FLUID-STRUCTURE INTERACTION FOR THE CLASSROOM: INTERPOLATION, HEARTS, AND SWIMMING! [*]

NICHOLAS A. BATTISTA[†]

Abstract.
While students may find spline interpolation quite digestible, based on their familiarity with continuity of a function and its derivatives, some of its inherent value may be missed when students only see it applied to standard data interpolation exercises. In this paper, we offer alternatives where students can qualitatively and quantitatively witness the resulting dynamical differences when objects are driven through a fluid using different spline interpolation methods. They say, *seeing is believing*; here we showcase the differences between linear and cubic spline interpolation using examples from fluid pumping and aquatic locomotion. Moreover, students can define their own interpolation functions and visualize the dynamics that unfold. To solve the fluid-structure interaction system, the open-source fluid dynamics software *IB2d* is used. In that vein, all simulation codes, analysis scripts, and movies are provided for streamlined use.

Key words. Numerical Analysis Education, Fluid Dynamics Education, Mathematical Biology Education, Immersed Boundary Method, Fluid-Structure Interaction, Biological Fluid Dynamics

AMS subject classifications. 65D05, 65D07, 97M10, 97M60, 97N40, 97N50, 97N80, 76M25, 76Z10, 76Z99, 92C10

1. Introduction.

Traditionally it is in numerical analysis and scientific computing courses where students are first introduced to the topic of interpolation. It is frequently motivated by posing the seemingly innocent question of, "If handed N unique data points, $\{x_j, y_j\}_{j=0}^{N}$, can you find a polynomial, $p(x)$, with the property that $p(x_j) = y_j, \forall j = 0, 1, 2, \ldots, N$?" It is customary to accompany this question with a uniqueness theorem that gives a somewhat surprising result for students - that if such a polynomial exists, it must be unique. The proof is even rather elegant [21, 14]!

What happens next? Well, surely a discussion of how to construct such a polynomial and alas the standard ways to find such an interpolation polynomial (monomial, Newton, and Lagrange) are derived. This effort, in essence, enforces that students once again see that existence and uniqueness go together, like peas and carrots.

This may leave the students usually wondering, "Well, how close is this polynomial to the actual function from which the data was originally sampled?" Not be disappointed, the class dives into estimating the error of such a polynomial, and after seeing a few exploitative examples using uniformly spaced nodes [32, 21, 18, 14], and going down the rabbit hole of Chebyshev nodes, students see the corresponding interpolation error and how it can be minimized.

If that is the best such a polynomial can do in terms of minimizing the error, instructors may encourage their class to contemplate whether there could be any other methods to interpolate the original data given. That is, motivating the students to move beyond constructing a single global polynomial that interpolates the data, but instead interpolating the data point-by-point. This, of course, leads to the introduction of spline interpolation, cubic splines, and/or Bezier curves! Splendid!

Unfortunately, a genuine difficultly for students during this onslaught of interpolation techniques, error analysis, and implementation, is sometimes seeing the practical

[*]Submitted to the editors 22/08/2018.

Funding: This work was funded by the NSF OAC-1828163 the Support of Scholarly Activities Grant from TCNJ (The College of New Jersey)

[†]Department of Mathematics and Statistics, The College of New Jersey, 2000 Pennington Road, Ewing Township, NJ 08628 (battistn@tcnj.edu, http://http://battistn.pages.tcnj.edu/).

applications of interpolation. Some possible (surprising) applications for students that may be mentioned include how letters are shaped in typography [1, 37], vector graphics and imaging [36], or data and digital signal processing [35, 23]. However, students generally interested in computational science and modeling may not be captivated or satisfied with these applications.

We would like to introduce an application of interpolation that unfortunately falls through the cracks for students - the use of interpolation in mathematical modeling, and in particular biological fluid dynamics. Simply stated interpolation can be used to prescribe the motion of objects. The enticing portion - these objects can be immersed within a fluid, where the fluid reacts and moves due to the prescribed motion of said object.

Not sold, yet? Numerous recent scientific studies have used this exact type of interpolation to successfully prescribe motion, ranging from diverse fields such as heart development [3, 22, 6], aquatic locomotion [19, 2, 12], animal flight [27, 31, 20], organismal feeding and filtering [17, 28, 33], and beyond.

We offer a software alternative that will allow students to test out varying kinds of spline interpolation to prescribe the motion between one or more feature states, within a framework that provides direct practical scientific applications.

In the remainder of this paper, we will provide three differing examples of how spline interpolation can be used to drive the motion of a structure immersed within a fluid, while also comparing different kinds of spline interpolation, e.g., linear and higher order polynomial (cubic). This will provide students intuition about splines that is not traditionally emphasized in the classroom that can help facilitate greater learning and further curiosity in computational science.

In Section 2 we motivate the ideas of spline interpolation through the presentation of a moving circular object immersed in a fluid. In Section 3 we introduce how to prescribe motion using a cartoon heart pumping example and provide a stencil for how to create your own example. In Section 4 we move beyond prescribing the motion of individual points to instead interpolate between different material property states of an immersed body, e.g., modeling a structure that has time-dependent curvature, which gives rise to forward locomotion (swimming)! For details regarding the fluid-structure interaction software, see Appendix A, or [4, 10, 9] for a more detailed overview. All simulations presented here are available on https://github.com/nickabattista/ib2d and can found in the sub-directory `IB2d/matIB2d/Examples/Examples_Education/` as well as the Supplementary Materials.

2. Spline Interpolation: Linear vs. Higher Order Polynomials.

When first introducing splines in numerical analysis, it may fruitful to tell students they have already seen an example of a linear spline in Multivariate Calculus, when parameterizing curves for line integrals. Have them consider two points, $\mathbf{a}$ and $\mathbf{b}$, (x_a, y_a) and (x_b, y_b), respectively. Students can then parameterize a straight line between the two points in a familiar way:

$$(x(t), y(t)) = \mathbf{h}_0(t) = \mathbf{a} + \frac{t}{t_1}(\mathbf{b} - \mathbf{a}), \tag{2.1}$$

for $t \in [0, t_1]$. We can see that $\mathbf{h}_0(0) = \mathbf{a}$ and $\mathbf{h}_0(t_1) = \mathbf{b}$. Of course, in calculus this is not introduced as a spline and the word interpolation probably doesn't echo off the classroom walls, but that is exactly what this process was - setting up a linear spline interpolant between two points. If we had a third point $\mathbf{c} = (x_c, y_c)$, we could construct another linear interpolant between the $\mathbf{b}$ and $\mathbf{c}$,

$$(x(t), y(t)) = \mathbf{h}_1(t) = \mathbf{b} + \frac{t - t_1}{t_2 - t_1}(\mathbf{c} - \mathbf{b}), \tag{2.2}$$

for $t \in [t_1, t_2]$. We note that $\mathbf{h}_1(t_1) = \mathbf{b}$ and $\mathbf{h}_1(t_2) = \mathbf{c}$. The piecewise linear interpolant between all three points could then be written as

$$(x(t), y(t)) = \begin{pmatrix} \mathbf{h}_0(t) \\ \mathbf{h}_1(t) \end{pmatrix} = \begin{cases} \mathbf{a} + \frac{t}{t_1}(\mathbf{b} - \mathbf{a}) & 0 \le t \le t_1 \\ \mathbf{b} + \frac{t - t_1}{t_2}(\mathbf{c} - \mathbf{b}) & t_1 \le t \le t_2 \end{cases}. \tag{2.3}$$

What we have done, although perhaps not emphasized too much in Calculus, is created a method to prescribe the motion of a point, $\mathbf{x}$ around the plane in $\mathbb{R}^2$,

$$\mathbf{a} \to \mathbf{b} \to \mathbf{c}.$$

There is no reason this cannot extend to a larger collection of points! Instead of points $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$, consider the following matrices, where each column contains N-(x, y) points, respectively,

$$\mathbf{A} = \begin{bmatrix} x_0^a & y_0^a \\ x_1^a & y_1^a \\ \vdots & \vdots \\ x_N^a & y_N^a \end{bmatrix}, \ \mathbf{B} = \begin{bmatrix} x_0^b & y_0^b \\ x_1^b & y_1^b \\ \vdots & \vdots \\ x_N^b & y_N^b \end{bmatrix}, \ \text{and} \ \mathbf{C} = \begin{bmatrix} x_0^c & y_0^c \\ x_1^c & y_1^c \\ \vdots & \vdots \\ x_N^c & y_N^c \end{bmatrix}. \tag{2.4}$$

We can write an analogous spline interpolant to (2.3) as follows,

$$(\mathbf{x}(t), \mathbf{y}(t)) = \begin{pmatrix} \mathbf{H}_0(t) \\ \mathbf{H}_1(t) \end{pmatrix} = \begin{cases} \mathbf{A} + \frac{t}{t_1}(\mathbf{B} - \mathbf{A}) & 0 \le t \le t_1 \\ \mathbf{B} + \frac{t - t_1}{t_2}(\mathbf{C} - \mathbf{B}) & t_1 \le t \le t_2 \end{cases}. \tag{2.5}$$

EXAMPLE 2.1. *Consider the circles given by the following N points $\{x_j^a, y_j^a\}_{j=0}^N$, $\{x_j^b, y_j^b\}_{j=0}^N$ and $\{x_j^c, y_j^c\}_{j=0}^N$. These are illustrated in Figure 1.*

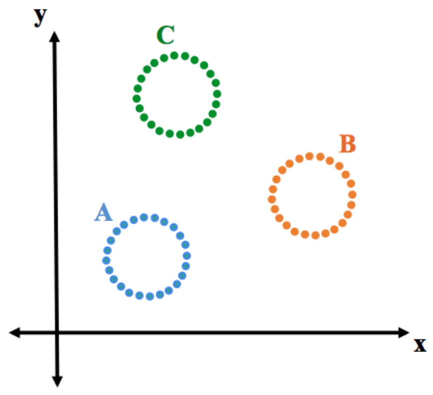


FIG. 1. 3 *circles in the xy plane, each composed of N points.*

*Next using (2.5), let's prescribe the motion of these circles starting from State **A** to State **B** and finally State **B** to State **C** for $0 \le t \le t_2$, with $t_1 \in (0, t_2)$ The* *positions, $(\boldsymbol{x}(t), \boldsymbol{y}(t))$ of these interpolated states are illustrated in Figure 2, given by the circle in red.*

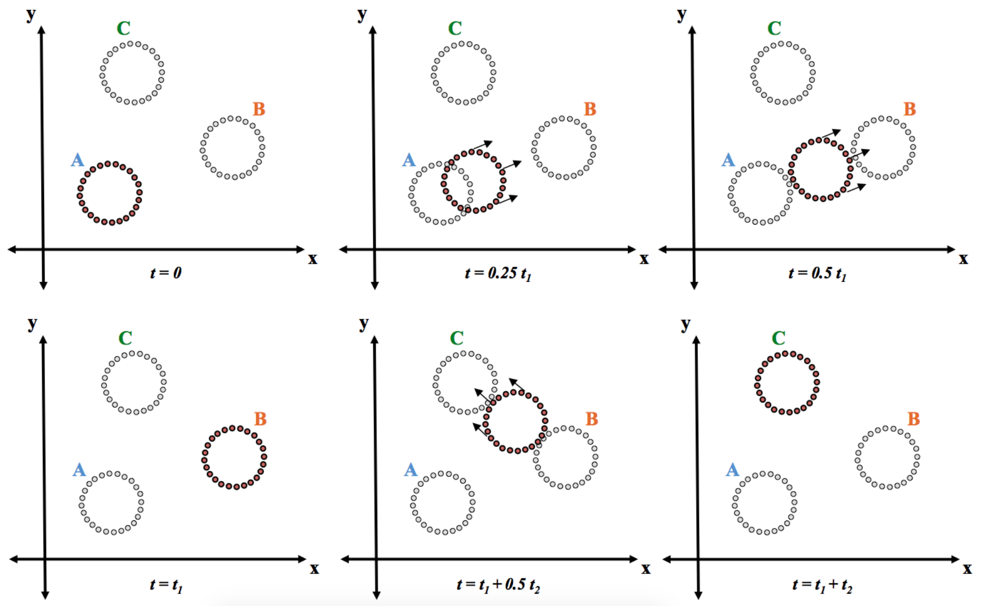


FIG. 2. *Timeslices of the N-point circle moving from State* ***A*** *to State* ***B*** *to State* ***C*** *using the piecewise linear interpolate to prescribe the motion.*

As mentioned earlier, we could imagine that beyond these circles simply moving around the xy-plane in a prescribed fashion, one could envision these objects immersed within a fluid. This is exactly an example found in IB2d, e.g., `Examples_Education/Interpolation/Moving_Circle/Linear_Interp`. *Immersing a circle within a fluid environment and then prescribing its motion will cause the fluid to react, and in turn, move in response. This is shown in Figure 3, where the colormap illustrates the magnitude of the fluid velocity and vector field represents the fluid velocity.*

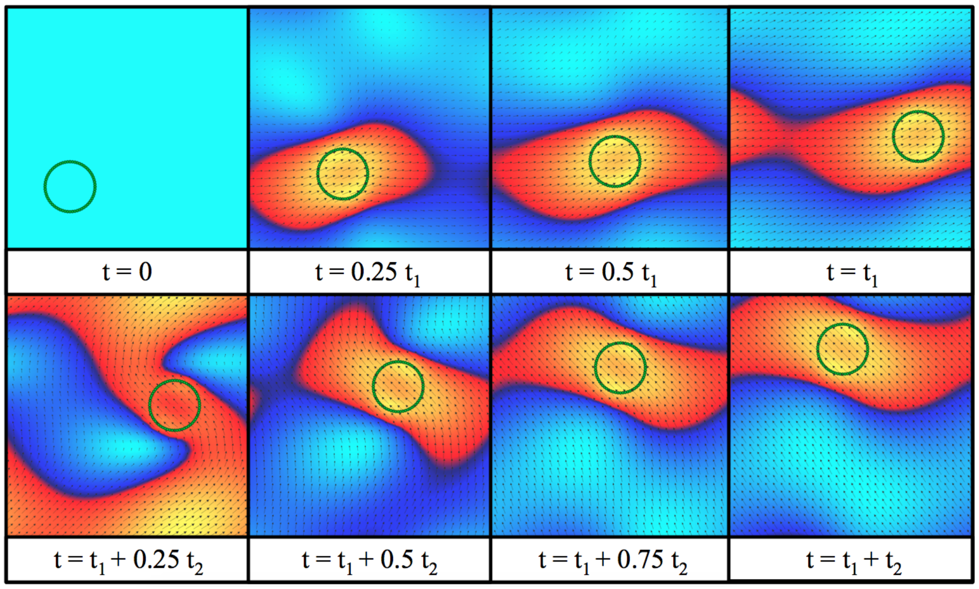


FIG. 3. *A circle undergoing prescribed motion in a fluid domain, causing the fluid to move in response. The colormap illustrates the magnitude of velocity, while the vector field depicts the fluid velocity itself.*

It is evident from Figure 3 that the fluid is moving the fastest right nearest to the circle, the immersed object. Students can change the fluid viscosity, μ, or interpolation time-points, t_1 or t_2, to see how the fluid motion changes. Furthermore, students can plot the simulation as it runs directly within MATLAB, or they can view the data using open-source visualization software, such as VisIt [15], which was used to construct Figure 3. Note that this simulation was designed to use a rather unresolved grid, e.g., 32×32, *for speed so students can watch the movement of the circle unfold directly in MATLAB .*

It should be emphasized that while this example only prescribed the motion of a circle, immersed within a fluid, to move between a few predetermined states, this is exactly the kind intepolation that is used in a lot of research applications, as mentioned in Section 1. One could imagine constructing a much more complex geometrical entity, such as a heart, fish, or other immersible structure, and prescribe it to move in rather complicated ways in order to test a hypothesis or engineering question!

From the way the linear interpolant in (2.3) and (2.5) was constructed, it should not be a surprise that the interpolant is continuous at all of the interpolation nodes, $\{x_j\}$, that is

$$\mathbf{h}_k(t_{k+1}) = \mathbf{h}_{k+1}(t_{k+1}). \tag{2.6}$$

At this stage, students are usually encouraged to consider what happens to the derivatives at the interpolation nodes. Simply differentiating either (2.3) or (2.5), one can show that that this linear interpolating scheme does not guarantee continuous derivatives at the nodes. Is this an issue?

Let's consider the movement of the circle from Example 2.1. When the circle is moving between State A to State B, what happens when $t \approx 0$ or $t \approx t_1$? We want to explore how fast the circle moving, its acceleration, and what implications these may have on the circle moving around. There are a couple things to consider:

1. First, we see that going from $t = 0$ to $t = \epsilon$, where $\epsilon > 0$, that the structure immediately begins to move at a constant speed, the constant speed it will move with between $0 \leq t \leq t_1$. This illustrates there is an instantaneous acceleration from not moving to moving at its constant speed.
2. Second, a similar phenomenon happens as $t \to t_1$; that is, an instantaneous deceleration from moving at its constant to speed to 0.
3. Third, if we are testing a hypothesis about the natural world or modeling an engineering device, no such situation occurs where we see such instantaneous accelerations (or decelerations for that matter).

We can encourage students to ask how can we ensure such accelerations do not happen? This can lead to a great discussion on not having enough *degrees of freedom* to enforce continuous derivatives, if we only have piecewise linear interpolating functions. Students may be obliged to try a polynomial of higher degree to interpolate between the positions, such as a quadratic or a cubic.

Before diving right in, note that the situation we were previously considering had the general linear interpolant

$$\mathbf{h}(t) = \left(\begin{array}{c} \mathbf{h}_0(t) \\ \mathbf{h}_1(t) \end{array} \right) = \left\{ \begin{array}{ll} \mathbf{a} + (d_0 + d_1 t)(\mathbf{b} - \mathbf{a}) & 0 \leq t \leq t_1 \\ \mathbf{b} + (d_2 + d_3 t)(\mathbf{c} - \mathbf{b}) & t_1 \leq t \leq t_2 \end{array} \right. , \tag{2.7}$$

with unknowns, $\{D_j\}_{j=0}^{3}$. Whether we knew it or not, we constructed (2.3) and (2.5) using the following continuity conditions to find the unknown coefficients:

$$\left. \begin{array}{r} \mathbf{h}_0(0) = \mathbf{a} \\ \mathbf{h}_0(t_1) = \mathbf{b} \\ \mathbf{h}_1(t_1) = \mathbf{b} \\ \mathbf{h}_1(t_2) = \mathbf{c} \end{array} \right\} \text{continuity} \tag{2.8}$$

That is, we had four unknowns, $\{d_j\}_{j=0}^{3}$, and used four conditions, all based on continuity, to find them. At this junction, if we wanted to impose more conditions such as continuity across one or more derivatives, we would not have enough degrees of freedom, or free parameters, satisfy all the conditions; we would have an over-constrained system.

Rather than use linear interpolation, which lead to instantaneous accelerations, let's try to use a cubic polynomial between successive points. Using a higher order polynomial interpolant will also provide more free parameters such that we are able to impose more continuity conditions. Keep in mind, although we will try a cubic polynomial interpolant, our goal is still interpolating between the two states $\mathbf{a} = (x_a, y_a)$ and $\mathbf{b} = (x_b, y_b)$.

Our goal is to use a familiar form of an interpolant, that looks awfully reminiscent of the linear case, but with a cubic function of the parameter, t, for $t \in [0, 1]$. We could attempt to use an interpolant such as the following

$$\mathbf{h}(t) = \mathbf{a} + g(t)(\mathbf{b} - \mathbf{a}), \tag{2.9}$$

where $g(t)$ is a cubic polynomial, rather than a line as in (2.7), e.g.,

$$g(t) = d_0 + d_1 t + d_2 t^2 + d_3^3. \tag{2.10}$$

Here we wish for continuity of the function, $\mathbf{h}(t)$, continuity in its velocity, $\mathbf{h}'(t)$, and no instantaneous accelerations ($\mathbf{h}''(t) = 0$ at the endpoints of the interpolation domain in t). However, when we write the conditions we wish to satisfy,

$$\begin{aligned} &\left.\begin{aligned} \mathbf{h}(0) &= \mathbf{a} \\ \mathbf{h}(1) &= \mathbf{b} \end{aligned}\right\} \text{ continuity} \\ &\left.\begin{aligned} \mathbf{h}'(0) &= 0 \\ \mathbf{h}'(1) &= 0 \end{aligned}\right\} \text{ continuous velocities} \\ &\left.\begin{aligned} \mathbf{h}''(0) &= 0 \\ \mathbf{h}''(1) &= 0 \end{aligned}\right\} \text{ no instantaneous accelerations} \end{aligned} \tag{2.11}$$

it is clear that we have an over-constrained system, that is, 6 conditions but only 4 unknowns, $\{d_j\}_{j=0}^{3}$. To circumvent this, we can introduce two interpolating mediary points, say p_1 and p_2, such that we partition the interval $t \in [0, 1]$ into three regions: (1) $t \in [0, p_1]$, (2) $t \in [p_1 < p_2]$, and (3) $t \in [p_2, 1]$. In each of those three regions, we could define an independent cubic interpolant, e.g.,

$$g(t) = \begin{cases} g_0(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 & 0 \le t \le p_1 \\ g_1(t) = b_0 + b_1 t + b_2 t^2 + b_3 t^3 & p_1 \le t \le p_2 \\ g_2(t) = c_0 + c_1 t + c_2 t^2 + c_3 t^3 & p_2 \le t \le 1 \end{cases}. \tag{2.12}$$

Upon imposing the conditions from (2.11) onto (2.12), we see that now we have 12 degrees of freedom but only 6 equations, leaving us with an under-constrained system. If we were to think physically about this, at the interfaces $t = p_1$ and $t = p_2$, we would want continuity of our interpolating functions and their first and second derivatives, providing continuity in velocity and acceleration, respectively. Hence the piecewise cubic interpolating functions must satisfy the following constraints:

$$\begin{aligned} &\left.\begin{aligned} g_0(0) &= 0 \\ g_2(1) &= 1 \\ g_0(p_1) &= g_1(p_1) \\ g_1(p_2) &= g_2(p_2) \end{aligned}\right\} \text{ continuity} \\ &\left.\begin{aligned} g_0'(0) &= 0 \\ g_2'(1) &= 0 \\ g_0'(p_1) &= g_1'(p_1) \\ g_1'(p_2) &= g_2'(p_2) \end{aligned}\right\} \text{ continuous velocities} \\ &\left.\begin{aligned} g_0''(0) &= 0 \\ g_2''(1) &= 0 \\ g_0''(p_1) &= g_1''(p_1) \\ g_1''(p_2) &= g_2''(p_2) \end{aligned}\right\} \text{ no instantaneous accelerations} \end{aligned} \tag{2.13}$$

This gives the following linear system to solve, with variables, p_1 and p_2,

(2.14)
$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & p_1 & p_1^2 & p_1^3 & -1 & -p_1 & -p_1^2 & -p_1^3 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2p_1 & 3p_1^2 & 0 & -1 & -2p_1 & -3p_1^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 6p_1 & 0 & 0 & -2 & -6p_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & p_2 & p_2^2 & p_2^3 & -1 & -p_2 & -p_2^2 & -x2^3 \\ 0 & 0 & 0 & 0 & 0 & 1 & 2p_2 & 3p_2^2 & 0 & -1 & -2p_2 & -3p_2^2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 6p_2 & 0 & 0 & -2 & -6p_2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 6 \end{bmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ b_0 \\ b_1 \\ b_2 \\ b_3 \\ c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

As an example, if we let $p_1 = 0.25$ and $p_2 = 0.925$, upon solving (2.14), we find the coefficients to be approximately

$$\begin{array}{lll} a_0 = 0 & b_0 = 0.123 & c_0 = -16.778 \\ a_1 = 0 & b_1 = -1.481 & c_1 = 53.333 \\ a_2 = 0 & b_2 = 5.923 & c_2 = -53.333 \\ a_3 = 4.324 & b_3 = -3.577 & c_3 = 17.778. \end{array} \tag{2.15}$$

A plot of the resulting interpolant, $h(t)$, $h'(t)$, and $h''(t)$ is provided in Figure 4. It is clear that all the conditions sought after in (2.13) are satisfied. Moreover by introducing two new parameters p_1 and p_2, we can essentially control the acceleration of the interpolated motion. The script used to solve this system is provided in the Supplemental Materials, e.g., the `interp_Function_Coeffs.m` script.

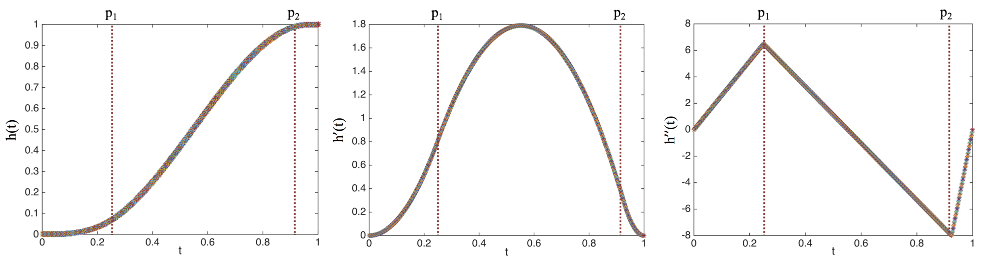


FIG. 4. *Plots of the piecewise cubic interpolant, $h(t)$, its derivative, $h'(t)$, and its second derivative, $h''(t)$, for $p_1 = 0.25$ and $p_2 = 0.925$ for $0 \le t \le 1$, respectively.*

As $p_1 \to 0$ (or $p_2 \to 1$), the initial acceleration (or final deceleration) becomes larger in magnitude. In practice we can use the parameters p_1 and p_2 to match the acceleration to the kinematics coming from a biological system or engineering system. These parameters p_1 and p_2 may actually provide a beneficial tool for capturing the correct kinematics of a system in a mathematical model!

Next, in Example 2.2, we will illustrate qualitative differences in the fluid dynamics when using a cubic interpolant rather than linear interpolant, as is in the previous example. The corresponding source code for this example with a cubic interpolant is found in `Examples_Education/Interpolation/Moving_Circle/Cubic_Interp`.

Example 2.2. *In this example we will use the same prescribed motion described in Example 2.1; however, we will use two different interpolation polynomials - one linear and one cubic to interpolate between successive states. Using the cubic interpolant that was determined above, with $p_1 = 0.25$ and $p_2 = 0.925$, we ran simulations and compared the results to those when using the linear interpolation scheme.*

Simulations were compared at time-points when the circle would be accelerating or decelerating between State $A \to B$ and the acceleration at the very beginning of State $B \to C$. This is illustrated in Figure 5, where the magnitude of velocity is used to demonstrate qualitative differences in the underlying fluid motion. It is clear that when using different interpolants to prescribe the motion between two states, it can lead to significant differences in the fluid motion. Movies illustrating the dynamical differences are provided in the Supplemental Materials (Supplemental/Circles/Linear_Interp or Supplemental/Circles/Cubic_Interp).

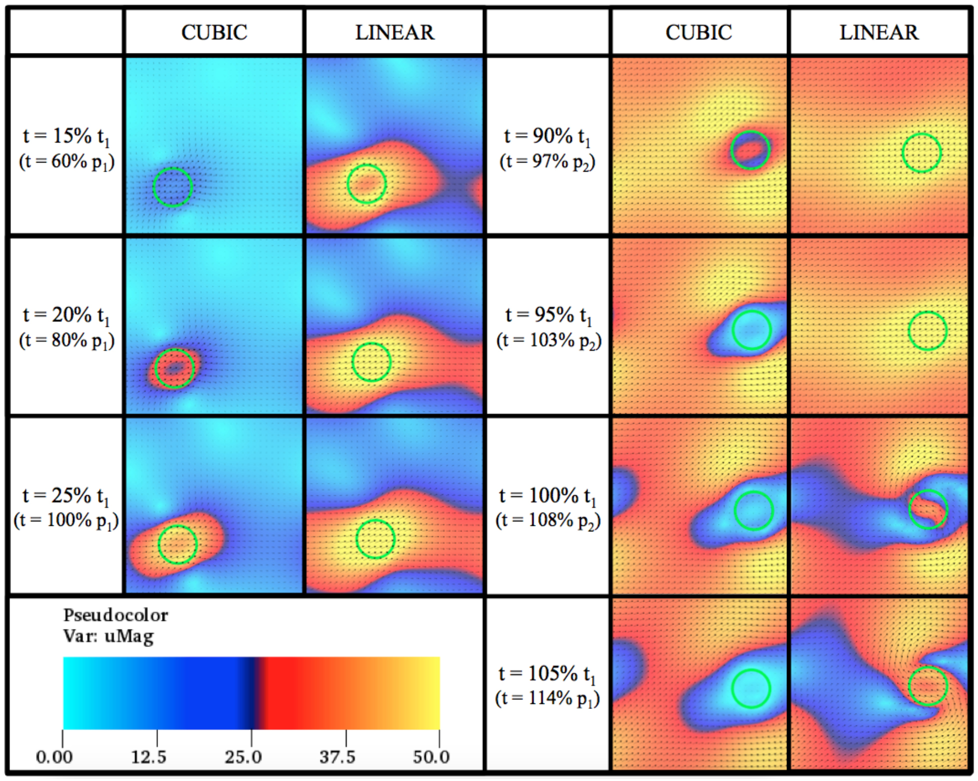


Fig. 5. *Images illustrating qualitative differences in the magnitude of velocity when using linear and cubic interpolants. Snapshots were taken when the circles were accelerating and decelerating from Phase $A \to B$, and then accelerating from Phase $B \to C$.*

We note that in both cases the circle moves between States $A \leftrightarrow B$ and $B \leftrightarrow C$ with the periods $t_1 = 0.01$ and $t_2 = 0.02$, respectively. In fact, qualitatively it appears that in both cases the circles look like they maybe moving in the same way; however, there are clear dynamical differences as seen by the underlying fluid velocity. Again, this is because the velocity and acceleration/deceleration of the circles moving between the states is significantly different. This is an important aspect that should get proper attention when mathematically modeling using prescribed motion. Not only *is it important to make the an object begin and end in the right place, but we must also make sure the way it moves between the states is biologically (or scientifically) relevant! Introducing higher order polynomial interpolants is a convenient way to introduce more degrees of freedom into a model, so it is able capture more kinematic accuracy.*

3. Interpolation and beating hearts: a virtual walk through.

Here we present an example of how to implement an object's prescribed motion within the *IB2d* software. We will consider the motion of a beating cartoon heart, that is, a heart that goes between two states, one larger and one smaller, see Figure 6. The hole in the heart is to allow fluid to flow in and out of it, thereby obeying fluid volume conversation.

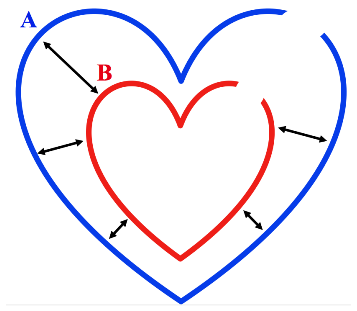


FIG. 6. *Moving between States A and B to model a beating heart.*

Running the simulation found in `Examples_Education/Interpolation/Beating_Heart`, will produce data that can be visualized, as in Figure 7. The corresponding movie is provided in the Supplemental Materials (`Supplement/Pulsing_Heart`). We are using the same cubic interpolation scheme that was discussed in Section 2 to move between State $A \rightarrow B$ and then State $B \rightarrow A$ with periods t_1 and t_2, respectively. However we also introduce an intermediate resting state of length t_R, before moving back from State B→A to introduce additional possible model complexity.

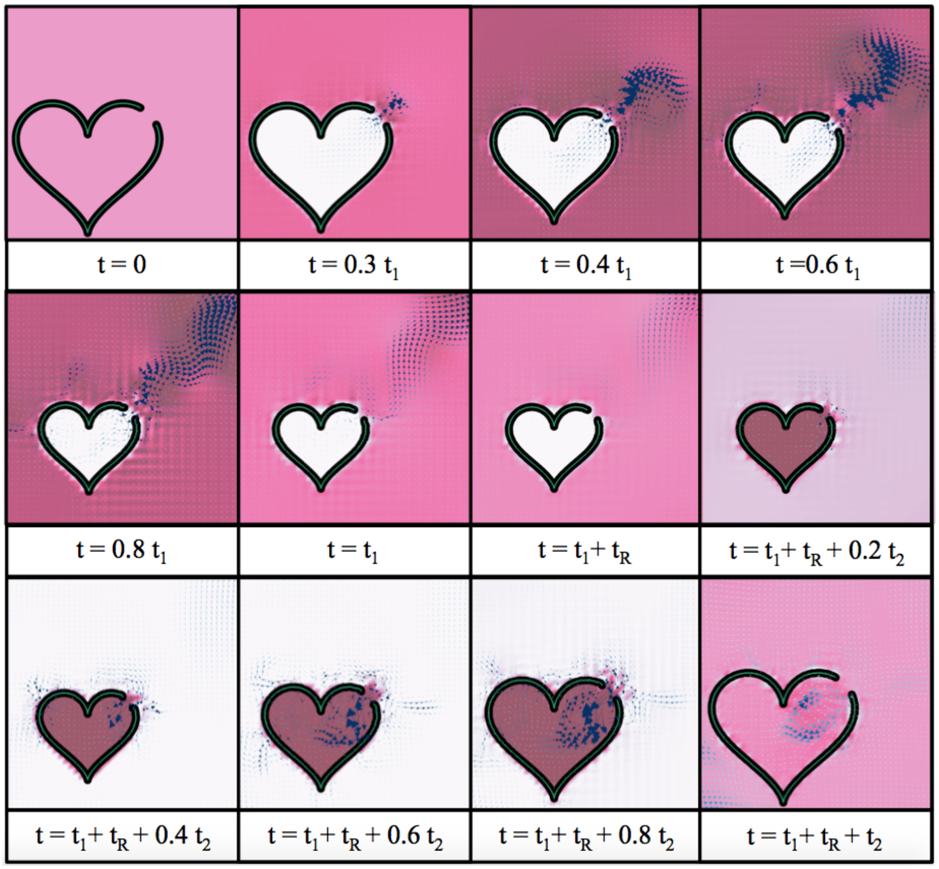


FIG. 7. *Snapshots of a simulation of a beating cartoon heart that is immersed within a fluid. The colormap depicts the underlying pressure, while the vector field depicts the fluid velocity itself.*

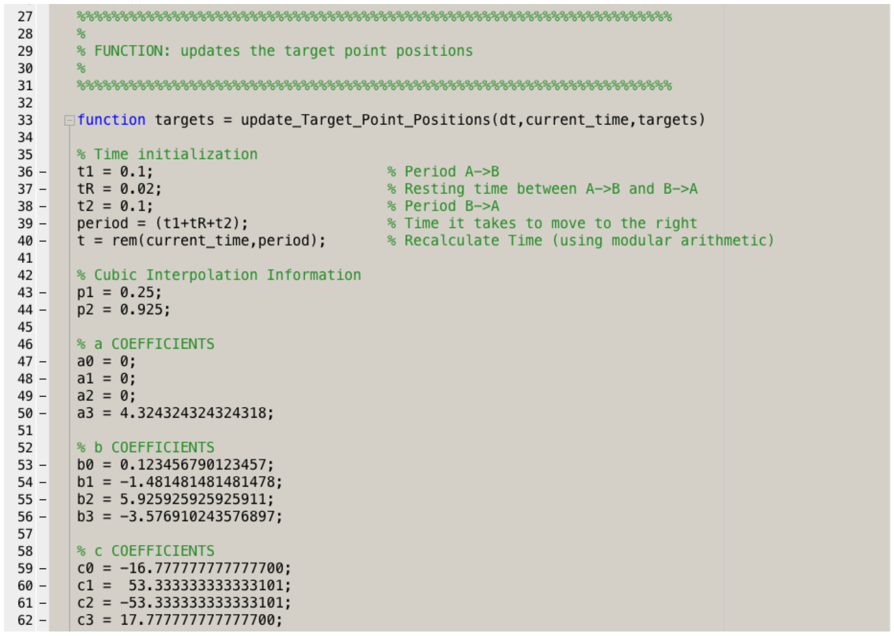
We will now dive into detail on how to implement the cubic interpolant to prescribe motion. Although, a beating heart example is introduced here, it should be noted that this will work for just about any geometry, as long as each state has both the same number of points, is ordered consistently, and has a 'hole' to obey volume conversation.

The script that actually prescribes the motion is `update_Target_Point_Positions.m`. This script does the following three things:

1. ***Specify the period spent moving between states and initialize the cubic interpolant.***

   We initialize the time spent in each phase moving between $A \rightarrow B$, resting, and finally $B \rightarrow A$ as t_1, t_R, and t_2, respectively. We also specify the parameters for the specific cubic interpolant we are going to use to move between States, that is, the coefficients of the cubic interpolant in each sub-phase, $\{a_j, b_j, c_j\}_{j=0}^{3}$, and location of the interpolation nodes, p_1 and p_2. The values of p_1 and p_2 were chosen to be 0.25 and 0.925, respectively, which is the same case as in Section 2.

   Note we also define a period of the total heart beat to be the sum of all the subsequent phases, $t_1 + t_R + t_2$, and use modular arithmetic, with respect to said period, for an adjusted time in the simulation in order to simulate repetitive heartbeats.

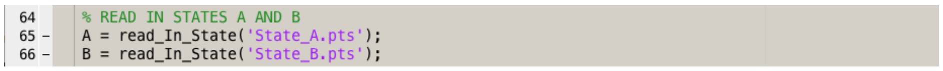


```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% FUNCTION: updates the target point positions
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function targets = update_Target_Point_Positions(dt,current_time,targets)

% Time initialization
t1 = 0.1;                                % Period A->B
tR = 0.02;                               % Resting time between A->B and B->A
t2 = 0.1;                                % Period B->A
period = (t1+tR+t2);                     % Time it takes to move to the right
t = rem(current_time,period);            % Recalculate Time (using modular arithmetic)

% Cubic Interpolation Information
p1 = 0.25;
p2 = 0.925;

% a COEFFICIENTS
a0 = 0;
a1 = 0;
a2 = 0;
a3 = 4.324324324324318;

% b COEFFICIENTS
b0 = 0.123456790123457;
b1 = -1.481481481481478;
b2 = 5.925925925925911;
b3 = -3.576910243576897;

% c COEFFICIENTS
c0 = -16.777777777777700;
c1 =  53.333333333333101;
c2 = -53.333333333333101;
c3 = 17.777777777777700;
```

FIG. 8. *Initializing the time for each phase of motion as well as the cubic interpolant's coefficients from Section 2.*

2. ***Read in the points associated for States*** A ***and*** B.

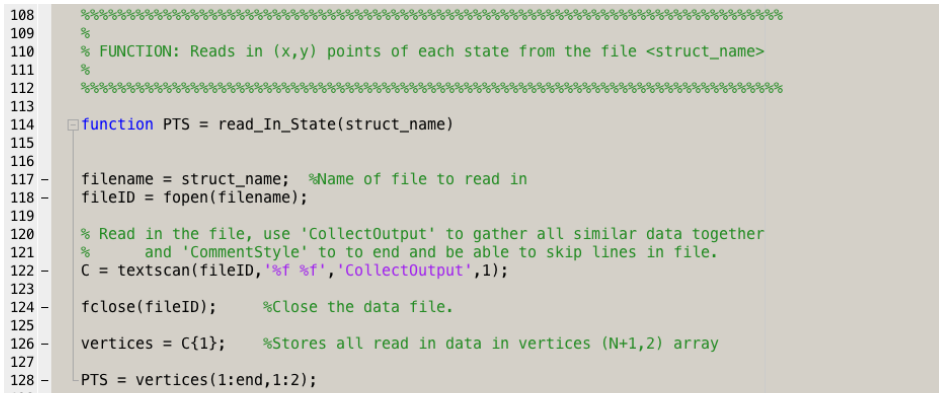
Next we read in the (x, y) positions for each state into $N \times 2$-sized matrices, where the columns give the x and y positions, respectively.



```
% READ IN STATES A AND B
A = read_In_State('State_A.pts');
B = read_In_State('State_B.pts');
```

FIG. 9. *Reading in the (x,y) positions for States A and B into matrices* ***A*** *and* ***B***.

For completeness the code that reads in the data from the files `State_A.pts` and `State_B.pts` is shown below.



```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% FUNCTION: Reads in (x,y) points of each state from the file <struct_name>
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function PTS = read_In_State(struct_name)


filename = struct_name;  %Name of file to read in
fileID = fopen(filename);

% Read in the file, use 'CollectOutput' to gather all similar data together
%      and 'CommentStyle' to to end and be able to skip lines in file.
C = textscan(fileID,'%f %f','CollectOutput',1);

fclose(fileID);     %Close the data file.

vertices = C{1};    %Stores all read in data in vertices (N+1,2) array

PTS = vertices(1:end,1:2);
```

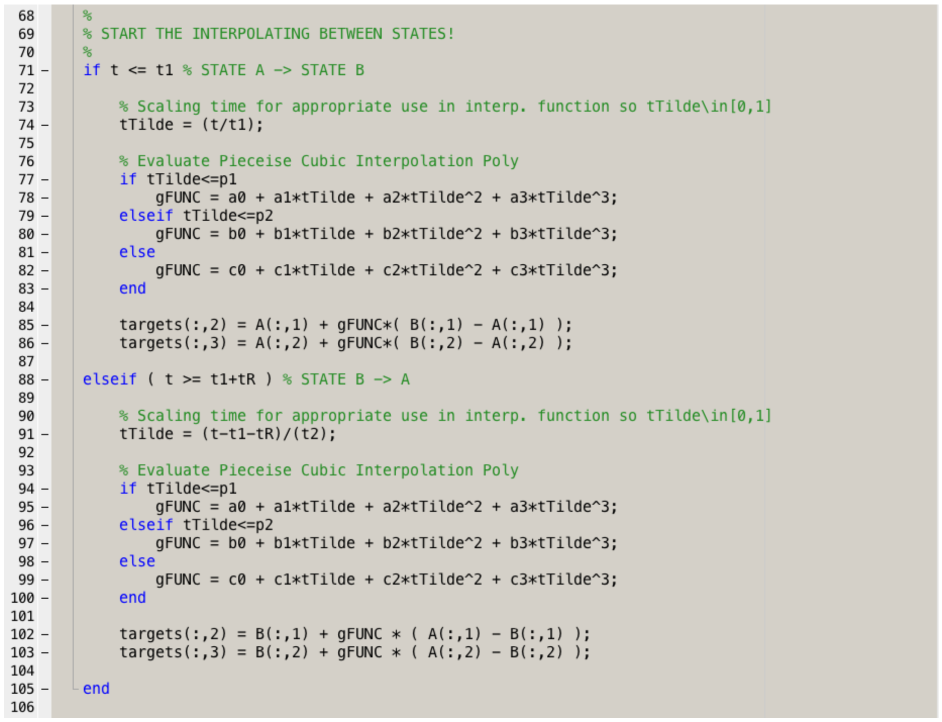
FIG. 10. *Function that reads in the (x, y) point data.*

We note that the information contained within the files `State_A.pts` and `State_B.pts` are lists of the x and y points for each phase, respectively. If you would like to substitute your own shape, rather than use a heart, one only needs to create *.txt* files that contain their own (x, y) point geometries. Note you must also make the *.vertex* file contain the (x, y) positions of the first state as well as include a similarly constructed *.target* file, see the Tutorials in Appendix A.1 for further details.

3. ***Check which phase of the beating heart it's in, e.g., contraction or expansion, and then update the target point positions to which prescribes the motion of the beating heart.***

Upon checking to see which phase of the simulation the adjusted time currently relates to gives three state possibilities: either the simulation is between States $A \to B$ or States $B \to A$, or no motion is being prescribed, e.g., heart is in a rest state.

For example, if the simulation time, t, is less than the period moving from $A \to B$, the script then inquiries to find the point between State A and B that it is in, that is, it scales the time appropriately to $\tilde{t} = t/t_1$, so that it is possible to compare $\tilde{t}$ to the interpolation nodes, p_1 and p_2.



```
%
% START THE INTERPOLATING BETWEEN STATES!
%
if t <= t1 % STATE A -> STATE B

    % Scaling time for appropriate use in interp. function so tTilde\in[0,1]
    tTilde = (t/t1);

    % Evaluate Pieceise Cubic Interpolation Poly
    if tTilde<=p1
        gFUNC = a0 + a1*tTilde + a2*tTilde^2 + a3*tTilde^3;
    elseif tTilde<=p2
        gFUNC = b0 + b1*tTilde + b2*tTilde^2 + b3*tTilde^3;
    else
        gFUNC = c0 + c1*tTilde + c2*tTilde^2 + c3*tTilde^3;
    end

    targets(:,2) = A(:,1) + gFUNC*( B(:,1) - A(:,1) );
    targets(:,3) = A(:,2) + gFUNC*( B(:,2) - A(:,2) );

elseif ( t >= t1+tR ) % STATE B -> A

    % Scaling time for appropriate use in interp. function so tTilde\in[0,1]
    tTilde = (t-t1-tR)/(t2);

    % Evaluate Pieceise Cubic Interpolation Poly
    if tTilde<=p1
        gFUNC = a0 + a1*tTilde + a2*tTilde^2 + a3*tTilde^3;
    elseif tTilde<=p2
        gFUNC = b0 + b1*tTilde + b2*tTilde^2 + b3*tTilde^3;
    else
        gFUNC = c0 + c1*tTilde + c2*tTilde^2 + c3*tTilde^3;
    end

    targets(:,2) = B(:,1) + gFUNC * ( A(:,1) - B(:,1) );
    targets(:,3) = B(:,2) + gFUNC * ( A(:,2) - B(:,2) );

end
```

FIG. 11. *Checks to see which phase of the motion the adjusted simulation time currently relates to and then updates the position of the target points in the x and y directions, which will effectively drive the motion of the beating heart.*

4. Interpolation between material property states: it swims!. Ready, Set, Swim! Here we present a simple, idealized model of anguilliform locomotion - swimming. Here we do not wish to prescribe the exact kinematics of the swimmer's locomotive patterns, but rather we will only model how the swimmer's body switches between two preferred curvature states. This is a biologically relevant modeling assumption as muscle activation patterns produce specific intrinsic curvatures for a swimmer's body [24, 25, 16]. By switching between two different curvature states, the swimmer's body bends and contorts, and locomotion emerges due to the swimmer's interactions with the surrounding fluid. How can model the process of switching between curvature states? That's right; you guessed it - interpolation!

We must first get in the water before we can swim; let's begin with the shape of the swimmer. To create a simplified scenario, the idealized swimmer's body was constructed by taking a line segment and attaching a polynomial section to it, see Figure 12, adapted from [9]. Thus the swimmer's geometry (morphology) is modeled as an infinitely thin $1D$ curve only. The straight portion composes 28% of the total length of the body, while the polynomial, i.e., $y = x^3$, portion makes up the remaining 72%. The polynomial section was determined by starting at $x = 0$ and adding equally spaced points until $x = L/10$.

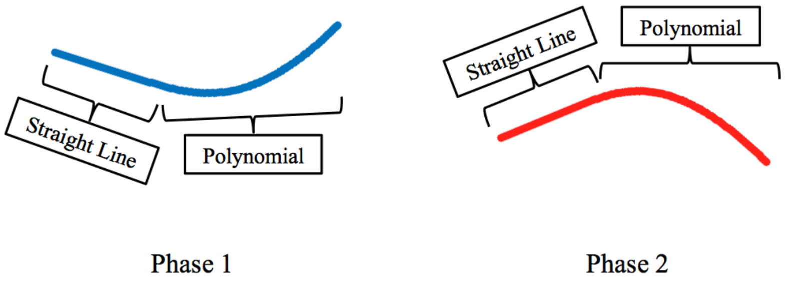


FIG. 12. *The two phases, in which, the preferred curvature was interpolated between to cause forward swimming, adapted from [9].*

Note that all the points are equally spaced at a distance twice of that of the fluid mesh ($ds = 2dx$). Each phase was defined by negating the y-coordinate of the polynomial portion of the body. The "curvatures" were computed as follows (to tie into the *IB2d* framework, see [9]):

$$\begin{aligned} C_x^P &= x_{Lag}^P(s) - 2x_{Lag}^P(s+1) + x_{Lag}^P(s+2) \\ C_y^P &= y_{Lag}^P(s) - 2y_{Lag}^P(s+1) + y_{Lag}^P(s+2) \end{aligned} \tag{4.1}$$

where s runs over all Lagrangian points along the swimmer's body and P refers to Phase 1 or 2.

This intrinsic curvature is the quantity we will now interpolate between. We are no longer interpolating between explicit *positions*, but instead material property states! Although seemingly different, the mathematics (spline interpolation) works out exactly the same. In lieu of changing explicit coordinates (or positions), we now update the curvatures, C_x^P and C_y^P in the `update_nonInv_Beams.m` script.

We also define the downstroke and upstroke to be moving between Phase 1 to Phase 2 and Phase 2 to Phase 1, respectively. Furthermore we also define 1 stroke period to encompass both the upstroke and downstroke. The same interpolation rigmarole, as in Section 3, follows.

Running the simulation found in `Examples_Education/Interpolation/Swimmer/Single_Swimmer` will produce locomotion data that can be visualized as in Figure 13. This figure shows the idealized anguilliform swimmer moving forward due to vortices being shed off its caudal end during each stroke. The background colormap represents the fluid's vorticity, e.g, the local swirling motion of the fluid (mathematically given by the curl of the velocity field, $\nabla \times \mathbf{u}(\mathbf{x}, t)$). The corresponding movie to Figure 13 is provided in the Supplementary Materials (`Supplemental/Swimmer/Individual_Swimmer/`). Furthermore, we can quantitatively track the position of the swimmer's head over time, using the script `Individual_Swimmer_Analysis.m`, to see what its forward swimming patterns (and performance) looks like, see Figure 14.

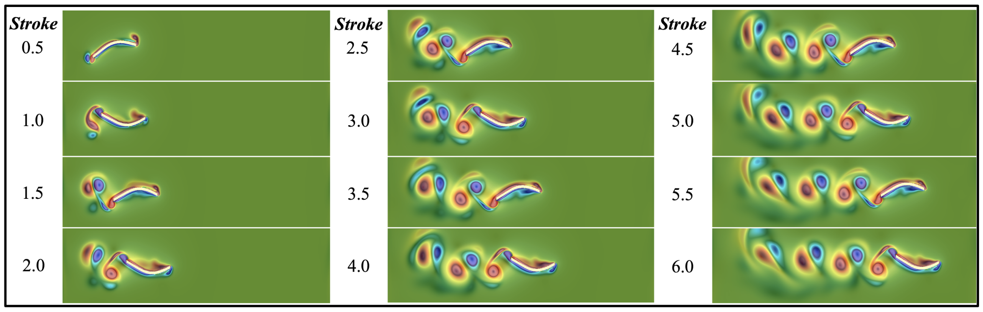


FIG. 13. *An idealized anguilliform swimmer progressing forward due to continually changes in the preferred curvature of its configuration with a stroke frequency $f = 1.0s^{-1}$. The background colormap illustrates the fluid's vorticity.*

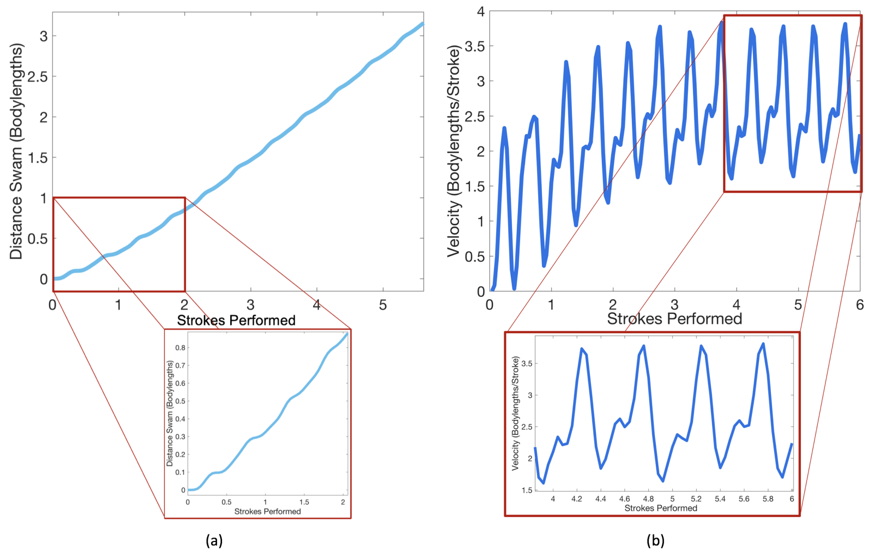


FIG. 14. *Swimming performance of the single anguilliform swimmer shown in Figure 13. (a) Distance (bodylengths) vs number of strokes performed and (b) velocity (bodylengths/stroke) vs. number of strokes performed.*

At this point while we have a single simulation of one anguilliform swimmer, there are many interesting questions one could ask, including a plethora of interesting biological questions. However, we will first focus on how subtle changes in interpolating between curvature states affects swimming performance. Note that for the simulation shown in Figure 13 that $(p_1, p_2) = (0.1, 0.9)$. In particular, we will ask three questions:

1. What happens when the interpolation mediary points p_1 and p_2 are changed? Remember these points help dictate the acceleration and velocity profile of the interpolation (see Section 2).
2. What happens if we make the interpolation mediary points (p_1, p_2) asymmetric (e.g., say if $p_1 = 0.1$ and $p_2 = 0.5$ rather than $p_2 = 0.9$)?
3. What happens if we have an asymmetric stroke pattern? (For example, if the upstroke is 25% of the total period while the downstroke is only 75%?)

Lastly, we can have a little fun with our swimmer, taking advantage of the fact it is immersed in a fluid, and ask how does changing the fluid environment affect swimming performance? To change its fluid environment, we will only have to vary the fluid's viscosity. This effectively asks how the swimmer performs in stickier and stickier fluid environments, like going from water to corn syrup. For those with previous experience in fluid dynamics, this equates to looking how swimming performance varies over a range of Reynolds Numbers, Re.

It is important to note that while asking these questions (and hopefully making hypothesis) we are only changing one parameter of a single simulation at a time, whether that it is (p_1, p_2), the upstroke and downstroke percentages of the total period, or the fluid's viscosity.

4.1. Changing (p_1, p_2) symmetrically.

First we will investigate how the choice of interpolation mediary points (p_1, p_2) affects swimming performance of our idealized anguilliform swimmer. These simulations are found in `Examples_Education/Interpolation/Swimmer/Case1`. We will vary the (p_1, p_2) points symmetrically about the interpolation interval and consider the following cases:

1. $(p_1, p_2) = (0.1, 0.9)$
2. $(p_1, p_2) = (0.2, 0.8)$
3. $(p_1, p_2) = (0.3, 0.7)$
4. $(p_1, p_2) = (0.4, 0.6)$

Upon varying these points, we need to make sure that our interpolation function is consistent, that is, we need to solve the linear system described in Section 2 accordingly to get the proper coefficients for the spline interpolant. These coefficients are listed in Supplement 2 of the Supplementary Materials. Once calculated, we can modify the `update_nonInv_Beams.m` script, which performs the curvature interpolation.

We will now compare the interpolation profiles ($h(x)$, $h'(x)$, and $h''(x)$) for two cases: $(p_1, p_2) = \{(0.1, 0.9), (0.4, 0.6)\}$. Comparison plots are given in Figure 15. We note that in every case we still have continuous first and second derivatives; however, the velocity and acceleration profiles are significantly different.

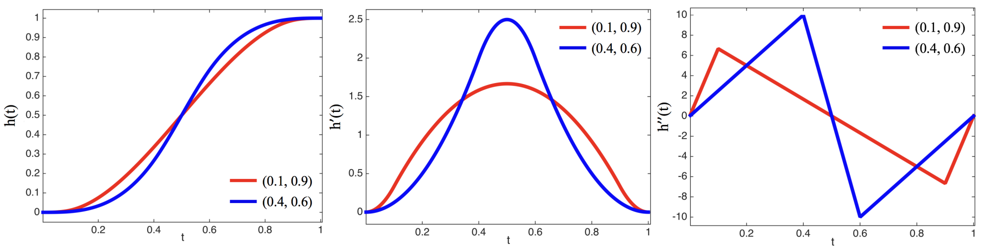


FIG. 15. *Plots of the piecewise cubic interpolant, $h(t)$, its derivative, $h'(t)$, and its second derivative, $h''(t)$, with $0 \leq t \leq 1$, for varying (p_1, p_2) symmetrically chosen.*

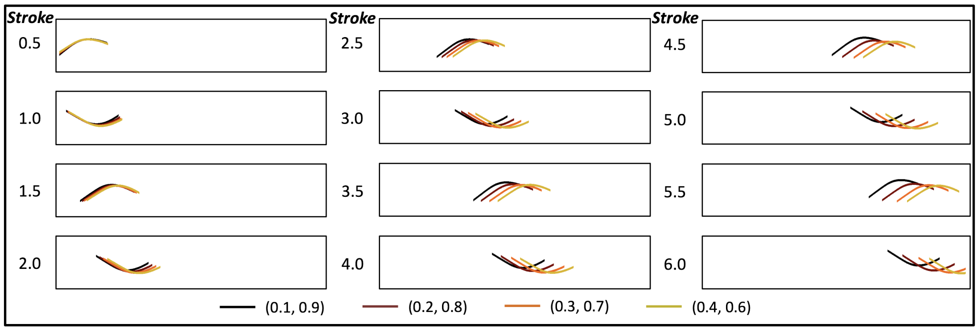


FIG. 16. *Snapshots from simulations for the case of symmetric interpolation points, given by* $(p_1, p_2) \in \{(0.1, 0.9), (0.2, 0.8), (0.3, 0.7), (0.4, 0.6)\}$.

Upon running the aforementioned simulations, it is evident that changing (p_1, p_2) affects swimming performance! Snapshots from the simulation are given in Figure 16. Note that although the swimmer's position from each case are over laid on each other, each simulation was independently performed; there are no swimmer-swimmer interactions. The case when $(p_1, p_2) = (0.4, 0.6)$ appears in the lead after 6 strokes followed by cases $(0.3, 0.7)$, $(0.2, 0.8)$, and then $(0.1, 0.9)$, respectively. The faster cases correspond to higher magnitudes of velocity and acceleration, see Figure 15. We also present the distance swam vs. swimming stroke as well as forward swimming speed vs. stroke in Figure 17, which further confirms those results. Furthermore, both peaks in the forward swimming speed's waveform are also higher in the faster cases. The corresponding movie for these simulations is provided in the Supplementary Materials (`Supplemental/Swimmer/Case1/`).

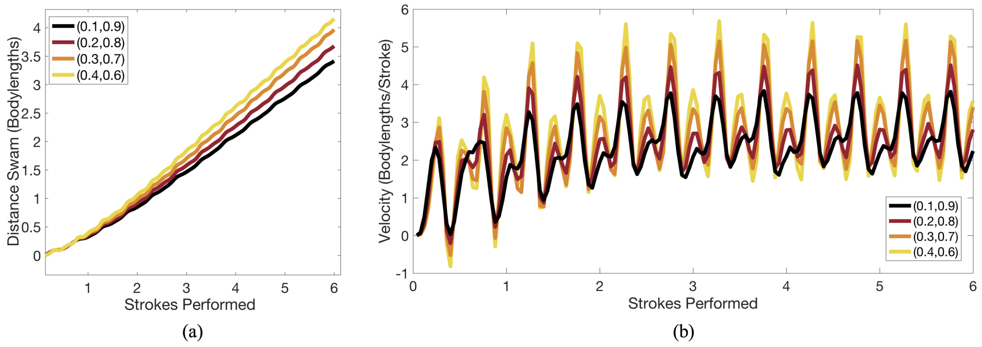


FIG. 17. *(a) Forward distance swam and (b) forward velocity vs. swimming strokes performed in the case of symmetric interpolation points* (p_1, p_2) *in* $[0, 1]$.

Simply changing the interpolation mediary points, p_1 and p_2, affects swimming performance even when everything else remains the same - the same cubic spline-based interpolating function , the same upstroke and downstroke periods, and the same fluid environment! Next we will once again ask how swimming performance is affected if we again change the interpolation points p_1 and p_2, but this time place them asymmetrically about the interpolation window $[0, 1]$.

4.2. Changing (p_1, p_2) asymmetrically. Here we will again will inquire into how changing the interpolation mediary points (p_1, p_2) affects swimming performance, but this time choose p_2 such that interpolation points are not symmetric within the interpolation interval $[0, 1]$. These simulations are found in `Examples_Education/Interpolation/Swimmer/Case2`. We selected the following (p_1, p_2) cases:

1. $(p_1, p_2) = (0.1, 0.9)$
2. $(p_1, p_2) = (0.1, 0.7)$
3. $(p_1, p_2) = (0.1, 0.5)$
4. $(p_1, p_2) = (0.1, 0.3)$

It is important to note that in this section, although we are asymmetrically varying p_2 about the interpolation interval, both the upstroke and downstroke have the same period. The only difference is that the rate of change of the interpolating function $h(t)$ during each portion of the stroke.

Again, to ensure that the interpolation function is consistent, we solve the linear system described in Section 2 for each different set of interpolation points, (p_1, p_2). These coefficients are listed in Supplement 2 of the Supplementary Materials and are used in each corresponding `update_nonInv_Beams.m` script to perform the curvature interpolation.

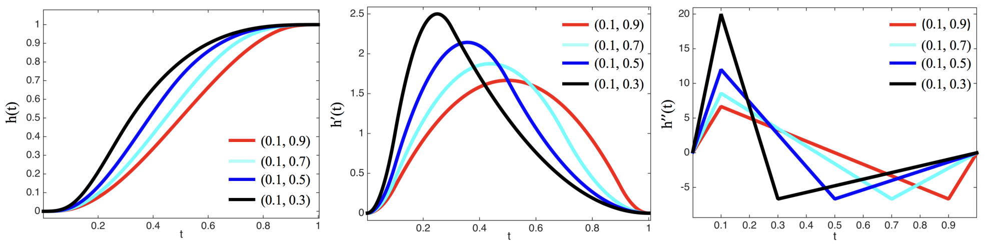


FIG. 18. *Plots of the piecewise cubic interpolant, $h(t)$, its derivative, $h'(t)$, and its second derivative, $h''(t)$, with $0 \le t \le 1$, for varying (p_1, p_2) asymmetrically chosen.*

The interpolation profiles $h(t)$, $h'(t)$, and $h''(t)$ look strikingly different than those shown in Section 4.1 due to the asymmetry introduced by choice of p_1 and p_2. The profiles are given in Figure 18.

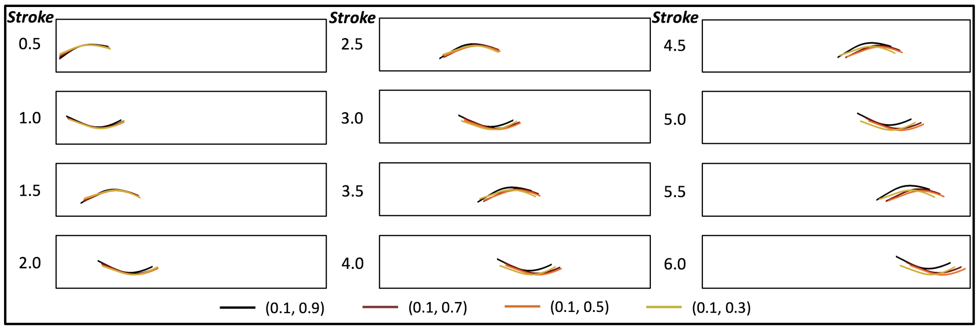


FIG. 19. *Snapshots from simulations for the case of asymmetric interpolation points, given by $p_1 = 0.1$ and $p_2 \in \{0.3, 0.5, 0.7, 0.9\}$.*

As hopefully hypothesized, the dynamics are different between each swimmer for the above cases; however, perhaps surprisingly, there appears to be less variation than the previous case of symmetric (p_1, p_2) choices in terms of forward swimming performance. Snapshots of the four swimmers are shown in Figure 19. In this case there was a non-linear relationship with choice of p_2 and how fast the swimmer went, e.g., the case with $p_2 = 0.5$ was the fastest, followed by $p_2 = 0.7$, then 0.3, and finally 0.9. This is confirmed when analyzing the data, shown in Figure 20, which gives the distance swam vs. swimming stroke as well as forward swimming velocity vs. stroke. The corresponding movie of these simulations is provided in the Supplementary Materials (`Supplemental/Swimmer/Case2/`). What do you think happens if we again sweep over $p_2 = \{0.3, 0.5, 0.7, 0.9\}$ but choose a different p_1, where $p_1 \in (0, p_2)$?

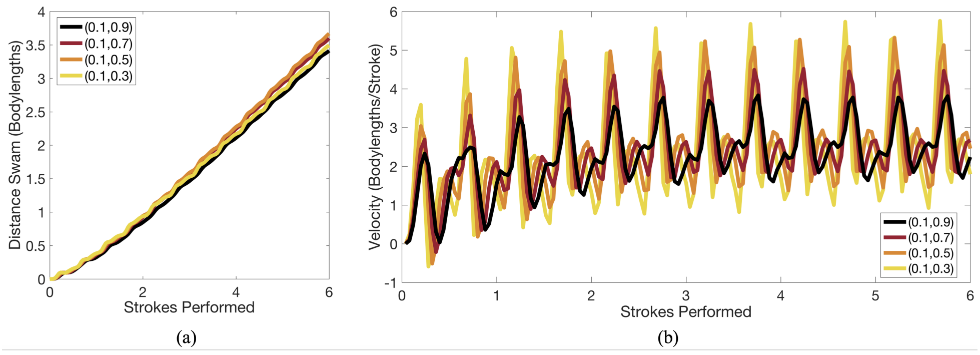


FIG. 20. *(a) Forward distance swam and (b) forward velocity vs. swimming strokes performed in the case of asymmetric interpolation points, given by $p_1 = 0.1$ and $p_2 = \{0.3, 0.5, 0.7, 0.9\}$.*

While Sections 4.1 and 4.2 used different interpolation mediary points, p_1 and p_2, they both used the same upstroke and downstroke periods as well as same fluid environment, e.g., fluid density and viscosity were the same. We will now investigate variances in swimming performance due to varying stroke periods, followed by changing the fluid environment via varying the fluid's viscosity.

4.3. Making asymmetric stroke periods. In this case we will keep the interpolation points fixed at $(p_1, p_2) = (0.1, 0.9)$ and fix the stroke period to $T = 2.0s$ (frequency of 0.5 Hz). We then asymmetrically vary the upstroke (UPS) and downstroke (DWS) percentages of the total stroke period (T). Recall that earlier we defined one stroke to be the upstroke and downstroke periods added together. To that end, we simulated the following cases:

1. UPS = DWS, e.g., (UPS,DWS)=(50%T,50%T)
2. UPS = 75% DWS, e.g., (UPS,DWS)=(42.9%T,57.1%T)
3. UPS = 50% DWS, e.g., (UPS,DWS)=(33%T,0.66%T)
4. UPS = 25% DWS, e.g., (UPS,DWS)=(20%T,0.80%T)

Note that although we have made each portion of a single full stroke have a different sub-period, we can still use the same piecewise interpolant, $h(t)$, to interpolate between each! These simulations are found in *Examples_Education/Interpolation/Swimmer/Case3*.

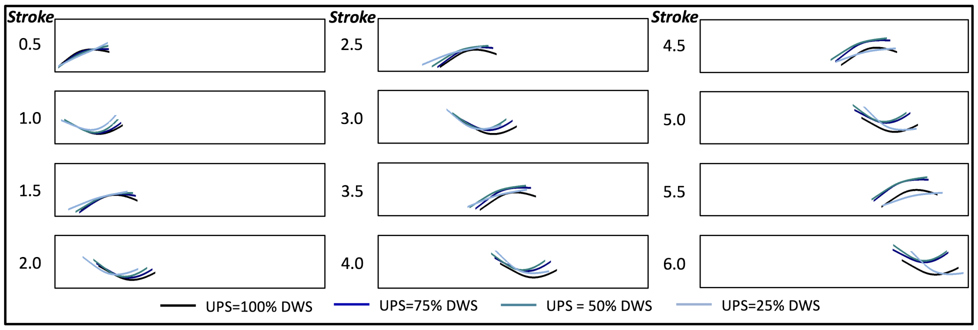


Fig. 21. *Snapshots from simulations with varying upstroke and downstroke percentages of a single stroke period.*

As the UPS percentage of a stroke decreases, the upstroke happens faster. However, although the swimmer that swims forward the fastest also has the quickest UPS, having a faster UPS does not always equate to a faster forward swimming speed, see Figures 21 and 22. The initial acceleration of the UPS=25%DWS case is the slowest but eventually it starts outswimming the others - truly a tortoise and a hare story (well not exactly, biologically). Figure 21 gives snapshots of the four swimmers and Figure 22 presents the distance swam vs. swimming stroke as well as forward swimming velocity vs. swimming stroke. The corresponding movie of these simulations is provided in the Supplementary Materials (`Supplemental/Swimmer/Case3/`). Interestingly, due to the asymmetric UPS and DWS, the swimming velocity profiles look significantly different than those in Figures 17 and 20. In particular, the waveforms appear trimodal rather than bimodal, which were observed in the cases of varying (p_1, p_2), especially in the cases of UPS = 25% DWS and UPS = 50% DWS.

What do you think would happen if we redid this same analysis, but with a different (p_1, p_2)? Or if we varied the stroke frequency cycle-by-cycle during the simulation?

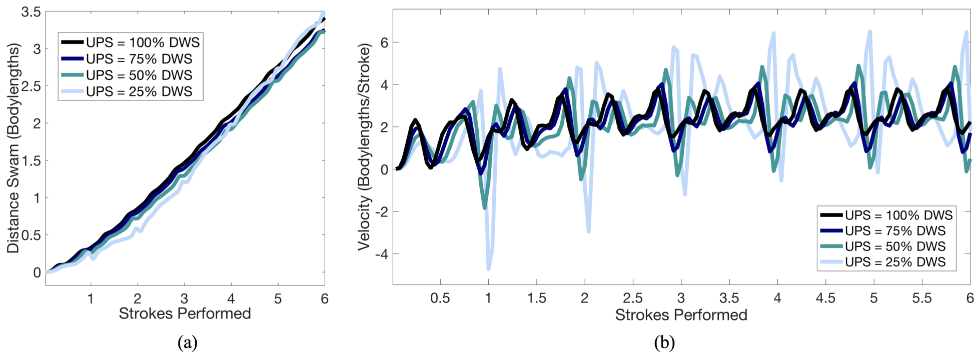


Fig. 22. *(a) Forward distance swam and (b) forward velocity vs. swimming strokes performed in the case of asymmetric upstroke and downstroke periods.*

4.4. Changing the fluid viscosity (Re). Finally, we will consider what happens if we put the swimmer in varying fluid environments, via changing the fluid's viscosity. This equates to placing the swimmer in less or more of a viscous fluid. Examples of highly viscous fluids include things like honey or corn syrup, or fluids that are generally "thicker" or "more sticky", while less viscous fluids, like water, are considerably less so. For these numerical experiments we keep all other parameters the same, i.e., all the interpolation parameters, upstroke and downstroke periods, geometry, etc. We considered fluid dynamic viscosities, μ, across 5 orders of magnitude from 0.05 to 5000. Note that the viscosity considered in all previous cases (Sections 4.1-4.3) was $\mu = 10$.

As briefly stated earlier, this is equivalent to varying the Reynolds Number, Re, which describes the ratio of inertial to viscous forces, which is quantitatively given by

$$Re = \frac{\rho V L}{\mu}. \tag{4.2}$$

Note that ρ and μ are the fluid's density and dynamic viscosity, respectively, while L and V are characteristic length and velocity scales for the system. We will not go into more depth regarding Reynolds Number; more information regarding Re "scaling" studies can be found in [13, 19, 11, 7, 5, 26]. Let's see how these idealized swimmers perform in different viscosities!

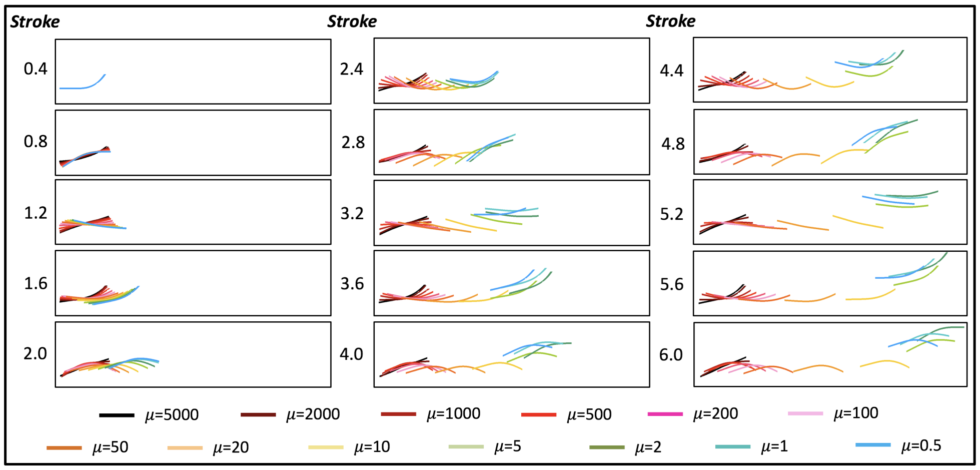


FIG. 23. *Snapshots from simulations with varying fluid viscosities.*

Snapshots from simulations of various swimmers in fluids with different viscosities are provided in Figure 23. The corresponding movie is provided in the Supplementary Materials (`Supplement/Swimmer/Viscosity_Race/`). Qualitatively it appears that swimming performance of our idealized anguilliform swimmer decreases as viscosity increases. When the fluid is "thick" or "sticky"-enough, the swimmer may not even able to move forward with this set of model parameters (see the $\mu = 5000$ case) unlike its anguilliform counterparts in less viscous fluid! This is confirmed in Figure 24, which gives the distance swam (bodylengths) vs. swimming strokes performed and average forward swimming speed (bodylengths/stroke) vs viscosity (μ). Interestingly, it appears that this particular anguilliform swimmer has a maximum speed at a particular viscosity around $\mu \sim 5$. That is, in this model of anguilliform locomotion, simply putting the swimmer into less and less viscous fluid will not always result in a faster swimming speed. How do you think this would change if you varied some of the interpolation parameters, (p_1, p_2), or the stroke frequency?

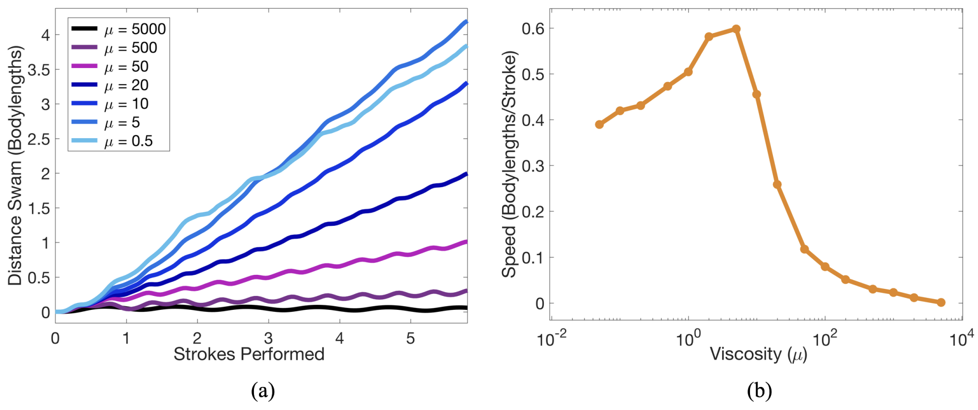


Fig. 24. *(a) Forward distance swam vs swimming strokes performed and (b) swimming speed (bodylengths/stroke) vs. viscosity.*

5. Discussion. Hopefully this has convinced you that there are some practical uses of interpolation in mathematical modeling, which are not generally discussed in traditional numerical analysis settings. In this paper we illustrated a few of the possibilities when applying spline interpolation techniques to mathematical modeling, including prescribing movement patterns (Sections 2 and 3) and material property states (Section 4). In particular, we demonstrated the following practical aspects of interpolation in mathematical modeling:

1. Interpolation can be used to prescribe the motion of an object.
2. Interpolation can be used to switch between different material property states of an object, which can give rise to unsuspecting, interesting dynamics.
3. When using spline interpolants, the number of continuous derivatives affects the resulting dynamics of the system. That is, it does not only matter that you get from A to B, but also *how you get there*, in terms of velocities and accelerations.
4. Thus to relinquish modeling artifacts, one could design their interpolant to match observed velocities and accelerations from experimental data, if possible.
5. Even when not prescribing the precise movement of an object, but rather the object's material property states (e.g., curvature), changing the spline interpolant affects the system's outcome.
6. In fact, subtly changing aspects of the interpolant can lead to significant changes in the unveiling dynamics.

We note that the simulations in Sections 2 and 3 were designed on a coarse mesh so that students can run them locally on laptops in a manner of a few minutes. However the swimmer simulations in Section 4 were constructed on much finer meshes, which have been observed to be required for locomotion previously [8]. Each of the swimmer simulations takes on the order of ~ 2 hours on a personal machine ($\sim$4-16GB RAM, $\sim$2-3GHz processor). In all of these examples, students have the opportunity to experience scientific computing research in practice, e.g., simulations that can greatly vary in computational time, produce a lot of data with non-trivial data analysis, and open the floor for discussions on effective data visualization.

The main purpose of this work was to bring interpolation to life for students, allowing them to visually witness how subtle differences in interpolation techniques can lead to significant differences in dynamics, particular within mathematical models. For this reason all codes, both simulation and analysis scripts, are made available. To that extent, this work allows students the opportunity to ask a variety of questions (e.g., such as those posed in Section 4), explore, and chase their answers. This encourages students to 'play' in a numerical and mathematical setting, experiencing mathematical material in a possibly unfamiliar way. Francis Su, former MAA President, has publicly said, "*Play is part of human flourishing. You cannot flourish without play. And if mathematics is for human flourishing, we should "play up" the role of play in how we teach and who we teach... and teaching play is hard work*" [34]. Granting students opportunities to take what can sometimes be digestible, but dry material, such as interpolation, and allowing them to get their hands dirty by experiencing its utility in mathematical models at the interface of education and contemporary research, could have a profound impact on their future mathematical or scientific journeys.

Acknowledgments. The author 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. He would also like to thank Austin Baird, Aaron Barrett, Christina Battista, Robert Booth, Karen Clark, Jana Gevertz, Christina Hamlet, Alexander Hoover, Shannon Jones, Andrea Lane, Laura Miller, Matthew Mizuhara, Arvind Santhanakrishnan, Michael Senter, Christopher Strickland, and Lindsay Waldrop for comments on the design of the *IB2d* software and suggestions for examples. He also wants to acknowledge his Spring 2018 Numerical Analysis class at The College of New Jersey (Yaseen Ayuby, Shalini Basu, Gina Lee Celia, Rebecca Conn, Alexander Cretella, Robert Dunphy, Alyssa Farrell, Sarah Jennings, Edward Kennedy, Nicole Krysa, Aidan Lalley, Jason Miles, Jessica Patterson, Brittany Reedman, Angelina Sepita, Nicole Smallze, Briana Vieira, and Ursula Widocki) as the original motivators for this project. Computational resources were provided by the NSF OAC #1826915 and the NSF OAC #1828163. Support 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 regarding *IB2d* and the Immersed Boundary Method (IB).

Here we will touch upon the major points regarding the fluid-structure interaction software used for computations, *IB2d*, as well as the numerical method it is built upon, the *immersed boundary method* (IB).

A.1. *IB2d*.

Biological fluid dynamics is a vast subject, in which nearly encompasses the entire natural world around us. From the way birds fly, fish swim, or the way you've taken a couple breaths in the past few seconds, fluid dynamics, or more precisely, fluid-structure interactions are ever present. Unfortunately, for such a significant practical area of mathematical modeling, it traditionally comes with a very steep learning curve, making it challenging to teach educational modules or give students meaningful first hand experience in course projects. Our open source software, *IB2d*, was designed specifically for these purposes. It has two full implementations in high-level programming environments most familiar to most undergraduate students, MATLAB and Python.

IB2d was created to be used for both teaching and research purposes. It comes equipped with over 60 built in examples that allow students to explore the world of fluid dynamics and fluid-structure interaction, from examples that illustrate fluid dynamics principles, such as flow around a cylinder for multiple Reynolds Numbers or the Rayleigh-Taylor Instability, to examples that purely illustrate interactions of a fluid with different immersed structure material properties to biological examples, such as jellyfish locomotion or embryonic heart development. Some of these examples are highlighted in [4, 10, 9]. Therefore *IB2d* can be used for either course projects or homework assignments for a multitude of courses, ranging from mathematical modeling and mathematical biology courses to fluid mechanics to scientific computing. It has also been used for research purposes [29, 26].

For these reasons, there have been tutorial videos created to help acquaint one with the software. All tutorial videos be found at github.com/nickabattista/IB2d:

- **Tutorial 1**: https://youtu.be/PJyQA0vwbgU
  An introduction to the immersed boundary method, fiber models, open source IB software, IB2d, and some FSI examples!
- **Tutorial 2**: https://youtu.be/jSwCKq0v84s
  A tour of what comes with the IB2d software, how to download it, what Example sub-folders contain and what input files are necessary to run a simulation
- **Tutorial 3**: https://youtu.be/I3TLpyEBXfE
  The basics of constructing immersed boundary geometries, printing the appropriate input file formats, and going through these for the oscillating rubberband example from Tutorial 2
- **Tutorial 4**: https://youtu.be/4D4ruXbeCiQ
  The basics of visualizing data using open source visualization software called VisIt (by Lawrence Livermore National Labs). Using the oscillating rubberband from Tutorial 2 as an example to visualize the Lagrangian Points and Eulerian Data (colormaps for scalar data and vector fields for fluid velocity vectors)

More explicit details about *IB2d*'s functionality can be found in [4, 10, 9].

A.2. Governing Equations of IB. In this section we will introduce the equations of fluid motion and how they can be coupled with the motion and deformations of an immersed body. The conservation of momentum equations that govern an incompressible and viscous fluid are written as the following set of coupled partial differential equations,

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

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

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, $\mathbf{x}$. We note that (A.1) is the conversation of momentum, while (A.2) is the conversation of mass, for an incompressible fluid.

The equations that couple the motion of the fluid to deformations of the structure are written as integral equations. These *interaction* equations handle all communication between the fluid (Eulerian) grid and immersed boundary (Lagrangian grid).

They are given as the following integral equations with delta function kernels,

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

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

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. (A.3) applies this force from the immersed boundary to the fluid through the external forcing term in (A.1). Equation (A.4) moves the boundary at the local fluid velocity. This enforces the no-slip condition. Each integral transformation uses a three-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 (A.3), is specific to the application. To either hold the geometry nearly rigid or prescribe the motion of the immersed structure, all of the Lagrangian points along the immersed boundary are tethered to target points. They can do this through a penalty force formulation of $\mathbf{f}(s,t)$. In this paper, in Sections 2 and Section 3, we have used target points to prescribe the motion of the immersed structure. The penalty force was written in the following way,

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

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, and that in these models k_{targ} was chosen to be large so that it would effectively drag the Lagrangian points into the preferred positions.

In Section 4, we construct a swimmer that is composed of springs and beams. Springs allow for stretching and compressing of the successive Lagrangian points, while beams allow for bending. Their corresponding deformation force equations can be written as the following,

$$\mathbf{F}_{spr} = -k_{spr}\left(1-\frac{R_L}{||\mathbf{X}_S-\mathbf{X}_M||}\right)\cdot\left(\mathbf{X}_M-\mathbf{X}_S\right). \tag{A.6}$$

$$\mathbf{F}_{beam} = -k_{beam}\frac{\partial^4}{\partial s^4}\Big(\mathbf{X}(s,t)-\mathbf{X}_B(s,t)\Big), \tag{A.7}$$

where k_{spr} and k_{beam} are the spring stiffness and beam stiffness coefficients for springs and beams, respectively. For the linear spring forces, the terms X_M and X_S represent the positions in Cartesian coordinates of the master and slave Lagrangian nodes at time, t, and R_L is the spring's corresponding resting length. For the bending force, $\mathbf{X}_B(s,t)$ represents the preferred curvature of the configuration at time, t. We note that in the swimmer model of Section 4, we interpolate between different curvature states given by different configurations of $\mathbf{X}_B^a(s,t)$ and $\mathbf{X}_B^b(s,t)$, rather than interpolate between positions in space for the swimmer.

Using delta functions as the kernel in (A.3)-(A.4) is the heart of IB. To approximate these integrals, discretized (and regularized) delta functions are used. We use the ones given from [30], e.g., $\delta_h(\mathbf{x})$,

$$\delta_h(\mathbf{x}) = \frac{1}{h^3}\phi\left(\frac{x}{h}\right)\phi\left(\frac{y}{h}\right)\phi\left(\frac{z}{h}\right), \tag{A.8}$$

where $\phi(r)$ is defined as

$$\phi(r) = \begin{cases} \frac{1}{8}(3-2|r|+\sqrt{1+4|r|-4r^2}), & 0 \le |r| < 1 \\ \frac{1}{8}(5-2|r|+\sqrt{-7+12|r|-4r^2}), & 1 \le |r| < 2 \\ 0 & 2 \le |r|. \end{cases} \tag{A.9}$$

A.2.1. Numerical Algorithm. As stated in the main text, we impose periodic and no slip boundary conditions on a rectangular domain. To solve A.1), (A.2),(A.3) and (A.4) we need to update the velocity, pressure, position of the boundary, as well as the force acting on the boundary at time $n+1$ using data from time n. The IB does this in the following steps [30, 10]:

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

Step 2: Use (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, (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 $\mathbf{f}^n$.

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

REFERENCES

[1] Adobe Systems, *Designing multiple master typefaces*, 1997, https://www.adobe.com/content/dam/acom/en/devnet/font/pdfs/5091.Design_MM_Fonts.pdf.

[2] S. Alben, L. A. Miller, and J. Peng, *Efficient kinematics for jet-propelled swimming*, J. Fluid Mech. 733, 733 (2013), pp. 100–133.

[3] A. J. Baird, T. King, and L. A. Miller, *Numerical study of scaling effects in peristalsis and dynamic suction pumping*, Biological Fluid Dynamics: Modeling, Computations, and Applications, 628 (2014), pp. 129–148.

[4] N. A. Battista, A. J. Baird, and L. A. Miller, *A mathematical model and matlab code for muscle-fluid-structure simulations*, Integr. Comp. Biol., 55(5) (2015), pp. 901–911.

[5] N. A. Battista, D. Douglas, A. Lane, L. Samsa, J. Liu, and L. Miller, *Vortex dynamics in embryonic trabeculated ventricles*, J. Cardiovasc. Dev. Dis., 6(1) (2019), p. 6.

[6] N. A. Battista, A. N. Lane, J. Liu, and L. A. Miller, *Fluid dynamics of heart development: Effects of trabeculae and hematocrit*, Math. Med. and Biol., 35(4) (2018), pp. 493–516.

[7] N. A. Battista, A. N. Lane, and L. A. Miller, *On the dynamic suction pumping of blood cells in tubular hearts*, in Women in Mathematical Biology: Research Collaboration, A. Layton and L. A. Miller, eds., Springer, New York, NY, 2017, ch. 11, pp. 211–231.

[8] N. A. Battista and M. S. Mizuhara, *Fluid-structure interaction for the classroom: Speed, accuracy, convergence, and jellyfish!*, arXiv: https://arxiv.org/abs/1902.07615, (2019).

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

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

[11] J. Baumgart and B. M. Friedrich, *Fluid dynamics: Swimming across scales*, Nature Physics, 10 (2014), p. 711712.

[12] A. D. Becker, H. Masoud, J. W. Newbolt, M. Shelley, and L. Ristroph, *Hydrodynamic schooling of flapping swimmers*, Nature Communications, 6 (2015), p. 8514.

[13] I. Borazjani and F. Sotiropoulos, *Numerical investigation of the hydrodynamics of carangiform swimming in the transitional and inertial flow regimes*, J. Exp. Biol., 211 (2008), pp. 1541–1558.

[14] R. L. Burden, D. J. Faires, and A. M. Burden, *Numerical Analysis (10th edition)*, Cengage Learning, Boston, MA, USA, 2014.

[15] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, D. Pugmire, K. Biagas, M. Miller, C. Harrison, G. H. Weber, H. Krishnan, T. Fogal, A. Sanderson, C. Garth, E. W. Bethel, D. Camp, O. Rübel, M. Durant, J. M. Favre, and P. Navrátil, *VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data*, in High Performance Visualization–Enabling Extreme-Scale Scientific Insight, Chapman and Hall/CRC Press, Oct 2012, pp. 357–372.

[16] C. Hamlet, K. A. Hoffman, E. D. Tytell, and L. J. Fauci, *The role of curvature feedback in the energetics and dynamics of lamprey swimming: A closed-loop model*, PLoS Comp. Biol., 14(8) (2018), p. e1006324.

[17] C. Hamlet and L. A. Miller, *Feeding currents of the upside-down jellyfish in the presence of background flow*, Bull. Math. Bio., 74(11) (2012), pp. 2547–2569.

[18] M. Heath, *Scientific Computing*, McGraw-Hill, New York, NY, USA, 2002.

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

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

[21] D. Kincaid and W. Cheney, *Numerical Analysis*, American Mathematical Society, Providence, RI, 2002.

[22] J. Lee, M. E. Moghadam, E. Kung, H. Cao, T. Beebe, Y. Miller, B. L. Roman, C.-L. Lien, N. C. Chi, A. L. Marsden, and T. K. Hsiai, *Moving domain computational fluid dynamics to interface with an embryonic model of cardiac morphogenesis*, PLoS One, 8 (2013), p. e72924.

[23] S. Marcus, M. Frigura-Iliasa, D. Vatau, and L. Matiu-Iovan, *New interpolation tools for digital signal processing*, in 2016 International Conference on Information and Digital Technologies (IDT), 2016, pp. 266–270.

[24] T. McMillen and P. Holmes, *An elastic rod model for anguilliform swimming*, Journal of Mathematical Biology, 53 (2006), pp. 843–886.

[25] T. McMillen, T. Williams, and P. Holmes, *Nonlinear muscles, passive viscoelasticity and body taper conspire to create neuromechanical phase lags in anguilliform swimmers*, PLoS Comp. Bio., 4(8) (2008), p. e1000157.

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

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

[28] L. T. Nielsen, S. S. Asadzadeh, J. Dolger, J. H. Walther, T. Kiorboe, and A. Andersen, *Hydrodynamics of microbial filter feeding*, PNAS, 114(35) (2017), pp. 9373–9378.

[29] F. PAllasdies, S. Goedeke, W. Braun, and R. Memmesheimer, *From single neurons to behavior in the jellyfish Aurelia aurita*, biorXiv: https://www.biorxiv.org/content/10.1101/698548v1, (2019). https://doi.org/10.1101/698548.

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

[31] S. Ruck and H. Oertel, *Fluid-structure interaction simulation of an avian flight model*, J. Exp. Biol., 213 (2010), pp. 4180–4192.

[32] C. Runge, *Uber empirische funktionen und die interpolation zwischen quidistanten ordinaten*, Zeitschrift fr Mathematik und Physik, 46 (1901), pp. 224–243.

[33] J. E. Samson, N. A. Battista, S. Khatri, and L. A. Miller, *Pulsing corals: a story of scale and mixing*, BIOMATH, 6(2) (2017), p. 1712169.

[34] F. E. Su, *Mathematics for human flourishing*, The American Mathematical Monthly, 124(6) (2017), pp. 483–493.

[35] M. Unser, *Splines: A perfect fit for signal and image processing*, IEEE Signal Processing Magazine, 16(6) (1999), pp. 22–38.

[36] J. Vince, *Vector Analysis for Computer Graphics*, Springer, Berlin, Germany, 2007.

[37] World Wide Web Consortium (W3C), *Svg 1.1 (second edition)*, 2011, https://www.w3.org/TR/SVG11/fonts.html.