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

A THESIS

Presented to the Department of Mathematics and Statistics

California State University, Long Beach

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Applied Mathematics

Committee Members:

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

College Designee:

Tangan Gao, Ph.D.

By Lucas Allen Wukmer

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

May 2019

WE, THE UNDERSIGNED MEMBERS OF THE COMMITTEE,

HAVE APPROVED THIS THESIS

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

By

Lucas Allen Wukmer

COMMITTEE MEMBERS

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

ACCEPTED AND APPROVED ON BEHALF OF THE UNIVERSITY

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

California State University, Long Beach

May 2019

ABSTRACT

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

By

Lucas Allen Wukmer

May 2019

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

Much work has been to develop reliable vascular network segmentation methods for well-known image domains (such as retinal MRA images) using Hessian-based filters, namely the multiscale Frangi filter. It is desirable to extend these techniques to study placental images, but the approach has been historically hindered by the comparative irregularity of the placental surface as a whole, which introduces significant noise into filtered result. Previous work has either involved additional filtering [2] or other techniques that are often time-consuming and resource intensive [3].

Here we provide an in-depth mathematical background of the multiscale Frangi filter.

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

We demonstrate the effectiveness of our sped-up implementation of the Frangi filter by performing a large (twenty scales) multiscale Frangi filter on a subset of 201 placental images from a private database provided by the National Children's Study (NCS). We then compare several approaches of merging the multiscale result into an approximation of the PCSVN and compare them to manual tracings of the network. Finally, we develop the notion of the *signed* Frangi filter, upon which we describe a novel-yet-straightforward segmentation method called "trough filling".

ACKNOWLEDGEMENTS

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

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

CHAPTER 1

INTRODUCTION

Recent statistical analysis has suggested a link between placental development and incidence of Autism Spectrum Disorder (ASD) [1]. There is some evidence as in [4] of correlation between this risk and placental health factors. Most ASD cases are not diagnosed until the child reaches three or four, so the benefit of an early detection technique would be massive, as the brain may be more receptive to treatment at a young age. In particular, it was shown in [4] that measurements of the placental chorionic surface vascular network (PCSVN) may be useful in identifying such risk. [1] has provided a method of automatically calculating such features from an extracted vascular network, but does so with manual tracing of the PCSVN in order to make these measurements. These manual tracings are labor-intensive, requiring 4 to 8 hours of user intervention to generate each trace. The present work follows several other efforts towards an automated extraction technique, such as [5], [2], and most recently [3]. Automating this procedure would not only allow more data acquisition to study the potential connection between ASD and placental health, but could even potentially serve as the basis of real-time diagnostic for neonatal risk factors as well. We continue the work of developing a procedure to automate extraction of the PCSVN.

Our basic goal of “vascular network extraction” is a common one in image processing. There are have been many techniques adapted to extracting vascular networks. However, the placenta in particular poses as a particularly difficult image domain to work with. It is a surface network with much irregularity: there are frequent gaps and crossings, and the "background" has a great degree of variation in color intensity and structure itself, causing many naïve approaches at segmentation to fail completely.

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

Here we provide an in-depth mathematical background of the Frangi filter and its justification as an image-processing technique, as well as an introduction to the Gaussian scale space theory common to many multiscale methods. A more throughout treatment of these underlying theories allows us to choose parameters much more meaningfully than previous efforts. In particular, a more thoughtful selection of the Frangi filter's two parameters reduces much of the noise previously encountered within this image domain. Finally, we discuss an important advancement in implementation–scale space conversion for differentiation (i.e. Gaussian blur) via Fast Fourier Transform, which offers a significant speedup. This allows us faster calculation of the eigenvalues of the Hessian, from which we calculate the Frangi filter, a vesselness measure.

We demonstrate the effectiveness of our sped-up implementation of the Frangi filter by performing a large-scale multiscale Frangi filter on a set of 201 placental images from a private database provided by the National Children's Study (NCS). We then demonstrate the usefulness of this filter by then demonstrating several approaches to segmentation of the PCSVN from our filtered result. These approaches include simple thresholding, scalewise percentile filtering, and finally a custom Frangi-based method we call "trough-filling." We compare each of these segmentation results to the ground truth manual tracings of the network for each of these 201 samples, quantifying the success of various segmentation methods using Matthew's correlation constant (MCC) and precision scoring. Finally, we discuss ways to extend our ideas to solve the

network connection problem.

CHAPTER 2

DIFFERENTIAL GEOMETRY AND SURFACE CURVATURE

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

2.1. 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)**.**

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

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

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

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

That is, the function h is identical to the pixel matrix $\mathtt{I}$ at all integer inputs, and simply a "smooth enough" interpolation of those points for all other values.

It is of course necessary to admit that `I` is not really a perfect representation of the underlying "content" within the picture. Not only is information lost when `I` is stored as an integer, there are also elements of noise and anomalies of lighting that would constitute noise to the original signal. There are multiple treatments of image processing that do address this discrepancy in a pragmatic way [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 7.1), there are numerous shortcomings therein, and improvements to the veracity of our original signal could be made from many angles. Though we shall draw upon the notion of the pixel matrix `I` as a sampling again to motivate our development of scale space theory in Chapter 4, we ultimately use these techniques because we find them successful to our problem.

2.2. 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 [7].

2.2.1. Preliminaries of Differential Geometry

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

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

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

Since the graph f is clearly one-to-one by definition, we may readily associate any input $u \in U$ with its corresponding output $p \in f[U]$, i.e. $p = f(u) = f(u_1,u_2) \;=\; \big(u_1,u_2,h(u_1,u_2)\big)$, depending on whether we wish to focus on a point of a graph in terms of its input or in terms of the structure of the graph itself.

Our development of curvature ultimately will hinge upon a careful consideration of the tangent plane of f at a point p, for we will require a concrete definition of both the tangent space within the domain and image of f, as well as the so called "differential" of f, the lattermost of which we will only define for the immediate case required.

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

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

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

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

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

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

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

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

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

Although not necessary presently, we could just as easily consider the differential of an arbitrary function as a map between tangent vectors in the function’s domain and tangent vectors in its range. We could also just identify this as mapping $U \to \mathbb{R}^3$ by the obvious isomorphism described above. and then differential of f at x is simply a linear transformation of between the

tangent spaces T_uU 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_uf, the tangent plane of f at an input u.

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

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

The vectors in this plane can thus be identified as tangent vectors from T_uU that have been passed through the differential mapping $Df|_u$. We shall denote a generic tangent vector $X \in T_uf$ 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, $\text{span}\left\{\frac{\partial f}{\partial u_1}, \frac{\partial f}{\partial u_2}\right\} = T_uf$.

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_uf.*

Quite obviously, we're assuming $(1,0),(0,1) \in U$. If this is not the case, we pick some α small enough so that $(\alpha,0)$ and $(0,\alpha)$ are contained and this scaled version would serve as a basis instead.

Proof. Given the definition of a graph f as in Definition 2.3, we can directly calculate the partial derivatives of f at a point u.

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

which are obviously linearly independent. Then $Df|_u(1,0) = f_{u_1}$, and $Df|_u(0,1) = f_{u_2}$, which

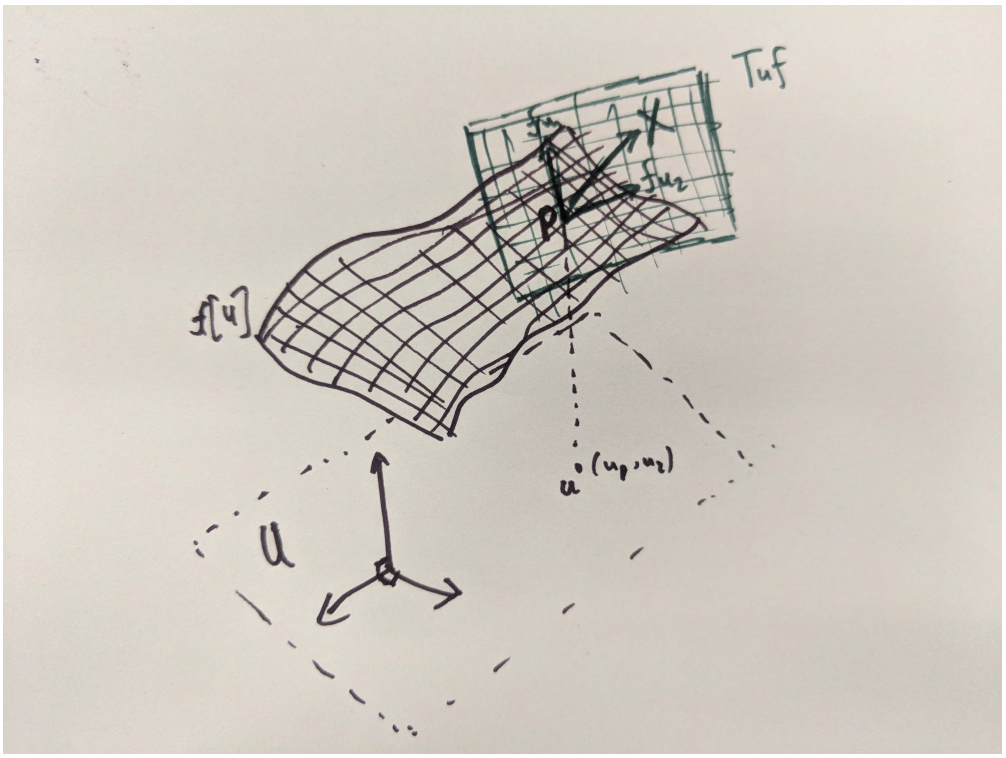


FIGURE 1: Tangent plane of a graph

shows $\left\{\frac{\partial f}{\partial u_1}, \frac{\partial f}{\partial u_2}\right\} \in T_u f$. Thus $\left\{\frac{\partial f}{\partial u_1}, \frac{\partial f}{\partial u_2}\right\}$ is a linearly independent subset of $T_u f$, and can serve as its basis. □

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

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

2.2.2. Curvature of a surface and its calculation

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

$\kappa(s) := \|c''(s)\|$.

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

Definition 2.8 (Surface curve)**.** *Given a closed interval $I \subset \mathbb{R}$, we call the regular curve $c : I \to \mathbb{R}^3$ a surface curve in the event that* $\text{image}(c) \subset \text{image}(f)$ *entirely. The one-to-one-ness of the graph f ensures that we can define (for the given curve) an intermediary parametrization θ_c so that $c = f \circ \theta_c$. That is,*

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

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

Note as well that the velocity of this particular curve lies within $T_u f$. This can be seen by an elementary application of chain rule:

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

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

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

Considering a point $p \in I$ and its associated point $u = \theta_c(p)$, we wish to compare the curvatures of all (regular) surface curves passing through the point p at some particular velocity.

We now present a main result that provides a notion of curvature of a surface.

Theorem 2.2 (Theorem of Meusnier)**.** *Given a point $u \in U$ and a tangent direction $X \in T_u f$, any regular curve on the surface $c : I \to \text{image}(f)$ with $p \in I : \theta_c(p) = u$ where $c'(p) = X$ will have the same curvature.*

In other words, any two curves on the surface with a common velocity at a given point on the surface will have the same curvature. To prove this, we require one final definition.

Definition 2.9 (The Gauss Map)**.** *The Gauss map at a point $p = f(u)$ is the unit normal to the tangent plane*

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

Each partial above understood to be evaluated at the input $u \in U$; that is, we calculate $\frac{\partial f}{\partial u_i}\Big|_u$. The existence of the cross product in its definition makes it clear that $\nu \perp \frac{\partial f}{\partial u_i}$ each $i = 1,2$. A simple dimensionality argument of $\mathbb{R}^3$ implies that these must exist in $T_u f$. However, we can also show it directly:

To show that $\left\{ \frac{\partial \nu}{\partial u_1}, \frac{\partial \nu}{\partial u_2} \right\} \in T_u f$, first note that at any particular $u \in U$, $\langle \nu, \nu \rangle = 1 \implies \frac{\partial}{\partial u_i} \langle \nu, \nu \rangle = 0$, and so by chain rule $2\langle \frac{\partial \nu}{\partial u_i}, \nu \rangle = 0 \implies \frac{\partial \nu}{\partial u_i} \perp \nu$. Since $\nu \perp \text{span} \left\{ \frac{\partial f}{\partial u_i} \right\}$ as well (since ν its outer product), in $\mathbb{R}^3$, this implies $\text{span} \left\{ \frac{\partial \nu}{\partial u_i} \right\} \parallel \text{span} \left\{ \frac{\partial f}{\partial u_i} \right\}$.

Thus, we have $\text{span} \left\{ \frac{\partial \nu}{\partial u_1}, \frac{\partial \nu}{\partial u_2} \right\} \subset T_u f$ as well and we can also use it as a basis.

We are finally ready to prove Theorem 2.2, the Theorem of Meusnier.

Proof. Let $X \in T_u f$ be given and consider some curve where $\frac{\partial c}{\partial t}(u) = X$ where $X \in T_u f$. We wish to decompose the curve's acceleration along the orthogonal vectors X and the Gauss map $\nu = \nu(u_1, u_2) = \frac{\frac{\partial f}{\partial u_1} \times \frac{\partial f}{\partial u_2}}{\| \frac{\partial f}{\partial u_1} \times \frac{\partial f}{\partial u_2} \|}$ as in Definition 2.9. Note that X and ν are indeed orthogonal, as $X \in \text{span} \left\{ \frac{\partial f}{\partial u_i} \right\} = T_u f$, and $\nu \perp T_u f$). We then have (at this fixed point $u = \theta_c(p)$)

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

.

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

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

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

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

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

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

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

$$\|c''\| = \|\langle c'', X \rangle X + \langle c'', \nu \rangle \nu\| = \|0 + \langle c'', \nu \rangle \nu\| \tag{2.8}$$

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

$$= -\langle X, \frac{\partial \nu}{\partial t} \rangle \tag{2.10}$$

$$= \langle X, -\frac{\partial \nu}{\partial t} \rangle \tag{2.11}$$

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

$$-\frac{d\nu}{dt} = -\frac{d}{dt}\left[\nu(u_1, u_2)\right] \tag{2.12}$$

$$= -\frac{d}{dt}\left[\nu(\theta_1(t), \theta_2(t))\right] \tag{2.13}$$

$$= \theta_1'(t)\left(-\frac{\partial \nu}{\partial u_1}\right) + \theta_2'(t)\left(-\frac{\partial \nu}{\partial u_2}\right) \tag{2.14}$$

Identifying span $\left\{-\frac{\partial \nu}{\partial u_i}\right\}_{i=1,2}$ as a subset of $T_u f$, we can identify a linear transformation which maps the basis $\left\{\frac{\partial f}{\partial u_i}\right\}_{i=1,2}$ to this subset, which we shall call the Weingarten map L.

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

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

That is, $\mathsf{L}(\frac{\partial f}{\partial u_i}) = -\frac{\partial \nu}{\partial u_i}$ 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

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

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

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

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

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

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

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

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

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

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

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

The map L introduced in the proof above is known as the Weingarten map and is implicitly

defined at each $u \in U$. We wish to make its existence rigorous as well as find a matrix representation for it, using the standard motivation that $\mathsf{L}(\frac{\partial f}{\partial u_i}) = -\frac{\partial \nu}{\partial u_i}$.

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

The Weingarten map can be formally shown to be well-defined, invariant under coordinate transformation in the general case (that is, for surfaces f that are not graphs). We refer to [7] for the general proof. Our present situation is much less delicate, as we're only concerned for cases when f is a graph. In this case, the linear transformation may be simply constructed, and we proceed by simply calculating its matrix representation.

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

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

$$\begin{bmatrix} \\ \widehat{\mathsf{L}} \\ \\ \end{bmatrix} \begin{bmatrix} | & | \\ \frac{\partial f}{\partial u_1}\Big|_{T_u f} & \frac{\partial f}{\partial u_2}\Big|_{T_u f} \\ | & | \end{bmatrix} = \begin{bmatrix} | & | \\ \widehat{\mathsf{L}}\frac{\partial f}{\partial u_1}\Big|_{T_u f} & \widehat{\mathsf{L}}\frac{\partial f}{\partial u_2}\Big|_{T_u f} \\ | & | \end{bmatrix} \tag{2.20}$$

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

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

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

We can simplify this greatly by defining

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

so that

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

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

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

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

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

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

2.2.3. Principal Curvatures and Principal Directions

To do so, we shall consider the relationship between the direction X and the normal curvature κ_ν Definition 2.11 in that direction at some specified u.

First, we need the following lemma:

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

Proof. The result is uninterestingly obtained by tracking each component of $\nabla_v\langle v, Av\rangle$:

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

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

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

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

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

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

□

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

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

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

Using the method of Lagrange multipliers, we define the Lagrangian:

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

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

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

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

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

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

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

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

directions at which normal curvature attains an extremal value are referred to as ***principal directions***.

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

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

$$\widehat{\mathsf{L}} = \mathrm{Hess}(h)\tilde{G}\,, \quad \text{where} \quad \tilde{G} := \frac{1}{\left(1+h_x^2+h_y^2\right)^{3/2}} \begin{bmatrix} 1+h_y^2 & -h_x h_y \\ -h_x h_y & 1+h_x^2 \end{bmatrix} \tag{2.35}$$

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

Proof. We begin from Eq. (2.25). First, consider each component from Eq. (2.23) and rewrite via chain rule:

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

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

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

and we have the unit normal vector (Gauss map)

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

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

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

We then calculate each h_{ij} as

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

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

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

To calcuate the second, we use

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

and thus

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

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

We stress that this map L is defined for each point $u \in U$. In the particular case that $u \in U$ is a critical point, where $\nabla h(u) = (h_x(u), h_y(u)) = 0$, then it is clear from the previous theorem that $\widehat{\mathsf{L}}$ is exactly the Hessian matrix $\mathrm{Hess}(h)$. Of course this implies that $\widehat{\mathsf{L}}$ and $\mathrm{Hess}(h)$ have the same eigenvalues and eigenvectors at any such point.

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

$$\tilde{G} = \frac{1}{1+h_x^2+h_y^2}(I + [\delta]), \quad [\delta] := \begin{bmatrix} h_y^2 & -h_x h_y \\ -h_x h_y & h_x^2 \end{bmatrix} \tag{2.44}$$

We can then rewrite Eq. (2.35) as

$$\widehat{\mathsf{L}} = \frac{1}{\left(1+h_x^2+h_y^2\right)^{3/2}} \left(\mathrm{Hess}(h) + \mathrm{Hess}(h)[\delta]\right) \tag{2.45}$$

We can see that as h_x, h_y are close to zero, $[\delta]$ will be very close to the zero matrix, and the constant $\left(1+h_x^2+h_y^2\right)^{-3/2}$ will be very close to 1 as well, so we should not expect the addition of a "close to 0" matrix to have much effect on the eigenvectors or eigenvalues. This intuition is confirmed by a result from Wilkinson [8], which we state without rigorous proof.

Theorem 2.7. *If A, B are matrices such that* $|A_{ij}| < 1, |B_{ij}| < 1$ *(a condition that can be ignored*

with scaling) and λ is a simple eigenvalue of A, then given $\epsilon > 0$, there exists a simple eigenvalue $\tilde{\lambda}$ of the matrix $A + \epsilon B$ with $|\lambda - \tilde{\lambda}| = O(\epsilon)$. Similarly, if v is an eigenvector of A, then $\tilde{v}$ is an eigenvector of $A + \epsilon B$ with $|v - \tilde{v}| = O(\epsilon)$.

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

We can bound the perturbation of eigenvalues from $\widehat{\mathsf{L}}$ to $\text{Hess}(h)$ by another result which we state without proof [9].

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

In the event that we do wish to rigorously compute the Weingarten map–that is, without concern for the magnitude of the gradient–we refer to [10] and survey papers mentioned therein.

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

2.2.4. The Weingarten map and Principal Curvatures of a Cylindrical Ridge

Let f be the graph given by

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

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

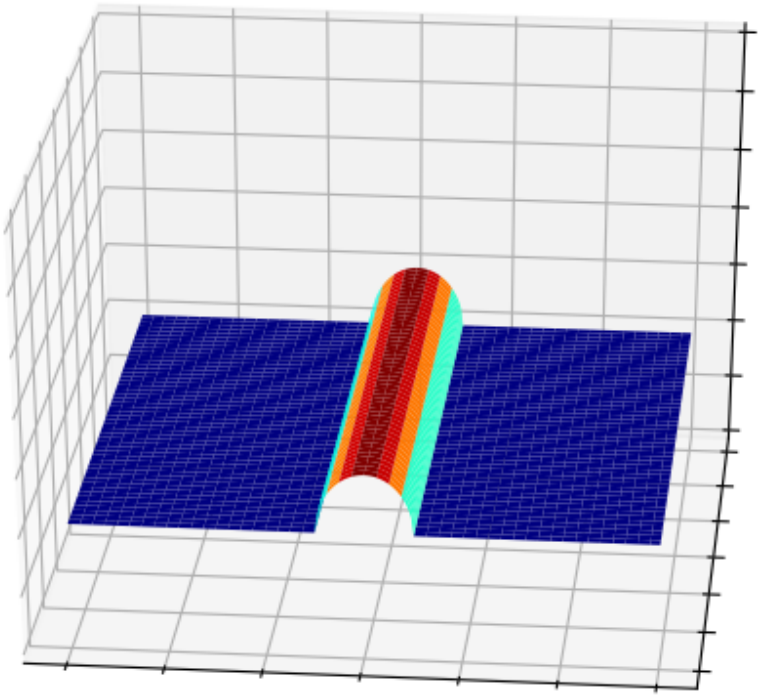


FIGURE 2: The graph of a cylindrical ridge of radius r

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

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

The gauss map is given by

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

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

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

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

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

$$= \begin{bmatrix} -\frac{1}{r} & 0 \\ 0 & 0 \end{bmatrix} \tag{2.53}$$

We see that $u_2 = (0,1)$ and $u_1 = (1,0)$ are eigenvectors for $\widehat{\mathsf{L}}$ with respective eigenvalues $\kappa_2 = -\frac{1}{r}, \kappa_1 = 0$. The Theorem of Olinde Rodriguez then 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, and 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.

2.3. Orthogonality of Principal Directions and Approximated Principal Directions

Our final point is to consider the relationship between the principal directions of a surface at a point. Viewing the surface in $\mathbb{R}^3$, we define the Hessian $\text{Hess}(x,y)$ of the surface L at a point (x,y) on the surface as the matrix of its second partial derivatives:

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

At any point (x,y) we denote the two eigenpairs of the Hessian matrix of h as

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

where κ_i and u_i are known as the *approximated principal curvatures* and *approximated principal directions* of $L(x,y)$, respectively, and we label such that $|\kappa_2| \geq |\kappa_1|$. Notably, $\text{Hess}(x,y)$ is a real, symmetric matrix (since $L_{xy} = L_{yx}$ and L is a real function) and thus its eigenvalues are real and its eigenvectors are orthonormal to each other, as given by following basic result from linear algebra [11].

Lemma 2.9. *Let A be a real, symmetric matrix. The eigenvalues of A are real and its eigenvectors are orthonormal to each other.*

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

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

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

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

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

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

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

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

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

and is perpendicular to v_1, since

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

□

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

CHAPTER 3

THE UNISCALE FRANGI FILTER

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

The Frangi filter, first described by Alejandro Frangi et al. in [12] is a widely used Hessian-based filter within image processing. Hessian-based filters make use of the logical "proximity" of the Hessian to notions of curvature of surfaces, as developed in Section 2.2. Several such Hessian-based filters exist–see [13] and [14], as well as a comparison given in [15]. These filters use information about the principal curvatures, approximated as eigenvalues of the Hessian) at each point in the image to identify regions of significant curvature within an image.

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

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

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

where

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

and β and c are tuning parameters. Before we discuss appropriate values for β and c, we first seek to highlight the significance of Eq. (3.1), and in particular, the ratios defined in Eq. (3.2). A and S

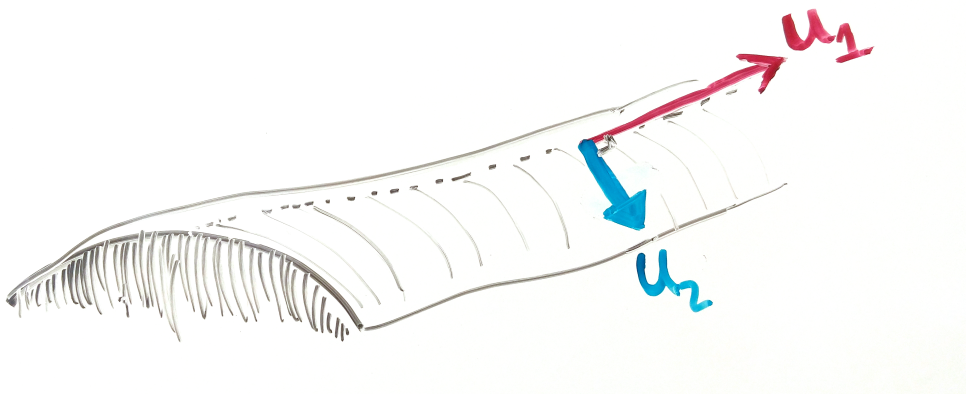


FIGURE 3: The principal eigenvectors at a ridge like structure

are known as the anisotropy measure and structureness measure, respectively. Consequently, we'll refer to the two factors in Eq. (3.1) as the anisotropy factor and structureness factor, respectively.

3.1. Anisotropy Measure

The anisotropy (or directionality) measure A is simply the ratio of magnitudes of λ_1 and λ_2. Since at a ridge point of a tubular structure, we should have $\lambda_1 \approx 0$ and $|\lambda_2| \gg |\lambda_1|$, a very small value of A would be present at a ridge of a tubular structure.

In Fig. 3, this situation is demonstrated. Here, u_1,u_2 form the orthogonal set of Hessian eigenvectors with corresponding eigenvalues λ_1 and λ_2. At such a ridgelike structure, we could predict the largest change in curvature to be straight down the ridge (in the direction of u_2), and the direction of least curvature to be directly along the ridge (in the direction of u_1). $\lambda_1 \approx 0$ and λ_2 is large and negative.

Of course, if the the ridge is perfectly circular along its cross section (as was in Section 2.2.4, it is of course apparent that λ_2 would be the same value at any place along the ridge (not just at its crest), and λ_1 would likewise be 0 at any such point. One could also imagine a similar situation in which the dropoff from crest to bottom gets increasingly steep. In such a case, λ_2 as a function of transverse position 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 7.1), there will be points where we seem to "lose" our ridgelike structure, but this is simply due to an error in the sample.

Importantly, this formulation does not require λ_1 to be approximately zero, just that the curvature in the downward direction is much more significant than in the transverse direction.

Also the crest could be really flat ("hangar shaped"), in which case both are around zero. At the crest of the ridge, we would actually expect both u_1 and u_2 to be around 0, whereas a point somewhere between the crest and the "foot" of the ridge to contain the maximum u_2. We will fix this issue specifically by casting this as a multiscale problem in Chapter 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 Chapter 9.

3.2. Structureness measure

There is another concern with using the pure ratio $S := |\lambda_1/\lambda_2|$ as an identifying feature of ridgelike structures apart from the ones listed above. We could 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.

3.3. The Frangi vesselness measure

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

The form Frangi arrived at in Eq. (3.1) in which a factor of $\exp(\cdots)$ and $(1-\exp(\cdots))$ 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 Eq. (3.1), ensuring that λ_2 is not positive. If we are indeed at a curvilinear ridge, we need the second derivative of the surface in the maximal direction to be negative, which hasn't been accounted for as yet in our formulation of A and S – we wish (for our purposes) to only identify when we are finding crests. A will still be small and S will still be large however if we identify a "trough".

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

3.4. Choosing parameters β and c

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

Frangi suggested for c that half of (the Frobenius norm of the) Hessian matrix is appropriate, simply because the minimum value of S is zero, and its maximum value is exactly the max Frobenius norm. With this in mind we would like to introduce the scaling factor γ, so that

$c = \gamma S_{\max}$. This creates a minor annoyance though: although the anisotropy factor can certainly attain a value of 1, if c is to take this "appropriate" value, the maximum value of the structureness factor is somewhat smaller than 1. In fact,

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

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

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

where, as before,

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

and

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

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

As we will show later, choosing the structureness parameter γ is rather important for the context especially if the background (non-ridgelike structure) is significant and noisy. β should be strengthened/relaxed depending on how "flat" the ridgelike structure is. We shall show empirically that contexts in which more 'bloblike' structures are known to be present than that for which the Frangi filter was originally designed, we will benefit from a smaller choice of β.

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

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

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

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

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

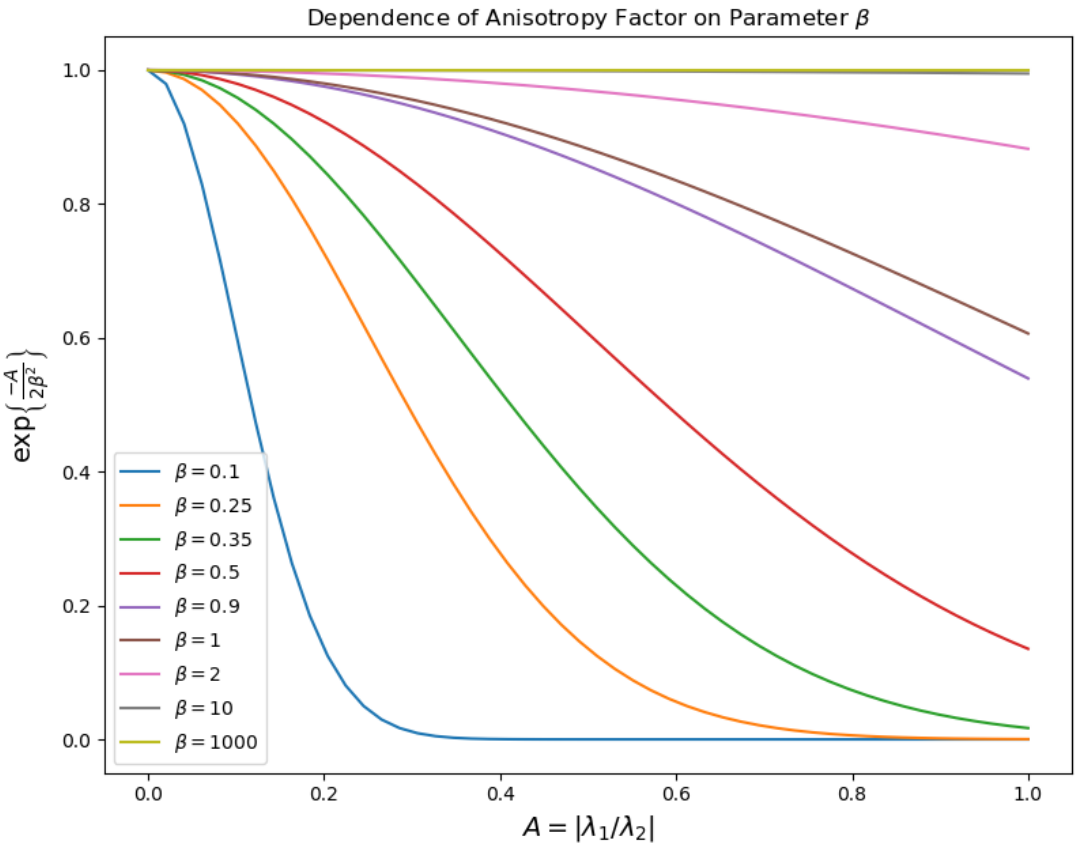


FIGURE 4: Dependence of the Anisotropy Factor on its Parameter

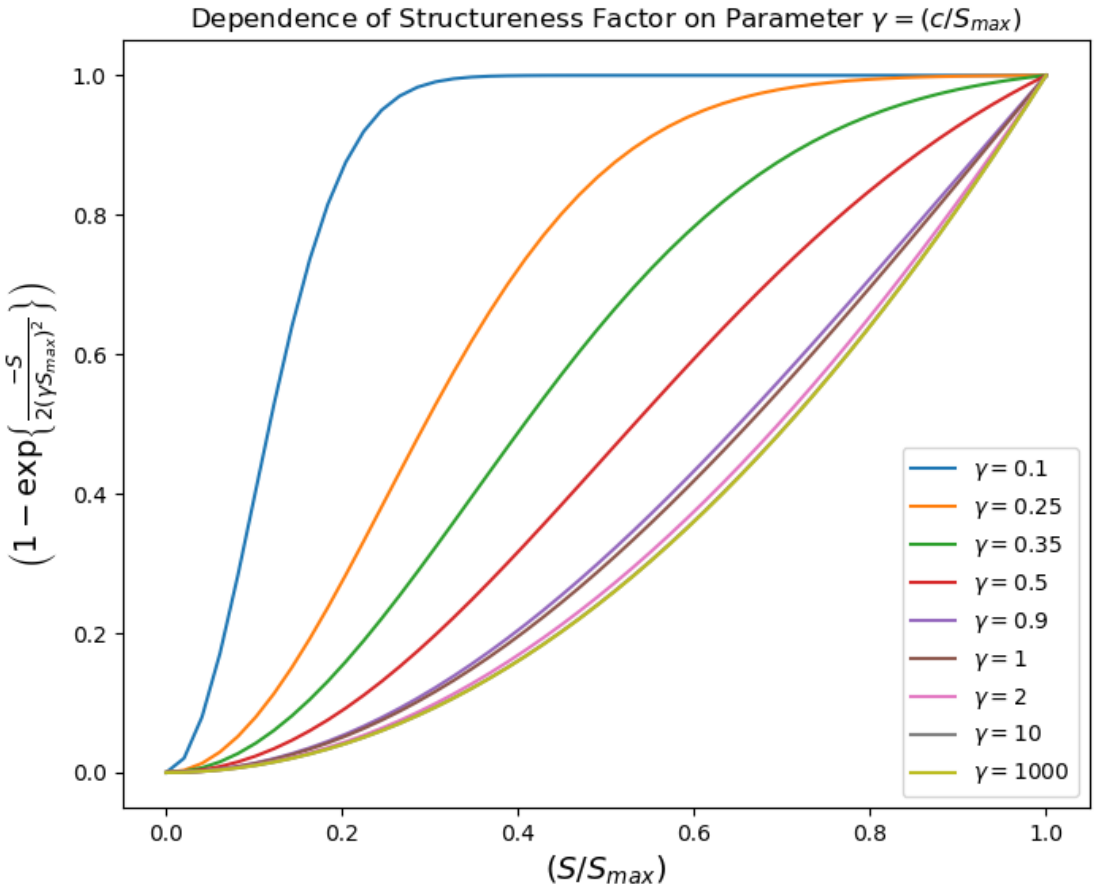


FIGURE 5: Dependence of the Structureness Factor on its Parameter

CHAPTER 4

LINEAR SCALE SPACE THEORY

Although the ideas presented above require differentiation of continuous surfaces, our image is in fact composed of discrete pixels. That is, our previous discussions have been in terms of an image as the continuous surface in Definition 2.2, rather than the more realistic discrete pixel matrix as in Definition 2.1. The present section seeks to address this disconnect. Divided differences can serve as an analogue of differentiation in discrete contexts, but our "derivative" and any use of it is then completely dependent on the bias of our limited sampling of the “true” 3D surface. Our main goal is to counter against some of the bias of our particular sampling. In particular, we wish to not over-represent structures that are clear at our resolution without giving appropriate weight to larger structures as well. Ideally, we would like statements we make about higher order phenomena to be stable enough at least that we would arrive at the same conclusions from analyzing a higher resolution image of the same surface. Koenderink [16] argued that "any image can be embedded in a one-parameter family of derived images, with resolution as the parameter in essentially only one unique way" given a few of the so-called scale space axioms. He, along with other independent efforts, showed that a small set of intuitive axioms imply that any such family of images $\{K_\sigma\}$ must satisfy the heat equation

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

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

This result is intuitive and desirable–we would anticipate a lower-resolution version of our

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

4.1. Properties/Axioms of Linear Scale Space Theory

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

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

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

Axiom 4.1 (Linear-shift and Rotational Invariance)**.** *We require that no position in the original signal is favored. This is intuitive, as our operation should apply to any image fairly, regardless of where content is found in the image, and cropping or rotating our initial image should not affect resolution of the content.*

Axiom 4.2 (Semigroup property)**.** *The semigroup property is simply that transforming the original image by some resolution σ should have the same overall effect of two successive transformations σ_1 and σ_2, i.e.*

$$T_\sigma u = T_{\sigma_1+\sigma_2} u \tag{4.3}$$

Axiom 4.3 (Continuity of Scale Parameter)**.** *There is no reason for the scale parameter to be discrete; we may alter the resolution with whatever precision we desire. That is, we take the resolution parameter σ to be any nonzero real number. Moreover, we require that the operator behaves continuously with respect to the scale parameter.*

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

Axiom 4.4 (Causality Condition)**.** *The causality condition is the one that, as resolution decreases, no finer detail is introduced into the image. That is, as the scale increases, there will be no creation of local extrema that did not exist at a smaller scale.*

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

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

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

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

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

4.2. Sufficiency of the Gaussian Kernel

Although we will omit it here, the above requirements are actually sufficient in proving not only that the operator T_σ is a convolution, but that the heat equation described in Eq. (4.1) must hold. This has been shown in various ways, both by Koenderink [16], Babaud [18], as well as Lindeberg in [17]. Of course, once Eq. (4.1) has been established, it is straightforward to show that

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

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

4.3. Scale Spaces over Discrete Structures

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

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

Definition 4.1 (Sampled Gaussian Kernel and Generated Family)**.**

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

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

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

In [19] and in particular [20], Lindeberg demonstrated that the sampled Gaussian kernel violates

not only semigroup property (Axiom 4.2), but–much less forgivably–the causality property (Axiom 4.2). There is absolutely no guarantee that convolution with a sampled Gaussian kernel will not create "spurious" structures as resolution increases.

Fortunately, Lindeberg was immediately able to remedy this by providing a discrete analogue of the Gaussian kernel, which does satisfy Axiom 4.4 and Axiom 4.2:

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

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

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

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

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

The derived family of 1D signals is then given by

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

The compatibility of scale space theory and derivatives on discrete structures and extension to two dimensions was also demonstrated by Lindeberg in [22] and [23]. In particular, we may take derivatives of the convolutions of our discrete images using, say, a local central difference. Lastly, the 2D version of the family given in Eq. (4.8) can be obtained by independent convolution of its dimensions (i.e. it is separable).

CHAPTER 5

THE MULTISCALE AND SIGNED FRANGI FILTER

With the ideas of scale established, we may return to our discussion of the Frangi filter. Our ideas of scale developed in the previous section imply that, if the ridgelike structures we wish to detect are more prominent at different scales, then a multiscale approach is the natural one. Considering the dependence of the Frangi filter's response on choice of scale as demonstrated in Chapter 3, we wish to probe at multiple scales regions that would receive a high vesselness score at any range and somehow merge the result. Frangi [12] approached this problem by simply taking the maximum vesselness measure over all scales. Thus the multiscale Frangi vesselness score at the pixel (x_0, y_0) would be

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

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

5.1. Rudimentary Thresholding

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

At this point, Frangi [12] refrained from explicitly interpreting the score assigned by Eq. (5.1); that is–whether a particular pixel (x_0, y_0) in the image definitely respresents a vessel or not based on its Frangi score. Instead, he cautioned that the result should not be used as a segmentation method alone; moreover, the width of the vasculature cannot be determined rigorously from the Frangi filter, as discussed in Chapter 3.

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

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

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

A good alternative method of thresholding would be to simply select the highest scores from each responses: we calculate a high percentile score and threshold at that value. Due to the large number of zeros outputted by the filter, we opt instead to take the qth percentile of only nonzero values of V_Σ. We briefly demonstrate this in Fig. 6 on a particularly well-behaved sample. The top left value image is the base image, the top right is $\mathcal{V}_{\max}$, the bottom left is $\mathcal{V}_{\max}$ thresholded at the 95th percentile, and the bottom left is thresholded at the 98th percentile.

Of course, the downside of this method is that we do not know in general the size of the network, and we may miss entire branches of the vascular network if there is a large amount of curvilinear content elsewhere in the image.

We will discuss alternatives methods of aggregating results from our multiscale method, as well as optimal values for parameters and scales in Chapter 9. As a final note, we admit that

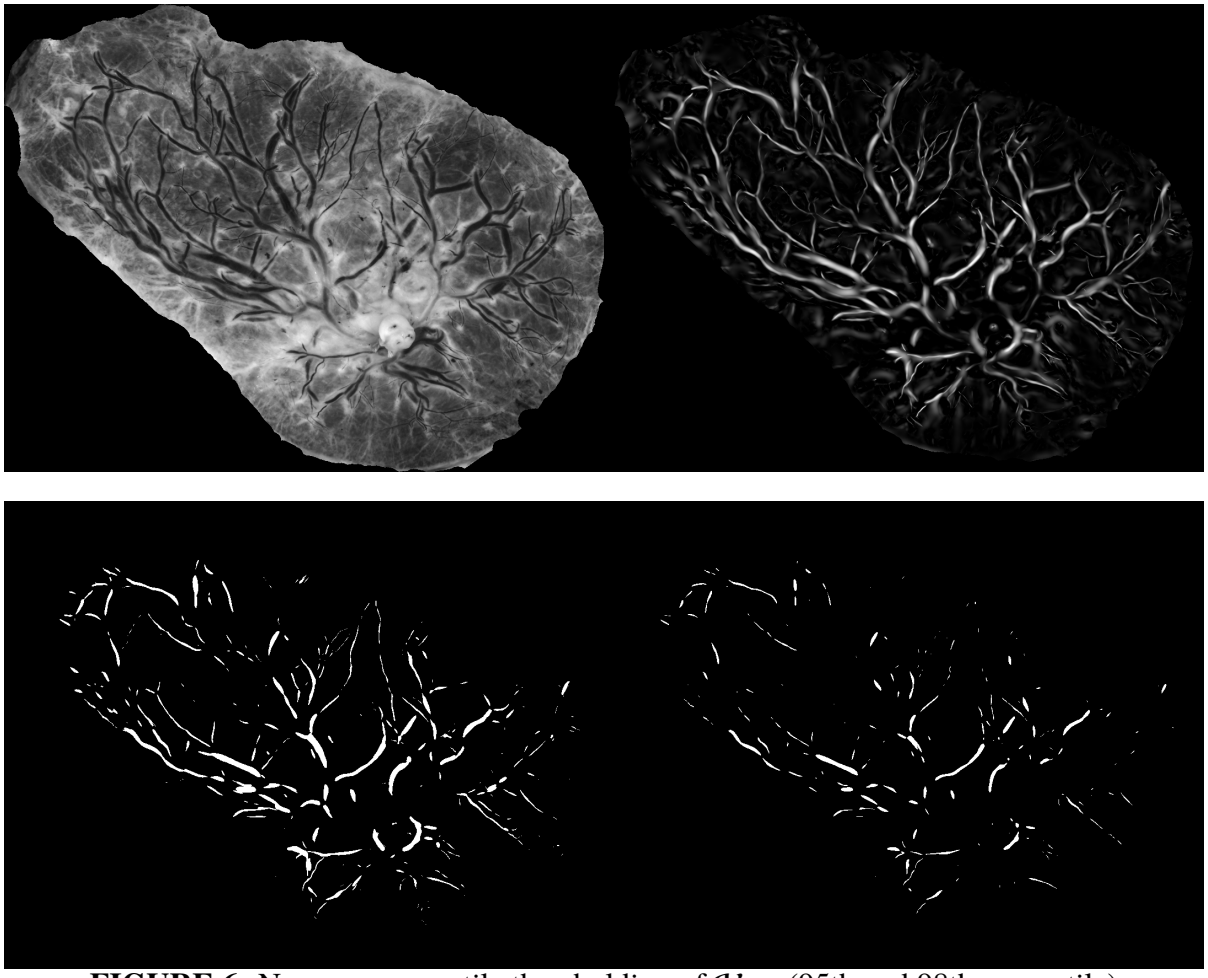


FIGURE 6: Nonzero-percentile thresholding of $\mathcal{V}_{\max}$ (95th and 98th percentile)

.

any future extensions of this work (as will be discussed in Chapter 10) need not proceed from either of these thresholdings; analyzing the raw vesselness score $\mathcal{V}_{\max}$, or even the un-merged scale-wise scores may prove far more rewarding.

5.2. The Signed Frangi Filter

We finally introduce the novel (yet straightforward) notion of the signed Frangi filter. As will be shown in Chapter 9, we can befefit from simultaneously calculate for a dark background and a light background. Since the Frangi filter normally throws away any response where $\lambda_2 < 0$ (if dark curvilinear features are targeted) or $\lambda_2 > 0$ (if light curvilinear features are targeted), we lose no computation time at all (although we must store more results). After computing the multiscale result, we can easily separate these into a positive and a negative strain, which we will denote $\mathcal{V}_{\max}^{(+)}$ and $\mathcal{V}_{\max}^{(-)}$. Our $\mathcal{V}_{\max}^{(+)}$ is the same as our $\mathcal{V}_{\max}$ before, and $\mathcal{V}_{\max}^{(-)}$ is the same result as if we had taken the Frangi filter while only looking for the opposite type (light/dark) curvilinear feature. Plotting $\mathcal{V}_{\max}$ over a scale of $[-1,1]$ demonstrates an interesting effect, as shown in Fig. 33. Whereas the Frangi filter generally is not reliable in terms of accurately predicting widths of trough-like (or ridge-like) curvilinear features, by somehow combining the results of our positive and negative strains. We *can* get a sense of the width by looking at where there is a relatively strong response of opposite sign. We will develop a method of utilizing this observation in Chapter 9.

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

CHAPTER 6

FFT-BASED DISCRETE DERIVATIVES

According to Section 4.3, we may calculate derivatives of our structure by calculating a gradient on our convolved image. Our method of calculating the gradient of a matrix uses a second-order accurate central difference, as in [24].

We note in passing that we may take the derivative of the Gaussian kernel and then convolve it, and the effect will be the same as if we had taken the derivative subsequently [6]. This could offer some computational speedup if we wish to run this procedure on many samples and fixed scale sizes. For the time being, we will convolve first, although our method will differ from standard convolution.

6.1. Fourier Transforms

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

6.1.1. Fourier Transform of a Continuous 1D Signal

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

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

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

$$F(\mu) := \mathcal{F}\{f(t)\} = \int_{-\infty}^{\infty} f(t)e^{i2\pi\mu}dt \tag{6.2}$$

An inverse transform will then recover our original signal:

$$f(t) = \mathcal{F}^{-1}\{F(\mu)\} = \int_{-\infty}^{\infty} F(\mu)e^{i2\pi\mu t}dt \tag{6.3}$$

Together, Eq. (6.2) and Eq. (6.3) are referred to as the *Fourier transform pair* of the signal $f(t)$.

6.1.2. Fourier Transform of a Discrete 1D signal

We wish to develop the Fourier transform pair for a discrete signal., following [6]. We frame the situation as follows: a continuous function $f(t)$ is represented as the sampled function $\tilde{f}(t)$ by multiplying it by a sampling (or impulse) function, an infinite series of discrete impulses with equal spacing ΔT:

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

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

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

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

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

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

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

6.1.3. 2D DFT Convolution Theorem

Theorem 6.1 (2D DFT Convolution Theorem)**.** *Given two discrete functions are sequences with the same length. $f(x,y)$ and $h(x,y)$ for integers $0 < x < M$ and $0 < y < N$, we can take the discrete fourier transform (DFT) of each, where $\mathcal{D}\{\cdots\}$ denotes the DFT.*

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

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

and given the convolution of the two functions

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

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

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

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

Proof.

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

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

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

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

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

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

$$= MN\cdot\mathcal{D}^{-1}\{FH\} \tag{6.17}$$

where

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

□

Above, we make use of the following lemma

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

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

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

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

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

$$z^N - 1 \;=\; (z-1)\left(z^{n-1} + \cdots + z + 1\right) \;=\; (z-1)\sum_{n=0}^{N-1} z^n. \tag{6.20}$$

thus giving us

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

To prove the claim in Eq. (6.19), we consider two cases: First, if $j-k$ is a multiple of N, we of course have $e^{2\pi in(j-k)/N} = \left(e^{2\pi i}\right)^{n(j-k)/N} = 1$ and thus the left side of Eq. (6.19) reduces to

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

.

In the case that $j-k$ is *not* a multiple of N, we refer to Eq. (6.21). The first factor is not zero since, $\left(e^{2\pi i(j-k)/N}\right) \neq 1$ (simply since $(j-k)/N$ is not an integer), and thus it must be that the second factor is 0:

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

We can combine these two cases by invoking the definition of Eq. (6.18), giving us the result. □

6.2. The Fast Fourier Transform

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:

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

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

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

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

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

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

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

To simplify our notation we will use a new notation for the Fourier kernels/basis functions. Let the 1D Fourier transform be given by

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

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

We use this to rewrite the series in Eq. (6.26) and split it into odd and even entries in the summation

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

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

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

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

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

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

Thus we can rewrite Eq. (6.28) as

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

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

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

transform K frequencies later :

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

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

Comparing Eq. (6.33) and Eq. (6.35), we see that the expressions within parentheses are identical. What's more, these parentetical expressions are functionally identical to discrete fourier transforms themselves. Let's notate them as follows:

$$\begin{aligned} \mathcal{D}_u\{f_{\text{even}}(t)\} &:= \sum_{x=0}^{K-1} f(2x) W_K^{ux} \\ \mathcal{D}_u\{f_{\text{odd}}(t)\} &:= \sum_{x=0}^{K-1} f(2x+1) W_K^{ux} \end{aligned} \tag{6.36}$$

If we're calculating an M point transform (i.e. we're wishing to calculate $F(1),...,F(M)$), once we've calculated the first K discrete frequencies (i.e. $F(1),\cdots,F(K)$) we may simply reuse the two values we've calculated in Eq. (6.36) to calculate the next $F(K+1),..,F(K+K) = F(M)$. Since each expression in parentheses involves K complex multiplications and $K-1$ complex additions, we are effectively saving $K(2K-1)$ calculations in computing the entire spectrum $F(1),..,F(M)$. When M is large, the payoff is undeniable.

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

Of course, since Eq. (6.36) are DFTs themselves, there's nothing stopping us from reiterating this procedure; if M is substantially large, we can just as easily repeat this process a few times.

Of course, our development was for $1D$. We can extend this to $2D$ by taking note of Eq. (6.22).

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.

6.3. Calculating the Hessian via FFT: A demonstration

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

In Fig. 7, we demonstrate the compatibility of standard convolution and FFT convolve. Each row corresponds to a different scale at which Gaussian blurring occurs. Column (a) is standard convolution with a sampled Gaussian kernel, column (b) is FFT-convolution with a Gaussian kernel, and column (c) is a FFT-convolution with the "discrete Gaussian kernel". In column (d), the 1D discrete Gaussian kernel (in green) is plotted against the sampled continuous Gaussian kernel (in black). Note that each of the images in the first three columns are scaled the same.

In Fig. 8, we show these same three methods of Gaussian blur but for a large scale ($\sigma = 45$). For each method of taking the Gaussian blur ((a) - standard convolution with sampled kernel, (b) FFT with sampled kernel, (c) FFT with discrete kernel), the top row is one round of Gaussian blur with $\sigma = 45$ and the bottom row is two progressive passes of Gaussian blur ($\sigma_1 = 10, \sigma_2 = 35$). The mean squared error and mean absolute error between the one-pass and two-pass versions are outputted in Table 1. 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 noticable after several iterations. Here, we've opted to crop out a radius of pixels from around the edges equal to the standard deviation of the Gaussian before we calculated the MAE or MSE, to reduce noise from the border.

We further confirm the commensurate nature of Gaussian blur techniques by comparing

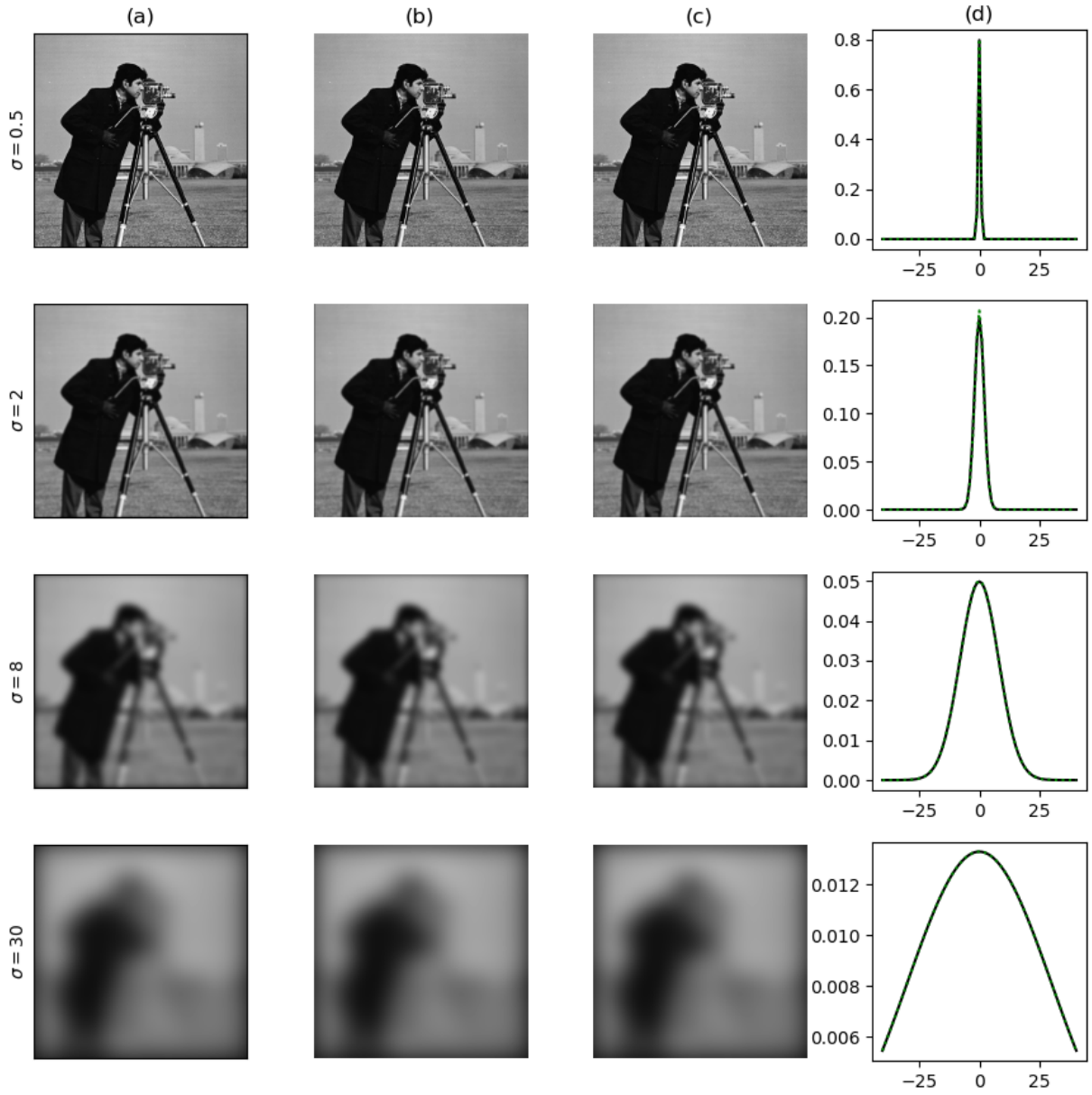


FIGURE 7: Compatibility of Gaussian convolution strategies

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

TABLE 1: Errors between different Gaussian blur implementations

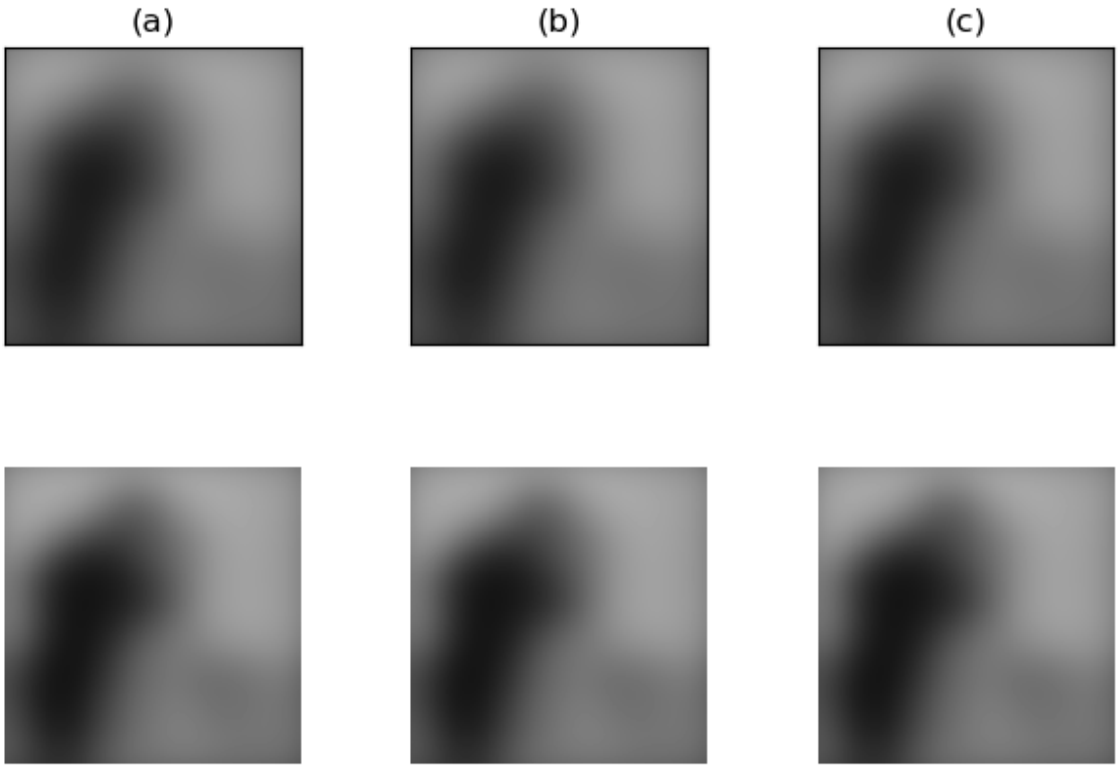


FIGURE 8: Iterative Gaussian blur

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

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

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

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

	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 $\sigma = 5$

the three techniques on a placental image and using each to calculate Frangi targets. The code can be found in `hfft_accuracy.py`. In Tables 2 to 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 grayscale intensity of the image from 0 to 1 and the Frangi targets themselves across an arbitrarily chosen horizontal cross section of the image Fig. 9, 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 (2200×2561) and timed each method of convolution (average of three trials) for a large number of samples: logarithmic between $\sigma = 1$ and $\sigma = 128$ with 32 steps. We plot the result in Fig. 10. It 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. This is to be expected, as convolving with a Gaussian kernel in spatial coordinates requires a greater number of calculations as σ increases, whereas the size of

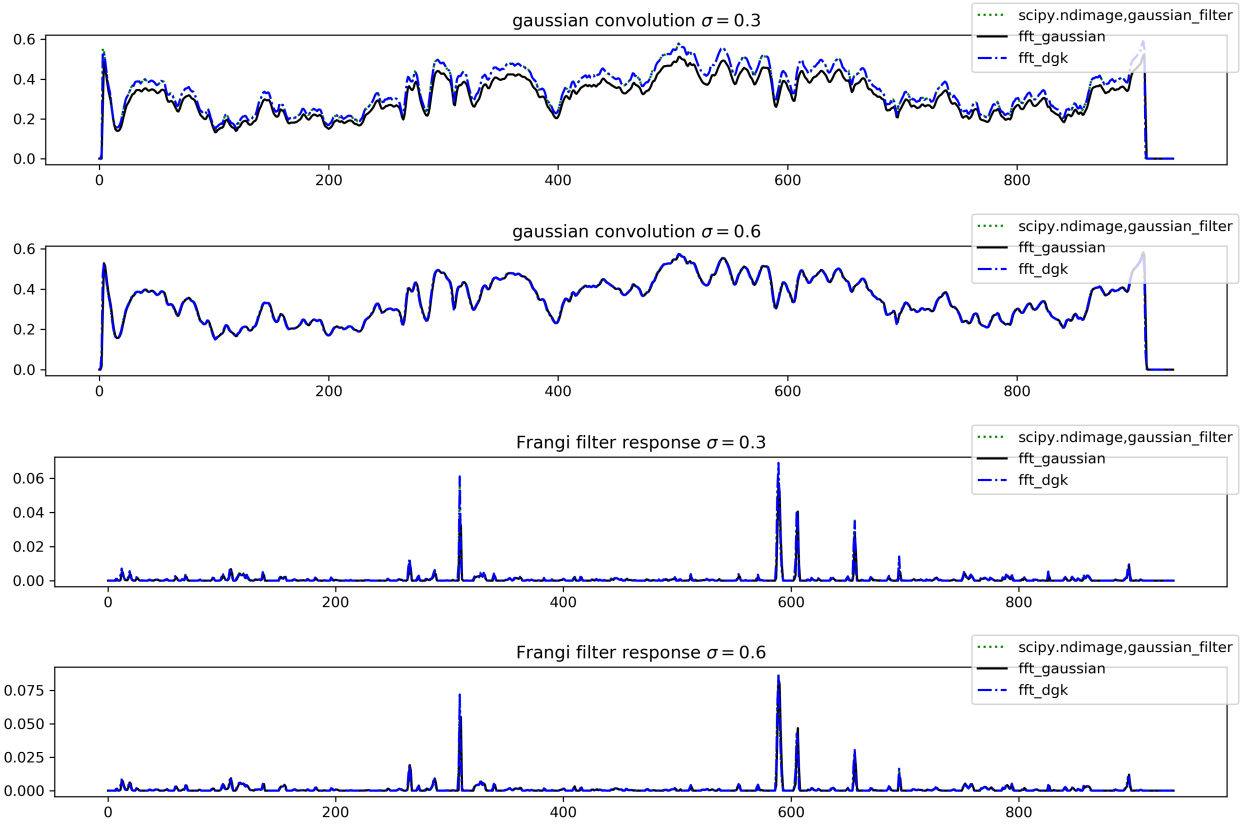


FIGURE 9: Image cross-section of Gaussian-blurred (grayscale) placental sample

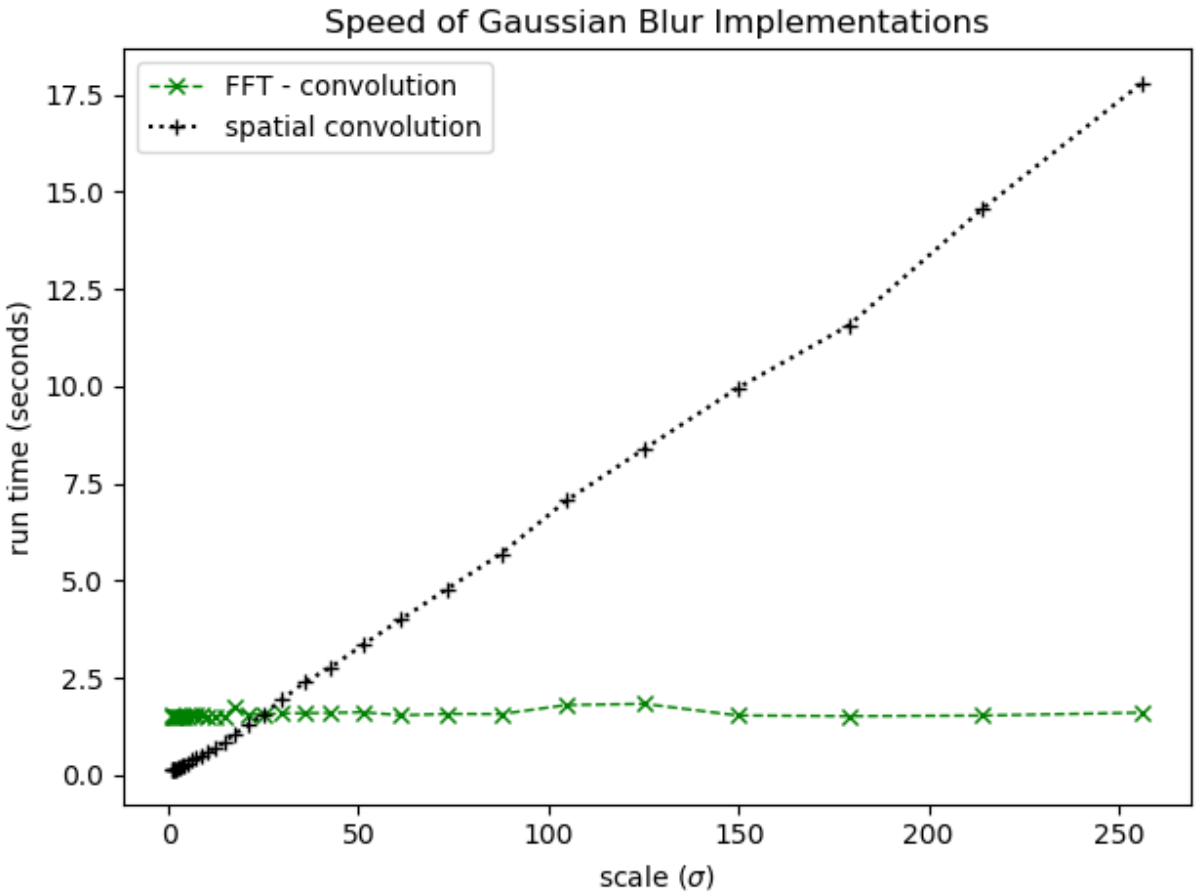


FIGURE 10: Runtime comparison of Gaussian convolution implementations

the kernel does not change in our frequency space convolution.

CHAPTER 7

RESEARCH PROTOCOL

7.1. Samples / Image Domain

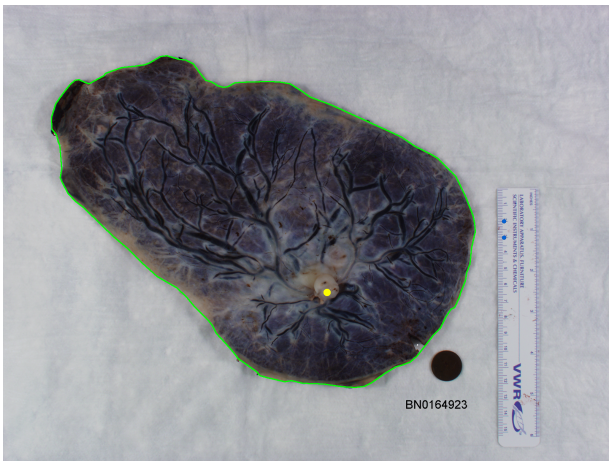
We ultimately perform a multiscale Frangi prefilter on a subset of 201 color images of placental samples from a private database provided by the National Children's Study, which had been prepared for a different study. A detailed description of the data set is given in [1], and a description of the cleaning and fixing procedure is given in [5]. The samples are provided as XCF files (the native project file for GIMP) and contain four major layers.

7.1.1. A representative sample

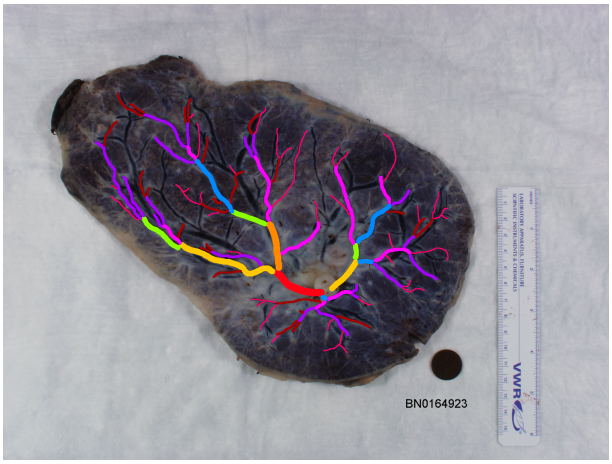
The layers together give a hand tracing of the vascular network and perimeter. A sample of overlaid layers in a representative sample (with ID number "BN0164923") is given in Fig. 11.

Each layer is roughly 1954x1200 pixels (with some occasional variation). In Fig. 11, we see these four layers of a characteristic sample. Fig. 11a is the base image. A cleaned, fixed placenta is placed on a table with a camera a fixed distance away, and a ruler and penny (presumably for redundancy) are placed nearby to aid registration and calibration of the resolution. The resolution of each sample is roughly 46 pixels per centimeter. Fig. 11b 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. 11c and Fig. 11d are each hand traces of the PCSVN, with a layer for each the arteries and veins. These layers are simultaneously overlain on the base image inFig. 11e. The coloration is meant to indicate the diameter of each vessel. The diameters are binned into 9 discrete widths, odd integers from 3 to 19 pixels. Vessels of smaller diameter are either binned to three or (quite frequently) left untraced. The correspondence between pencil

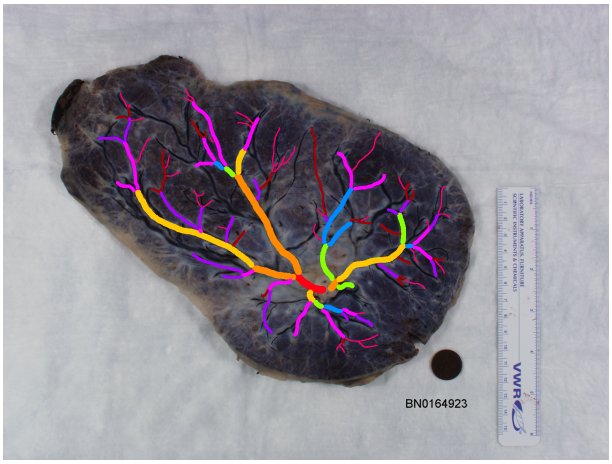
(a) Fixed Placental Sample



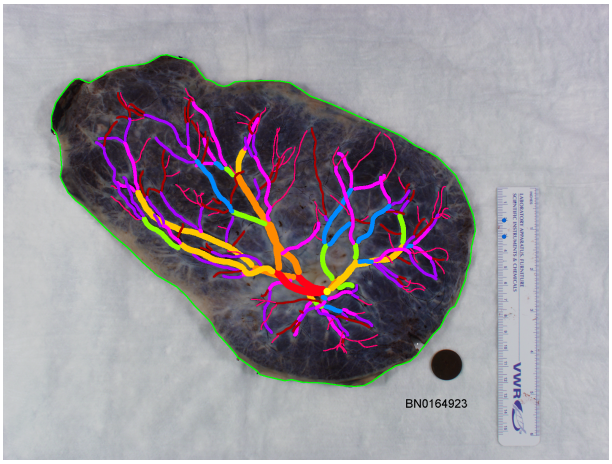
(b) Perimeter tracing



(c) Arterial tracing



(d) Venous tracing



(e) Total Vascular Network

FIGURE 11: A representative placental sample and tracing

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

TABLE 6: Vessel width color code for manual tracing protocol

color and (binned) vessel width used in the tracing protocol is given in Table 6.

As stated in the introduction, the task of creating these samples, in particular the tracing and estimation involved in creating Fig. 11c and Fig. 11d is very labor intensive–requiring between 4 and 8 hours to trace a single sample. A closer look at many of the samples often reveals that a great deal of subjectivity in providing this "ground truth," as it is not often clear what the underlying truth really is. Often it's hard to see where the vein is, vascular networks are obscured by the umbilical stem, the blood in the vessels dries unevenly or ruptures, and the vessel seems to disappear momentarily. These situations and more will be showcased in Fig. 20, where we will demonstrate how the Frangi filter reacts to these problem areas. Our efforts at the eventual task of network completion must deal with these shortcomings and, in some circumstances, make subjective decisions like the manual tracer did.

7.1.2. Knowns and Unknowns

Since our final goal is a fully automated procedure, we wish to simply operate on the placental sample itself, without any understanding of its provided tracing (except for judging the strength of our algorithm); our goal is to develop an algorithm that can produce a "ground truth" tracing similar to Fig. 11e or Fig. 12d without any user intervention.

For our purposes however, we will concede and provide a limited amount of information from the tracings, namely the provided placental perimeter (shown in green in Fig. 11). In

developing a fully automated algorithm, it would be relatively straightforward to obtain this boundary ourselves using various techniques, such as an Active Contour Model [25] or, or even a simple edge finding algorithm followed by watershedding and largest object selection. In fact, we have implemented this later algorithm, but unfortunately we achieve a subpar result on several images, and therefore we currently use it to improve upon the perimeter by removing "cuts", which were previously reported as being inside the plate and sometimes led to large false positives.

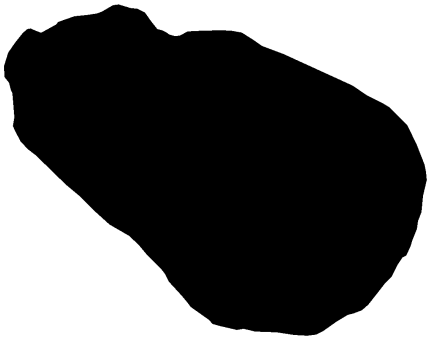
Finally, we will consider the location of the umbilical insertion point as a "known", as the vessels around it are frequently impossible to see and we wish to exclude them from consideration. It is not unreasonable, however, to consider this to be a known–in future preparations of samples, we could simply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

7.2. Data Cleaning and Preprocessing

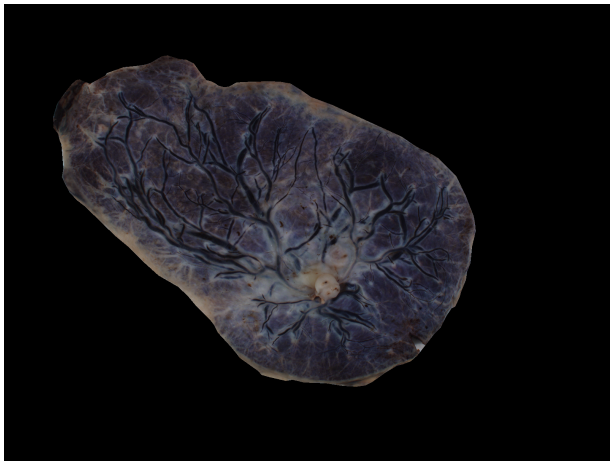
Building a sample suitable for use in our algorithm from Fig. 11 is relatively simple. We zero outside the boundary of the plate (so as to not waste computational time calculating the differential geometry of a ruler, say), and also generate a binary mask to identify the plate. Finally, our vessel layers are combined and given as a binary trace, which we will use later for scoring. An example of the preprocessed samples used by the algorithm are given in Section 7.2.

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.

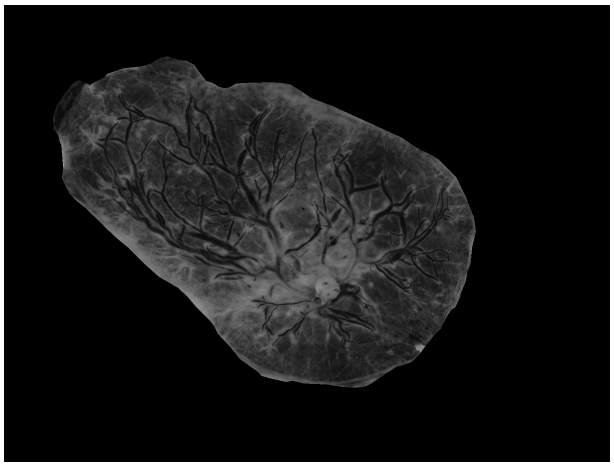
As a point of technicality, the grayscale image in Fig. 12c is not actually produced directly by the extractor plug-in, but created when the 3 channel RGB image Fig. 12b is imported at the start of the algorithm. This grayscale conversion is simply done for ease of analysis on the sample: although the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



(a) Background Mask (in white)



(b) Sample with BG removed



(c) Grayscale

(d) Trace / “Ground Truth”

FIGURE 12: Preprocessed files from an NCS sample

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:

$$L = \frac{299}{1000} R + \frac{587}{1000} G + \frac{114}{1000} B \tag{7.1}$$

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

7.2.1. Boundary Dilation

All images are grayscale, M,N pixels as a masked array (of type `numpy.ma.MaskedArray`), where pixels outside of the placental region are masked so they will not be considered by the algorithm. However, some standard implementations of algorithms, namely `numpy.gradient and scipy.signal.convolve2d` are not designed to handle masked regions. Although it would be potentially useful to adapt such methods in a way to, say, calculate a gradient or perform a convolution by a "reflection" across an arbitrary closed boundary (as opposed to the edge of the image matrix), we opted instead to "zero out" unwanted background pixels and simply exclude affected areas from consideration. This exclusion could be achieved by simply dilating the mask, but we opt to achieve it in a much more resource efficient manner: we iterate through an array of indices for the image where the boundary occurs and simply extend the mask R pixels in each direction (like a giant plus sign). Since the boundary of the placental plate forms a closed loop, the effect is very similar to convolving with a disk of radius R, but is much faster.

Fig. 13 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

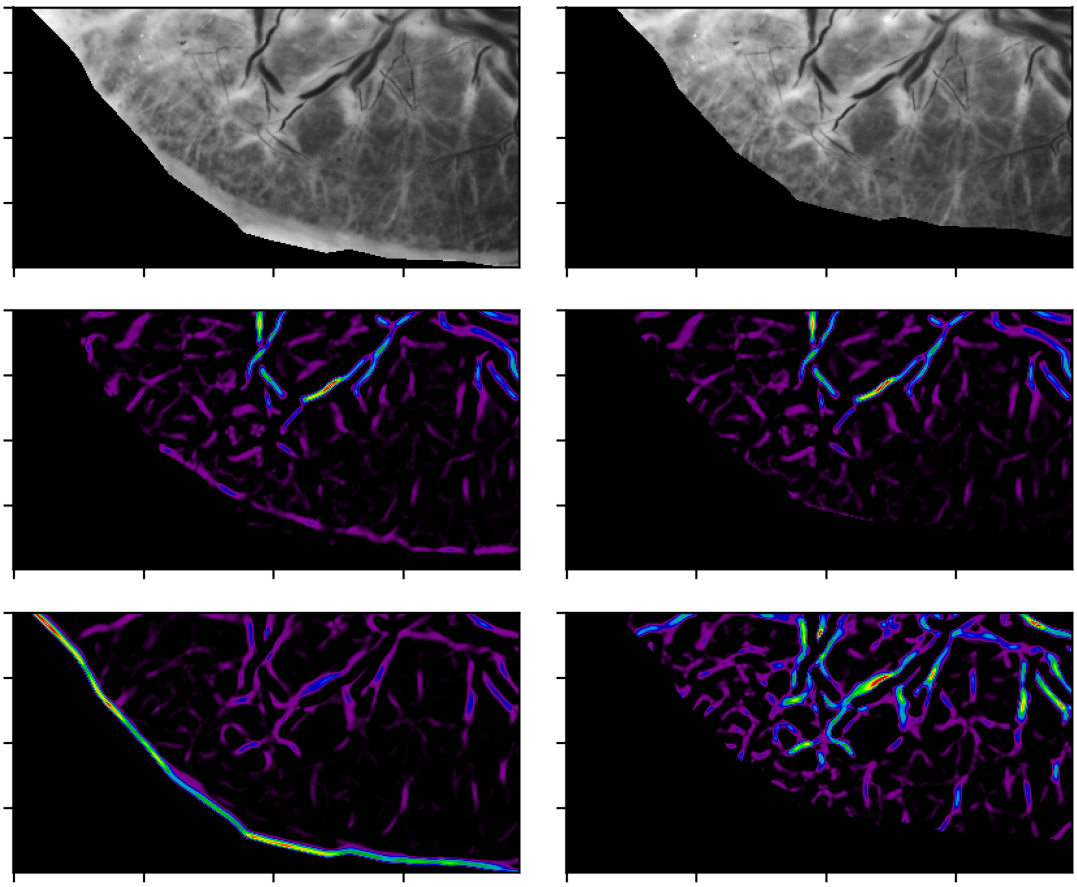


FIGURE 13: Effect of boundary dilation on Frangi responses

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. 13 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], [5]). It should be noted that, while the figure on the right shows a much larger interior response, this is simply because the intensity of the output in each of these images is being independently scaled between the minimum and maximum intensity in the image. However, we argue that this is an appropriate and desired depiction of the situation, as we will frequently consider only the relative maxima of Frangi response per scale in our analysis.

We end our discussion by noting that we perform this boundary dilation within the Frangi algorithm itself when we set the structureness parameter γ as half of the maximum Hessian norm found at that scale–this ensures that the maximum occurs sufficiently away from the boundary of the plate, and does not occur from a noise phenomenon.

The code for generating Fig. 13 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.

7.2.2. Deglaring

Despite best efforts when harvesting samples, a select number of the placental samples exhibit substantial glare, which leads to inaccuracies in identifying curvilinear content. Our protocol for deglaring is analogous to that performed in [5] and [2]. Unfortunately, the method relied upon by those previous papers (`MATLAB`'s `imfill`, which relies on inpainting by solving the

Dirichlet problem for masked regions) was not immediately available in a Python environment. Instead, we used an already implemented inpainting algorithm, `scikit-image`'s `inpaint_biharmonic()`, which should be expected to achieve similar results, albeit at the expense of processing time.

The function `inpaint_biharmonic` is based on [27], and relies on solving a biharmonic equation i.e. $\nabla\nabla f = 0$ for the surface f subject to boundary conditions (as compared to `imfill`'s solving the Laplace equation $\nabla f = 0$ in regions marked as glare.

The method for deciding what is considered glare is similar to [5], in which we consider any intensities close the maximum intensity in the image (Almoussa et al. used 80% of max intensity, and we use $175/255 \approx 68\%$). This threshold is unfortunately dependent on the image domain.

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

A comparison of these methods is shown in Fig. 14, and a zoomed in portion is shown in Fig. 15. 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

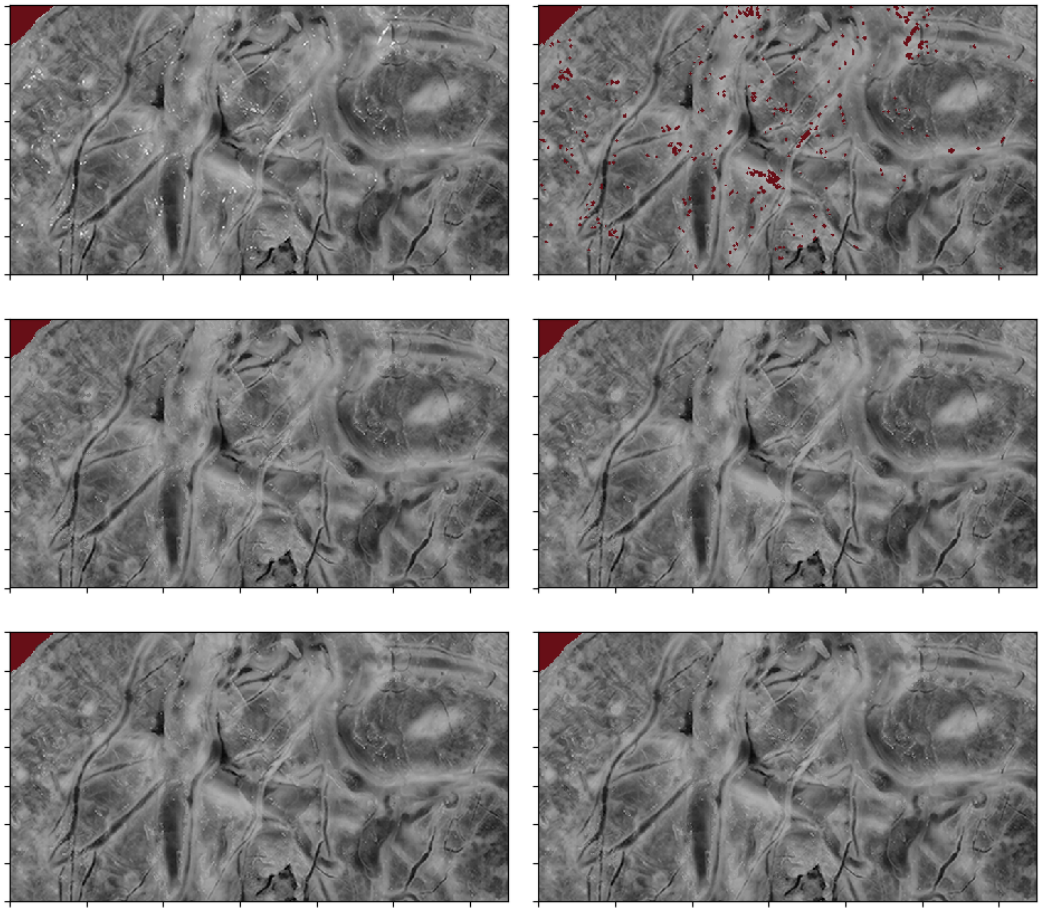


FIGURE 14: Deglaring a sample using a hybrid inpainting method

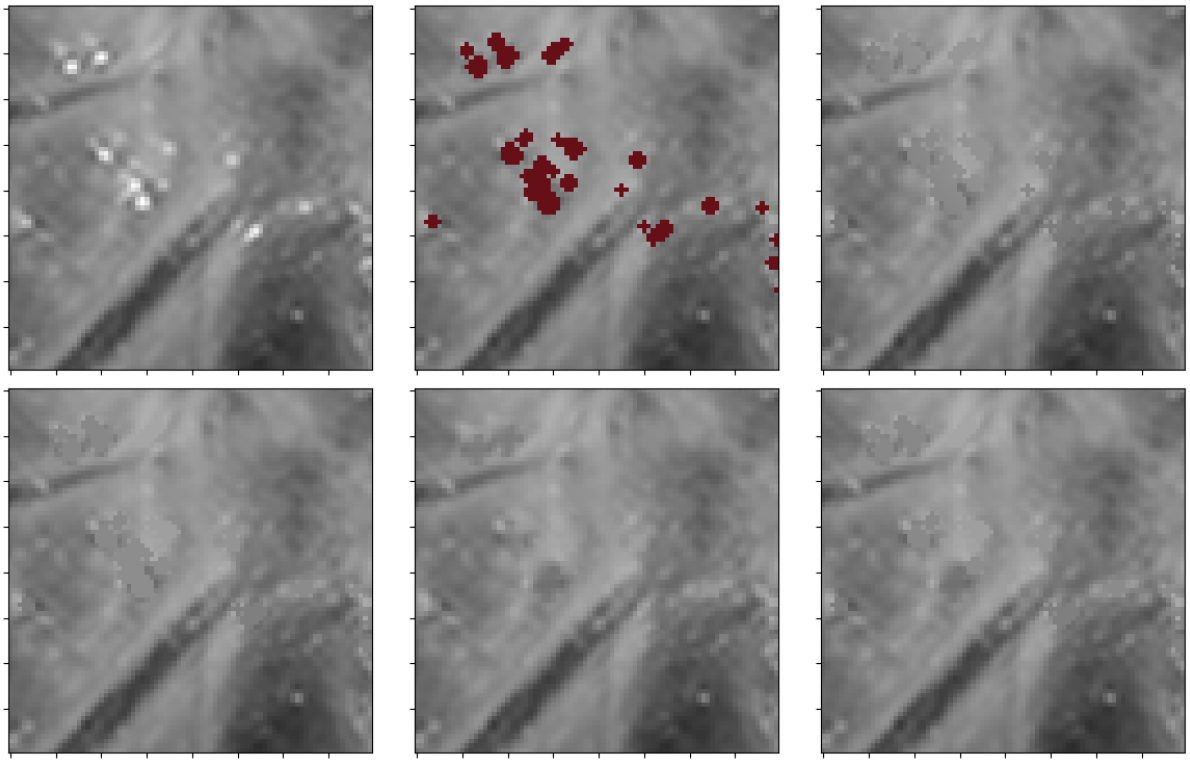


FIGURE 15: Comparison of glare inpainting methods (detail)

uses a “hybrid” strategy.

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

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

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

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

In [4]: px_inpainted = np.sum(np.logical_and(masked.mask, np.invert(img.mask)))
In [5]: px_plate = np.sum(np.invert(img.mask))
In [6]: px_inpainted / px_plate  # ratio of inpainted pixels to total plate
Out[6]: 0.011444460505513942
```

We stress again that only a small subset our image domain exhibits disruptive amounts of glare. Future improvements in this direction should probably seek to implement more robust method such as [28] that are not dependent on an arbitrary global threshold for deciding what regions exhibit glare.

7.3. Multiscale Setup

Our multiscale Frangi filter requires a list of scales at which to probe. Each scale is chosen to accentuate features (i.e. vessel diameter) of a particular size. This set of scales at which to probe will be denoted as $\Sigma := \{\sigma_1, \sigma_2, \ldots, \sigma_N\}$, where each scale in indexed by increasing order.

Although we cannot expect *a priori* that there is an direct proportionality between our scale size σ and (even some function of) the width of a particular vessel [12], we generally expect to isolate narrower curvilinear structures at smaller scales, and thicker curvilinear structures at larger scales. The smallest one should be an effective size where details are expected to be isolated, and the largest should be an effective size as well. In fact, following [16], it is reasonable and natural to select these logarithmically; that is, for some selected inputs $m < M$ we have

$$\sigma_1 = 2^m \, , \cdots \, , \sigma_j = 2^{\left(m + \frac{M-m}{N-1} j\right)} \, , \cdots \, \sigma_N = 2^M \tag{7.2}$$

That is, the exponents are spaced linearly from m to M. This is achieved by the command `np.logspace(m,M,num=N)`. The idea is that the curvilinear content of the image will respond better at some particular scale, but there are diminishing returns as σ increases; while the filter's response may vary substantially between, say $\sigma = 1$ and $\sigma = 2$, we would not expect a substantial difference in response between, say, $\sigma = 46$ and $\sigma = 47$. Historically, there was another benefit of using a logarithmic scale space: computing the vesselness measure was very expensive, and thus it was simply not feasible to collect so many large scale readings. This is much less of an issue with the present implementation, although we still obviously wish to avoid frivolous, redundant calculations no matter the speed of the implementation.

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

7.4. The Research Protocol

Once we have chosen this set of scales Σ, we simply convolve the image with a discrete Gaussian kernel with that standard deviation, then take gradients enough to get a matrix of partial second derivatives, the Hessian. We calculate the eigenvalues of each (2x2) Hessian matrix and then compute the Frangi filter according to **??** and **??**. We use these to provide a couple examples of estimating the PCSVN network. The entire decision tree can be shown in the outline below. Indentations with "+" and "." characters are lists of options at that point, where "+" is the default and "." is for alternatives discussed elsewhere in the text.

```
% DECISIONTREE
```

```
For each sample:

A) Preprocessing
  1) RGB to single channel via Luminance Transform
  2) Cut removal
  3) Remove glare
    a) Mask glare
      + Threshold (*175/255*)
      . Threshold at *80%* of max intensity (Almoussa)
      . Lange (2005) (multistep procedure, done in RGB space actually)
    b) Post-process mask
      + Dilate with radius *2*
      . Do nothing
    c) Inpaint glare
      + Hybrid inpainting, with size threshold *32*
      . Biharmonic inpainting
      . Mean value of boundary
      . Median value of boundary
      . Windowed mean (radius: *15*)
B) Multiscale Frangi filter
  1) Define parameters
    a) Scales
      = n_scales (default: *40*)
      = scale_range (default *[-2, 3.5]*)
      = scale_type (*logarithmic base 2* or linear or custom)
      -> build scales
    b) Betas
      = *0.5* each scale or custom range
    c) Gammas
      = strategy: (half L2 hessian norm or *half hessian frobenius norm*)
                  or custom value each scale
      = redilate plate per scale (?)
    d) Dilate per scale
      + Custom function of scale
         (default *max{10, int(4sigma)} if (sigma < 20) else int(2*sigma))*)
      . No dilation
    e) Scale space convolution method
      + Discrete Gaussian kernel with FFT
      . Sampled gaussian kernel with FFT
      . Sample gaussian kernel, standard convolution
  2) For each sigma: do Uniscale Frangi Filter
    a) gauss blur image with method from (1e)
    b) take gradient across each axis, take gradient across each axis
    of gradient to get Hxx, Hxy, Hyy
    c) find eigenvalues of hessian at each point (using np.eig)
       and sort by magnitude
```

```
    d) zero out principal directions according to Dilate Per Scale
    e) zero out hessian according to max(ceil(sigma),10)
    f) Calculate Frangi Vesselness Measure
C) Estimate PCSVN
  1) Approximate using strategy
    a) Calculate Fmax and Fmax.where -> Fmax
    b) Threshold at 95th percentile -> approx
    c) Threshold at fixed alpha
    d) Margin adding to one of the above
    e) Random walker after margin adding
  2) Compare to Trace
  3) Calculate Network Coverage and MCC score
```

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

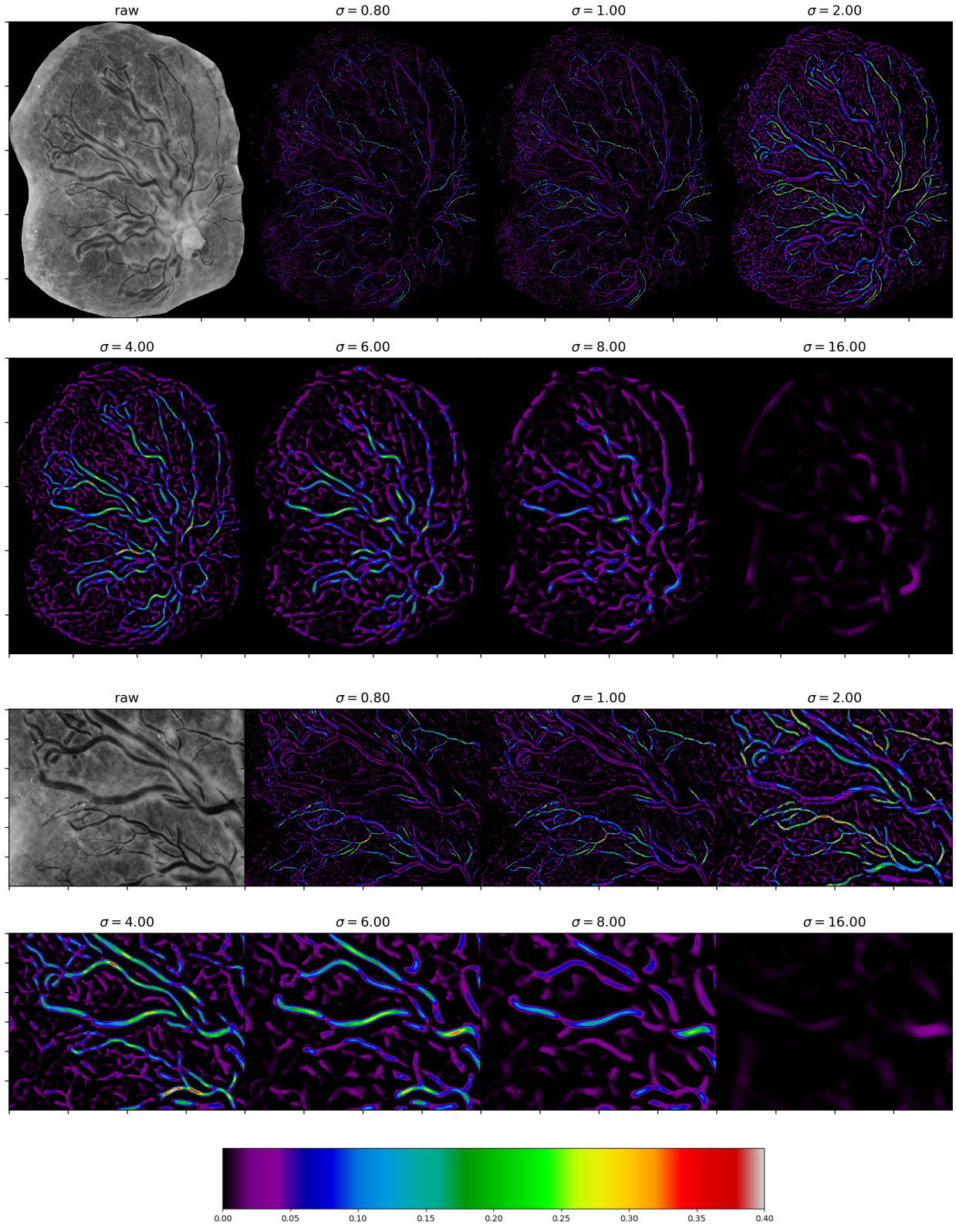


FIGURE 16: Example scalewise Frangi output (plate and inset) (Example 1)

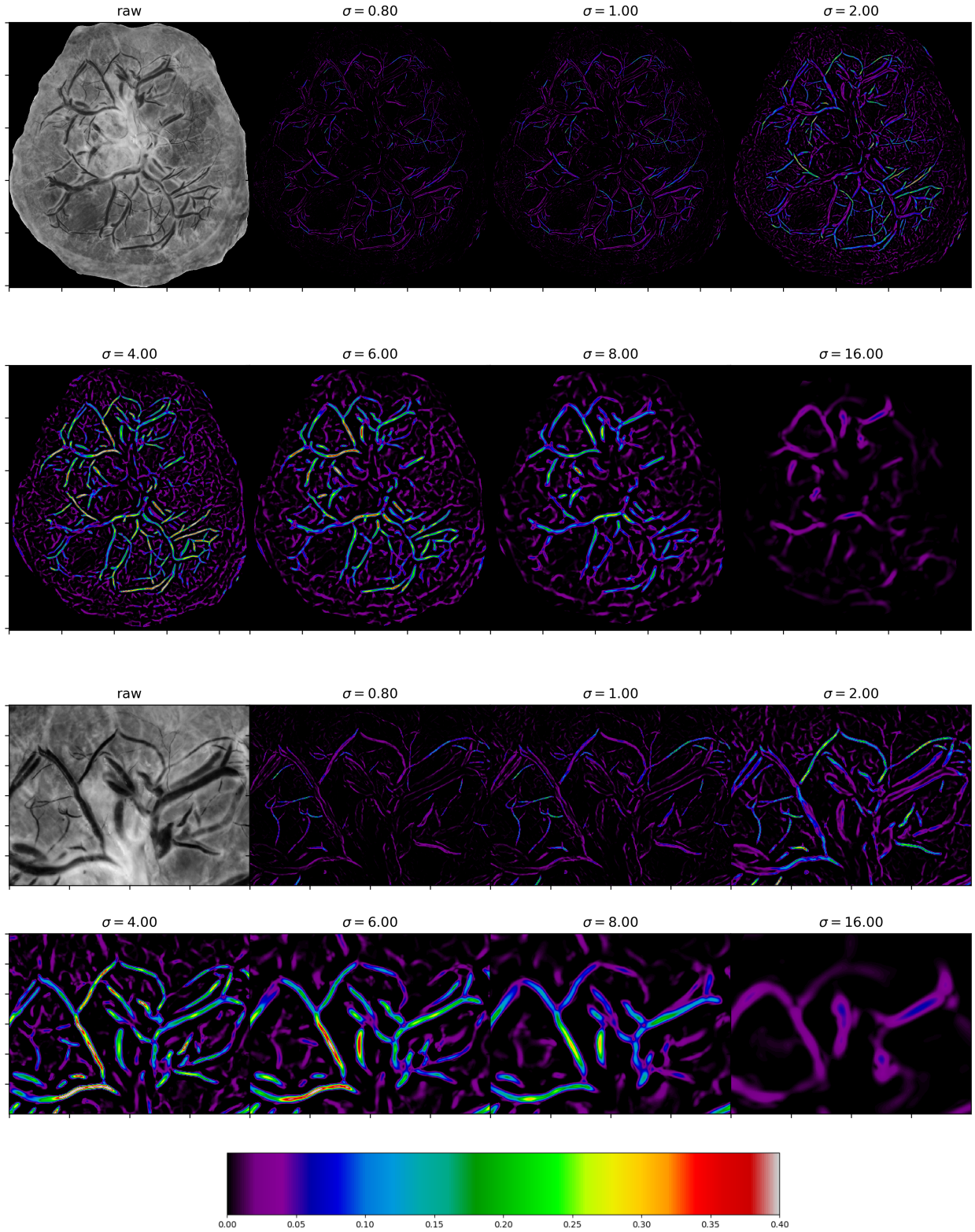


FIGURE 17: Example scalewise Frangi output (plate and inset) (Example 2)

CHAPTER 8

RESULTS AND ANALYSIS

We demonstrate the output of the Frangi filter on our samples after running a multiscale technique with $N = 20$ scales with a stricter anisotropy parameter $\beta = 0.35$ and standard structureness parameter $\gamma = 0.5$, with scales spaced logarithmically from $\sigma_1 = 2^{-1}$ to $\sigma_N = 2^{3.5}$, performing glare and cut removal in preprocessing, and using a discrete gaussian kernel and dilation border of 20. Our goal in this section is to provide a close up look at the Frangi filter on two samples, and then provide some measures of the Frangi output's correspondence with the "ground truth" network tracings across all 201 images, without explicitly performing segmentation. However, for visual demonstration, we will employ both simple thresholding techniques (arbitrary fixed and nonzero-percentile) described in Chapter 5. In Chapter 9 we will develop and analyze more sophisticated Frangi-based segmentation techniques and compare their performance to these rudimentary thresholding techniques across the entire dataset. For this chapter we will content ourselves with analysis of the raw Frangi vesselness measure within our image domain, and use our simple thresholded results for visual demonstration alone.

8.1. Sample Visual Output

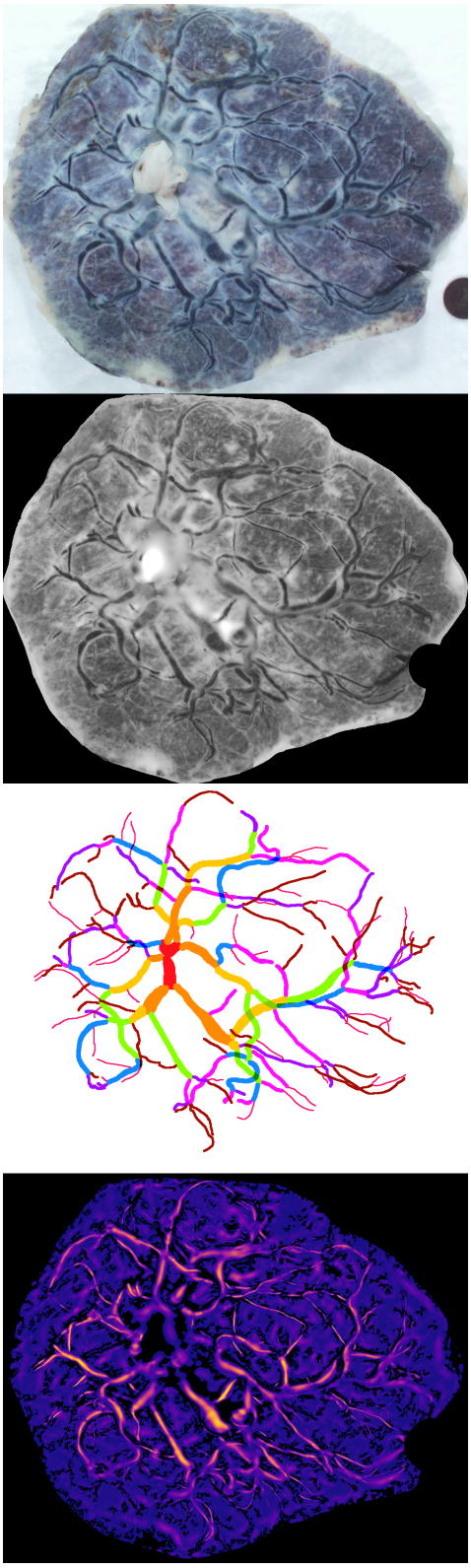
In Fig. 18 and Fig. 19 we take a look at the Frangi output ($\mathcal{V}_{\max}$) for a well-behaved sample. In the left column we have (from top to bottom) the un-preprocessed placental sample, the preprocessed sample, the color trace(ground truth), and $\mathcal{V}_{\max}$. In the top-right, we have for each $n = 0,\ldots,19$ scales, we have the size of the scale σ_n, the 95th percentile score at that scale α_p where $p = 95$, and the maximum vesselness score at that scale ($\max(V_\sigma)$. The bottom-right set of images are two rudimentary thresholds of the Frangi result. The top is the scalewise nonzero percentile threshold (95th percentile) and the bottom is a strict threshold of 0.4. left and right images are two

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

8.2. Ground Truth Imperfections

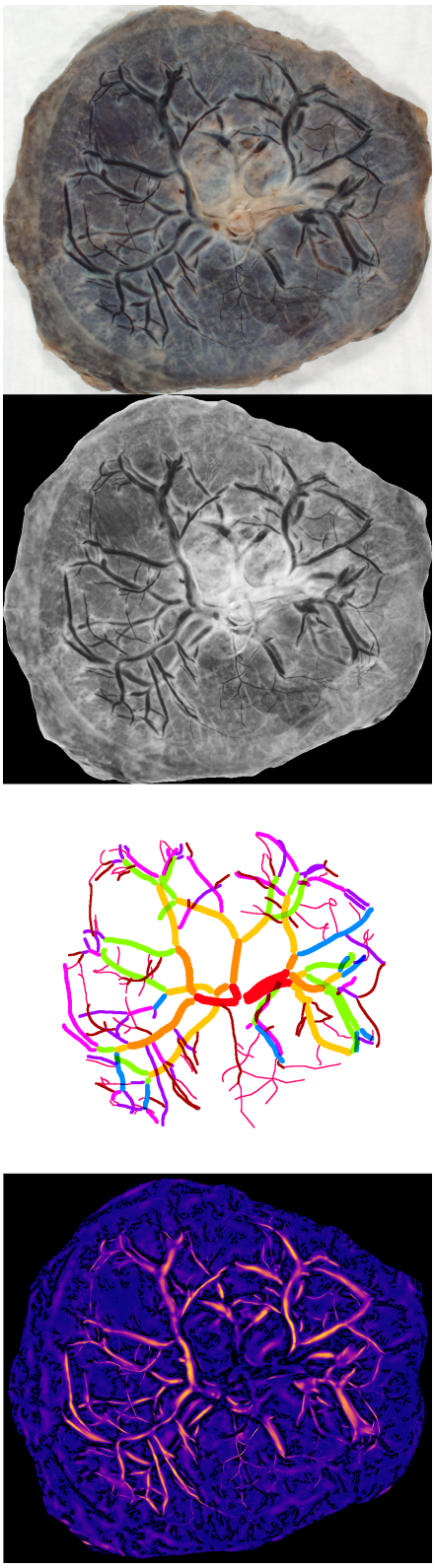
We must first qualify binary classification. There are limitations intrinsic to the image domain and tracing protocol that make our ground truth tracing somewhat inaccurate. In Fig. 20 we demonstrate a few common issues with the samples. The four figures show (top left) the original colored raw sample, (top right) the ground truth tracing, (bottom left) $\mathcal{V}_{\max}$, and (bottom right) the confusion matrix after some segmentation strategy. The green arrow points to the umbilical stump. You can see there is circular noise around this point, and in general perfusion around this point is very low–many of these vessels have been estimated by the tracer. The orange arrows represent points where perfusion is very low, perhaps caused by a clot in the blood vessel. The pink arrows point to vessels that were not traced, although they are clearly visible. The blue arrows point to where the shape of the vessel and the trace clearly do not agree. There any many more examples of these in the frame, and many more across all samples. The red arrow points to an issue unrelated to the ground truth, but an issue that arise when selecting scales in our multiscale methodology– this is a point of dark curvature that represents noise at larger scales. As you can see from the $\mathcal{V}_{\max}$ of this inset, there is a positive response in this dead space between vessels, and there is another one that appears in the bottom left between two close vessels of similar size.

The sample we examined was a relatively well-behaved sample compared to others. Conversely, in Fig. 21, we give examples of some sampled that did not respond well to the Frangi filter, nor any of our Frangi-based segmentation methods. As can be seen from this gallery, poor perfusion of the blood vessels and large amounts of high constrast background is the common



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

FIGURE 18: Vesselness score, percentile thresholds, and simple thresholds, Example 1



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

FIGURE 19: Vesselness score, percentile thresholds, and simple thresholds, Example 2

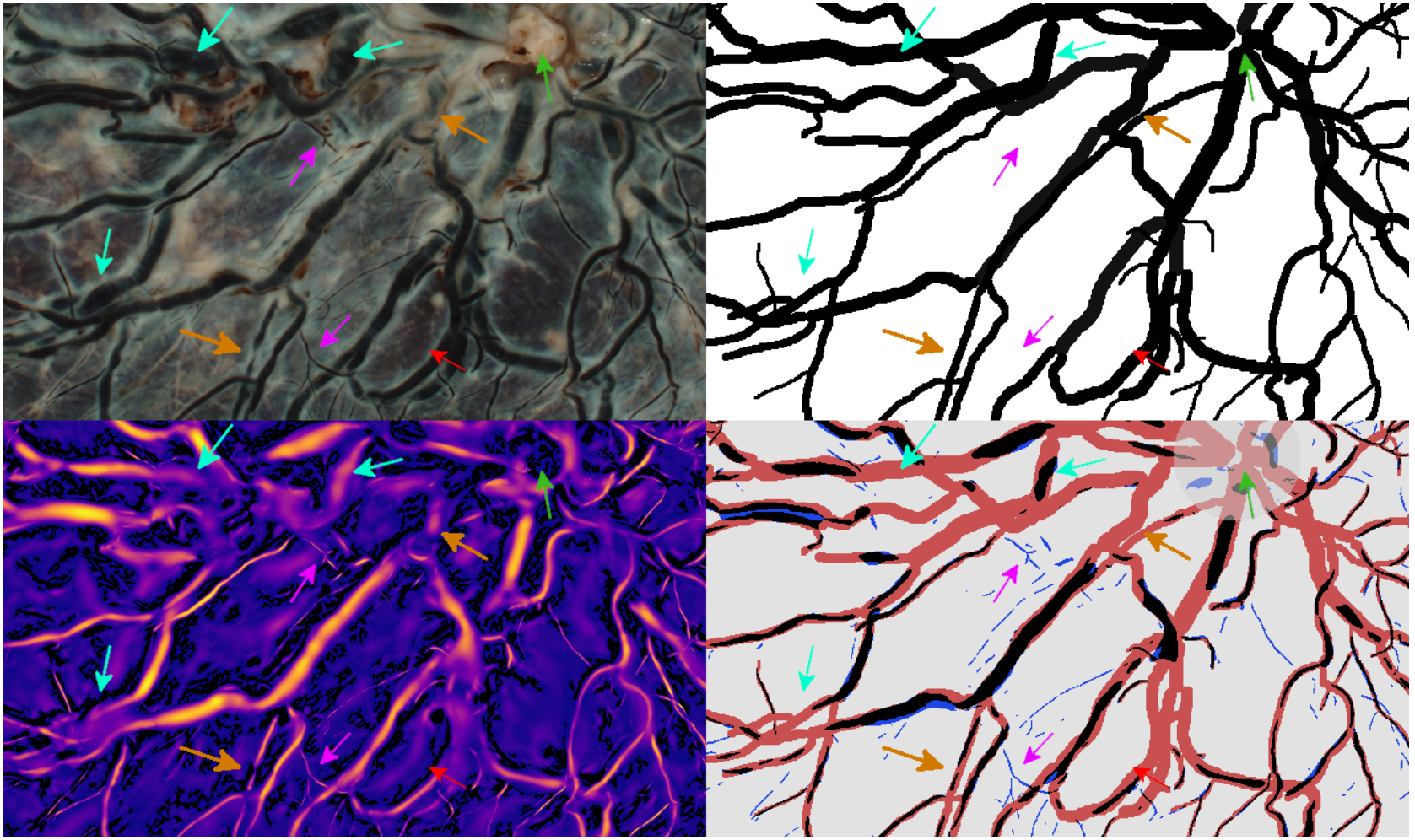


FIGURE 20: Issues with the ground truth manifesting in Frangi vesselness scores

thread. One particular sample has very large arteries that were not reported. Increasing the scale size might fix the issue for this particular sample.

We have empirically identified a few reasons for issues. Although we have in many cases removed border areas from consideration where they would cause a large false response (see Section 7.2.1 for our discussion on boundary dilation), some samples still demonstrate noise around the "collar" of each plate. This can marginally be seen in Fig. 19–there is a large Frangi response in the bottom left that was not removed, and thus shows up in each simple segmentation.

8.3. Results

In Fig. 22, we demonstrate the usefulness of scricter parameters for the Frangi filter. Since the Frangi vesselness measure is a "probability-like" score, we should be interested–without doing any actual segmentation yet–to what extent this score aligns with the ground truth in a cumulative sense. We should hope at least that larger values of our vesselness measure should occur along vessels and not within the background. We define the cumulative vesselness ratio by "integrating" the max vesselness score over pixels in the ground truth and over the entire image and considering the ratio:

Definition 8.1. *The cumulative vesselness ratio for a particular parametrization of the multiscale Frangi filter is given by*

$$CVR(\mathcal{V}_{\max}) := \frac{\sum_{G \subset \mathtt{I}} \mathcal{V}_{\max}(x_0, y_0)}{\sum_{\mathtt{I}} \mathcal{V}_{\max}(x_0, y_0)} \tag{8.1}$$

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

Fig. 22 and Fig. 23 show $\mathcal{V}_{\max}$ for two well-behaved samples with reported CVR for 9 various combinations of β and γ. Two combinations in particular, labeled "Anisotropy Factor" and "Structureness Factor," have parameter choices so that the structureness and anisotropy factors (respectively) are exactly one throughout the entire image. Comparing all parameter combinations, we see in both cases that stricter parameters (smaller β and larger γ) correspond to

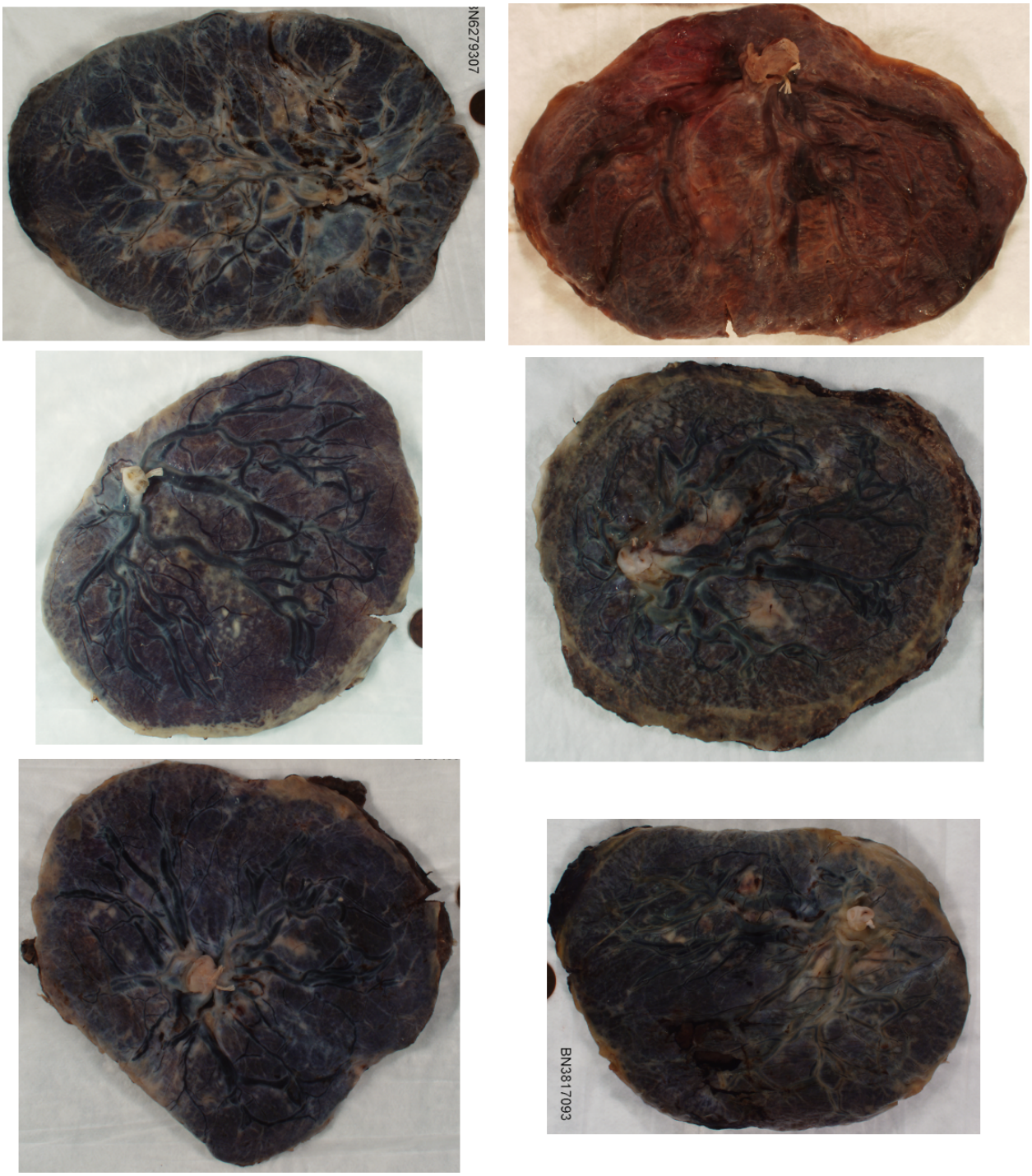


FIGURE 21: Bad samples

an increased CVR. That is, the more selective our Frangi filter is, the better it aligns with the vascular content within the image. This suggests not only that the Frangi filter is well suited to this image domain, but also that choosing stricter parameters for the Frangi filter concentrates the Frangi response within vascular features more and reduces background noise. Each of these sample was run over the same scales (logarithmically spaced from $2^{-1.5}$ to $2^{3.5}$ with $N = 20$ and the color output of each $\mathcal{V}_{\max}$ is independently normalized between 0 and 1.

We wish to see if the observations in Fig. 22 and Fig. 23 are common to our entire sample set. In Fig. 24 we show how the CVR is affected for 25 of the "best" samples, that is, those that performed better in segmentation techniques rekative to other samples, regardless of Frangi parametrization or particular segmentation method. This shows that the results of Fig. 22 and Fig. 23 hold in general for all "well-behaved" samples in our image domain.

Fig. 25 depicts, for pixels identified as vascular within the ground truth of a particular sample, the relationship between the scale at which $\mathcal{V}_{\max}$ occurs for a particular pixel and the width of the vessel according the ground truth, and the bottom figure is a coloration of the ground truth with this same information. We can see that in general, larger scales correspond with larger reported vessel widths, and smaller scales correspond with smaller reported vessel widths. In Fig. 26 we repeat these same observations but limit ourselves only to those pixels identified in the ground truth for which some $\mathcal{V}_\sigma$ occured above some scale's 95th percentile (as will be explained in Chapter 9, these pixels are the "true positives" of the nz-percentile segmentation method). The y-axis of each of the charts is the ratio of pixels at which $\mathcal{V}_{\max}$ is attained at the given width over all scales for that particular reported width.

Fig. 27 is a recontextualization of Fig. 26, where the total number of pixels (rather than a ratio) is considered for the same sample. We see that many more vascular pixels are identified as having width 7, and that in general we cannot provide a completely faithful mapping of reported vascular width to scale of largest response, but the trend of larger widths having larger scale responses and small widths having smaller scale responses remains true.

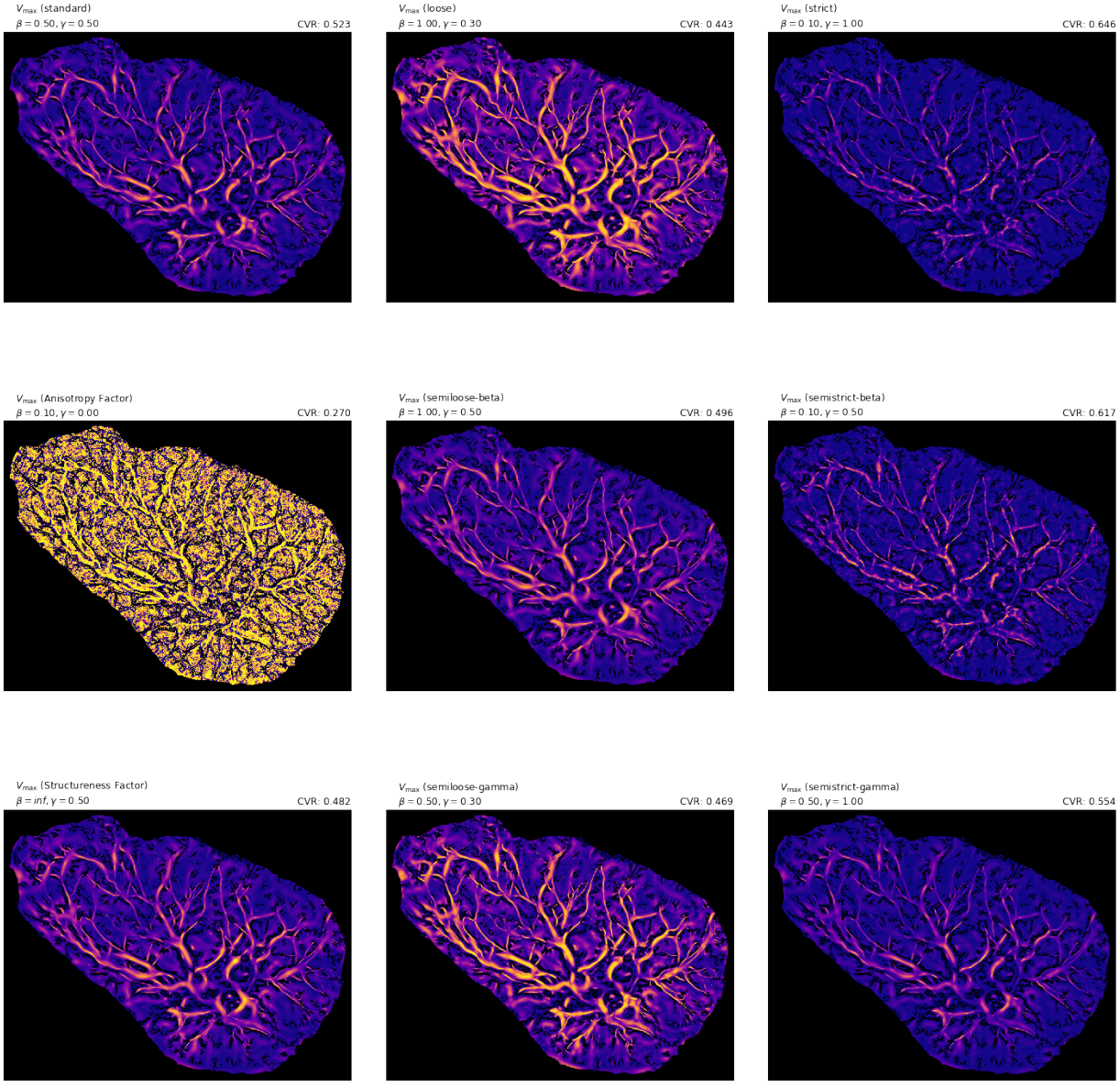


FIGURE 22: $\mathcal{V}_{\max}$ and CVR for varying multiscale Frangi parametrizations (Example 1)

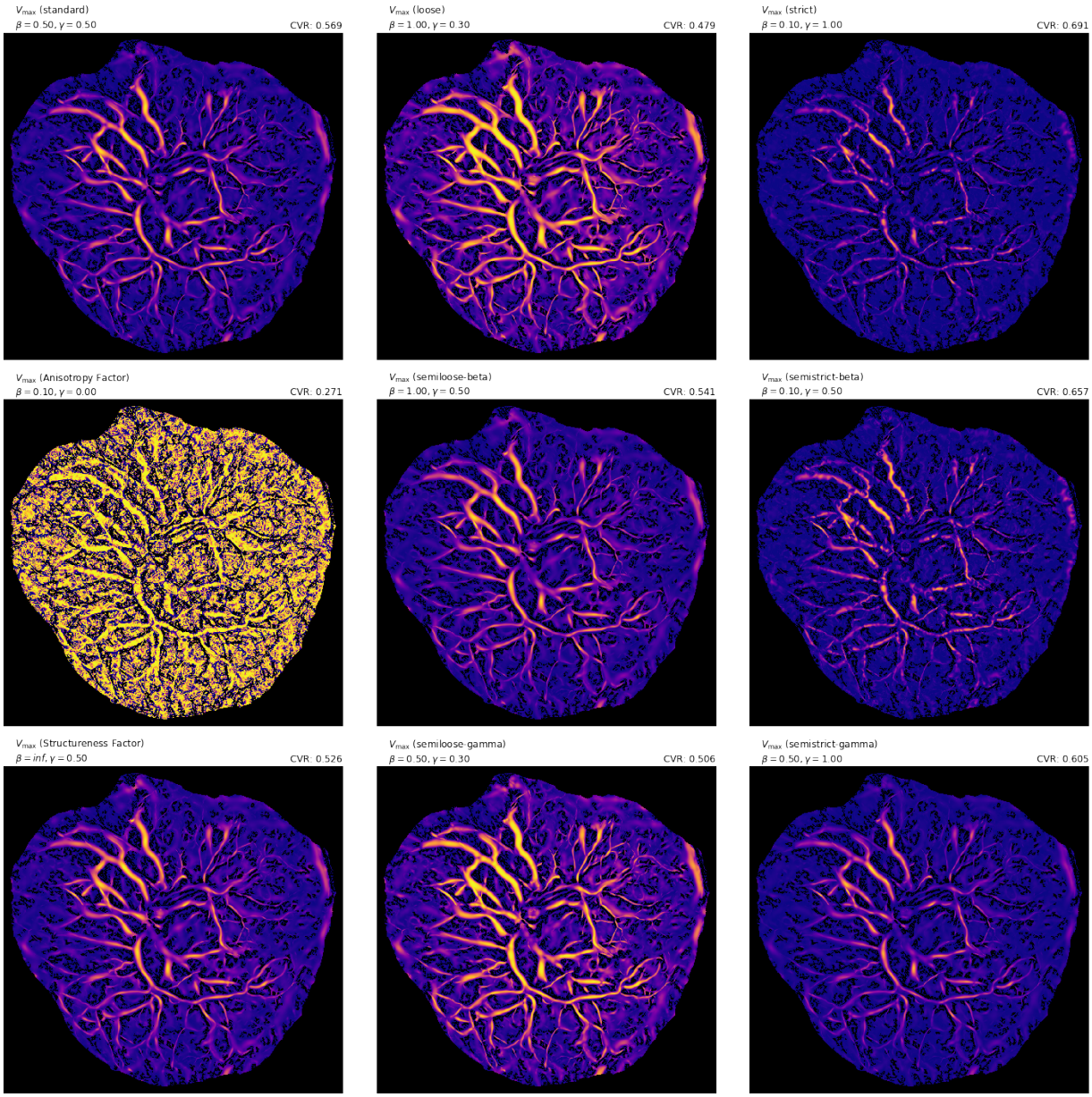


FIGURE 23: $\mathcal{V}_{\max}$ and CVR for varying multiscale Frangi parametrizations (Example 2)

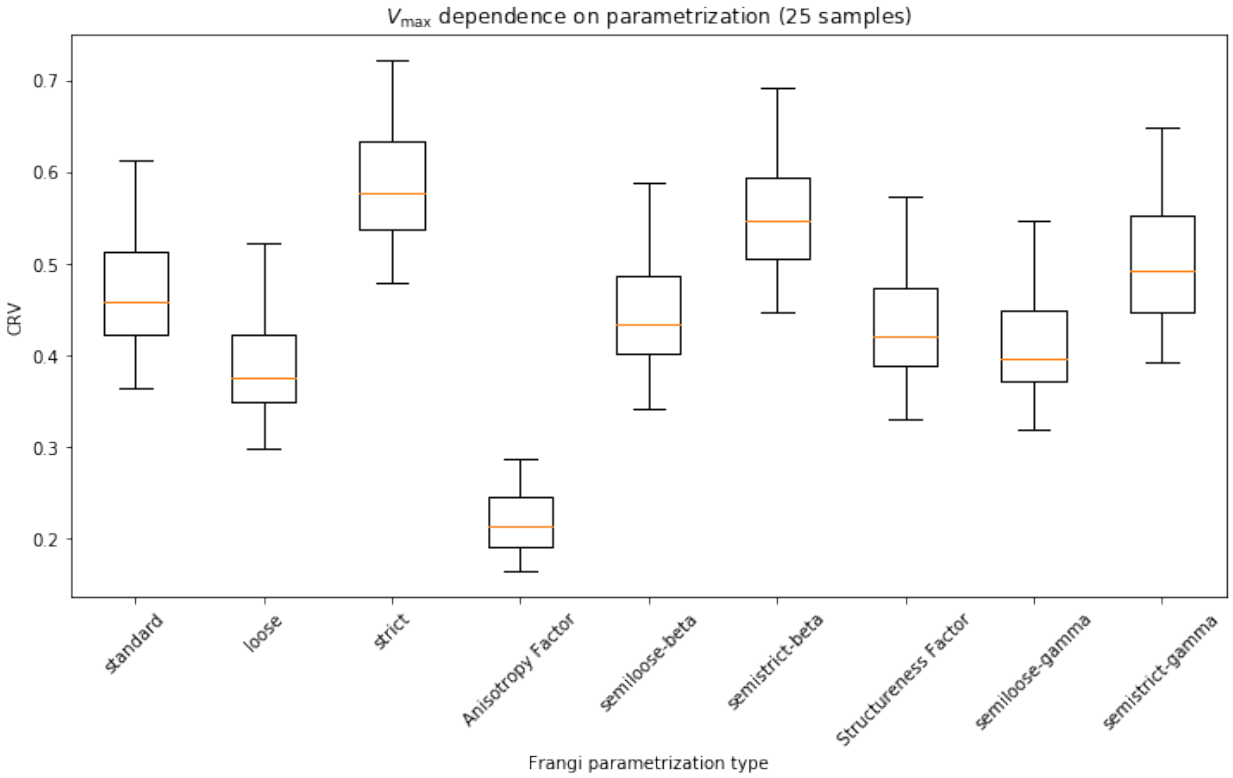


FIGURE 24: *CVR* scores of 25 samples under varying parametrizations

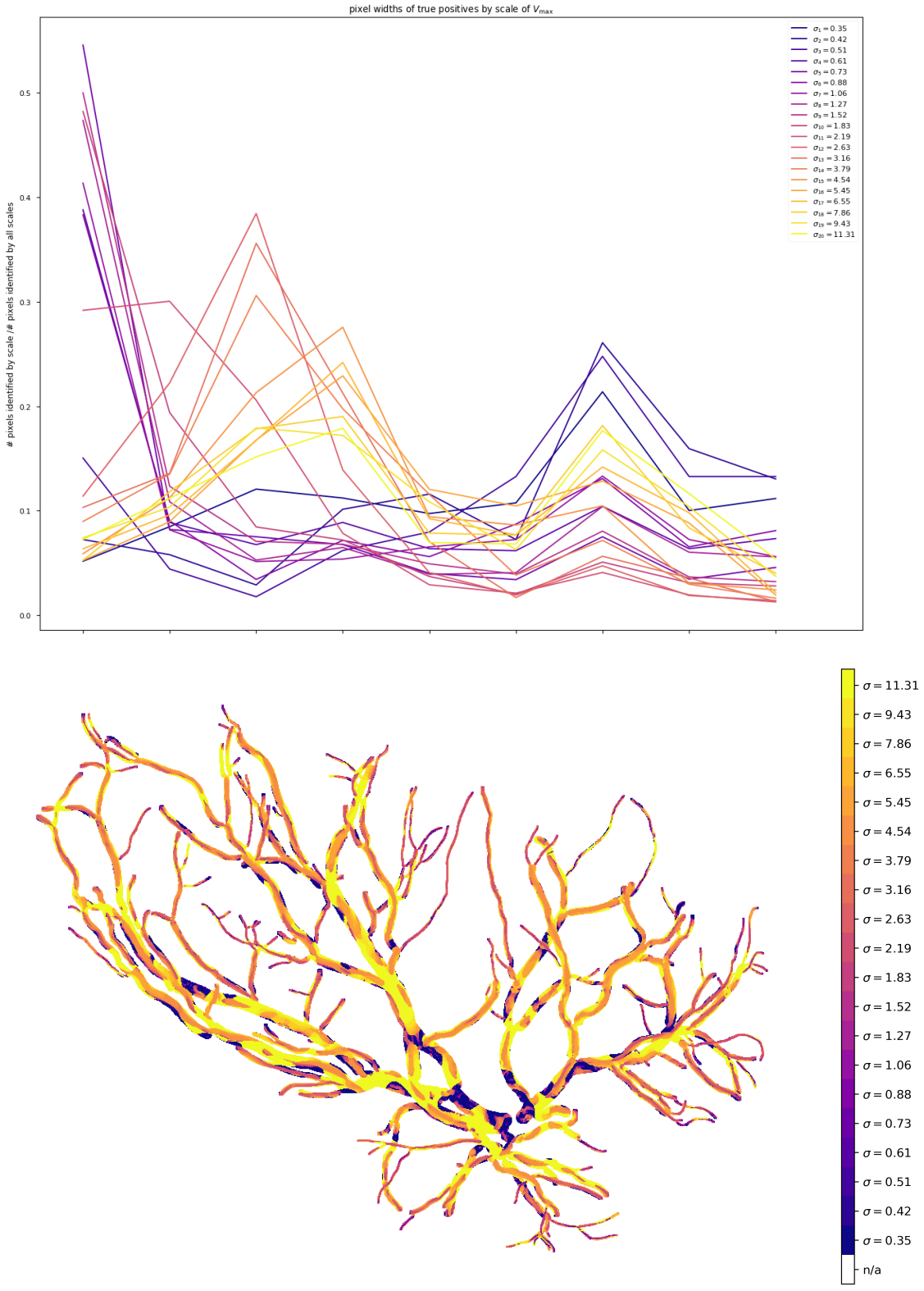


FIGURE 25: Scale of maximum Frangi score for true positives and false negatives

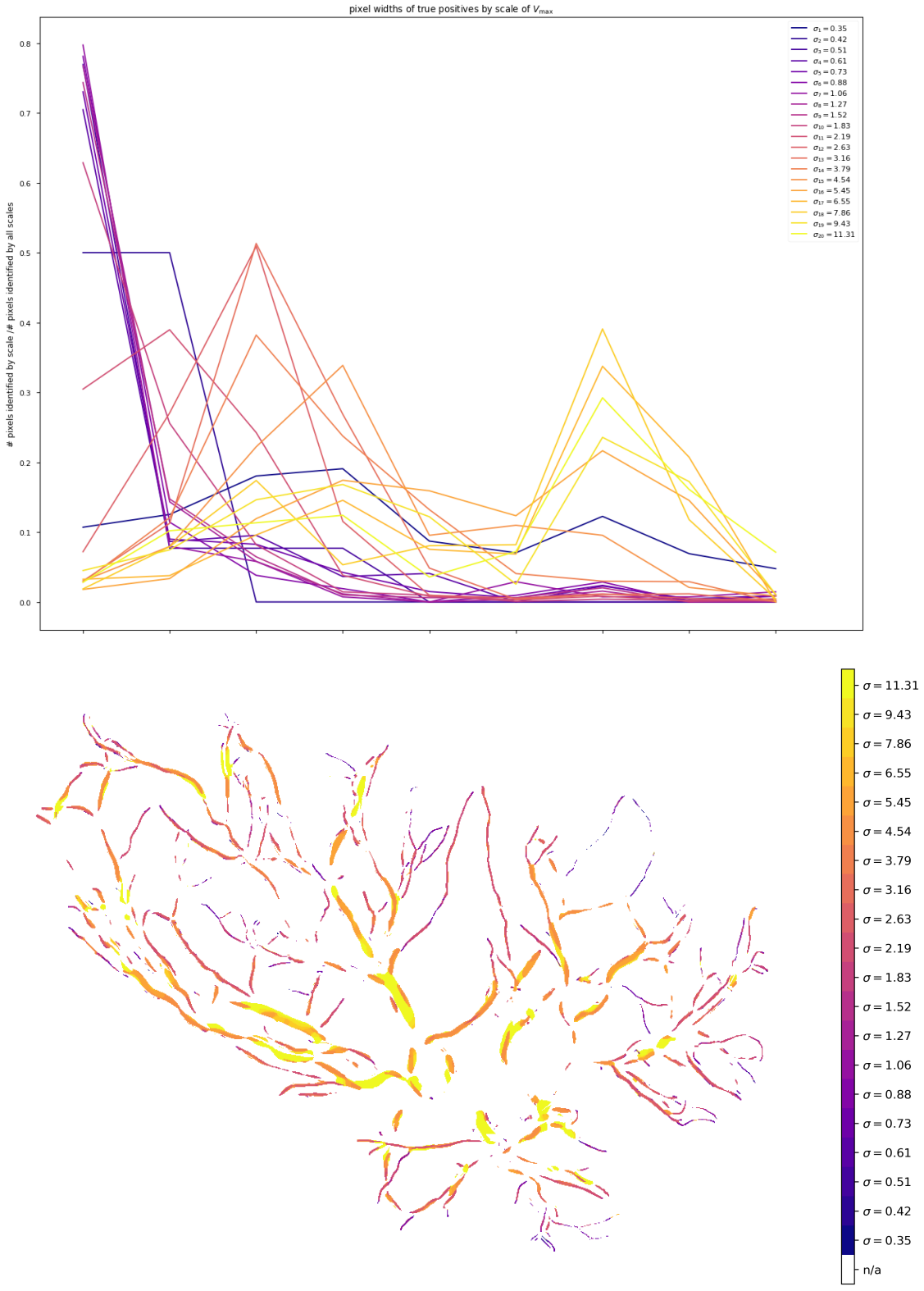


FIGURE 26: Scale of maximum Frangi score for true positives only (95th-percentile filtering)

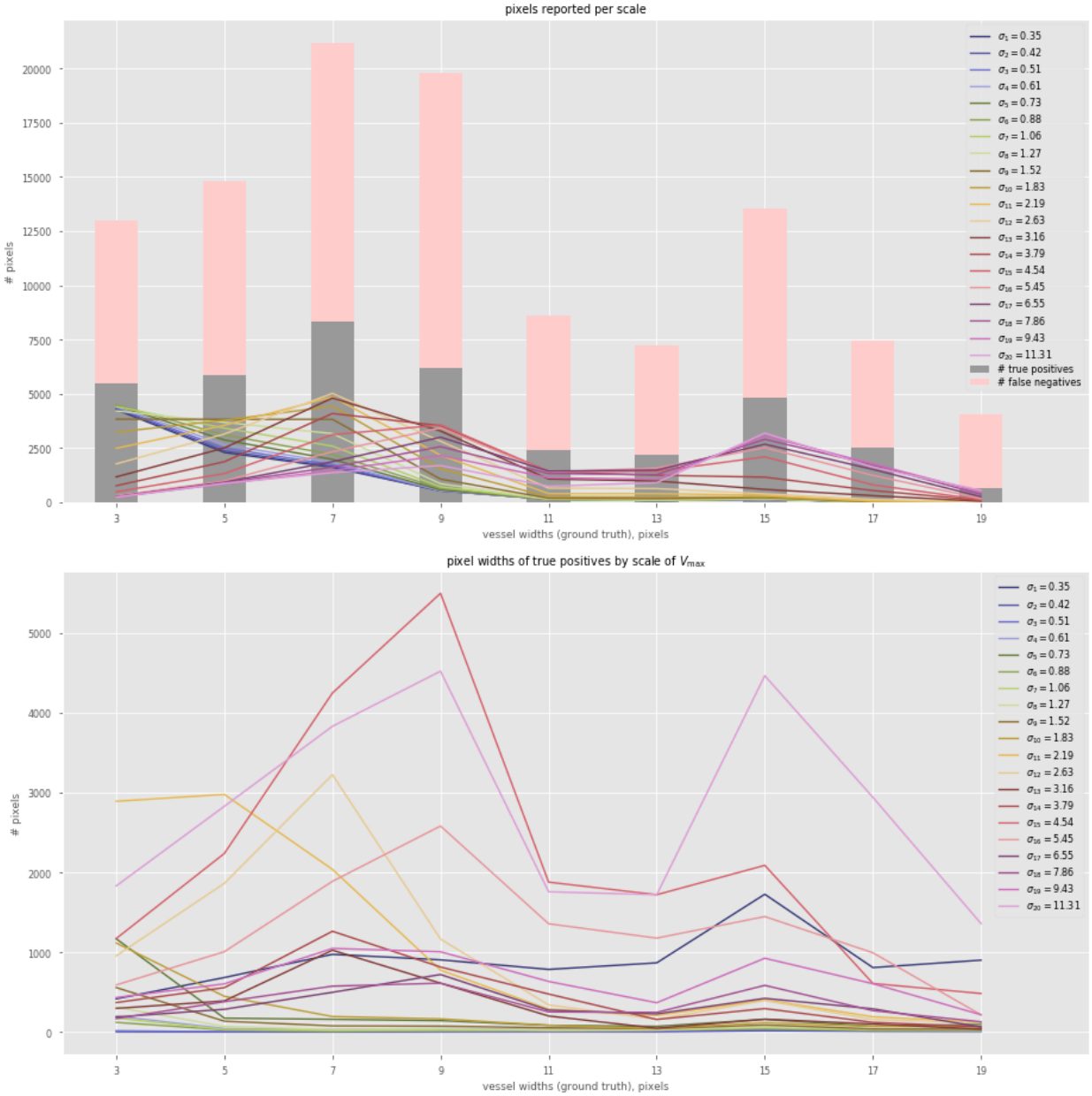


FIGURE 27: Pixel Width of Ground Truth vs. Scale Length for True Positives

CHAPTER 9

SEGMENTATION

We discuss how we use the Frangi as a prefilter and discuss several different segmentation strategies based upon our Frangi-filtered result. We will compare these methods to an unrelated segmentation strategy, the ISODATA threshold. First, we define some standard quantitative measures of success for segmentation methods.

9.1. Binary classifications and the confusion matrix

We wish to come up wish a means of gauging the success of an arbitrary segmentation and will adopt a binary classification model to do so. We end up with a boolean matrix the size of the image, and compare it to the ground truth, the trace, and then compare them to get the number of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). We demonstrate these four labels visually with a confusion matrix, as in Fig. 28. The lighter gray represents true negatives (TN), red represents false negatives (FN), blue represents false positives (FP), and black represents true positives. The darkest gray are is the background mask (pixels are not considered at all in gauging the success of segmentation).

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

$$\text{precision} = \frac{TP}{TP+FP} \tag{9.1}$$

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

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \tag{9.2}$$

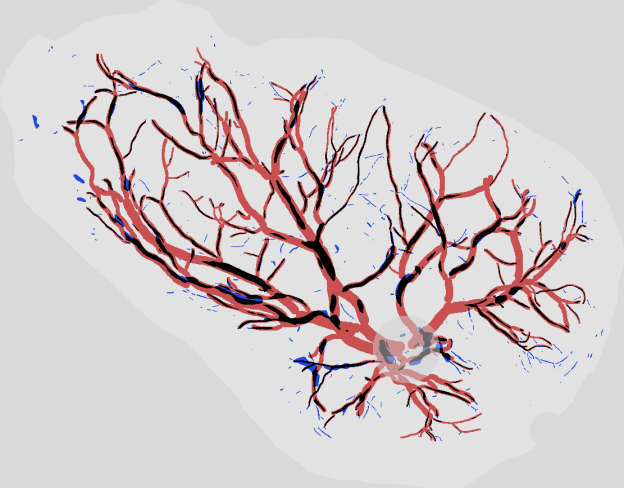


FIGURE 28: Sample confusion matrix

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

Of course, we cannot rely on precision as our sole quantiative factor alone–we could simply return everything negative and recieve a perfect score of 100% precision. Therefore we will gauge that measure with that of the MCC [29]. The main advantage of the MCC is that it is well balanced no matter what the size of the two classes are, and will only be high if the approximation scores well against both labels. A score of 1.0 means the approximation is 100% correct, a score of -1.0 means that everything is completely incorrect, and a score of 0 means that the test performs only as well as random guessing. In our analysis, we will consider both the MCC and precision of a particular segmentation simultaneously. We would like an MCC score as high as possible, but will contextually settle for a lower score as long as the approximation is still *precise*.

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

9.2. Postprocessing Techniques

Here we develop four relatively straightforward methods of postprocessing the multiscale Frangi output to obtain an actual PCSVN extraction. Again, we stress that the Frangi filter itself does not produce a segmentation itself. In fact, Frangi in his original paper [12] refrained from any explicit analysis of the Frangi score apart from taking the maximum across all scales. Still, we wish to demonstrate some immediate methods of postprocessing these samples in order to illustrate the usefulness of this optimized Frangi filter.

9.2.1. Method A: Fixed Threshold

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

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

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

$$\mathcal{V}_{\Sigma\alpha}(x_0, y_0) = \begin{cases} 1 & \text{if} \quad \mathcal{V}_{\max}(x, y) \geq \alpha \\ 0 & \text{else} \end{cases}, \ \alpha > 0 \text{ for } \alpha \text{ fixed.} \tag{9.3}$$

If we insist on such a performing such a thresholding, the "correct" choice of α unfortunately seems to depend on the image domain, so user intervention when dealing with the problem domain seems to be the best strategy. A more sincere use of thresholding might be to choose a relatively high threshold, and then use the result for a further technique. We will discuss alternatives methods of aggregating results from our multiscale method, as well as optimal values for parameters and scales. As a final note, we admit that any future extensions of this work (as will be discussed in Chapter 10) should not hold too much stock in this thresholded result, and analyzing the raw vesselness score Eq. (5.1), or even the un-merged scale-wise scores, would be far more rewarding.

9.2.2. 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

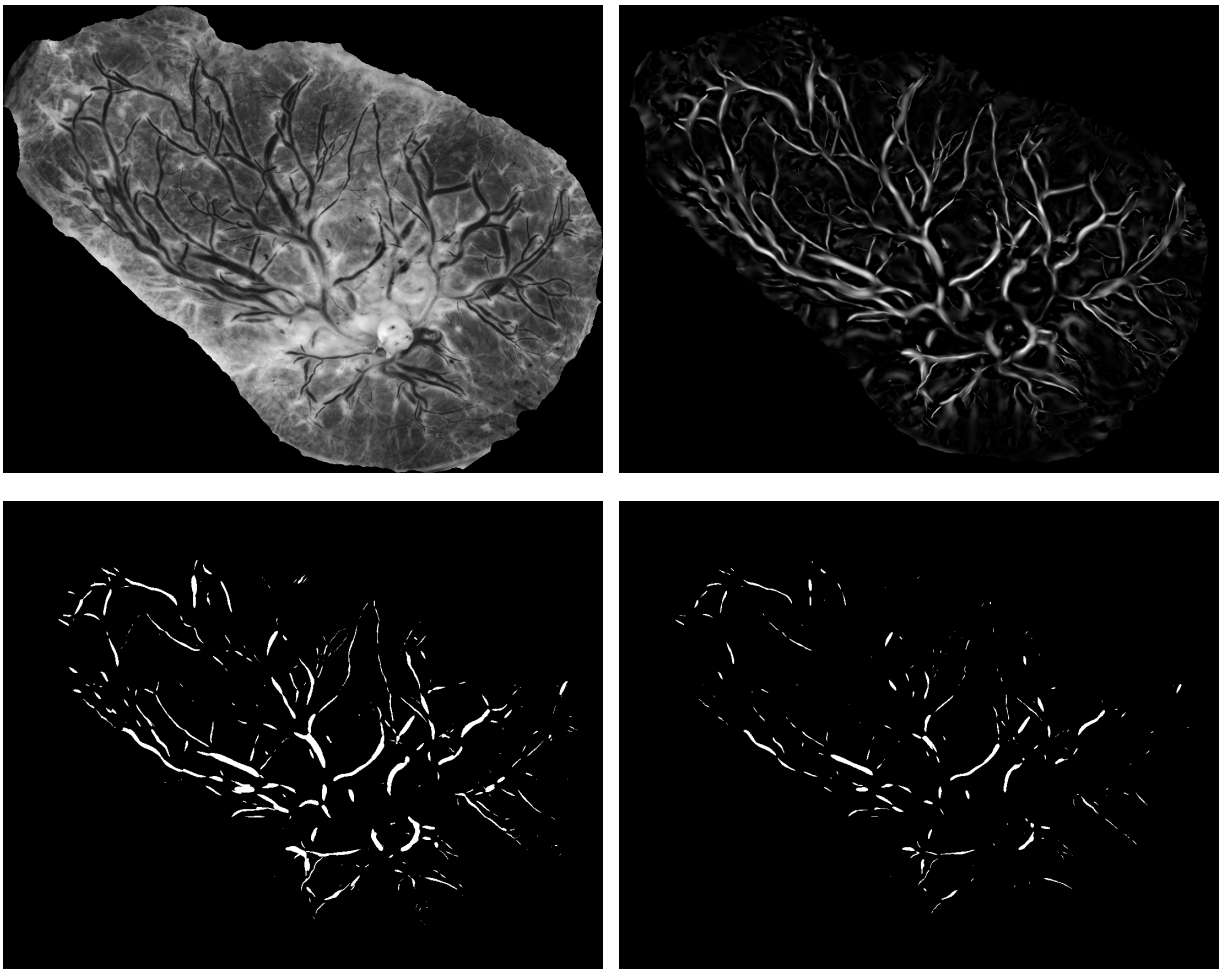


FIGURE 29: Nonzero-percentile thresholding of $\mathcal{V}_{\max}$

.

$\sigma_{\min}$ and $\sigma_{\max}$.

We briefly demonstrate this in Fig. 29 on a particularly well-behaved sample. The top left value image is the original (grayscale image), the top right is $\mathcal{V}_{\max}$, the bottom left is the 95th percentile of nonzero scores of each scale, and the bottom right is the 98th percentile of nonzero scores at each scale.

9.2.3. Method C: Scale-Based Random Walker

We observed that areas where Frangi scores are zero in well-behaved samples seem to neatly outline prominent vascular features. Following this idea, we employed a random walker segmentation [30] (which is implemented by `scikit-image`). Random walk segmentation comes about by solving a diffusion problem over a discrete array (in this case, the Frangi vesselness score itself) given starting markers. At each scale, we positively labeled pixels whose Frangi score was very high ($\mathcal{V}_\sigma(x_0, y_0) > .4$), and negatively labeled pixels whose score was 0 (i.e. where the leading principal eigenvalue was positive). The result of this technique is demonstrated in Fig. 30 and the result (along with the original sample for comparison) is shown in Fig. 31. In Fig. 30, the first column is the Frangi vesselness score at that scale, where black is a score of 0, to emphasize the difference between a score of zero and even a very small positive score, which appear in blue. The middle score are markers passed to the random walker–blue are seeds labelled with a "1" (where the Frangi vesselness score is 0), green is labeled "2" (where $\mathcal{V}_{\max} > .4$), and purple represents unknowns that will either assigned either label. In the last column, the result of the random walker is given–areas that have been added to the label "2" are shown in yellow. Although the result of random walker segmentation is technically a binary matrix, we still show the original seeds of label 2 in green for easier comparison. Similarly, the purple in the right column has actually been labeled "1" for non-vascular, but is left in its original color to emphasize what was assigned background. In Fig. 31 we show the original image and the result of merging all positively marked pixels at each scale. Black means the pixel was unmatched, while increasing colors of blue (larger scales) to white (smaller scales) indicate the smallest scale from which a pixel was marked after the random walker technique. Though we shall set up the multiscale method slightly differently in Chapter 8, we used a Frangi anisotropy coefficient of $\beta = 0.35$, and 12 scales logarithmically spaced from $\sigma_1 = 2^{-1.5}$ to $\sigma_{12} = 2^{3.5}$ to generate these figures. There is a parameter for the random walker algorithm (unfortunately also called β) which serves as a diffusion penalization coefficient (larger values making diffusion over the image less likely). We

used `scikit-image`'s default value of 130.

9.2.4. Method D: Trough Filling

As we discussed in Section 5.2, we can simultaneously calculate the Frangi filter for light and dark curvilinear structures without any added computation time. We shown the signed result of the Frangi filter at different scales two examples in Fig. 32 and Fig. 33, we can simultaneously calculate calculate the Frangi filter response for bright curvilinear features (dark background) and dark curvilinear features (dark background). Since the Frangi filter normally throws away any response where $\lambda_2 < 0$ (if dark curvilinear features are targeted) or $\lambda_2 > 0$ (if light curvilinear features are targeted), we lose no computation time at all by simply keeping both, although we must store more results. After computing the multiscale result, we can easily separate these into a positive and a negative strain, which we will denote $\mathcal{V}_\Sigma^{(+)}$ and $\mathcal{V}_\Sigma^{(-)}$, and then merge each as we would with a traditional multiscale Frangi filter, giving us $\mathcal{V}_{\max}^{(+)}$ and $\mathcal{V}_{\max}^{(-)}$. That is, our $\mathcal{V}_{\max}^{(+)}$ is the same as our $\mathcal{V}_{\max}$ in a conventional Frangi filter, and $\mathcal{V}_{\max}^{(-)}$ is the same result as if we had taken the Frangi filter while only looking for the opposite type (light/dark) curvilinear feature. Plotting $\mathcal{V}_{\max}$ over a scale of $[-1, 1]$ demonstrates an interesting effect. Whereas the Frangi filter generally is not reliable in terms of accurately predicting widths, we *can* get a sense of the width by looking at where there is a relatively strong response of opposite sign.

That is, at the "foot" of every trough on either side, we can see a bordering curvilinear structure of opposite sign. We perform strict Frangi filtering and separate the positive and negative components. We then perform a different threshold for each signed portion of the Frangi response–a strict one (say $\alpha^{(+)} > .3$) for the conventional $\mathcal{V}_{\max}$, and a much looser one for our opposite signed $\mathcal{V}_{\max}^{(-)} > \alpha^{(-)} = .01$. We also truncate the scales we consider for calculating $\mathcal{V}_{\max}^{(-)}$, considering only the 6 smallest scales of 20, since we empirically notice that the bordering curvilinear features are consistently narrower than the vessels themselves.

We will refer to any pixel where $\mathcal{V}_{\max}^{(+)} > \alpha^{(+)}$ as “in the trough” (assuming dark curvilinear features), and any pixel where $\mathcal{V}_{\max}^{(-)} > \alpha^{(-)}$ as potentially “on the lip” of the trough. The process

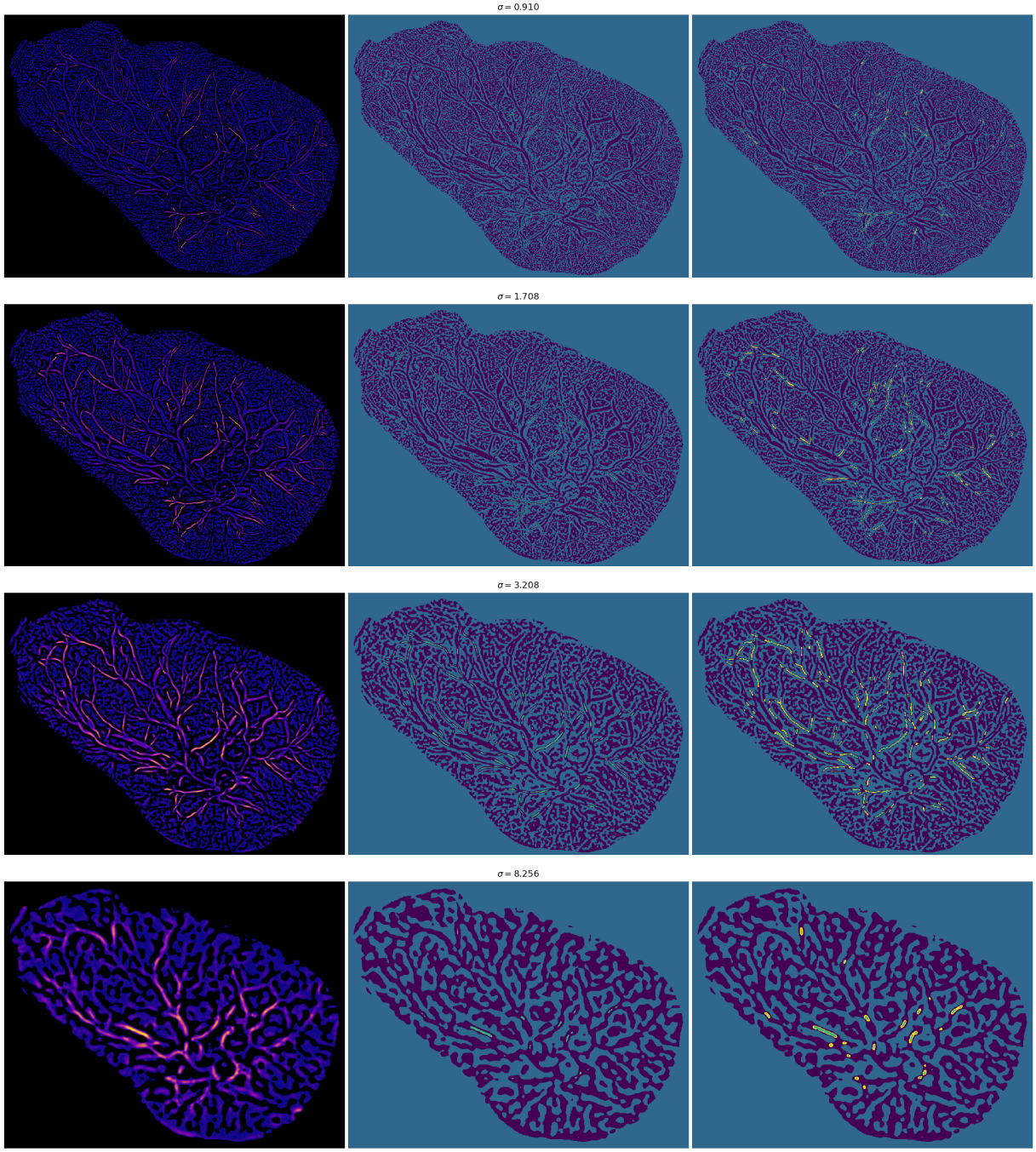


FIGURE 30: Scale-wise random walker segmentation (select scales)

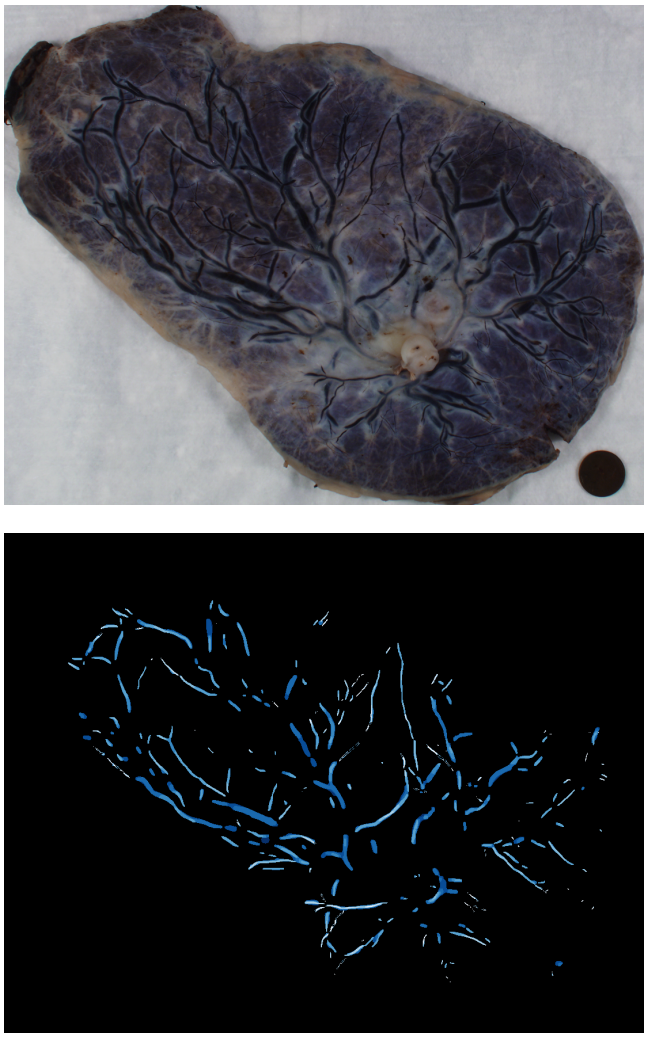


FIGURE 31: Random walker segmentation (result and sample)

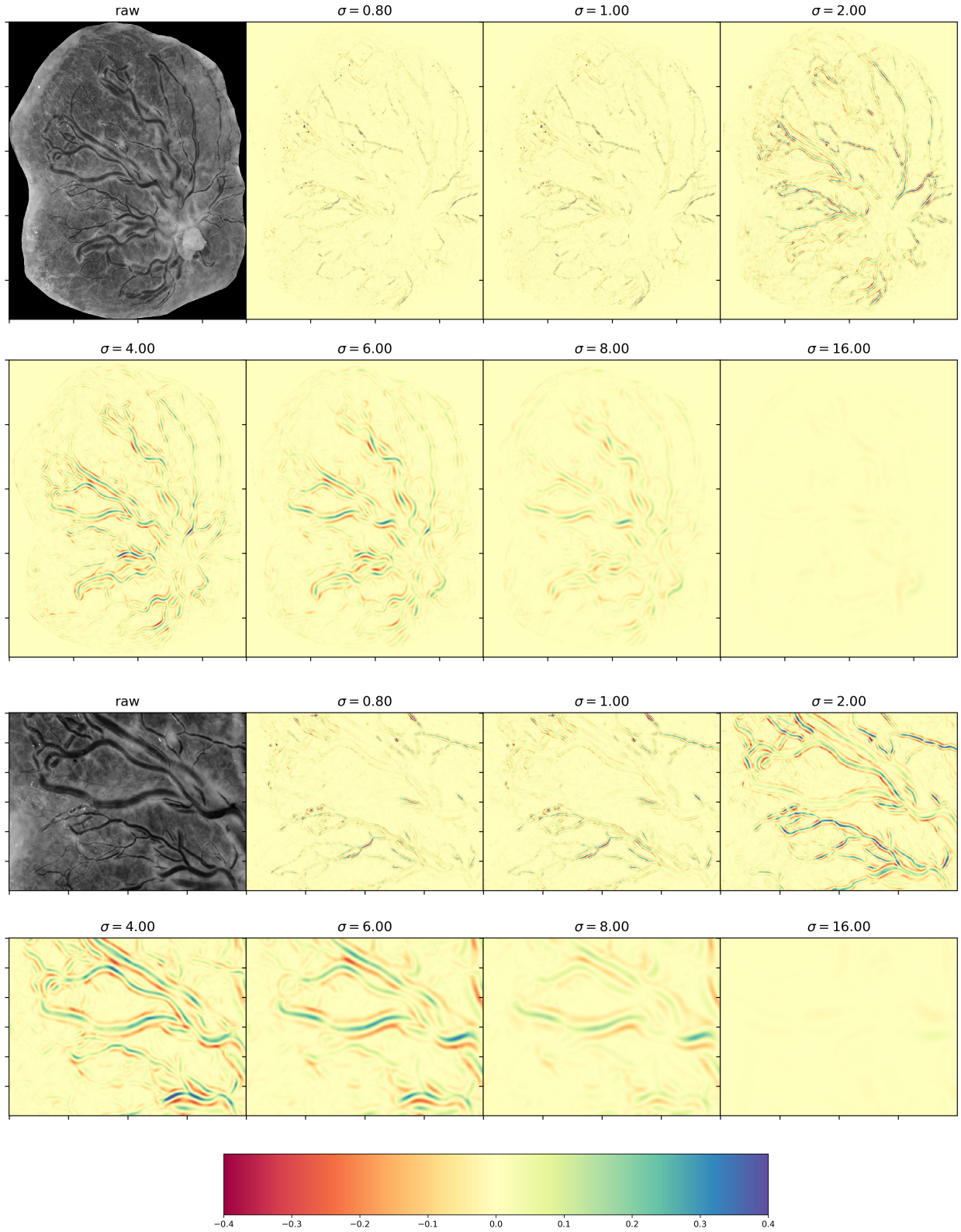


FIGURE 32: Signed Frangi output (plate and inset) (Example 1)

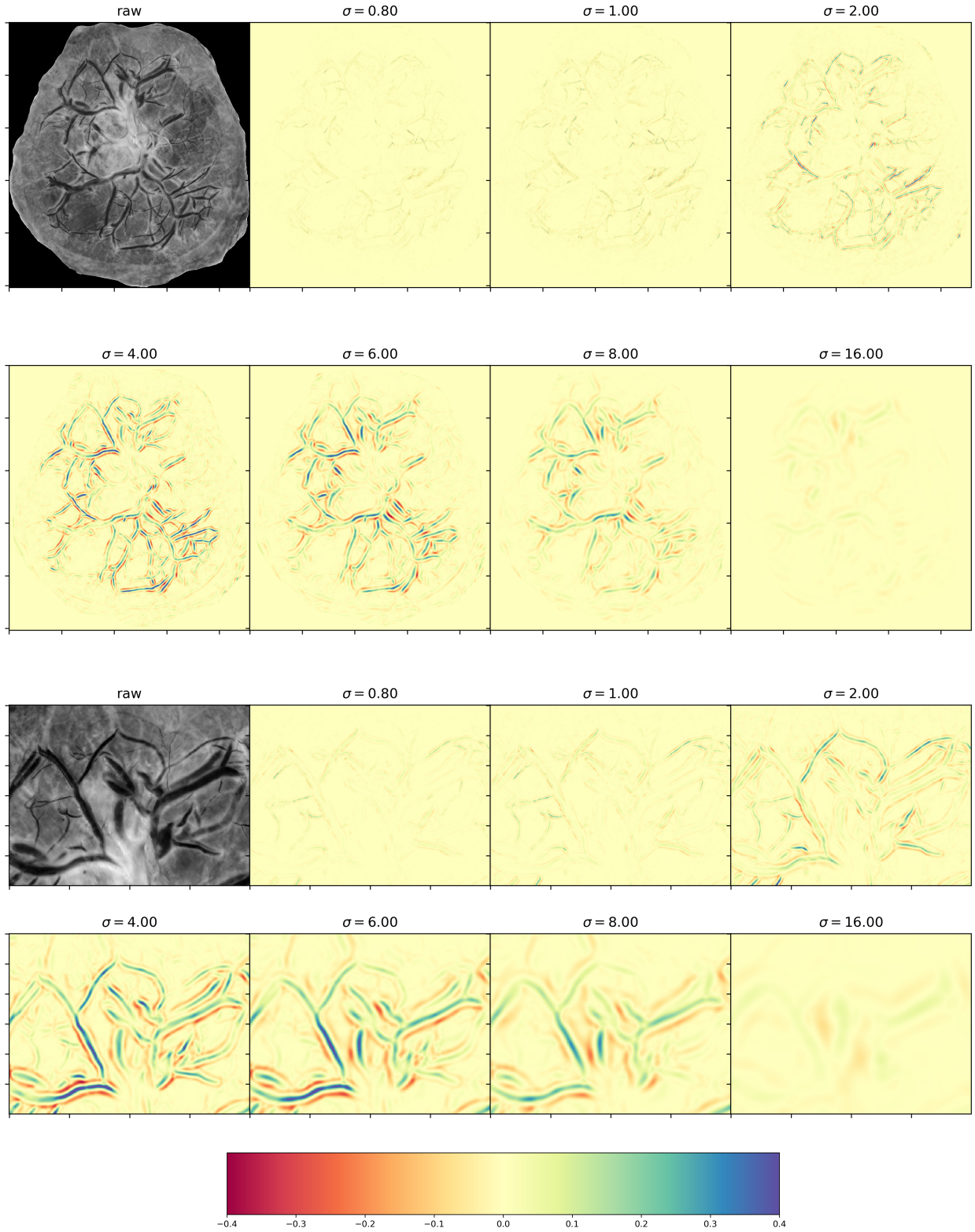


FIGURE 33: Signed Frangi output (plate and inset) (Example 1)

of "trough-filling" then proceeds as follows. For each pixel within the trough, we iterate over disks of integer radius and dilate the pixel by a disk of that radius if that disk includes a point on the lip (where $\mathcal{V}_{\max}^{(-)} > \alpha^{(-)}$. This allows us to extend the Frangi output from the point where it's strongest all the way to the base of the curvilinear structure, providing a much better indication of the true width of the curvilinear structure.

In Fig. 34 we demonstrate the process of creating a trough dilation segments. The top left is the original sample. This was a $N = 20$ Frangi filter with log range from -1.5 to 3.2 and Frangi parameters $\beta = 0.15$ and $\gamma = 1.0$ The middle top and top right shows $\mathcal{V}_{\max}^{(+)}$ and $\mathcal{V}_{\max}^{(-)}$ respectively. The bottom then shows the two markers $\mathcal{V}_{\max}^{(-)} > \alpha^{(-)} = 0.01$ (in red) and $\mathcal{V}_{\max}^{(+)} > \alpha^{(-)} = .3$ (in blue). We build $\mathcal{V}_{\max}^{(-)}$ only using the 6 smallest scales, since the lighter curvilinear 'lip' of the trough is of smaller radius than the radius of the trough itself. Probing at too large scales could perhaps introduce noise. We will call this subset $\Sigma^{(-)} \subset \Sigma$ and choose $\Sigma^{(-)} = \{\sigma_1, \ldots, \sigma_6\}$.

9.3. Nonfrangi Segmentation Methods

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

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

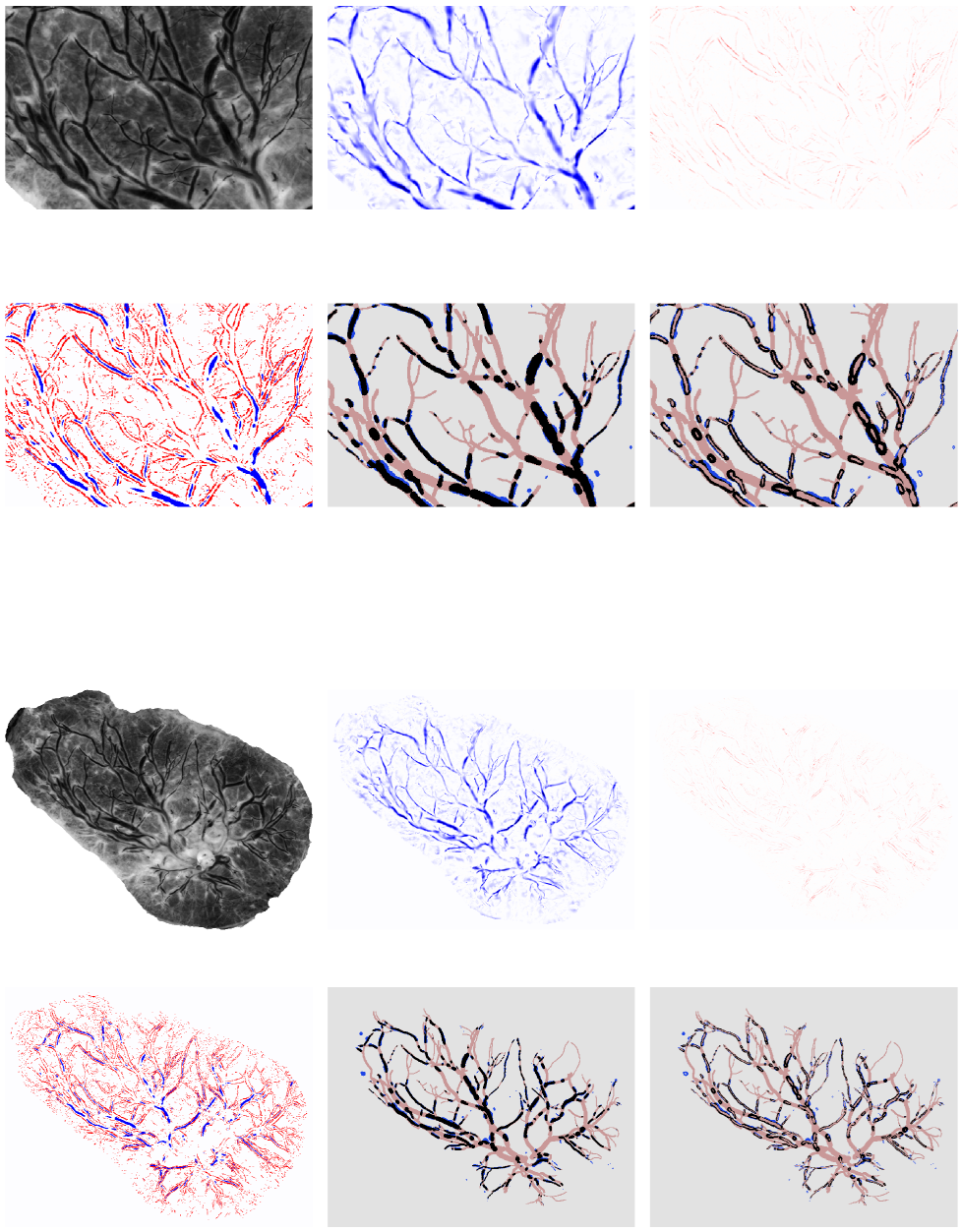


FIGURE 34: Trough dilation process (plate and inset)

$$\alpha_{\text{ISO}} = \arg\min_{\alpha} \left(\frac{1}{2} \left[\text{mean} \{ \texttt{I}(x,y) \mid \texttt{I}(x,y) \leq \alpha \} + \text{mean} \{ \texttt{I}(x,y) \mid \texttt{I}(x,y) > \alpha \} \right] \right) \tag{9.4}$$

Since the vascular structure in our image domain is darker than the background, we select pixels where $\texttt{I}(x,y) < \alpha_{\text{ISO}}$.

9.4. Comparison of Segmentation Methods

In our demonstration of segmentation methods across all 201 samples in our image domain, we use the preprocessing procedure described in Chapter 7.

Our multiscale Frangi filter setup involves 20 scales spaced logarithmically from $\sigma_1 = 2^{-1.5} \approx .35$ to $\sigma_{20} = 2^{3.2} \approx 9.20$. We will compare the results of 6 different segmentation methods using three different parametrizations of the Frangi filter. These parametrizations and segmentation methods with specific parameter choices are summarized in Tables 7 and 8.

Label	Description	Parameter(s)
thresh-high	Fixed threshold of Frangi filter $\mathcal{V}_{\max} > \alpha$	$\alpha = 0.3$
thresh-low	Fixed threshold of Frangi filter $\mathcal{V}_{\max} > \alpha$	$\alpha = 0.2$
snz-p-high	scalewise nonzero percentile filtering of $\mathcal{V}_\Sigma$	$q = 95$
snz-p-low	scalewise nonzero percentile filtering of $\mathcal{V}_\Sigma$	$q = 98$
TF	trough-filling method	$\alpha^{(+)} = 0.3$, $\alpha^{(-)} = .01$, $\Sigma^{(-)} = \{\sigma_1, \cdots, \sigma_6\}$
ISODATA	Non-Frangi global threshold	see Eq. (9.4)

TABLE 7: Summary of segmentation methods

Label	β	γ
standard	0.5	0.5
semistrict	0.15	0.5
strict	0.15	1.0

TABLE 8: Summary of Frangi parametrizations for segmentation demo

In Fig. 35, we look at the results of our various segmentation procedures for a typical well behaved sample. We can see that there are fewer false negatives for strict parametrization than

with a standard parametrization. Even though the best MCC score across the 10 demonstrated in Fig. 35 is achieved by trough filling under standard parametrization, we note that trough filling in the strict parametrization has comparatively fewer false negatives and much higher precision. The highest precision is achieved by our higher fixed threshold. Thus, we should see the high MCC score achieved by the trough filling method as a testament to the usability of a simpler threshold (such as FT-high) as a precursor to more complete segmentation methods.

In Fig. 37 we graph the MCC and precision scores across our entire set of 201 images. In each boxplot, the median is labeled, with first and third quartiles making up the edges of the box. The ends of the whiskers represent (cite matplotlib) $median \pm 1.5 * (Q3 - Q1)$ ($Q3 - Q1$ is the so-called interquartile range). Outliers outside of those whispkers, from samples such as those in Fig. 21, are also plotted as circles.

Across all samples, maximum precision was achieved by the higher fixed threshold of $\alpha = 0.3$ in all but 22 samples, and the trough filling method offered the best MCC in about three-fourths of the samples (51 of 201). Where the trough filling method was not most accurate, we noticed that the $q = 95$ nz-percentile method had a slightly higher MCC score. From viewing the samples, we see that noise from the umbillical cord insertion point still hinders a number of samples, causing noise within the Frangi filter.

From viewing Fig. 37 we can see that, for any of the three parametrizations, we achieve a higher precision and lower MCC from increasing the fixed threshold. The same trend occurs between scalewise nz-p segmentations when we increase the quartile (from q=95 to q=98). The median MCC of the ISODATA method was lower than all Frangi-based segmentations across all with the exception of the highest fixed threshold under strict parametrization–although the latter method has the highest median precision score by far. ISODATA has a much lower precision than any of the other methods present here.

We notice a larger number of MCC outliers for scalewise methods than fixed threshold based methods. We theorize that this is due to the difficulty in knowing *a apriori* an appropriate

range of scales to probe. Future work therefore could immediately improve upon the results of this chapter by finding an appropriate range or, more likely, a normalization factor for scales that makes the success of segmenation less dependent on σ_{min} and σ_{max}.

From Fig. 37 we also notice that the MCC and precision for each scalewise method is not as affected by changes in parameterization as the fixed methods. We suggest that is because stricter parameters have the effect of rescaling to some degree within each step of the multiscale method, so that the same pixels at each scale ultimately occur at each scale's highest percentiles.

Looking at Fig. 36, we see a situation in which scalewise methods do poorly. The $p = 95$ scalewise method under either parametrization, as well as the lower fixed threshold under standard parametrization show some curvilinear noise between vessels. These come from larger scales, so there is apparently no signficant vasculature being picked up at the largest scales, causing noise to appear. This is less pronounced for fixed thresholding using stricter parameters, although some of this noise can still be seen in the strict Frangi's $\mathcal{V}_{\max}$ plot.

9.5. Further directions

We can see from looking at the $\mathcal{V}_{\max}$ outputs of our samples that even where gaps exist in the segmentation output, there is still *some* Frangi output where the gap exists. For various reasons, the output might not be as strong at that particular region, but we can use that fact to bridge some of these gaps. To do so would be to achieve "simple gap completion"–that is, connecting the vascular network to compensate for the shortcomings of a particular segmentation method. We differentiate this from "predictive gap completion," which we will use to signify the problem of connecting the vascular network where the tracer themselves had to guess what was happening with the network. For example, the entire area surrounding the umbilical cord insertion point in Fig. 36 is difficult to make out. The tracer ultimately used some knowledge of vascular network growth to predict what was happening in this region to produce the ground truth. Predictive gap completion would also need to be used to solve crossings of veins and arteries.

First, once we've produced a segmentation, we use morphological thinning to reduce it to

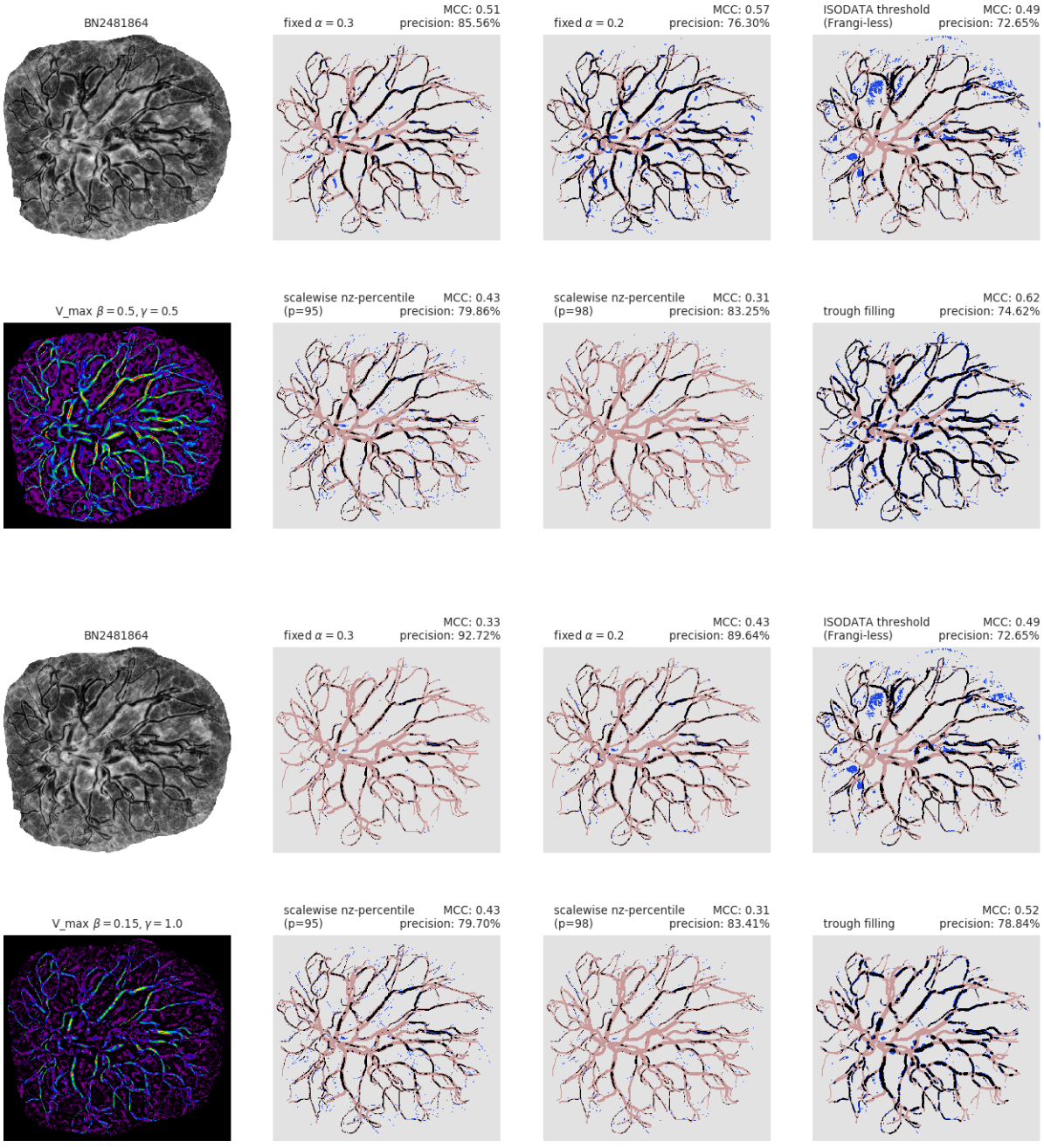


FIGURE 35: Segmentation results, example 1 (standard and strict parametrization)

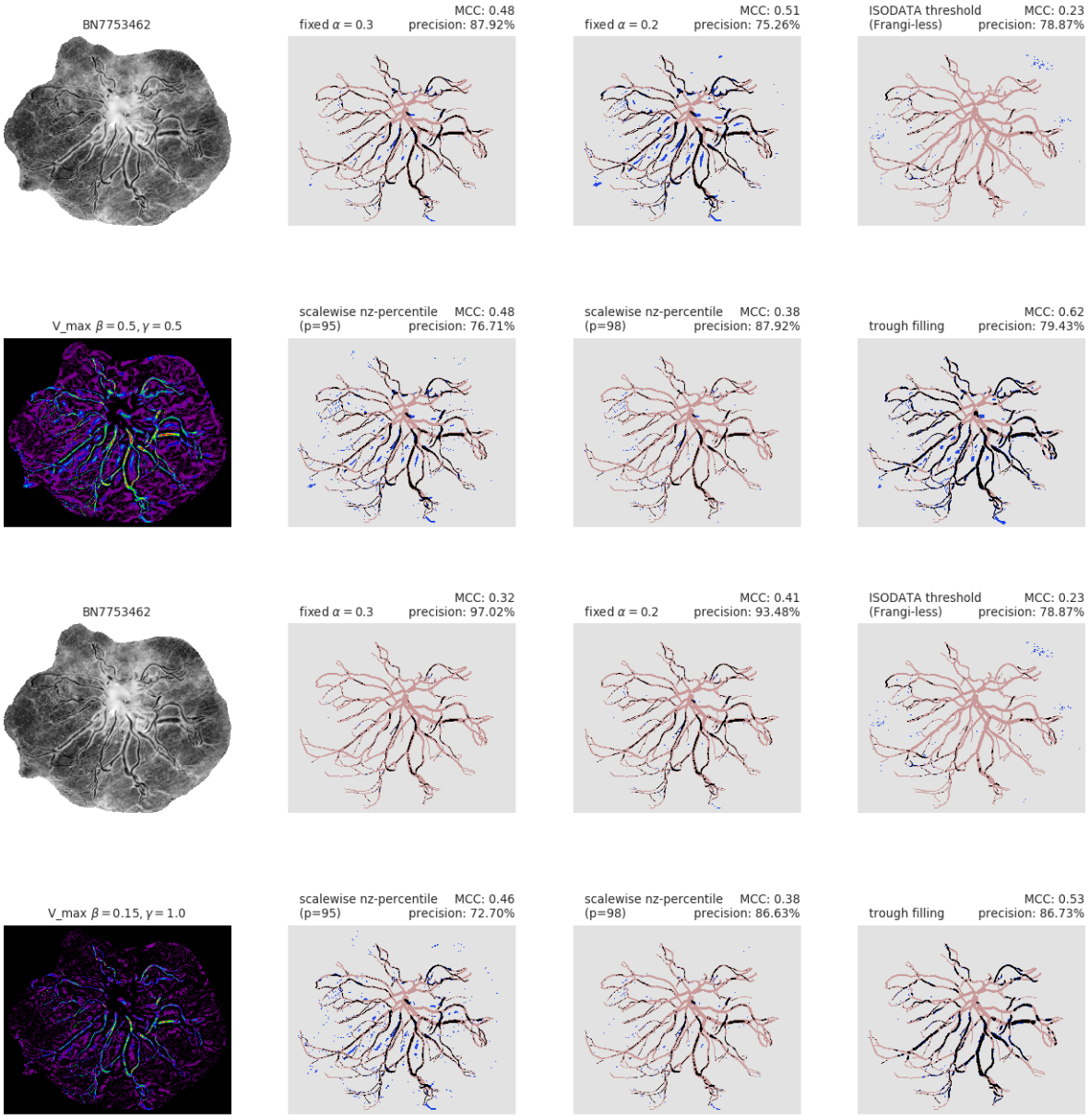
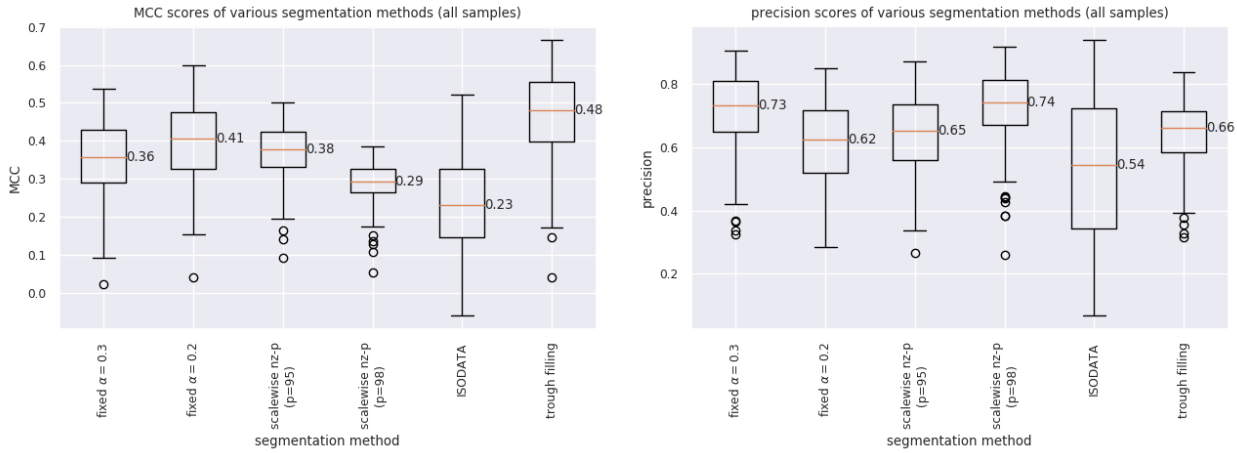
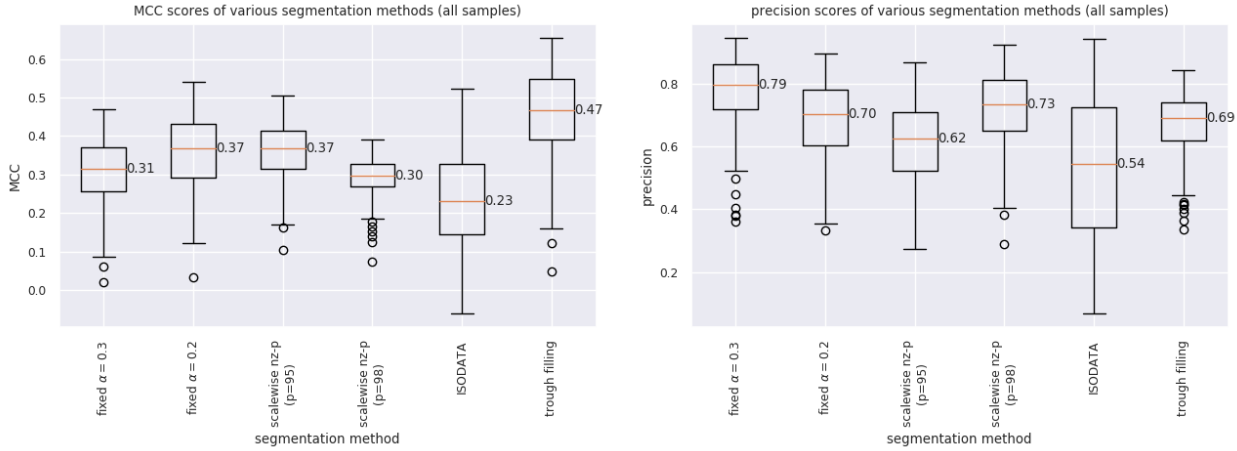


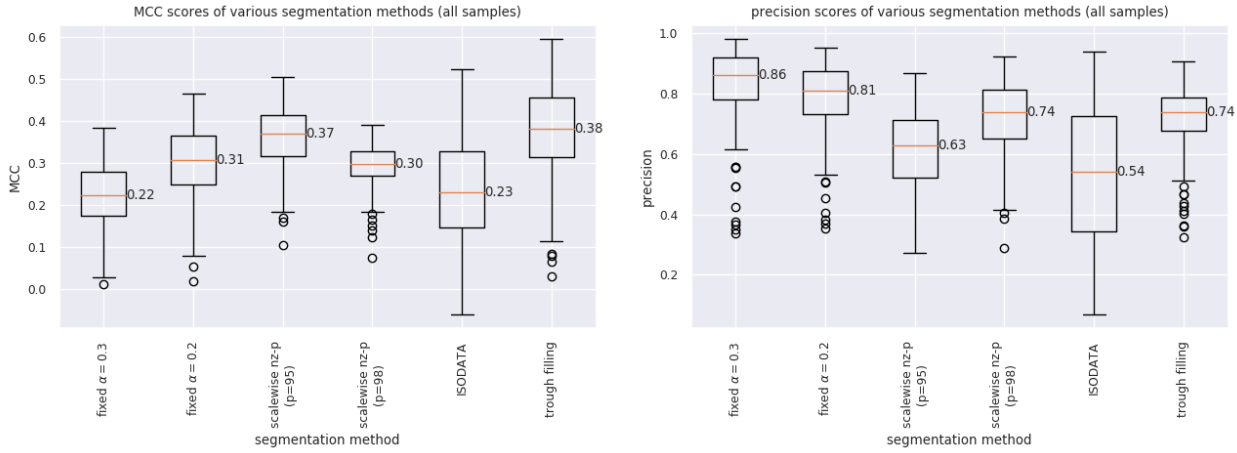
FIGURE 36: Segmentation results, example 2 (standard and strict parametrization)



(a) standard Frangi parametrization ($\beta = 0.5, \gamma = 0.5$)



(b) semistrict Frangi parametrization ($\beta = 0.15, \gamma = 0.5$)



(c) strict Frangi parametrization ($\beta = 0.15, \gamma = 1.0$)

FIGURE 37: MCC and precision of segmentation methods (201 samples)

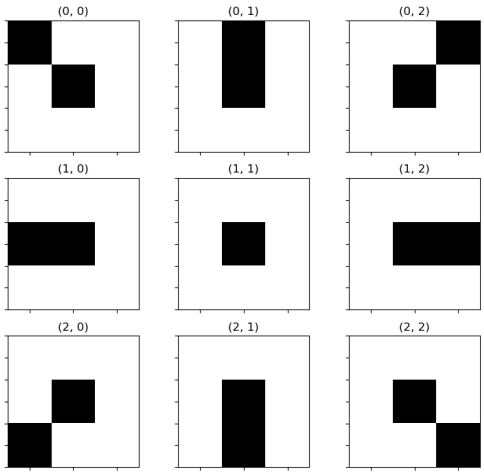


FIGURE 38: Endpoints labels based on adjacent neighbor location

one pixel width with [33] and look for endpoints of an otherwise connected point. Using a 3×3 structuring element, we iterate over each pixel and identify how many local neighbors it has. If a pixel has zero or one local neigbors, we identify it as an endpoint of the partial network. After identifying these endpoints, we assign each a label (i, j) depending on where the neighboring pixel is located, as according to Fig. 38. We deem two endpoints potentially connectible only if they're not connected on the same side. That is, their labels (i, j) and (i', j') must have $i \neq i'$ or $j \neq j'$ (unless i or j is 1). For example, if an endpoint is connected to the partial network on its top side, any endpoint that connects to it cannot also be connected to a network on its top. If a pixel has no connections at all, with label $(1,1)$, we do not restrict its connetions at all. To save time (though we don't anticipate it will affect the result much), we also restrict pais from being more than a set distance away (in this case, 100 pixels in Euclidean length).

After we limit the connections, we consider each pair of endpoints and draw a straight line segment between the two. If that passes through a point where $\mathcal{V}_{\max}$ is 0, we disallow that pair as well. We also disallow any line which crosses any part of the network which is known to exist. Finally, from the list of all remaining pairs of endpoints, we simply select the path along which the

maximum mean value of $\mathcal{V}_{\max}$ is achieved. Fig. 39 shows non-violating paths between end-point pairs. The coloration shows the average value of $\mathcal{V}_{\max}$ along any straight path between compatible endpoints for which there does not occur any pixel with zero $\mathcal{V}_{\max}$. From these we can choose the path with largest average $\mathcal{V}_{\max}$. This partially completed network is shown in Fig. 40(in yellow) overlaid on $\mathcal{V}_{\max}$. We show this result as an indication of what can be done with the simple generation of $\mathcal{V}_{\max}$, although we simply demonstrate it here rather than including it in our main segmentation results. This is because, for its complexity, we hope we can instead solve both simple and predictive gap completion using a single method.

Finally, one final comment we can make. For the scale the maximum Frangi output occurs, at, we can look at the Hessian matrix itself to glean additional information. Recalling our discussion from **??**, we can instead of calculating the leading eigenvalue, we can calculate the leading *eigenvector.* Just as the leading eigenvalue of the Hessian is a good approximation of the principal curvature at that point, the leading eigenvector is a good approximation of the principal direction. In Fig. 41 we show the vectors of the leading (and trailing) eigenvectors of each endpoint of the thinned, approximated network. We note that these could additionally be used towards the goal of network completion, but again we hesitate and instead look for a better overarching method that would address the gap problem.

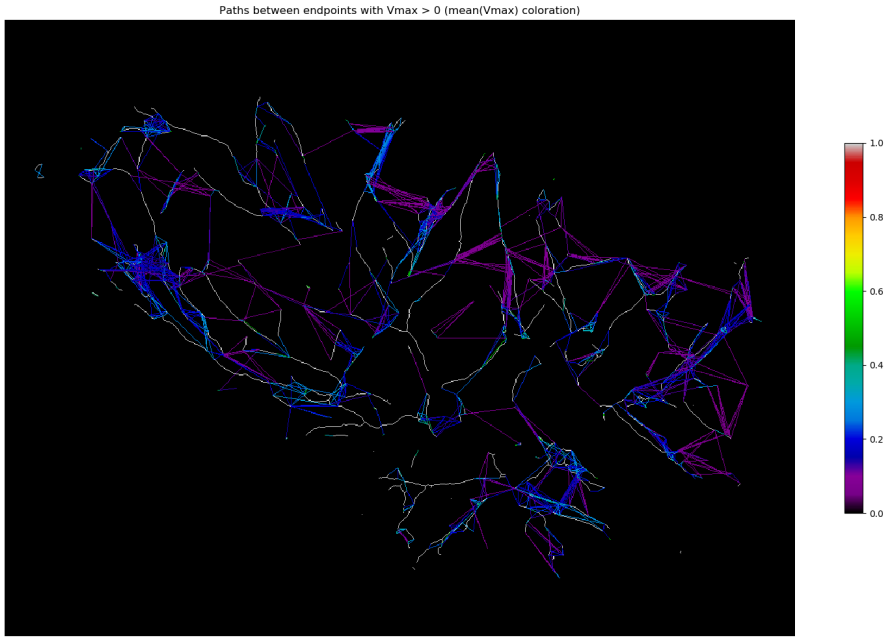


FIGURE 39: All lines between endpoints with nonzero $\mathcal{V}_{\max}$

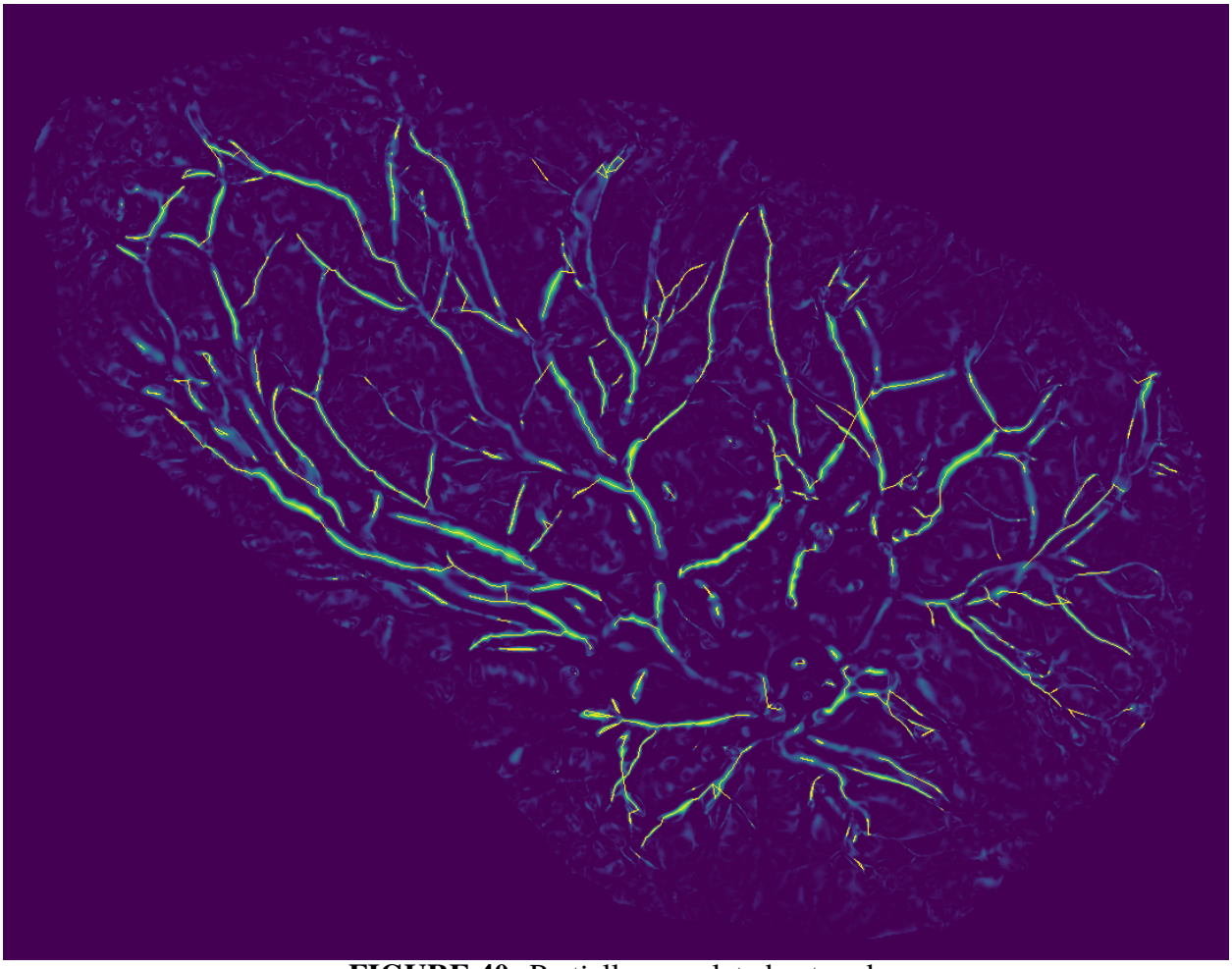


FIGURE 40: Partially completed network

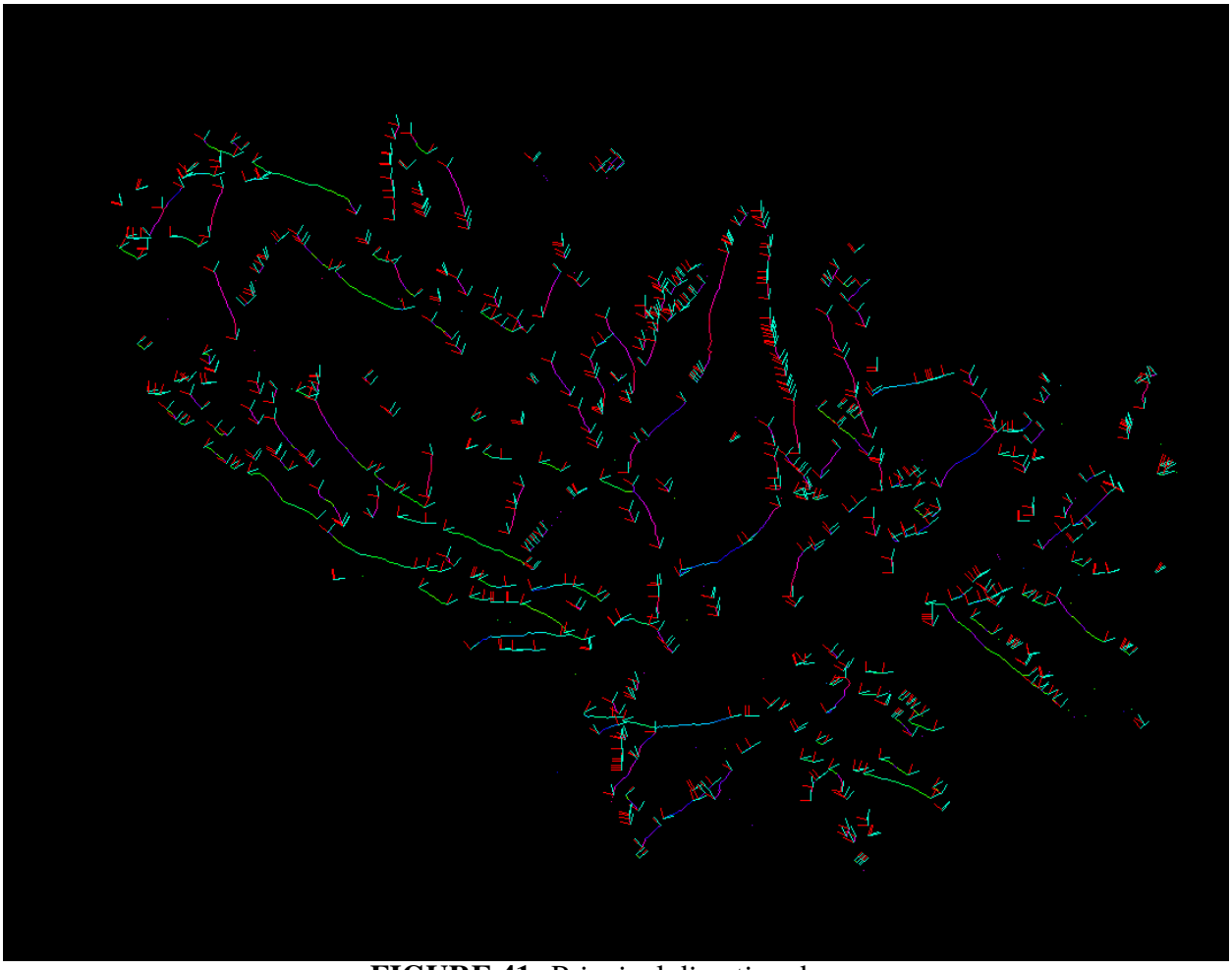


FIGURE 41: Principal direction demo

CHAPTER 10

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. Specifically we compared

Future research will first aim toward automating more of the preprocessing, specifically toward identifying the perimeter, umbilical cord insertion point, and any cuts without relying on a manual trace. As mentioned in Chapter 7, a more careful preparation of samples (i.e. as the picture is taken) would alleviate some of the difficulty of image registration. We also should improve our glare reduction algorithm, as it currently relies on an arbitrary threshold. As far as our multiscale method goes, we would like to develop a notion of automatic scale selection, that would help us better normalize smaller scales, as well as allow us to find a specific largest scale (as some vessels in specific samples were only easily identified at very large scales, whereas these scales would introduce only noise in most other samples).

Any additional research on this problem that wishes to use the Frangi filter as a prefilter (e.g. the trough completion method detailed in Chapter 9 or some more involved algorithm) will likely need to solve the network completion problem.

Finally, we would like to demonstrate the adaptibility of this method by applying it to more image domains.

APPENDICES

APPENDIX A

CODE LISTINGS

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

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

Earlier versions of these packages may be compatible but are not guaranteed to be so. The scripts listed in this appendix are also hosted at `github.com/wukm/pycake`.

listings/add_margins.py

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

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

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

finv_thresh = nz_percentile(finv, 80)

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

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

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

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

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

listings/compare_parameters_alt.py

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

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

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

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

import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
from skimage.util import img_as_float
from skimage.io import imread
from placenta import (get_named_placenta, list_by_quality, cropped_args,
                      mimg_as_float)

from frangi import frangi_from_image
import numpy.ma as ma
from hfft import fft_gradient, fft_hessian, fft_gaussian
from merging import nz_percentile
from plate_morphology import dilate_boundary
import os.path, os
from skimage.morphology import thin
from postprocessing import dilate_to_rim
from scoring import confusion, mcc, integrate_score
from placenta import open_tracefile, open_typefile
from preprocessing import inpaint_hybrid
from placenta import measure_ncs_markings, add_ucip_to_mask


STRICT = {'label':'strict', 'beta':0.10, 'gamma':1.0, 'alpha':.15 }

STANDARD = {'label':'standard', 'beta':0.5, 'gamma':0.5, 'alpha':.4}

SEMISTRICT = {'label':'semistrict', 'beta':0.35, 'gamma':0.5, 'alpha':.4}

LOOSE = {'label':'loose', 'beta':1.0, 'gamma':0.5, 'alpha':.8}

cm = mpl.cm.plasma
#cmscales = mpl.cm.magma
cm.set_bad('k', 1)  # masked areas are black, not white
#cmscales.set_bad('w', 1)

scales = np.logspace(-1.5, 3.5, num=12, base=2)

integral_scores = list()

for filename in list_by_quality(0):

    cimg = open_typefile(filename, 'raw')
    ctrace = open_typefile(filename, 'ctrace')
    trace = open_tracefile(filename)
    img = get_named_placenta(filename)
    crop = cropped_args(img)
    ucip = open_typefile(filename, 'ucip')
    img = inpaint_hybrid(img)

    # make the size of figures more consistent
    if img[crop].shape[0] > img[crop].shape[1]:
        # and rotating it would be fix all this automatically
        cimg = np.rot90(cimg)
        ctrace = np.rot90(ctrace)
        trace = np.rot90(trace)
        img = np.rot90(img)
        ucip = np.rot90(ucip)
        crop = cropped_args(img)

    ucip_midpoint, resolution = measure_ncs_markings(ucip)
```

```
    ucip_mask = add_ucip_to_mask(ucip_midpoint, radius=60, mask=img.mask)

    name_stub = filename.rstrip('.png').strip('T-')


    F_demos = list()
    integrals = list()
    PARAMS = [LOOSE, SEMISTRICT, STANDARD, STRICT]

    for params in PARAMS:
        print(f"running {params['label']} Frangi on {name_stub}")
        F_demo = np.stack([frangi_from_image(img, sigma, beta=params['beta'],
                                             gamma=params['gamma'], dark_bg=False,
                                             dilation_radius=20, rescale_frangi=True)
                                             for sigma in scales])

        F_max = F_demo.max(axis=0)

        integral = integrate_score(F_max, trace, mask=img.mask)

        F_demos.append(F_max)
        integrals.append(integral)

        del F_demo


    fig, ax = plt.subplots(nrows=2, ncols=3, figsize=(20,12))
    A = ax.T.ravel()

    A[0].imshow(cimg[crop])
    A[0].set_title(name_stub)

    A[1].imshow(ctrace[crop])
    A[1].set_title('ground truth')

    for i, (Fmax, integral, params) in enumerate(zip(F_demos, integrals, PARAMS), 2):
        beta = params['beta']
        gamma = params['gamma']
        label = params['label']
        A[i].imshow(ma.masked_where(Fmax==0,Fmax)[crop], cmap=cm, vmin=0, vmax=1)
        A[i].set_title(rf'    $V_{\max}$ ({label})'+'\n'+
                       rf'    $\beta = beta:.2f, \gamma = gamma:.3f$', loc='left')
        A[i].set_title(rf'CVR: {integral:3f}', loc='right')

    [a.axis('off') for a in A]
    fig.tight_layout()
    plt.show()

    integral_scores.append((name_stub, integrals))
```

listings/cut_demo.py

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

import numpy as np
import matplotlib.pyplot as plt

from placenta import (open_typefile, list_placentas, get_named_placenta)
from plate_morphology import mask_cuts_simple, dilate_boundary

from skimage.color import gray2rgb
from skimage.morphology import thin, binary_dilation, disk, square
import numpy.ma as ma
```

```
import os.path

def l2_dist(p,q):
    return int(np.round(np.sqrt((p[0]-q[0])**2 + (p[1]-q[1])**2)))

placentas = list_placentas('T-BN')
samples_with_cuts = list()

for filename in list_placentas('T-BN'):

    # this one has two cuts, another one has two cuts as well
    #if filename != "T-BN2459820.png":
    #       continue

    img = get_named_placenta(filename)
    ucip = open_typefile(filename, 'ucip')

    #C, has_cut = mask_cuts(img, ucip, return_success=True, in_place=False)

    #if has_cut:

    #    dilcut = img.copy()

    #    print(filename, "has a cut!")
    #    samples_with_cuts.append(filename)

    #    B = np.all(ucip==(0,0,255), axis=-1)
    #    G = np.all(ucip==(0,255,0), axis=-1)

    #    cutmarks = np.nonzero(thin(B))
    #    perimeter = np.nonzero(G)

    #    #for array in the tuple that comes out of np.nonzero(thin(B))
    #    # or just one if it's just a single thing i guess?

    #    # the x, y points of the cutmarks are in columns
    #    cutinds = np.stack(cutmarks)

    #    for P in cutinds.T:

    #        # consider larger and larger window sizes
    #        for W in [100,200,300]:
    #            # consider all perimeter elements within these bounds

    #            rmin, rmax = max(0, P[0]-W), min(img.shape[0], P[0]+W)
    #            cmin, cmax = max(0, P[1]-W), min(img.shape[1], P[1]+W)
    #            window = np.s_[rmin:rmax, cmin:cmax]

    #            # perimeter indices within the window
    #            pinds = [(x,y) for x, y in zip(*perimeter)
    #                        if x > rmin and x < rmax and y > cmin and y < cmax
    #                        ]
    #            if pinds:
    #                break
    #    if pinds:

    #        # max distance to boundary point in the window
    #        # we really only need to keep the largest; deque?
    #        dists = sorted([(pp, l2_dist(P,pp)) for pp in pinds],
    #                        key=lambda t: t[1])
    #        r = int(dists[-1][1]) + 1 # get largest radius but closest point
    #        P = dists[0][0]
    #        B = np.zeros_like(img.mask)

```

```
    #         B[cutmarks] = True
    #
    #         # center a disk of found radius there
    #         D = disk(r)
    #         winx = max(P[0]-r,0), min(P[0]+r+1,B.shape[0])
    #         winy = max(P[1]-r,0), min(P[1]+r+1,B.shape[1])
    #         try:
    #             B[winx[0]:winx[1] , winy[0]:winy[1]] = D
    #         except ValueError:
    #             # they're out of bounds so it's a size mismatch. fix it
    #             # by starting/ending D index with opposite sign of the initial
    #             # p +/- radius that was out of bounds
    #             # for example P[0]-r was -9 and everything else was fine
    #             # so you just need to set left side to D[9:,:]
    #             # but you should  wrap this up in a function so the three times
    #             # you do it here and the one time in ucip all gets the same
    #             # code
    #             pass
    #         dilcut[B] = ma.masked
    #
    #     else:
    #         # this is probably not going to happen, but just in case no
    #         # nearby perimeter was found, just... give up
    #         pass
    #
    #     rminv, rmaxv = max(0, rmin-W//2), min(img.shape[0], rmax+W//2)
    #     cminv, cmaxv = max(0, cmin-W//2), min(img.shape[1], cmax+W//2)
    #     view = np.s_[rminv:rmaxv, cminv:cmaxv]
    #     montage = np.hstack((gray2rgb(img.filled(0)[view]),
    #                          ucip[view],
    #                          gray2rgb(C.filled(0)[view]),
    #                          gray2rgb(dilcut.filled(0)[view])))
    #     filestub, _ = os.path.splitext(filename)
    #     plt.imsave(f'demo_output/cut_demo/{filestub}_cutopts.png', montage)
    #     #plt.imshow(montage)
    #     plt.show()
    #     plt.close()
    mimg, success = mask_cuts_simple(img, ucip, return_success=True)
    if success:
        montage = np.hstack((img.filled(0),
                             mimg.filled(0)))
        plt.imshow(montage)
        plt.show()
        plt.close()
print("*"*80)
print(f"there were {len(placentas)} total samples",
      f"and {len(samples_with_cuts)} of them had cuts")
```

listings/diffgeo.py

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

import numpy as np
import numpy.ma as ma

from skimage.feature import hessian_matrix, hessian_matrix_eigvals
from hfft import fft_hessian
from numpy.linalg import eig


def principal_curvatures(img, sigma=1.0, H=None):
    """Calculate the approximated principal curvatures of an image

```

```
    Return the (approximated) principal curvatures {κ1, κ2} of an image,
    that is, the eigenvalues of the Hessian at each point (x,y). The output
    is arranged such that |κ1| <= |κ2|.  Note that the Hessian of the image,
    if not provided, is computed using skimage.feature.hessian_matrix, which
    can be very slow for large sigmas.

    Parameters
    ----------
    img: array or ma.MaskedArray

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

    sigma: float, optional
        Standard deviation of the Gaussian (used to calculate the hessian
        matrix).
    H:  list of array, optional
        The hessian itself (Hxx,Hxy,Hyy) whose eigenvalues will be calculated.
        Use this option if you're going to calculate the Hessian using faster
        means, e.g. via FFT.

    Returns
    -------
    (K1, K2):   tuple of arrays
        K1, K2 each are the exact dimension of the input image, ordered in
        magnitude such that |κ1| <= |κ2| in all locations.

    Examples
    --------
        >>> K1, K2 = principal_curvatures(img)
        >>> K1.shape == img.shape
        True
        >>> (K1 <= K2).all()
        True
        >>> K1.mask == img.mask
        True
    """

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

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

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

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

        channel = img[:,:,ic]

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

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

        # Make K2 larger in magnitude, as consistent with Frangi paper
        K1[:, :, ic] = L[0, :, :]
```

```
        K2[:, :, ic] = L[1, :, :]

    try:
        mask = img.mask  # get mask to add to each if input was a masked array

    except AttributeError:

        pass  # there's no mask, so do nothing

    else:
        K1 = ma.masked_array(K1, mask=mask)
        K2 = ma.masked_array(K2, mask=mask)

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

    return K1, K2


def reorder_eigs(L):
    """reorder eigenvalues by decreasing magnitude.

    Eigenvalues are outputted from hessian_matrix_eigvals so that L1 >= L2.
    This reorders this so that |L1| >= |L2| instead (where L1,L2=L)
    Parameters
    ----------
    L:  ndarray or iterable of ndarrays
        As outputted by, say, hessian_matrix_eigs. If a single ndarray, it
        should be the shape (N, *img.shape) where there are N eigenvalues to
        reorder. You may also input a tuple like (L1,L2).
    Returns
    -------
    eigs: ndarray
        The eigenvalues in decreasing order of magnitude; that is
        eigs[i,j,k] is the ith-largest eigenvalue at position (j, k).
        Each of these is the same shape the original inputs, but
        np.abs(L1r) >= np.abs(L2r) will be true. See warning below.


    Warnings / Notes
    ----------------
    Please note the order! Outputs are given in *decreasing* magnitude. This is
    done to align with the behavior of skimage.feature.hessian_matrix_eigvals,
    but if you want to label them according to the Frangi filter (where k2
    denotes the *larger* magnitude eigenvalue, you should reverse the labels:

        >>>k2, k1 = reorder_eigs(L)  # k2, k1 as frangi labeled them
        >>>np.all(np.abs(k2) >= np.abs(k1))
        True

    It doesn't actually matter the order in which inputs are inputted (they
    will be sorted the same regardless).

    Example
    -------
    >>>K1,K2 = hessian_matrix_eigvals(H)
    >>>(K1 >= K2).all()
    True
    >>>(np.abs(K1) <= np.abs(K2)).all()
    False
    >>>K1r, K2r = reorder_eigs(K1,K2)
    >>>(K1r <= K2r).all()
```

```
    False
    >>>(np.abs(K1r) <= np.abs(K2r)).all()
    True

    TODO
    ----
    Support out= keyword
    """
    # this will do nothing if L is already an array but will make it an array
    # if it's a tuple/list/iterable
    L = np.stack(L)
    mag =  np.argsort(np.abs(L), axis=0)

    # now L2 is larger in absolute value, as consistent with Frangi paper
    return np.take_along_axis(L, mag, axis=0)


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

    mask should  be positive where the PD's should *NOT* be calculated
    this function actually returns the theta corresponding to
    leading and trailing principal directions, i.e. angle w / x axis
    """

    if H is None:
        H = fft_hessian(img, sigma)

    Hxx, Hxy, Hyy = H


    # determine if there was a supplied mask or use images if it exists
    if mask is None:
        try:
            mask = img.mask
        except AttributeError:
            masked = False
        else:
            masked = True
    else:
        masked = True

    dims = img.shape

    # where to store
    trailing_thetas = np.zeros_like(img, dtype='float64')
    leading_thetas = np.zeros_like(img, dtype='float64')

    # maybe implement a small angle correction
    for i, (xx, xy, yy) in enumerate(np.nditer([Hxx, Hxy, Hyy])):

        # grab the (x,y) coordinate of the hxx, hxy, hyy you're using
        subs = np.unravel_index(i, dims)

        # ignore masked areas (if masked array)
        if masked and mask[subs]:
            continue

        h = np.array([[xx, xy], [xy, yy]])  # per-pixel hessian
        l, v = eig(h)  # eigenvectors as columns

        # reorder eigenvectors by (increasing) magnitude of eigenvalues
        v = v[:, np.argsort(np.abs(l))]
```

```
206
207         # angle between each eigenvector and positive x-axis
208         # arccos of first element (dot product with (1,0) and eigvec is already
209         # normalized)
210         trailing_thetas[subs] = np.arccos(v[0, 0])  # first component of each
211         leading_thetas[subs] = np.arccos(v[0, 1])  # first component of each
212
213     if masked:
214         leading_thetas = ma.masked_array(leading_thetas, mask)
215         trailing_thetas = ma.masked_array(trailing_thetas, mask)
216
217     return trailing_thetas, leading_thetas
218
219
220 if __name__ == "__main__":
221
222     pass
223
224     #from get_base import get_preprocessed
225     #import matplotlib.pyplot as plt
226     #from functools import partial
227     #from fpd import get_targets
228     #b = partial(plt.imshow, cmap=plt.cm.Blues)
229     #sp = partial(plt.imshow, cmap=plt.cm.spectral)
230     #s = plt.show
231
232     #import time
233
234     #img = get_preprocessed(mode='G')
235
236     #for sigma in [0.5, 1, 2, 3, 5, 10]:
237
238     #    print('-'*80)
239     #    print('σ=',sigma)
240     #    print('calculating hessian H')
241
242     #    tic = time.time()
243     #    H = hessian_matrix(img, sigma=sigma)
244
245     #    toc = time.time()
246     #    print('time elapsed: ', toc - tic)
247     #    tic = time.time()
248     #    print('calculating hessian via FFT (F)')
249     #    h = fft_hessian(img, sigma)
250
251     #    toc = time.time()
252     #    print('time elapsed: ', toc - tic)
253     #    tic = time.time()
254     #    print('calculating principal curvatures for σ={}'.format(sigma))
255     #    K1,K2 = principal_curvatures(img, sigma=sigma, H=H)
256     #    toc = time.time()
257     #    print('time elapsed: ', toc - tic)
258     #    tic = time.time()
259     #    print('calculating principal curvatures for σ={} (fast)'.format(sigma))
260     #    k1,k2 = principal_curvatures(img, sigma=sigma, H=h)
261
262     #    toc = time.time()
263     #    print('time elapsed: ', toc - tic)
264     #    tic = time.time()
265
266     #    #####
267
268     #    print('calculating targets for σ={}'.format(sigma))
269     #    T = get_targets(K1,K2, threshold=False)
```

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

listings/extract_NCS_pcsvn.py

```
1  #!/usr/bin/env python3
2
3  """
4  This is the main program. It approximates the PCSVN of a list of samples.
5  It does not do network completion.
6
7  """
8
9  from placenta import (get_named_placenta, cropped_args, cropped_view,
10                        list_placentas, list_by_quality, open_typefile,
11                        open_tracefile, add_ucip_to_mask, measure_ncs_markings)
12
13 from merging import nz_percentile, apply_threshold, sieve_scales, view_slices
14
15 from scoring import (compare_trace, rgb_to_widths, merge_widths_from_traces,
16                      filter_widths, mcc, confusion, skeletonize_trace)
17
18 from pcsvn import extract_pcsvn, scale_label_figure, get_outname_lambda
19 from preprocessing import inpaint_hybrid
20
21 import numpy as np
22 import numpy.ma as ma
23
24 import matplotlib.pyplot as plt
25
26 import os.path
27 import os
28 import json
29 import datetime
30 import pandas
```

```
31
32 # for some post_processing, this needs to be moved elsewhere
33 from skimage.filters import sobel
34 from frangi import frangi_from_image
35 from plate_morphology import dilate_boundary, mask_cuts_simple
36 from skimage.morphology import remove_small_holes, remove_small_objects
37 from skimage.segmentation import random_walker
38 from postprocessing import random_walk_fill, random_walk_scalewise
39
40
41 # INITIALIZE SAMPLES ____________________________________________________
42 #   There are several ways to initialize samples. Uncomment one.
43
44 # load all 201 samples
45 # placentas = list_placentas('T-BN')
46 # load placentas from a certain quality category 0=good, 1=okay, 2=fair, 3=poor
47
48 #placentas = list_by_quality(2)
49 #placentas.extend(list_by_quality(3))
50
51 placentas = list_by_quality(0, N=1)
52 # load from a file (sample names are keys of the json file)
53 # placentas = list_by_quality(json_file='manual_batch.json')
54
55 # for a single named sample, use a 1 element list.
56 # placentas = ['T-BN0204423.png']
57
58 #placentas = ['barium1.png',]
59 # RUNTIME OPTIONS _______________________________________________________
60 #   Where to save and whether or not to use old targets.
61
62 MAKE_NPZ_FILES = False # pickle frangi targets if you can
63 USE_NPZ_FILES = False # use old npz files if you can
64 NPZ_DIR = 'output/181204-test'  # where to look for npz files
65 OUTPUT_DIR = 'output/181204-test'  # where to save outputs
66
67 # add in a meta switch for verbosity (or levels)
68 #VERBOSE = False
69
70 # FRANGI / EXTRACT_PCSVN OPTIONS ________________________________________
71
72 # Find bright curvilinear structure against a dark background -> True
73 # Find dark curvilinear structure against a bright background -> False
74 # DARK_BG -> ignore and return signed Frangi scores
75 DARK_BG = False
76
77 # Along with the above, this will return "opposite" signed frangi scores.
78 # if this is True, then DARK_BG controls the "polarity" of the filter.
79 # See frangi.get_frangi_targets for details.
80 SIGNED_FRANGI = True
81
82 # Do not calculate hessian scores close to the boundary (this is important
83 # mainly in terms of ensuring that the hessian is very large on the edge of
84 # the plate (which would influence gamma calculation)
85 DILATE_PER_SCALE = True
86
87 # Attempt to remove glare from sample (some are OK, some are bad)
88 FLATTEN_MODE = 'L' # 'G' or 'L'
89 REMOVE_GLARE = True
90
91 # Which scales to use
92 SCALE_RANGE = (-1.5, 3.2); SCALE_TYPE = 'logarithmic'
93 #SCALE_RANGE = (.2, 12); SCALE_TYPE = 'linear'
94 N_SCALES = 20
```

```
95
96 # use this if you want to use a custom argument (comment out the above)
97 SCALES = None
98 #SCALE_RANGE = None, SCALE_TYPE == 'custom'
99
100
101 # Explicit Frangi Parameters (pass a scalar, array as long as scales)
102 BETAS = 0.35
103 GAMMAS = 0.5
104 CS = None # pass scalar, array, or None
105 ALPHAS = None # set custom alphas or calculate later
106 FIXED_ALPHA = .4
107
108 RESCALE_FRANGI = True
109 GRADIENT_FILTER = False
110
111
112 # Scoring Decisions (don't need to touch these)
113 UCIP_RADIUS = 60  # area around the umbilical cord insertion point to ignore
114
115
116
117
118 # CODE BEGINS HERE ____________________________________________________________
119
120 if SCALES is None:
121     if SCALE_TYPE == 'linear':
122         scales = np.linspace(*SCALE_RANGE, num=N_SCALES)
123     elif SCALE_TYPE == 'logarithmic':
124         scales = np.logspace(*SCALE_RANGE, num=N_SCALES, base=2)
125 else:
126     scales = SCALES
127     SCALE_TYPE = 'custom'  # this and the next three lines are just for logging
128     N_SCALES = len(SCALES)
129     SCALES = (min(SCALES), max(SCALES))
130
131 mccs = dict()  # empty dict to store MCC's of each sample
132 pncs = dict()  # empty dict to store percent network covered for each sample
133 precisions = dict()
134
135 n_samples = len(placentas)
136
137 if not os.path.exists(OUTPUT_DIR):
138     os.makedirs(OUTPUT_DIR)
139
140 print(n_samples, "samples total!")
141
142 for i, filename in enumerate(placentas):
143
144     print('*'*80)
145     print(f'extracting PCSVN of {filename}\t ({i} of {n_samples})')
146
147     # --- Setup, Preprocessing, Frangi Filter (it's mixed up) -----------------
148
149     raw_img = get_named_placenta(filename, mode=FLATTEN_MODE)
150     cimg = open_typefile(filename, 'raw')
151
152     ucip = open_typefile(filename, 'ucip')
153
154     if REMOVE_CUTS:
155         #img, has_cut = mask_cuts_simple(raw_img, ucip, return_success=True)
156         #img.data[img.mask] = 0 # actually zero out that area
157         print("removing cuts doesn't do anything anymore")
158         pass
```

```
    else:
        img = raw_img.copy()

    if REMOVE_GLARE:
        img = inpaint_hybrid(img)


    if USE_NPZ_FILES:
        # find the first npz file with the sample name in it in the
        # specified directory.
        stub = filename.rstrip('.png')
        for f in os.scandir(NPZ_DIR):
            if f.name.endswith('npz') and f.name.startswith(stub):
                npz_filename = os.path.join(NPZ_DIR, f.name)
                print(f'using the npz file {npz_filename}')
                break  # we'll just use the first one we can find.
        else:
            print(f'no npz file found for {filename}.')
            npz_filename = None
    else:
        npz_filename = None

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

    if npz_filename is not None:

        F = np.load(npz_filename)['F']

        # in case preprocessing happens inside extract_pcsvn, do it out here

        print('successfully loaded the frangi targets!')

    else:
        print('finding multiscale frangi targets')

        # F is an array of frangi scores of shape (*img.shape, N_SCALES)
        F, jfile = extract_pcsvn(img, filename, dark_bg=DARK_BG, beta=BETAS,
                                 scales=scales, gamma=GAMMAS, c=CS,
                                 kernel='discrete', dilate_per_scale=True,
                                 verbose=False, signed_frangi=SIGNED_FRANGI,
                                 generate_json=True, output_dir=OUTPUT_DIR,
                                 rescale_frangi=RESCALE_FRANGI,
                                 gradient_filter=GRADIENT_FILTER)

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

    # - Multiscale Analysis, etc ------------------------------------------

    # This is the maximum frangi response over all scales at each location
    Fmax = F.max(axis=-1)

    print("...making outputs")

    if ALPHAS is None:
        print("thresholding ALPHAS with top 5% scores at each scale")
        ALPHAS = np.array([nz_percentile(F[..., k], 95.0)
                           for k in range(N_SCALES)]
                          )
    # the maximum value of the entire image at each scale
```

```
    scale_maxes = np.array([F[...,i].max() for i in range(F.shape[-1])])

    table = pandas.DataFrame(np.dstack((scales, ALPHAS, scale_maxes)).squeeze(),
                             columns=('σ', 'α_p', 'max(F_σ)'))

    print(table)
    # threshold the responses at each of these values and get labels of max

    # --- Segmentation Postprocessing -----------------------------------------------

    # get the main (boolean) tracefile and the RGB tracefiles
    trace = open_tracefile(filename, as_binary=True)
    A_trace = open_typefile(filename, 'arteries')
    if A_trace is None:
        # there are no special trace files for this sample
        skeltrace = skeletonize_trace(trace)
    else:
        V_trace = open_typefile(filename, 'veins')
        skeltrace = skeletonize_trace(A_trace, V_trace)

        # get a matrix of pixel widths in the trace
        widths = merge_widths_from_traces(A_trace, V_trace, strategy='arteries')

    # find cord insertion point and resolution of the image
    ucip_midpoint, resolution = measure_ncs_markings(ucip)
    # if verbose:
    # print(f"The umbilicial cord insertion point is at {ucip_midpoint}")
    # print(f"The resolution of the image is {resolution} pixels per cm.")

    if ucip_midpoint is None:
        ucip_mask = img.mask
    # mask anywhere close to the UCIP
    else:
        ucip_mask = add_ucip_to_mask(ucip_midpoint,
                                     radius=int(UCIP_RADIUS), mask=img.mask)

    # The following are examples of things you can do:

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

    # trace ignoring largest vessels (19 pixels wide)
    # trace_smaller_only = filter_widths(min_widths, min_width=3, max_width=17)
    # trace_smaller_only != 0


    # -- Segmentation Strategies -------------------------------------------------

    # strawman
    from skimage.filters import threshold_mean
    from functools import partial

    confusion_matrix = partial(confusion, truth=trace, bg_mask=ucip_mask)
    mcc_with_counts = partial(mcc, truth=trace, bg_mask=ucip_mask,
                              return_counts=True)
    percent_network_coverage = lambda a: np.sum(skeltrace&a)/np.sum(skeltrace)
    precision = lambda t: int(t[0]) / int(t[0] + t[2])
    V = np.transpose(F, axes=(2, 0, 1))


    approx_sm = threshold_mean(img.filled(img.compressed().mean()))
    mcc_sm, counts_sm = mcc_with_counts(approx_sm)
    prec_sm = precision(counts_sm)
```

```
    confuse_sm = confusion_matrix(approx_sm)
    pnc_sm = percent_network_coverage(approx_sm)

    approx_PF, labs_PF = apply_threshold(F, ALPHAS, return_labels=True)
    mcc_PF, counts_PF = mcc_with_counts(approx_PF)
    confuse_PF = confusion_matrix(approx_PF)
    pnc_PF = percent_network_coverage(approx_PF)
    prec_PF = precision(counts_PF)

    approx_FA, labs_FA = apply_threshold(F, FIXED_ALPHA)
    mcc_FA, counts_FA = mcc(approx_FA, trace, ucip_mask, return_counts=True)
    confuse_FA = confusion_matrix(approx_FA)
    pnc_FA = percent_network_coverage(approx_FA)
    prec_FA = precision(counts_FA)

    approx_RW, labs_RW = random_walk_scalewise(F, FIXED_ALPHA, return_labels=True)
    confuse_RW = confusion_matrix(approx_RW)
    mcc_RW, counts_RW = mcc_with_counts(approx_RW)
    pnc_RW = percent_network_coverage(approx_RW)
    prec_RW = precision(counts_RW)

    sieved = sieve_scales(V, 98, 95)
    approx_S, labs_S = (sieved != 0), sieved
    confuse_S = confusion_matrix(approx_S)
    mcc_S, counts_S = mcc_with_counts(approx_S)
    pnc_S = percent_network_coverage(approx_S)
    prec_S = precision(counts_S)

    # PUT MARGIN ADD IN HERE
    #
    #
    #
    #

    mccs[filename] =  (mcc_PF, mcc_FA, mcc_RW, mcc_S, mcc_sm)
    pncs[filename] = (pnc_PF, pnc_FA, pnc_RW, pnc_S, pnc_sm)
    precisions[filename] = (prec_PF, prec_FA, prec_RW, prec_S, prec_sm)

    scoretable = pandas.DataFrame(np.vstack((mccs[filename], pncs[filename],
                                             precisions[filename])),
                                  columns=('PF', 'FA', 'RW', 'PS', 'SM'),
                                  index=('MCC', 'skel coverage', 'precision'))

    print(scoretable)
    print('\n\n')
    print(scoretable.to_latex())


    # use only some scales
    #approx_LO, labs_LO = apply_threshold(F[:,:, LO_offset:], ALPHAS[LO_offset:])
    # fix labels to incorporate offset
    #labs_LO = (labs_LO != 0)*(labs_LO + LO_offset)
    #confuse_LO = confusion(approx_LO, trace, bg_mask=ucip_mask)

    #TP, TN, FP, FN = counts

    # this all just verifies that the 4 categories were added up
    # correctly and match the total number of pixels in the reported
    # placental plate.
    #total = np.sum(~ucip_mask)
    #print(f'TP: {TP}\t TN: {TN}\nFP: {FP}\tFN: {FN}')
    # just a sanity check
    #print(f'TP+TN+FP+FN={TP+TN+FP+FN}\ttotal pixels={total}')

```

```
351
352     #approx_rw, markers, margins_added = random_walk_fill(img, Fmax, .3, .01,
353     #                                                     DARK_BG)
354
355
356
357
358     #view_slices(F[crop], axis=-1, scales=scales)
359
360     # --- Generating Visual Outputs-------------------------------------------
361
362     # cmap and the "set bad" argument / mask color
363     SCALE_CMAP = ('plasma', (1,1,1,1))
364
365     crop = cropped_args(img)  # these indices crop out the mask significantly
366
367     fmax_colors = plt.cm.plasma
368     fmax_colors.set_bad('k', 1)
369
370     # save the raw, unaltered image
371     plt.imsave(outname('0_raw'), cimg[crop])
372
373     # save the preprocessed image
374     plt.imsave(outname('1_img'), img[crop].filled(0), cmap=plt.cm.gray)
375
376     # save the maximum frangi output over all scales
377     plt.imsave(outname('2_fmax'), ma.masked_where(Fmax==0,Fmax)[crop], vmin=0,
378                vmax=1.0, cmap=fmax_colors)
379
380     # only save the colorbar the first time
381     #save_colorbar = (i==0)
382     #scale_label_figure(labs, scales, crop=crop,
383     #                   savefilename=outname('3_labeled'), image_only=True,
384     #                   save_colorbar_separate=save_colorbar,
385     #                   basecolor=SCALE_CMAP[1], base_cmap=SCALE_CMAP[0],
386     #                   output_dir=OUTPUT_DIR)
387
388     #plt.imsave(outname('4_confusion'), confuse[crop])
389
390     #scale_label_figure(labs_rw, scales, crop=crop,
391     #                   savefilename=outname('A_labeled_rw'), image_only=True,
392     #                   save_colorbar_separate=save_colorbar,
393     #                   basecolor=SCALE_CMAP[1], base_cmap=SCALE_CMAP[0],
394     #                   output_dir=OUTPUT_DIR)
395
396
397     #plt.imsave(outname('7_confusion_FA'), confuse_FA[crop])
398     #plt.imsave(outname('B_confusion_rw'), confuse_rw[crop])
399     ##plt.imsave(outname('A_markers_rw'), markers[crop])
400     ##plt.imsave(outname('9_margin_for_rw'), confuse_margins[crop])
401
402
403
404     st_colors = {
405         'TN': (79,79,79),  # true negative# 'f7f7f7'
406         'TP': (0, 0, 0),  # true positive  # '000000'
407         'FN': (201,53,108),  # false negative # 'f1a340' orange
408         'FP': (92,92,92),  # false positive
409         'mask': (247, 200, 200)  # mask color (not used in MCC calculation)
410     }
411
412     #plt.imsave(outname('5_coverage'), confusion(approx, skeltrace,
413     #                                            colordict=st_colors)[crop])
414     #plt.imsave(outname('8_coverage_FA'), confusion(approx_FA, skeltrace,
```

```
    #                                                 colordict=st_colors)[crop])
    #plt.imsave(outname('C_coverage_rw'), confusion(approx_rw, skeltrace,
    #                                                 colordict=st_colors)[crop])

    # make the graph that shows what scale the max was pulled from

    #scale_label_figure(labs_FA, scales, crop=crop,
    #                   savefilename=outname('6_labeled_FA'), image_only=True,
    #                   basecolor=SCALE_CMAP[1], base_cmap=SCALE_CMAP[0],
    #                   save_colorbar_separate=False, output_dir=OUTPUT_DIR)


    #scale_label_figure(labs_S, scales, crop=crop,
    #                   savefilename=outname('D_labeled_S'), image_only=True,
    #                   basecolor=SCALE_CMAP[1], base_cmap=SCALE_CMAP[0],
    #                   save_colorbar_separate=False, output_dir=OUTPUT_DIR)

    #plt.imsave(outname('E_confusion_S'), confuse_S[crop])




    ### THIS IS ALL A HORRIBLE MESS. FIX IT
    # why don't you just return the dict instead
    with open(jfile, 'r') as f:
        slog = json.load(f)

    c2d = lambda t: dict(zip(('TP','TN', 'FP', 'FN'), [int(c) for c in t]))

    slog['counts'] = c2d(counts)
    slog['counts_FA'] = c2d(counts_FA)
    slog['counts_rw'] = c2d(counts_rw)
    slog['counts_S'] = c2d(counts_S)
    slog['pnc'] = pncs[filename]
    slog['mcc'] = mccs[filename]
    slog['scale_maxes'] = list(scale_maxes)
    slog['ALPHAS'] = list(ALPHAS)
    slog['precision'] = precisions[filename]


    with open(jfile, 'w') as f:
        json.dump(slog, f)

    plt.close('all')

# Post-run Meta-Output and Logging ____________________________________________

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

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

runlog = {
    'time': timestring,
    'DARK_BG': DARK_BG,
    'DILATE_PER_SCALE': DILATE_PER_SCALE,
    'SCALE_RANGE': SCALE_RANGE,
    'SCALE_TYPE' : SCALE_TYPE,
    'N_SCALES': N_SCALES,
    'scales': list(scales),
    'ALPHAS': list(ALPHAS),
    'BETAS': None,
    'use_npz_files': False,
    'remove_glare': REMOVE_GLARE,
```

```
    'files': list(placentas),
    'MCCS': mccs,
    'PNCS': pncs,
    'precisions': precisions
}

# save to a json file
with open(mccfile, 'w') as f:
    json.dump(runlog, f, indent=True)
```

listings/find_plate.py

```
# coding: utf-8
#filename = 'T-BN4981652.png'
#img = get_named_placenta(filename)
#crop = cropped_args(img)
#plt.imshow(img)
#s = plt.show
#s()
#plt.imshow(img > .9*img.mask)
#plt.show()
#plt.imshow(img > .9*img.max())
#plt.show()
#plt.imshow(img > .8*img.max())
#plt.show()
#plt.imshow(img > .75*img.max())
#plt.show()
#from skimage.filters import sobel
#sobel(img)
#plt.imshow(_)
#plt.show()
#from skimage import filters
#filters.laplace(img)
#plt.imshow(_)
#plt.show()
#dilate_boundary(img, 20)
#plt.imshow(_)
#plt.show()
#filters.laplace(dilate_boundary(img, 30))
#plt.imshow(_)
#plt.show()
#filters.laplace(dilate_boundary(img, 30) == 0)
#plt.imshow(_)
#s()
#filters.laplace(dilate_boundary(img, 30))==0
#plt.imshow(_)
#plt.show()
#from skimage import morphology as mph
#from frangi import frangi_from_image
#fft_gradient(img, sigma=20)
#plt.imshow(_)
#plt.show()
#fft_gradient(img, sigma=15)
#plt.imshow(_)
#plt.show()
#raw_img
#rimg
from placenta import open_typefile
open_typefile(filename, 'raw')
plt.imshow(_)
plt.show()
raw = open_typefile(filename, 'raw')
plt.imshow(raw[...,1])
```

```
plt.show()
plt.imshow(fft_gradient(raw[...,1],sigma=15) )
plt.show()
plt.imshow(fft_gradient(raw[...,1],sigma=.01))
plt.show()
plt.imshow(fft_gradient(raw[...,1],sigma=.001))
plt.show()
from skimage.segmentation import watershed
get_ipython().run_line_magic('pinfo', 'watershed')
marks = np.zeros(img.shape, np.int32)
marks[0,0] = 1
marks[img.shape//2]
marks[img.shape[0]//2,img.shape[1]//2]
marks[img.shape[0]//2,img.shape[1]//2] = 2
watershed(fft_gradient(raw[...,1],sigma=.01))
watershed(fft_gradient(raw[...,1],sigma=.01), marks)
plt.imshow(_)
plt.show()
watershed(fft_gradient(raw[...,1],sigma=.001), marks)
plt.show()
plt.imshow(_)
plt.show()
g = fft_gradient(raw[...,1],sigma=.01)
plt.imshow(g)
plt.show()
g = fft_gradient(raw[...,1],sigma=.001)
plt.imshow(g)
plt.show()
from skimage.segmentation import find_boundaries
find_boundaries(g)
plt.imshow(_)
plt.show()
plt.imshow(g>g.mean())
plt.show()
marks[g>g.mean()] = 2
watershed(g, marks)
plt.imshow(_)
plt.show()
from scipy.ndimage import find_objects
find_objects(watershed(g,marks))
help(find_objects)
from scipy.ndimage import label
get_ipython().run_line_magic('pinfo', 'label')
label(watershed(g,marks))
help(label)
plt.imshow(watershed(g,marks))
plt.show()
plt.imshow(watershed(g,marks))
plt.show()
from skimage.morphology import binary_erosion
plt.imshow(watershed(g,marks), vmin=0, vmax=2)
plt.show()
w = watershed(g,marks)
plt.imshow(w)
plt.show()
from skimage.morphology import binary_erosion, disk
binary_erosion(w, disk(10))
plt.imshow(_)
plt.show()
plt.imshow(w-1)
plt.show()
plt.imshow(binary_erosion(w-1, disk(10)))
plt.show()
# oops need to make watershed thing binary. w-1 is hacky but works
```

```
eroded = binary_erosion(w-1, disk(10))
find_boundaries(eroded)
plt.imshow(_)
plt.show()
import scipy.ndimage as ndi
ndi.label(eroded)
plt.imshow(_[0])
plt.show()
labs, nlabs = ndi.label(eroded)
for i in range(nlabs):
    print(i, np.sum(labs==i))

labs==1
plt.imshow(_)
plt.show()
plt.imshow(np.logical_xor(labs==1, ~img.mask))
plt.show()
sum(~img.mask)
np.sum(~img.mask)
np.sum(labs==1)
```

listings/frangi_graphing.py

```
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
from numpy import exp
from skimage.io import imread
from skimage.util import montage
from itertools import product

def s(x, gamma):
    """normalized structureness factor.
    x is the ratio of the input to the maximum possible structureness factor
    (smax) at that scale (which would cancel out with the c in the denominator)
    """
    return (1 - exp(-x**2 / (2*gamma**2))) / (1 - exp(-1/(2*gamma**2)))


def r(y, beta):
    """normalized anistropy factor
    y is the ratio |k1 / k2|, so y=0 corresponds to perfectly isotropic
    and y->1 corresponds to highly anisotropic
    """
    return np.exp(-y**2 / (2*beta**2))

#plt.rc('text', usetex=True)
#plt.rc('font', family='serif')


dom = np.linspace(0, 1)

plt.close('all')

# show dependence of structureness factor on its parameter
for gamma in [0.1, 0.25, 0.35, 0.5, 0.9, 1, 2, 10, 1000]:
    plt.plot(dom, s(dom, gamma), label=r'$\gamma = $'.format(gamma))

plt.ylabel(r'$\left(1 - \exp\left\{\frac{-S}{2(\gamma S_{max})^2}\right\}\right)$',
           fontsize=14)
plt.xlabel(r'$(S/S_{max})$', fontsize=14)
```

```
plt.title(r'Dependence of Structureness Factor on Parameter $\gamma = (c/S_{max})$')
plt.legend()

#plt.show()
plt.close('all')

for beta in [0.1, 0.25, 0.35, 0.5, 0.9, 1, 2, 10, 1000]:
    plt.plot(dom, r(dom, beta), label=r'$\beta = $'.format(beta))

plt.ylabel(r'$\exp\left\{\frac{-A}{2\beta^2}\right\}$',
           fontsize=14)
plt.xlabel(r'$A = |\lambda_1/\lambda_2|$', fontsize=14)
plt.title(r'Dependence of Anisotropy Factor on Parameter $\beta$')
plt.legend()

#plt.show()
plt.close('all')

prange = [0.1, 0.25, 0.5, 0.9, 1, 1.5]

for n, (beta, gamma) in enumerate(product(prange, prange)):

    fig = plt.figure(figsize=(8,5))
    ax = fig.gca(projection='3d')
    X, Y = np.meshgrid(dom, dom)

    Z = r(X,beta)*s(Y,gamma)

    surf = ax.plot_surface(X,Y,Z, cmap='coolwarm', linewidth=0)

    ax.set_xlabel(r'$|\lambda_1/\lambda_2|$')
    ax.set_ylabel(r'$(S/S_{\max})$ ')

    ax.set_title(r"Rescaled Frangi filter, "
                 r"$\beta = ,\gamma = $".format(beta,gamma))

    #fig.colorbar(surf, shrink=0.5, aspect=5)
    fig.tight_layout()

    plt.savefig(f'demo_output/frangi3d/{n}.png', dpi=300)

    plt.close()

imgs = [imread(f'demo_output/frangi3d/{n}.png') for n in range(36*1)]
imgs = np.stack(imgs)

for n in range(6):
    plt.imsave(f'demo_output/frangi3d/frangi3dpart{n}.png',
               montage(imgs[(n*6):((n+1)*6)], multichannel=True,
                       grid_shape=(3,2)))
```

listings/frangi.py

```
import numpy as np
import numpy.ma
from hfft import fft_hessian, fft_gradient
from diffgeo import principal_curvatures
from plate_morphology import dilate_boundary
from merging import nz_percentile

def frangi_from_image(img, sigma, beta=0.5, gamma=0.5, c=None, dark_bg=True,
```

```
                    dilation_radius=None, kernel=None, signed_frangi=False,
                    return_debug_info=False, verbose=False,
                    rescale_frangi=False, gradient_filter=False):
    """Calculate the (uniscale) Frangi vesselness measure on a grayscale image

    Parameters
    ----------
    img: ndarray or ma.MaskedArray
        a one-channel image. If this is a masked array (preferred), ignore the
        masked regions of the image
    sigma: float
        Standard deviation of the gaussian, used to calculate derivatives.
    beta: float, optional
        The anisotropy parameter of the Frangi filter (default is 0.5)
    gamma: float, optional
        Scaling factor for the structureness parameter of the Frangi
        filter. The structureness parameter will be set to gamma * maximum
        of the hessian norm. (Default is 0.5)
    c: float or None, optional
        The strutureness parameter of the Frangi filter. If this is set then
        gamma is ignored. (Default is None).
    dilation_radius: int or None
        If dilation radius is supplied, then areas within that amount of pixels
        will not be calculated. This is preferable in certain contexts,
        especially when there is a dark background and dark_bg=True. This is
        especially recommended for small sigmas and when gamma is not provided.
        None to forgo this procedure (default).  A mask must be supplied for
        this to make sense.
    dark_bg: boolean or None
        if True, then frangi will select only for bright curvilinear
        features; if False, then Frangi will select only for dark
        curvilinear structures. if None instead of a bool, then curvilinear
        structures of either type will be reported.
    signed_frangi: bool, optional
        if signed is True, the result will be the same as if dark_bg is set
        to None, except that the sign will change to match the desired
        features. See example below.
    return_debug info: bool, optional
        will return a large dict consisting of several large matrices,
        calculated hessian, etc.

        scale_dict = {'sigma': sigma,
                      'beta': beta,
                      'gamma': gamma,
                      'c': c,
                      'H': hesh,
                      'F': targets,
                      'k1': k1,
                      'k2': k2,
                      'border_radius': dilation_radius
                      }
    Returns
    -------
    ...

    Notes
    -----
    Although default is 0.5, this means that the structureness factor of the
    Frangi score will only be 0.86 at its maximum. Larger values of gamma
    will only dampen the frangi filter more. Smaller values toward 0 will
    result in a "looser" filter. For example, if gamma = .25, then the
    maximum score is (1-exp{-8}) around .999 (it may be desirable that the
    franginess score should be able to achieve a score of 1).

```

```
    This function will accept 0 an input, and the structureness factor will
    be set to 1 everywhere (the limiting case as gamma -> 0)

    Frangi structureness factor is (1 - exp((-S**2)/(2*c**2))
    """
    hesh = fft_hessian(img, sigma, kernel=kernel)  # the triple (Hxx,Hxy,Hyy)
    # calculate principal curvatures with |k1| <= |k2|



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

    if dilation_radius is not None:
        # pass None to just get the mask back
        collar = dilate_boundary(None, radius=dilation_radius, mask=img.mask)

        # get rid of "bad" K values before you calculate gamma and Frangi
        k1[collar] = 0
        k2[collar] = 0
        hesh[0][collar] = 0
        hesh[1][collar] = 0
        hesh[2][collar] = 0
    else:
        collar = img.mask.copy()


    # no need to set gamma or c anymore. will be set inside get_frangi_targets
    #if c is None:
    # Frangi suggested 'half the max Hessian norm' as an empirical
    # half the max spectral radius is easier to calculate so do that
    # shouldn't be affected by mask data but should make sure the
    # mask is *well* far away from perimeter
    # we actually calculate half of max hessian norm
    # using frob norm = sqrt(trace(AA^T))
    # alternatively you could use gamma = .5 * np.abs(k2).max()
    #hnorm = hessian_norm(hesh, mask=collar)
    #print(f'σ={sigma:2f}')
    #gamma0 = .5*hessian_norm(hesh).max()
    #print(f'\t{gamma0:.5f} = frob-norm γ pre-dilation')

    #gamma1 = .5*hessian_norm(hesh, mask=collar).max()
    #print(f'\t{gamma1:.5f} = frob-norm γ post-collar dilation {dilation_radius}')
    #l2gamma = .5*np.max(np.abs(k2))
    #print(f'\t{l2gamma:.5f} =  from L2-norm γ (K2 with collar)')

    #hdilation = int(max(np.ceil(sigma),10))
    #hcollar = dilate_boundary(None, radius=hdilation, mask=img.mask)
    #gamma = .5 * max_hessian_norm(hesh, mask=hcollar)

    #print(f'\t{gamma:.5f} = γ post-hdilation (radius {hdilation}) (old γ)')

    #print('changing γ to L2-norm with collar')
    #gamma = max(gamma1, l2gamma, gamma, gamma0)

    # wish this scaled a little better

    # a very large gamma here will make the Frangi score zero
    # a very small gamma means that we are artificially inflating the
    # structureness measure
    #import matplotlib.pyplot as plt
    #plt.imshow(hnorm*(~collar))
    #plt.show()
    #print(hnorm[~collar].min(), hnorm[~collar].max())
    #if hnorm[~collar].max() < 0.1:
```

```
    #    print(f'max hessian norm is very small at this scale ({sigma},{hnorm[~collar].max
    #            'you should maybe skip this scale')
    #elif hnorm[~collar].min() <  0.001:
    #    # only trigger if the first one didn't
    #    print(f'min hessian norm is very small at this scale ({sigma}, {hnorm[~collar].mi
    #          'be carefully of artificially inflated scores')
    #S = np.sqrt(k1**2 + k2**2)
    #import matplotlib.pyplot as plt
    #plt.imshow(S)
    #plt.show()
    #plt.close()
    #print('max hessian norm (Frob): ', hnorm.max())
    #print('max structureness: ', S.max())
    #c = gamma*S.max()

    if verbose:
        print(f'finding Frangi targets with β={beta} and γ={c:.2}')

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

    if gradient_filter:
        # obviously you could compute this at the same time as the hessian :/

        g = fft_gradient(img, sigma)
        g = dilate_boundary(g, radius=20, mask=img.mask)

        # you could technically pass a function to switch between these
        # behaviors
        low_g = (g < nz_percentile(g, 50)).filled(0)

        targets[~low_g] = 0

    if not return_debug_info:
        return targets
    else:

        # for logging we have to recalculate this
        if c is not None:
            c = gamma * max(np.sqrt(k1**2 + k2**2))

        scale_dict = {'sigma': sigma,
                      'beta': beta,
                      'gamma': gamma,
                      'c': c,
                      'H': hesh,
                      'F': targets,
                      'k1': k1,
                      'k2': k2,
                      'border_radius': dilation_radius
                      }

        return targets, scale_dict


def get_frangi_targets(K1, K2, beta=0.5, gamma=0.5, c=None,
                       dark_bg=True, signed=False, rescale_frangi=False):
    """Calculate the Frangi vesselness measure from eigenvalues.

    Parameters
    ----------
        K1, K2 : ndarray (each)
            each is an ndarray of eigenvalues (approximated principal
```

```
201             curvatures) for some image.
202         beta: float
203             the anisotropy parameter (default is 0.5)
204         gamma: float or None
205             Scaling factor for the the structureness parmeter. The structurness
206             parameter c will be set to gamma times the maximum of the Hessian
207             norm, sqrt(K1**2 + K2**2). Default is 0.5
208         c: float or None
209             The frangi structurness parameter. If this is set, gamma above will
210             be ignored.
211         dark_bg: boolean or None
212             if True, then frangi will select only for bright curvilinear
213             features; if False, then Frangi will select only for dark
214             curvilinear structures. if None instead of a bool, then curvilinear
215             structures of either type will be reported.
216         signed: boolean
217             if signed is True, the result will be the same as if dark_bg is set
218             to None, except that the sign will change to match the desired
219             features. See example below.
220
221     Returns
222     -------
223         F: ndarray, same shape as K1
224             the Frangi vesselness measure.
225
226     Notes
227     -----
228     If beta or gamma are set to 0, then the frangi anisotropy factor will be
229     set to 0 or 1 everywhere (which is the limiting case as beta->0 or
230     gamma->0) You can set beta = 'inf' or np.inf to set anisotropy factor to 1.
231
232     Examples
233     --------
234     >>>f1 = get_frangi_targets(K1,K2, dark_bg=True, signed=True)
235     >>>f2 = get_frangi_targets(K1,K2, dark_bg=False, signed=True)
236     >>>np.all(f1 == -f2)
237     True
238
239     >>> F = get_frangi_targets(K1,K2, gamma=0.5)
240     >>> Falt = get_frangi_targets(K1,K2, c=0.5*np.sqrt(K1**2 + K2**2))
241     >>> np.all(F == Falt)
242     True
243     """
244
245     A = anisotropy(K1, K2, beta=beta)
246     S = structureness(K1, K2, gamma=gamma, c=c)
247
248     anisotropy_factor = np.exp(-A)
249     structureness_factor = (1 - np.exp(-S))
250
251     F = anisotropy_factor * structureness_factor
252
253     if rescale_frangi:
254         if c is not None:
255             # more like will not
256             print('c was set to an arbitrary value. cannot rescale')
257         else:
258             if gamma==0:
259                 # prevent division by zero
260                 max_theoretical = 1
261             else:
262                 max_theoretical = (1 - np.exp(-1/(2*gamma**2)))
263             F = F / max_theoretical
264
```

```
    # now just filter/ change sign as appropriate.
    if not signed:
        # calculate the regular frangi filter
        if dark_bg is None:
            #keep F the way it is
            pass
        elif dark_bg:
            # zero responses from positive curvatures
            F = (K2 < 0)*F
        else:
            # zero responses from negative curvatures
            F = (K2 > 0)*F
    else:
        if dark_bg is None:
            # output is already signed
            pass
        elif dark_bg:
            # positive curvature spots will be made negative
            F[K2 > 0] = -1 * F[K2 > 0]
        else:
            # negative curvature spots will be made positive
            F[K2 < 0] = -1 * F[K2 < 0]

    # finally, reapply the mask if the inputs came with one
    if numpy.ma.is_masked(K1):
        F = numpy.ma.masked_array(F, mask=K1.mask)
        # zero out masked region (there can be issues here if the output is
        # signed, just makes stacking easier
        F[K1.mask] = 0

    return F


def hessian_norm(hesh, mask=None):
    """Calculate Frobenius norm of Hessian.

    Calculates the maximal value (over all pixels of the image) of the
    Frobenius norm of the Hessian. This should be the same as the square root
    of unscaled structureness.

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

    Returns
    -------
    float
    """

    hxx, hxy, hyy = hesh

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

    if mask is not None:
        hnorm[mask] = 0

    hnorm = np.sqrt(hnorm)
    return hnorm
```

```
def anisotropy(K1,K2, beta=0.5):
    """Convenience function for the exponential argument in the Frangi
    anisotropy factor.

    According to Frangi (1998) this is technically (A**2) / (2*beta**2)
    unless beta is None, in which case just A**2 is returned

    The frangi vesselness factor is formally (np.exp(-R))
    where R is what's returned by this function
    """

    A = (K1 / K2) ** 2
    #print(f'inside anisotropy, β={beta}')
    if beta == 0:
        return np.full(A.shape, np.inf)  # the limiting case as beta -> 0

    elif beta == 'inf' or np.isinf(beta):
        return np.zeros_like(A)  # the limiting case as beta -> inf

    elif beta is None:
        return A  # just return the A**2 part (why though)
    else:
        return A / (2*beta**2)


def structureness(K1, K2, gamma=0.5, c=None):
    """Convenience function for Structureness measure.
    According to Frangi (1998) this is technically S**2
    """
    S = K1**2 + K2**2


    # is c is not provided, calculate it
    if c is None:
        c = gamma * np.sqrt(S).max()  # the max Frob norm of the Hessian

    #print(f'inside structureness, γ={gamma}, c={c}')

    if c == 0:
        return np.full(S.shape, np.inf)

    elif c == 'inf' or np.isinf(c):
        return np.zeros_like(S)

    elif c is None:
        return S

    else:
        return S / (2*c**2)
```

listings/gradient_filter_demo.py

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

import numpy as np
import matplotlib.pyplot as plt
from skimage.util import img_as_float
from skimage.io import imread
from placenta import (get_named_placenta, list_by_quality, cropped_args,
                      mimg_as_float)

from frangi import frangi_from_image
```

```
from hfft import fft_gradient, fft_hessian, fft_gaussian
from merging import nz_percentile
from plate_morphology import dilate_boundary
import os.path, os

MAKE_OUTPUTS = False
OUTPUT_DIR = 'demo_output/gradient_filter_demo'

BETA = .5

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

filename = list_by_quality(N=1)[0]
img = get_named_placenta(filename)
crop = cropped_args(img)

F0 = list()
F1 = list()

scales = np.logspace(-1, 3, num=12, base=2)

for n, sigma in enumerate(scales):

    f0 = frangi_from_image(img, sigma, beta=BETA, dark_bg=False,
                           dilation_radius=20, gradient_filter=False)

    f1 = frangi_from_image(img, sigma, beta=BETA, dark_bg=False,
                           dilation_radius=20, gradient_filter=True)

    # simulate g
    #g = fft_gradient(img, sigma)
    #g = dilate_boundary(g, radius=20, mask=img.mask)
    #g = g < nz_percentile(g, 50)
    #g_filter = (~g).filled(0)

    if MAKE_OUTPUTS:
        fig, ax = plt.subplots(ncols=2, nrows=1, figsize=(10,4))

        ax[0].imshow(f0.filled(0)[crop], vmin=0, vmax=1, cmap='nipy_spectral')
        ax[0].axis('off')
        ax[0].set_title(f'Standard Frangi σ={sigma:.2f}', fontsize=10)

        ax[1].imshow(f1.filled(0)[crop], vmin=0, vmax=1, cmap='nipy_spectral')
        ax[1].axis('off')
        ax[1].set_title(f'w/ gradient filter', fontsize=10)

        #ax[2].imshow(g_filter[crop].T, cmap='nipy_spectral')
        #ax[2].axis('off')
        #ax[2].set_title(f'Gradient filter σ={sigma:.2f}')

        #plt.show()
        plt.tight_layout()
        plt.savefig(os.path.join(OUTPUT_DIR, f'gf_scale_{n:0{2}}.png'))
        plt.close('all')

    F0.append(f0)
    F1.append(f1)

F0 = np.stack(F0)
F1 = np.stack(F1)


if MAKE_OUTPUTS:
```

```
    fig, ax = plt.subplots(ncols=2, nrows=1, figsize=(10,4))

    ax[0].imshow(F0.max(axis=0).filled(0)[crop], vmin=0, vmax=1,
            cmap='nipy_spectral')
    ax[0].axis('off')
    ax[0].set_title(f'Standard Frangi F_max', fontsize=10)

    ax[1].imshow(F1.max(axis=0).filled(0)[crop], vmin=0, vmax=1,
            cmap='nipy_spectral')
    ax[1].axis('off')
    ax[1].set_title(f'w/ gradient filter', fontsize=10)

    plt.tight_layout()
    #plt.show()
    plt.savefig(os.path.join(OUTPUT_DIR, f'gf_Fmax.png'))
    plt.close('all')
```

listings/hfft_accuracy.py

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

"""
here you want to show  the accuracy of hfft.py

BOILERPLATE

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

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

pseudocode

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

zero_order_accurate = isclose(A, B, tol)

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

first_order_accurate = isclose(J_A, J_B, tol)

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

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

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

"""

from placenta import get_named_placenta, cropped_args

from itertools import combinations_with_replacement
from skimage.exposure import rescale_intensity

from hfft import fft_hessian, fft_gaussian, fft_dgk
from scipy.ndimage import gaussian_filter
import matplotlib.pyplot as plt
```

```
from placenta import show_mask, list_by_quality

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

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

from plate_morphology import dilate_boundary

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

from skimage.util import img_as_float

def plot_image_slices(arrs, fixed_axis=0, fixed_index=None, labels=None,
                      formats=None, title=None):
    """
    arrs needs to be the same shape and dimension
    could pass it to np.stack and check for a value error?

    """
    fig, ax = plt.subplots(figsize=(12,2))
    # hopefully the fixed axis is 0 or 1. this gets the other one
    it_axis = 1 if fixed_axis==0 else 0

    # if it's a tuple, make it an array, etc. etc.
    arrs = np.stack(arrs)

    # make sure we can iterate over it if there's just as single image
    if arrs.ndim < 3:
        arrs = np.expand_dims(arrs,0)

    if labels is None:
        labels = [None for a in arrs]
    if formats is None:
        formats = ['' for a in arrs]

    if fixed_index is None:
        # find halfway point of the appropriate dimension from the first array
        fixed_index = arrs[0].shape[fixed_axis] // 2

    for a, lab, fmt in zip(arrs, labels, formats):
        ax.plot(np.arange(a.shape[it_axis]),
                np.moveaxis(a, fixed_axis, 0)[fixed_index, :],
                fmt, label=lab)

    if title is not None:
        ax.set_title(title)

    # can this be at least a little object-oriented? :(
    fig.legend()

def multiway_comparison(arrs, scorefunc):

    scores = np.zeros((len(arrs), len(arrs)))

    for j in range(len(arrs)):
        for k in range(j+1,len(arrs)):
            scores[j,k] = scorefunc(arrs[j], arrs[k])

```

```
    return scores

filename = list_by_quality(0)[5]

img = get_named_placenta(filename)

# so that scipy.ndimage.gaussian_filter doesn't use uint8 precision (jesus)
img = ma.masked_array(img_as_float(img), mask=img.mask)

test_sigmas = [.3, .6, 1.0, 5.0, 15, 30, 60, 90]

for sigma in test_sigmas:

    print("*"*80, '\n\n', f"σ={sigma}")
    #print('applying standard gauss blur')

    # this is exactly how it's passed to skimage.feature.hessian_matrix(...)
    A = gaussian_filter(img.filled(0), sigma, mode='constant', cval=0)

    #print('applying fft gauss blur')
    B = fft_gaussian(img, sigma, kernel='sampled')
    C = fft_gaussian(img, sigma, kernel='discrete')

    #print('calculating first derivatives')
    # zero the masks before calculating derivates if they're masked
    Agrad = np.gradient(A)
    Bgrad = np.gradient(B)
    Cgrad = np.gradient(C)


    axes = range(img.ndim)

    #print('calculating second derivatives')
    # this is the same way it's done in skimage.feature.hessian_matrix(...)
    H_A = [np.gradient(Agrad[ax0], axis=ax1)
        for ax0, ax1 in combinations_with_replacement(axes, 2)]
    H_B = [np.gradient(Bgrad[ax0], axis=ax1)
        for ax0, ax1 in combinations_with_replacement(axes, 2)]
    H_C = [np.gradient(Cgrad[ax0], axis=ax1)
        for ax0, ax1 in combinations_with_replacement(axes, 2)]

    #print('calculating eigenvalues of hessian')
    ak1, ak2 = principal_curvatures(img, sigma=sigma, H=H_A)
    bk1, bk2 = principal_curvatures(img, sigma=sigma, H=H_B)
    ck1, ck2 = principal_curvatures(img, sigma=sigma, H=H_C)


    #RA = anisotropy(ak1,ak2)
    #RB = anisotropy(bk1,bk2)
    #RC = anisotropy(ck1,ck2)

    #SA = structureness(ak1, ak2)
    #SB = structureness(bk1, bk2)
    #SC = structureness(ck1, ck2)

    ## ugh, apply masks here. too large to be conservative?
    ## otherwise structureness only shows up for small sizes
    new_mask = dilate_boundary(None, radius=int(3*sigma), mask=img.mask)

    crop = cropped_args(img)

    A = A[crop]
    B = B[crop]
    C = C[crop]
```

```

    ak1 = ma.masked_array(ak1,new_mask)[crop]
    ak2 = ma.masked_array(ak2,new_mask)[crop]
    bk1 = ma.masked_array(bk1,new_mask)[crop]
    bk2 = ma.masked_array(bk2,new_mask)[crop]
    ck1 = ma.masked_array(ck1,new_mask)[crop]
    ck2 = ma.masked_array(ck2,new_mask)[crop]

    FA = get_frangi_targets(ak1,ak2, dark_bg=False).filled(0)
    FB = get_frangi_targets(bk1,bk2, dark_bg=False).filled(0)
    FC = get_frangi_targets(ck1,ck2, dark_bg=False).filled(0)


    # the following shows a random vertical slice of A & B (when scaled)
    labels = ('scipy.ndimage,gaussian_filter', 'fft_gaussian', 'fft_dgk')
    formats = ('g:', 'k', 'b-.')
    plot_image_slices((A,B,C), labels=labels, formats=formats,
                      title=r'gaussian convolution $\sigma=$'.format(sigma))
    plt.tight_layout()
    plt.savefig('Gslice_sigma={:d}.png'.format(int(sigma*10)), dpi=300)
    plot_image_slices((FA,FB,FC), labels=labels, formats=formats,
                      title=r'Frangi filter response $\sigma=$'.format(sigma))
    plt.tight_layout()
    plt.savefig('Fslice_sigma={:d}.png'.format(int(sigma*10)), dpi=300)
    #plt.show()

    print('comparing gaussians (mean squared error)')
    print(multiway_comparison((A,B,C), mean_squared_error))
    print('comparing frangi response (mean squared error)')
    print(multiway_comparison((FA,FB,FC), mean_squared_error))
```

listings/hfft_demo.py

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

import numpy as np
from skimage.data import camera
from skimage.io import imread
from skimage.util import img_as_float

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

from scipy.linalg import norm
import timeit

#img = camera() / 255.
img = imread('samples/barium1.png', as_grey=True) / 255.
mask = imread('samples/barium1.mask.png', as_grey=True)

img = img_as_float(img)

# compare computation speed over sigmas

# N logarithmically spaced scales between 1 and 2^m
N = 32
m = 8
sigmas = np.logspace(0,m, num=N, base=2)

fft_results = list()
std_results = list()
```

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

listings/hfft.py

```
1 #!/usr/bin/env python3
2
3 import numpy as np
4 from scipy import signal
5 import scipy.fftpack as fftpack
6 from scipy.special import iv, ive
7 from scipy.ndimage import gaussian_filter
8 from itertools import combinations_with_replacement
9
10 # for demos
11 import matplotlib.pyplot as plt
12
13 from skimage.data import camera
14 from skimage.util import img_as_float
15 from skimage.measure import compare_mse, compare_nrmse
16 """
17 hfft.py is the implementation of calculating the hessian of a real
18
19 image based in frequency space (rather than direct convolution with a gaussian
20 as is standard in scipy, for example).
21
22 TODO: PROVIDE MAIN USAGE NOTES
23 """
24
25 def fft_gaussian(img,sigma, kernel=None):
26
27     """
```

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

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

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

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

    note that here, you actually take the FFT of a gaussian (rather than
    build it in frequency space). there are ~6 ways to do this.
    """
    #create a 2D gaussian kernel to take the FFT of
    # output of signal.gaussian is normalized to 1 so you need to scale
    # it back to work
    #A = 1 / (2*np.pi*sigma**2) # scale factor for 2D

    if kernel in ('discrete', None):
        kern_x = discrete_gaussian_kernel(img.shape[0], sigma)
        kern_y = discrete_gaussian_kernel(img.shape[1], sigma)
    elif kernel == 'sampled':
        # scaling factor for 1d gaussian (use it twice)
        A = 1 / (np.sqrt((2*np.pi)*sigma**2))
        kern_x = A*signal.gaussian(img.shape[0], sigma)
        kern_y = A*signal.gaussian(img.shape[1], sigma)
    else:
        raise ValueError("Key must be 'discrete' or 'sampled'")

    kernel = np.outer(kern_x, kern_y)

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

def discrete_gaussian_kernel(n_samples, sigma):
    """
    sigma is the scale, n_samples is the number of samples to compute
    will return a window centered a zero
    i.e. arange(-n_samples//2, n_samples//2+1)

    not sure how to center it on zero though, since integers only

    note! to make this work similarly to fft_gaussian, this uses
    sigma = np.sqrt(t). Usually you'll find this in terms of t

    by using scipy.special.ive instead we prevent blowups
    """
    dom = np.arange(-(n_samples//2), (n_samples//2) + 1)
    #there should be a scaling parameter alpha but whatever
    #return np.exp(-t) * iv(dom,t)
    if (n_samples % 2) == 0:
        dom = dom[1:]
    return ive(dom,sigma**2)

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

```
92
93      return signal.fftconvolve(img, kernel, mode='same')
94
95  def fft_fdgk(img,sigma):
96      """
97      convolve with discrete gaussian kernel in freq. space
98      """
99      # this would be a lot better since you wouldn't have to deal
100     # with an arbitrary cutoff of size of the discrete kernel
101     # since the freq. space version is just
102     # exp{\alpha*t (cos\theta - 1)}
103     # see formula 22 of lindeberg discrete paper
104
105     pass
106
107 def fft_hessian(image, sigma=1., kernel=None):
108     """
109     a reworking of skimage.feature.hessian_matrix that uses
110     e FFT to compute gaussian, which results in a considerable speedup
111
112     INPUT:
113         image - a 2D image (which type?)
114         sigma - coefficient for gaussian blur
115         kernel - input to fft_gaussian
116         gradient - if you've already computed this
117
118     OUTPUT:
119         (Lxx, Lxy, Lyy) - a triple containing three arrays
120             each of size image.shape containing the xx, xy, yy derivatives
121             respectively at each pixel. That is, for the pixel value given
122             by image[j][k] has a calculated 2x2 hessian of
123             [ [Lxx[j][k], Lxy[j][k]],
124               [Lxy[j][k], Lyy[j][k]] ]
125     """
126
127     gaussian_filtered = fft_gaussian(image, sigma=sigma, kernel=kernel)
128
129     gradients = np.gradient(gaussian_filtered)
130
131     axes = range(image.ndim)
132
133     H_elems = [np.gradient(gradients[ax0], axis=ax1)
134                for ax0, ax1 in combinations_with_replacement(axes, 2)]
135
136     return H_elems
137
138
139 def fft_gradient(image, sigma=1.):
140     """ returns gradient norm """
141
142     gaussian_filtered = fft_gaussian(image, sigma=sigma)
143
144     Lx, Ly = np.gradient(gaussian_filtered)
145
146     return np.sqrt(Lx**2 + Ly**2)
147
148
149 def demo(img=None):
150     """
151     old main function for testing.
152
153     This simply tests fft_gaussian on a test image,
154     """
155
```

```
    if img is None:
        img = img_as_float(camera())
    else:
        img = img_as_float(img)

    sample_sigmas = (.5, 2, 8, 30)
    #sample_sigmas = (.2, 2)

    # build the graphs here side by side
    # show regular blur, sampled blur, discrete blur, 1d plot of signals
    # so a 4 by 4 grid

    fig, axes = plt.subplots(nrows=len(sample_sigmas), ncols=4,
                             figsize=(10, 10))

    for cax, sigma in enumerate(sample_sigmas):

        # convolve the image with a gaussian kernel, one of three ways
        fft_dgk = fft_gaussian(img, sigma, kernel='discrete')
        fft_sampled = fft_gaussian(img, sigma, kernel='sampled')
        xy_sampled = gaussian_filter(img, sigma, mode='constant', cval=0)

        # make the fancy sample
        N = 80
        dom = np.arange(-(N//2), N//2 + 1)
        dgk = discrete_gaussian_kernel(N,  sigma)
        A = np.sqrt(2*np.pi*sigma**2)
        A = 1 / A
        sgk = A * signal.gaussian(N+1, sigma)

        axes[cax, 0].imshow(xy_sampled, cmap='gray', vmin=0, vmax=1)
        axes[cax, 0].set_ylabel(r'$\sigma = $'.format(sigma))
        #axes[cax, 0].set_title(f'ndi.gaussian_filter, $\sigma$={sigma}')
        #axes[cax, 0].axis('off')
        axes[cax, 0].set_xticks([])
        axes[cax, 0].set_yticks([])

        axes[cax, 1].imshow(fft_sampled, cmap='gray', vmin=0, vmax=1)
        #axes[cax, 1].imshow(fft_sampled, cmap='gray')
        #axes[cax, 1].set_title(f'fft sampled kernel, $\sigma$={sigma}')
        axes[cax, 1].axis('off')

        axes[cax, 2].imshow(fft_dgk, cmap='gray', vmin=0, vmax=1)
        #axes[cax, 2].set_title(f'fft discrete kernel, $\sigma$={sigma}')
        axes[cax, 2].axis('off')


        axes[cax, 3].plot(dom, sgk, 'k', dom, dgk, 'g:')
        #axes[cax, 3].set_title(f'discrete vs. sampled kernel $\sigma$={sigma}')
        #axes[cax, 3].axes.set_aspect('equal')

    # set titles for the first column
    axes[0,0].set_title('(a)')
    axes[0,1].set_title('(b)')
    axes[0,2].set_title('(c)')
    axes[0,3].set_title('(d)')

    plt.tight_layout()
    plt.show()

def compare_mae(arr1, arr2):

    assert arr1.shape == arr2.shape
    return np.abs(arr1 - arr2).sum() / arr1.size
```

```
220
221
222 def semigroup_demo(img=None):
223     """
224     the step ones don't look anywhere near as blurred as the initial image
225     in any case! don't use this till it's good!
226     """
227     if img is None:
228         img = img_as_float(camera())
229     else:
230         img = img_as_float(img)
231
232     sigma = 45.
233     n_steps = 2
234     sigmas = (10,35)
235
236     fft_discrete = fft_gaussian(img, sigma, kernel='discrete')
237     fft_sampled = fft_gaussian(img, sigma, kernel='sampled')
238     xy_sampled = gaussian_filter(img, sigma, mode='constant', cval=0)
239
240     step_discrete = img.copy()
241     step_fft_sampled = img.copy()
242     step_xy_sampled = img.copy()
243
244     #sigma_n = sigma / n_steps
245     #sigma_n = np.power(sigma, 1/n_steps)
246     counter = 0
247     for sigma_n in sigmas:
248         step_discrete = fft_gaussian(step_discrete, sigma_n, kernel='discrete')
249         step_fft_sampled = fft_gaussian(step_fft_sampled, sigma_n,
250                                         kernel='sampled')
251         step_xy_sampled = gaussian_filter(step_xy_sampled, sigma_n,
252                                           mode='constant', cval=0)
253         counter += sigma_n
254         #print(counter, end=' ')
255
256     print()
257     er = int(sigma)
258     crop = np.s_[er:-er,er:-er]
259
260     fft_discrete = fft_discrete[crop]
261     fft_sampled = fft_sampled[crop]
262     xy_sampled = xy_sampled[crop]
263     step_discrete = step_discrete[crop]
264     step_fft_sampled = step_fft_sampled[crop]
265     step_xy_sampled = step_xy_sampled[crop]
266
267     fig, axes = plt.subplots(ncols=3, nrows=2)
268     axes[0,0].imshow(xy_sampled, vmin=0, vmax=1, cmap='gray')
269     axes[0,0].set_title('(a)')
270     axes[0,0].set_xticks([]), axes[0,0].set_yticks([])
271
272     axes[0,1].imshow(fft_sampled, vmin=0, vmax=1, cmap='gray')
273     axes[0,1].set_title('(b)')
274     axes[0,1].set_xticks([]), axes[0,1].set_yticks([])
275
276     axes[0,2].imshow(fft_discrete, vmin=0, vmax=1, cmap='gray')
277     axes[0,2].set_title('(c)')
278     axes[0,2].set_xticks([]), axes[0,2].set_yticks([])
279
280     axes[1,0].imshow(step_xy_sampled, vmin=0, vmax=1, cmap='gray')
281     axes[1,0].axis('off')
282     axes[1,1].imshow(step_fft_sampled, vmin=0, vmax=1, cmap='gray')
283     axes[1,1].axis('off')
```

```
    axes[1,2].imshow(step_discrete, vmin=0, vmax=1, cmap='gray')
    axes[1,2].axis('off')

    MSE_sampled = compare_mse(xy_sampled, step_xy_sampled)
    MSE_fft_sampled = compare_mse(fft_sampled, step_fft_sampled)
    MSE_discrete = compare_mse(fft_discrete, step_discrete)

    print(f'MSE sampled:{MSE_sampled}')
    print(f'MSE fft_sampled:{MSE_fft_sampled}')
    print(f'MSE discrete:{MSE_discrete}')
    print()
    #NRMSE_sampled = compare_nrmse(xy_sampled, step_xy_sampled)
    #NRMSE_fft_sampled = compare_nrmse(fft_sampled, step_fft_sampled)
    #NRMSE_discrete = compare_nrmse(fft_discrete, step_discrete)

    #print(f'NRMSE sampled:{NRMSE_sampled}')
    #print(f'MSE fft_sampled:{NRMSE_fft_sampled}')
    #print(f'NRMSE discrete:{NRMSE_discrete}')
    #for ax, title in zip(axes.ravel(), ['(a)', '(b)', '(c)', '(d)']):
    #    ax.axis('off')
    #    ax.set_title(title)


    MAE_sampled = compare_mae(xy_sampled, step_xy_sampled)
    MAE_fft_sampled = compare_mae(fft_sampled, step_fft_sampled)
    MAE_discrete = compare_mae(fft_discrete, step_discrete)

    print(f'MAE sampled:{MAE_sampled}')
    print(f'MAE fft_sampled:{MAE_fft_sampled}')
    print(f'MAE discrete:{MAE_discrete}')

    plt.show()

if __name__ == "__main__":
    from skimage.io import imread
    from placenta import list_by_quality, get_named_placenta
    #A = list_by_quality(0)[0]
    #A = get_named_placenta(A)
    #A = imread('samples/5.3.02.tiff')
    A = None
    demo(A)
    semigroup_demo(A)
```

listings/make_sample_qualities.py

```
# coding: utf-8
import numpy as np
import json

# what runlog to use
jname = 'output/181117-deglared/runlog_181118_0610.json'

with open(jname) as f:
    runlog = json.load(f)

M = {k:np.round(v,decimals=3) for k,v in runlog['MCCS'].items()}
avgs = [m.mean() for m in M.values()]

# get divisions for the four sections
# good and bad will be half as big as the other two
p1 = np.percentile(avgs, 12.5)
p2 = np.percentile(avgs, 50.0)
p3 = np.percentile(avgs, 87.5)
```

```

print(f'thresholds calculated at {p1:.3%}, {p2:.3}, {p3:.3}')
qualities = dict()
poor = {k:m.mean() for k,m in M.items() if m.mean() <= p1}
fair = {k:m.mean() for k,m in M.items() if m.mean() > p1 and m.mean() <= p2}
okay = {k:m.mean() for k,m in M.items() if m.mean() > p2 and m.mean() <= p3}
good = {k:m.mean() for k,m in M.items() if m.mean() > p3}

qualities['good'] = good
qualities['okay'] = okay
qualities['fair'] = fair
qualities['poor'] = poor

print("""
      run:
    with open('sample-qualities.json', 'w') as f:
        json.dump(qualities, f, indent=True)

      ...to save this. won't do it for you!
      """
      )
```

listings/margin_add_demo.py

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

import numpy as np
import matplotlib.pyplot as plt
from skimage.util import img_as_float, img_as_int
from skimage.io import imread
from placenta import (get_named_placenta, list_by_quality, cropped_args,
                      mimg_as_float, open_typefile, open_tracefile)

from frangi import frangi_from_image
import numpy.ma as ma
from hfft import fft_gradient, fft_hessian, fft_gaussian
from merging import nz_percentile, apply_threshold
from plate_morphology import dilate_boundary
import os.path, os

from scipy.ndimage import distance_transform_edt as edt
from skimage.filters.rank import enhance_contrast_percentile as ecp
from skimage.morphology import disk, binary_dilation, thin
from scoring import confusion, mcc

from skimage.filters import threshold_isodata
from skimage.exposure import rescale_intensity
from postprocessing import dilate_to_rim
from preprocessing import inpaint_hybrid
import json

placentas = list_by_quality(0)
OUTPUT_DIR = 'output/190130-margin_add_demo'

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

beta =0.15
gamma = .5
THRESHOLD = .4
scales = np.logspace(-1.5, 3.5, base=2, num=20)

```

```
mccs = list()
precs = list()
for filename in placentas:
    basename = filename.rstrip('.png')
    basename = basename.strip('T-')

    print(filename, '*'*30)
    img = get_named_placenta(filename)
    img = inpaint_hybrid(img)
    crop = cropped_args(img)
    trace = open_tracefile(filename)

    # threshold according to ISODATA threshold for a strawman
    straw = img.filled(0) < threshold_isodata(img.filled(0))
    straw[img.mask] = False

    F = np.stack([frangi_from_image(img, sigma, beta, gamma,
                                    dark_bg=False, dilation_radius=20,
                                    rescale_frangi=True, signed_frangi=True).filled(0)
                                    for sigma in scales])

    f = F[:-4].max(axis=0)
    nf = ((-F*(F<0))[:12]).max(axis=0)

    nf = dilate_boundary(nf, mask=img.mask, radius=20).filled(0)
    spine = dilate_boundary(f, mask=img.mask, radius=20).filled(0)
    spine = rescale_intensity(spine, in_range=(0,1), out_range='uint8')
    spine = spine.astype('uint8')
    bspine = ecp(spine, disk(3)) > (THRESHOLD*255)

    approx, radii = dilate_to_rim(bspine, nf > .05, thin_spine=True,
                                  return_radii=True)

    approx2, radii2 = dilate_to_rim(spine > (THRESHOLD*255), nf > .05,
                                    thin_spine=True, return_radii=True)

    approx_FA = (spine > (THRESHOLD*255))
    approx_PFA = bspine

    ALPHAS = np.array([nz_percentile(F[k], 95.0)
                       for k in range(len(scales))])
    approx_PF = apply_threshold(np.transpose(F,(1,2,0)), ALPHAS,
                                return_labels=False)

    m_st, counts_st = mcc(straw, trace, bg_mask=img.mask, return_counts=True)
    m, counts = mcc(approx, trace, bg_mask=img.mask, return_counts=True)
    m2, counts2 = mcc(approx2, trace, bg_mask=img.mask, return_counts=True)
    m_PFA, counts_PFA = mcc(approx_PFA, trace, bg_mask=img.mask,
                            return_counts=True)
    m_FA, counts_FA = mcc(approx_FA, trace, bg_mask=img.mask,
                            return_counts=True)

    m_PF, counts_PF = mcc(approx_PF, trace, bg_mask=img.mask,
                          return_counts=True)

    precision_score = lambda t: int(t[0]) / int(t[0] + t[2])

    p = precision_score(counts)
    p_st = precision_score(counts_st)
    p2 = precision_score(counts2)
    p_PFA = precision_score(counts_PFA)
    p_PF = precision_score(counts_PF)
    p_FA = precision_score(counts_FA)

```

```
    fig, ax = plt.subplots(nrows=2, ncols=3, figsize=(25,15))

    ax[0,0].imshow(img[crop], cmap=plt.cm.gray)
    ax[0,0].set_title(basename)

    ax[1,0].imshow(confusion(straw,trace)[crop])
    ax[1,0].set_title(rf'   ISODATA threshold (Frangi-less)', loc='left')
    ax[1,0].set_title(f'MCC: {m_st:.2f}\n'
                      f'precision: {p_st:.2%}', loc='right')

    ax[0,1].imshow(confusion(approx_FA, trace)[crop])
    ax[0,1].set_title(rf'   fixed $\alpha = THRESHOLD$', loc='left')
    ax[0,1].set_title(f'MCC: {m_FA:.2f}\n'
                      f'precision: {p_FA:.2%}', loc='right')

    #ax[1,1].imshow(confusion(bspine,trace)[crop])
    #ax[1,1].set_title(rf'   local percentile $\alpha = THRESHOLD$', loc='left')
    #ax[1,1].set_title(f'MCC: {m_PFA:.2f}\n'
    #                   f'precision: {p_PFA:.2%}', loc='right')

    ax[1,1].imshow(confusion(approx_PF,trace)[crop])
    ax[1,1].set_title(rf'   nz-percentile threshold (p=95)', loc='left')
    ax[1,1].set_title(f'MCC: {m_PF:.2f}\n'
                      f'precision: {p_PF:.2%}', loc='right')

    ax[0,2].imshow(confusion(approx2,trace)[crop])
    ax[0,2].set_title(rf'   dilate_to_margin', loc='left')
    ax[0,2].set_title(f'MCC: {m2:.2f}\n'
                      f'precision: {p2:.2%}', loc='right')
    im = ax[1,2].imshow(2*radii2[crop], cmap='tab20c')

    #fig.subplots_adjust(right=0.9, wspace=0.05, hspace=0.1)
    #cbar_ax = fig.add_axes([0.9, 0.15, 0.05, 0.7])
    #fig.colorbar(im, cax=cbar_ax, shrink=0.5)

    [a.axis('off') for a in ax.ravel()]
    fig.subplots_adjust(wspace=0.05, hspace=0.1)
    plt.savefig(os.path.join(OUTPUT_DIR, ''.join(('fig-', basename, '.png'))))
    plt.show()

    mccs.append((filename, m_FA, m_PF, m_PFA, m, m2, m_st))
    precs.append((filename, p_FA, p_PF, p_PFA, p, p2, p_st))

runlog = { 'mccs': mccs, 'precs': precs}
with open(os.path.join(OUTPUT_DIR,'runlog.json'), 'w') as f:
    json.dump(runlog, f, indent=True)
```

listings/margin_add.py

```
# coding: utf-8
plt.imshow(basic.py)
plt.imshow(img.filled(0))
plt.show()
from skimage.filters import scharr
scharr(img)
plt.imshow(_)
plt.show()
scharr(img, mask=img.mask)
plt.imshow(_)
plt.show()
scharr(img, mask=‘img.mask)
scharr(img, mask=~img.mask)
plt.imshow(_)
```

```
plt.show()
from skimage.filters.rank import enhance_contrast_percentile as ecp
from skimage.morphology import selem
scales = np.logspace(-1.5, 3.5, base=2, num=20)
beta = 0.35
gamma = 0.5
F = np.stack([frangi_from_image(img, sigma, beta, gamma, dark_bg=False, dilation_radius=20
plt.imshow(F.max(axis=0))
plt.show()
plt.imshow(F[:-2].max(axis=0))
plt.show()
plt.imshow(F[:-2].max(axis=-1))
plt.imshow(F[:-2].max(axis=0))
plt.show()
plt.imshow(F[:-2].max(axis=0))
ecp(F[:-2].max(axis=0), selem=disk(2))
from skimage.morphology import disk
ecp(F[:-2].max(axis=0), selem=disk(2))
plt.imshow(_)
plt.show()
ecp(F[:-2].max(axis=0), selem=disk(3))
plt.show()
plt.imshow(_)
plt.show()
ecp(F[:-2].max(axis=0), selem=disk(3))
ef = _
plt.imshow(ef > .4)
plt.show()
ef.max()
ef.min()
.4*255
plt.imshow(ef > 102)
plt.show()
plt.imshow(ef > 102)
from placenta import open_typefile
open_typefile(filename, 'trace')
from placenta import open_tracefile
open_tracefile(filename)
trace = _
from scoring import confusion
confusion(ef > 102, trace)
plt.show()
confusion(ef > 102, trace)
plt.imshow(_)
plt.show()
confusion(ef > 102, trace)
plt.show()
plt.imshow(_)
plt.show()
mcc
confusion(F.max(axis=0) > .4, trace)
plt.show()
plt.imshow(_)
plt.show()
fig, ax = plt.subplots(ncols=2, nrows=1)
ax[0].imshow(confusion(F.max(axis=0) > .4, trace)[crop])
ax[1].imshow(confusion(ef > 102, trace)[crop])
plt.show()
fig, ax = plt.subplots(ncols=2, nrows=1)
ax[0].imshow(confusion(F.max(axis=0) > .4, trace)[crop])
ax[1].imshow(confusion(ef > 102, trace)[crop])
[a.axis('off') for a in ax]
plt.show()
fig, ax = plt.subplots(ncols=2, nrows=1)
```

```
ax[0].imshow(confusion(F[:-2].max(axis=0) > .4, trace)[crop])
ax[1].imshow(confusion(ef > 102, trace)[crop])
[a.axis('off') for a in ax]
plt.show()
F = np.stack([frangi_from_image(img, sigma, beta, gamma, dark_bg=False, dilation_radius=20
plt.imshow(np.abs(F).max(axis=0)[crop])
plt.show()
absF = np.abs(F).max(axis=0)
eps(absF, selem=disk(3))
epc(absF, selem=disk(3))
ecp(absF, selem=disk(3))
plt.imshow(_)
plt.show()
plt.imshow(ecp > .3)
ecp(absF, selem=disk(3))
eaf = _ / 255
eaf
plt.imshow(eaf / 255)
plt.show()
plt.imshow(eaf > .4)
plt.show()
plt.imshow(eaf > .5)
plt.show()
plt.imshow(eaf > .5)
plt.show()
F[F < 0}.max(axis=0)
F[F < 0).max(axis=0)
F[F < 0].max(axis=0)
plt.show()
plt.imshow(_)
-F[F < 0].max(axis=0)
(-F[F < 0])
plt.imshow(_)
(-F*(F<0)).max(axis=0)
plt.imshow(_)
plt.show()
(-F[:12]*(F<0)).max(axis=0)
(-F[]*(F<0))[:12].max(axis=0)
(-F*(F<0))[:12].max(axis=0)
plt.imshow(_)
plt.show()
(-F*(F<0))[:10].max(axis=0)
plt.show()
plt.imshow(_)
plt.show()
nf = (-F*(F<0))[:10].max(axis=0)
plt.imshow(nf)
plt.show()
plt.imshow(nf+Fmax)
plt.imshow(nf+F[:-2].max(axis=0))
plt.show()
plt.imshow(nf+F[:-4].max(axis=0))
plt.show()
ecp(nf+F[:-4].max(axis=0), selem=disk(3))
ecp(max(nf,F[:-4].max(axis=0)), selem=disk(3))
help(np.max)
get_ipython().run_line_magic('pinfo', 'np.choose')
get_ipython().run_line_magic('pinfo', 'np.take')
get_ipython().run_line_magic('pinfo', 'np.maximum')
ecp(np.maximum(nf,F[:-4].max(axis=0)), selem=disk(3))
plt.imshow(_)
plt.show()
ecp(np.maximum(nf,F[:-4].max(axis=0))[crop], selem=disk(3))
plt.show()
```

```
plt.imshow(_)
plt.show()
np.maximum(nf,F[:-4].max(axis=0))[crop]
plt.imshow(_)
plt.show()
np.maximum(nf,F[:-4].max(axis=0))[crop]
plt.imshow(_)
plt.show()
spine = F[:-4].max(axis=0)
margins = nf
np.maximum(spine,margins)
np.maximum(spine,margins)*~img.mask
c = _
plt.imshow(c)
plt.show()
np.maximum(spine,margins)
plt.ismhow(_)
plt.show()
plt.imshow()
plt.imshow(c)
plt.show()
c = ma.masked_array(np.maximum(spine,margins), mask=img.mask)
plt.imshow(c)
plt.show()
dilate_boundary(c, radius=20)
plt.imshow(_)
plt.show()
c = ma.masked_array(np.maximum(spine,margins), mask=img.mask)
dilate_boundary(c, 20).filled(0)
c = _
plt.imshow(c)
plt.show()
epc(c, disk(4))
ecp(c, disk(4))
plt.imshow(_)
plt.show()
spine
spine*img.mask
spine*~img.mask
plt.imshow(_)
plt.show()
dilate_boundary(spine, mask=img.mask, radius=20)
plt.imshow(_)
plt.show()
spine = dilate_boundary(spine, mask=img.mask, radius=20).filled(0)
plt.imshow(spine[crop])
plt.show()
from skimage.segmentation import thin
from skimage.morphology import thin
thin(spine)
plt.imshow(_)
plt.show()
ecp(spine, disk=3)
ecp(spine, disk(3))
plt.imshow(_)
plt.show()
ecp(spine, disk(3) > 102)
plt.imshow(_)
plt.show()
ecp(spine, disk(3)) > 102
plt.imshow(_)
plt.show()
bspine = ecp(spine, disk(3)) > 102
plt.imshow(thin(bspine))
```

```
plt.show()
plt.imshow(np.maximum(margins, bspine))
plt.show()
plt.imshow(np.maximum(margins, bspine))
plt.show()
plt.imshow(np.maximum(margins*255, bspine))
plt.show(_)
plt.imshow(np.maximum(margins*255, bspine))
plt.show()
plt.imshow(np.maximum(margins, bspine))
plt.show()
plt.imshow(np.maximum(margins, thin(bspine)))
plt.show()
margins
plt.imshow(margins)
plt.show()
margins > .2
plt.imshow(_)
plt.show()
margins > .15
plt.ishow()
plt.imshow()
plt.imshow(margins > .1)
plt.show()
plt.imshow(margins > .05)
plt.show()
D = np.zeros(margins, np.bool)
D = np.zeros(margins.shape, np.bool)
from scipy.ndimage import distance_transform_edt
from scipy.ndimage import distance_transform_edt as edt
D[thin(bspine)] = 1
D[:] = 1
D[margins>.05] = 0
plt.imshow(_)
pl.timshow(D)
plt.imshow(D)
plt.show()
edt(D)
spine_dists = D.copy()
spine_dists[~thin(bspine)] = 0
plt.imshow(spine_dists)
plt.show()
spine_dists = edt(D)
spine_dists[~thin(bspine)] = 0
plt.imshow(spine_dists)
plt.show()
spine_dists<=10
plt.imshow(_)
plt.show()
(spine_dists<=10) & (spine_dists > 0)
plt.show()
plt.imshow()
plt.imshow(_255)
plt.show()
(spine_dists<=6) & (spine_dists > 0)
plt.imshow(_)
plt.show()
from skimage.morphology import binary_dilation
help(np.round)
help(np.around)
help(np.fix)
spine_radii = np.round(spine_dists).astype('int')
plt.imshow(spine_radii)
plt.shwo()
```

```
plt.show()
\plt.imshow(spine_radii)
np.stack([binary_dilation(spine_radii==r, selem=disk(r)) for r in range(1,10)])
dilstack = _
plt.imshow(dilstack.max(axis=0))
plt.show()
confusion(dilstack.max(axis=0), trace)
plt.show()
plt.imshow(_)
plt.show()
np.stack([binary_dilation(spine_radii==r, selem=disk(r)) for r in range(1,12)])
plt.imshow(_)
dilstack = np.stack([binary_dilation(spine_radii==r, selem=disk(r)) for r in range(1,12)])
plt.imshow(dilstack.any(axis=0))
plt.show()
confusion(dilstack.any(axis=0), approx)
confusion(dilstack.any(axis=0), trace)
plt.show()
plt.imshow(_)
plt.show()
dilstack = np.stack([binary_dilation(spine_radii==r, selem=disk(r)) for r in range(1,15)])
plt.imshow(confusion(dilstack.any(axis=0), trace))
plt.show()
from scoring import mcc
approx = dilstack.any(axis=0)
mcc(approx, trace, bg_mask=img.mask, return_counts=True)
counts = _[1]
TP, TN, FP, FN = counts
precision = TP / (TP+FN)
precision
precision = TP / (TP+FP)
precision
```

listings/make_output_montage.py

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

listings/merging.py

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

import numpy as np
import numpy.ma as ma
from scipy.ndimage import label
from skimage.morphology import remove_small_objects
import matplotlib.pyplot as plt

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

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

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

```
22
23      Returns
24      -------
25      out: float
26
27      """
28
29      if ma.is_masked(A):
30          A = A.filled(0)
31
32      return np.percentile(A[A > 0], q, axis=axis, interpolation=interpolation)
33
34
35  def apply_threshold(targets, alphas, return_labels=True):
36      """Threshold targets at each scale, then return max target over all scales.
37
38      A unique alpha can be given for each scale (see below). Return a 2D boolean
39      array, and optionally another array representing what at what scale the max
40      filter response occurred.
41
42      Parameters
43      ----------
44      targets : ndarray
45          a 3D array, where targets[:,:,k] is the result of the Frangi filter
46          at the kth scale.
47      alphas : float or array_like
48          a list / 1d array of length targets.shape[-1]. each alphas[k] is a
49          float which thresholds the Frangi response at the kth scale. Due to
50          broadcasting, this can also be a single float, which will be applied
51          to each scale.
52      return_labels : bool, optional
53          If True, return another ndarray representing the scale (see Notes
54          below). Default is True.
55
56      Returns
57      -------
58      out : ndarray, dtype=bool
59          if return labels is true, this will return both the final
60          threshold and the labels as two separate matrices. This is
61          a convenience, since you could easily find labels with
62      labels : ndarray, optional, dtype=uint8
63          The scale at which the largest filter response was found after
64          thresholding. Element is 0 if no scale passed the threshold,
65          otherwise an int between 1 and targets.shape[-1] See Notes below.
66
67      Notes / Examples
68      ----------------
69      Despite the name, this does *NOT* return the thresholded targets itself,
70      but instead the maximum value after thresholding. If you wanted the
71      thresholded filter responses alone, you should simply run
72
73      >>>(targets > alphas)*targets
74
75      The optional output `labels` is a 2D matrix indicating where the max filter
76      response occured. For example, if the label is K, the max filter response
77      will occur at targets[:,:,K-1].  In other words,
78
79      >>>passed, labels = apply_threshold(targets,alphas)
80      >>>targets.max(axis=-1) == targets[:,:,labels -1 ]
81      True
82
83      It should be noted that returning labels is really just for convenience
84      only; you could construct it as shown in the following example:
85
```

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

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

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

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

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

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

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

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

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

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

    return passed, wheres


def sieve_scales(multiscale, high_percentile, low_percentile, min_size=None,
                 axis=0):
    """
    multiscale is a 3 dimensional where 2 dimensions are image and `axis`
    parameter is which one is the scale space (i.e. resolution). hopefully
    axis is 0 or 1 (this won't handle stupider cases)

    this gathers points contiguous points at a low threshold and adds them
    to the output it contains at least only if that blob contains at least one
    high percentile point.

    min_size is a size requirement can either be an integer or an array of
    integers """

    assert multiscale.ndim == 3

    if axis in (-1, 2):
        # this won't change the input, just creates a view
        V = np.transpose(multiscale, axes=(2, 0, 1))

    elif axis == 0:
        V = multiscale  # just to use the same variable name
    else:
        raise ValueError('Please make resolution the first or last dimension.')

    if np.isscalar(min_size):
        min_size = [min_size for x in range(multiscale.shape[0])]
```

```
150
151     # label matrix the size of one of the images
152     sieved = np.zeros(V.shape[1:], dtype=np.int32)
153
154     print('sieving ', end='')
155     for n, v in enumerate(V):
156         print('σ', end='', flush=True)
157
158         if min_size is not None:
159             z = remove_small_objects(v, min_size=min_size[n])
160         else:
161             z = v  # relabel to use same variable
162
163         high_thresh = nz_percentile(v, high_percentile)
164         low_thresh = nz_percentile(v, low_percentile)
165
166         labeled, n_labels = label(z > low_thresh)
167         high_passed = (z > high_thresh)
168
169         for lab in range(n_labels):
170             if lab == 0:
171                 continue
172             if np.any(high_passed[labeled == lab]):
173                 sieved[labeled == lab] = n
174
175     print()
176     return sieved
177
178 def view_slices(multiscale, axis=0, scales=None, crop=None, cmap='nipy_spectral',
179                 vmin=0, vmax=1.0, outnames=None, show_colorbar=True):
180     """ scales is just to use for a figure title
181         crop before you get in here.
182
183         if outname is an iterable returning filenames, then we'll assume
184         non-interative mode
185         """
186     assert multiscale.ndim == 3
187
188     if axis in (-1, 2):
189         # this won't change the input, just creates a view
190         V = np.transpose(multiscale, axes=(2, 0, 1))
191
192     elif axis == 0:
193         V = multiscale  # just to use the same variable name
194     else:
195         raise ValueError('Please make resolution the first or last dimension.')
196
197     if scales is None:
198         scales = [None for x in range(multiscale.shape[0])]
199     if outnames is None:
200         outnames = [None for x in range(multiscale.shape[0])]
201
202     plt.close('all')
203     for v, sigma, outname in zip(V, scales, outnames):
204
205         if crop is None:
206             viewable = v
207         else:
208             viewable = v[crop]
209         if outname is None:
210
211             plt.imshow(viewable, cmap=cmap, vmin=vmin, vmax=vmax)
212
213             mng = plt.get_current_fig_manager()
```

```
            #mng.window.showMaximized()
            mng.window.showFullScreen()
            plt.tight_layout()
            if sigma is not None:
                plt.title(r'$\sigma = %.2f$'.format(sigma))
            plt.axis('off')
            if show_colorbar:
                plt.colorbar()
            plt.tight_layout()
            plt.show()
            plt.close()

        else:
            # save them non interactively with imsave
            plt.imsave(outname, viewable, cmap=cmap, vmin=vmin, vmax=vmax)
```

listings/network_completion.py

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

from skimage.filters.rank import sum as local_count
from skimage.draw import line
import numpy as np
from itertools import combinations

def find_endpoints(arr):
    """Find endpoints of a boolean array (like a thinned approximation)

    Will return any pixels that have between 1 and 2 adjacent neighbors using
    2-connectivity. 2 neighbors means it's directly connected to one other
    pixel, and 1 pixel means it's not connected to anything else.
    """
    # 2-connectivity
    selem = np.ones((3,3), np.bool)

    a = arr.astype('bool')

    neighbors = local_count(arr.astype('uint8'), selem)

    return arr & ((neighbors==1)|(neighbors==2))

def categorize_endpoints(arr):
    """Find endpoints, as in find_endpoints, but label each with the *kind*
    of endpoint it is describing its orientation. Is it 2 connected by:

        _ _ _ <--- top (0)
        _ . _ <--- middle (1)
        _ _ _ <--- bottom (2)
        ^ ^ ^
        | | |_______ right (0)
        | |_________ middle (1)
        |___________ left (2)

    these are the same as the row/column indices into the 3x3 square. so
    if a pixel's only neighboring pixel is to its immediate right, then
    the label for the pixel is (1,0). If the endpoint is isolated (i.e. it
    has no neighbors) then its label is (1,1) (since 1 is the more forgivable
    connection condition).

    The idea is that
    -   a left-connected pixel should not connect with another left-connected
        pixel
```

```
    -   a right-connected pixel should not connect with another right-connected
        pixel.
    -   a bottom-connected pixel should not be connected with another
        bottom-connected pixel
    -   a top-connected pixel should not be connected with another
        top-connected pixel

    There are other ideas (that a connection shouldn't be made through a point
    where the frangi score is 0; that a connection shouldn't be made that
    crosses an established part of the skeleton; that connections shouldn't
    be made between two endpoints whose distance exceeds some specified amount)
    that cannot be cast in terms of these labels, but this does reduce the
    number of possible connections substantially.

    this returns two lists of tuples, where the first contains the indices of
    each endpoint, and the corresponding place in the second list contains
    its label (also a tuple)
    """

    # still get a list of endpoints because this is faster.
    selem = np.ones((3,3), np.bool)
    a = arr.astype('bool')
    endpoint_list = list()
    label_list = list()
    bd = np.array([[1,1,1],[1,0,1],[1,1,1]],dtype='bool')
    for px, py in zip(*np.where(a)):
        local = a[px-1:px+2,py-1:py+2]

        if local.sum() == 1:
            endpoint_list.append((px,py))
            label_list.append((1,1))
        elif local.sum() == 2:
            endpoint_list.append((px,py))
            label_list.append(tuple(map(int, np.where(local&bd))))
        else:
            pass # this isn't an endpoint

    return endpoint_list, label_list


def mean_colored_connections(arr, scores=None, double_connect=True):
    """Lines that connect nearest endpoints, colored by the mean value of
        that line through the matrix scores

        if scores is None, just return a boolean array
    """
    # list of endpoints as ordered pairs
    a = arr.astype('bool') # cast to bool first to prevent weirdness
    neighbors = local_count(a.astype('uint8'), np.ones((3,3), np.bool))
    endpoints = a & ((neighbors==1)|(neighbors==2))
    endpoint_list = list(zip(*np.where(endpoints)))

    lines = np.zeros(endpoints.shape) # but type float
    for p0 in endpoint_list:
        if (neighbors[p0] == 2) or not double_connect:
            nearest = endpoint_list[np.argmin([
                        (p0[0]-p[0])**2 + (p0[1] - p[1])**2
                        if p0!=p else 10000 for p in endpoint_list]
                        )]
            #print(p0, nearest)
            line_to_add = line(*p0, *nearest)

            if scores is None:
                lines[line_to_add] = 1
```

```
            else:
                lines[line_to_add] = scores[line_to_add].mean()
                #print(scores[line_to_add].mean())
        else:
            # same thing but get the two nearest
            nearest, second, *__ = np.argpartition([
                        (p0[0]-p[0])**2 + (p0[1] - p[1])**2
                        if p0!=p else 10000 for p in endpoint_list], 2)
            nearest = endpoint_list[nearest]
            second = endpoint_list[second]
            print(p0, nearest, second, '<---------------')
            line_to_add = line(*p0, *nearest)
            line2_to_add = line(*p0, *second)
            if scores is None:
                lines[line_to_add] = 1
                lines[line2_to_add] = 1
            else:
                lines[line_to_add] = scores[line_to_add].mean()
                lines[line2_to_add] = scores[line2_to_add].mean()
                #print(scores[line_to_add].mean())
                #print(scores[line2_to_add].mean())

    return lines


def connect_iterative(arr, scores=None, max_iterations=10):
    """Lines that connect nearest endpoints, colored by the mean value of
        that line through the matrix scores

        if scores is None, just return a boolean array
    """
    # list of endpoints as ordered pairs
    a = arr.astype('bool') # cast to bool first to prevent weirdness
    bad_pairs = list()
    lines = a.copy() # but type float
    for i in range(max_iterations):
        neighbors = local_count(a.astype('uint8'), np.ones((3,3), np.bool))
        endpoints = a & ((neighbors==1)|(neighbors==2))
        endpoint_list = list(zip(*np.where(endpoints)))
        print(f"at start of {i}th iteration, {len(bad_pairs)} many bad pairs")
        old_network_size = lines.sum()
        for k, p0 in enumerate(endpoint_list):
            candidates = [p for p in endpoint_list[k:]
                          if ((p,p0) not in bad_pairs and
                              (p0,p) not in bad_pairs)]
            if not candidates:
                continue
            nearest = candidates[np.argmin([
                           (p0[0]-p[0])**2 + (p0[1] - p[1])**2
                           if p0!=p else 10000 for p in candidates]
                           )]
            line_to_add = line(*p0, *nearest)

            if scores is None:
                lines[line_to_add] = 1
            else:
                mean_score = scores[line_to_add].mean()
                if (mean_score < .3) or (scores[line_to_add]==0).any():
                    bad_pairs.append((p0,nearest))
                    continue
                else:
                    lines[line_to_add] = 1
                #print(scores[line_to_add].mean())
        new_network_size =lines.sum()
```

```
        size_diff =  new_network_size - old_network_size
        print(f'after {i}th iteration,', size_diff, 'pixels were added')
        if size_diff == 0:
            break
    else:
        print(f'returned after {max_iterations} iterations')

    return lines


def _euclidean_distance(p0,p1):
    return np.sqrt((p0[0]-p1[0])**2 + (p0[1]-p1[1])**2)


def connect_iterative_by_label(arr, scores, max_iterations=10,
                               max_dist=None):
    """Lines that connect nearest endpoints, colored by the mean value of
        that line through the matrix scores

        if scores is None, just return a boolean array
    """
    # list of endpoints as ordered pairs
    a = arr.astype('bool') # cast to bool first to prevent weirdness
    matched = list()


    endlist, endlabs = categorize_endpoints(arr)
    N_ends = len(endlist)
    # handshake matrix, matchable[j,k]==0 if endlist[j] and endlist[k]
    # cannot be connected, otherwise 1. this is upper triangular
    # on the first diagonal, so make sure k > j
    matchable = np.triu(np.ones((len(endlist), len(endlist))), k=1)

    # distances between these two points with lazy filling. could be overloaded
    # into matchable but let's keep it separate
    dists = np.zeros(matchable.shape, dtype=np.float64)
    print(f'there are {len(endlist)} endpoints')

    print('checking compatibility of endpoints by labels and calculating distances')

    for j, jlab in enumerate(endlabs):

        # still calculate the distances below
        #if jlab == (1,1):
        #    continue # this can connect with anything

        for k, klab in enumerate(endlabs[j+1:],j+1):
            if jlab[0] != 1:
                if jlab[0] == klab[0]:
                    matchable[j,k] = 0
                    continue
            if jlab[1] != 1:
                if jlab[1] == klab[1]:
                    matchable[j,k] = 0
                    continue
            # if they are matchable, let's find the distance between them
            dist = _euclidean_distance(endlist[j], endlist[k])
            if max_dist is not None:
                if dist > max_dist:
                    matchable[j,k] = 0
                else:
                    dists[j,k] = dist
            else:
                dists[j,k] = dist
```

```
237
238
239     lines = a.copy() # but type float
240
241     print(f"removed {N_ends*(N_ends-1)/2 - matchable.sum():n} "
242           "pairs of endpoints from consideration "
243           f'(out of {N_ends*(N_ends-1) / 2:n} possible pairs)')
244
245     size_before = matchable.sum()
246     for j, pj in enumerate(endlist):
247         for k, pk in enumerate(endlist[j+1:], j+1):
248             if not matchable[j,k]:
249                 continue
250             l = line(*pj, *pk)
251             # lines will have the two endpoints so sum is at least 2
252             if (lines[l].sum() > 2) or (scores[l]==0).any():
253                 matchable[j,k] = 0
254                 dists[j,k] = 0
255
256     size_after = matchable.sum()
257
258     print(f"removed {size_before-size_after:n} more pairs",
259           "by avoiding crossings/zero score paths")
260
261     for j, pj in enumerate(endlist):
262         mean_score = lambda p: scores[line(*pj,*p)].mean()
263         point_scores = [(k, pk, mean_score(pk))
264                         for k,pk in enumerate(endlist[j+1:], j+1)
265                         if matchable[j,k]]
266         if not point_scores:
267             continue
268         s = sorted(point_scores, key=lambda x: x[2])
269         p_best = s[-1][1]
270         lines[line(*pj, *p_best)] = 1
271
272     return lines
273     #for i in range(max_iterations):
274     #    old_network_size = lines.sum()
275
276     #    print(f"at start of {i}th iteration:")
277     #    print(f"\t{matchable.any(axis=1).sum()} unmatched endpoints")
278     #    print(f"\t{old_network_size} is the size of the network")
279
280     #    for j, p0 in enumerate(endlist):
281
282     #        if not matchable[k,:].any():
283     #            continue # we can't connect this endpoint or it's already
284     #                     # been connected to
285
286     #        for k, p in enumerate(endlist[j+1:],j+1):
287     #            if not matchable[j,k]:
288     #                continue
289
290     #        if not candidates:
291     #            continue # this endpoint can't connect to anything anymore
292
293     #        for candidate in candidate:
294     #            pass
295     #        #nearest = candidates[np.argmin([
296     #        #                (p0[0]-p[0])**2 + (p0[1] - p[1])**2
297     #        #                if p0!=p else 10000 for p in candidates]
298     #        #                )]
299     #        #line_to_add = line(*p0, *nearest)
300
```

```
    #         #if scores is None:
    #         #    lines[line_to_add] = 1
    #         #else:
    #         #    mean_score = scores[line_to_add].mean()
    #         #    if (mean_score < .3) or (scores[line_to_add]==0).any():
    #         #        bad_pairs.append((p0,nearest))
    #         #        continue
    #         #    else:
    #         #        lines[line_to_add] = 1
    #         #    #print(scores[line_to_add].mean())
    #     new_network_size = lines.sum()
    #     size_diff =  new_network_size - old_network_size
    #     print(f'after {i}th iteration,', size_diff, 'pixels were added')
    #     if size_diff == 0:
    #         break
    #     else:
    #         old_network_size = new_network_size
    #else:
    #    print(f'returned after {max_iterations} iterations')

    #return lines


def colored_connections_any_nonzero(arr, scores):
    """Lines that connect nearest endpoints, colored by the mean value of
        that line through the matrix scores

        WOULD BE BETTER IF YOU USED SCORES FROM THE APPROPRIATE SCALE
        if scores is None, just return a boolean array
    """
    # list of endpoints as ordered pairs
    a = arr.astype('bool') # cast to bool first to prevent weirdness
    neighbors = local_count(a.astype('uint8'), np.ones((3,3), np.bool))
    endpoints = a & ((neighbors==1)|(neighbors==2))
    endpoint_list = list(zip(*np.where(endpoints)))

    lines = np.zeros(endpoints.shape) # but type float
    #good_lines = list()
    for i, p0 in enumerate(endpoint_list):
        for j, p1 in enumerate(endpoint_list[i+1:], i+1):

            #nearest = endpoint_list[np.argmax([
            #            (p0[0]-p[0])**2 + (p0[1] - p[1])**2
            #            if p0!=p else 10000 for p in endpoint_list]
            #            )]

            candidate = line(*p0, *p1)

            if (scores[candidate] > 0).all():
                lines[candidate] = scores[candidate].mean()

    return lines


def colored_connections_max_path(arr, scores):
    """Lines that connect nearest endpoints, colored by the mean value of
        that line through the matrix scores

        WOULD BE BETTER IF YOU USED SCORES FROM THE APPROPRIATE SCALE
        if scores is None, just return a boolean array
    """
    # list of endpoints as ordered pairs
    a = arr.astype('bool') # cast to bool first to prevent weirdness
    neighbors = local_count(a.astype('uint8'), np.ones((3,3), np.bool))
```

```
    endpoints = a & ((neighbors==1)|(neighbors==2))
    endpoint_list = list(zip(*np.where(endpoints)))

    lines = np.zeros(endpoints.shape) # but type float
    #good_lines = list()
    for i, p0 in enumerate(endpoint_list):
        mean_score = lambda p: scores[line(*p0,*p)].mean() if ((p0!=p) and np.all(scores[li
        point_scores = [(p, mean_score(p)) for p in endpoint_list[i+1:]]
        point_scores = [(p, score) for p, score in point_scores if score!=0]
        s = sorted(point_scores, key=lambda x: x[1])
        if i<5:
            print(f'{i}:', s)
        for p, score in s:
            path = line(*p0, *p)
            lines[path] = score

    return lines

if __name__ == "__main__":



    import matplotlib.pyplot as plt
    import matplotlib as mpl
    from skimage.util import img_as_float
    from skimage.io import imread
    from placenta import (get_named_placenta, list_by_quality, cropped_args,
                          mimg_as_float)

    from frangi import frangi_from_image
    import numpy.ma as ma
    from hfft import fft_gradient, fft_hessian, fft_gaussian
    from merging import nz_percentile
    from plate_morphology import dilate_boundary
    import os.path, os
    from skimage.morphology import thin
    from postprocessing import dilate_to_rim
    from scoring import confusion, mcc
    from placenta import open_tracefile, open_typefile
    from preprocessing import inpaint_hybrid
    from placenta import measure_ncs_markings, add_ucip_to_mask

    import json

    mccs = list()
    precs = list()

    scales = np.logspace(-1.5, 3.5, num=20, base=2)
    threshold = .15
    neg_threshold = 0.001
    max_pos_scale = -6
    max_neg_scale = 2
    beta = 0.10
    gamma = 1.0

    cm = mpl.cm.plasma
    #cmscales = mpl.cm.magma
    cm.set_bad('k', 1)  # masked areas are black, not white
    #cmscales.set_bad('w', 1)

    QUALITY = 3
    for filename in list_by_quality(QUALITY):

        cimg = open_typefile(filename, 'raw')
```

```
        ctrace = open_typefile(filename, 'ctrace')
        trace = open_tracefile(filename)
        img = get_named_placenta(filename)
        crop = cropped_args(img)
        ucip = open_typefile(filename, 'ucip')
        img = inpaint_hybrid(img)

        # make the size of figures more consistent
        if img[crop].shape[0] > img[crop].shape[1]:
            # and rotating it would be fix all this automatically
            cimg = np.rot90(cimg)
            ctrace = np.rot90(ctrace)
            trace = np.rot90(trace)
            img = np.rot90(img)
            ucip = np.rot90(ucip)
            crop = cropped_args(img)

        ucip_midpoint, resolution = measure_ncs_markings(ucip)
        ucip_mask = add_ucip_to_mask(ucip_midpoint, radius=60, mask=img.mask)

        name_stub = filename.rstrip('.png').strip('T-')


        F = np.stack([frangi_from_image(img, sigma, beta=beta, gamma=gamma, dark_bg=False,
                                        signed_frangi=True, dilation_radius=20,
                                        rescale_frangi=True)
                                        for sigma in scales])
        # need to fix this in the signed_frangi logic
        F = F*~dilate_boundary(None, mask=img.mask, radius=20)

        Fmax = (F*(F>0))[:max_pos_scale].max(axis=0)
        Fneg = -(F*(F<0))[:max_neg_scale].min(axis=0)

        approx = Fmax > threshold
        rim_approx = (Fneg > neg_threshold)
        skel = thin(approx)
        completed = connect_iterative_by_label(skel, Fmax, max_dist=100)
        completed_dilated = dilate_to_rim(completed, rim_approx, max_radius=10)
        approx_dilated = dilate_to_rim(approx, rim_approx, max_radius=10)

        network = np.maximum(skel*3., (completed & ~skel)*2)
        network = np.maximum(network, rim_approx*1.)


        precision = lambda t: int(t[0]) / int(t[0] + t[2])

        mcc_FA, counts_FA = mcc(approx, trace, bg_mask=img.mask, return_counts=True)
        mcc_FAD, counts_FAD = mcc(approx_dilated, trace, bg_mask=img.mask, return_counts=Tr
        mcc_NCD, counts_NCD = mcc(completed_dilated, trace, bg_mask=img.mask, return_counts

        prec_FA = precision(counts_FA)
        prec_FAD = precision(counts_FAD)
        prec_NCD = precision(counts_NCD)

        mccs.append((name_stub, mcc_FA, mcc_FAD, mcc_NCD))
        precs.append((name_stub, prec_FA, prec_FAD, prec_NCD))

        fig, ax = plt.subplots(nrows=2, ncols=3, figsize=(20,12))
        A = ax.ravel()

        A[0].imshow(cimg[crop])
        A[0].set_title(name_stub)

        A[1].imshow(ma.masked_where(Fmax==0,Fmax)[crop], cmap=cm, vmin=0, vmax=1)
```

```
        A[1].set_title(rf'$V_{max}, \beta = beta:.2f, \gamma = gamma:.3f$')

        A[2].imshow(network[crop], cmap=plt.cm.magma)
        A[2].set_title('skeleton, completed network, and rim_approx')

        A[3].imshow(confusion(approx, trace, bg_mask=img.mask)[crop])
        A[3].set_title(fr'    fixed $\alpha = threshold:.2f$', loc='left')
        A[3].set_title(f'MCC: {mcc_FA:.2f}\n'
                       f'precision: {prec_FA:.2%}', loc='right')

        A[4].imshow(confusion(approx_dilated, trace, bg_mask=img.mask)[crop])
        A[4].set_title(fr'    dilate_to_rim $\alpha = threshold:.2f$', loc='left')
        A[4].set_title(f'MCC: {mcc_FAD:.2f}\n'
                       f'precision: {prec_FAD:.2%}', loc='right')

        A[5].imshow(confusion(completed_dilated, trace, bg_mask=img.mask)[crop])
        A[5].set_title(fr'    network_completed, dilated', loc='left')
        A[5].set_title(f'MCC: {mcc_NCD:.2f}\n'
                       f'precision: {prec_NCD:.2%}', loc='right')

        [a.axis('off') for a in A]
        fig.tight_layout()
        plt.show()



runlog = { 'mccs':mccs,
           'precs':precs,
           'scales':list(scales),
           'beta':beta,
           'gamma':gamma,
           'max_pos_scale':max_pos_scale,
           'max_neg_scale':max_neg_scale,
           'threshold':threshold,
           'neg_threshold':neg_threshold
           }


with open(f'output/network_completion_{QUALITY}.json', 'w') as f:
    json.dump(runlog, f, indent=True)
```

listings/pcsvn.py

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

"""
the contents of this file should be split up between frangi (multiscale)
and scoring/output
"""
from placenta import get_named_placenta
from diffgeo import principal_directions
from frangi import frangi_from_image
from skimage.util import img_as_float
import numpy as np
from preprocessing import inpaint_hybrid

from merging import nz_percentile

from plate_morphology import dilate_boundary

import matplotlib.pyplot as plt
import matplotlib as mpl
```

```
import numpy.ma as ma

import os.path
import json
import datetime


def make_multiscale(img, scales, beta=0.5, gamma=0.5, c=None, dark_bg=True,
                    find_principal_directions=False, dilate_per_scale=True,
                    signed_frangi=False, kernel=None, verbose=True,
                    rescale_frangi=False, gradient_filter=False):
    """Returns an ordered list of dictionaries for each scale of Frangi info.

    beta, gamma, and c can all be vectors as long as scales or constants
    if c is None it will be set.

    Each element in the output contains the following info:
        {'sigma': sigma,
         'beta': beta,
         'gamma': gamma,
         'H': hesh,
         'F': targets,
         'k1': k1,
         'k2': k2,
         't1': t1, # if find_principal_directions
         't2': t2 # if find_principal_directions
         }

    is it necessary to lug all this shit around?
    """

    # store results of each scale (create as empty list)
    multiscale = list()

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

    vectorize = lambda x: np.repeat(x, len(scales)) if (x is None or np.isscalar(x)) else x

    # vectorize any scalar inputs here
    beta = vectorize(beta)
    gamma = vectorize(gamma)
    c = vectorize(c)
    print('finding multiscale targets ', end='')
    for i, (sigma, b, g, cx) in enumerate(zip(scales, beta, gamma, c)):

        print('σ', end='', flush=True)

        if dilate_per_scale:
            if sigma > 20:
                radius = int(2*sigma)
            elif sigma < 3:
                radius = 12
            else:
                radius = int(4*sigma)
        else:
            radius = None

        targets, this_scale = frangi_from_image(img, sigma, beta=b, gamma=g,
                                                c=cx, dark_bg=dark_bg,
                                                dilation_radius=radius,
                                                kernel=kernel,
                                                signed_frangi=signed_frangi,
                                                return_debug_info=True,
                                                rescale_frangi=rescale_frangi,
```

```
                                                    gradient_filter=gradient_filter)

        if find_principal_directions:
            # principal directions should only be computed for critical regions
            # this mask is where PD's will *NOT* be calculated
            # is targets a masked array?
            cutoff = nz_percentile(targets, 80)
            pd_mask = np.bitwise_or(targets < cutoff, img.mask).filled(1)
            percent_calculated = (pd_mask.size - pd_mask.sum()) / pd_mask.size

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

            # add them to this scale's output
            this_scale['t1'] = t1
            this_scale['t2'] = t2

        else:
            if verbose:
                print('skipping principal direction calculation')

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

    print()
    return multiscale


def extract_pcsvn(img, filename, scales, beta=0.5, gamma=0.5, c=None,
                  dark_bg=True, dilate_per_scale=True, verbose=True,
                  generate_json=True, output_dir=None, kernel=None,
                  signed_frangi=False, rescale_frangi=False,
                  gradient_filter=False):
    """Run PCSVN extraction on the sample given in the file.

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

    As a matter of fact, this function currently just is a wrapper for
    make_multiscale that logs some output
    The original main use of this function has kind of bled into
    extract_NCS_pcsvn.py. that needs fixing. You should load the image
    outside of this function, do post processing there, pass it inside here
    with a dictionary of things to add to the json file

    """

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

    # output is a dictionary of relevant info at each scale
    multiscale = make_multiscale(img, scales, beta=beta, gamma=gamma, c=None,
                                 find_principal_directions=False,
                                 dilate_per_scale=dilate_per_scale,
                                 kernel=kernel, signed_frangi=signed_frangi,
                                 dark_bg=dark_bg, verbose=verbose,
                                 rescale_frangi=rescale_frangi,
                                 gradient_filter=gradient_filter)

    # extract these for logging
```

```
    c = [scale['c'] for scale in multiscale]
    border_radii = [scale['border_radius'] for scale in multiscale]

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

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

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

    if generate_json:

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

        # numpy arrays have to be turned into lists first
        vectorize = lambda x: x if x is None or np.isscalar(x) else list(x)

        logdata = {'time': timestring,
                   'filename': filename,
                   'betas': vectorize(beta),
                   'gammas': vectorize(gamma),
                   'c': vectorize(c),
                   'sigmas': list(scales)
                   }

        if dilate_per_scale:
            logdata['border_radii'] = border_radii

        if output_dir is None:
            output_dir = 'output'

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

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

    return F_all, dumpfile


def get_outname_lambda(filename, output_dir=None, timestring=None):
    """
    return a lambda function which can build output filenames
```

```
    """

    if output_dir is None:
        output_dir = 'output'

    base = os.path.basename(filename)
    *base, suffix = base.split('.')

    if timestring is None:
        time_of_run = datetime.datetime.now()
        timestring = time_of_run.strftime("%y%m%d_%H%M")

    outputstub = ''.join(base) +'_' + timestring +  '_{}.'+ suffix
    return lambda s: os.path.join(output_dir, outputstub.format(s))


def _build_scale_colormap(N_scales, base_colormap, basecolor=(0,0,0,1)):
    """
    returns a mpl.colors.ListedColormap with N samples,
    based on the colormap named "default_colormap" (a string)

    the N colors are given by the default colormap, and
    basecolor (default black) is added to map to 0.
    (you could change this, for example, to (1,1,1,1) for white)

    reversed colormaps often work better if the basecolor is black
    you should make sure there's good contrast between the basecolor
    and the first color in the colormap
    """

    map_range = np.linspace(0, 1, num=N_scales)

    colormap = plt.get_cmap(base_colormap)

    colorlist = colormap(map_range)

    # add basecolor as the first entry
    colorlist = np.vstack((basecolor, colorlist))

    return mpl.colors.ListedColormap(colorlist)


def scale_label_figure(wheres, scales, savefilename=None,
                       crop=None, show_only=False, image_only=False,
                       base_cmap='viridis_r', save_colorbar_separate=False,
                       basecolor=(0, 0, 0, 1), savecolorbarfile=None,
                       output_dir=None):
    """
    crop is a slice object.
    if show_only, then just plt.show (interactive).
    if image_only, then this will *not* be printed with the colorbar

    if save_colormap_separate, then the colormap will be saved as a separate
    file
    """
    if crop is not None:
        wheres = wheres[crop]

    fig, ax = plt.subplots()  # not sure about figsize
    N = len(scales)  # number of scales / labels

    tabemap = _build_scale_colormap(N, base_cmap, basecolor)

    if image_only:
```

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

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

        cbar.set_ticks(tick_locs)
        # label each tick with the sigma value
        scalelabels = [r"$\sigma = :.2f$".format(s) for s in scales]
        scalelabels.insert(0, "(no match)")
        # label with their sigma value
        cbar.set_ticklabels(scalelabels)
        # ax.set_title(r"Scale ($\sigma$) of maximum vesselness ")
        plt.tight_layout()
        # plt.savefig(outname('labeled'), dpi=300)
        if show_only or (savefilename is None):
            plt.show()
        else:
            plt.savefig(savefilename, dpi=300)

        plt.close()

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

listings/pd_demo_uniscale.py

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

import numpy as np
import numpy.ma as ma

import matplotlib.pyplot as plt
import matplotlib as mpl

from skimage.io import imread
from skimage.util import img_as_float

from placenta import (get_named_placenta, list_by_quality, cropped_args,
                      mimg_as_float)

from frangi import frangi_from_image
from hfft import fft_gradient, fft_hessian, fft_gaussian
from merging import nz_percentile
from plate_morphology import dilate_boundary
```

```
import os.path, os

from diffgeo import principal_curvatures, principal_directions


filename = list_by_quality(N=1)[0]
img = get_named_placenta(filename)
crop = cropped_args(img)

sigma = 1.5
img = mimg_as_float(img)

print('calculating frangi filter')

f = frangi_from_image(img, sigma=1.5, dark_bg=False, dilation_radius=20,
                      beta=0.35)

print('calculating hessian again (oops)')
H = fft_hessian(img, sigma=1.5)
print('calculating pd where f > .05')
v1, v2 = principal_directions(img, 1.5, H=H, mask=(f < 0.05))
print('done')
vm = ma.masked_array(v2, mask=f<.05)

# this colormap doesn't have any black in it!
cmap = mpl.cm.hsv
# so set the mask to black

cmap.set_bad(color=(0,0,0), alpha=1)

fig, ax = plt.subplots()
cax = ax.imshow(vm[crop], cmap=cmap, vmin=0, vmax=np.pi)
ax.axis('off')
cbar = fig.colorbar(cax, ticks=[0, np.pi/3, np.pi/2, 2*np.pi/3, np.pi])
cbar.ax.set_yticklabels(['0', r'$\frac{\pi}{3}$', r'$\frac{\pi}{2}$',
                         r'$\frac{2\pi}{3}$', r'$\pi$'])

ax.set_title(r'leading (local) principal direction, $\sigma = 1.5$')
fig.tight_layout()

plt.show() # save manually with the name pd_demo_uniscale.png
```

listings/placenta.py

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

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

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

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

import numpy as np
import numpy.ma as ma
import os.path
import os
import json
```

```
from scipy.ndimage import imread
from skimage import morphology

from numpy.ma import is_masked
from skimage.color import gray2rgb
from skimage.util import img_as_float
import matplotlib.pyplot as plt
from hfft import fft_gradient
from skimage.segmentation import watershed
from skimage.morphology import binary_erosion, disk
import scipy.ndimage as ndi


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

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

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

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

    if sample_dir is None:
        sample_dir = 'samples'

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

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

    except FileNotFoundError:
        print('Could not find file', typefile)
        return None

    return img


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

    """

```

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

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

    #TODO: expand this later to handle arterial traces and venous traces
    INPUT:
        base_filename: the name of the base file, not the tracefile itself
        as_binary: if True
    """

    if as_binary:
        mode = 'L'
    else:
        mode = 'RGB'

    T = open_typefile(base_filename, 'trace', sample_dir=sample_dir, mode=mode)

    if as_binary:

        return np.invert(T != 0)

    else:
        return T

def mimg_as_float(mimg):

    if not ma.is_masked(mimg):

        return img_as_float(mimg)

    else:
        return ma.masked_array(img_as_float(mimg.data),
                               mask=mimg.mask)

def get_named_placenta(filename, sample_dir=None, masked=True,
                       maskfile=None, mode='L'):
    """
    This function is to be replaced by a more ingenious/natural
    way of accessing a database of unregistered and/or registered
    placental samples.

    Parameters
    ----------

    filename: name of file (including suffix?) but NOT directory
    masked: return it masked.
    maskfile: if supplied, this use the file will use a supplied 1-channel
            mask (where 1 represents an invalid/masked pixel, and 0
            represents a valid/unmasked pixel. the supplied image must be
            the same shape as the image. if not provided, the mask is
            calculated (unless masked=False)
            the file must be located within the sample directory

            If maskfile is 'None' then this function will look for
            a default maskname with the following  pattern:

                test.jpg -> test.mask.jpg
                ncs.1029.jpg  -> ncs.1029.mask.jpg

    sample_directory: Relative path where sample (and mask file) is located.
            defaults to './samples'

```

```
    if masked is true (default), this returns a masked array.

    NOTE: A previous logical incongruity has been corrected. Masks should have
    1 as the invalid/background/mask value (to mask), and 0 as the
    valid/plate/foreground value (to not mask)
    """
    if sample_dir is None:
        sample_dir = 'samples'

    full_filename = os.path.join(sample_dir, filename)

    if mode.lower() in ('g', 'green'):
        # first channel of RGBA (or RGBA!)
        raw_img = imread(full_filename)[...,1]

    else:
        raw_img = imread(full_filename, mode=mode)

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

    if filename.startswith('T-BN'):
        if filename not in FAILS:
            #print('tightening the mask')
            raw = open_typefile(filename, 'raw')
            plate = find_plate_in_raw(raw)

            stuff = ma.masked_array(raw_img, mask=mask)
            return ma.masked_array(stuff, mask=plate)
        else:
            #print('leaving this mask as it is')
            pass

    return ma.masked_array(raw_img, mask=mask)


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

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

    N is the number of placentas to return (will return # of placentas
```

```
    of that quality or N, whichever is smaller)

    if json_name is not None just use that filename directly

    if return_empty then silenty failing is OK
    """

    quality_keys = ('good', 'okay', 'fair', 'poor')

    if quality in quality_keys:
        pass
    elif quality in (0, 1, 2, 3):
        quality = quality_keys[quality]
    else:
        try:
            quality = quality.lower()
        except AttributeError:
            if return_empty:
                return list()
            else:
                print(f'unknown quality {quality}')
                raise
        else:
            # if no json file is provided, and quality is a string,
            # just assume it follows a template format
            if json_file is None:
                json_file = f"{quality}-mccs.json"

    # if it's still not provided in the main file, it's in the main file
    if json_file is None:
        json_file = 'sample-qualities.json'

    try:
        with open(json_file, 'r') as f:
            D = json.load(f)
    except FileNotFoundError:
        if return_empty:
            return list()
        else:
            print('cannot find', json_file)
            raise FileNotFoundError

    if json_file == 'sample-qualities.json':
        # go one level deep
        placentas = [k for k in D[quality].keys()]
    else:
        placentas = [k for k in D.keys()]

    if N is not None:
        return placentas[:N]
    else:
        return placentas


def check_filetype(filename, assert_png=True, assert_standard=False):
    """
    'T-BN8333878.raw.png' returns 'raw'
    'T-BN8333878.mask.png' returns 'mask'
    'T-BN8333878.png' returns 'base'

    if assert_png is True, then raise assertion error if the file
    is not of type png

    if assert_standard, then assert the filetype is
```

```
    mask, base, trace, or raw.

    etc.
    """
    basename, ext = os.path.splitext(filename)

    if ext != '.png':
        if assert_png:
            assert ext == '.png'

    sample_name, typestub = os.path.splitext(basename)

    if typestub == '':
        # it's just something like  'T-BN8333878.png'
        return 'base'
    elif typestub in ('.mask','.trace', '.raw', '.ctrace', '.arteries', '.veins', '.ucip'):
        # return 'mask' or 'trace' or 'raw'
        return typestub.strip('.')
    else:
        print('unknown filetype:', typestub)
        print('is it a weird filename?')

        print('warning: lookup failed, unknown filetype:' + typestub)

        return typestub

def list_placentas(label='T-BN', sample_dir=None):
    """
    label is the specifier, basically just ''.startswith()

    only real use is to find all the T-BN* files

    this is hackish, if you ever decide to use a file other than
    png then this needs to change
    """

    if sample_dir is None:
        sample_dir = 'samples'

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

    placentas = list()

    for f in os.listdir(sample_dir):

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

    return sorted(placentas)


def show_mask(img, mask=None, interactive=False, mask_color=None):
    """
    rename this color_mask since showing the mask is just a secondary feature
    show a masked grayscale image with a dark blue masked region

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

```

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

    if mask provided, add it to the image. (pass img.data instead if you don't
    want to use the original mask)
    """

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

    # if there's no mask at all
    if (mask is None) and (not is_masked(img)):
        if interactive:
            plt.imshow(img, cmap=plt.cm.gray)
            return  # we're done
        else:
            # return as an rgb image so output is uniform
            return gray2rgb(img)

    elif not is_masked(img):
        # add mask to the image / add to existing mask
        # if i just rewrite img will it change outside this function?
        new_img = ma.masked_array(img, mask=mask)
    else:
        new_img = img.copy()

    # otherwise, get an RGB array, black where the mask is
    mimg = gray2rgb(new_img.filled(0))

    # fill masked regions with the mask color
    mimg[new_img.mask, :] = mask_color

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


def _cropped_bounds(img, mask=None):

    if mask is not None:

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

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

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

    return Y[0], Y[-1], X[0], X[-1]


def cropped_args(img, mask=None):
    """
    get a slice that would crop image
    i.e. img[cropped_args(img)] would be a cropped view
    """

    x0, x1, y0, y1 = _cropped_bounds(img, mask=None)
```

```
403     return np.s_[x0:x1, y0:y1]
404
405
406 def cropped_view(img, mask=None):
407     """
408     removes entire masked rows and columns from the borders of a masked array.
409     will return a masked array of smaller size
410
411     don't ask me about data
412
413     the name sucks too
414     """
415
416     # find first and last row with content
417     x0, x1, y0, y1 = _cropped_bounds(img, mask=mask)
418
419     return img[x0:x1, y0:y1]
420
421
422 CYAN = [0, 255, 255]
423 YELLOW = [255, 255, 0]
424
425 def find_plate_in_raw(raw, sigma=.01):
426     g = fft_gradient(raw[...,1],sigma=.01)
427     marks=np.zeros(g.shape, np.int32)
428     marks[0,0] = 1
429     marks[g > g.mean()] = 2
430     #marks[g > np.percentile(g,25)] = 2
431     w = watershed(g,marks)
432
433     eroded = binary_erosion(w==2, disk(15))
434
435     labeled, n_labs = ndi.label(eroded)
436
437     # get largest object (0 is gonna be background)
438     # sort labels by decreasing magnitude
439     labs_by_size = sorted(list(range(1,n_labs+1)),
440                           key=lambda l: np.sum(labeled==l), reverse=True)
441
442     # unless something went horribly wrong
443     plate_index = labs_by_size[0]
444
445     return ~(labeled == plate_index)
446
447
448 FAILS = [
449     "T-BN0687730.png",
450     "T-BN1629357.png",
451     "T-BN2050224.png",
452     "T-BN6381701.png",
453     "T-BN7476220.png",
454     "T-BN7644170.png",
455     "T-BN7767693.png",
456 ]
457
458 def measure_ncs_markings(ucip_img=None, filename=None, verbose=False):
459     """
460     find location of ucip and resolution of image based on input
461     (similar to perimeter layer in original NCS data set
462
463     Parameters
464     ----------
465
466     ucip_img: an RGB ndarray or None
```

```
        The perimeter layer of an NCS sample (colorations according to the
        tracing protocol). if None, filename must be included. Default is None.
    filename:
        the filename of the SAMPLE (not the ucip image file itself)

    Returns
    -------
    m : tuple of ints
        the coordinates of (the center of) of the umbilical cord point
        (depicted as a yellow dot) in the original image.
    resolution: a float
        measured distance between the two cyan dots
    """

    if ucip_img is None:
        ucip_img = open_typefile(filename, 'ucip')

    if ucip_img is None:
        # if it's still none (no file), return None
        return None, None

    # just in case it's got an alpha channel, remove it
    img = ucip_img[:, :, 0:3]

    # given the image img (make sure no alpha channel)
    # find all cyan pixels (there are two boxes of 3 pixels each and we
    # just want to extract the middle of each
    if verbose:
        print('the image size is {}x{}'.format(img.shape[0], img.shape[1]))

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

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

    assert X.shape == Y.shape

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

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

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

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

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

    if verbose:
        print('the middle of the UCIP location is', mid)
```

```
        print('the radius outward is', radius)
        print('the total measurable diameter is', radius*2 + 1)

    return mid, ruler_distance


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

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

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

    # this is way faster than dilating the point in the matrix,
    # just set this at the centered point

    # doesn't check for out of bounds stuff. use at your own peril
    D = morphology.disk(radius)
    to_add[m[0]-radius:m[0]+radius+1, m[1]-radius:m[1]+radius+1] = D

    # merge with supplied mask
    return mask | to_add


if __name__ == "__main__":

    """test that this works on an easy image."""

    test_filename = 'barium1.png'

    img = get_named_placenta(test_filename, maskfile=None)

    print('showing the mask of', test_filename)
    print('run plt.show() to see masked output')

    show_mask(img, interactive=True)
```

listings/plate_test.py

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

from placenta import (get_named_placenta, list_placentas, open_typefile,
                      show_mask)
import numpy as np
import matplotlib.pyplot as plt
from preprocessing import mask_stump
import numpy.ma as ma
import sys
from skimage.exposure import equalize_adapthist
```

```
from skimage.util import img_as_int
from hfft import fft_gradient

from skimage.segmentation import watershed
import scipy.ndimage as ndi
from skimage.morphology import binary_erosion, disk

def find_plate_in_raw(raw, sigma=.01):
    g = fft_gradient(raw[...,1],sigma=.01)
    marks=np.zeros(img.shape, np.int32)
    marks[0,0] = 1
    marks[g > g.mean()] = 2
    #marks[g > np.percentile(g,25)] = 2
    w = watershed(g,marks)

    eroded = binary_erosion(w==2, disk(15))

    labeled, n_labs = ndi.label(eroded)

    # get largest object (0 is gonna be background)
    # sort labels by decreasing magnitude
    labs_by_size = sorted(list(range(1,n_labs+1)),
                          key=lambda l: np.sum(labeled==l), reverse=True)

    # unless something went horribly wrong
    plate_index = labs_by_size[0]

    return ~(labeled == plate_index)

FAILS = [
    "T-BN0687730.png",
    "T-BN1629357.png",
    "T-BN2050224.png",
    "T-BN6381701.png",
    "T-BN7476220.png",
    "T-BN7644170.png",
    "T-BN7767693.png",
]

for n, filename in enumerate(FAILS):

    print(filename)

    raw = open_typefile(filename, 'raw')
    img = get_named_placenta(filename)

    fig, ax = plt.subplots(ncols=3, nrows=1, figsize=(30,12))

    ax[0].imshow(raw)
    ax[0].set_title(filename)

    plate = find_plate_in_raw(raw)
    newmask = show_mask(ma.masked_array(img, mask=plate))
    ax[1].imshow(newmask)
    ax[2].imshow(plate*1. + img.mask*2, vmin=0, vmax=3)

    plt.show()

    if input('make more?') == 'n':
        sys.exit(0)
```

listings/plate_morphology.py

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

from skimage.morphology import (disk, binary_erosion, binary_dilation,
                                convex_hull_image, thin)
from skimage.segmentation import find_boundaries, watershed

from placenta import open_typefile, get_named_placenta

import numpy as np
import numpy.ma as ma

def dilate_boundary(img, radius=10, mask=None):
    """
    grows the mask by a specified radius of a masked 2D array
    Manually remove (erode) the outside boundary of a plate.
    The goal is remove any influence of the zeroed background
    on reporting derivative information.

    There is varying functionality here (maybe should be multiple functions
    instead?)

    If img is a masked array and mask=None, the mask will be dilated and a
    masked array is outputted.

    If img is any 2D array (masked or unmasked), if mask is specified, then
    the mask will be dilated and the original image will be returned as a
    masked array with a new mask.

    If the img is None, then the specified mask will be dilated and returned
    as a regular 2D array.

    """

    if mask is None:
        # grab the mask from input image
        # if img is None this will break too but not handled
        try:
            mask = img.mask
        except AttributeError:
            raise('Need to supply mask information')

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

    maskpad = np.zeros_like(perimeter)

    M,N = maskpad.shape
    for i,j in np.argwhere(perimeter):
        # just make a cross shape on each of those points
        # these will silently fail if slice is OOB thus ranges are limited.
        maskpad[max(i-radius,0):min(i+radius,M),j] = 1
        maskpad[i,max(j-radius,0):min(j+radius,N)] = 1

    new_mask = np.bitwise_or(maskpad, mask)

    if img is None:
        return new_mask # return a 2D array
    else:
        # replace the original mask or create a new masked array
        return ma.masked_array(img, mask=new_mask)


def l2_dist(p,q):
    return int(np.round(np.sqrt((p[0]-q[0])**2 + (p[1]-q[1])**2)))
```

```
64
65
66  def mask_cuts_simple(img, ucip, mask_only=False, in_place=False,
67                       return_success=False):
68      """
69      this covers up the cut with a disc originating at the perimeter of
70      significant radius
71      """
72
73      cutmarks = np.all(ucip==(0,0,255), axis=-1)
74      B = np.all(ucip==(0,0,255), axis=-1)
75      dilcut = img.copy()
76
77      if not np.any(cutmarks):
78
79          #print("no cutmarks found on image")
80
81          if return_success:
82              return img, False
83          else:
84              return img
85      else:
86          #print("found a cutmark!")
87          pass
88
89      cutmarks = np.nonzero(cutmarks)
90      # get the first pixel of it (we don't need to be too precise here)
91      G = np.all(ucip==(0,255,0), axis=-1) # perimeter elements
92      cutmarks = np.nonzero(thin(B))
93      perimeter = np.nonzero(G)
94
95      cutinds = np.stack(cutmarks).T
96
97      for P in cutinds:
98
99          # consider larger and larger window sizes
100         for W in [100,200,300]:
101             # consider all perimeter elements within these bounds
102
103             rmin, rmax = max(0, P[0]-W), min(img.shape[0], P[0]+W)
104             cmin, cmax = max(0, P[1]-W), min(img.shape[1], P[1]+W)
105             #window = np.s_[rmin:rmax, cmin:cmax]
106
107             # perimeter indices within the window
108             pinds = [(x,y) for x, y in zip(*perimeter)
109                         if x > rmin and x < rmax and y > cmin and y < cmax
110                         ]
111             if pinds:
112                 break #otherwise increase the size of the window
113     if pinds:
114
115         # max distance to boundary point in the window
116         # we really only need to keep the largest; deque?
117         dists = sorted([(pp, l2_dist(P,pp)) for pp in pinds],
118                        key=lambda t: t[1])
119         r = 2*int(dists[0][1]) + 1 # get largest radius but closest point
120         P = dists[0][0]
121         B = np.zeros_like(img.mask)
122
123         B[cutmarks] = True
124
125         # center a disk of found radius there
126         D = disk(r)
127         winx = max(P[0]-r,0), min(P[0]+r+1,B.shape[0])
```

```
        winy = max(P[1]-r,0), min(P[1]+r+1,B.shape[1])
        try:
            B[winx[0]:winx[1] , winy[0]:winy[1]] = D
        except ValueError:
            # they're out of bounds so it's a size mismatch. fix it
            # by starting/ending D index with opposite sign of the initial
            # p +/- radius that was out of bounds
            # for example P[0]-r was -9 and everything else was fine
            # so you just need to set left side to D[9:,:]
            # but you should  wrap this up in a function so the three times
            # you do it here and the one time in ucip all gets the same
            # code
            print("too close to the boundary or size mismatch?")
            success = False
        else:
            dilcut[B] = ma.masked
            success = True
    else:
        print("we completely failed to mask the cut. too close to the",
              "boundary to fit an unmodified disk in. fix this")
        success = False

    return dilcut, success


def mask_cuts_watershed(img, ucip, mask_only=False, in_place=False,
                        return_success=False):
    """

    this doesn't handle any image, io. just provide the ucip img and the
    base (masked) image and we'll fix the mask

    ucip is the actual RGB array, not the file. do io elsewhere.

    if mask_only, this will simply return the new mask as a 2D boolean array.
    Otherwise, it returns a masked_array.
    The cut region will be added to the img's mask. If you really want just the
    difference, you'll have to to run
    >>>(cut_mask & ~img.mask) youself.

    yourself.

    If in_place, this changes the mask of the image directly (but still returns
    a masked array. If mask_only is True, in_place will automatically be set to
    False to prevent hideous side effects

    if return_success, this function returns True if there was a cutmark found,
    otherwise false as a second output
    """
    # get indices where the blue square indicating center of a cut apprears
    cutmarks = np.all(ucip==(0,0,255), axis=-1)

    if not np.any(cutmarks):

        #print("no cutmarks found on image")

        if return_success:
            return img, False
        else:
            return img
    else:
        #print("found a cutmark!")
        pass

```

```
    cutmarks = np.nonzero(cutmarks)
    # get the first pixel of it (we don't need to be too precise here)
    X, Y = cutmarks[0][0], cutmarks[1][0]

    # get a value somewhat lower than the value of bg in the cut
    # (this should be a high number before we take 85%)
    # sometimes this is in a shadowy region which fucks everything up though
    #threshold = max(img[cutmarks].mean() * .85, 175)
    # get the brightest value in a smallish window around the cut * .85
    threshold = np.max(img[X-10:X+10,Y-10:Y+10])

    rmin, rmax = max(0, X-100), min(img.shape[0], X+100)
    cmin, cmax = max(0, Y-100), min(img.shape[1], Y+100)
    cutregion = np.s_[rmin:rmax, cmin:cmax] # get a window around the mark

    # mark inside of the placenta with label 2, original mask and cutmarks with
    # label 1, and the rest with 0 (i dunno)
    markers = np.zeros(img.shape, dtype='int32')
    markers[img.filled(255) < threshold] = 2
    markers[img.mask] = 1
    markers[cutmarks] = 1

    # perform watershedding on the thresholded image to fill in the cut with
    # label 1
    cutfix = watershed(img.filled(255) < threshold, markers=markers)

    # this is a waste considering the in_place, but eh
    new_mask = img.mask.copy()

    new_mask[cutregion] = (cutfix[cutregion] == 1)

    if mask_only:

        out = new_mask

    elif not in_place:

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

    else:

        # will this work?
        img[new_mask] = ma.masked

        out = img

    # now return succeed if asked to
    if return_success:
        return out, True

    else:
        return out


if __name__ == "__main__":

    # DEMO FOR SHOWING OFF DILATE_BOUNDARY EFFECT

    from placenta import get_named_placenta
    from frangi import frangi_from_image
    import matplotlib.pyplot as plt

    import os.path

```

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

    sigma = 3
    radius = 25

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

    D = dilate_boundary(img, radius=radius)

    Fimg = frangi_from_image(img, sigma, dark_bg=False, dilation_radius=None)
    FD = frangi_from_image(D, sigma, dark_bg=False)
    FDinv = frangi_from_image(D, sigma, dark_bg=True)
    Finv = frangi_from_image(img, sigma, dark_bg=True, dilation_radius=None)

    fig, axes = plt.subplots(ncols=2, nrows=3)

    axes[0,0].imshow(img[inset].filled(0), cmap=plt.cm.gray)
    axes[0,1].imshow(D[inset].filled(0), cmap=plt.cm.gray)
    axes[1,0].imshow(Fimg[inset].filled(0), cmap=plt.cm.nipy_spectral)
    axes[1,1].imshow(FD[inset].filled(0), cmap=plt.cm.nipy_spectral)
    axes[2,0].imshow(Finv[inset].filled(0), cmap=plt.cm.nipy_spectral)
    axes[2,1].imshow(FDinv[inset].filled(0), cmap=plt.cm.nipy_spectral)

    for a in axes.ravel():
        # get rid of all the labels
        plt.setp(a.get_xticklabels(), visible=False)
        plt.setp(a.get_yticklabels(), visible=False)

    # lol matlab
    for i in range(5):
        fig.tight_layout()

    plt.savefig(os.path.join(dest_dir, "boundary_dilation_demo.png"), dpi=300)
```

listings/postprocessing.py

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

"""
doing things to the Frangi targets, i.e. feeding them into other algorithms
"""

from skimage.filters import sobel
from skimage.morphology import (remove_small_objects, thin, disk,
                                binary_dilation)

from frangi import frangi_from_image
from merging import apply_threshold, nz_percentile
from plate_morphology import dilate_boundary
from skimage.segmentation import random_walker
import numpy as np

from skimage.filters.rank import enhance_contrast_percentile as ecp
from scipy.ndimage import distance_transform_edt as edt


def random_walk_fill(img, Fmax, high_thresh, low_thresh, dark_bg):
    """
    # this is deprecated, it's trash
    """

```

```
    s = sobel(img)
    s = dilate_boundary(s, mask=img.mask, radius=20)

    finv = frangi_from_image(img, sigma=0.8, beta=0.5, dark_bg=(not dark_bg),
                             dilation_radius=20)

    finv_thresh = (finv > nz_percentile(finv, 50)).filled(0)
    margins = remove_small_objects(finv_thresh, min_size=32)

    markers = np.zeros(img.shape, dtype=np.int32)
    markers[Fmax < low_thresh] = 1

    margins_added = (margins | (Fmax > high_thresh))
    #margins_added = remove_small_holes(margins_added, area_threshold=50)

    markers[Fmax < low_thresh] = 1

    markers[margins_added] = 2

    rw = random_walker(1-Fmax, markers, beta=1000)

    approx_rw = (rw == 2)

    return approx_rw, markers, margins_added


def random_walk_scalewise(F, high_thresh=0.4, rw_beta=130,
                          return_labels=False):
    """Random walker on each a multiscale Frangi result"""
    print('doing scalewise random walk', end=' ')
    V = np.transpose(F, axes=(2, 0, 1))
    W = np.zeros(V.shape, np.bool)
    for n, v in enumerate(V):
        print('σ', end='', flush=True)
        markers = np.zeros(v.shape, np.int32)
        markers[v == 0] = 1
        # this could be a vector too
        markers[v > high_thresh] = 2
        # or 1-v
        W[n] = (random_walker(v, markers, rw_beta) == 2)
    print()
    if not return_labels:
        return W.any(axis=0)
    else:
        # argmax grabs the first scale where it was satisfied
        # so this will grab the lowest scale that matches
        return W.any(axis=0), W.argmax(axis=0)


def dilate_to_rim(spine, rim, thin_spine=False, max_radius=15,
                  return_radii=False):
    """Grow spine region by dilating towards closest rim point of the trough.

    Parameters:
    -----------
    spine, rim: 2D ndarray type bool
        spine are points, dilate to rim
    thin_spine: bool, optional
        Thin the spine before performing dilation. This makes it much faster
        at the risk of giving an incomplete fill if the spine isn't in the
        middle of the margins, also makes returning the labels more meaningful

    Returns
    -------
```

```
    ...

    """

    D = np.ones(spine.shape, np.bool)
    D[rim] = 0

    spine_dists = edt(D)

    # either thin first or don't, i dunno
    if thin_spine:
        spine_dists[~thin(spine)] = 0
    else:
        spine_dists[~spine] = 0

    spine_radii = np.round(spine_dists).astype('int')

    dilation_stack = np.stack([binary_dilation(spine_radii==r, selem=disk(r))
                               for r in range(1,max_radius+1)])

    approx = dilation_stack.any(axis=0)

    if not return_radii:
        return approx
    else:
        return approx, dilation_stack.argmax(axis=0)
```

listings/preprocessing.py

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

# TODO: refactor this so inpaint glare is the main function that takes
#       a keyword argument strategy='hybrid' or whatever then you can run
#       >>>for s in ['mean_window', 'median_boundary', 'biharmonic', 'hybrid']:
#           timeit.timeit('inpaint_glare(img, strategy=s))', globals=globals())
#
#       ... but it's annoying since you'll need a way to pass args to the
#           particular strategy

from skimage.morphology import (binary_dilation, disk, remove_small_objects,
                                convex_hull_object)
from skimage.restoration import inpaint_biharmonic
import numpy as np
import numpy.ma as ma
from scipy.ndimage import label
from skimage.util import img_as_float
from skimage.segmentation import find_boundaries
from plate_morphology import dilate_boundary

import scipy.ndimage as ndi
import matplotlib.pyplot as plt

def inpaint_glare(img, threshold=175, window_size=15, mask=None):
    """
    img is a masked array type uint [0,255]
    """

    # bool array, true where glare
    if mask is None:
        glared = mask_glare(img, threshold=threshold, mask_only=True)
    else:
        glared = mask
```

```
34
35      B = ma.masked_array(img, mask=glared)  # masked background *and* glare
36      new_img = img.copy()  # copy values of original image (will rewrite)
37      d = int(window_size)
38
39      for j, k in zip(*np.where(glared)):
40          # rewrite all glared pixels with the mean of nonmasked elements
41          # in a window_size window. (this doesn't check OoB, be careful!)
42          new_img[j, k] = B[j-d:j+d, k-d:k+d].compressed().mean()
43
44      return new_img
45
46
47  def inpaint_with_boundary_median(img, threshold=175, mask=None):
48      """
49      mask glare pixels, then replace by the median value on the mask's boundary
50      """
51      if mask is None:
52          glared = mask_glare(img, threshold=threshold, mask_only=True)
53      else:
54          glared = mask
55
56      B = ma.masked_array(img, mask=glared)
57
58      new_img = img.copy()  # copy values of original image (will rewrite)
59      bounds = find_boundaries(glared)
60      lb, _ = label(bounds)
61      fill_vals = np.zeros_like(img.data)
62
63      # for each boundary of masked region, find the median value of the img
64      for lab in range(1, lb.max()+1):
65          inds = np.where(lb == lab)
66          fill_vals[inds] = nz_median(B[inds])
67
68      # label masked regions together with their boundaries (they'll be
69      # connected)
70      lm, _ = label(np.logical_or(glared, lb != 0))
71
72      # fill the masked areas with the corresponding fill value
73      for lab in range(1, lm.max()+1):
74          inds = np.where(lm == lab)
75          # find locations of filled values corresponding to this label
76          # median in case there's overlapped regions? (sloppy)
77          replace_value = nz_median(fill_vals[inds])
78
79          if replace_value == 0:
80              raise
81
82          fill_vals[inds] = replace_value
83
84      # now fill in the values
85      new_img[glared] = fill_vals[glared]
86
87      return new_img
88
89  def nz_median(A):
90
91      if ma.is_masked(A):
92          relevant = A[A > 0].compressed()
93      else:
94          relevant = A[A > 0]
95
96      return np.median(relevant)
97
```

```
98
99  def inpaint_hybrid(img, threshold=175, min_size=64, boundary_radius=10):
100     """
101     use biharmonic inpainting in larger, inner areas (important stuff)
102     and median inpainting in smaller areas and along boundary
103     """
104
105     glare = mask_glare(img, threshold=threshold, mask_only=True)
106
107     glare_inside = dilate_boundary(glare, mask=img.mask,
108                                    radius=boundary_radius).filled(0)
109
110     large_glare = remove_small_objects(glare_inside, min_size=min_size,
111                                        connectivity=2)
112     small_glare = np.logical_and(glare, np.invert(large_glare))
113
114     # inpaint smaller and less important values with less expensive method
115     inpainted = inpaint_with_boundary_median(img, mask=small_glare)
116     hybrid = img_as_float(inpainted) # scale 0 to 1
117
118     # inpaint larger regions with biharmonic inpainting
119     large_inpainted = inpaint_biharmonic(img.filled(0), mask=large_glare)
120
121     # now overwrite with these values
122     hybrid[large_glare] = large_inpainted[large_glare]
123
124     # put on old image mask
125     return ma.masked_array(hybrid, mask=img.mask)
126
127 def inpaint_with_biharmonic(img, threshold=175):
128     """
129     use biharmonic inpainting *all* glare
130     """
131     glare = mask_glare(img, threshold=threshold, mask_only=True)
132     inpainted = inpaint_biharmonic(img_as_float(img.filled(0)), mask=glare)
133
134     if ma.is_masked(img):
135         return ma.masked_array(inpainted, mask=img.mask)
136     else:
137         return inpainted
138
139 def mask_glare(img, threshold=175, mask_only=False):
140     """
141     for demoing purposes, with placenta.show_mask
142
143     if mask_only, just return the mask. Otherwise return a copy of img with
144     that added to the mask. If you want the original mask to be ignored,
145     just pass img.filled(0) ya doofus
146
147     threshold is expected to be of the same dtype as img *unless# it assumes
148     its default value, in which case the threshold will be converted to a float
149
150     """
151     # if img.dtype is floating but threshold value is still the default
152     # this could be generalized
153     if np.issubdtype(img.dtype, np.floating) and (threshold == 175):
154         threshold = 175 / 255
155     # region to inpaint
156     inp = (img > threshold)
157
158     # get a larger area around the specks
159     inp = binary_dilation(inp, selem=disk(2))
160
161     # remove anything large
```

```
    #inp = white_tophat(inp, selem=disk(3))

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


def mask_stump(img, mask=None, mask_only=True):

    if img.ndim < 3:
        print('better to pass the raw image')
        channel = img
    else:
        channel = img[...,0]

    C = channel.copy()
    if mask is not None:
        C[mask] = 0
    elif ma.is_masked(img):
        C[img.mask] = 0
        mask = img.mask
    else:
        mask = np.zeros(img.shape, np.bool)

    thresh = (170/255)*C.max()
    b = ndi.white_tophat(C > thresh, 90)
    b = remove_small_objects(b, 1000)
    #b = convex_hull_object(b)
    b[mask] = 0

    # sort by size of object (largest first)

#    incl_count = 0 #objects used
#    mags = sorted(list(range(1,n_labs+1)), key=lambda lb: np.sum(labs==lb),
#            reverse=True)
#    print(mags)
#    for l in mags:
#        # big things get weird
#        if np.sum(b==l) > 5000:
#            print('skipping very large object')
#            # get rid of it, whatever
#            plt.imshow(b==l)
#            b[b==l] = 0
#        else:
#            incl_count += 1
#
#        if incl_count > 4:
#            print('only removing a few things here')
#            b[b==l] = 0
#
#    print('b after')
#    plt.imshow(b)
#    b = ndi.binary_dilation(b, disk(15))
#
    if mask_only:
        return b

    else:
        # will add to existing mask
        return ma.masked_array(img, mask=b)

```

```
226
227
228 DARK_RED = np.array([103, 15, 23]) / 255.
229
230 # test it on a particularly bad sample
231 if __name__ == "__main__":
232
233     from placenta import get_named_placenta, show_mask
234     import matplotlib.pyplot as plt
235
236     filename = 'T-BN0204423.png'  # a particularly glary sample
237     img = get_named_placenta(filename)
238
239     img = ma.masked_array(img_as_float(img), mask=img.mask)
240     crop = np.s_[150:500, 150:800]  # indices to zoom in on the region
241     zoom = np.s_[300:380, 300:380]  # even smaller region
242
243     inset = zoom  # which view to use
244
245     masked = mask_glare(img)  # for viewing
246     inpainted = inpaint_glare(img)
247     minpainted = inpaint_with_boundary_median(img)
248     hinpainted = inpaint_hybrid(img)
249     binpainted = inpaint_with_biharmonic(img)
250
251     # view the closeup like this
252     minpainted_view = show_mask(minpainted, interactive=False,
253                                 mask_color=DARK_RED)
254     inpainted_view = show_mask(inpainted, interactive=False,
255                                mask_color=DARK_RED)
256     masked_view = show_mask(masked, interactive=False,
257                             mask_color=DARK_RED)
258     img_view = show_mask(img, interactive=False,
259                          mask_color=DARK_RED)
260     hinpainted_view = show_mask(hinpainted, interactive=False,
261                                 mask_color=DARK_RED)
262     binpainted_view = show_mask(binpainted, interactive=False,
263                                 mask_color=DARK_RED)
264
265     # view them all next to each other
266
267     fig, axes = plt.subplots(ncols=3, nrows=2)
268
269     axes[0,0].imshow(img_view[inset])
270     axes[0,1].imshow(masked_view[inset])
271     axes[0,2].imshow(inpainted_view[inset])
272     axes[1,0].imshow(minpainted_view[inset])
273     axes[1,1].imshow(binpainted_view[inset])
274     axes[1,2].imshow(hinpainted_view[inset])
275
276     for a in axes.ravel():
277         # get rid of all the labels
278         plt.setp(a.get_xticklabels(), visible=False)
279         plt.setp(a.get_yticklabels(), visible=False)
280
281     # lol matlab
282     for i in range(5):
283         fig.tight_layout()
284
285     IMGS = np.vstack((
286         np.hstack((img_view, masked_view, inpainted_view)),
287         np.hstack((minpainted_view, binpainted_view, hinpainted_view))))
288
289     # THEN IMSAVE
```

```
290
291     # plt.imsave('preprocessing_comparison_cropped.png',  IMGS)
292     # plt.imsave('preprocessing_comparison_zoomed.png',  IMGS)
293
294     # if it's zoomed, then rescale the output in GIMP to 4x
```

listings/process_NCS_xcfs.py

```
1  #!/usr/bin/env python
2
3  """
4  This should be a plugin to take images from the folder NCS_vessel_GIMP_xcf
5  and create trace, mask, and backgrounded images from each xcf file.
6
7  to use:
8  chmod +x and then copy or link to  ~/gimp-2.x/plug-ins/
9  """
10
11 from gimpfu import *
12 import os.path
13 from functools import partial
14
15 #basefile, ext = os.path.splitext(xcffile)
16
17 def _outname(base, s=None):
18
19     #base = base.split("_", maxsplit=1)[0]
20     if s is None:
21         stubs = (base, 'png')
22     else:
23         stubs = (base, s, 'png')
24     file
25     filename = '.'.join(stubs)
26
27     return os.path.join(os.getcwd(), filename)
28
29 # get active image
30 def process_NCS_xcf(timg,tdrawable):
31     img = timg
32     basename, _ = os.path.splitext(img.name) # split off extension .xcf
33     basename = basename.split("_")[0] # only get T-BN-kjlksf part
34     print "*"*80
35     print '\n\n'
36
37     print "Processing " , img.name
38     # generate output names easier
39     outname = partial(_outname, base=basename)
40
41     # get coordinates of the center
42     cx, cy = img.height // 2 , img.width // 2
43
44     # disable the undo buffer
45     img.disable_undo()
46
47     #perimeter = pdb.gimp_image_get_layer_by_name(img, 'perimeter')
48
49     for layer in img.layers:
50         if layer.name.lower() in ('perimeter', 'perimeters'):
51             # .copy() has optional arg of "add_alpha_channel"
52             mask = layer.copy()
53             break
54     else:
```

```
        print "Could not find a perimeter layer."
        print "Layers of this image are:"
        for n,layer in enumerate(img.layers):
            print "\t", n, ":", layer.name
        print "Skipping this file."

        return

    for layer in img.layers:
        layer.visible = False

    mask.name = "mask" # name the new layer
    img.add_layer(mask,0) # add in position 0 (top)

    pdb.gimp_layer_flatten(mask) # Remove Alpha Channel.

    # save the annotated perimeter file (for calculations later)
    pdb.gimp_file_save(img,mask, outname(s="ucip"), '')

    # remove unneeded annotations from mask layer
    # color exchange yellow & blue to black
    pdb.plug_in_exchange(img,mask,255,255,0,0,0,0,1,1,1)
    pdb.plug_in_exchange(img,mask,0,0,255,0,0,0,1,1,1)

    # set FG color to black (for tools, not of image)
    gimp.set_foreground(0,0,0)

    # Bucket Fill Inside black (center pixel is hopefully fine,
    # do rest manually
    pdb.gimp_edit_bucket_fill(mask,0,0,100,0,0,cx,cy)

    # Color Exchange Green to White.
    pdb.plug_in_exchange(img,mask,0,255,0,255,255,255,1,1,1)

    # Color Exchange Cyan (00ffff) to White.
    pdb.plug_in_exchange(img,mask,0,255,255,255,255,255,1,1,1)

    # Export Layer as Image called "f".mask.png
    pdb.gimp_file_save(img,mask, outname(s="mask"), '')

    # invert (so exterior is now black)
    pdb.gimp_invert(mask)
    mask.mode = DARKEN_ONLY_MODE # the constant 9

    # set bottom layer (placenta) to visible
    raw = img.layers[-1]
    raw.visible = True

    # now make a new layer called 'raw_img' from visible
    base = pdb.gimp_layer_new_from_visible(img,img,'base')
    img.add_layer(base,0)
    pdb.gimp_file_save(img , base, outname(s=None) , '')

    # now get rid of mask and save the raw image
    mask.visible = False
    pdb.gimp_file_save(img, base, outname(s='raw') ,'')


    # now make the other one visible (this is dumb)
    for layer in img.layers:
        if layer.name.lower() in ("arteries", "veins"):
            layer.visible = True
        else:
            layer.visible = False
```

```
    # now with these two visible, merge them and add layer
    trace = pdb.gimp_layer_new_from_visible(img,img,'trace')
    img.add_layer(trace,0)

    pdb.gimp_layer_flatten(trace) # remove alpha channel

    # don't turn binary anymore
    #pdb.gimp_desaturate(trace) # turn to grayscale
    #pdb.gimp_threshold(trace,255,255) # anything not 255 turns black

    pdb.gimp_file_save(img, trace, outname(s='ctrace') ,'')

    # now extract an each type individually.
    found = 0
    for subtype in ("arteries", "veins"):
        for layer in img.layers:
            if layer.name.lower() == subtype:
                layer.visible = True
                pdb.gimp_layer_flatten(layer) # remove alpha channel
                pdb.gimp_file_save(img, layer, outname(s=subtype), '')
                layer.mode = 9 # set to darken only (for merging)
                found += 1
            else:
                layer.visible = False
    if found < 2:
        print "WARNING! Could not find appropriate artery/vein layers."


    print "Saved. "


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

main()
```

listings/rw_demo.py

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

import numpy as np
import numpy.ma as ma

import matplotlib.pyplot as plt
from skimage.util import img_as_float
from skimage.io import imread
from placenta import (get_named_placenta, list_by_quality, cropped_args,
                      mimg_as_float, open_typefile, open_tracefile,
                      measure_ncs_markings, add_ucip_to_mask)

from frangi import frangi_from_image
from hfft import fft_gradient, fft_hessian, fft_gaussian
```

```
from merging import nz_percentile
from plate_morphology import dilate_boundary
import os.path, os
from scoring import confusion, mcc
from preprocessing import inpaint_hybrid

import matplotlib as mpl

from skimage.segmentation import random_walker

import json


def rw_demo(filename, rw_beta, threshold, output_dir=None):


    # ideally this would be a class with all of these
    cimg = open_typefile(filename, 'raw')
    ctrace = open_typefile(filename, 'ctrace')
    trace = open_tracefile(filename)
    img = get_named_placenta(filename)
    crop = cropped_args(img)
    ucip = open_typefile(filename, 'ucip')
    img = inpaint_hybrid(img)

    # make the size of figures more consistent
    if img[crop].shape[0] > img[crop].shape[1]:
        # and rotating it would be fix all this automatically
        cimg = np.rot90(cimg)
        ctrace = np.rot90(ctrace)
        trace = np.rot90(trace)
        img = np.rot90(img)
        ucip = np.rot90(ucip)
        crop = cropped_args(img)

    ucip_midpoint, _ = measure_ncs_markings(ucip)
    ucip_mask = add_ucip_to_mask(ucip_midpoint, radius=60, mask=img.mask)

    plt.close('all')

    cm = mpl.cm.plasma
    cmscales = mpl.cm.magma
    cm.set_bad('k', 1)  # masked areas are black, not white
    cmscales.set_bad('w', 1)

    scales =np.logspace(-1.5, 3.5, num=12, base=2)

    W = np.zeros((len(scales), *img.shape), dtype=np.bool)
    WL = np.zeros((len(scales), *img.shape), dtype=np.bool)
    V = np.zeros((len(scales), *img.shape))

    #fig, ax = plt.subplots(ncols=3, nrows=len(scales), figsize=(10,10))

    for n, sigma in enumerate(scales):

        f = frangi_from_image(img, sigma, dark_bg=False, dilation_radius=20,
                              beta=0.35)
        V[n] = f

        f[f == 0] = ma.masked

        fig, ax = plt.subplots(ncols=4, nrows=1, figsize=(20,6))

        ax[0].imshow(f[crop], cmap=cm)
```

```
        ax[0].axis('off')
        ax[0].set_title(rf'Frangi score, ($\sigma_n = sigma:.3f$)')

        markers = np.zeros(img.shape, np.int32)
        markers[f.mask] = 1
        markers[f > THRESHOLD] = 2

        ax[1].imshow(markers[crop], cmap=plt.cm.viridis, vmin=0, vmax=3)
        ax[1].axis('off')
        ax[1].set_title('markers')

        rw = random_walker(1-f.filled(0), markers, beta=RW_BETA)
        rw_L = random_walker(f.filled(0) > 0, markers, beta=RW_BETA)
        W[n] = (rw == 2)
        WL[n] = (rw_L == 2)

        # set the new stuff to a higher number so you can see what was added
        show_added = rw.copy()
        show_added_L = rw_L.copy()
        show_added[~(markers == 2) & (rw==2)] = 3
        show_added_L[~(markers == 2) & (rw_L==2)] = 3
        # set the zero stuff back to 0 so you can tell what wasn't filled
        show_added[(rw == 1) & (markers == 0)] = 0
        show_added_L[(rw_L == 1) & (markers == 0)] = 0

        ax[2].imshow(show_added[crop], vmin=0, vmax=3)
        ax[2].axis('off')
        ax[2].set_title('rw')
        ax[3].imshow(show_added_L[crop], vmin=0, vmax=3)
        ax[3].axis('off')
        ax[3].set_title('rw-loose')
        fig.tight_layout()
        fig.subplots_adjust(hspace=0.00, wspace=0.01)

        if output_dir is not None:
            fig.savefig(f'./output/{output_dir}/{basename}_{n:{0}2}.png')
        if INTERACTIVE:
            plt.show()

    Vmax, Vargmax = V.max(axis=0), V.argmax(axis=0)
    Vmax = ma.masked_where(Vmax==0, Vmax)
    #Vargmax = ma.masked_where(~trace, Vargmax)

    labs_FA = Vargmax*(Vmax > THRESHOLD).filled(0)
    approx_FA =  labs_FA!=0
    confuse_FA = confusion(approx_FA, trace, bg_mask=ucip_mask)
    # get the smallest label that matched

    labs = np.argmax(W, axis=0) # returns the first index of boolean
    labs =ma.masked_where(labs==0, labs)
    approx = labs.filled(0)!=0
    confuse = confusion(approx, trace, bg_mask=ucip_mask)


    labs_L = np.argmax(WL, axis=0) # returns the first index of boolean
    labs_L =ma.masked_where(labs_L==0, labs_L)
    approx_L = labs_L.filled(0)!=0
    confuse_L = confusion(approx_L, trace, bg_mask=ucip_mask)

    fig, ax = plt.subplots(ncols=4, nrows=2, figsize=(20,12))

    ax[0,0].imshow(cimg[crop])
    ax[0,0].axis('off')
    ax[0,0].set_title(basename)
```

```
143
144     ax[1,0].imshow(ctrace[crop])
145     ax[1,0].axis('off')
146     ax[1,0].set_title('ground truth')
147
148     ax[0,1].imshow(Vmax[crop], cmap=cm)
149     ax[0,1].axis('off')
150     ax[0,1].set_title('max($V_\sigma$)')
151
152
153     ax[0,2].imshow(labs[crop], cmap='magma')
154     ax[0,2].axis('off')
155     ax[0,2].set_title('segmentation (rw)')
156
157     ax[0,3].imshow(labs_L[crop], cmap='magma')
158     ax[0,3].axis('off')
159     ax[0,3].set_title('segmentation (rw-loose)')
160
161     precision_score = lambda t: int(t[0]) / int(t[0] + t[2])
162
163     m, counts = mcc(approx, trace, bg_mask=ucip_mask, return_counts=True)
164     m_L, counts_L = mcc(approx_L, trace, bg_mask=ucip_mask, return_counts=True)
165     m_FA, counts_FA = mcc(approx_FA, trace, bg_mask=ucip_mask,
166                           return_counts=True)
167
168     p = precision_score(counts)
169     p_L = precision_score(counts_L)
170     p_FA = precision_score(counts_FA)
171
172     ax[1,1].imshow(confuse_FA[crop])
173     ax[1,1].axis('off')
174     ax[1,1].set_title(rf'   fixed $\alpha = THRESHOLD$', loc='left')
175     ax[1,1].set_title(f'MCC: {m_FA:.2f}\n'
176                       f'precision: {p_FA:.2%}', loc='right')
177     ax[1,2].imshow(confuse[crop])
178     ax[1,2].axis('off')
179     ax[1,2].set_title('   scalewise-RW', loc='left')
180     ax[1,2].set_title(f'MCC: {m:.2f}\n'
181                       f'precision: {p:.2%}', loc='right')
182
183     ax[1,3].imshow(confuse_L[crop])
184     ax[1,3].axis('off')
185     ax[1,3].set_title('   scalewise-RW (loose)', loc='left')
186     ax[1,3].set_title(f'MCC: {m_L:.2f}\n'
187                       f'precision: {p_L:.2%}', loc='right')
188
189     fig.tight_layout()
190     fig.subplots_adjust(hspace=0.05, wspace=0.01)
191
192     if output_dir is not None:
193         fig.savefig(f'./output/{output_dir}/{basename}_m.png')
194
195     return (filename, m_FA, p_FA, m, p, m_L, p_L)
196
197
198
199 INTERACTIVE = True
200 filenames = list_by_quality(1)
201 filenames.extend(list_by_quality(2))
202 filenames.extend(list_by_quality(3))
203 filenames.extend(list_by_quality(0))
204 RW_BETA = 10
205 THRESHOLD = .4
206
```

```
run_data = list()

for N, filename in enumerate(filenames):


    basename = filename.strip('T-').rstrip('.png')
    print('running rw_demo on', basename, f'({N+1} of {len(filenames)})')
    row = rw_demo(filename, RW_BETA, THRESHOLD, None)
    run_data.append(row)

    if INTERACTIVE:

        plt.show()

    #print(row)

    # do this incrementally; i'm afraid
    if (N % 25 == 0) and (N > 0):
        print('backing up data!')
        with open(f'rw_demo_scores_1206{N//25}.json', 'w') as f:
            json.dump(run_data, f, indent=True)

with open(f'rw_demo_scores_all_1206.json', 'w') as f:
        json.dump(run_data, f, indent=True)
```

listings/rwpf_demo.py

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

import numpy as np
import numpy.ma as ma

import matplotlib.pyplot as plt
from skimage.util import img_as_float
from skimage.io import imread
from placenta import (get_named_placenta, list_by_quality, cropped_args,
                      mimg_as_float, open_typefile, open_tracefile,
                      measure_ncs_markings, add_ucip_to_mask)

from frangi import frangi_from_image
from hfft import fft_gradient, fft_hessian, fft_gaussian
from merging import nz_percentile, apply_threshold
from plate_morphology import dilate_boundary
import os.path, os
from scoring import confusion, mcc

import matplotlib as mpl

from skimage.segmentation import random_walker

import json



INTERACTIVE = True
filenames = list_by_quality(1)
filenames.extend(list_by_quality(2))
filenames.extend(list_by_quality(3))
filenames.extend(list_by_quality(0))
RW_BETA = 10
P_THRESHOLD = 95

```

```
run_data = list()

for N, filename in enumerate(filenames):
    basename = filename.strip('T-').rstrip('.png')
    print('running rwpf_demo on', basename, f'({N+1} of {len(filenames)})')

    # ideally this would be a class with all of these
    cimg = open_typefile(filename, 'raw')
    ctrace = open_typefile(filename, 'ctrace')
    trace = open_tracefile(filename)
    img = get_named_placenta(filename)
    crop = cropped_args(img)
    ucip = open_typefile(filename, 'ucip')

    # make the size of figures more consistent
    if img[crop].shape[0] > img[crop].shape[1]:
        # and rotating it would be fix all this automatically
        cimg = np.rot90(cimg)
        ctrace = np.rot90(ctrace)
        trace = np.rot90(trace)
        img = np.rot90(img)
        ucip = np.rot90(ucip)
        crop = cropped_args(img)

    ucip_midpoint, _ = measure_ncs_markings(ucip)
    ucip_mask = add_ucip_to_mask(ucip_midpoint, radius=60, mask=img.mask)

    plt.close('all')

    cm = mpl.cm.plasma
    cmscales = mpl.cm.magma
    cm.set_bad('k', 1)  # masked areas are black, not white
    cmscales.set_bad('w', 1)

    scales =np.logspace(-1.5, 3.5, num=12, base=2)

    W = np.zeros((len(scales), *img.shape), dtype=np.bool)
    WL = np.zeros((len(scales), *img.shape), dtype=np.bool)
    V = np.zeros((len(scales), *img.shape))
    p_alphas = np.zeros_like(scales)

    #fig, ax = plt.subplots(ncols=3, nrows=len(scales), figsize=(10,10))

    for n, sigma in enumerate(scales):

        f = frangi_from_image(img, sigma, dark_bg=False, dilation_radius=20,
                              beta=0.35)
        V[n] = f

        f[f == 0] = ma.masked

        fig, ax = plt.subplots(ncols=4, nrows=1, figsize=(20,6))

        ax[0].imshow(f[crop], cmap=cm)
        ax[0].axis('off')
        ax[0].set_title(rf'Frangi score, ($\sigma_n = sigma:.3f$)')
        p = nz_percentile(f, P_THRESHOLD)
        p_alphas[n] = p
        hi_marks = (f > p).filled(0)
        markers = np.zeros(img.shape, np.int32)
        markers[f.mask] = 1
        markers[hi_marks] = 2

        ax[1].imshow(markers[crop], cmap=plt.cm.viridis, vmin=0, vmax=3)
```

```
        ax[1].axis('off')
        ax[1].set_title('markers')

        rw = random_walker(1-f.filled(0), markers, beta=RW_BETA)
        rw_L = random_walker(f.filled(0) > 0, markers, beta=RW_BETA)
        W[n] = (rw == 2)
        WL[n] = (rw_L == 2)

        # set the new stuff to a higher number so you can see what was added
        show_added = rw.copy()
        show_added_L = rw_L.copy()
        show_added[~(markers == 2) & (rw==2)] = 3
        show_added_L[~(markers == 2) & (rw_L==2)] = 3
        # set the zero stuff back to 0 so you can tell what wasn't filled
        show_added[(rw == 1) & (markers == 0)] = 0
        show_added_L[(rw_L == 1) & (markers == 0)] = 0

        ax[2].imshow(show_added[crop], vmin=0, vmax=3)
        ax[2].axis('off')
        ax[2].set_title('rw')
        ax[3].imshow(show_added_L[crop], vmin=0, vmax=3)
        ax[3].axis('off')
        ax[3].set_title('rw-loose')
        fig.tight_layout()
        fig.subplots_adjust(hspace=0.00, wspace=0.01)

        fig.savefig(f'./output/RWPFDEMO/{basename}_{n:{0}2}.png')
        if INTERACTIVE:
            plt.show()

    Vmax, Vargmax = V.max(axis=0), V.argmax(axis=0)
    Vmax = ma.masked_where(Vmax==0, Vmax)
    #Vargmax = ma.masked_where(~trace, Vargmax)

    approx_FA, labs_FA = apply_threshold(np.transpose(V, axes=(1,2,0)), p_alphas)
    #labs_FA = Vargmax*(hi_marks)
    #approx_FA =  labs_FA!=0
    confuse_FA = confusion(approx_FA, trace, bg_mask=ucip_mask)
    # get the smallest label that matched

    labs = np.argmax(W, axis=0) # returns the first index of boolean
    labs =ma.masked_where(labs==0, labs)
    approx = labs.filled(0)!=0
    confuse = confusion(approx, trace, bg_mask=ucip_mask)


    labs_L = np.argmax(WL, axis=0) # returns the first index of boolean
    labs_L =ma.masked_where(labs_L==0, labs_L)
    approx_L = labs_L.filled(0)!=0
    confuse_L = confusion(approx_L, trace, bg_mask=ucip_mask)

    fig, ax = plt.subplots(ncols=4, nrows=2, figsize=(20,12))

    ax[0,0].imshow(cimg[crop])
    ax[0,0].axis('off')
    ax[0,0].set_title(basename)

    ax[1,0].imshow(ctrace[crop])
    ax[1,0].axis('off')
    ax[1,0].set_title('ground truth')

    ax[0,1].imshow(Vmax[crop], cmap=cm)
    ax[0,1].axis('off')
    ax[0,1].set_title('max($V_\sigma$)')
```

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

listings/sampled_gauss_problem.py

```
# coding: utf-8
from hfft import discrete_gaussian_kernel
help(discrete_gaussian_kernel)
discrete_gaussian_kernel(100,1)
import matplotlib.pyplot as plt
np.arange(-25,25)
import numpy as np
np.linspace(-1,1,n=50)
np.linspace(-1,1,num=50)
dom = _
plt.plot(dom, discrete_gaussian_kernel(len(dom), 1))
plt.plot(dom, discrete_gaussian_kernel(len(dom), 1))
len(dom)
dgk = discrete_gaussian_kernel
dgk(50)
dgk(50,1)
_.shape
dgk(49,1)
_.shape
dgk(25,1)
dgk(25,1)
dom = np.linspace(-1,1,num=25)
plt.plot(dom,dgk(25,1))
plt.show()
plt.plot(dom,dgk(25,.5))
plt.show()
plt.plot(dom,dgk(25,.1))
plt.show()
plt.plot(dom,dgk(25,.05))
plt.show()
plt.plot(dom,dgk(25,.01))
plt.show()
plt.plot(dom,dgk(25,.001))
plt.show()
plt.plot(dom,dgk(25,.0001))
plt.show()
plt.plot(dom,dgk(25,0))
plt.show()
from scipy.signal import gaussian
get_ipython().run_line_magic('pinfo', 'gaussian')
gaussian(25,.1)
gaussian(25,.1)
plt.plot(dom, gaussian(25,.1))
plt.show()
plt.plot(dom, gaussian(25,.1), dom, dgk(25,.1))
plt.show()
plt.plot(dom, gaussian(25,10), dom, dgk(25,10))
plt.show()
plt.plot(dom, gaussian(25,10), dom, dgk(25,10))
plt.plot(dom, (1/np.sqrt(2*np.pi*10**2))*gaussian(25,10), dom, dgk(25,10))
plt.show()
plt.plot(dom, (1/np.sqrt(2*np.pi*10**2))*gaussian(25,10), dom, dgk(25,10**2))
plt.show()
plt.plot(dom, np.sqrt((1/np.sqrt(2*np.pi*10**2)))*gaussian(25,10), dom, dgk(25,10**2))
plt.show()
plt.plot(dom, np.sqrt((1/np.sqrt(2*np.pi*10**2)))*gaussian(100,10), dom, dgk(100,10**2))
dom = np.linspace(-1,1,num=100)
plt.plot(dom, np.sqrt((1/np.sqrt(2*np.pi*100**2)))*gaussian(100,10), dom, dgk(100,10**2))
plt.plot(dom, np.sqrt((1/np.sqrt(2*np.pi*10**2)))*gaussian(101,10), dom, dgk(101,10**2))
dom = np.linspace(-1,1,num=101)
plt.plot(dom, np.sqrt((1/np.sqrt(2*np.pi*10**2)))*gaussian(101,10), dom, dgk(101,10**2))
plt.show()
plt.plot(dom, np.sqrt((1/np.sqrt(2*np.pi*10**2)))*gaussian(101,10), dom, dgk(101,10**2))
```

```
plt.show()
plt.plot(dom, np.sqrt((1/np.sqrt(2*np.pi*10**2)))*gaussian(101,10), dom, dgk(101,10))
plt.show()
plt.plot(dom, np.sqrt((1/np.sqrt(2*np.pi*10**2)))*gaussian(101,10), dom, dgk(101,10**2))
pt.show()
plt.show()
plt.plot(dom, np.sqrt((1/np.sqrt(2*np.pi*10**2)))*gaussian(101,10), dom, dgk(101,10))
plt.show()
dgk(5,1)
dgk(15,1)
dom = np.arange(15)
plt.imshow(dom, dgk(len(dom),1))
plt.imshow(dom, dgk(dom.size,1))
plt.imshow(dom, dgk(dom.size,1))
plt.imshow(dom, dgk(dom.size,1))
plt.imshow(dom, dgk(dom.size,1))
plt.imshow(dom, dgk(dom.size,1.))
dom = np.linspace(15)
dom = np.linspace(0,1,15)
dom
plt.imshow(dom, dgk(dom.size,1.))
plt.imshow(dom, dgk(15,1.))
get_ipython().run_line_magic('clear', '')
plt.imshow(dom, dgk(15,1.))
plt.plot(dom, dgk(15,1.))
plt.show()
plt.plot(dom, dgk(15,1.))
plt.show()
plt.plot(dom, dgk(15,.05))
plt.show()
plt.plot(dom, dgk(15,0))
plt.show()
plt.plot(dgk(100,0))
plt.show()
plt.scatter(dgk(100,0))
get_ipython().run_line_magic('pinfo', 'plt.scatter')
plt.plot(dgk(100,0), 'o')
plt.show()
plt.plot(dgk(100,0), 'o', markersize=1)
pt.show()
plt.show()
plt.plot(dgk(100,1), 'o', markersize=1)
plt.show()
plt.plot(dgk(100,15), 'o', markersize=1)
plt.show()
gaussian(100,15)
gaussian(100,15) / (15*np.sqrt(2*np.pi))
plot(gaussian(100,15) / (15*np.sqrt(2*np.pi)), 'or', markersize=1)
plt.plot(gaussian(100,15) / (15*np.sqrt(2*np.pi)), 'or', markersize=1)
plt.plot(dgk(100,15), 'o', markersize=1)
plt.show()
plt.plot(gaussian(105,15) / (15*np.sqrt(2*np.pi)), 'or', markersize=1)
plt.plot(dgk(105,15), 'o', markersize=1)
plt.show()
plt.plot(dgk(105,5), 'o-', markersize=1)
plt.plot(gaussian(105,5) / (5*np.sqrt(2*np.pi)), 'or', markersize=1)
plt.show()
plt.plot(gaussian(105,2) / (2*np.sqrt(2*np.pi)), 'or', markersize=1)
plt.plot(dgk(105,2), 'o-', markersize=1)
plt.show()
plt.plot(dgk(105,1.2), 'o-', markersize=1)
plt.show()
sigma = 2; plt.plot(gaussian(105,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=1); pl
plt.show()
```

```
sigma = 2; plt.plot(gaussian(105,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2); pl
plt.show()
sigma = 1.2; plt.plot(gaussian(105,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);
plt.show()
sigma = 1.2; plt.plot(gaussian(105,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);
plt.show()
sigma = .8; plt.plot(gaussian(105,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2); p
plt.show()
sigma = .3; plt.plot(gaussian(105,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2); p
plt.show()
plt.show()
sigma = .01; plt.plot(gaussian(105,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);
plt.show()
sigma = .01; plt.plot(gaussian(105,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);
plt.show()
sigma = .10; plt.plot(gaussian(105,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);
plt.show()
sigma = .10; plt.plot(gaussian(100,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);
plt.show()
sigma = .20; plt.plot(gaussian(100,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);
plt.show()
sigma = .20; plt.plot(gaussian(100,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);
plt.show()
sigma = .20; plt.plot(gaussian(100,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);
plt.show()
np.arange(-(n_samples//2), (n_samples//2) + 1)
N=100; np.arange(-(N//2), (N//2) + 1)
N=100; np.arange(-N//2, (N//2) + 1)
N=100; np.arange((-N)//2, (N//2) + 1)
N=100; np.arange(-(N+1)//2, (N//2) + 1)
N=100; np.arange(-(N-1)//2, (N//2) + 1)
_.shape
N=100; np.arange(-(N-1)//2, (N//2))
N=100; np.arange(-(N+1)//2, (N//2))
N=100; np.arange(-(N)//2-1, (N//2))
N=100; np.arange(-((N)//2-1), (N//2))
N=100; np.arange(-((N)//2-1), (N//2)+1)
_.shape
N=101; np.arange(-((N)//2-1), (N//2)+1)
_.shape
from importlib import reload
reload(hfft)
import hfft
reload(hfft)
from hfft import discrete_gaussian_kernel as dgk
N=101; np.arange(-((N)//2-1), (N//2)+1)
dgk(50,.1)
_.shape
sigma = .20; plt.plot(gaussian(100,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);
plt.show()
dgk(100,sigma).shape
gaussian(100,sigma).shape
dgk(100,sigma)
_.max()
gaussian(100,sigma).max()
gaussian(100,sigma) / (sigma*np.sqrt(2*np.pi))
_.max()
sigma = .20; plt.plot(gaussian(100,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);
plt.show()
sigma = .20; plt.plot(gaussian(100,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);
plt.show()
sigma = .20; plt.plot(gaussian(100,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);
plt.show()
sigma = .20; plt.plot(gaussian(100,sigma) / (sigma*np.sqrt(2*np.pi)), 'or', markersize=2);
```

```
plt.show()
```

listings/scaledecay.py

```
# coding: utf-8
from placenta import get_named_placenta, list_by_quality
list_by_quality(0)
filename = _[-2]
img = get_named_placenta(filename)
import matplotlib.pyplot as plt
import numpy as np
plt.imshow(img)
pls.show()
plt.show()
filename =  list_by_quality(0)[0]
img = get_named_placenta(filename)
from placenta import cropped_args
crop = cropped_args(img)
img[crop]
plt.imshow(img[crop])
plt.show()
from hfft import fft_gaussian
get_ipython().run_line_magic('pinfo', 'fft_gaussian')
C = fft_gaussian(img, 32, 'discrete')
B = fft_gaussian(img, 5, 'discrete')
A = fft_gaussian(img, .12, 'discrete')
plt.imshow(A)
plt.show()
A = fft_gaussian(img, .25, 'discrete')
plt.imshow(A)
plt.show()
plt.imshow(B)
plt.show()
plt.imshow(C)
plt.show()
gA = np.gradient(A)
gA
gB = np.gradient(B)
gC = np.gradient(C)
gA.shape
gA[0].shape
aa = lambda g: np.sqrt((1+g[0]*g[0] + g[0]*g[0]))
plt.imshow(aa(gA))
plt.show()
from plate_morphology import dilate_boundary
from functools import partial
dilate = partial(dilate_boundary, mask=img.mask)
dilate(aa(gA), 20)
plt.imshow(_)
plt.show()
dilate(aa(gA), 20).filled(0)
plt.show()
plt.imshow(_)
plt.show()
Aaa = dilate(aa(gA), 20).filled(0)
Aaa.max()
Aaa.min()
Aaa[~img.mask].min()
Aaa[img.mask].min()
Aaa[img.mask].max()
Aaa[~img.mask].max()
Baa = dilate(aa(gB), 20).filled(0)
Caa = dilate(aa(gC), 20).filled(0)
```

```
plt.imshow(Baa)
plt.show()
plt.imshow(Caa)
plt.show()
Caa.min()
Caa[~img.mask].min()
Caa == 0
plt.imshow(_)
plt.show()
Caa[~img.mask].max()
Caa[~img.mask].min()
Caa[~img.mask].argmin()
dilate_boundary(img.mask, 20)
dilate_boundary(img.mask, 20, mask_only=True)
get_ipython().run_line_magic('pinfo', 'dilate_boundary')
dilate_boundary(img, radius=20)
dil = _.mask.copy()
dil
plt.imshow(dil)
plt.show()
Caa[~dil].argmin()
Caa[~dil]
Caa[~dil].min()
Caa[~dil].max()
Baa[~dil].max()
Baa[~dil].min()
Aaa[~dil].min()
Aaa[~dil].max()
plt.imshow(Caa)
plt.show()
plt.imshow(Aaa)
plt.show()
#for sigma in np.logspace(-3, 8, base=2, num=20):
#    D = fft_gaussian(img, sigma, 'discrete')
#    gD = np.gradient(D)
#    Daa = dilate(aa(dG), 20).filled(1)
#    print(Daa[~dil].min(), Daa[~dil].max()
aas = list()
for sigma in np.logspace(-3, 8, base=2, num=20):
    D = fft_gaussian(img, sigma, 'discrete')
    gD = np.gradient(D)
    Daa = dilate(aa(dG), 20).filled(1)
    aas.append(Daa)
    print(f"sigma={sigma:.3f}", "min: {:6f}, max:{:6f}".format(
          Daa[~dil].min(), Daa[~dil].max())
          )

for sigma in np.logspace(-3, 8, base=2, num=20):
    D = fft_gaussian(img, sigma, 'discrete')
    gD = np.gradient(D)
    Daa = dilate(aa(gD), 20).filled(1)
    aas.append(Daa)
    print(f"sigma={sigma:.3f}", "min: {:6f}, max:{:6f}".format(
          Daa[~dil].min(), Daa[~dil].max())
          )

for sigma in np.logspace(-4, 8, base=2, num=50):
    D = fft_gaussian(img, sigma, 'discrete')
    gD = np.gradient(D)
    Daa = dilate(aa(gD), 20).filled(1)
    aas.append(Daa)
    print(f"sigma={sigma:.3f}", "min: {:6f}, max:{:6f}".format(
          Daa[~dil].min(), Daa[~dil].max())
          )
```

```
124
125 bb lambda g: np.linalg.norm(np.array(
126         [[1+g[1]**2, -g[0]*g[1]],
127          [-g[0]*g[1], 1 + g[0]**2]]))
128 bb = lambda g: np.linalg.norm(np.array(
129         [[1+g[1]**2, -g[0]*g[1]],
130          [-g[0]*g[1], 1 + g[0]**2]]))
131 bb(gA)
132 get_ipython().set_next_input('man np.linalg.norm');get_ipython().run_line_magic('pinfo', 'n
133 ginv = lambda g: np.array(
134         [[1+g[1]**2, -g[0]*g[1]],
135          [-g[0]*g[1], 1 + g[0]**2]])
136 ginv(gA)
137 _.shape
138 G[:,:,34,289]
139 ginvA = _101
140 ginvA[:,:,340,289]
141 bb = lambda g: np.sqrt((1+g[1]**2)**2 + 2*(g[0]*g[1])**2 + (1+g[0]**2)**2)
142 bb(gA)
143 _.shape
144 plt.imshow(bb)
145 _.shape
146 bb(gA)
147 plt.imshow(_)
148 plt.show()
149 bb(gA) / aa(gA)
150 plt.imshow(_)
151 plt.show()
152 dilate(bb(gA) / aa(gA))
153 plt.imshow(dilate(bb(gA) / aa(gA)).filled(0))
154 plt.show()
155 bb = lambda g: np.sqrt((1+g[1]**2)**2 + 2*(g[0]*g[1])**2 + (1+g[0]**2)**2)
156 plt.imshow(dilate(bb(gA),20))
157 plt.show()
158 bb(gA)[~dil].min()
159 bb(gA)[~dil].max()
160 (bb(gA) / aa(gA))[~dil].min()
161 (bb(gA) / aa(gA))[~dil].max()
162 (bb(gA) / aa(gA))
163 plt.imshow(dilate(_,20).filled(1.414))
164 plt.show()
165 for sigma in np.logspace(-4, 8, base=2, num=50):
166     D = fft_gaussian(img, sigma, 'discrete')
167     gD = np.gradient(D)
168     Daa, Dbb = aa(gD), bb(gD)
169     Dcc = Daa / Dbb
170     aas = Daa[~dil].min(), Daa[~dil].max()
171     bbs = Dbb[~dil].min(), Dbb[~dil].max()
172     ccs = Dcc[~dil].min(), Dcc[~dil].max()
173     print(f"sigma={sigma:.3f}",
174           "\tmin: {:.6f},\tmax:{:.6f}".format(aas),
175           "\tmin: {:.6f},\tmax:{:.6f}".format(bbs),
176           "\tmin: {:.6f},\tmax:{:.6f}".format(ccs), sep='\n')
177
178
179 for sigma in np.logspace(-4, 8, base=2, num=50):
180     D = fft_gaussian(img, sigma, 'discrete')
181     gD = np.gradient(D)
182     Daa, Dbb = aa(gD), bb(gD)
183     Dcc = Daa / Dbb
184     aas = Daa[~dil].min(), Daa[~dil].max()
185     bbs = Dbb[~dil].min(), Dbb[~dil].max()
186     ccs = Dcc[~dil].min(), Dcc[~dil].max()
187     print(f"sigma={sigma:.3f}",
```

```
            "\tmin: {:.6f},\tmax:{:.6f}".format(*aas),
            "\tmin: {:.6f},\tmax:{:.6f}".format(*bbs),
            "\tmin: {:.6f},\tmax:{:.6f}".format(*ccs), sep='\n')



for sigma in np.logspace(-4, 8, base=2, num=50):
    D = fft_gaussian(img, sigma, 'discrete')
    gD = np.gradient(D)
    Daa, Dbb = aa(gD), bb(gD)
    Dcc = Dbb / Daa
    aas = Daa[~dil].min(), Daa[~dil].max()
    bbs = Dbb[~dil].min(), Dbb[~dil].max()
    ccs = Dcc[~dil].min(), Dcc[~dil].max()
    print(f"sigma={sigma:.3f}",
            "\tmin: {:.6f},\tmax:{:.6f}".format(*aas),
            "\tmin: {:.6f},\tmax:{:.6f}".format(*bbs),
            "\tmin: {:.6f},\tmax:{:.6f}".format(*ccs), sep='\n')

data = _
data = list()
for sigma in np.logspace(-4, 8, base=2, num=50):
    D = fft_gaussian(img, sigma, 'discrete')
    gD = np.gradient(D)
    Daa, Dbb = aa(gD), bb(gD)
    Dcc = Dbb / Daa
    aas = Daa[~dil].min(), Daa[~dil].max()
    bbs = Dbb[~dil].min(), Dbb[~dil].max()
    ccs = Dcc[~dil].min(), Dcc[~dil].max()
    print(f"sigma={sigma:.3f}",
            "\tmin: {:.6f},\tmax:{:.6f}".format(*aas),
            "\tmin: {:.6f},\tmax:{:.6f}".format(*bbs),
            "\tmin: {:.6f},\tmax:{:.6f}".format(*ccs), sep='\n')
    data.append(
        [sigma, *aas, *bbs, *ccs])

import pandas
table = pandas.DataFrame(data)
print(table)
from hfft import fft_hessian
get_ipython().run_line_magic('pinfo', 'fft_hessian')
helems = fft_hessian(img, sigma=1, kernel='discrete')
np.sqrt(helems[0]**2 + 2*helems[1]**2 + helems[2]**2)
_.shape
plt.imshow(np.sqrt(helems[0]**2 + 2*helems[1]**2 + helems[2]**2))
plt.show()
data = list()
for sigma in np.logspace(-4, 8, base=2, num=50):
    D = fft_gaussian(img, sigma, 'discrete')
    gD = np.gradient(D)
    Daa, Dbb = aa(gD), bb(gD)
    Dcc = Dbb / Daa
    h = fft_hessian(img,sigma, 'discrete')
    hnorm = np.sqrt(h[0]**2 + 2*h[1]**2 + h[2]**2)
    Lnorm = hnorm*Dcc
    aas = Daa[~dil].min(), Daa[~dil].max()
    bbs = Dbb[~dil].min(), Dbb[~dil].max()
    ccs = Dcc[~dil].min(), Dcc[~dil].max()
    dds = hnorm[~dil].min(), hnorm[~dil].max()
    lls = Lnorm[~dil].min(), Lnorm[~dil].max()
    print(f"sigma={sigma:.3f}",
            "\tmin: {:.6f},\tmax:{:.6f}".format(*aas),
            "\tmin: {:.6f},\tmax:{:.6f}".format(*bbs),
            "\tmin: {:.6f},\tmax:{:.6f}".format(*ccs),
```

```
            "\tmin: {:.6f},\tmax:{:.6f}".format(*dds),
            "\tmin: {:.6f},\tmax:{:.6f}".format(*lls), sep='\n')
    data.append(
        [sigma, *aas, *bbs, *ccs, *dds, *lls])


data
table = pandas.DataFrame(data)
table
table.columns
help(table.columns)
plt.imshow(Lnorm)
plt.show()
Ls = list()
for sigma in np.logspace(-4, 8, base=2, num=50):
    D = fft_gaussian(img, sigma, 'discrete')
    gD = np.gradient(D)
    Daa, Dbb = aa(gD), bb(gD)
    Dcc = Dbb / Daa
    h = fft_hessian(img,sigma, 'discrete')
    hnorm = np.sqrt(h[0]**2 + 2*h[1]**2 + h[2]**2)
    Lnorm = hnorm*Dcc
    Ls.append(Lnorm)
    aas = Daa[~dil].min(), Daa[~dil].max()
    bbs = Dbb[~dil].min(), Dbb[~dil].max()
    ccs = Dcc[~dil].min(), Dcc[~dil].max()
    dds = hnorm[~dil].min(), hnorm[~dil].max()
    lls = Lnorm[~dil].min(), Lnorm[~dil].max()
    print(f"sigma={sigma:.3f}",
            "\tmin: {:.6f},\tmax:{:.6f}".format(*aas),
            "\tmin: {:.6f},\tmax:{:.6f}".format(*bbs),
            "\tmin: {:.6f},\tmax:{:.6f}".format(*ccs),
            "\tmin: {:.6f},\tmax:{:.6f}".format(*dds),
            "\tmin: {:.6f},\tmax:{:.6f}".format(*lls), sep='\n')
    #data.append(
    #    [sigma, *aas, *bbs, *ccs, *dds, *lls])

L[0]
Ls[0]
plt.imshow(_)
plt.show()
for Lnorm, sigma in zip(Ls, np.logspace(-4,8, base=2,num=50)):
    plt.imshow(Lnorm[crop], cmap='nipy_spectral')
    mng = plt.get_current_fig_manager()
    mng.window.showMaximized()
    plt.colorbar()
    plt.title(r'Lnorm $\sigma=:.3f$'.format(sigma))
    plt.axis('off')
    plt.tight_layout()
    plt.show()
    plt.close('all')

for Lnorm, sigma in zip(Ls, np.logspace(-4,8, base=2,num=50)):
    L = dilate(Lnorm, min(20,int(sigma))).filled(0)
    plt.imshow(Lnorm[crop], cmap='nipy_spectral')
    mng = plt.get_current_fig_manager()
    mng.window.showMaximized()
    plt.colorbar()
    plt.title(r'Lnorm $\sigma=:.3f$'.format(sigma))
    plt.axis('off')
    plt.tight_layout()
    plt.show()
    plt.close('all')

```

```python
for Lnorm, sigma in zip(Ls, np.logspace(-4,8, base=2,num=50)):
    L = dilate(Lnorm, min(20,int(sigma))).filled(0)
    plt.imshow(L[crop], cmap='nipy_spectral')
    mng = plt.get_current_fig_manager()
    mng.window.showMaximized()
    plt.colorbar()
    plt.title(r'Lnorm σ = :.3f'.format(sigma))
    plt.axis('off')
    plt.tight_layout()
    plt.show()
    plt.close('all')

for Lnorm, sigma in zip(Ls, np.logspace(-4,8, base=2,num=50)):
    L = dilate(Lnorm, max(20,int(sigma))).filled(0)
    plt.imshow(L[crop], cmap='nipy_spectral')
    mng = plt.get_current_fig_manager()
    mng.window.showMaximized()
    plt.colorbar()
    plt.title(r'Lnorm σ = :.3f'.format(sigma))
    plt.axis('off')
    plt.tight_layout()
    plt.show()
    plt.close('all')

for Lnorm, sigma in zip(Ls, np.logspace(-4,8, base=2,num=50)):
    L = dilate(Lnorm, max(20,int(sigma))).filled(0)
    plt.imshow(L[crop], cmap='nipy_spectral')
    mng = plt.get_current_fig_manager()
    mng.window.showMaximized()
    plt.colorbar()
    plt.title(r'Lnorm σ = :.3f'.format(sigma))
    plt.axis('off')
    plt.tight_layout()
    plt.show()
    plt.close('all')

for Lnorm, sigma in zip(Ls, np.logspace(-4,8, base=2,num=50)):
    L = dilate(Lnorm, max(20,int(2*sigma))).filled(0)
    plt.imshow(L[crop], cmap='nipy_spectral')
    mng = plt.get_current_fig_manager()
    mng.window.showMaximized()
    plt.colorbar()
    plt.title(r'Lnorm σ = :.3f'.format(sigma))
    plt.axis('off')
    plt.tight_layout()
    plt.show()
    plt.close('all')

print(table)
table
table + 2
table[:,0]
table[0]
table / table[0]
T = np.array(table)
T
T.shape
T.dtype
T / T[:,0]
T / T[...,0]
T[...,-1] / T[...,0]
T[...,-1] / np.sqrt(T[...,0])
T[...,-1] * np.sqrt(T[...,0])
T[...,-1] * T[...,0]
```

```
T[...,-1] * T[...,0]**2
table
T[...,-3] * T[...,0]**2
T[...,-3] * T[...,0]
T[...,-4] * T[...,0]
T[...,-4] / T[...,0]
T[...,-4] / T[...,0] > .01
which = _
T[...,0][which]
(T[...,-4] / T[...,0]) > .001
scales
scales = np.logspace(-4,8,num=50,base=2)
scales
(T[...,-3] / T[...,0]) > .001
(T[...,-3] / T[...,0]) > .005
scales[_]
(T[...,-3] / T[...,0]) > .001
scales
scales
(T[...,-3] / T[...,0]) > .001
scales[_]
(T[...,-3] / T[...,0]) > .05
T[...,-3]
(T[...,-3] / T[...,0])
(T[...,-3] / T[...,0]**2)
(T[...,-3] *np.sqrt(1/T[...,0]))
(T[...,-4] / T[...,0]) > .05
table
from frangi import frangi_from_image
g[0]g[1]
g[0]*g[1]
gA[0]*gA[1]
plt.imshow(_)
plt.show()
```

listings/scale_sweep_demo.py

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

from placenta import (get_named_placenta, list_placentas, _cropped_bounds,
                      cropped_view, cropped_args, show_mask)
from frangi import frangi_from_image

import numpy as np
import numpy.ma as ma

from plate_morphology import dilate_boundary

import os.path
import matplotlib.pyplot as plt
import matplotlib as mpl
from itertools import product

# pick two samples and two insets (should be the same size)
demo1 = 'BN2315363', np.s_[370:670, 530:930]
demo2 = 'BN5280796', np.s_[150:450, 530:930]

make_individual = False

for sample_name, inset_slice in (demo1, demo2):
    img = get_named_placenta(f'T-{sample_name}.png')

    crop = cropped_args(img)
```

```
27
28      F, fi = list(), list() # make some empty lists to store for inspection
29
30      scales = [0.2, 0.8, 1.0, 2.0, 4.0, 6.0, 8.0, 16.0]
31      CMAP = plt.cm.nipy_spectral
32      cmin, cmax = (0, 0.4)
33
34      for n, sigma in enumerate(scales):
35          R = max(int(sigma*3), 10) # only really necessary for signed
36          target = frangi_from_image(img, sigma, dark_bg=False,
37                                     signed_frangi=False, dilation_radius=R)
38          plate = target[crop].filled(0)
39          inset = target[inset_slice].filled(0)
40          F.append(plate)
41          fi.append(inset)
42
43          if not make_individual:
44              continue
45
46          # else this might be nice for the actual thesis defense
47          for label in ['plate', 'inset']:
48              if label == 'inset':
49                  printable = inset
50              else:
51                  printable = plate
52
53              plt.imshow(printable, cmap=CMAP)
54              plt.title(r'$\sigma = :.2f$'.format(sigma))
55              plt.tight_layout()
56              c = plt.colorbar()
57              c.set_ticks = np.linspace(cmin, cmax, num=len(scales)+1)
58              plt.clim(cmin, cmax)
59              plt.axis('off')
60              outname = f'demo_output/scalesweep_{sample_name}_{label}_{n}.png'
61              plt.savefig(outname, dpi=300, bbox_inches='tight')
62              print('saved', outname)
63              plt.close()
64      # now make a stitched together version
65      for label in ['plate', 'inset']:
66          if label == 'inset':
67              L = fi
68              imgview = img[inset_slice].filled(0)
69              figsize = (12, 6)
70
71          else:
72              L = F
73              imgview = img[crop].filled(0)
74              figsize = (12, 9)
75          #adjust this manually depending on how many scales you end up using!
76
77          nrows, ncols = 2, 4
78          fig, axes = plt.subplots(nrows=nrows, ncols=ncols, figsize=figsize)
79
80          for n, (i, j) in enumerate(product(range(nrows), range(ncols))):
81
82              if n == 0:
83                  axes[i,j].imshow(imgview, cmap=plt.cm.gray)
84                  axes[i,j].set_title('raw')
85              else:
86                  im = axes[i,j].imshow(L[n], cmap=CMAP, vmin=cmin, vmax=cmax)
87                  axes[i,j].set_title(r'$\sigma = :.2f$'.format(scales[n]))
88
89              plt.setp(axes[i,j].get_xticklabels(), visible=False)
90              plt.setp(axes[i,j].get_yticklabels(), visible=False)
```

```

        fig.subplots_adjust(top=0.954, bottom=0.025, left=0.010,
                            right=0.989, hspace=0.0, wspace=0.0)

        plt.savefig(f'demo_output/scalesweep_stitch_{sample_name}_{label}.png',
                    dpi=300)

        cfile = 'demo_output/scalesweep_colorbar.png'
        if os.path.isfile(cfile):
            continue # no need to make another

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

listings/scoring.py

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

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

import matplotlib as mpl
import matplotlib.pyplot as plt

import itertools
from collections import deque

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

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

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

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

    Notes:

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

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

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

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

    for pix, color in TRACE_COLORS.items():

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


    return W

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

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

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

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

    Notes
    -----
    Since arteries grow over the veins on the PCSVN and are generally easier
```

```
    to extract, it might be preferable to indicate "arteries". In reality,
    each strategy is a compromise, and only by keeping track of both would
    you get the complete picture.

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

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

    # collisions (where are widths both reported)
    c = (A!=0)& (V!=0)

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

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

    return W

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

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

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

    Inputs:

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

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

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

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

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

    else:
        # use numpy.isin(T, widths) but that's only in version 1.13 and up
```

```
        # of numpy this is basically the code for that though
        to_keep = np.in1d(W, widths, assume_unique=True).reshape(W.shape)
        Wout[~to_keep] = 0
    return Wout


TRACE_COLORS = {
    3: (255, 0, 111),
    5: (168, 0, 0),
    7: (168, 0, 255),
    9: (255, 0, 255),
    11: (0, 138, 255),
    13: (138, 255, 0),
    15: (255, 200, 0),
    17: (255, 138, 0),
    19: (255, 0, 21)
}


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

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

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

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

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

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

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


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

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


def skeletonize_trace(T, T2=None):
    """
    if T is a boolean matrix representing a trace, then thin it

    if T is an RGB trace, then register it according to the
    tracing protocol then thin it
```

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

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

    if colordict is None:
#         colordict = {
#             'TN': (247, 247, 247),  # true negative# 'f7f7f7'
#             'TP': (0, 0, 0),  # true positive  # '000000'
#             'FN': (241, 163, 64),  # false negative # 'f1a340' orange
#             'FP': (153, 142, 195),  # false positive # '998ec4' purple
#             'mask': (247, 200, 200)  # mask color (not used in MCC calculation)
#             }
#
        #colordict = {
        #    'TN': (49,49,49),  # true negative# 'f7f7f7'
        #    'TP': (0, 0, 0),  # true positive  # '000000'
        #    'FN': (201,53,108),  # false negative # 'f1a340' orange
        #    'FP': (0,112,163),  # false positive # '998ec4' purple
        #    'mask': (247, 200, 200)  # mask color (not used in MCC calculation)
        #    }

        colordict = {
                     'TP': (0,0,0),
                     'TN': (226,226,226),
                     'FN': (201,152,152),
                     'FP': (30,69,230),
                     'mask': (209,209,209)
                     }
    #TODO: else check if mask is specified and add it as color of TN otherwise
    if truth is None:
        truth = np.zeros_like(test)

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

    assert test.shape == truth.shape

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

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

    # truth conditions
    true_pos = (test==truth & truth)
    true_neg = (test==truth & ~truth)
    false_neg = (truth & ~test)
    false_pos = (test & ~truth)

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

    # try to find a mask
    if bg_mask is None:
        try:
```

```
            bg_mask =  test.mask
        except AttributeError:
            # no mask is specified, we're done.
            return output

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

    return output

def binary_counts(test, truth, bg_mask=None, score_bg=False):
    """returns TP,TN,FP,FN"""

    true_pos = ((test == truth) & truth)
    true_neg = ((test == truth) & ~truth)
    false_neg = (truth & ~test)
    false_pos = (test & ~truth)

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

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

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

    return TP, TN, FP, FN

def compare_trace(approx, trace=None, filename=None,
                  sample_dir=None, colordict=None):
    """
    compare approx matrix to trace matrix and output a confusion matrix.
    if trace is not supplied, open the image from the tracefile.
    if tracefile is not supplied, filename must be supplied, and
    tracefile will be opened according to the standard pattern

    colordict are parameters to pass to confusion()

    returns a matrix
    """

    # load the tracefile if not supplied
    if trace is None:
        if filename is not None:
```

```
            try:
                trace = open_typefile(filename, 'trace')
            except FileNotFoundError:
                print("No trace file found matching ", filename)
                print("no trace found. generating dummy trace.")
                trace = np.zeros_like(approx)
        else:
            print("no trace supplied/found. generating dummy trace.")
            trace = np.zeros_like(approx)

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

    return C


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

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

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

    """
    if truth is None:
        truth = np.zeros_like(test)

    TP, TN, FP, FN = binary_counts(test, truth, bg_mask=bg_mask,
                                   score_bg=score_bg)


    denom = np.sqrt(TP+FP)*np.sqrt(TP+FN)*np.sqrt(TN+FP)*np.sqrt(TN+FN)

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

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


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

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

```

```
    output:

        mse:   a single number.
    """

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

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

    return mse

def chain_lengths(iterable):

    pos, s = 0, 0

    for b, g in itertools.groupby(iterable):

        if not b:
            # alternative if the bottom doesn't work or something
            #d = deque(enumerate(g,1), maxlen=1)
            #pos += d[0][0] if d else 0

            pos += sum((1 for i in g if not i))

        else:

            s = sum(g)

            yield pos, s

            pos += s

    if not s:
        # so it will return something even if iterable is empty
        yield 0, 0


def _longest_chain_1d(iterable):
    """ will return a tuple of ind, length
    where ind is the position in the iterable the chain starts and length is the
    length of the chain
    """
    return max(chain_lengths(iterable), key=lambda x: x[1])


def longest_chain(arr, axis):
    """Find where the longest chain of boolean values and occurs across an array
    and also return its length
    """

    C = np.apply_along_axis(_longest_chain_1d, axis, arr.astype('bool'))

    start_inds, chain_lens =  np.split(C, 2, axis)

    return np.squeeze(start_inds), np.squeeze(chain_lens)


def _bunch_hists(H, bunches):

    return np.stack((np.sum(np.atleast_2d(H[b,:]), axis=0) for b in bunches))

```

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

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

```
    but can be something like
    ( (0,1,2,3), (4,5), 6, 7, 8, (9,10,11) )

    or even

    (2,3,4,5,(0,1,6,7))

    this is to prevent similar scales from clogging, you can just bin them
    all together.

    approx method is a label to use in the fig titles
    """

    if bunches is None:
        if bunch_until is not None:
            indices = list(range(len(scales)))
            bunches = [indices[:bunch_until],] + indices[bunch_until:]

    plt.style.use(style)

    fig, ax = plt.subplots(figsize=figsize)

    wbins = np.arange(3,20,2)  # bins of widths in ground truth

    max_hists = np.array([[np.sum((max_labels == s) & (widths==w)) for w in wbins]
                  for s in range(len(scales))])
    if normalize:
        max_hists = max_hists / max_hists.sum(axis=1, keepdims=True)

    if cmap is None:
        # this will just use the default cycle of colors
        colors = np.repeat(None, len(scales))
    else:
        if not isinstance(cmap, mpl.colors.LinearSegmentedColormap):
            # try this
            cmap = plt.get_cmap(cmap)

        colors = cmap(np.linspace(0,1,len(scales)))

    labels = [rf'$\sigma_k = sigma:.2f$'
              for k, sigma in enumerate(scales,1)]

    # number of true positives, false negatives for each width

    if bunches is not None:

        # just return \sigma_{1,2,3} or something rather than listing
        bunch_label = lambda b: r"$\sigma$".format(','.join(( str(x+1)
                                                               for x in b)
                                                             ))

        labels = [labels[b] if np.isscalar(b) else bunch_label(b)
                  for b in bunches]

        if cmap is None:
            # just make it the appropriate length
            cmap = np.repeat(None, len(bunches))
        else:
            colors = [colors[b] if np.isscalar(b) else colors[b[0]]
                      for b in bunches]

    #ax.bar(wbins, tp_hists, color=(0.6,0.6,0.6),
    #        label='# true positives')
    #ax.bar(wbins, fn_hists, bottom=tp_hists, color=(1,.8,.8),
```

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

listings/signed_sweep_demo.py

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

from placenta import (get_named_placenta, list_placentas, _cropped_bounds,
                      cropped_view, cropped_args, show_mask)
from frangi import frangi_from_image

import numpy as np
import numpy.ma as ma

from plate_morphology import dilate_boundary

import os.path
import matplotlib.pyplot as plt
import matplotlib as mpl
from itertools import product

# pick two samples and two insets (should be the same size)
demo1 = 'BN2315363', np.s_[370:670, 530:930]
demo2 = 'BN5280796', np.s_[150:450, 530:930]

for sample_name, inset_slice in (demo1, demo2):
    img = get_named_placenta(f'T-{sample_name}.png')

    crop = cropped_args(img)

    F, fi = list(), list() # make some empty lists to store for inspection

    #scales = np.logspace(-3, 4, num=8, base=2)
    scales = [0.2, 0.8, 1.0, 2.0, 4.0, 6.0, 8.0, 16.0]
    CMAP = plt.cm.Spectral
    cmin, cmax = (-0.4, 0.4)

    for n, sigma in enumerate(scales):
        R = max(int(sigma*3), 10)
        target = frangi_from_image(img, sigma, dark_bg=False,
                                   signed_frangi=True, dilation_radius=R)
        plate = target[crop].filled(0)
        inset = target[inset_slice].filled(0)
        F.append(plate)
        fi.append(inset)
        #for label in ['plate', 'inset']:
        #    if label == 'inset':
        #        printable = inset
        #    else:
        #        printable = plate

        #    plt.imshow(printable, cmap=CMAP)
        #    plt.title(r'$\sigma = :.2f$'.format(sigma))
        #    plt.tight_layout()
        #    c = plt.colorbar()
        #    c.set_ticks = np.linspace(cmin, cmax, num=len(scales)+1)
        #    plt.clim(cmin, cmax)
        #    plt.axis('off')
        #    outname = f'demo_output/signsweep_{sample_name}_{label}_{n}.png'
        #    #plt.savefig(outname, dpi=300, bbox_inches='tight')
        #    print('saved', outname)
        #    plt.close()

    # now make a stitched together version
    for label in ['plate', 'inset']:
        if label == 'inset':
            L = fi
```

```
            imgview = img[inset_slice].filled(0)
            figsize = (12, 6)

        else:
            L = F
            imgview = img[crop].filled(0)
            figsize = (12, 9)
        #adjust this manually depending on how many scales you end up using!

        nrows, ncols = 2, 4
        fig, axes = plt.subplots(nrows=nrows, ncols=ncols, figsize=figsize)

        for n, (i, j) in enumerate(product(range(nrows), range(ncols))):

            if n == 0:
                axes[i,j].imshow(imgview, cmap=plt.cm.gray)
                axes[i,j].set_title('raw')
            else:
                im = axes[i,j].imshow(L[n], cmap=CMAP, vmin=cmin, vmax=cmax)
                axes[i,j].set_title(r'$\sigma =: .2f$'.format(scales[n]))

            plt.setp(axes[i,j].get_xticklabels(), visible=False)
            plt.setp(axes[i,j].get_yticklabels(), visible=False)

        #for i in range(5):
        #    fig.tight_layout()

        #fig.subplots_adjust(right=0.8)
        #cax = fig.add_axes([.85,.15,.05,.7])
        #c = fig.colorbar(im, ax=cax)

        fig.subplots_adjust(top=0.954, bottom=0.025, left=0.010,
                            right=0.989, hspace=0.0, wspace=0.0)

        plt.savefig(f'demo_output/signsweep_stitch_{sample_name}_{label}.png',
                    dpi=300)

        cfile = 'demo_output/signsweep_colorbar.png'
        if os.path.isfile(cfile):
            continue # no need to make another

        fig = plt.figure(figsize=(figsize[0],2))
        ax1 = fig.add_axes([0.15, 0.25, 0.75, 0.5])
        cbar = mpl.colorbar.ColorbarBase(ax1, cmap=CMAP,
                                         norm=mpl.colors.Normalize(cmin,cmax),
                                         orientation='horizontal')
        plt.savefig(cfile, dpi=300)
        #top = np.concatenate(L[:4],axis=1)
        #bottom = np.concatenate(L[4:],axis=1)
        #stitched = np.concatenate((top,bottom),axis=0)
        #imga = plt.imshow(stitched, cmap=CMAP)
        #plt.imsave(f'demo_output/signsweep_stitch_{sample_name}_{label}.png',
        #        stitched, cmap=CMAP, vmin=cmin, vmax=cmax)

        # also save the original pic
        #plt.imsave(f'demo_output/signsweep_{sample_name}_{label}_raw',
        #        imgview, cmap=plt.cm.gray)
```

listings/stump_test.py

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

from placenta import (get_named_placenta, list_placentas, open_typefile,
```

```
                      show_mask, measure_ncs_markings)
import numpy as np
import matplotlib.pyplot as plt
from preprocessing import mask_stump
import numpy.ma as ma
import sys
from skimage.exposure import equalize_adapthist
from scipy.ndimage import label
from skimage.morphology import thin, binary_opening, disk, convex_hull_image


for n, filename in enumerate(list_placentas('T-BN')):

    print(filename)

    raw = open_typefile(filename, 'raw')
    #raw = (equalize_adapthist(raw))
    img = get_named_placenta(filename)

    stump = mask_stump(raw, mask=img.mask, mask_only=True)
    if not stump.any():
        print('no stump found at all, continuing')
        plt.imshow(raw)
        plt.show()
        continue

    stump_labs = label(stump)

    ucip_mid, _ = measure_ncs_markings(filename=filename)

    labeled, n_labs = label(stump)
    dists = list()
    for lab in range(1, n_labs+1):
        X, Y = np.where(labeled==lab)
        dists.append((lab, min([(x-ucip_mid[0])**2 + (y-ucip_mid[1])**2
                        for x,y in zip(X,Y)])))

    closest_lab = sorted(dists, key=lambda v: v[1])[0][0]

    closest_stump = (labeled==closest_lab)
    closest_opening = binary_opening(closest_stump, disk(10))
    final_stump = convex_hull_image(closest_opening)

    fig, ax = plt.subplots(ncols=3, nrows=2, figsize=(30,12))

    ax[0,0].imshow(raw)
    ax[0,0].set_title(filename)

    newmask = show_mask(ma.masked_array(img, mask=stump))
    ax[0,1].imshow(newmask)
    ax[0,2].imshow(stump*1. + img.mask*2, vmin=0, vmax=3)
    ax[1,0].imshow(closest_stump)
    ax[1,1].imshow(closest_opening)
    ax[1,2].imshow(final_stump)

    plt.show()

    if input('make more?') == 'n':
        sys.exit(0)
```

listings/threshold_demo.py

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

import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
from skimage.util import img_as_float
from skimage.io import imread
from placenta import (get_named_placenta, list_placentas, cropped_args)

import numpy.ma as ma
from plate_morphology import dilate_boundary
import os.path, os
from postprocessing import dilate_to_rim
from scoring import confusion, mcc
from placenta import open_tracefile, open_typefile
from preprocessing import inpaint_hybrid
from placenta import measure_ncs_markings, add_ucip_to_mask

import json

mccs = list()
precs = list()

scales = np.logspace(-1.5, 3.5, num=20, base=2)
threshold = .15
neg_threshold = 0.001
max_pos_scale = -6
max_neg_scale = 2
beta = 0.10
gamma = 1.0

cm = mpl.cm.plasma
#cmscales = mpl.cm.magma
cm.set_bad('k', 1)  # masked areas are black, not white
#cmscales.set_bad('w', 1)

QUALITY = 3
for filename in list_by_quality(QUALITY):

    cimg = open_typefile(filename, 'raw')
    ctrace = open_typefile(filename, 'ctrace')
    trace = open_tracefile(filename)
    img = get_named_placenta(filename)
    crop = cropped_args(img)
    ucip = open_typefile(filename, 'ucip')
    img = inpaint_hybrid(img)

    # make the size of figures more consistent
    if img[crop].shape[0] > img[crop].shape[1]:
        # and rotating it would be fix all this automatically
        cimg = np.rot90(cimg)
        ctrace = np.rot90(ctrace)
        trace = np.rot90(trace)
        img = np.rot90(img)
        ucip = np.rot90(ucip)
        crop = cropped_args(img)

    ucip_midpoint, resolution = measure_ncs_markings(ucip)
    ucip_mask = add_ucip_to_mask(ucip_midpoint, radius=60, mask=img.mask)

    name_stub = filename.rstrip('.png').strip('T-')


    F = np.stack([frangi_from_image(img, sigma, beta=beta, gamma=gamma, dark_bg=False,
```

```
                                    signed_frangi=True, dilation_radius=20,
                                    rescale_frangi=True)
                                    for sigma in scales])
    # need to fix this in the signed_frangi logic
    F = F*~dilate_boundary(None, mask=img.mask, radius=20)

    Fmax = (F*(F>0))[:max_pos_scale].max(axis=0)
    Fneg = -(F*(F<0))[:max_neg_scale].min(axis=0)

    approx = Fmax > threshold
    rim_approx = (Fneg > neg_threshold)
    skel = thin(approx)
    completed = connect_iterative_by_label(skel, Fmax, max_dist=100)
    completed_dilated = dilate_to_rim(completed, rim_approx, max_radius=10)
    approx_dilated = dilate_to_rim(approx, rim_approx, max_radius=10)

    network = np.maximum(skel*3., (completed & ~skel)*2)
    network = np.maximum(network, rim_approx*1.)


    precision = lambda t: int(t[0]) / int(t[0] + t[2])

    mcc_FA, counts_FA = mcc(approx, trace, bg_mask=img.mask, return_counts=True)
    mcc_FAD, counts_FAD = mcc(approx_dilated, trace, bg_mask=img.mask, return_counts=True)
    mcc_NCD, counts_NCD = mcc(completed_dilated, trace, bg_mask=img.mask, return_counts=Tru

    prec_FA = precision(counts_FA)
    prec_FAD = precision(counts_FAD)
    prec_NCD = precision(counts_NCD)

    mccs.append((name_stub, mcc_FA, mcc_FAD, mcc_NCD))
    precs.append((name_stub, prec_FA, prec_FAD, prec_NCD))

    fig, ax = plt.subplots(nrows=2, ncols=3, figsize=(20,12))
    A = ax.ravel()

    A[0].imshow(cimg[crop])
    A[0].set_title(name_stub)

    A[1].imshow(ma.masked_where(Fmax==0,Fmax)[crop], cmap=cm, vmin=0, vmax=1)
    A[1].set_title(rf'$V_{\max},\beta = beta:.2f,\gamma = gamma:.3f$')

    A[2].imshow(network[crop], cmap=plt.cm.magma)
    A[2].set_title('skeleton, completed network, and rim_approx')

    A[3].imshow(confusion(approx, trace, bg_mask=img.mask)[crop])
    A[3].set_title(fr'    fixed $\alpha = threshold:.2f$', loc='left')
    A[3].set_title(f'MCC: {mcc_FA:.2f}\n'
                f'precision: {prec_FA:.2%}', loc='right')

    A[4].imshow(confusion(approx_dilated, trace, bg_mask=img.mask)[crop])
    A[4].set_title(fr'    dilate_to_rim $\alpha = threshold:.2f$', loc='left')
    A[4].set_title(f'MCC: {mcc_FAD:.2f}\n'
                f'precision: {prec_FAD:.2%}', loc='right')

    A[5].imshow(confusion(completed_dilated, trace, bg_mask=img.mask)[crop])
    A[5].set_title(fr'    network_completed, dilated', loc='left')
    A[5].set_title(f'MCC: {mcc_NCD:.2f}\n'
                f'precision: {prec_NCD:.2%}', loc='right')

    [a.axis('off') for a in A]
    fig.tight_layout()
    plt.show()
```

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

APPENDIX B

3D VISUALIZATION OF THE FRANGI FILTER

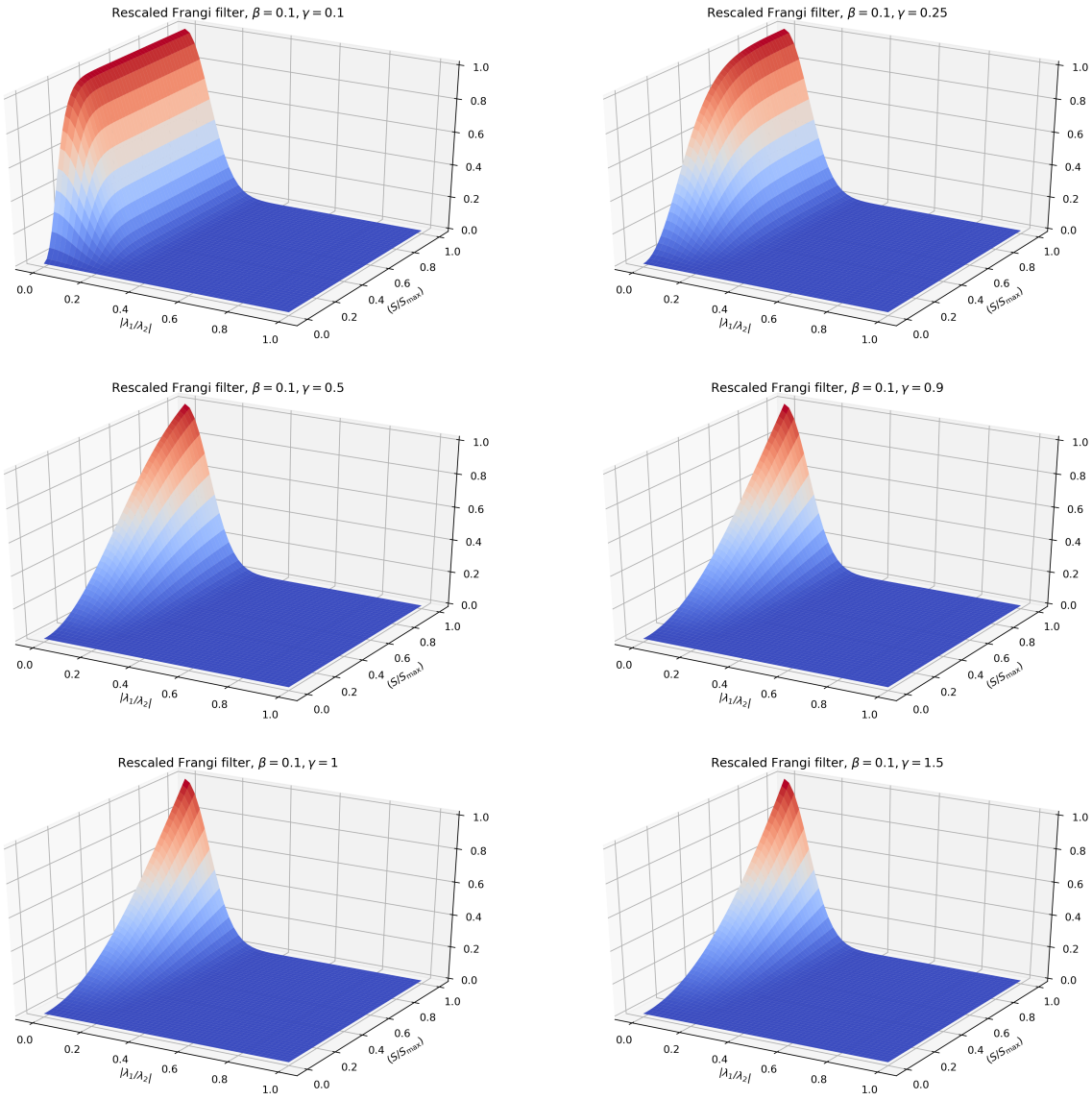


FIGURE 42: 3D graph of the Frangi Vesselness Measure, variable γ, $\beta = 0.1$

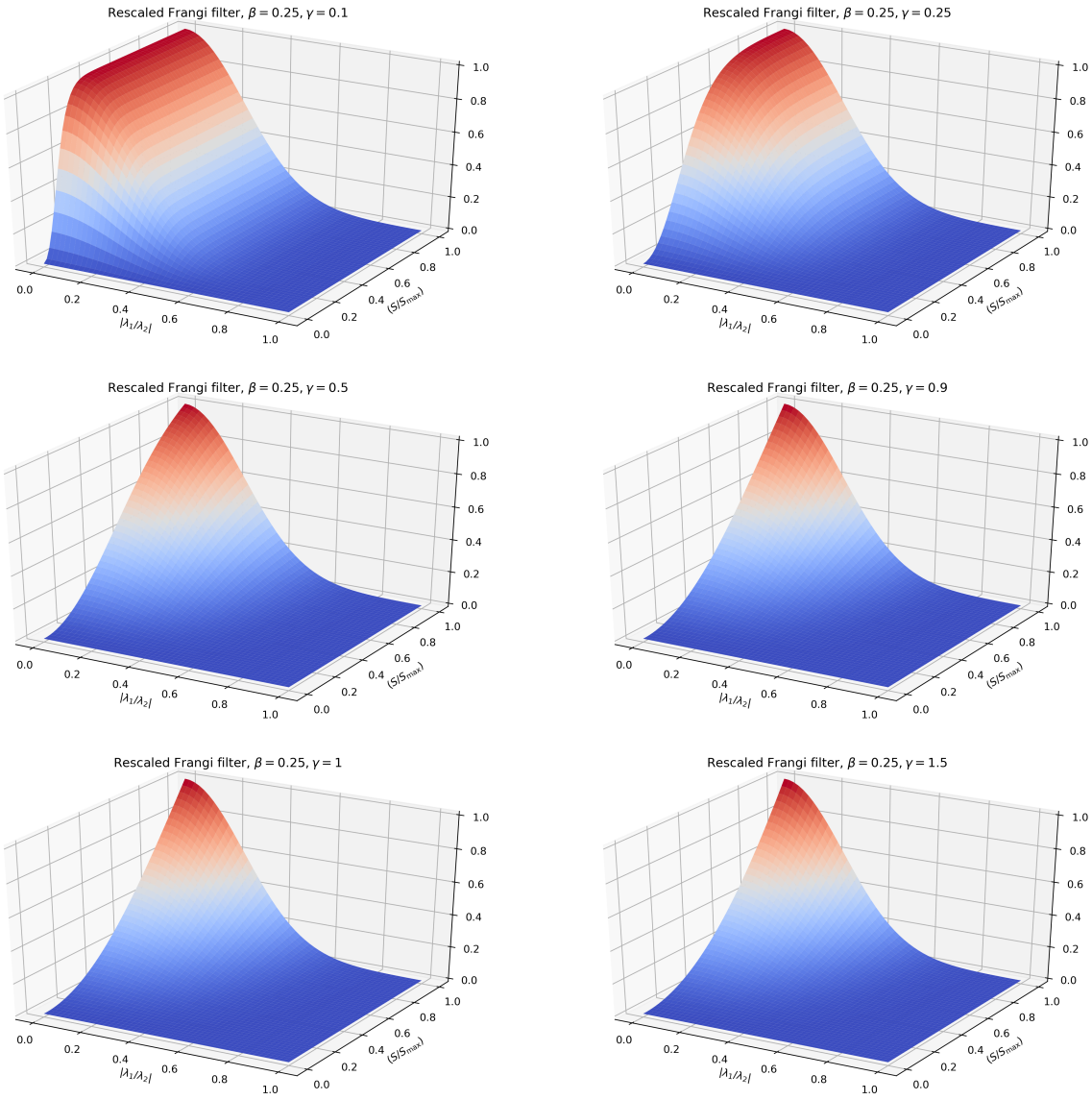


FIGURE 43: 3D graph of the Frangi Vesselness Measure, variable γ, $\beta = 0.25$

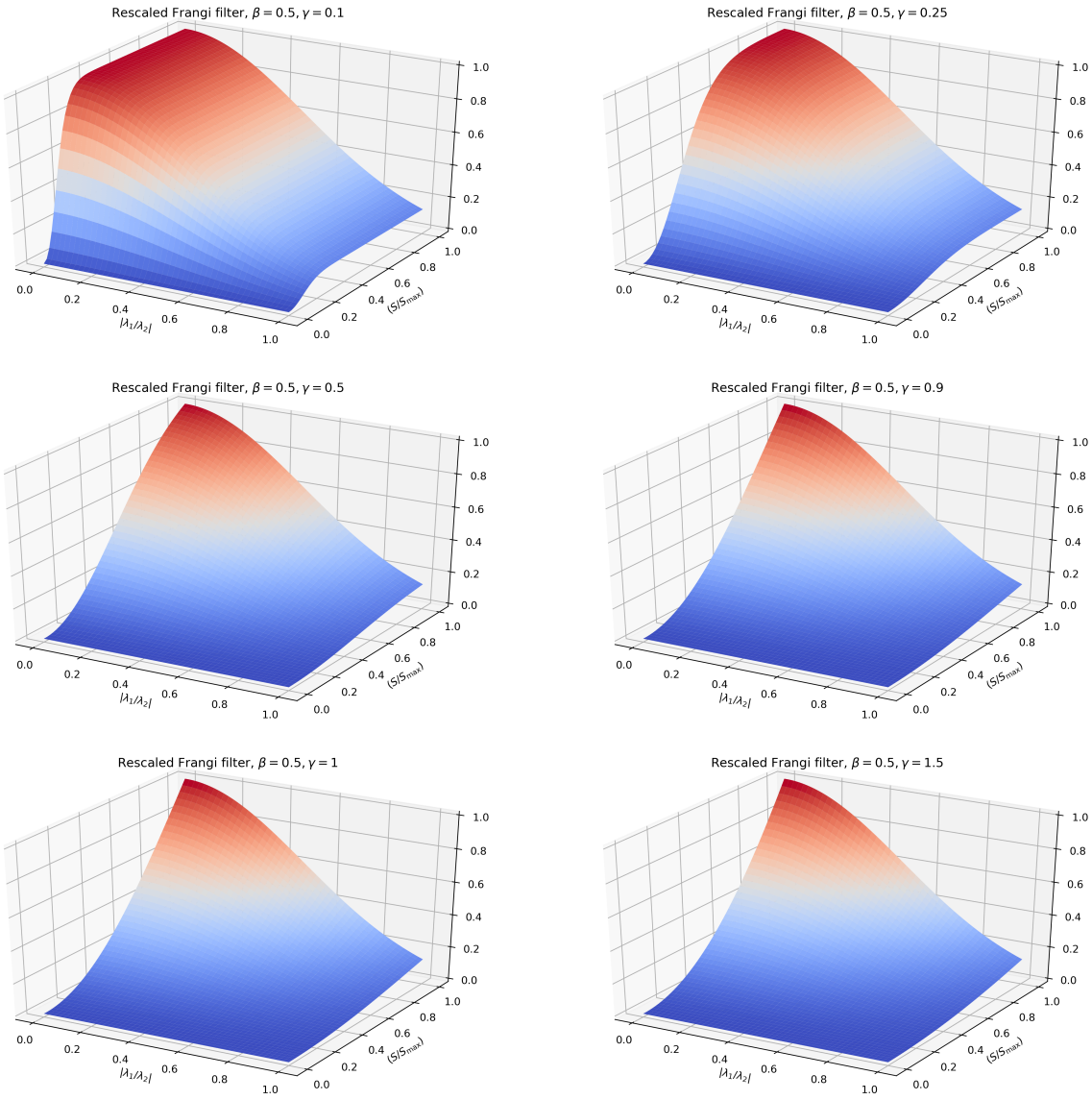


FIGURE 44: 3D graph of the Frangi Vesselness Measure, variable γ, $\beta = 0.5$

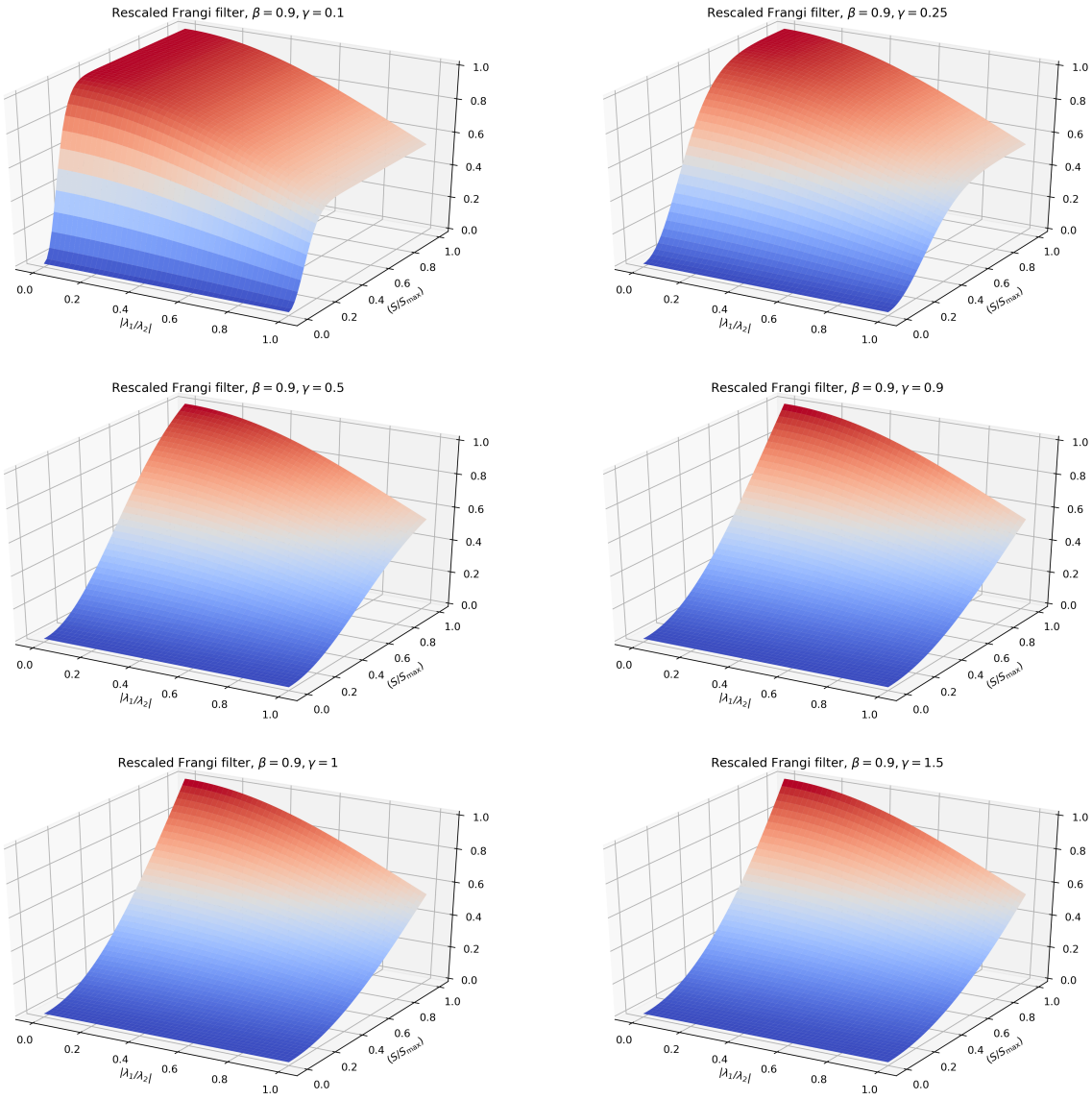


FIGURE 45: 3D graph of the Frangi Vesselness Measure, variable γ, $\beta = 0.9$

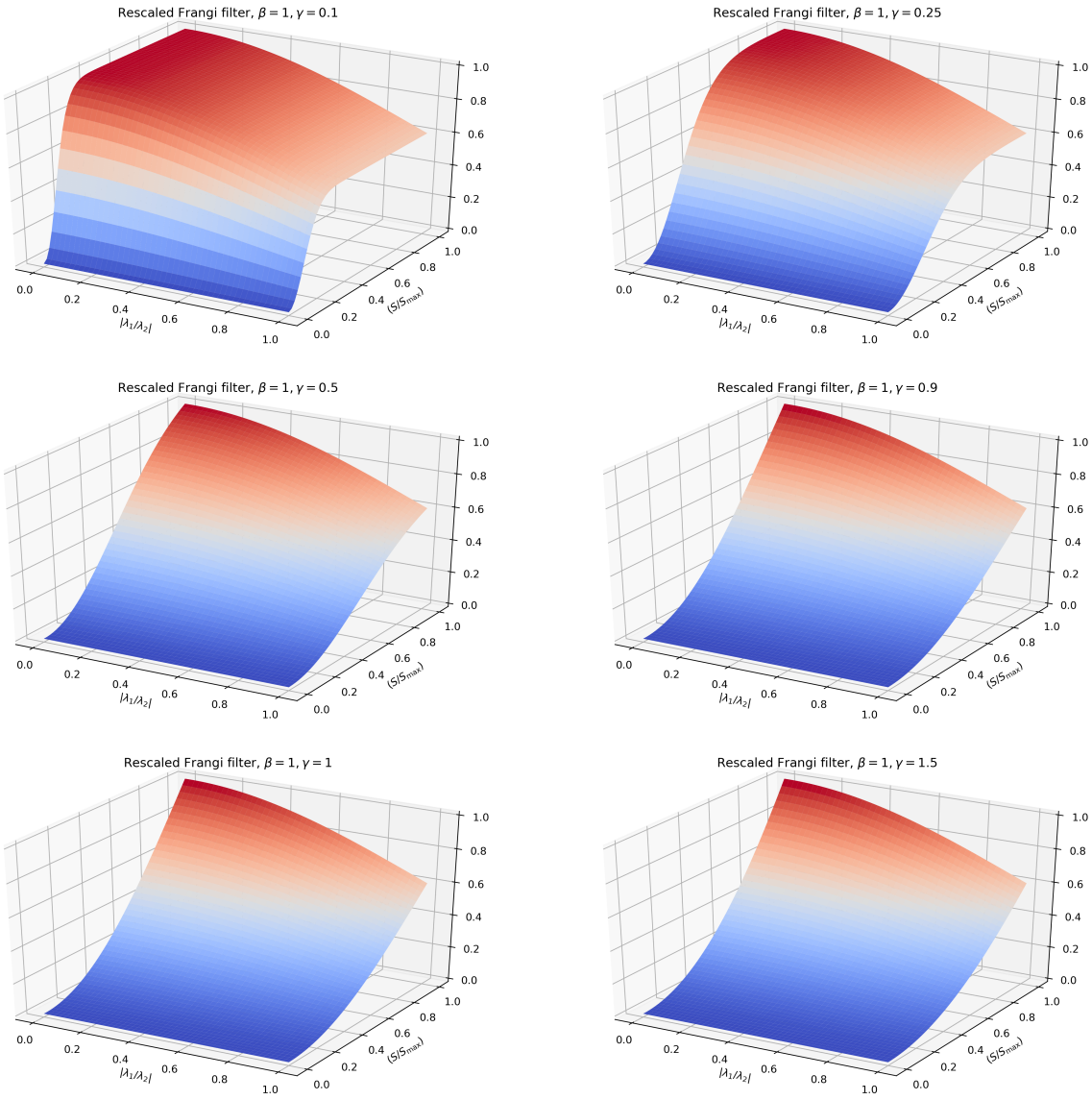


FIGURE 46: 3D graph of the Frangi Vesselness Measure, variable γ, $\beta = 1$

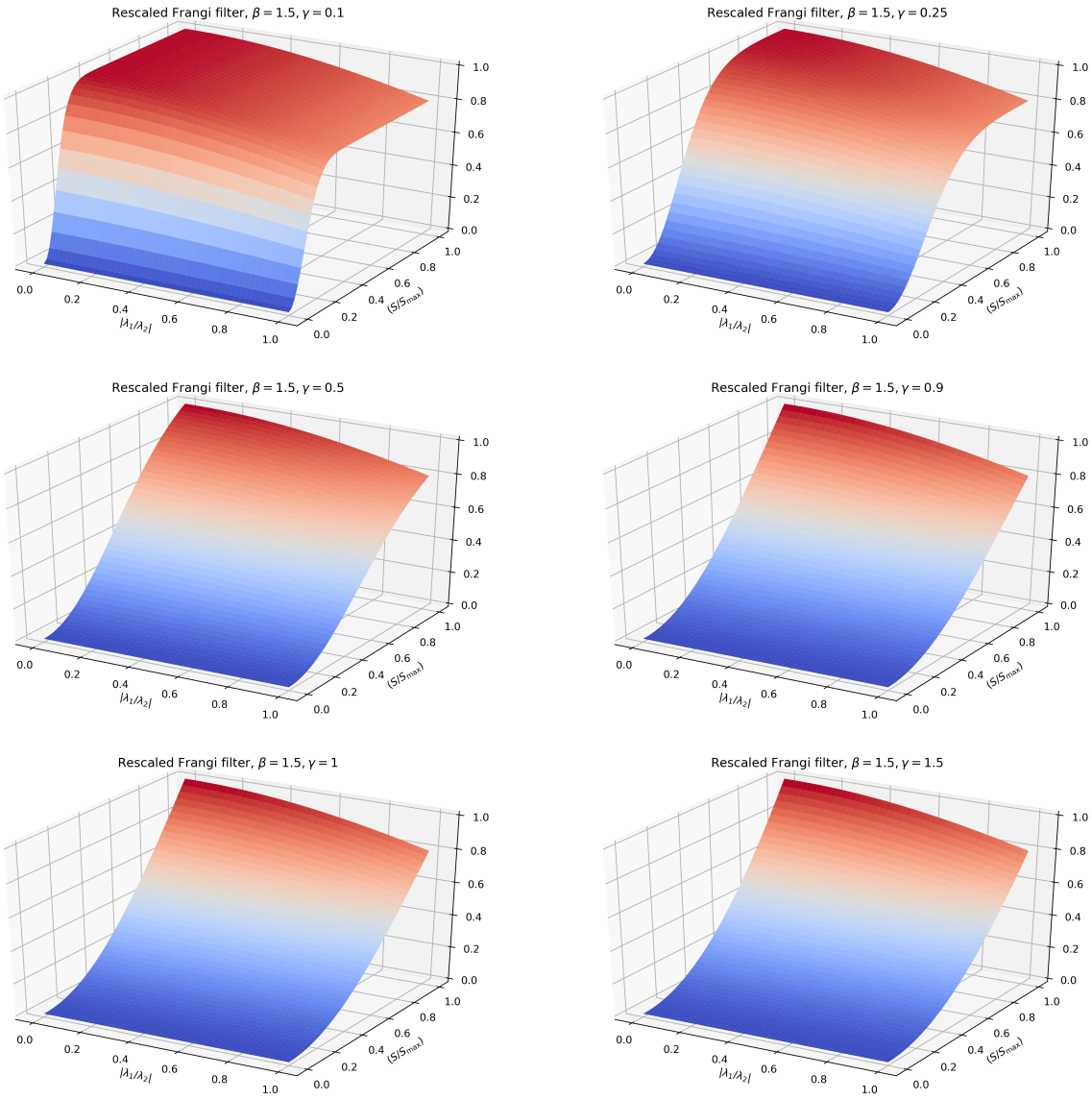


FIGURE 47: 3D graph of the Frangi Vesselness Measure, variable γ, $\beta = 1.5$

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] J.-M. Chang, H. Zeng, R. Han, Y.-M. Chang, R. Shah, C. M. Salafia, C. Newschaffer, R. K. Miller, P. Katzman, J. Moye, *et al.*, "Autism risk classification using placental chorionic surface vascular network features," *BMC medical informatics and decision making*, vol. 17, no. 1, p. 162, 2017.

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

[3] K. Y. Djima, C. Salafia, R. K. Miller, R. Wood, P. Katzman, C. Stodgell, and J.-M. Chang, "Enhancing placental chorionic surface vasculature from barium-perfused images with directional and multiscale methods," *Placenta*, vol. 57, pp. 292–293, 2017.

[4] Y.-M. Chang, R. Han, H. Zeng, R. Shah, C. Newschaffer, R. Miller, P. Katzman, J. Moye, C. Salafia, *et al.*, "Whole chorionic surface vessel feature analysis with the boruta method, and autism risk," *Placenta*, vol. 45, p. 75, 2016.

[5] N. Almoussa, B. Dutra, B. Lampe, P. Getreuer, T. Wittman, C. Salafia, and L. Vese, "Automated vasculature extraction from placenta images," in *Medical Imaging 2011: Image Processing*, vol. 7962, p. 79621L, International Society for Optics and Photonics, 2011.

[6] R. C. Gonzalez and R. E. Woods, "Digital image processing prentice hall," *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, 2006.

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

[9] R. Horn and C. Johnson, *Matrix Analysis*. Matrix Analysis, Cambridge University Press, 2012.

[10] X. Jiao and H. Zha, "Consistent computation of first-and second-order differential quantities for surface meshes," in *Proceedings of the 2008 ACM symposium on Solid and physical modeling*, pp. 159–170, ACM, 2008.

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

[12] A. F. Frangi, W. J. Niessen, K. L. Vincken, and M. A. Viergever, "Multiscale vessel enhancement filtering," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pp. 130–137, Springer, 1998.

[13] Y. Sato, S. Nakajima, N. Shiraga, H. Atsumi, S. Yoshida, T. Koller, G. Gerig, and R. Kikinis, "Three-dimensional multi-scale line filter for segmentation and visualization of curvilinear structures in medical images," *Medical image analysis*, vol. 2, no. 2, pp. 143–168, 1998.

[14] C. Lorenz, I. C. Carlsen, T. M. Buzug, C. Fassnacht, and J. Weese, "Multi-scale line segmentation with automatic estimation of width, contrast and tangential direction in 2d and 3d medical images," in *CVRMed-MRCAS'97* (J. Troccaz, E. Grimson, and R. Mösges, eds.), (Berlin, Heidelberg), pp. 233–242, Springer Berlin Heidelberg, 1997.

[15] S. D. Olabarriaga, M. Breeuwer, and W. Niessen, "Evaluation of hessian-based filters to enhance the axis of coronary arteries in ct images," in *International Congress Series*, vol. 1256, pp. 1191–1196, Elsevier, 2003.

[16] J. J. Koenderink, "The structure of images," *Biological Cybernetics*, vol. 50, pp. 363–370, Aug 1984.

[17] J. Sporring, *Gaussian Scale-Space Theory*. Norwell, MA, USA: Kluwer Academic Publishers, 1997.

[18] J. Babaud, M. Baudin, R. O. Duda, and A. P. Witkin, "Uniqueness of the gaussian kernel for scale-space filtering," *IEEE Transactions on Pattern Analysis & Machine Intelligence*, vol. 8, pp. 26–33, 01 1986.

[19] T. Lindeberg, "Scale-space for discrete signals," *IEEE transactions on pattern analysis and machine intelligence*, vol. 12, no. 3, pp. 234–254, 1990.

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

[21] M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. New York: Dover, ninth dover printing, tenth gpo printing ed., 1964.

[22] T. Lindeberg, "Discrete derivative approximations with scale-space properties: A basis for low-level feature extraction," *Journal of Mathematical Imaging and Vision*, vol. 3, no. 4, pp. 349–376, 1993.

[23] T. Lindeberg, "Feature detection with automatic scale selection," *International journal of computer vision*, vol. 30, no. 2, pp. 79–116, 1998.

[24] B. Fornberg, "Generation of finite difference formulas on arbitrarily spaced grids," *Mathematics of computation*, vol. 51, no. 184, pp. 699–706, 1988.

[25] A. Morar, F. Moldoveanu, and E. Gröller, "Image segmentation based on active contours without edges," in *2012 IEEE 8th International Conference on Intelligent Computer Communication and Processing*, pp. 213–220, IEEE, 2012.

[26] E. Jones, T. Oliphant, P. Peterson, *et al.*, "SciPy: Open source scientific tools for Python," 2001–. [Online; accessed <today>].

[27] S. Damelin and N. Hoang, "On surface completion and image inpainting by biharmonic functions: Numerical aspects," *International Journal of Mathematics and Mathematical Sciences*, vol. 2018, 2018.

[28] H. Lange, "Automatic glare removal in reflectance imagery of the uterine cervix," in *Medical Imaging 2005: Image Processing*, vol. 5747, pp. 2183–2193, International Society for Optics and Photonics, 2005.

[29] B. Matthews, "Comparison of the predicted and observed secondary structure of t4 phage lysozyme," *Biochimica et Biophysica Acta (BBA) - Protein Structure*, vol. 405, no. 2, pp. 442–451, 1975.

[30] L. Grady, "Random walks for image segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 28, pp. 1768–1783, Nov. 2006.

[31] C. Anghel, K. Archer, J.-M. Chang, A. Cochran, A. Radulescu, C. M. Salafia, R. Turner, K. Y. Djima, and L. Zhong, "Placental vessel extraction with shearlets, laplacian eigenmaps, and a conditional generative adversarial network," in *Understanding Complex Biological Systems with Mathematics*, pp. 171–196, Springer, 2018.

[32] T. Ridler, S. Calvard, *et al.*, "Picture thresholding using an iterative selection method," *IEEE trans syst Man Cybern*, vol. 8, no. 8, pp. 630–632, 1978.

[33] Z. Guo and R. W. Hall, "Parallel thinning with two-subiteration algorithms," *Communications of the ACM*, vol. 32, no. 3, pp. 359–373, 1989.