for Optics and Photonics (PUBLISHER, ADDRESS, 2005), Vol. 5747, pp. 2183–2193.
- [30] B. Matthews, *Biochimica et Biophysica Acta (BBA) - Protein Structure* **405**, 442 (1975).

- [31] I. Laptev, H. Mayer, T. Lindeberg, W. Eckstein, C. Steger, and A. Baumgartner, Machine Vision and Applications **12**, 23 (2000).