A Semi-Automated Finite Difference Mesh Creation Method for Use with Immersed Boundary Software IB2d and IBAMR

D. Michael Senter[a,b,*], Dylan R. Douglas[a,d], W. Christopher Strickland[a,c], Steven G. Thomas[a], Anne M. Talkington[a,b], Laura A. Miller[a,b,d], Nicholas A. Battista[e]

[a]*Dept. of Mathematics, CB 3250, University of North Carolina, Chapel Hill, NC, 27599*
[b]*Bioinformatics. and Comp. Biology, CB 7264, University of North Carolina at Chapel Hill, Chapel Hill, NC 27599*
[c]*Department of Mathematics, 227 Ayres Hall, 1403 Circle Drive, Knoxville, TN 37996*
[d]*Dept. of Biology, CB 3280, University of North Carolina, Chapel Hill, NC, 27599*
[e]*Dept. of Mathematics and Statistics, The College of New Jersey, 2000 Pennington Rd., Ewing, NJ 08628*

Abstract

Numerous fluid-structure interaction problems in biology have been investigated using the immersed boundary method. The advantage of this method is that complex geometries, e.g., internal or external morphology, can easily be handled without the need to generate matching grids for both the fluid and the structure. Consequently, the difficulty of modeling the structure lies often in discretizing the boundary of the complex geometry (morphology). Both commercial and open source mesh generators for finite element methods have long been established; however, the traditional immersed boundary method is based on a finite difference discretization of the structure. Here we present a software library for obtaining finite difference discretizations of boundaries for direct use in the 2D immersed boundary method. This library provides tools for extracting such boundaries as discrete mesh points from digital images. We give several examples of how the method can be applied that include passing flow through the veins of insect wings, within lymphatic capillaries, and around starfish using open-source immersed boundary software.

Keywords: immersed boundary method, fluid-structure interaction, mathematical biology, biomechanics, biofluids

1. Introduction

The immersed boundary method (IBM) is a mathematical formulation and numerical method for fully-coupled fluid-structure interaction problems that dates back to Peskin in

*I am corresponding author

Email addresses: dmsenter@live.unc.edu (D. Michael Senter), dylan_ray@med.unc.edu (Dylan R. Douglas), cstric12@utk.edu (W. Christopher Strickland), stevent3115@gmail.com (Steven G. Thomas), annemt@email.unc.edu (Anne M. Talkington), lam9@unc.edu (Laura A. Miller), battistn@tcnj.edu (Nicholas A. Battista)

URL: http://battistn.pages.tcnj.edu (Nicholas A. Battista)

1972 [1]. Since its creation, the IBM has been used to study a wide variety of problems in biological fluid dynamics and fundamental fluid dynamics at low to intermediate Reynolds numbers ($Re < 10,000$). Diverse examples include the aerodynamics of insect flight [2, 3, 4], lamprey swimming [5, 6], jellyfish swimming [7, 8], and fluid flows through organs such as the heart and esophagus [9, 10, 11]. The relative ease of implementation and the availability of open source codes has made it particularly useful in research and education [12, 13, 14].

The original IBM formulation discretizes immersed, elastic boundaries on a curvilinear finite difference mesh. Many immersed boundary studies are performed in 2D and use simple geometries with easy mathematical descriptions such as plates [2, 15], strings [16, 17], tubes [10, 18, 19], ellipses [20, 21], hemiellipses [22, 8], and circles [23, 24, 25], or in 3D with spheres [26] or cylinders [27]. In other cases, more complicated geometries are manually constructed by the user via explicit mathematical functions or sets of functions that describe the elastic boundary [28, 9, 29]; this endeavor is, however, non-trivial. David Baraff, a Senior Research Scientist at Pixar Animation Studios has publicly said, "I hate meshes. I cannot believe how hard this is. Geometry is hard." [30].

This immediately highlights a challenge in performing immersed boundary simulations for many biological applications that have complicated geometries. Most meshing tools are finite element based, such as MeshLab [31], Gmsh [32], or TetGen [33], all of which are open source. As far as we are aware, there is not an openly available and easy to use tool for generating curvilinear finite difference meshes. A few finite difference meshing tools that are available constrain the mesh to a Cartesian grid using cuboids, such as the open source AEG Mesher [34], which was designed for electromagnetic simulations, and the propriety software Argus ONE [35], which was designed to help incorporate geographic information system (GIS) into numerical models [36].

What's more, many applications, especially those in biology and medicine, usually have some imaging data from which a mesh is estimated. For example, imagine generating a numerical model of blood flow through an arterial network. There are highly resolved images that clearly illustrate arterial branching patterns from which desirable geometric data, e.g., artery diameters, lengths, branching locations, etc., can be obtained. To perform numerical simulations that reveal the spatial variations in the flow due to small scale geometric effects, one must reconstruct this arterial network in detail. Even for 2D simulations this process is non-trivial, as one would first need to recreate the structure using parametric functions and then select particular parameter values that sample the structure's geometry to obtain a computational mesh with equally-spaced points. While this laborious approach may work for some simplified arterial geometries, if the actual arterial network contains walls that are not perfectly smooth or flat, simple tubular models may be insufficiently detailed and hence give rise to non-realistic results. Our software aims to fill this gap using edge detection on the original image, thereby preserving the original pattern and information from the images. Bézier curves are then used to mathematically describe the images, after which they are appropriately sampled to obtain a curvilinear mesh.

The challenge here is two-fold: first a continuous, parameterized description of the boundary of interest must be found, potentially through image segmentation, and then this boundary must be represented as a finite difference mesh with sampling to give the

desired geometric spacing between adjacent geometric nodes. Given the widespread use of the IBM approach in both research and education [14, 12, 13, 37, 38], we found it useful to create the open source software package *MeshmerizeMe*, a tool that both detects boundaries in image data and creates finite difference meshes where the nodes are nearly uniformly spaced. The output files are designed to be coupled with IB2d [39, 12, 13], IBAMR [40], and other immersed boundary 2D software.

We provide an overview of the software *MeshmerizeMe* in Section 2. In particular, we detail *MeshmerizeMe*'s implementation, workflow (Section 2.1), and how the software computes a discretized mesh from parametric curves (Section2.2). We then present a variety of examples using the software in Section 3; including two internal flow examples and one external flow example. The examples include hemolymph flow through dragonfly wing veins (Section 3.1), flow through a lymphatic capillary (Section 3.2), and oscillatory flow past a starfish (Section 3.3).

The most straightforward immersed boundary simulations using this software would be developed to simulate the flow past or through nearly rigid, complex, biological boundaries, as demonstrated in these examples. We do not currently have a method for using multiple images to simulate moving boundaries. There are, however, a few ways that one can model moving and deformable boundaries. To begin, each of the Lagrangian points can be translated or rotated using a prescribed mathematical function rather than moving the boundary based on image tracking. This approach has been used for a variety of biological applications, including flapping insect wings [2], swimming jellyfish [41, 8], flapping swimmerets [42] and heart pumping [19]. If, in addition to vertex points, springs and beams are added to pairs or triads of vertex points, elastic deformations due to the fluid-structure interaction can also be simulated. This type of approach has been used for leaves in flow [43], flapping filaments [16], the deformation of prey prior to puncture [44], and the movement of red blood cells [45, 46].

2. *MeshmerizeMe* Implementation

MeshmerizeMe is a software package for the creation of 2D geometry files for use with open source immersed boundary software such as IB2d and IBAMR. The software comes with two main scripts: 1) *ContourizeMe*, which reads in an image file and uses automatic edge detection to extract contours of interest into an SVG file, and 2) *MeshmerizeMe*, which processes SVG files and IB2d- or IBAMR-style input2d files to create *.vertex files describing the geometry of the SVG file at the appropriate resolution. Both SVG and vertex files are UTF encoded text files. SVG is a widely supported vector graphics format, while the vertex-format is used by IB2d and IBAMR to describe Lagrangian mesh points in an immersed boundary simulation. The *MeshmerizeMe* script also includes a tool that uses Matplotlib to allow the user to plot the geometry created by *MeshmerizeMe* for visual verification. These scripts are written to run in Python 3.x, and upon installation both scripts are added to the path on a Linux and Mac environment. This dual-script setup allows the end-user two distinct entry points into the workflow; see Fig. (1) for an illustration.

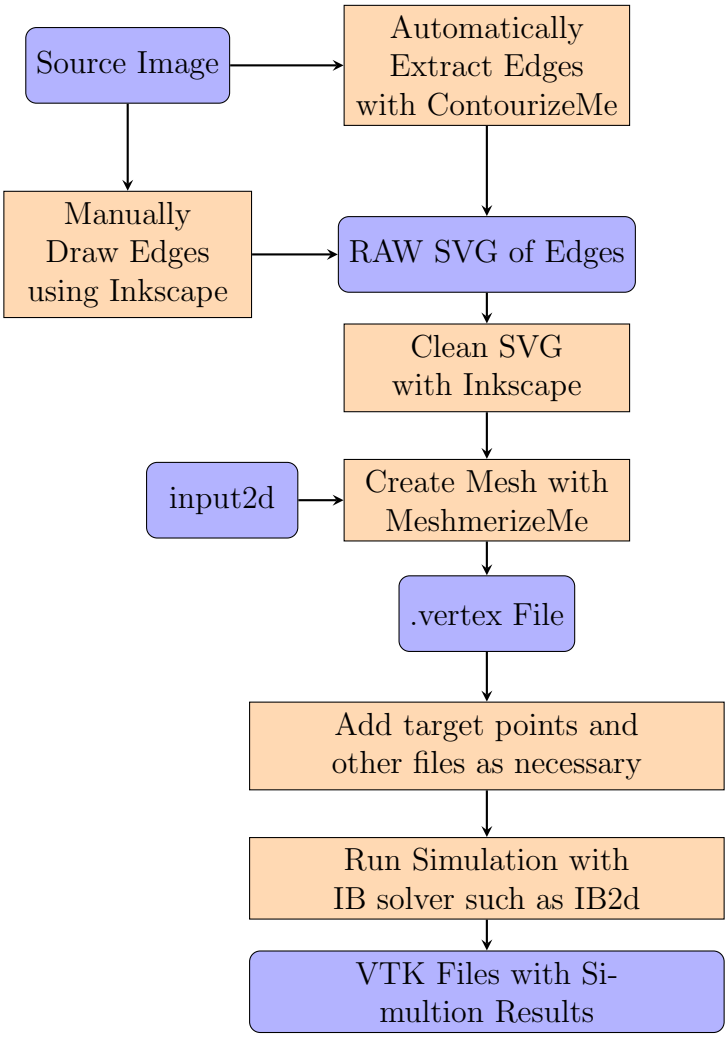

Figure 1: Flow chart illustrating the MeshmerizeMe workflow. Blue rounded boxes represent files, while square orange boxes represent user actions. The flow chart illustrates the two main entry points into the work flow from the source image: automatic edge extraction using the ContourizeMe script as well as manually creating the SVG by drawing "over" the image using software like Inkscape.

To provide a concrete example of the workflow, suppose an image is available either from the field or an experiment. To detect the edges and generate an SVG file, the user would run the *ContourizeMe* script on the desired image file (e.g., by typing `ContourizeMe image.jpg` in the commandline). This opens a GUI with several features that may be used to modify and enhance the image (see Figure 2). Note that common image formats such as jpg, png, and tiff are supported. For best performance, the image should provide a good contrast between the object boundary and background, while also having little noise. If this is not the case, *ContourizeMe* allows the user to adjust the image contrast and saturation[1] using simple sliders to better highlight the boundary of interest. Using a slider for the pixel cutoff, the smoothness of the matched curve can be adjusted to account for noise. All of these sliders update in real time. If this proves insufficient, unwanted edges that were detected can easily be deleted at a later step. Once the user is satisfied with the result, the curve is exported to an SVG file to be used as input for the *MeshmerizeMe* script.

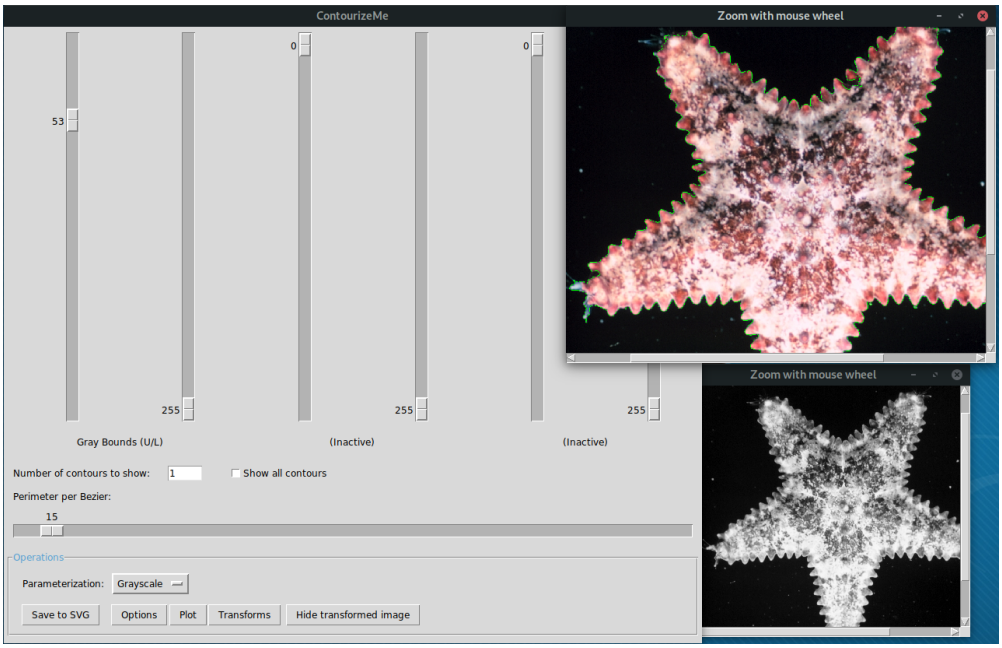

Figure 2: The *ContourizeMe* GUI in action on Manjaro Linux 18.0.4 with Gnome DE emulated in VirtualBox. The main window to the left allows the user to select the desired type of parameterization of the source image. The user can chose grayscale, RGB, or HSV. The sliders allow the user to set the desired threshholding in that parameterization. The detected edges are displayed in the live image (upper right) in green. The user may optionally display the result of the paramterization and filtering (lower right).

In some cases, the original image quality may be high enough to proceed directly to the discretization phase. In most cases, however, the user will want to make minor edits to the SVG file to remove any potential artifacts, such as curves corresponding to background noise in addition to the boundaries of interest. These edits can be done using common

[1]those changes are temporary and do not affect the original image data

vector graphics software such as Inkscape (open-source) or Adobe Illustrator (commercial). Note that this also provides an alternate entry-step in the *MeshmerizeMe* workflow: curves can be freely drawn using such software if image data is either not available, the boundary structure is purely hypothetical, or in cases when the original image is too poor in quality for reliable edge detection. In the latter case, image layers may be used in software such as Inkscape that allows the user to trace over the desired edges to create the SVG file with the necessary boundaries. Once the SVG file has been cleaned in this manner, it is advised to collapse underlying groups and simplify contiguous paths, which in some vector graphics software can be done from the "save" menu. In other cases, software to do this is freely available online, such as SVGO and SVGOMG [47, 48].

With the desired geometry extracted into an SVG image, the next step consists of making a folder in which the SVG file itself can be found, as well as an IB2d-style file called `input2d`. This latter file includes information such as the spatial discretization step size (more details on this can be found below). Our script reads this file to calculate and sample the appropriate mesh. *MeshmerizeMe* is run by pointing it to the appropriate SVG file (e.g., typing `MeshmerizeMe image.svg` from the command line). It will then create the `.vertex` file containing the mesh. Note that the filename of the vertex file will be taken straight from the input2d file, regardless of the SVG filename. If multiple meshes are to be created, *MeshmerizeMe* can be run in batch mode by providing it with a list of file names. We also support piping from STDIN. This allows the user to easily pass a list of file names, such as one created by the find command, to *MeshmerizeMe* for batch processing.

The resulting `.vertex` file can then be used as input for immersed boundary simulations using IB2d and IBAMR. Note that the user will need to supply some additional information as to the relationship between the boundary points, for example whether or not they are connected with springs, beams, masses, and so forth. Currently, files that store this information must be manually created, although a few of these relations are implemented as classes in the MeshmerizeMe library to help with writing such scripts. Once this stage is completed, the immersed boundary simulation is ready to run.

2.1. Overview of contour extraction

In this section, we provide an overview of how *ContourizeMe* extracts contours from images. This is motivated by the need to accurately estimate the shapes of objects from planar images or within some cross-section. Many techniques, ranging from edge-finding using image gradients to image segmentation accomplished through supervised training of Deep Neural Nets, have been proposed as generalized methods to extract such edges [49, 50]. The niche filled by *MeshmerizeMe* is to easily obtain 2d meshes from image data that can be directly used in IB2d and IBAMR. In essence, it allows for the semi-automated generation of meshes from image data using simple contour estimation from hand chosen thresholds of pixel values [51] as a first step in the IBM workflow. This replaces the need to completely create the structure mesh manually by finding idealized functions approximating the shape of interest. This method was chosen for its simplicity and fast estimation in obtaining user-verified 2d shapes of arbitrary smoothness and precision.

The *ContourizeMe* GUI was developed with the Python package Tkinter. Contour estimation from image thresholding works by first applying noise reduction to a given image if needed. The contour is assumed to be represented in the image by a gradient or steep change in the pixel values that separates the foreground, or object of interest (OOI), from the background. For many images this means the existence of one or three inequalities or pixel-value bounds (3 for the case of RGB and HSV values) that quantify this separation. Image noise from one or multiple sources can make these inequalities ill-defined. Possible sources of image noise are numerous and include sensor and electronic-circuit noise, analog-to-digital conversion errors, and even statistical quantum fluctuations [52, 53]. *ContourizeMe* provides implementations of various common noise reduction techniques that the user may choose from depending on the source and strength of the noise present in their own images.

In the next step after determining an appropriate noise reduction technique, the user manually determines one or more pixel value bounds depending on a given parameterization (RGB, grayscale, HSV, etc.) that forms the lowest-area hull that corresponds to the object of interest. This closed region is used to produce a binary image with pixel values of 1 corresponding to those contained in the provided region and 0 corresponding to those not in this region. A topological algorithm in OpenCV [51] is applied to this binary image to give contours that fully describe 'separate' clusters of homogeneous pixels (in this case pixels that all equal 1). This algorithm yields integer pixel estimates of the boundaries, which are then refined to sub-pixel estimates with user specified smoothness via the Chan-Vese algorithm. More details are provided in the following subsections.

These contours themselves are estimations of the shapes of interest that are then used to yield precise descriptions of the shapes as a set of continuous Bézier curves (see Appendix B). Bézier curves are constructed using evenly spaced points from the sub-pixel boundary estimates and exported in the SVG format.

2.1.1. Noise reduction

Filtering algorithms, the topological contour estimation algorithm, and most of the image manipulations (such as RGB to HSV conversion, thresholding, etc.) are accomplished in *ContourizeMe* via Python bindings of the OpenCV package [51]. OpenCV is a computer vision suite developed in C++ built to tackle various problems including segmentation, 3D reconstruction, edge-finding, and other related tasks.

ContourizeMe's main GUI includes:

- An average filter which essentially is a type of down sampling that assumes the true value of any pixel can be estimated by the average pixel value of a K by K window surrounding that pixel. This is equivalent to convolving the image with a low-pass filter kernel.

- A Gaussian filter which convolves the image with a Gaussian kernel of a specified size and standard deviation in both the x and y directions. This is similar to the average filter, but pixels are weighted via the 2D Gaussian function specified before they are averaged.

- A median filter which instead of the average over a window, takes the median pixel value. This kind of filter is typically used for "salt and pepper" type image noise, and has the advantage of leaving only pixel values that would have been observed in the original image.

- A bilateral filter which is the recommended choice. The bilateral filter behaves similarly to the Gaussian filter, but in addition to weighting pixels by their spatial distance it also weights them by their difference in intensity in an attempt to preserve edges or gradient information.

While the bilateral filter is recommended because of its intended edge-preservation [54], one may want to employ one of the other convolution filters as they can smooth boundaries produced by the thresholding and topological algorithms. We must also stress that this selection is highly limited in scope and much more sophisticated and robust techniques for the denoising of images exist depending on the image acquisition method and content. It may be that making a model of the noise via a deep neural net such as UNet [55], CAIR [56], Noise2Noise or Noise2Void [57] may be required or produce better results. Any method may of course be employed before using this segmentation GUI.

2.1.2. Smoothing the results from OpenCV's algorithm

In order to give the user control over the smoothness of the resulting curve they obtain, we use a Python implementation of the Chan-Vese level set algorithm [58]. This method of smoothing the curve ensures that reductions in the curvature are chosen such that they have minimal costs to accuracy and that the contour remains true to the original image. We allow users to specify both the error tolerance in pixels, as estimated from each iteration of the Chan-Vese algorithm, and the parameter α which controls the contribution of the total curvature of the boundary to the energy functional and thus the smoothness of the obtained contour.

2.2. Going from curves to mesh

The second part of our software package consists of a script that takes vector based graphics, specifically the "Scalable Vector Graphics" (SVG) standard, to obtain a discretized curvilinear mesh that describes the boundary of the object of interest. The idea behind vector graphics is to represent shapes in terms of control points of non-uniform rational basis splines (NURBS). Only control points of the parameterized curves are stored while the standard defines the basis polynomials themselves. The resulting curves can be represented smoothly at any scaling or resolution of interest and can easily be mapped to the simulation space using an affine transformation.

A variety of vector graphics file formats are available, several of which are proprietary. We have chosen to implement our software using the SVG standard because it is a popular, open-source standard and for its ease of use. SVG files are UTF encoded text files following an XML schema, making them amenable to XML parsing methods. A particular benefit of the SVG standard is that it is widely supported; if edge detection fails or gives insufficient

resolution, multiple vector graphics programs such as Inkscape or Adobe Illustrator may be used to clean up or directly hand-draw the boundaries of interest from an image. The standard has support both for Bézier curves as well as geometric primitives (rectangles, triangles, etc.), and the current version of *MeshmerizeMe* utilizes the free path element, which encodes curves as Bézier curves.

To reduce the need for additional configuration files, *MeshmerizeMe* has been built to utilize the 'input2d' file format that is utilized by IB2d and IBAMR. This file is required, and the *MeshmerizeMe* code expects the following variables to be defined in the input2d file:

- `Lx`, `Ly`: the length of the computational domain in the x and y direction, respectively.
- `Nx`: number of points in the x direction.[2]

Even if the user chooses to use a different CFD software for the simulation,*MeshmerizeMe* can still be used to create the requisite mesh points. Strict adherence to the IB2d format is not required. A minimal working example of the input2d file required for *MeshmerizeMe* requires only four lines. The example below will create a mesh appropriate for a $[0, 0.5] \times [0, 0.5]$ domain with a 64×64 mesh.

```
Nx =  64
Lx = 0.5
Ly = 0.5
string_name = test
```

Please note that this minimal example is only sufficient for *MeshmerizeMe*. A simulation for IB2d or IBAMR will require additional settings in the `input2d` file, such as the fluid paramaters `mu` and `rho` and temporal information such as the desired time step `dt` and time the simulation is to run. Any such additional settings may be present in the `input2d` file, but will be ignored by *MeshmerizeMe*.

MeshmerizeMe will automatically compute the appropriate boundary point spacing of $\Delta s = \frac{1}{2}\Delta x$, where $\Delta x = \frac{L_x}{N_x}$. We note that it is standard in the immersed boundary literature to set the spacing between the immersed boundary points to half that of the spatial step for the Navier-Stokes solver, $\Delta s = \frac{1}{2}\Delta x$ [14]. This choice of spacing allows the boundary points to move independently while also restricting most of the flow between the points. Using these parameters, the software will parse the supplied SVG file itself and extract the path objects, splitting them up into individual Bézier curve objects. The control points are then converted to the experimental coordinate system defined by `Lx` and `Ly`.

Let $\boldsymbol{\gamma}(t) \in \mathbb{R}^2$ represent a particular Bézier curve. To create the mesh, we seek n parameters $\{t_i\}$ with $0 \leq t_1 < t_2 < \ldots < t_n \leq 1$ such that the distance between points on the curve with these parameters is fixed:

$$d(t_i) = \|\boldsymbol{\gamma}(t_{i+1}) - \boldsymbol{\gamma}(t_i)\| = \Delta s, \quad i = 1, ..., n-1\,. \tag{1}$$

[2] *MeshmerizeMe* expects a square discretization, that is $\Delta x = \Delta y$, but does not require a square computational domain.

We utilize a gradient descent method to find these parameters. Specifically, we define our cost function $J(\boldsymbol{t})$ using the mean squared relative error of all $d(t_i)$ values (scaled by $\frac{1}{2}$ to cancel the power of 2 coming from the partial derivatives in (4))

$$J(\boldsymbol{t}) = \frac{1}{2(n-1)} \sum_{i=1}^{n-1} \left(\frac{d(t_i) - \Delta s}{\Delta s} \right)^2, \boldsymbol{t} = \begin{bmatrix} t_1 & \dots & t_n \end{bmatrix} \tag{2}$$

and minimize $J(\boldsymbol{t})$ by iteratively updating $\boldsymbol{t}$ with a variable learning rate α:

$$\boldsymbol{t}_{new} = \boldsymbol{t}_{old} - \alpha \nabla J(\boldsymbol{t}), \tag{3}$$

where

$$\frac{\partial J}{\partial t_j} = \begin{cases} \dfrac{-1}{(n-1)(\Delta s)^2} \left[\dfrac{1}{d(t_j)} \left(d(t_j) - \Delta s \right) \langle \boldsymbol{\gamma}(t_{j+1}) - \boldsymbol{\gamma}(t_j), \nabla \boldsymbol{\gamma}(t_j) \rangle \right], & \text{if } j = 1 \\ \dfrac{1}{(n-1)(\Delta s)^2} \left[\dfrac{1}{d(t_{j-1})} \left(d(t_{j-1}) - \Delta s \right) \langle \boldsymbol{\gamma}(t_j) - \boldsymbol{\gamma}(t_{j-1}), \nabla \boldsymbol{\gamma}(t_j) \rangle \right], & \text{if } j = n-1 \\ \dfrac{1}{(n-1)(\Delta s)^2} \left[\begin{array}{c} \dfrac{1}{d(t_{j-1})} \left(d(t_{j-1}) - \Delta s \right) \langle \boldsymbol{\gamma}(t_j) - \boldsymbol{\gamma}(t_{j-1}), \nabla \boldsymbol{\gamma}(t_j) \rangle \\ -\dfrac{1}{d(t_j)} \left(d(t_j) - \Delta s \right) \langle \boldsymbol{\gamma}(t_{j+1}) - \boldsymbol{\gamma}(t_j), \nabla \boldsymbol{\gamma}(t_j) \rangle \end{array} \right], & \text{otherwise.} \end{cases} \tag{4}$$

Here, we use the notation $\langle \mathbf{a}, \mathbf{b} \rangle$ to denote the inner product between $\mathbf{a}$ and $\mathbf{b}$.

The number of points n to be found per path are estimated by dividing the arc-length by the desired length Δs. This may result in more dense than optimal spacing of points, but in practice achieves sufficient accuracy. The error will depend on the curvature of $\boldsymbol{\gamma}$.

We then evaluate our curve at the points t_i obtained from this technique to determine the discretized boundaries of interest. The produced Lagrangian mesh is output to a file called `fname.vertex` where 'fname' is based on the value of `string_name` taken from the 'input2d' file. The vertex file itself is a simple text file. The first line consists of an integer giving the total number of mesh points and the following lines contain one mesh point each, given as a space delimited pair of floats representing the x and y direction coordinates.

2.2.1. Distribution of Errors in Approximation

To test the relative accuracy of our script, we created 5,000 SVG files each containing a randomly generated cubic Bézier curve. For purposes of uniformity during testing, each curve was generated on a 1000×1000 pixel domain to be mapped onto a 1×1 mesh domain using a 256×256 grid. All parameters were set to default values. A script was then run to calculate the minimum, maximum, mean, and median relative errors for each curve. The mean of the median relative error of curves in the script is 3.2% The middle 50% of median relative errors is in the 2.6-3.5% range.See figure (3) for the distribution of the median relative errors.

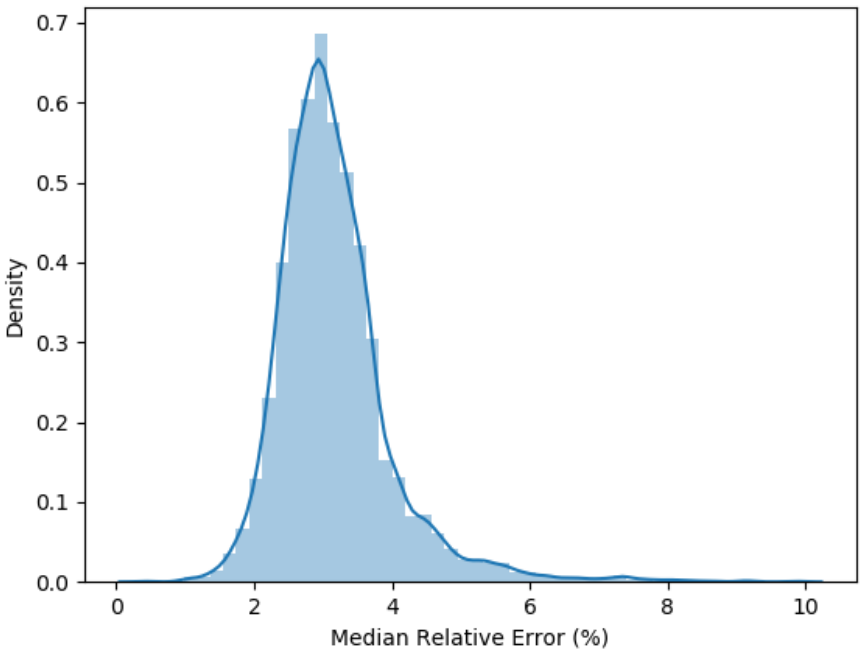

Figure 3: A distribution plot of median errors calculated from the meshes created for the error experiments in Section (2.2.1).

Note that if a higher degree of accuracy is desired, the user is able to experiment with different parameters that can influence the average error. *MeshmerizeMe* allows the user to set both the learning rate as well as the convergence threshold of the mean squared error (MSE) as command-line options when creating the mesh.

One potential limitation on the error is the method we have chosen to seed the curves. Specifically, after reading the SVG our script merges all individual SVG path objects into a single path object. This path object is then split into several sub-paths of equal arc-length. Each of these sub-paths are seeded with $n = L_{sp}/\Delta s$ points, where L_{sp} is the length of the sub-path and Δs is the desired Euclidean distance between mesh-points. For sub-paths with very large curvature, this seeding method may overestimate the number of points necessary for ideal discretization. In such cases, the user may re-run the MeshmerizeMe script with the `num-points` flag to manually specify a lower number of points per sub-path. In our experience, this is an unusual occurrence that can usually be solved by experimenting with the `num-points` flag.

It should also be noted that our current algorithm assumes the object of interest is a contiguous path, that is we assume the object can be represented by a single poly-Bézier curve with at least geometric continuity. In the SVG file, this representation is not limited to representation by a single path element. When the SVG file is parsed, all path elements are merged into a single poly-Bézier path object. This object is then divided into several sub-paths of approximately equal length which are then processed in parallel. If the object

of interest consists of two non-contiguous paths or multiple objects are represented in the source SVG, the *MeshmerizeMe* script will report a large error resulting from the distance between two sequential points on non-contiguous paths. In the presence of several non-contiguous paths, the minimization algorithm will lead to a slight skewing of points towards path boundaries.

3. Examples: Bringing everything together

We present several examples that illustrate the software's ability to recreate complex geometries. In each example, *ContourizeMe* is used to extract contours from images, *MeshermizeMe* is then used to compute the model's discretized geometry. The flow within or around the geometries is solved using an open-source implementation of IBM, either IB2d or IBAMR. The following examples are illustrated:

1. Hemolymph flow through dragonfly wing veins (Section 3.1)
2. Lymph flow through a branching lymphatic capillary (Section 3.2)
3. Oscillatory flow past a starfish or array of starfish (Section 3.3)

In every example, we present the original image on which the computational geometry is based, followed by images that illustrate how *MeshmerizeMe* computed its associated discretized mesh. Finally, we present computational results to illustrate successful integration of the geometry into the IBM software.

3.1. IB2d: Dragonfly Wing Veins Example

For our first example, we chose a public domain image of a dragonfly wing shot with a Canon EOS 5D Mark with a 100 mm lens [59]. For the purpose of running a tractable fluid-structure interaction simulation, we cropped this image to a section of the wing and manually occluded parts of the veins using the open source image manipulation software GIMP [60]. Figure 4 shows the original image of the dragonfly wing and the region that was chosen for numerical simulation.

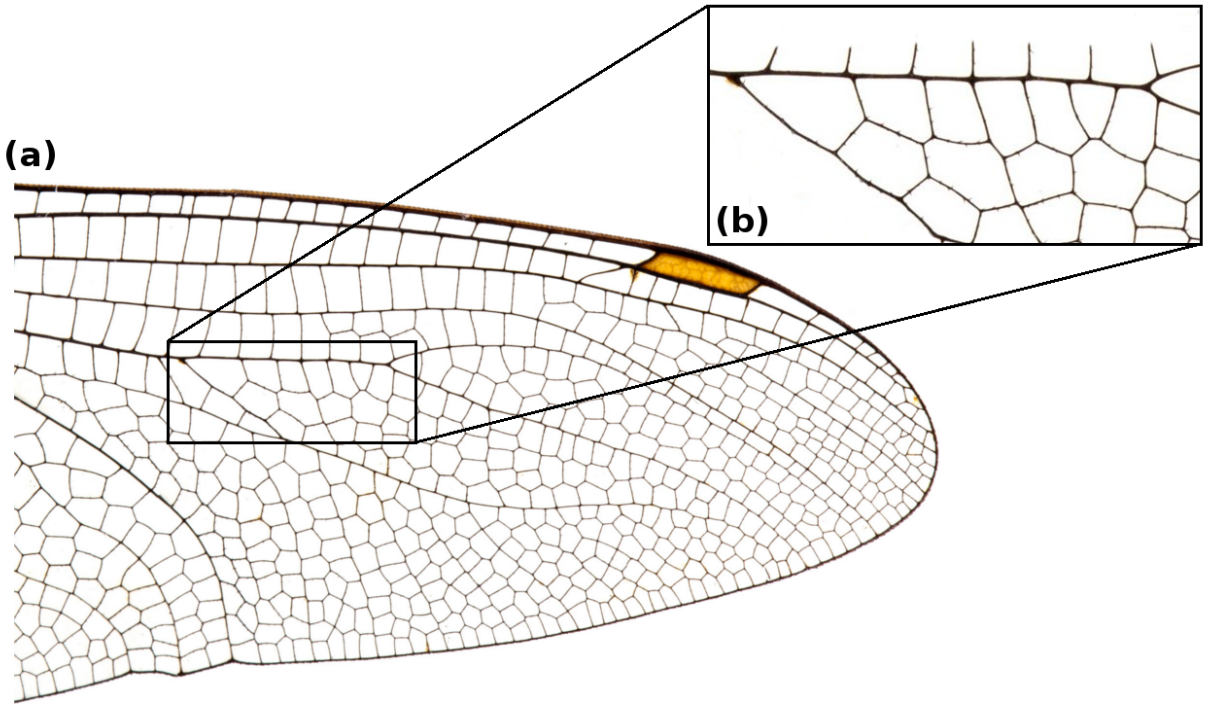

Figure 4: **(a)** The original public domain image of a dragonfly wing [59]. **(b)** The partial region of the dragonfly wing that was chosen for numerical simulation in IB2d. Some vessels were opened so that flow would have obvious entry and exit paths.

As shown in Figure 2, we used the *ContourizeMe* GUI to create boundaries describing the subsection of the wing vasculature. Briefly, the image was cropped so that only a couple dozen vessel segments would be considered. The image was then loaded into the *ContourizeMe* GUI. Noise reduction was then applied to the image, and a pixel bound was selected such that the lowest-area hull sufficiently matches the edge of the vessel network. The edges were then smoothed using the Chan-Vese algortihm, and the result was exported to SVG.

The *MeshmerizeMe* script was then used to obtain a curvilinear mesh from the SVG file. The x, y coordinates of this mesh were written to the `.vertex` file. The file contained the coordinates for approximately equally-spaced Lagrangian points that were then used as an input into IB2d. For the purpose of running an IB2d simulation where the complex geometry remains relatively fixed while the fluid is driven through it, we only require the vertex point of each Lagrangian Point, i.e. the (x, y) values of each point along the insect wing. Since the wing veins should be relatively rigid and not move, each Lagrangian vertex point was tethered to a target point (see Appendix A.1). This has the effect of applying a force proportional to the distance between the location of the actual boundary and the desired position. In other words, the boundary is pushed back into place as fluid moves through the network. The necessary input information for IB2d is written in the `wing_veins.vertex`

and `wing_veins.target` files.

This example can be found in the open source IB2d software available at `github.com/nickabattista/IB2d` and this example can be found in the following subdirectory, `IB2d/matIB2d/Examples/Example_MeshmerizeMe/Dragonfly_Wing/`. More details on IB2d and the immersed boundary method in general can be found in Appendix A.

To drive flow through the wing, a penalty force is applied to the fluid that is proportional to the difference between the local fluid speed and the local target velocity (see Appendix A.1). For our example, a parabolic flow profile is enforced at the inlet of the insect wing, and all subsequent flows through the veins result from that inflow and are not themselves prescribed (see Figure A.11). The full implementation of this simulation can be found in the `please_Compute_External_Forcing.m` script.

To illustrate flow through the wing vein geometry, we ran multiple simulations corresponding to various Reynolds numbers (Re), given by

$$Re = \frac{\rho L U}{\mu}, \tag{5}$$

where ρ is the density of the fluid (kg/m^2), μ is the dynamic viscosity ($N \cdot s/m$), and L and U are characteristic length and velocity scales respectively. We note that in these simulations, we varied μ while holding $\rho = 1000\ kg/m^2$, $L = w$ (width of the vein where the inflow is produced), and $U = 5\ m/s$ (the maximum speed of the parabolic inflow). Simulations were run over a couple orders of magnitude of Re ranging through $Re \in [0.1, 100]$ on a $[0, 0.5] \times [0, 0.25]$ grid with resolution of $dx = 0.5/1024 = 0.25/512 = dy$.

The result illustrates the flow through the complex geometry as shown in Figs. 5 and 6. We note that the example of flow through a subset of the veins in a dragonfly wing was chosen to showcase the functionality of the software to capture and digitize intricate complex structures.

Figs. 5 and 6 compare the flow profiles and pressures generated for three simulations when inflow reaches its steady state through the dragonfly's complex wing vein geometry at $Re = 0.6, 6, 60$. It is clear there is more flow through the complex morphology with higher pressures for higher Re. Note that when using the immersed boundary method, the entire structure is fully immersed within a fluid, so there is fluid in the region enclosed between veins. Hence the pressure fields in such regions are not physical but instead are artifacts of these regions being within a fluid environment and being enclosed.

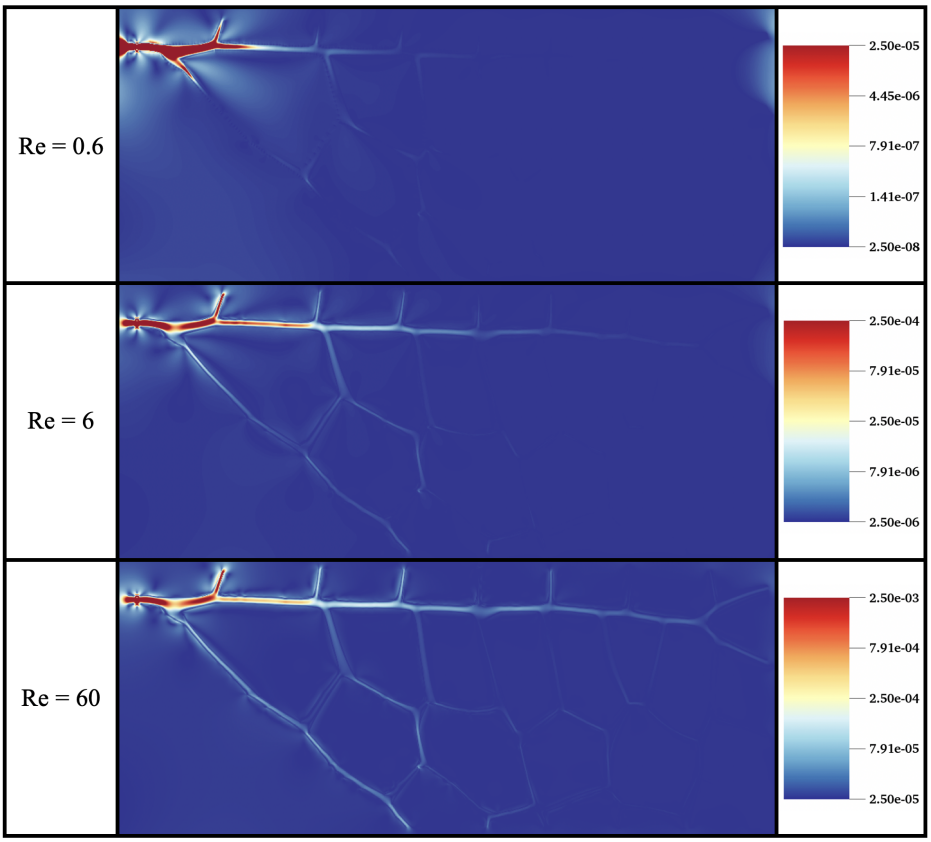

Figure 5: Snapshots comparing the magnitude of velocity for different Re, $Re = 0.6, 6, 60$.

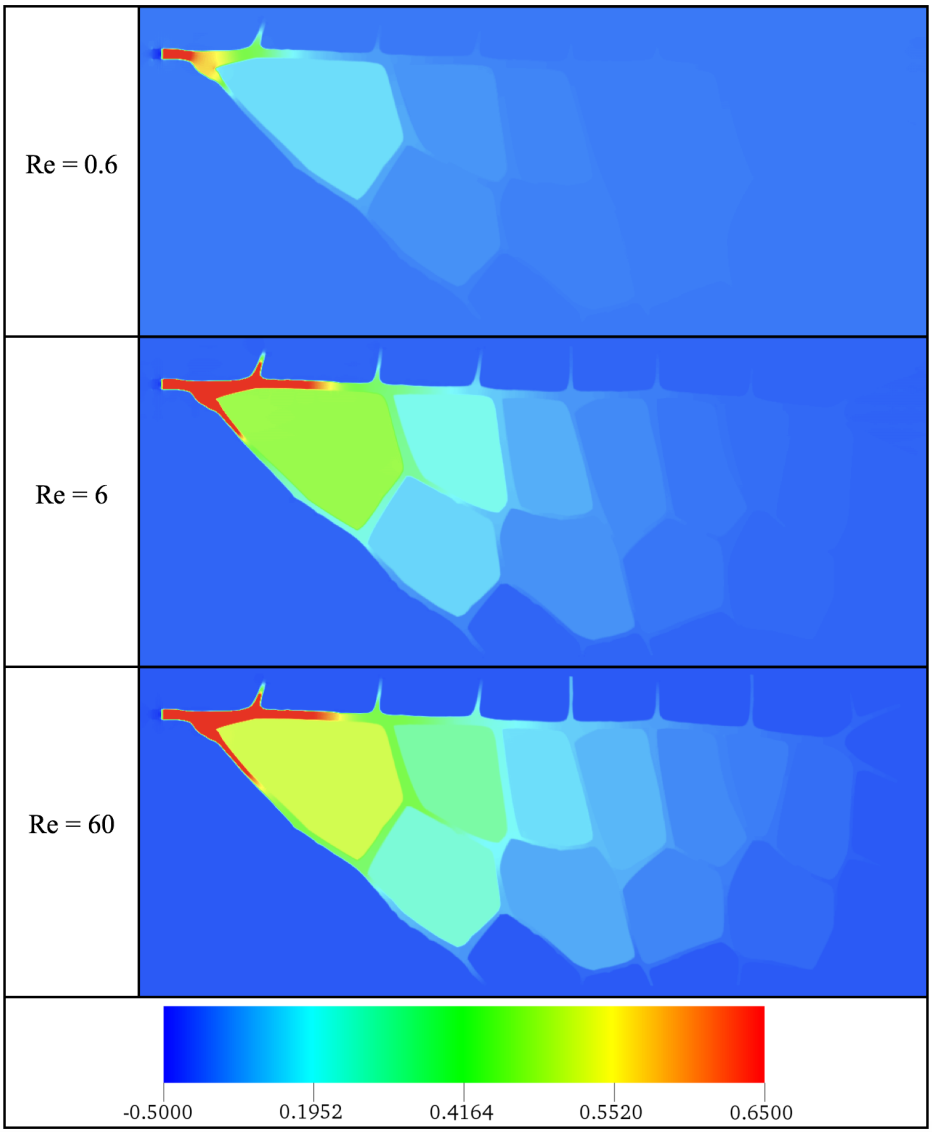

Figure 6: Snapshots comparing the pressure for different Re, $Re = 0.6, 6, 60$.

3.2. IBAMR: Lymphatic Capillary Example

As another example of how our software can be used, we present a case in which the vessel walls of a junction from a dermal lymphatic capillary are reconstructed from an image. This image is courtesy of Dr. Wenjing Xu from the Kathleen Caron lab (UNC-CH) and was taken from the back region of a wild type mouse embryo. The image was generated with fluorescence microscopy to highlight the lymphatic vessel boundaries as shown in Figure 7a. The simulations described below were performed using the immersed boundary method with adaptive mesh refinement (IBAMR) (See Appendix A).

After the contours were extracted using *ContourizeMe*, the ends of the vessels were extended using image software to allow the flow within the vessels to fully develop before reaching the actual vessel geometry. Parabolic outflow was prescribed as a boundary condi-

tion to effectively "pull" the fluid into the vessel ends with a maximum velocity of 10^{-5}m/s. This velocity is consistent with the reported range of observed lymphatic flow velocities, which are as low as 10^{-7} and as high as 10^{-3}m/s [61, 62]. Note that the vessel ends were placed at the left domain edge for this purpose. Neumann boundary conditions were used at the right edge of the domain to allow volume conservation (fluid escapes on this side), and periodic boundary conditions were used at the top and bottom of the domain. The vessel was assumed to be nearly rigid over the time scale of a simulation. The Navier-Stokes equations were discretized on a 512×512 grid with 3 levels of mesh refinement and a refinement ratio of 4. The fluid domain size was set to $L = 1.2 \times 10^{-3}$m, where the spatial step size was set to $\Delta x = L/512$. The vessel walls were described using a curvilinear mesh where the distance between immersed boundary points was set to $\Delta s = \Delta x/2$. The time step size was taken as $dt = 5.0 \times 10^{-6}$s. The lymph was parameterized with mass density $\rho = 1000$ kg/m^3 and dynamic viscosity $\mu = 10^{-3}$ Ns/m^2.

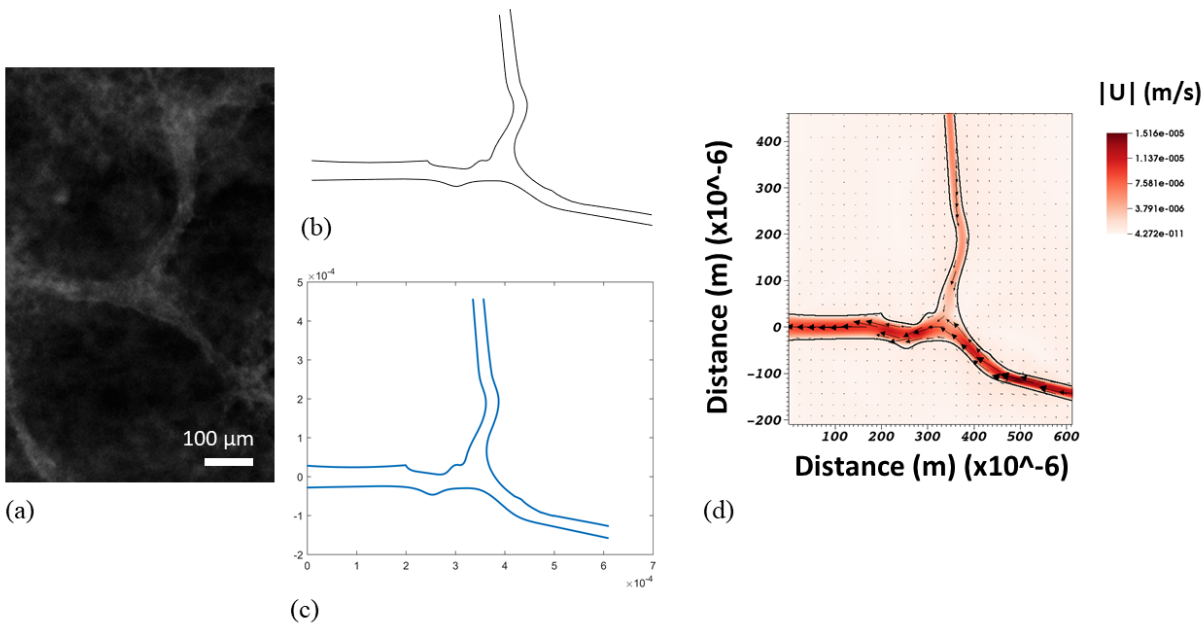

Figure 7: (a) Source image for modeling flow through a junction in a lymphatic capillary, courtesy of Dr. Wenjing Xu from the Kathleen Caron lab. (b) SVG rendering of bifurcating vascular structure. (c) Vertex points discretized from SVG file. (d) Colormap of the magnitude of velocity of flow through a bifurcating dermal lymphatic capillary. Note that the vessel ends were artificially extended to allow for fully developed flow within the vessels.

3.3. IB2d: Starfish Example

The last example we present is that of flow around a starfish using IB2d. The original JPG image of the starfish was taken from WikiMedia Commons, courtesy of NOAA Sea Grant Program in the The Coral Kingdom Collection [63]. The SVG file was produced by *ContourizeMe*, and the boundaries were discretized using *MeshmerizeMe* as shown in

Figure 8. Figures 8c, 8d, and 8f give the discretized geometries for the starfish at resolutions appropriate for an immersed boundary simulation in a square fluid domain with $L_x = L_y = 1$ where the spatial step size within such domain was set to $\Delta x = L_x/128$ (128×1238) and $\Delta x = L_x/256$ (256×256), and $\Delta x = L_x/1024$ (1024×1024) respectively. Note that in Figure 8f that the starfish's outline is still composed of discretized points. Recall that the average spacing between boundary points is set to one half of the spatial step size, e.g., $\Delta s = 0.5\Delta x$. Once the boundary describing the starfish is discretized at the desired resolution, it is placed inside of a rigid channel, where oscillatory flow will be prescribed to rush past the starfish as shown in Figure 8e.

We acknowledge that the starfish geometry is rather complex, and our simulation is relatively coarse. As such, we are likely not resolving the details of the flow very close to the starfish body. We would like to point out, however, that the purpose of this example is to further illustrate *MeshmerizeMe*'s ability to resolve and capture the fine structure detail of an SVG image. Thus, our goal in this example is not necessarily to resolve these fine scale flow structures. This example also highlights that once the geometry has been created for a single starfish, it can be easily altered. For example, this geometry can be copied, translated to different regions of the domain, or rotated since the software provides a parameterized set of points.

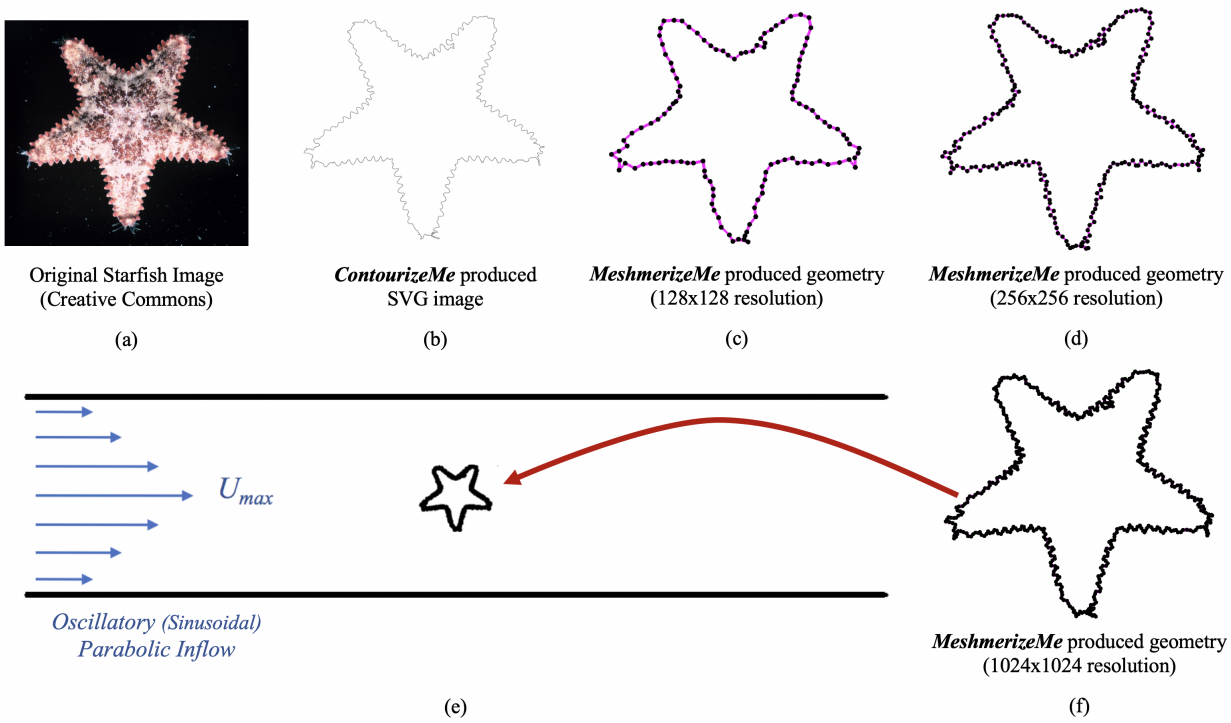

Figure 8: (a) Original Starfish image courtesy of NOAA Sea Grant Program [63] (b) SVG image of the starfish generated by *ContourizeMe* script (c), (d), (f) The discretized geometry at grid resolutions of 128×128, 256×256, and 1024×1024 respectively. (e) Computational geometry containing a starfish in a channel with prescribed oscillatory parabolic inflow (see Appendix A.1).

We ran a single starfish simulation at $Re = 800$ (see Eq. 5), where $L = 0.08\ m$, the height of the starfish, $V = 1.0\ m/s$, half the maximum oscillatory inflow speed, and ρ and μ are 1000 kg/m^2 and 0.1 $N \cdot s/m$, respectively. An oscillatory flow condition was used to produce flow past the starfish with frequency of $f = 2\ Hz$, see Appendix A.1. The numerical simulation was performed for a fluid domain with lengths $[0, 1] \times [0, 0.25]$ and a consistent spatial step size in each direction, e.g., $dx = 1.0/1024 = 0.25/256 = dy$. Snapshots from the numerical simulation that illustrate a colormap of vorticity are found in Figure 9.

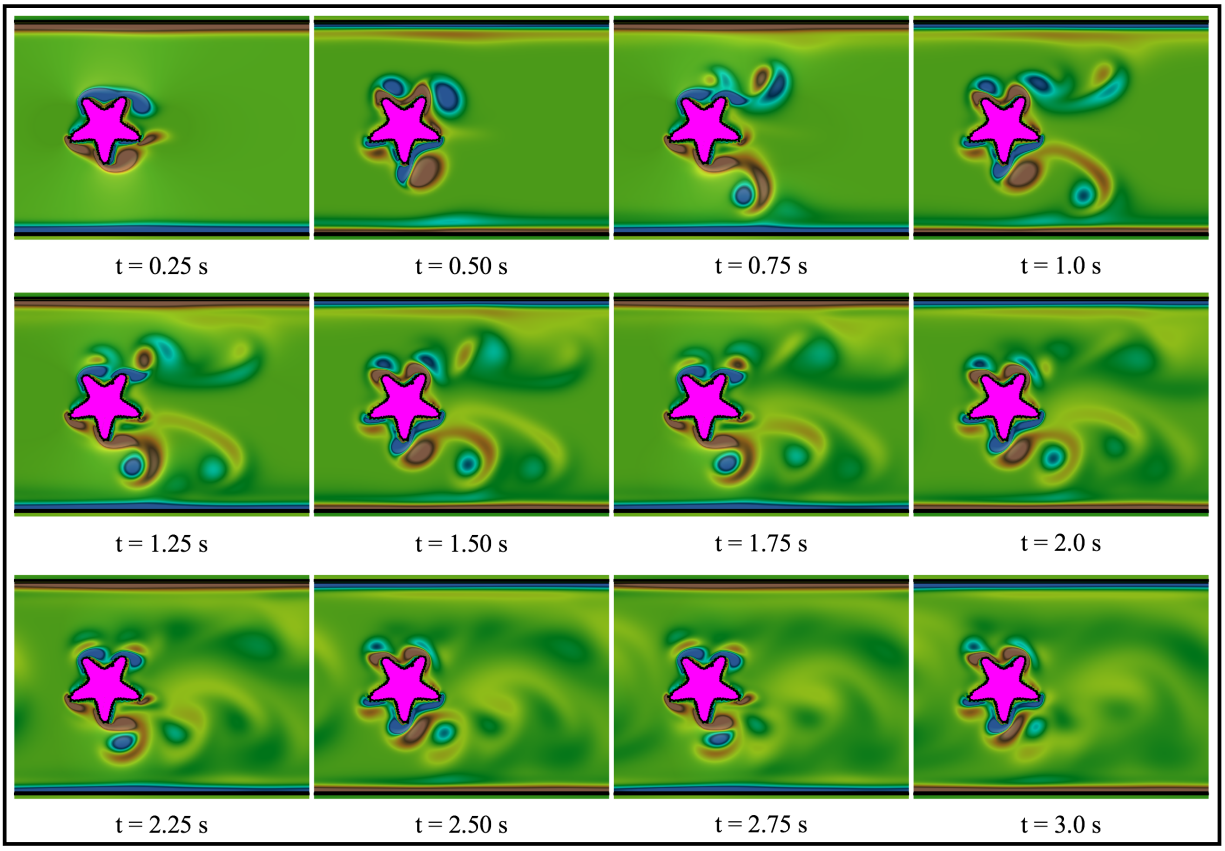

Figure 9: Snapshots showing oscillatory flow past a rigid starfish at $Re = 800$ during the first 6 pulsation periods. The background colormap illustrates vorticity.

Previously, one of the main difficulties in performing this immersed boundary simulation would be the finite difference discretization of the starfish. *MeshmerizeMe* provides a convenient way to do this, without having to manually piece together the geometry either by point-by-point construction or combining user-defined piecewise functions or splines. To demonstrate the versatility of this method, we insert multiple starfish into the channel. Figure 10 provides snapshots showing the vorticity during the first pulsation period of oscillatory flow around one, three, or five starfish within a channel. The example for flow around a single starfish can be found in the open source IB2d software's sub-directory, `IB2d/matIB2d/Examples/Example_MeshmerizeMe/Starfish/`.

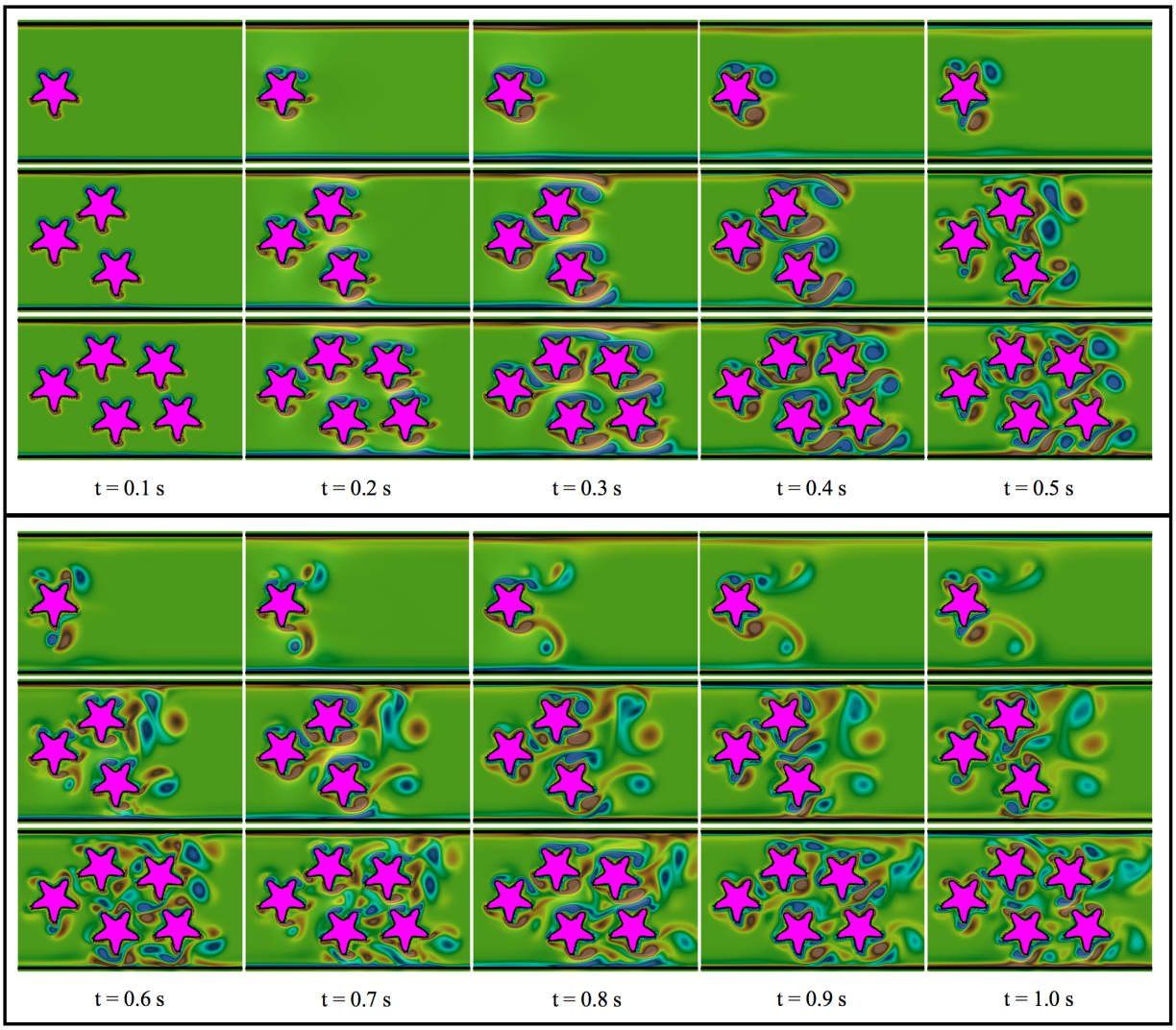

Figure 10: Snapshots showing oscillatory flow past 1, 3, or 5 rigid starfish at $Re = 800$ during the first pulsation period. The background colormap illustrates vorticity.

4. Discussion

In this paper, we introduce a software library that will extract edges from images, fit these images with Bézier curves, and discretize the curves into a curvilinear finite difference mesh with nearly constant spacing between points. Such meshes are useful in a variety of mathematical applications, including $2D$ numerical simulations of fluid-structure interaction (FSI) problems using the immersed boundary method. We present three such applications of the tool used in conjunction with the immersed boundary method including 1) flow of hemolymph in the veins of an insect wing (internal flow), 2) flow of lymph in a mouse lymphatic capillary (internal flow), and 3) flow of water around starfish (external flow). These images were taken with either high resolution digital cameras or fluorescence microscopy. Prior to this software release paper, the software was successfully applied in another internal

flow application of blood flows over the trabeculae in zebrafish embryonic hearts [64]. Here the meshes were created from images taken from an inverted (light) microscope [65]. This illustrates the software's robustness in its ability to construct discretized meshes from various imaging methods. Note that all of the above examples were performed using open-source implementations of the immersed boundary method, either IB2d [13, 12] or IBAMR [66].

While MeshmerizeMe merely provides 2D geometries, its output format can serve as a starting point in the development of 3D models. Commercial CAD software such as Fusion360 allow the import of SVG images, such as those produced by the ContourizeMe script, as sketches. Fusion360's built-in scripts allow the import of a CSV file describing a spline using XYZ coordinates. The latter is easily produced via commandline tools like `tail` and `sed` from the vertex files produced by the MeshmerizeMe script. This likewise imports the curves as a sketch. These imported sketches can then be turned into 3D objects using extrude and rotate commands. From here on, existing 3D meshing tools may be used to produce a mesh suitable for a 3D simulation.

To create simple 3D geometries, the 2D mesh could be extruded manually by adding a third coordinate, and this coordinate could be varied by Δs to obtain a finite difference mesh that describes an outer wall. Similarly, the 2D mesh could be rotated about a central axis to obtain another simple 3D geometry. Sample applications of these simple geometries could include wings or fins with constant cross sections and axisymmetric structures such as tubular hearts, jellyfish, and some worms. The meshes could also be used in other finite difference approaches to FSI problems, including the Method of Regularized Stokeslets [67], the immersed interface method [68], sharp interface methods [69, 70], or the blob projection method [71]. The software library could also be applied in the numerical simulation of other physics problems, including the uptake of particles [72] and electrodiffusion [73]. In future releases of the software library, we plan to add additional functionality that includes the automation for material property model input files, e.g., springs, beams, etc., for the geometry's discretized points.

In addition to use in research, this library may serve as a powerful tool for student research and education, particularly in mathematical modeling at the undergraduate and graduate levels. *MeshmerizeMe* provides students with open source tools that can easily be used to build relatively complicated $2D$ meshes from images. These boundary meshes are easily imported and used in IB2d and IBAMR, both of which are also open source libraries. One of the coauthors has developed a series of online videos to make the use of this software even easier for students[74]. Both IB2d and *MeshmerizeMe* have been used in the authors' undergraduate and graduate courses, including mathematical modeling, mathematical biology, numerical analysis, and a first year seminar on biological fluid dynamics. Furthermore, the libraries have been successfully used in numerous undergraduate research projects [75, 76] and contemporary locomotion research endeavors [77].

5. Acknowledgements

The authors would like to thank Charles Peskin for the development of immersed boundary method and Boyce Griffith for IBAMR, to which many of the input files structures of

IB2d are based. We would also like to thank Christina Battista, Robert Booth, Christina Hamlet, Alexander Hoover, Shannon Jones, Julia Samson, Arvind Santhanakrishnan, and Lindsay Waldrop for comments on the design of the software and suggestions for examples. This project was funded by NSF DMS CAREER #1151478, NSF CBET #1511427, NSF DMS #1151478, NSF POLS #1505061 awarded to L.A.M. and NSF IOS #1558052 awarded to Jake Socha. Computational resources for N.A.B. were provided by the NSF OAC #1826915 and the NSF OAC #1828163. Funding for N.A.B. was provided by the TCNJ Support of Scholarly Activity (SOSA) Grant, the TCNJ Department of Mathematics and Statistics, and the TCNJ School of Science.

Appendix A. Details on the Immersed Boundary Method

The two-dimensional formulation of the immersed boundary (IB) method used in this paper to study flow through dragonfly wing veins, lymphatic capillaries, and around starfish is provided below. *IB2d* [39, 12, 13] and *IBAMR* [40] are the two open source implementations that were used for these studies. For a full review of the immersed boundary method, please see Peskin [14].

Appendix A.1. Governing Equations of IB

The conservation of momentum equations that govern an incompressible and viscous fluid are listed below:

$$\rho\Big[\frac{\partial \mathbf{u}}{\partial t}(\mathbf{x},t) + (\mathbf{u}(\mathbf{x},t)\cdot\nabla)\mathbf{u}(\mathbf{x},t)\Big] = \nabla p(\mathbf{x},t) + \mu\Delta\mathbf{u}(\mathbf{x},t) + \mathbf{f}(\mathbf{x},t) \tag{A.1}$$

$$\nabla\cdot\mathbf{u}(\mathbf{x},t) = 0 \tag{A.2}$$

where $\mathbf{u}(\mathbf{x},t)$ is the fluid velocity, $p(\mathbf{x},t)$ is the pressure, $\mathbf{f}(\mathbf{x},t)$ is the force per unit area applied to the fluid by the immersed boundary, ρ and μ are the fluid's density and dynamic viscosity, respectively. The independent variables are the time t and the position $\mathbf{x}$. The variables $\mathbf{u}, p$, and $\mathbf{f}$ are all written in an Eulerian frame on the fixed Cartesian mesh, $\mathbf{x}$.

The interaction equations, which handle all communication between the fluid (Eulerian) grid and curvilinear mesh describing the immersed boundary (Lagrangian grid) are given by the following two integral equations:

$$\mathbf{f}(\mathbf{x},t) = \int \mathbf{F}(s,t)\delta\left(\mathbf{x}-\mathbf{X}(s,t)\right)ds \tag{A.3}$$

$$\mathbf{U}(s,t) = \int \mathbf{u}(\mathbf{x},t)\delta\left(\mathbf{x}-\mathbf{X}(s,t)\right)d\mathbf{x} \tag{A.4}$$

where $\mathbf{F}(s,t)$ is the force per unit length applied by the boundary to the fluid as a function of Lagrangian position, s, and time, t, $\delta(\mathbf{x})$ is a three-dimensional delta function, and $\mathbf{X}(s,t)$ gives the Cartesian coordinates at time t of the material point labeled by the Lagrangian parameter, s. The Lagrangian forcing term, $\mathbf{F}(s,t)$, gives the deformation forces along the boundary at the Lagrangian parameter, s. Equation (A.3) applies this force from

the immersed boundary to the fluid through the external forcing term in Equation (A.1). Equation (A.4) moves the boundary at the local fluid velocity. This enforces the no-slip condition. Each integral transformation uses a two-dimensional Dirac delta function kernel, δ, to convert Lagrangian variables to Eulerian variables and vice versa.

The way deformation forces are computed, e.g., the forcing term, $\mathbf{F}(s,t)$, in the integrand of Equation (A.3), is specific to the application. To hold the geometry nearly rigid, all of the Lagrangian points along the immersed boundary were tethered to target points. This has the effect of holding the boundary in place through a penalty forcing term where the force applied to the fluid is proportional to the difference between the actual location of the boundary and the desired location. In this model, the target force penalty term took the following form,

$$\mathbf{F}(s,t) = k_{targ}\left(\mathbf{Y}(s,t) - \mathbf{X}(s,t)\right), \tag{A.5}$$

where k_{targ} is a stiffness coefficient and $\mathbf{Y}(s,t)$ is the prescribed position of the target boundary. Note that $\mathbf{Y}(s,t)$ is a function of both the Lagrangian parameter, s, and time, t; however, in this model k_{targ} was chosen to be very large to minimize movement of the boundary.

For the case of the dragonfly wings, another penalty forcing term was used to prescribe the inflow conditions into the wing veins. This penalty force was applied directly onto the Eulerian (fluid) grid. The penalty force was proportional to the difference between the local fluid velocity and the desired fluid velocity and is given as

$$\mathbf{f}_{inflow} = k_{flow}\left(\mathbf{u}(\mathbf{x},t) - \mathbf{u}_{flow}(\mathbf{x},t)\right), \tag{A.6}$$

where k_{flow} is the penalty-strength coefficient, and $\mathbf{u}_{flow}(\mathbf{x},t)$ is the desired background flow profile as in [78]. For the simulations involving hemolymph flow through wing veins, we enforce the following parabolic inflow into the wing vein along the x-direction,

$$\mathbf{u}_{flow}(\mathbf{x},t) = \begin{cases} -U_{max}\tanh(2t)\left(\frac{(MP+w/2-y)(MP-w/2-y)}{w^2/4}\right) & \text{if inside prescribed region} \\ 0 & \text{elsewhere} \end{cases}, \tag{A.7}$$

where U_{max} is the desired max peak velocity in the parabolic inflow, MP is the midpoint of the vein and w is the width of the vein. Note that a hyperbolic tangent is used to ramp up the inflow during the course of the simulation.

Similarly, for the simulations of oscillatory flow past one or more starfish, we enforce the following oscillatory parabolic inflow into the starfish channel in the x-direction,

$$\mathbf{u}_{flow}(\mathbf{x},t) = \begin{cases} -U_{max}\sin(2\pi f t)\left(\frac{(MP+w/2-y)(MP-w/2-y)}{w^2/4}\right) & \text{if inside prescribed region} \\ 0 & \text{elsewhere} \end{cases}, \tag{A.8}$$

where f is the frequency of pulsation (set to 2 Hz) and the other parameters are analogous to before except with w and MP being the width and midpoint of the channel, respectively.

Using a regularlized delta function as the kernel in the interaction equations given by Eqs.(A.3-A.4) makes the immersed boundary method relatively easy to implement and flexible. To approximate these integrals, a discretized (and regularized) delta function was used.

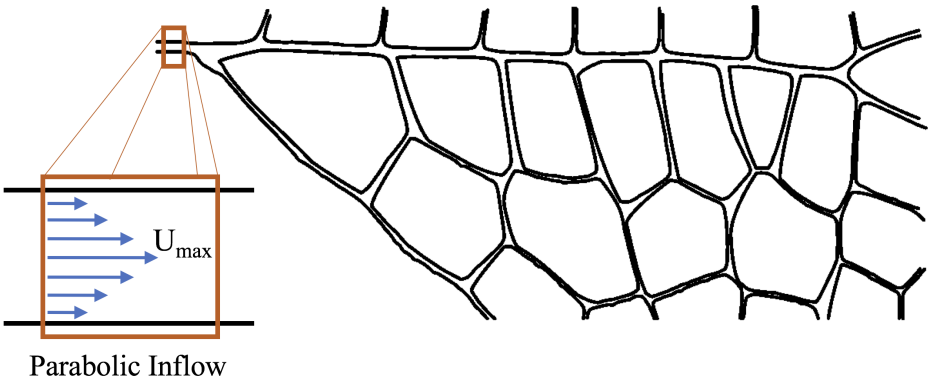

Figure A.11: Illustrating the subset of the insect vein geometry where the prescribed inflow condition is enforced.

In this paper, we use one described in [14], e.g., $\delta_h(\mathbf{x})$,

$$\delta_h(\mathbf{x}) = \frac{1}{h^3}\phi\left(\frac{x}{h}\right)\phi\left(\frac{y}{h}\right)\phi\left(\frac{z}{h}\right), \tag{A.9}$$

where $\phi(r)$ is defined as

$$\phi(r) = \begin{cases} \frac{1}{8}(3 - 2|r| + \sqrt{1 + 4|r| - 4r^2}), & 0 \le |r| < 1 \\ \frac{1}{8}(5 - 2|r| + \sqrt{-7 + 12|r| - 4r^2}), & 1 \le |r| < 2 \\ 0 & 2 \le |r|. \end{cases} \tag{A.10}$$

Appendix A.2. Numerical Algorithm

For the wing vein and starfish examples that use IB2d, we impose periodic and no slip boundary conditions on a rectangular domain. To solve Equations (A.1), (A.2),(A.3) and (A.4) we need to update the velocity, pressure, position of the boundary, and force acting on the boundary at time $n+1$ using data from time n. The IB does this in the following steps [14], with an additional step (4*b*) for IBAMR [79, 40]:

Step 1: Find the force density, $\mathbf{F}^n$ on the immersed boundary, from the current boundary configuration, $\mathbf{X}^n$.

Step 2: Use Equation (A.3) to spread this boundary force from the Lagrangian boundary mesh to the Eulerian fluid lattice points.

Step 3: Solve the Navier-Stokes equations, Equations (A.1) and (A.2), on the Eulerian grid. Upon doing so, we are updating $\mathbf{u}^{n+1}$ and p^{n+1} from $\mathbf{u}^n$, p^n, and $\mathbf{f}^n$. Note that a staggered grid projection scheme is used to perform this update.

Step 4:

4a. Update the material positions, $\mathbf{X}^{n+1}$, using the local fluid velocities, $\mathbf{U}^{n+1}$, using $\mathbf{u}^{n+1}$ and Equation (A.4).

4b. (*IBAMR* only) Refine Eulerian grid in areas of the domain that contain an immersed structure or where the vorticity exceeds a predetermined threshold, if on a selected time-step for adaptive mesh refinement.

Appendix B. Background on Bézier Curves

Bézier curves are a type of interpolating polynomial known as a spline. An nth degree Bézier polynomial may conveniently be written as a sum of $n+1$ weighted control points $\mathbf{P}_i$. The weights are known as Bernstein basis polynomials and take the form

$$b_{i,n}(t) = \binom{n}{i}(1-t)^{n-i}t^i.$$

A curve $\boldsymbol{\gamma}(t)$ may then be written as

$$\boldsymbol{\gamma}(t) = \sum_{i=0}^{n} \mathbf{P}_i \, b_{i,n}(t).$$

The parameter t is defined to be on the closed interval $[0,1]$. The derivative of a Bézier curve is itself a Bézier curve. Specifically, the first derivative is given by

$$\boldsymbol{\gamma}'(t) = n \sum_{i=0}^{n-1} (\mathbf{P}_{i+1} - \mathbf{P}_i) \, b_{i,n-1}(t).$$

Paths are modeled as a curve $\boldsymbol{\Gamma}(s)$ that is at least a C^0 sequence of curves, where $s = [0,1]$ is a parameter used to map to the individual Bézier curves that make up the path. If $\mathbf{P}_i^j$ is the ith control point of the jth curve in $\boldsymbol{\Gamma}$, then C^0 continuity translates into the requirement that

$$\mathbf{P}_n^j = \mathbf{P}_0^{j+1}.$$

To achieve C^1 continuity, we additionally require that

$$\mathbf{P}_n^j - \mathbf{P}_{n-1}^j = \mathbf{P}_1^{j+1} - \mathbf{P}_0^{j+1}.$$

Perhaps the most common Bézier curve in applications is the cubic Bézier which takes the explicit form

$$\boldsymbol{\gamma}(t) = (1-t)^3\mathbf{P}_0 + 3(1-t)^2 t\mathbf{P}_1 + 3(1-t)t^2\mathbf{P}_2 + t^3\mathbf{P}_3.$$

The explicit derivative of the cubic Bézier is given by

$$\boldsymbol{\gamma}'(t) = 3(1-t)^2(\mathbf{P}_1 - \mathbf{P}_0) + 6(1-t)t(\mathbf{P}_2 - \mathbf{1}) + 3t^2(\mathbf{P}_3 - \mathbf{P}_2).$$

To rescale a curve from one domain V to another domain U, an affine transform of the control points is sufficient. In the particular case $U, V \subset \mathbb{R}^2$ this transform may be represented as a simple matrix operator $A : \mathbb{R}^3 \to \mathbb{R}^3$ by representing a point $\vec{p} \in U, \vec{q} \in V$ in the form $(x, y, 1)^T$. Scaling and translating of control points may be achieved by the function

$$\vec{q} = \mathbf{A}\vec{p} = \begin{bmatrix} s_x & 0 & t_x \\ 0 & s_y & t_y \\ 0 & 0 & 1 \end{bmatrix} \vec{p}. \tag{B.1}$$

For our particular use case we want to map a point P_i from the SVG coordinate system $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$ to the coordinate system $[0, L_x] \times [0, L_y]$ used in the simulations. Note that the origin of the SVG coordinates is in the upper left-hand corner of the image, while the coordinate system used in IB2d and IBAMR has its origin in the lower left hand corner. Accounting for this and letting $w = x_{\max} - x_{\min}$ and $h = y_{\max} - y_{\min}$ our operator will be defined as

$$\begin{bmatrix} \frac{L_x}{w} & 0 & -\frac{L_x}{w} x_{\min} \\ 0 & -\frac{L_y}{h} & L_y \\ 0 & 0 & 1 \end{bmatrix} \tag{B.2}$$

References

[1] C. Peskin, Flow patterns around heart valves: A numerical method, J. Comput. Phys. 10(2) (1972) 252–271.

[2] L. A. Miller, C. S. Peskin, When vortices stick: an aerodynamic transition in tiny insect flight, J. Exp. Biol. 207 (2004) 3073–3088.

[3] L. A. Miller, C. S. Peskin, A computational fluid dynamics of clap and fling in the smallest insects, J. Exp. Biol. 208 (2009) 3076–3090.

[4] S. K. Jones, R. Laurenza, T. L. Hedrick, B. E. Griffith, L. A. Miller, Lift- vs. drag-based for vertical force production in the smallest flying insects, J. Theor. Biol. 384 (2015) 105–120.

[5] E. Tytell, C. Hsu, T. Williams, A. Cohen, L. Fauci, Interactions between internal forces, body stiffness, and fluid environment in a neuromechanical model of lamprey swimming, Proc. Natl. Acad. Sci. 107 (2010) 19832–19837.

[6] E. Tytell, C. Hsu, L. Fauci, The role of mechanical resonance in the neural control of swimming in fishes, Zoology 117 (2014) 48–56.

[7] A. P. Hoover, L. A. Miller, A numerical study of the benefits of driving jellyfish bells at their natural frequency, J. Theor. Biol. 374 (2015) 13–25.

[8] G. Hershlag, L. A. Miller, Reynolds number limits for jet propulsion: a numerical study of simplified jellyfish, J. Theor. Biol. 285 (2011) 84–95.

[9] B. E. Griffith, X. Luo, D. M. McQueen, C. S. Peskin, Simulating the fluid dynamics of natural and prosthetic heart valves using the immersed boundary method, Int. J. Appl. Mech. 1(1) (2009) 137–177.

[10] E. Jung, C. Peskin, 2-d simulations of valveless pumping using immersed boundary methods, SIAM Journal on Scientific Computing 23 (2001) 19–45.

[11] D. McQueen, C. S. Peskin, Shared-memory parallel vector implementation of the immersed boundary method for the computation of blood flow in the beating mammalian heart, J. Supercomputing 11 (1997) 213–236.

[12] N. A. Battista, W. C. Strickland, L. A. Miller, IB2d: a Python and MATLAB implementation of the immersed boundary method, Bioinspir. Biomim. 12(3) (2017) 036003.

[13] N. A. Battista, W. C. Strickland, A. Barrett, L. A. Miller, IB2d Reloaded: a more powerful Python and MATLAB implementation of the immersed boundary method, Math. Method. Appl. Sci 41 (2018) 8455–8480.

[14] C. S. Peskin, The immersed boundary method, Acta Numerica 11 (2002) 479–517.

[15] L. Zhu, G. He, S. Wang, L. A. Miller, X. Zhang, Q. You, S. Fang, An immersed boundary method by the lattice boltzmann approach in three dimensions, Computers and Mathematics with Applications 61 (2011) 3506–3518.
[16] L. Zhu, C. S. Peskin, Simulation of a flapping flexible filament in a flowing soap film by the immersed boundary method, J. Comp. Phys. 179 (2002) 452–468.
[17] J. Ryu, S. G. Park, B. Kim, H. Jinsung, Flapping dynamics of an inverted flag in a uniform flow, J. Fluids and Structures 57 (2015) 159–169.
[18] A. J. Baird, T. King, L. A. Miller, Numerical study of scaling effects in peristalsis and dynamic suction pumping, Biological Fluid Dynamics: Modeling, Computations, and Applications 628 (2014) 129–148.
[19] L. D. Waldrop, L. A. Miller, Large-amplitude, short-wave peristalsis and its implications for transport, Biomechanics and Modeling in Mechanobiology (2015) 1–14.
[20] Y. Kim, C. S. Peskin, Penalty immersed boundary method for an elastic boundary with mass, Physics of Fluids 19 (2007) 053103.
[21] J. M. Stockie, Modelling and simulation of porous immersed boundaries, Computers and Structures 87 (2009) 701–709.
[22] Y. Kim, C. S. Peskin, 2d parachute simulation by the immersed boundary method, SIAM J. Sci. Comput. 28 (2006) 2294–2312.
[23] D. S. Lee, M. Y. Ha, S. J. Kim, H. S. Yoon, Application of immersed boundary method for flow over stationary and oscillating cylinders, J. Mech. Sci. and Tech. 20(6) (2006) 849–863.
[24] A. Pinelli, I. Z. Nagavi, U. Piomelli, J. Favier, Immersed-boundary methods for general finite-difference and finite-volume navier-stokes solvers, J. Comp. Phys 229(24) (2010) 9073–9091.
[25] D. C. Lo, C. P. Lee, I. F. Lin, An efficient immersed boundary method for fluid flow simulations with moving boundaries, Appl. Math. and Comp. 328(1) (2018) 312–337.
[26] R. Campregher, J. Militzer, S. S. Mansur, N. Silveira, A. da Silveira Neto, Computations of the flow past a still sphere at moderate reynolds numbers using an immersed boundary method, J. Braz. Soc. Mech. Sci. & Eng. 31(4) (2009) 333–352.
[27] W. C. Strickland, L. A. Miller, A. Santhanakrishnan, C. Hamlet, V. P. N. A. Battista, Three-dimensional low reynolds number flows near biological filtering and protective layers, Fluids 2(4) (2017) 62.
[28] C. S. Peskin, D. M. McQueen, Fluid dynamics of the heart and its valves, in: F. R. Adler, M. A. Lewis, J. C. Dalton (Eds.), Case Studies in Mathematical Modeling: Ecology, Physiology, and Cell Biology, Prentice-Hall, New Jersey, 1996, Ch. 14, pp. 309–338.
[29] N. A. Battista, A. N. Lane, J. Liu, L. A. Miller, Fluid dynamics of heart development: Effects of trabeculae and hematocrit, Math. Med. & Biol. 35(4) (2018) 493–516.
[30] T. J. Wilson, Simultaneous untangling and smoothing of hexahedral meshes (masters thesis), Universitat Politečnica de Catalunya.
[31] P. Cignoni, M. Callieri, M. Corsini, M. Dellepiane, F. Ganovelli, G. Ranzuglia, MeshLab: an Open-Source Mesh Processing Tool, in: V. Scarano, R. D. Chiara, U. Erra (Eds.), Eurographics Italian Chapter Conference, The Eurographics Association, 2008, pp. 129–136. doi:10.2312/LocalChapterEvents/ItalChap/ItalianChapConf2008/129-136.
[32] C. Geuzaine, J. F. Remacle, Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities, Int. J. Num. Meth. Eng. 79(11) (2009) 1309–1331.
[33] S. Hang, Tetgen, a delaunay-based quality tetrahedral mesh generator, ACM Trans. on Math. Soft. 41(2) (2015) 1–36.
[34] M. K. Berens, I. D. Flintoft, J. F. Dawson, Structured mesh generation: Open-source automatic nonuniform mesh generation for fdtd simulation, IEEE Antennas & Propagation Magazine 58(3) (2016) 45–55.
[35] Argus Holdings, LTD, Argus one: Open numerical environments (2015).
URL `http://www.argusone.com/index.html`
[36] C. I. Voss, D. Boldt, A. M. Shapiro, A graphical-user interface for the u.s. geological survey's sutra code using argus one (for simulation of variable-density saturated-unsaturated ground-water flow with solute or energy transport), U.S. Geological Survey Open-File Report 1 (1997) 97–421.

[37] N. A. Battista, Fluid-structure interaction for the classroom: Interpolation, hearts, and swimming!, arXiv: https://arxiv.org/abs/1808.08122.
[38] N. A. Battista, M. S. Mizuhara, Fluid-structure interaction for the classroom: Speed, accuracy, convergence, and jellyfish!, arXiv: https://arxiv.org/abs/1902.07615.
[39] N. A. Battista, A. J. Baird, L. A. Miller, A mathematical model and matlab code for muscle-fluid-structure simulations, Integr. Comp. Biol. 55(5) (2015) 901–911.
[40] B. E. Griffith, An adaptive and distributed-memory parallel implementation of the immersed boundary (ib) method (2014) [cited October 21, 2014].
URL `https://github.com/IBAMR/IBAMR`
[41] C. Hamlet, L. A. Miller, Feeding currents of the upside-down jellyfish in the presence of background flow, Bull. Math. Bio. 74(11) (2012) 2547–2569.
[42] C. Zhang, R. D. Guy, B. Mulloney, Q. Zhang, T. J. Lewis, Neural mechanism of optimal limb coordination in crustacean swimming, Proceedings of the National Academy of Sciences 111 (38) (2014) 13840–13845. arXiv:https://www.pnas.org/content/111/38/13840.full.pdf, doi:10.1073/pnas.1323208111.
URL `https://www.pnas.org/content/111/38/13840`
[43] L. A. Miller, A. Santhanakrishnan, S. K. Jones, C. Hamlet, K. Mertens, L. Zhu, Reconfiguration and the reduction of vortex-induced vibrations in broad leaves, J. Exp. Biol. 215 (2012) 2716–2727.
[44] C. Hamlet, W. Strychalski, L. Miller, Dynamics of ballistic strategies in nematocyst firing, Fluids 5.
[45] L. M. Crowl, A. L. Fogelson, Computational model of whole blood exhibiting lateral platelet motion induced by red blood cells, Int. J. Numer. Meth. Biomed. Engng. 26 (2009) 471–487.
[46] L. M. Crowl, A. L. Fogelson, Analysis of mechanisms for platelet near-wall excess under arterial blood flow conditions, J. Fluid Mech. 676 (2011) 348–375.
[47] Svg optimizer is a nodejs-based tool for optimizing svg vector graphics files, `https://github.com/svg/svgo`, accessed: 2019-06-25.
[48] J. Archibald, Svgomg is svgo's missing gui, aiming to expose the majority, if not all the configuration options of svgo, `https://jakearchibald.github.io/svgomg/`, accessed: 2019-06-25.
[49] J. Long, E. Shelhamer, T. Darrell, Fully convolutional networks for semantic segmentation, in: 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015, pp. 3431–3440. doi:10.1109/CVPR.2015.7298965.
[50] A. Garcia-Garcia, S. Orts-Escolano, S. Oprea, V. Villena-Martinez, P. Martinez-Gonzalez, J. Garcia-Rodriguez, A survey on deep learning techniques for image and video semantic segmentation, Applied Soft Computing 70 (2018) 41 – 65. doi:https://doi.org/10.1016/j.asoc.2018.05.018.
URL `http://www.sciencedirect.com/science/article/pii/S1568494618302813`
[51] G. Bradski, The OpenCV Library, Dr. Dobb's Journal of Software Tools.
[52] R. C. Gonzalez, R. E. Woods, Digital Image Processing, 2nd Edition, Prentice Hall, Englewood, Cliffs, NJ, 2002.
[53] A. Bovick, Handbook of Image and Video processing, Academic Press, New York, NY, 2000.
[54] C. Tomasi, R. Manduchi, Bilateral filtering for gray and color images, in: Proceedings of the Sixth International Conference on Computer Vision, ICCV '98, IEEE Computer Society, Washington, DC, USA, 1998, pp. 839–.
URL `http://dl.acm.org/citation.cfm?id=938978.939190`
[55] O. Ronneberger, P. Fischer, T. Brox, U-net: Convolutional networks for biomedical image segmentation, CoRR abs/1505.04597. arXiv:1505.04597.
URL `http://arxiv.org/abs/1505.04597`
[56] G. T. Berge, O. Granmo, T. O. Tveit, M. Goodwin, L. Jiao, B. V. Matheussen, Using the tsetlin machine to learn human-interpretable rules for high-accuracy text categorization with medical applications, CoRR abs/1809.04547. arXiv:1809.04547.
URL `http://arxiv.org/abs/1809.04547`
[57] J. Lehtinen, J. Munkberg, J. Hasselgren, S. Laine, T. Karras, M. Aittala, T. Aila, Noise2noise: Learning image restoration without clean data, CoRR abs/1803.04189. arXiv:1803.04189.
URL `http://arxiv.org/abs/1803.04189`

[58] X. Wang, D. Huang, H. Xu, An efficient local chanvese model for image segmentation, Pattern Recognition 43 (3) (2010) 603 – 618. doi:https://doi.org/10.1016/j.patcog.2009.08.002.

[59] P. Kratochvil, Insect wing structure: Macro photo of a dragonfly wing structure, [Camera: Canon EOS 5D Mark II 1/160s, f 16.0, ISO 100, 100 mm; uploaded September 26, 2013; accessed February 22, 2018] (2013).
URL `http://www.publicdomainpictures.net/view-image.php?image=25113&large=1&picture=insect-wing-structure`

[60] Gnu image manipulation program, `https://www.gimp.org/`, accessed: 2019-08-21.

[61] C. D. Bertram, C. Macaskill, J. E. J. Moore, An improved model of an actively contracting lymphatic vessel composed of several lymphangions: pumping characteristics, OALib.

[62] J. B. DIXON, S. T. GREINER, A. A. GASHEV, G. L. COTE, J. E. MOORE Jr., D. C. ZAWIEJA, Lymph flow, shear stress, and lymphocyte velocity in rat mesenteric prenodal lymphatics, Microcirculation 13 (7) (2006) 597–610. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1080/10739680600893909, doi:10.1080/10739680600893909.
URL `https://onlinelibrary.wiley.com/doi/abs/10.1080/10739680600893909`

[63] J. P. McVey, Reef0297.jpg, `https://commons.wikimedia.org/wiki/File:Reef0297.jpg`, NOAA Sea Grant Program: The Coral Kingdom Collection; Accessed: 2019-06-25 (2007).

[64] N. A. Battista, D. R. Douglas, A. N. Lane, L. A. Samsa, J. Liu, L. A. Miller, Vortex dynamics in trabeculated embryonic ventricles, J. Cardiovasc. Dev. and Dis. 6 (2019) 6.

[65] J. Liu, M. Bressan, D. Hassel, J. Huisken, D. Staudt, K. Kikuchi, K. Poss, T. Mikawa, D. Stainier, A dual role for erbb2 signaling in cardiac trabeculation, Development 137 (2010) 3867–3875.

[66] An adaptive and distributed-memory parallel implementation of the immersed boundary (ib) method, `https://github.com/IBAMR/IBAMR`, accessed: 2019-06-18.

[67] R. Cortez, The method of regularized stokeslets, SIAM J. Sci. Comput. 23(4) (2001) 12041225.

[68] Z. Li, An overview of the immersed interface method and its applications, Taiwanese J. Math. 7(1) (2003) 1–49.

[69] O. Ubbink, R. I. Issa, Method for capturing sharp fluid interfaces on arbitrary meshes, J. Comp. Phys. 153 (1999) 26–50.

[70] H. Udaykumar, R. Mittal, P. Rampunggoon, A. Khanna, A sharp interface cartesian grid method for simulating flows with complex moving boundaries, J. Comp. Phys. 20 (2001) 345–380.

[71] R. Cortez, M. Minion, The blob projection method for immersed boundary problems, J. Comp. Phys. 161 (2000) 428–453.

[72] L. D. Waldrop, L. A. Miller, S. Khatri, A tale of two antennules: the performance of crab odour-capture organs in air and water, J. R. Soc. Interface 13. doi:http://doi.org/10.1098/rsif.2016.0615.

[73] P. Lee, B. E. Griffith, C. S. Peskin, The immersed boundary method for advection-electrodiffusion with implicit timestepping and local mesh refinement, J Comput Phys 229 (2010) 52085227. doi:10.1016/j.jcp.2010.03.036.

[74] N. A. Battista, Ib2d video tutorials!, `https://github.com/nickabattista/IB2d/`, accessed: 2019-06-18.

[75] J. G. Miles, N. A. Battista, Don't be jelly: Exploring effective jellyfish locomotion, arXiv: https://arxiv.org/abs/1904.09340.

[76] J. G. Miles, N. A. Battista, Naut your everyday jellyfish model: Exploring how tentacles and oral arms impact locomotion, Fluids 4(3) (2019) 169.

[77] F. Pallasdies, S. Goedeke, W. Braun, R. Memmesheimer, From single neurons to behavior in the jellyfish *Aurelia aurita*, biorXiv: https://www.biorxiv.org/content/10.1101/698548v1Https://doi.org/10.1101/698548.

[78] A. Santhanakrishnan, N. Nguyen, J. Cox, L. A. Miller, Flow within models of the vertebrate embryonic heart, J. Theor. Biol. 259 (2009) 449–461.

[79] B. E. Griffith, Simulating the blood-muscle-vale mechanics of the heart by an adaptive and parallel version of the immsersed boundary method (ph.d. thesis), Courant Institute of Mathematics, New York

University.