JOURNAL OF
EMERGING INVESTIGATORS

Article

Study of PINN sensor layout in evaluating WSS with application to patient-specific carotid flow

Eric Nie[1], Jianhu Nie[2]
[1] Phillips Academy, Andover, Massachusetts
[2] Global Technology Center, Johnson Controls, Cranston, Rhode Island

SUMMARY

Atherosclerosis, a chronic inflammatory disease characterized by lipid-rich plaque buildup within arterial walls, is a major contributor to stroke and heart attack. It is strongly influenced by blood flow characteristics, such as wall shear stress (WSS). Traditional methods like computational fluid dynamics (CFD) and magnetic resonance imaging (MRI) struggle to accurately capture near-wall flow due to the complex and uncertain nature of blood dynamics. This study investigates whether sensor density and location affect the accuracy of Physics-Informed Neural Networks (PINNs) in modeling blood flow in carotid artery bifurcations. We hypothesized that: (a) increasing the number of sensors would improve model accuracy due to increased quantities of probed data, and (b) sensors placed closer to the artery wall would more accurately predict WSS, better capturing the flow gradient within the boundary layer. Using idealized bifurcation models, we tested various sensor configurations. Higher sensor densities improved accuracy to a certain extent, with no significantly positive impact beyond Nsample = 100, and sensors placed at 10% of the radius from the wall produced balanced predictions for both the near-wall flow and the general flow patterns. To test real-world applicability, we applied the optimized PINN model to a patient-specific carotid geometry and achieved an average velocity error of only 0.0037 m/s. These results successfully demonstrate how sensor configuration significantly affects PINN accuracy and highlight PINNs' potential as an efficient tool for personalized diagnosis of cardiovascular disease.

INTRODUCTION

Atherosclerosis is a prevalent precursor to various cardiovascular conditions, such as coronary, carotid, and peripheral artery disease (1). The deposition of atherosclerotic plaque, composed of lipids, cholesterol, and other substances, onto the endothelium induces stenosis by obstructing blood flow in the artery. Carotid atherosclerosis is particularly significant because plaque buildup in the carotid arteries directly increases the likelihood of severe cardiovascular events, such as stroke and myocardial infarction (2, 3). As low or oscillating levels of wall shear stress (WSS) promote endothelial dysfunction by reducing endothelial nitric oxide production, the development of atherosclerosis and plaque formation is strongly affected by hemodynamic conditions (4).

Located where the common carotid artery (CCA) branches into the internal carotid artery (ICA) and external carotid artery (ECA), carotid artery bifurcations are particularly prone to plaque formation due to their geometric configuration. Specifically, atherosclerotic plaque formation is notably prevalent in the widened portion of the ICA, known as the carotid sinus, due to hemodynamic factors, such as low WSS and high WSS gradients (5–7). In fact, numerous studies have investigated the hemodynamic effects of various geometric variations, including bifurcation angle and the severity and localization of stenosis (8–11).

However, due to the sharp curvature and flow disturbances present, such as recirculation zones, vortex formation, and flow separation, blood flow within stenosed and bifurcated arteries is often complex; the presence of disturbed flow patterns complicates the precise quantification of hemodynamic parameters, making obtaining essential hemodynamic metrics, especially near-wall flow and WSS, challenging (12, 13). Furthermore, the non-Newtonian properties of blood, particularly in regions of low shear rates, introduce additional challenges in accurately modeling and measuring WSS (14, 15).

Given these difficulties, various experimental and computational methodologies have been employed to analyze hemodynamic forces and their influence on atherosclerotic plaque progression. With rapid advances in computer hardware and software, computational fluid dynamics (CFD) has become an essential tool for simulating vascular flow and obtaining high-resolution hemodynamic parameters over space and time (16). Nevertheless, CFD simulations are prohibitively expensive and time-consuming. Moreover, the simulations are highly sensitive to boundary conditions, model assumptions, and imaging-derived arterial geometries, leading to potential uncertainties in WSS estimation (17, 18).

In addition to CFD, in vivo and in vitro techniques, such as velocity measurements based on magnetic resonance imaging (MRI), particle image velocimetry (PIV), and ultrasound Doppler imaging, have been utilized to assess blood flow characteristics in carotid bifurcations (19–21). While these methods provide valuable insights into arterial hemodynamics, they are often limited by spatial resolution, acquisition time, and measurement accuracy, especially in detecting near-wall flow phenomena (13).

As conventional CFD and experimental methods often struggle to fully capture the nonlinear dynamics of patient-specific flow, we consider Physics-Informed Neural Networks (PINNs) as a promising approach for modeling complex hemodynamic systems (22, 23). These neural networks integrate governing laws, such as the Navier-Stokes equations that describe how fluid velocity, pressure, and forces interact to govern fluid motion, into their loss functions in evaluating a model's prediction accuracy, making PINNs capable of learning solutions that are consistent with the underlying

https://doi.org/10.59720/25-261

physics. This approach allows for the incorporation of sparse and noisy experimental data, making it particularly suitable for biomedical applications where acquiring high-fidelity data is often challenging (22). Since their introduction in 2017, PINNs have gained significant traction in computational science and engineering (22). Across areas such as fluid flow, heat transfer, and biomedical simulation, PINNs have proven effective, demonstrating their potential to unite traditional physics-based models and modern data-driven approaches.

In this study, we explored the application of PINNs to model blood flow in carotid artery bifurcations, focusing on their ability to accurately predict near-wall flow and WSS. The impact of sensor location and density on the accuracy of PINN predictions has not been extensively explored. By systematically varying the placement and number of sensors within the carotid sinus, we aimed to identify optimal configurations for capturing near-wall flow phenomena and WSS gradients. We hypothesized that higher sensor density and increased sensor proximity to walls will result in more accurate predictions for near-wall flow and WSS, with the latter impacting the accuracy of the model more significantly due to the steep velocity gradients located at the wall. However, both these qualities require more precise imaging, which current methods lack. Additionally, we evaluated the patient-specific applicability of PINNs by applying the methodology to a patient-derived carotid artery geometry, comparing the results with ground-truth CFD simulations. This approach highlighted the potential of PINNs to bridge the gap between idealized models and real-world clinical scenarios, where patient-specific geometries and flow conditions are critical for accurate hemodynamic assessment. We found that increasing sensor density improves predictive accuracy, up to N_{sample} = 100, where accuracy gains plateau. Additionally, sensor placement exhibited region-specific effects, with sub-boundary flow accuracy optimized at 0.8706 arterial radius (R) and near-wall accuracy improved at more proximal locations (0.95 R). A placement of 0.9 R provided balanced performance, and when applied to a patient-specific carotid bifurcation model, the optimized configuration achieved close agreement with CFD results, demonstrating the potential for PINNs for effective and clinically applicable hemodynamic prediction.

RESULTS

To determine how sensor density and placement influence the accuracy of PINNs, we first validated a CFD model against experimental measurements to establish a reference flow field. We then systematically varied sensor density and location to evaluate PINN performance within an idealized carotid bifurcation before testing the optimized configuration in a patient-specific model. We found that PINN accuracy improved with increasing sensor density up to a threshold, after which gains plateaued, and that sensor placement had region-specific effects on both sub-boundary and near-wall flow prediction.

Ground-Truth Validation

First, we sought to determine the accuracy of the CFD model as a reference, or "ground truth," for evaluating PINNs. The CFD model offers a high-resolution, continuous representation of the flow field, as opposed to the limited spatial resolution of experimental measurements. To validate the model's accuracy, we compared the computed axial velocity profiles at selected locations in the carotid with published laser-Doppler measurements of steady-state, Newtonian flow in an identical carotid bifurcation geometry (14) (**Figure 1, Table 1**). The locations labeled l_{00}–l_{20} represent parallel lines across the carotid sinus with increasing distance from the region of flow separation. Numerical and experimental results agreed strongly overall, with minor discrepancies observed closest to regions of flow separation (l_{00} and l_{05}). These differences diminished further downstream, with predictions at l_{20} closely matching the measured flow profile (**Figure 1**).

Calculated normalized root-mean square error (nRMSE)

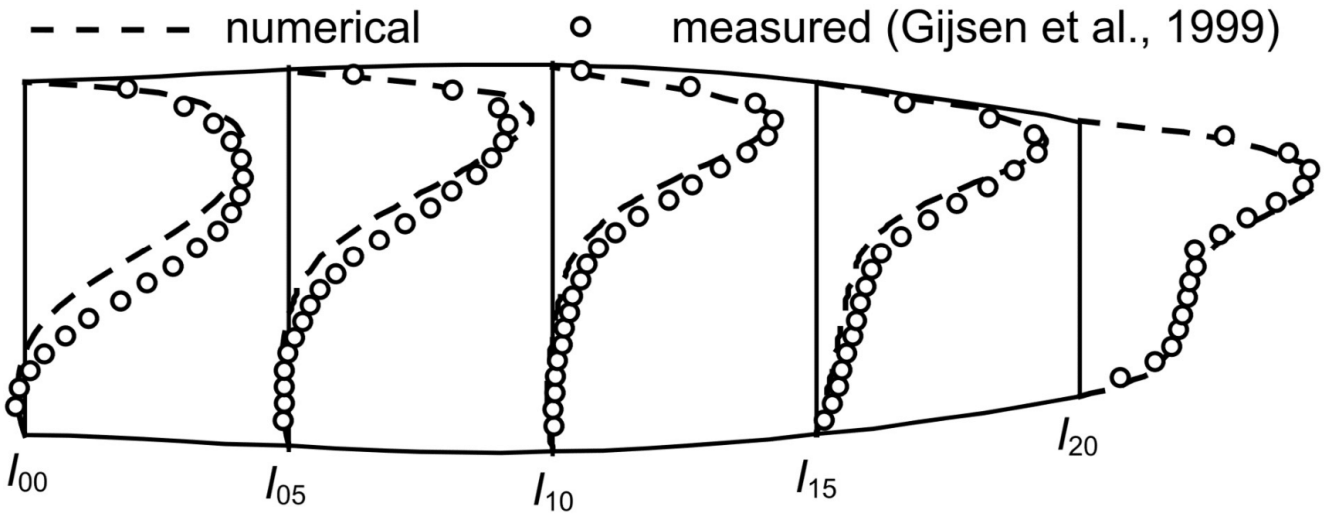

Figure 1. The axial velocity distributions of computational fluid dynamics (CFD) ground-truth: numerical vs. measured. To verify the CFD ground-truth (dashed line) to evaluate Physics-Informed Neural Network (PINN) performance, the axial velocity profiles at lines l_{00}–l_{20}across the carotid sinus were compared with published laser-Doppler measurements (circled points) (12).

https://doi.org/10.59720/25-261

Line	Mean Error (m/s)	RMSE (m/s)	nRMSE
I_{00}	0.0811	0.1662	0.1926
I_{05}	0.0454	0.1391	0.1633
I_{10}	0.0555	0.1035	0.1239
I_{15}	0.0714	0.0866	0.1072
I_{20}	0.0573	0.0696	0.0970

Table 1. Ground truth validation summary. Mean velocity errors (m/s), root-mean square error (m/s), and normalized root-mean square values (nRMSE) are reported for the computational fluid dynamics (CFD) reference, known as the ground truth, relative to laser-Doppler measurements at lines I_{00}–I_{20} across the carotid sinus (12). With nRMSE values across all lines below 0.2, CFD results agreed well with the observed data, justifying the present flow simulation model and providing confidence for the next simulations.

quantitatively supported this trend. The nRMSE values decreased with decreased proximity to the bifurcation, ranging from 0.1926 at the upstream location (I_{00}) to 0.0970 at the downstream location (I_{20}), indicating improved prediction accuracy as lower nRMSE indicates better agreement between predicted and reference values (**Table 1**). All CFD-derived nRMSEs were below 0.2, indicating good agreement with experimental data; these consistently low values support the validity of the present flow simulation model for subsequent simulation tests.

Sensor Density

After establishing and validating the CFD model as the ground-truth representation of blood flow within the idealized carotid bifurcation, we next evaluated the ability of PINNs to reconstruct the flow field from sparse velocity measurements. To investigate how the availability of measurement data influences PINN accuracy, we evaluated the effect of varying sensor densities within the idealized carotid sinus. We modified N_{sample} values, controlling sensor density by placing sensors every Nsample points within the sensor domain. We constructed velocity profiles for ground-truth (CFD), 1159 (N_{sample} = 50), 579 (N_{sample} = 100), and 289 (N_{sample} = 200) sensors, representing velocity magnitudes along the sub-boundary layer region (0.8706 of the arterial radius) of the upper carotid sinus (**Figure 2a**). The average difference between the ground-truth and N_{sample} = 200 velocity magnitudes was 0.012 m/s (p = 0.470, d = 0.24), while the average differences for N_{sample} = 100 (p = 1.000, d = 0.09) and N_{sample} = 50 (p = 0.466, d = 0.24) velocity magnitudes when compared to ground-truth data was less than 0.01 m/s. Given a significance threshold of α = 0.0056, the differences for all evaluated PINN sample densities and CFD ground truth were not statistically significant. Additionally, with $|d|$ < 0.5 for all tested sample densities, the effect size is considered small (**Table 2**).

Sample Location

To assess how the spatial placement of measurements affects PINN accuracy, evaluated the effect of sensor location within the idealized carotid sinus. We defined sensor domains by their radial positions divided by vessel radius (R), with sensors placed at 0.8706 R (sub-boundary layer), 0.9 R, and 0.95 R (near-wall region). We then measured the effect of sensor placement on PINN accuracy on hemodynamic quantities (**Figure 2b**). We ran a PINN model with N_{sample} = 50

Density (N_{sample})	Sensor Location	Region	Mean Error (m/s)	p-value	Effect Size
50	Full Domain	Sub-boundary	+0.0072	0.466	d = 0.24 (△)
100	Full Domain	Sub-boundary	+0.0039	1.000	d = 0.09 (△)
200	Full Domain	Sub-boundary	+0.0122	0.470	d = 0.24 (△)
50	0.8706 R	Sub-boundary	+0.0039	0.569	d = 0.11 (△)
50	0.9 R	Sub-boundary	-0.0212	0.056	d = -0.40 (△)
50	0.95 R	Sub-boundary	-0.0577	**7.3e-08***	d = -1.17 (▼)
50	0.8706 R	Near-wall	-0.0097	**3.5e-15***	d = -1.03 (▼)
50	0.9 R	Near-wall	-0.0068	**8.9e-06***	d = -0.53 (○)
50	0.95 R	Near-wall	-0.0066	**1.0e-08***	d = -0.70 (○)

Table 2. Sensor Configuration Performance Summary. Mean velocity errors are reported relative to computational fluid dynamics (CFD) ground truth, with significance assessed using one-sample t-tests (Bonferroni-corrected α = 0.0056). Effect sizes (Cohen's d) are indicated. Sensor densities (N_{sample} = 50–200) and normalized radial locations (0.8706–0.95 R) are varied to observe impact on PINN accuracy. Bolded values represent statistically significant velocity differences between the ground truth and PINN prediction.
Symbol Key:▼ Large effect ($|d|$ ≥ 0.8) | ○ Medium (0.5 ≤ $|d|$ < 0.8) | ▲ Small ($|d|$ < 0.5)
*Significant after Bonferroni correction (α = 0.0056 for 9 tests)

https://doi.org/10.59720/25-261

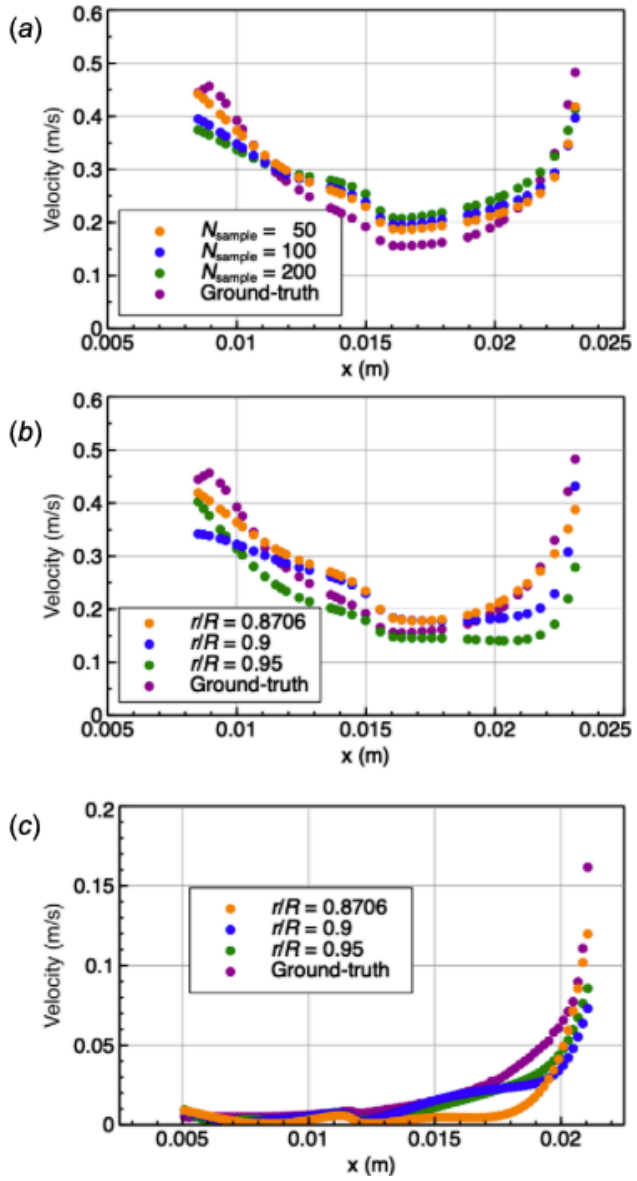

Figure 2. Plot of velocity magnitude (m/s) against x-coordinates (m) on idealized carotid sinus from Physics-Informed Neural Network (PINN) with varied Nsample values and sensor localization. (a) Datasets with N_{sample} = 50, 100, and 200 (orange, blue, green, respectively) have differing levels of accuracy along the sub-boundary layer velocity magnitude compared to ground-truth (purple). N_{sample} = 100 was most accurate (mean error = 0.0039 m/s, p = 0.466, d = 0.24) and N_{sample} = 200 was least accurate (mean error = 0.0122 m/s, p = 0.470, d = 0.24). (b) Datasets with sensor localization of 0.8706 R, 0.9 R, and 0.95 R (orange, blue, green, respectively) have differing levels of accuracy along the sub-boundary layer velocity magnitude compared to ground-truth (purple). 0.8706 R (mean error = 0.0039 m/s, p = 0.569, d = 0.11) was most accurate and 0.95 R (mean error = -0.0577 m/s, p = 7.3×10^{-8}, d = -1.17) was least accurate. (c) Datasets with sensor localization of 0.8706 R, 0.9 R, and 0.95 R (orange, blue, green, respectively) have differing levels of accuracy along the near-wall velocity magnitude compared to ground-truth (purple). 0.95 R was most accurate (mean error = -0.0066 m/s, p = 1.0×10^{-8}, d = -0.70) and 0.8706 R was least accurate (mean error = -0.0097, p = 3.5×10^{-15}, d = -1.03). The ground-truth (computational fluid dynamics (CFD)) velocity profile for each location is also shown for comparison.

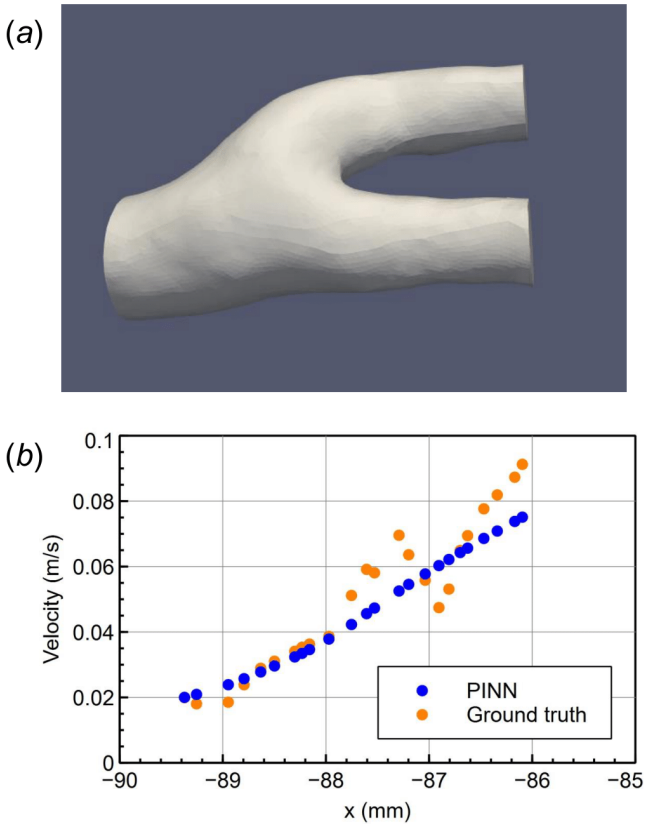

Figure 3. Patient-specific geometry and plot of near-wall velocity magnitude (m/s) against *x*-coordinates (mm) patient-specific carotid sinus. (a) Patient-specific geometry of right carotid bifurcation used in Physics-Informed Neural Network (PINN) simulations obtained from Eulzer *et al.* (24). (b) Sensors were located at 0.9 R, with N_{sample} = 100 (blue). The ground-truth (CFD) velocity profile (orange) from Eulzer *et al.* is also shown for comparison (24).

for 3000 epochs with each sensor domain. We constructed velocity profiles across the sub-boundary layer (0.8706 R) of the upper carotid sinus for ground truth and PINN models with varying sensor location. We then found average differences in velocity magnitude compared to ground-truth CFD for 0.95 R (mean error = -0.0577 m/s, p = 7.3×10^{-8}, d = -1.17), 0.9 R (mean error = -0.0212 m/s, p = 0.056, d = -0.40), and 0.8706 R (mean error = 0.0039 m/s, p = 0.569, d = 0.11) models. While 0.9 R and 0.8706 R did not reach statistical significance at α = 0.0056 and exhibited small effect size with $|d| < 0.5$, 0.95 R reached both a significant p-value and large effect size ($|d| \geq 0.8$), indicating substantial and systematic deviation from the CFD ground-truth.

Furthermore, we constructed velocity profiles across the near-wall region of the upper carotid sinus, which shows strong correlation with WSS due to WSS being a gradient of radial velocity (**Figure 2c**). Average differences in velocity magnitude for 0.95 R, 0.9 R, and 0.8706 R models were -0.0066 m/s, -0.0068 m/s, and 0.0097m/s, respectively. As such, although we observed substantial differences between sensor placement of 0.8706 R (p = 3.5×10^{-15}, d = -1.03) and 0.9 R (p = 8.9×10^{-6}, d = -0.53), there were minimal differences between 0.9 R and 0.95 R (p = 1.0×10^{-8}, d = -0.70). All sample locations exhibited significantly different velocities compared

https://doi.org/10.59720/25-261

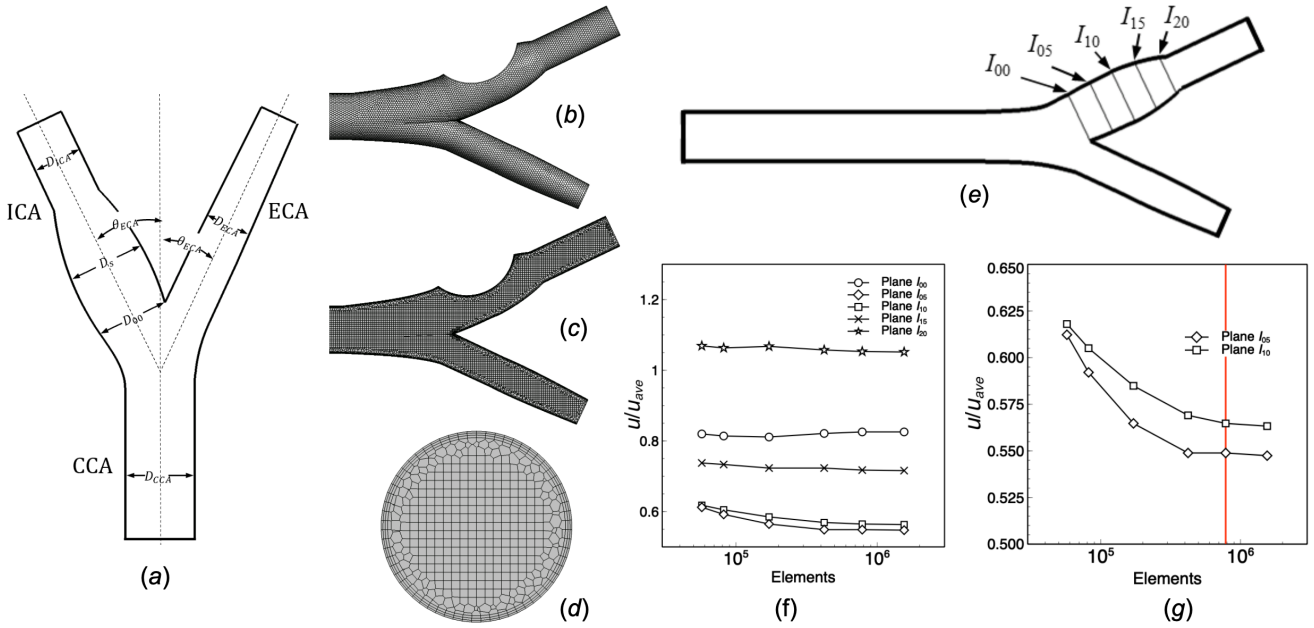

Figure 4. Labeled idealized carotid bifurcation geometry, cross-section of computational mesh, and computational fluid dynamics (CFD) grid independence test. (a) The geometry of the idealized carotid bifurcation used in CFD ground-truth construction and Physics-Informed Neural Network (PINN) sensor testing is shown. (b) Sample computational mesh of idealized carotid bifurcation geometry. (c) Polyhedral volume elements with hexahedral core were employed in the simulation, (d) with the finer boundary layer mesh near the solid walls where the flow gradient was expected to be high, in order to ensure the accuracy of the numerical simulation. (e) Planes I_{00}- I_{20}. are labeled within the carotid sinus of the internal carotid artery (ICA). (f) Normalized axial fluid velocity for the CFD ground-truth is plotted against element number at planes I_{00}- I_{20}. to determine suitable mesh element sizes. (g) Among the evaluated locations, planes I05 and I10 exhibited the greatest sensitivity to mesh refinement. Velocity variation converged beyond 783,738 elements (red line), justifying the use of the corresponding mesh size (0.4 mm) for all subsequent CFD simulations.

to the CFD ground truth. Medium effect size ($0.5 \leq |d| < 0.8$) was observed for 0.9 R and 0.95 R, and large effect size ($|d| \geq 0.8$) was observed for 0.8706 R, indicating progressively increasing deviation from the CFD solution as sensor location moves further from the near-wall region (**Table 2**).

Patient-Specific Model

To evaluate the applicability of PINNs to realistic anatomical conditions, we constructed a new PINN informed by the trends observed in idealized geometries for the patient-specific right carotid bifurcation model (**Figure 3a**), with N_{sample} = 100 (samples placed every 100 points in the sample domain) and at 0.9 R (90% of the vessel radius). Additionally, we set the fluid's density (ρ) and dynamic viscosity (η) to 1000 kg/m3 and 0.00345 Pa·s, respectively, according to properties of blood outlined inside the source database (24). We obtained flow data from the PINN model after 3000 epochs. We compared near-wall velocity magnitude profiles along the top wall of the carotid sinus with ground-truth data obtained from CFD simulations (**Figure 3b**). The average difference in velocity magnitude was 0.0037 m/s, although the PINN prediction did not capture the localized fluctuation found in the ground-truth between x = 87–88.

DISCUSSION

This study investigated the effect of sensor density and placement on the accuracy of PINNs for modeling blood flow in carotid artery bifurcations. Our statistical analyses support both components of our initial hypothesis. First, we found that increasing sensor density up to N_{sample} = 100 improved velocity prediction accuracy when compared with the CFD ground truth when averaged across all measurement locations. These results demonstrate a clear plateau effect, where accuracy gains diminish beyond N_{sample} = 100, suggesting that physics constraints in the PINN loss function become dominant over additional data points at higher sensor densities.

The investigation of sensor placement revealed more complex, region-dependent effects that were equally supported by our statistical tests. For sub-boundary layer analysis, positioning sensors at 0.8706R provided the most accurate predictions, while 0.95R placement led to clinically meaningful underestimation. Near-wall measurements showed the opposite pattern, with 0.95 R performing adequately but 0.8706R producing significant underestimation. Importantly, although these near-wall deviations were statistically significant, possibly due to the high gradients present, effect sizes indicate a persistent trend. The 0.9 R location emerged as the optimal compromise, delivering balanced accuracy for both regions, while maintaining physiological relevance for WSS estimation.

We confirmed the clinical potential of these findings through patient-specific validation, where the optimized PINN configuration (N_{sample} =100, 0.9 R) achieved close agreement with CFD results. This performance, attained with sparse clinical data, highlights a key advantage over traditional methods: while CFD requires complex meshing and MRI

https://doi.org/10.59720/25-261

struggles with near-wall resolution, our approach provides accuracy through intelligent sensor placement and density optimization. This result demonstrates the adaptability of PINNs to complex geometries and real-world conditions, reinforcing their potential for personalized hemodynamic assessments.

Although CFD can accurately simulate blood flow, it requires complex mesh generation and assumptions for boundary conditions (17, 18). Experimental methods like magnetic resonance imaging (MRI) also face challenges in capturing near-wall flow with high spatial and temporal resolution (13). PINNs, by contrast, offer a computationally efficient and flexible alternative. By combining physics-based modeling with sparse data assimilation, they bypass many limitations of traditional approaches and hold particular promise for clinical settings where detailed flow measurements are impractical (22, 23, 25).

The ability to accurately characterize patient-specific hemodynamics may improve the early identification and management of atherosclerotic disease. Growing evidence suggests that early identification and treatment of vascular dysfunction can substantially affect long-term cardiovascular outcomes; the Progression of Early Subclinical Atherosclerosis (PESA) study, which followed 4,184 healthy adults aged 40-54, found that 63% exhibited subclinical atherosclerosis despite many being classified as low-risk by traditional measures, and younger individuals showed the greatest benefit from aggressive disease control in slowing plaque progression (26). By reconstructing detailed flow fields from sparse clinical measurements, PINNs offer a practical pathway toward personalized cardiovascular risk assessment without the computational expertise of CFD or the imaging limitations of conventional modalities, supporting earlier intervention and improved monitoring of disease progression.

Despite the advantages demonstrated, several limitations remain. Notably, the PINN did not capture a localized fluctuation in the ground-truth velocity profile between x = 87–88 mm for the patient-specific model. This discrepancy likely reflects the combined effects of limited local sensor support in the region and the smoothing bias introduced by the physics-constrained loss function, which prioritizes globally consistent solutions that may miss sharp, spatially confined variations found in nature. In this context, the accuracy of PINN predictions depends on the quality and coverage of training data, and on choices like learning rate, network architecture, and the strength of physical constraints in the loss functions. Future studies should explore adaptive sampling techniques that automatically select informative probe locations during training. Additionally, extending the framework to model time-dependent flow, fluid-structure interactions, and incorporate non-Newtonian blood properties would more accurately reflect physiological conditions.

This study confirms that sensor density and placement strongly influence the predictive performance of PINNs in carotid artery flow modeling, supporting both components of our initial hypothesis. An optimal sensor configuration can improve prediction accuracy for both near-wall and sub-boundary flows. The successful application to a patient-specific model further demonstrates clinical relevance. As PINNs evolve, they offer a promising tool for bridging the gap between idealized modeling and patient-specific cardiovascular diagnostics.

MATERIALS AND METHODS

Ground-Truth Construction (CFD)

The ground-truth, idealized geometry of a three-dimensional model of the carotid bifurcation was derived from published works (14, 27), representing a cylindrical CCA forking into the ECA and the ICA (**Figure 4a, Table 3**).

The dimensionless parameter describing the flow, Re (Reynolds number), is defined as follows:

$$Re = \frac{\rho u_{ave} D_{CCA}}{\eta} \qquad \text{(Equation. 1)}$$

where u_{ave} is the cross-sectional average velocity in the common carotid artery, D_{CCA} is the diameter of the common

Geometric parameter	Dimension
Internal carotid artery diameter, D_{ICA} (mm)	5.6
External carotid artery diameter, D_{ECA} (mm)	4.6
Common carotid artery diameter, D_{CCA} (mm)	8
Carotid sinus maximum diameter, D_s (mm)	8.9
Diameter at Line I_{00}, D_{00} (mm)	8.3
Internal carotid artery angle, θ_{ICA}	25.4°
External carotid artery angle, θ_{ECA}	25.1°

Table 3. Specific geometric parameters of the idealized carotid bifurcation. Dimensions correlate with the geometric parameters of an idealized carotid bifurcation obtained from Gijsen et al., which was also used in their laser-Doppler velocimetry measurements (14). Adopting the same geometry enables direct comparison between the computational fluid dynamics (CFD) results and the experimentally measured velocities. Notably, the common carotid artery diameter (DCCA) serves as the characteristic length scale for calculating the Reynolds number.

https://doi.org/10.59720/25-261

Density (mm)	Elements	Faces	Nodes	$\overline{u_c}$ @ I_{05}	$\overline{u_c}$ @ I_{10}
1.5	57,241	284,248	86,522	0.61226	0.61802
1.0	81,794	401,499	261,846	0.59209	0.60506
0.8	171,401	853,900	555,739	0.56472	0.58489
0.5	421,399	2,383,561	1,757,072	0.54887	0.56904
0.4	783,738	3,426,421	2,009,088	0.54887	0.56472
0.3	1,552,236	7,197,421	4,416,721	0.54743	0.56328

Table 4. Grid independence test. The mesh used to construct the computational fluid dynamics (CFD) ground-truth model was of 0.4 mm element sizes, corresponding with 783,738 elements and 2,009,088 nodes. Using a denser grid with 1,552,236 elements and 4,416,721 nodes resulted in less than 0.5 percent difference in the predicted axial velocity component ($\bar{u}_c$) at lines I_{05} and I_{10}.

carotid artery, and ρ and η are the density and the dynamic viscosity of fluid, respectively.

In order to simulate the flow field, we solved the transient laminar three-dimensional Navier-Stokes equation using the finite volume scheme. These governing equations consist of the continuity equation and the momentum equation.

Continuity equation:

$$\frac{\partial u_i}{\partial x_i} = 0 \qquad \text{(Equation. 2)}$$

Momentum equation:

$$\rho\left(\frac{\partial u_j}{\partial t} + u_i\frac{\partial u_j}{\partial x_j}\right) = -\frac{\partial p}{\partial x_j} + \frac{\partial \tau_{ij}}{\partial x_i} + b_j \qquad \text{(Equation. 3)}$$

where u_i are the fluid velocity components, ρ is the isotropic pressure, b_j are the body forces (here, assumed to be zero), and τ_{ij} are the components of the extra stress tensor.

Numerical solution of the governing equations together with the applied boundary conditions was performed by utilizing the pressure-based coupled algorithm for the pressure velocity coupling in the iteration procedure. Using the coupled scheme offers some advantages in obtaining a robust and efficient solution with superior performance over the non-coupled or segregated approach, such as Semi-Implicit Methods for Pressure-Linked Equations (SIMPLE) (28). The finite volume method is used to discretize the governing partial differential equations. The Quadratic Upstream Interpolation for Convective Kinematics (QUICK) scheme is used to treat convection transport and diffusion. Polyhedral volume elements with a hexahedral core were employed in the simulation (**Figure 4b, Figure 4c**). In order to capture the high flow gradient near the vessel wall, a finer boundary layer mesh near the solid was used (Figure 4d). The CFD package Ansys Fluent was used for solving the flow and pressure fields. More details about the employed numerical methods can be found in the ANSYS Fluent Theory Guide (29).

To confirm the accuracy of the present models, we selected available laser-Doppler measurements for steady-state fluid flows for Newtonian fluids (14). The simulated Reynolds number (Re), based on the average inlet velocity (u_{ave} = 0.0694 m/s), the Newtonian viscosity (1410 kg/m^3), the dynamic viscosity (0.0029 Pa s), and the inlet diameter of common carotid artery (0.008 m), is 270. For the Newtonian fluid, a model was created with a parabolic axial velocity profile directly specified for the fully developed laminar flow, $u = 2u_{ave}[1 - (2r/D_{CCA})2]$, where r is the radial coordinate defined at the inlet of the common carotid artery.

Grid independence tests were performed using several grid densities and the axial fluid velocity profiles at five selected locations in the sinus of internal carotid artery (ICA) were used as the criteria for grid independence solution. The selected locations are noted as: I_{00}, I_{05}, I_{10}, I_{15} and I_{20} (**Figure 4e**). The normalized axial fluid velocity at the center of the selected lines ($\bar{u}_c = u_c/u_{ave}$), selected as a parameter sensitive to local flow features within the carotid sinus, exhibited variation across differing mesh sizes for all lines (**Figure 4f, Table 4**). Among the evaluated locations, planes I_{05} and I_{10} exhibited the greatest sensitivity to mesh refinement, with a grid with 783,738 elements and 2,009,088 nodes finally selected for the numerical simulation of blood flow in the carotid bifurcation (**Figure 4g**). Using a denser grid with 1,552,236 elements and 4,416,721 nodes resulted in less than 0.5 percent difference in the predicted axial velocity component at the selected locations. The corresponding mesh size is 0.4 mm.

PINN Construction

We selected neural networks (NNs) for their ability to approximate high-dimensional and nonlinear relationships (22). In this study, we employed four fully connected NNs to independently approximate the three components of the velocity field and pressure as functions of spatial coordinates. Each network received the spatial coordinates and sensor domain as inputs and performed an affine transformation followed by a nonlinear activation function, such as tanh or ReLU in order to represent three-dimensional hemodynamics.

The network uses a squared error loss function to guide training, while employing a combination of sigmoid and linear activation functions to approximate velocity and pressure fields. For optimization, an Adam optimizer was used with an initial learning rate of 5×10^{-4}; this value avoids oscillations associated with higher learning rates while maintaining reasonable training speed. A decay rate of 0.5 was applied every 800 epochs, which we found by trial-and-error to allow coarse exploration of parameter space early in training

https://doi.org/10.59720/25-261

followed by more precise convergence in later epochs.

Whereas traditional numerical methods like the finite element method (FEM) require expensive data-assimilation to handle sparse or noisy measurements, PINNs more readily assimilate data across different fidelities and modalities, integrating physical laws into their loss functions (30). This makes PINNs particularly suited for applications where boundary conditions or material parameters are incompletely known, as is often the case in biomedical and engineering problems. The physics-based loss, L_{phys}, is defined as:

$$L_{phys} = \|\rho \boldsymbol{u} \cdot \nabla \boldsymbol{u} + \nabla p - \mu \nabla^2 \boldsymbol{u}\|_{\Omega} + \|\nabla \cdot \boldsymbol{u}\|_{\Omega}$$

(Equation. 4)

where ρ is density, μ is viscosity, and Ω represents the computational domain (22). This formulation ensures that the PINN predictions adhere to the governing physical laws, even when training data is sparse or noisy. Particularly well-suited for solving forward and inverse tasks, PINNs eliminate the need for elaborate mesh generation and can handle complex geometries and boundary conditions with ease (23).

We adapted and modified the PINN used in this study from the implementation by Arzani et al. (25). Sample density was controlled by varying N_{sample} values, where sensors were placed at intervals of every N_{sample} points within the sensor domain. Sensor location was adjusted by constructing sensor domains as surfaces at specified radial positions within the artery using Onshape (31). For the patient-specific model, the irregular vessel geometry prevented consistent radial surface definition; therefore, the entire geometry was used as the sensor domain.

Statistical Analysis

To evaluate the accuracy of the CFD ground truth, we computed the normalized root-mean square error (nRMSE) between CFD-predicted axial velocity profiles (u_{CFD}) and the corresponding experimental measurements (u_{exp}) at lines l_{00}-l_{20}.The nRMSE was calculated by normalizing the root-mean square error by the range of the experimental data:

$$RMSE = \frac{1}{u_{exp,max} - u_{exp,min}} \sqrt{\frac{\sum_{i=1}^{N} (u_{CFD} - u_{exp})^2}{N}}$$

(Equation. 5)

To assess the accuracy of PINN-predicted velocity profiles, we compared model outputs to ground-truth data generated from CFD simulations. The velocity errors were calculated across the sensor domain (either sub-boundary or near-wall), and statistical significance was evaluated using one-sample t-tests against the CFD baseline (32). To account for multiple comparisons across sensor density and placement conditions, the Bonferroni correction was applied (33). In this study, nine comparisons were conducted, resulting in an adjusted significance threshold at $\alpha = 0.0056$. Additionally, Cohen's d was computed to evaluate the effect size of deviations from ground-truth, with $|d| < 0.5$ classified as small, $0.5 \leq |d| < 0.8$ as medium, and $|d| \geq 0.8$ as large. All statistical analyses were performed in Python 3.11 using the SciPy and NumPy packages (34, 35).

ACKNOWLEDGMENTS

Valuable discussions and help from Professor Amir Arzani, Department of Mechanical Engineering, University of Utah, were gratefully appreciated.

REFERENCES

1. Libby, Peter. "The Changing Landscape of Atherosclerosis." *Nature*, vol. 592, no. 7855, 2021, pp. 524-33, https://doi.org/10.1038/s41586-021-03392-8.
2. Ihle-Hansen, Håkon, *et al*. "Carotid Plaque Score for Stroke and Cardiovascular Risk Prediction in a Middle-Aged Cohort from the General Population." *Journal of the American Heart Association*, vol. 12, no. 17, 5 Sept. 2023, https://doi.org/10.1161/jaha.123.030739.
3. Poredos, Pavel, *et al*. "Inflammation of Carotid Plaques and Risk of Cerebrovascular Events." *Annals of Translational Medicine*, vol. 8, no. 19, Oct. 2020, p. 1281, https://doi.org/10.21037/atm-2020-cass-15.
4. Caro, C.G. "Discovery of the Role of Wall Shear in Atherosclerosis." *Arteriosclerosis, Thrombosis, and Vascular Biology*, vol. 29, no. 2, 2009, pp. 158-61, https://doi.org/10.1161/atvbaha.108.166736.
5. Abhilash, H.N, *et al*. "Effect of Vascular Geometry on Haemodynamic Changes in a Carotid Artery Bifurcation Using Numerical Simulation." *Clinical Neurology and Neurosurgery*, vol. 237, 2024, p. 1081, https://doi.org/10.1016/j.clineuro.2024.108153.
6. Li, Cong-Hui, *et al*. "Hemodynamic Factors Affecting Carotid Sinus Atherosclerotic Stenosis." *World Neurosurgery*, vol. 121, 2019, pp. e262-e276, https://doi.org/10.1016/j.wneu.2018.09.091.
7. Marshall, Ian, *et al*. "MRI and CFD Studies of Pulsatile Flow in Healthy and Stenosed Carotid Bifurcation Models." *Journal of Biomechanics*, vol. 37, no. 5, 2004, pp. 679-87, https://doi.org/10.1016/j.jbiomech.2003.09.032.
8. Chaichana, Thanapong, *et al*. "Computation of Hemodynamics in the Left Coronary Artery with Variable Angulations." *Journal of Biomechanics*, vol. 44, no. 10, 2011, pp. 1869-78, https://doi.org/10.1016/j.jbiomech.2011.04.033.
9. Fisher, M., and S. Fieman. "Geometric Factors of the Bifurcation in Carotid Atherogenesis." *Stroke*, vol. 21, no. 2, 1990, pp. 267-71, https://doi.org/10.1161/01.str.21.2.267.
10. Chiastra, Claudio, *et al*. "Healthy and Diseased Coronary Bifurcation Geometries Influence Near-wall and Intravascular Flow: A Computational Exploration of the Hemodynamic Risk." *Journal of Biomechanics*, vol. 58, 2017, pp. 79-88, https://doi.org/10.1016/j.jbiomech.2017.04.016.
11. Shakya, Kshitij, and Shubhajit Roy Chowdhury. "A Fluid-structure Interaction Study to Analyze the Severity of Carotid Artery Stenosis at Different Locations and Its Effect on Various Hemodynamic Biomarkers." *European Journal of Mechanics - B/Fluids*, vol. 106, 2024, pp. 227-37, https://doi.org/10.1016/j.euromechflu.2024.04.006.
12. Antiga, Luca, and David A. Steinman. "Rethinking Turbulence in Blood." *Biorheology*, vol. 46, no. 2, 2009, pp. 77-81, https://doi.org/10.3233/bir-2009-0538.

13. Arzani, Amirhossein, *et al*. "Uncovering Near-wall Blood Flow from Sparse Data with Physics-informed Neural Networks." *ArXiv*, 2021, https://doi.org/10.48550/ARXIV.2104.08249.
14. Gijsen, F.j.h, *et al*. "The Influence of the Non-Newtonian Properties of Blood on the Flow in Large Arteries: Steady Flow in a Carotid Bifurcation Model." *Journal of Biomechanics*, vol. 32, no. 6, 1999, pp. 601-08, https://doi.org/10.1016/s0021-9290(99)00015-9.
15. Perktold, K., *et al*. "Pulsatile Non-newtonian Flow Characteristics in a Three-dimensional Human Carotid Bifurcation Model." *Journal of Biomechanical Engineering*, vol. 113, no. 4, 1991, pp. 464-75, https://doi.org/10.1115/1.2895428.
16. Steinman, David A., *et al*. "Reconstruction of Carotid Bifurcation Hemodynamics and Wall Thickness Using Computational Fluid Dynamics and MRI." *Magnetic Resonance in Medicine*, vol. 47, no. 1, 2001, pp. 149-59, https://doi.org/10.1002/mrm.10025.
17. Jung, Jonghwun, *et al*. "Multiphase Hemodynamic Simulation of Pulsatile Flow in a Coronary Artery." *Journal of Biomechanics*, vol. 39, no. 11, 2006, pp. 2064-73, https://doi.org/10.1016/j.jbiomech.2005.06.023.
18. Taylor, Charles A., and David A. Steinman. "Image-Based Modeling of Blood Flow and Vessel Wall Dynamics: Applications, Methods and Future Directions." *Annals of Biomedical Engineering*, vol. 38, no. 3, 2010, pp. 1188-203, https://doi.org/10.1007/s10439-010-9901-0.
19. Frydrychowicz, Alex, *et al*. "Four-dimensional Phase Contrast Magnetic Resonance Angiography: Potential Clinical Applications." *European Journal of Radiology*, vol. 80, no. 1, 2011, pp. 24-35, https://doi.org/10.1016/j.ejrad.2011.01.094.
20. Gharib, Ahmed M., *et al*. "Noninvasive Coronary Imaging for Atherosclerosis in Human Immunodeficiency Virus Infection." *Current Problems in Diagnostic Radiology*, vol. 40, no. 6, Nov. 2011, pp. 262-67, https://doi.org/10.1067/j.cpradiol.2011.06.001.
21. Nakano, Atushi, *et al*. "Velocity Profiles of Pulsatile Blood Flow in Arterioles with Bifurcation and Confluence in Rat Mesnetery Measured by Particle Image Velocimetry." *JSME International Journal Series C*, vol. 48, no. 4, 2005, pp. 444-52, https://doi.org/10.1299/jsmec.48.444.
22. Raissi, M., *et al*. "Physics-informed Neural Networks: A Deep Learning Framework for Solving Forward and Inverse Problems Involving Nonlinear Partial Differential Equations." *Journal of Computational Physics*, vol. 378, 2019, pp. 686-707, https://doi.org/10.1016/j.jcp.2018.10.045.
23. Karniadakis, George Em, *et al*. "Physics-informed Machine Learning." *Nature Reviews Physics*, vol. 3, no. 6, 2021, pp. 422-40, https://doi.org/10.1038/s42254-021-00314-5.
24. Eulzer, P., *et al*. "A Fully Integrated Pipeline for Visual Carotid Morphology Analysis." *Computer Graphics Forum*, vol. 42, no. 3, June 2023, pp. 25-37, https://doi.org/10.1111/cgf.14808.
25. Arzani, Amirhossein. "PINN-wss." GitHub. https://github.com/amir-cardiolab/PINN-wss. Accessed 13 August 2025.
26. Fernández-Friera, Leticia, *et al*. "Prevalence, Vascular Distribution, and Multiterritorial Extent of Subclinical Atherosclerosis in a Middle-Aged Cohort," vol. 131, no. 24, 16 June 2015, pp. 2104-13, https://doi.org/10.1161/circulationaha.114.014310.
27. Bharadvaj, B.K, *et al*. "Steady Flow in a Model of the Human Carotid Bifurcation. Part I: flow Visualization." *Journal of Biomechanics*, vol. 15, no. 5, 1982, pp. 349-62, https://doi.org/10.1016/0021-9290(82)90057-4.
28. Patankar, Suhas. Numerical Heat Transfer and Fluid Flow. CRC Press, 2018.
29. ANSYS. "Fluent Theory Guide." Ansys Fluent 2024 R2. https://ansyshelp.ansys.com/public/account/secured?returnurl=/Views/Secured/corp/v242/en/flu_th/flu_th.html. Accessed 13 August 2025.
30. He, QiZhi, *et al*. "Physics-informed Neural Networks for Multiphysics Data Assimilation with Application to Subsurface Transport." *Advances in Water Resources*, vol. 141, July 2020, p. 103610, https://doi.org/10.1016/j.advwatres.2020.103610.
31. Onshape. Onshape, 2026, www.onshape.com/en/spotlight/3d-cad-software. Accessed 8 April 2026.
32. Ross, Amanda, and Victor L. Willson. "One-Sample T-Test." *Basic and Advanced Statistical Tests*, SensePublishers, 2017, https://doi.org/10.1007/978-94-6351-086-8_2.
33. Armstrong, Richard A. "When to Use the Bonferroni Correction." *Ophthalmic and Physiological Optics*, vol. 34, no. 5, 2 Apr. 2014, pp. 502-08, https://doi.org/10.1111/opo.12131
34. The SciPy Community. "SciPy." SciPy. https://docs.scipy.org/doc/scipy/. Accessed 13 August 2025.
35. NumPy Developers. "NumPy." NumPy. https://numpy.org/doc/stable/. Accessed 13 August 2025.

Received: July 13, 2025
Accepted: April 14, 2026
Published: September 04, 2026

Appendix A.

Source code of the Physics-Informed Neural Network (PINN) used to predict hemodynamic flow fields in the idealized carotid bifurcation. Adapted and modified from Arzani et al. (24).

```
import torch
import numpy as np
from matplotlib import pyplot as plt
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import pdb
import csv
from torch.utils.data import DataLoader, TensorDataset,RandomSampler
from math import exp, sqrt,pi
import time
import vtk
from vtk.util import numpy_support as VN
import sys

def
geo_train(device,x_in,y_in,z_in,xb,yb,zb,ub,vb,wb,xd,yd,zd,ud,vd,wd,batchsize,lea
rning_rate,epochs,path,Flag_batch,Diff,rho,Flag_BC_exact,Lambda_BC,nPt,T,xb_inlet
,yb_inlet,zb_inlet,ub_inlet,vb_inlet,wb_inlet):
    if (Flag_batch):
     x = torch.Tensor(x_in).to(device)
     y = torch.Tensor(y_in).to(device)
     z = torch.Tensor(z_in).to(device)
     xb = torch.Tensor(xb).to(device)
     yb = torch.Tensor(yb).to(device)
     zb = torch.Tensor(zb).to(device)
     ub = torch.Tensor(ub).to(device)
     vb = torch.Tensor(vb).to(device)
     wb = torch.Tensor(wb).to(device)
     xd = torch.Tensor(xd).to(device)
     yd = torch.Tensor(yd).to(device)
     zd = torch.Tensor(zd).to(device)
     ud = torch.Tensor(ud).to(device)
     vd = torch.Tensor(vd).to(device)
     wd = torch.Tensor(wd).to(device)
     xb_inlet = torch.Tensor(xb_inlet).to(device)
     yb_inlet = torch.Tensor(yb_inlet).to(device)
     zb_inlet = torch.Tensor(zb_inlet).to(device)
     ub_inlet = torch.Tensor(ub_inlet).to(device)
```

```
     vb_inlet = torch.Tensor(vb_inlet).to(device)
     wb_inlet = torch.Tensor(wb_inlet).to(device)
     if(1): #Cuda slower in double?
         x = x.type(torch.cuda.FloatTensor)
         y = y.type(torch.cuda.FloatTensor)
         xb = xb.type(torch.cuda.FloatTensor)
         yb = yb.type(torch.cuda.FloatTensor)
         ub = ub.type(torch.cuda.FloatTensor)
         vb = vb.type(torch.cuda.FloatTensor)
         xb_inlet = xb_inlet.type(torch.cuda.FloatTensor)
         yb_inlet = yb_inlet.type(torch.cuda.FloatTensor)
         ub_inlet = ub_inlet.type(torch.cuda.FloatTensor)
         vb_inlet = vb_inlet.type(torch.cuda.FloatTensor)
         xd = xd.type(torch.cuda.FloatTensor)
         yd = yd.type(torch.cuda.FloatTensor)
         ud = ud.type(torch.cuda.FloatTensor)
         vd = vd.type(torch.cuda.FloatTensor)

     dataset = TensorDataset(x,y,z)

     dataloader = DataLoader(dataset,
batch_size=batchsize,shuffle=True,num_workers = 0, drop_last = True )
    else:
     x = torch.Tensor(x_in).to(device)
     y = torch.Tensor(y_in).to(device)
    #y_test  = torch.Tensor(y_test).to(device)
    h_n = 200 #128 #Width for u,v,p
    input_n = 3 # this is what our answer is a function of (x,y,z)
    class Swish(nn.Module):
        def __init__(self, inplace=True):
            super(Swish, self).__init__()
            self.inplace = inplace

        def forward(self, x):
            if self.inplace:
                x.mul_(torch.sigmoid(x))
                return x
            else:
                return x * torch.sigmoid(x)
    class MySquared(nn.Module):
        def __init__(self, inplace=True):
            super(MySquared, self).__init__()
            self.inplace = inplace

        def forward(self, x):
```

```
        return torch.square(x)


class Net2_u(nn.Module):

    #The __init__ function stack the layers of the
    #network Sequentially
    def __init__(self):
        super(Net2_u, self).__init__()
        self.main = nn.Sequential(
            nn.Linear(input_n,h_n),
            Swish(),
            nn.Linear(h_n,h_n),
            Swish(),
            nn.Linear(h_n,h_n),
            Swish(),
            nn.Linear(h_n,h_n),

            Swish(),
            nn.Linear(h_n,h_n),

            Swish(),
            nn.Linear(h_n,h_n),

            Swish(),
            nn.Linear(h_n,h_n),

            Swish(),
            nn.Linear(h_n,h_n),

            Swish(),
           nn.Linear(h_n,h_n),

            Swish(),

            nn.Linear(h_n,1),
        )
    #This function defines the forward rule of
    #output respect to input.
    #def forward(self,x):
    def forward(self,x):
        output = self.main(x)
        if (Flag_BC_exact):
```

```
                output = output*(x- xStart) *(y- yStart) * (y- yEnd ) + U_BC_in +
(y- yStart) * (y- yEnd )  #modify output to satisfy BC automatically #PINN-
transfer-learning-BC-20
            return output

    class Net2_v(nn.Module):

        #The __init__ function stack the layers of the
        #network Sequentially
        def __init__(self):
            super(Net2_v, self).__init__()
            self.main = nn.Sequential(
                nn.Linear(input_n,h_n),

                Swish(),
                nn.Linear(h_n,h_n),

                Swish(),
                nn.Linear(h_n,h_n),

                Swish(),
                nn.Linear(h_n,h_n),

                Swish(),
                nn.Linear(h_n,h_n),

                Swish(),
                nn.Linear(h_n,h_n),

                Swish(),
                nn.Linear(h_n,h_n),

                Swish(),
                nn.Linear(h_n,h_n),

                Swish(),
                nn.Linear(h_n,h_n),

                Swish(),
                nn.Linear(h_n,h_n),

                Swish(),

                nn.Linear(h_n,1),
            )
        #This function defines the forward rule of
        #output respect to input.
        #def forward(self,x):
```

```
        def forward(self,x):
            output = self.main(x)
            if (Flag_BC_exact):
                output = output*(x- xStart) *(x- xEnd )*(y- yStart) *(y- yEnd ) +
(-0.9*x + 1.) #modify output to satisfy BC automatically #PINN-transfer-learning-
BC-20
            return output

    class Net2_w(nn.Module):

        #The __init__ function stack the layers of the
        #network Sequentially
        def __init__(self):
            super(Net2_w, self).__init__()
            self.main = nn.Sequential(
                nn.Linear(input_n,h_n),

                Swish(),
                nn.Linear(h_n,h_n),

                Swish(),
                nn.Linear(h_n,h_n),

                Swish(),
                nn.Linear(h_n,h_n),

                Swish(),
                nn.Linear(h_n,h_n),

                Swish(),
                nn.Linear(h_n,h_n),

                Swish(),
                nn.Linear(h_n,h_n),

                Swish(),
                nn.Linear(h_n,h_n),

                Swish(),
                nn.Linear(h_n,h_n),

                Swish(),

                nn.Linear(h_n,1),
            )
```

```
        #This function defines the forward rule of
        #output respect to input.
        #def forward(self,x):
        def forward(self,x):
            output = self.main(x)
            if (Flag_BC_exact):
                output = output*(x- xStart) *(x- xEnd )*(y- yStart) *(y- yEnd ) +
(-0.9*x + 1.) #modify output to satisfy BC automatically #PINN-transfer-learning-
BC-20
            return output

    class Net2_p(nn.Module):

        #The __init__ function stack the layers of the
        #network Sequentially
        def __init__(self):
            super(Net2_p, self).__init__()
            self.main = nn.Sequential(
                nn.Linear(input_n,h_n),

                Swish(),
                nn.Linear(h_n,h_n),

                Swish(),
                nn.Linear(h_n,h_n),

                Swish(),
                nn.Linear(h_n,h_n),

                Swish(),
                nn.Linear(h_n,h_n),

                Swish(),
                nn.Linear(h_n,h_n),

                Swish(),
                nn.Linear(h_n,h_n),

                Swish(),
                nn.Linear(h_n,h_n),

                Swish(),
                nn.Linear(h_n,h_n),

                Swish(),
```

```
                nn.Linear(h_n,h_n),

                Swish(),

                nn.Linear(h_n,1),
            )
        #This function defines the forward rule of
        #output respect to input.
        def forward(self,x):
            output = self.main(x)
            if (Flag_BC_exact):
                output = output*(x- xStart) *(x- xEnd )*(y- yStart) *(y- yEnd ) +
(-0.9*x + 1.) #modify output to satisfy BC automatically #PINN-transfer-learning-
BC-20
            return  output

    ######################################################################
    net2_u = Net2_u().to(device)
    net2_v = Net2_v().to(device)
    net2_w = Net2_w().to(device)
    net2_p = Net2_p().to(device)


    def init_normal(m):
        if type(m) == nn.Linear:
            nn.init.kaiming_normal_(m.weight)

    # use the modules apply function to recursively apply the initialization
    net2_u.apply(init_normal)
    net2_v.apply(init_normal)
    net2_w.apply(init_normal)
    net2_p.apply(init_normal)


    ###########################################################################

    optimizer_u = optim.Adam(net2_u.parameters(), lr=learning_rate, betas =
(0.9,0.99),eps = 10**-15)
    optimizer_v = optim.Adam(net2_v.parameters(), lr=learning_rate, betas =
(0.9,0.99),eps = 10**-15)
    optimizer_w = optim.Adam(net2_w.parameters(), lr=learning_rate, betas =
(0.9,0.99),eps = 10**-15)
    optimizer_p = optim.Adam(net2_p.parameters(), lr=learning_rate, betas =
(0.9,0.99),eps = 10**-15)
```

```
    def criterion(x,y,z):
        x.requires_grad = True
        y.requires_grad = True
        z.requires_grad = True

        net_in = torch.cat((x,y,z),1)
        u = net2_u(net_in)
        u = u.view(len(u),-1)
        v = net2_v(net_in)
        v = v.view(len(v),-1)
        w = net2_w(net_in)
        w = w.view(len(w),-1)
        P = net2_p(net_in)
        P = P.view(len(P),-1)



        u_x =
torch.autograd.grad(u,x,grad_outputs=torch.ones_like(x),create_graph =
True,only_inputs=True)[0]
        u_xx =
torch.autograd.grad(u_x,x,grad_outputs=torch.ones_like(x),create_graph =
True,only_inputs=True)[0]
        u_y =
torch.autograd.grad(u,y,grad_outputs=torch.ones_like(y),create_graph =
True,only_inputs=True)[0]
        u_yy =
torch.autograd.grad(u_y,y,grad_outputs=torch.ones_like(y),create_graph =
True,only_inputs=True)[0]
        u_z =
torch.autograd.grad(u,z,grad_outputs=torch.ones_like(z),create_graph =
True,only_inputs=True)[0]
        u_zz =
torch.autograd.grad(u_z,z,grad_outputs=torch.ones_like(z),create_graph =
True,only_inputs=True)[0]


        v_x =
torch.autograd.grad(v,x,grad_outputs=torch.ones_like(x),create_graph =
True,only_inputs=True)[0]
```

```
        v_xx =
torch.autograd.grad(v_x,x,grad_outputs=torch.ones_like(x),create_graph =
True,only_inputs=True)[0]
        v_y =
torch.autograd.grad(v,y,grad_outputs=torch.ones_like(y),create_graph =
True,only_inputs=True)[0]
        v_yy =
torch.autograd.grad(v_y,y,grad_outputs=torch.ones_like(y),create_graph =
True,only_inputs=True)[0]
        v_z =
torch.autograd.grad(v,z,grad_outputs=torch.ones_like(z),create_graph =
True,only_inputs=True)[0]
        v_zz =
torch.autograd.grad(v_z,z,grad_outputs=torch.ones_like(z),create_graph =
True,only_inputs=True)[0]


        w_x =
torch.autograd.grad(w,x,grad_outputs=torch.ones_like(x),create_graph =
True,only_inputs=True)[0]
        w_xx =
torch.autograd.grad(w_x,x,grad_outputs=torch.ones_like(x),create_graph =
True,only_inputs=True)[0]
        w_y =
torch.autograd.grad(w,y,grad_outputs=torch.ones_like(y),create_graph =
True,only_inputs=True)[0]
        w_yy =
torch.autograd.grad(w_y,y,grad_outputs=torch.ones_like(y),create_graph =
True,only_inputs=True)[0]
        w_z =
torch.autograd.grad(w,z,grad_outputs=torch.ones_like(z),create_graph =
True,only_inputs=True)[0]
        w_zz =
torch.autograd.grad(w_z,z,grad_outputs=torch.ones_like(z),create_graph =
True,only_inputs=True)[0]

        P_x =
torch.autograd.grad(P,x,grad_outputs=torch.ones_like(x),create_graph =
True,only_inputs=True)[0]
        P_y =
torch.autograd.grad(P,y,grad_outputs=torch.ones_like(y),create_graph =
True,only_inputs=True)[0]
        P_z =
torch.autograd.grad(P,z,grad_outputs=torch.ones_like(z),create_graph =
True,only_inputs=True)[0]
```

```
        XX_scale = U_scale * (X_scale**2)
        YY_scale = U_scale * (Y_scale**2)
        ZZ_scale = U_scale * (Z_scale**2)
        UU_scale  = U_scale **2

        loss_2 = u*u_x / X_scale + v*u_y / Y_scale + w*u_z / Z_scale  -
Diff*( u_xx/XX_scale  + u_yy /YY_scale + u_zz /ZZ_scale   )+ 1/rho* (P_x /
(X_scale*UU_scale)   )  #X-dir
        loss_1 = u*v_x / X_scale + v*v_y / Y_scale + w*v_z / Z_scale -
Diff*( v_xx/ XX_scale + v_yy / YY_scale + v_zz / ZZ_scale )+ 1/rho*(P_y /
(Y_scale*UU_scale)   ) #Y-dir
        loss_3 = (u_x / X_scale + v_y / Y_scale + w_z / Z_scale) #continuity

        loss_4 = u*w_x / X_scale + v*w_y / Y_scale + w*w_z / Z_scale -
Diff*( w_xx/ XX_scale + w_yy / YY_scale + w_zz / ZZ_scale )+ 1/rho*(P_z /
(Z_scale*UU_scale)   ) #Z-dir

        # MSE LOSS
        loss_f = nn.MSELoss()

        #Note our target is zero. It is residual so we use zeros_like
        loss =
loss_f(loss_1,torch.zeros_like(loss_1))+  loss_f(loss_2,torch.zeros_like(loss_2))
+  loss_f(loss_3,torch.zeros_like(loss_3))
+loss_f(loss_4,torch.zeros_like(loss_4))

        return loss

    def Loss_BC(xb,yb, zb, ub,vb, wb, xb_inlet, yb_inlet, ub_inlet, x, y,z):



        net_in1 = torch.cat((xb, yb,zb), 1)
        out1_u = net2_u(net_in1)
        out1_v = net2_v(net_in1)
        out1_w = net2_w(net_in1)

        out1_u = out1_u.view(len(out1_u), -1)
        out1_v = out1_v.view(len(out1_v), -1)
        out1_w = out1_w.view(len(out1_w), -1)
```

```
        loss_f = nn.MSELoss()
        loss_noslip = loss_f(out1_u, torch.zeros_like(out1_u)) + loss_f(out1_v,
torch.zeros_like(out1_v)) + loss_f(out1_w, torch.zeros_like(out1_w))

        return loss_noslip


    def Loss_data(xd,yd,zd,ud,vd,wd ):

        net_in1 = torch.cat((xd, yd,zd), 1)
        out1_u = net2_u(net_in1)
        out1_v = net2_v(net_in1)
        out1_w = net2_w(net_in1)

        out1_u = out1_u.view(len(out1_u), -1)
        out1_v = out1_v.view(len(out1_v), -1)
        out1_w = out1_w.view(len(out1_w), -1)



        loss_f = nn.MSELoss()
        loss_d = loss_f(out1_u, ud) + loss_f(out1_v, vd)  + loss_f(out1_w, wd)


        return loss_d

    # Main loop

    tic = time.time()


    if(Flag_pretrain):
        print('Reading (pretrain) functions first...')
        net2_u.load_state_dict(torch.load(path+"IA3D_data3velsmall_u_{}.pt".forma
t(N_sample)))
        net2_v.load_state_dict(torch.load(path+"IA3D_data3velsmall_v_{}.pt".forma
t(N_sample)))
        net2_w.load_state_dict(torch.load(path+"IA3D_data3velsmall_w_{}.pt".forma
t(N_sample)))
        net2_p.load_state_dict(torch.load(path+"IA3D_data3velsmall_p_{}.pt".forma
t(N_sample)))
```

```
    if (Flag_schedule):
        scheduler_u = torch.optim.lr_scheduler.StepLR(optimizer_u,
step_size=step_epoch, gamma=decay_rate)
        scheduler_v = torch.optim.lr_scheduler.StepLR(optimizer_v,
step_size=step_epoch, gamma=decay_rate)
        scheduler_w = torch.optim.lr_scheduler.StepLR(optimizer_w,
step_size=step_epoch, gamma=decay_rate)
        scheduler_p = torch.optim.lr_scheduler.StepLR(optimizer_p,
step_size=step_epoch, gamma=decay_rate)

    if(Flag_batch):# This one uses dataloader

            for epoch in range(epochs):
                loss_eqn_tot = 0.
                loss_bc_tot = 0.
                loss_data_tot = 0.
                n = 0
                for batch_idx, (x_in,y_in,z_in) in enumerate(dataloader):

                    net2_u.zero_grad()
                    net2_v.zero_grad()
                    net2_w.zero_grad()
                    net2_p.zero_grad()
                    loss_eqn = criterion(x_in,y_in,z_in)
                    loss_bc =
Loss_BC(xb,yb,zb,ub,vb,wb,xb_inlet,yb_inlet,ub_inlet,x,y,z)
                    loss_data = Loss_data(xd,yd,zd,ud,vd,wd)
                    loss = loss_eqn + Lambda_BC* loss_bc + Lambda_data*loss_data
                    loss.backward()
                    optimizer_u.step()
                    optimizer_v.step()
                    optimizer_w.step()
                    optimizer_p.step()
                    loss_eqn_tot += loss_eqn
                    loss_bc_tot += loss_bc
                    loss_data_tot  += loss_data
                    n += 1
                    if batch_idx % 40 ==0:
                        #loss_bc = Loss_BC(xb,yb,ub,vb) #causes out of memory
issue for large data in cuda
                        print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss eqn
{:.10f} Loss BC {:.8f} Loss data {:.8f}'.format(
                            epoch, batch_idx * len(x_in),
len(dataloader.dataset),
```

```
                            100. * batch_idx / len(dataloader), loss_eqn.item(),
loss_bc.item(),loss_data.item()))
                        #print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss eqn
{:.10f} '.format(
                        #   epoch, batch_idx * len(x_in),
len(dataloader.dataset),
                        #   100. * batch_idx / len(dataloader), loss.item()))
                if (Flag_schedule):
                        scheduler_u.step()
                        scheduler_v.step()
                        scheduler_w.step()
                        scheduler_p.step()
                loss_eqn_tot = loss_eqn_tot / n
                loss_bc_tot = loss_bc_tot / n
                loss_data_tot = loss_data_tot / n
                print('*****Total avg Loss : Loss eqn {:.10f} Loss BC {:.10f}
Loss data {:.10f} ****'.format(loss_eqn_tot, loss_bc_tot,loss_data_tot) )

                epoch_string = str(epoch)
                eqn_loss = '{:.10f}'.format(loss_eqn_tot)
                bc_loss = '{:.10f}'.format(loss_bc_tot)
                data_loss = '{:.10f}'.format(loss_data_tot)

                row = [epoch_string, eqn_loss, bc_loss, data_loss]
                filename = path+'.vtu/loss_{}.csv'.format(N_sample)
                with open(filename, 'a') as csvfile:
                    csvwriter = csv.writer(csvfile)
                    csvwriter.writerow(row)

                print('learning rate is ', optimizer_u.param_groups[0]['lr'],
optimizer_v.param_groups[0]['lr'])

                if( epoch%1000 ==0 and epoch > 3000 ): #This causes out of memory
in cuda in autodiff
                    torch.save(net2_p.state_dict(),path+".pt/"+"IA3D_data3velsmal
l_p_{}.pt".format(N_sample))
                    torch.save(net2_u.state_dict(),path+".pt/"+"IA3D_data3velsmal
l_u_{}.pt".format(N_sample))
                    torch.save(net2_v.state_dict(),path+".pt/"+"IA3D_data3velsmal
l_v_{}.pt".format(N_sample))
                    torch.save(net2_w.state_dict(),path+".pt/"+"IA3D_data3velsmal
l_w_{}.pt".format(N_sample))


    else:
```

```
        for epoch in range(epochs):
            #zero gradient
            #net1.zero_grad()
            ##Closure function for LBFGS loop:
            #def closure():
            net2.zero_grad()
            loss_eqn = criterion(x,y)
            loss_bc = Loss_BC(xb,yb,cb)
            if (Flag_BC_exact):
                loss = loss_eqn #+ loss_bc
            else:
                loss = loss_eqn + Lambda_BC * loss_bc
            loss.backward()
            #return loss
            #loss = closure()
            #optimizer2.step(closure)
            #optimizer3.step(closure)
            #optimizer4.step(closure)
            optimizer_u.step()
            optimizer_v.step()
            optimizer_p.step()
            if epoch % 10 ==0:
                print('Train Epoch: {} \tLoss: {:.10f} \t Loss BC {:.6f}'.format(
                    epoch, loss.item(),loss_bc.item()))

    toc = time.time()
    elapseTime = toc - tic
    print ("elapse time in parallel = ", elapseTime)
    ####################
    #plot
    if(1):#save network
        torch.save(net2_p.state_dict(),path+".pt/"+"IA3D_data3velsmall_p_{}.pt".f
ormat(N_sample))
        torch.save(net2_u.state_dict(),path+".pt/"+"IA3D_data3velsmall_u_{}.pt".f
ormat(N_sample))
        torch.save(net2_v.state_dict(),path+".pt/"+"IA3D_data3velsmall_v_{}.pt".f
ormat(N_sample))
        torch.save(net2_w.state_dict(),path+".pt/"+"IA3D_data3velsmall_w_{}.pt".f
ormat(N_sample))
        print ("Data saved!")

    net_in =
torch.cat((x.requires_grad_(),y.requires_grad_(),z.requires_grad_()),1)
    output_u = net2_u(net_in)  #evaluate model (runs out of memory for large GPU
problems!)
```

```
    output_v = net2_v(net_in)  #evaluate model
    output_w = net2_w(net_in)


    output_u = output_u.cpu().data.numpy() #need to convert to cpu before
converting to numpy
    output_v = output_v.cpu().data.numpy()
    output_w = output_w.cpu().data.numpy()

    fieldname = 'Vel_PINN'

    print ('Loading', mesh_file)
    reader = vtk.vtkUnstructuredGridReader()
    reader.SetFileName(mesh_file)
    reader.Update()
    data_iso = reader.GetOutput()
    n_points_iso = data_iso.GetNumberOfPoints()

    print ('number of nodes in new mesh', n_points_iso)

    # Initialize a numpy array with zeros for the velocity components
    Vel_interped  = np.zeros((n_points_iso, 3))
    for j in range(n_points_iso):
        Vel_interped[j,0] = output_u[j]
        Vel_interped[j,1] = output_v[j]
        Vel_interped[j,2] = output_w[j]

    #write output
    output_vtk = VN.numpy_to_vtk(Vel_interped)
    output_vtk.SetName(fieldname)
    data_iso.GetPointData().AddArray(output_vtk)

    output_filename2 = path+'.vtu/my_output_{}.vtu'.format(N_sample)
    myoutput = vtk.vtkXMLDataSetWriter()
    myoutput.SetInputData(data_iso)
    myoutput.SetFileName(output_filename2)
    myoutput.Write()




    return
```

```
############################################################
#Main code:
#device = torch.device("cpu")
device = torch.device("cuda")

path = "Results/"
N_sample = sys.argv[1] # sample every other N_sample pts based on command line
inputs

Flag_batch = True #False #USe batch or not  #With batch getting error...
Flag_BC_exact = False #If True enforces BC exactly HELPS ALOT!!! Not implemented
in 2D
Lambda_BC  = 20.  ## If not enforcing BC exacctly.

Lambda_data = 20.

fields = ['Epoch', 'Loss_eqn', 'Loss_BC', 'Loss_data']
filename = path+'.vtu/'+'loss_{}.csv'.format(N_sample)
with open(filename, 'w') as csvfile:
    csvwriter = csv.writer(csvfile)
    csvwriter.writerow(fields)


Directory = ""
mesh_file = Directory + "fluid_domain.vtk" #"IA3D_nearwall_correct.vtu"
outer_wall_location = Directory + "fluid_domain.vtk" #probe locations
bc_file_wall = Directory + "wall_hub.vtk" #wall boundaries

File_data = Directory + "rho1_0_0.vtu"
fieldname = 'velocity' #The velocity field name in the vtk file (see from
ParaView)

batchsize = 512
learning_rate = 1e-5

#epochs  = 5500 + 3000
epochs  = 2

Flag_pretrain = False # True #If true reads the nets from last run


Diff = 0.001
rho = 1.
T = 0.5 #total duraction
```

```
Flag_x_length = True #if True scales the eqn such that the length of the domain
is = X_scale
X_scale = 0.0184496 #The length of the  domain (need longer length for separation
region)
Y_scale = 0.0136581
Z_scale = 0.00900029
U_scale = 20.3021
U_BC_in = 0.5


Lambda_div = 1.  #penalty factor for continuity eqn (Makes it worse!?)
Lambda_v = 1.  #penalty factor for y-momentum equation

#https://stackoverflow.com/questions/60050586/pytorch-change-the-learning-rate-
based-on-number-of-epochs
Flag_schedule = True #If true change the learning rate
if (Flag_schedule):
    learning_rate = 5e-4  #starting learning rate
    step_epoch = 800
    decay_rate = 0.5


if (not Flag_x_length):
    X_scale = 1.
    Y_scale = 1.
    Z_scale = 1.


print ('Loading', mesh_file)
#reader = vtk.vtkXMLUnstructuredGridReader()
reader = vtk.vtkUnstructuredGridReader()
reader.SetFileName(mesh_file)
reader.Update()
data_vtk = reader.GetOutput()
n_points = data_vtk.GetNumberOfPoints()
print ('n_points of the mesh:' ,n_points)
x_vtk_mesh = np.zeros((n_points,1))
y_vtk_mesh = np.zeros((n_points,1))
z_vtk_mesh = np.zeros((n_points,1))
VTKpoints = vtk.vtkPoints()
for i in range(n_points):
    pt_iso  =  data_vtk.GetPoint(i)
    x_vtk_mesh[i] = pt_iso[0]
    y_vtk_mesh[i] = pt_iso[1]
    z_vtk_mesh[i] = pt_iso[2]
```

```
    VTKpoints.InsertPoint(i, pt_iso[0], pt_iso[1], pt_iso[2])

point_data = vtk.vtkUnstructuredGrid()
point_data.SetPoints(VTKpoints)

x  = np.reshape(x_vtk_mesh , (np.size(x_vtk_mesh [:]),1)) / X_scale
y  = np.reshape(y_vtk_mesh , (np.size(y_vtk_mesh [:]),1)) / Y_scale
z  = np.reshape(z_vtk_mesh , (np.size(z_vtk_mesh [:]),1)) / Z_scale



nPt = 130
xStart = 0.
xEnd = 1.
yStart = 0.
yEnd = 1.0
delta_circ = 0.2




t = np.linspace(0., T, nPt*nPt)
t=t.reshape(-1, 1)
print('shape of x',x.shape)
print('shape of y',y.shape)
print('shape of z',z.shape)
#print('shape of t',t.shape)



## Define boundary points
print ('Loading', outer_wall_location)
reader = vtk.vtkUnstructuredGridReader()
#reader = vtk.vtkPolyDataReader()
reader.SetFileName(outer_wall_location)
reader.Update()
data_vtk = reader.GetOutput()
n_points = data_vtk.GetNumberOfPoints()
print ('n_points of at outer wall' ,n_points)
x_vtk_mesh = np.zeros((n_points,1))
y_vtk_mesh = np.zeros((n_points,1))
z_vtk_mesh = np.zeros((n_points,1))
VTKpoints = vtk.vtkPoints()
for i in range(n_points):
    pt_iso  =  data_vtk.GetPoint(i)
```

```
    x_vtk_mesh[i] = pt_iso[0]
    y_vtk_mesh[i] = pt_iso[1]
    z_vtk_mesh[i] = pt_iso[2]
    VTKpoints.InsertPoint(i, pt_iso[0], pt_iso[1], pt_iso[2])
point_data = vtk.vtkUnstructuredGrid()
point_data.SetPoints(VTKpoints)
xb_in  = np.reshape(x_vtk_mesh , (np.size(x_vtk_mesh[:]),1)) / X_scale
yb_in  = np.reshape(y_vtk_mesh , (np.size(y_vtk_mesh[:]),1)) / Y_scale
zb_in  = np.reshape(z_vtk_mesh , (np.size(z_vtk_mesh[:]),1)) / Z_scale

print ('Loading', bc_file_wall)
reader = vtk.vtkUnstructuredGridReader()
#reader = vtk.vtkPolyDataReader()
reader.SetFileName(bc_file_wall)
reader.Update()
data_vtk = reader.GetOutput()
n_pointsw = data_vtk.GetNumberOfPoints()
print ('n_points of at wall' ,n_pointsw)
x_vtk_mesh = np.zeros((n_pointsw,1))
y_vtk_mesh = np.zeros((n_pointsw,1))
z_vtk_mesh = np.zeros((n_pointsw,1))
VTKpoints = vtk.vtkPoints()
for i in range(n_pointsw):
    pt_iso  =  data_vtk.GetPoint(i)
    x_vtk_mesh[i] = pt_iso[0]
    y_vtk_mesh[i] = pt_iso[1]
    z_vtk_mesh[i] = pt_iso[2]
    VTKpoints.InsertPoint(i, pt_iso[0], pt_iso[1], pt_iso[2])
point_data = vtk.vtkUnstructuredGrid()
point_data.SetPoints(VTKpoints)
xb_wall  = np.reshape(x_vtk_mesh , (np.size(x_vtk_mesh [:]),1)) / X_scale
yb_wall  = np.reshape(y_vtk_mesh , (np.size(y_vtk_mesh [:]),1)) / Y_scale
zb_wall  = np.reshape(z_vtk_mesh , (np.size(z_vtk_mesh [:]),1)) / Z_scale




#u_in_BC = np.linspace(U_BC_in, U_BC_in, n_points) #constant uniform BC
u_in_BC = (yb_in[:]) * ( 0.3 - yb_in[:] )  / 0.0225 * U_BC_in #parabolic


v_in_BC = np.linspace(0., 0., n_points)
w_in_BC = np.linspace(0., 0., n_points)
u_wall_BC = np.linspace(0., 0., n_pointsw)
v_wall_BC = np.linspace(0., 0., n_pointsw)
```

```
w_wall_BC = np.linspace(0., 0., n_pointsw)

xb = xb_wall
yb = yb_wall
zb = zb_wall

ub = u_wall_BC
vb = v_wall_BC
wb = w_wall_BC

xb_inlet = xb_in
yb_inlet = yb_in
zb_inlet = zb_in


ub_inlet = u_in_BC
vb_inlet = v_in_BC
wb_inlet = w_in_BC

#tb= tb.reshape(-1, 1) #need to reshape to get 2D array
xb= xb.reshape(-1, 1) #need to reshape to get 2D array
yb= yb.reshape(-1, 1) #need to reshape to get 2D array
zb= zb.reshape(-1, 1) #need to reshape to get 2D array
ub= ub.reshape(-1, 1) #need to reshape to get 2D array
vb= vb.reshape(-1, 1) #need to reshape to get 2D array
wb= wb.reshape(-1, 1) #need to reshape to get 2D array
xb_inlet= xb_inlet.reshape(-1, 1) #need to reshape to get 2D array
yb_inlet= yb_inlet.reshape(-1, 1) #need to reshape to get 2D array
zb_inlet= zb_inlet.reshape(-1, 1) #need to reshape to get 2D array
ub_inlet= ub_inlet.reshape(-1, 1) #need to reshape to get 2D array
vb_inlet= vb_inlet.reshape(-1, 1) #need to reshape to get 2D array
wb_inlet= wb_inlet.reshape(-1, 1) #need to reshape to get 2D array

print('shape of xb',xb.shape)
print('shape of yb',yb.shape)
print('shape of ub',ub.shape)



##### Read data here#########################

#!!specify pts location here:
#x_data = [1., 1.2, 1.22, 1.31, 1.39 ]
#y_data =[0.15, 0.07, 0.22, 0.036, 0.26 ]
#z_data  = [0.,0.,0.,0.,0. ]
```

```
#x_data = np.asarray(x_data)  #convert to numpy
#y_data = np.asarray(y_data) #convert to numpy

print("Sampling every {} points".format(N_sample))

print ('Loading', outer_wall_location)
reader = vtk.vtkUnstructuredGridReader()
#reader = vtk.vtkPolyDataReader()
reader.SetFileName(outer_wall_location)
reader.Update()
data_vtk = reader.GetOutput()
n_points = data_vtk.GetNumberOfPoints()
print ('n_points of at outer wall (locations randomly data probed
from)' ,n_points)
x_vtk_mesh = np.zeros((n_points,1))
y_vtk_mesh = np.zeros((n_points,1))
z_vtk_mesh = np.zeros((n_points,1))
VTKpoints = vtk.vtkPoints()
N_pts_data = 0
for i in range(n_points):
    if i%int(N_sample) ==0:
        pt_iso  =  data_vtk.GetPoint(i)
        x_vtk_mesh[N_pts_data] = pt_iso[0]
        y_vtk_mesh[N_pts_data] = pt_iso[1]
        z_vtk_mesh[N_pts_data] = pt_iso[2]
        N_pts_data +=1

print ('n_points sampled' ,N_pts_data)
x_data = np.zeros((N_pts_data,1))
y_data = np.zeros((N_pts_data,1))
z_data = np.zeros((N_pts_data,1))

x_data[:,0] = x_vtk_mesh[0:N_pts_data,0]
y_data[:,0] = y_vtk_mesh[0:N_pts_data,0]
z_data[:,0] = z_vtk_mesh[0:N_pts_data,0]


print ('Loading', File_data)
reader = vtk.vtkXMLUnstructuredGridReader()
reader.SetFileName(File_data)
reader.Update()
data_vtk = reader.GetOutput()
n_points = data_vtk.GetNumberOfPoints()
print ('n_points of the data file read:' ,n_points)
```

```
VTKpoints = vtk.vtkPoints()
for i in range(len(x_data)):
    VTKpoints.InsertPoint(i, x_data[i] , y_data[i]  , z_data[i])

point_data = vtk.vtkUnstructuredGrid()
point_data.SetPoints(VTKpoints)

probe = vtk.vtkProbeFilter()
probe.SetInputData(point_data)
probe.SetSourceData(data_vtk)
probe.Update()
array = probe.GetOutput().GetPointData().GetArray(fieldname)
data_vel = VN.vtk_to_numpy(array)



data_vel_u = data_vel[:,0] / U_scale
data_vel_v = data_vel[:,1] / U_scale
data_vel_w = data_vel[:,2] / U_scale

x_data = x_data / X_scale #convert to normalized coordinates for PINN
y_data = y_data / Y_scale
z_data = z_data / Z_scale

print('Using input data pts: pts: ',x_data, y_data, z_data)
print('Using input data pts: vel u: ',data_vel_u)
print('Using input data pts: vel v: ',data_vel_v)
print('Using input data pts: vel w: ',data_vel_w)

for i in range(len(x_data)):
    print('u: ',data_vel_u[i] )
    print('w: ',data_vel_w[i] )


xd= x_data.reshape(-1, 1) #need to reshape to get 2D array
yd= y_data.reshape(-1, 1) #need to reshape to get 2D array
zd= z_data.reshape(-1, 1) #need to reshape to get 2D array
ud= data_vel_u.reshape(-1, 1) #need to reshape to get 2D array
vd= data_vel_v.reshape(-1, 1) #need to reshape to get 2D array
wd= data_vel_w.reshape(-1, 1) #need to reshape to get 2D array


#path = pre+"aneurysmsigma01scalepara_100pt-tmp_"+str(ii)
```

```
geo_train(device,x,y,z,xb,yb,zb,ub,vb,wb,xd,yd,zd,ud,vd,wd,batchsize,learning_rat
e,epochs,path,Flag_batch,Diff,rho,Flag_BC_exact,Lambda_BC,nPt,T,xb_inlet,yb_inlet
, zb_inlet, ub_inlet,vb_inlet, wb_inlet )
#tic = time.time()

#elapseTime = toc - tic
#print ("elapse time in serial = ", elapseTime)
```