ABSTRACT

Title of Dissertation: Automated Simulation and the Discovery of Mechanical Devices

Kevin Chiu
Doctor of Philosophy, 2024

Dissertation Directed by: Professor Mark Fuge
Department of Mechanical Engineering

Automatically designing or finding novel devices that accomplish new or existing functions remains one of the greatest unsolved problems in Design Automation. In part, this is due to 1) the interplay of physical form and usage, 2) the emergence of complex behaviors from combinations of simple geometries, and 3) the sparsity and instability of "interesting" physical phenomena with small changes in the design space, which have historically stymied past efforts, since most approaches required 1) human intuition and creativity, 2) infeasibly large amounts of computational power, or 3) *a priori* targeted desired behavior. In contrast, this dissertation takes a data-driven approach to addressing the general question "What device functionality emerges organically from knowledge of various physical laws?" To make this high-level question more precise, this dissertation tackles three interrelated sub-questions that address challenges that arise when attempting to deploy data-driven methods on function discovery tasks.

First, to generate diverse and high-quality datasets from which an algorithm might find novel behavior, this dissertation asks, "How do we enumerate possible boundary conditions for a

given physical law that can lead to well-defined solutions to a given partial differential equation?" Chapter 3 proposes a type-based indexing scheme and two properties of that scheme that can generate valid Finite Element Method (FEM) formulations, resulting in a three-fold increase in the number of simulations we generated from our limited set of boundary conditions. Chapter 4 proposes a regression formulation for predicting physical realizability in Stokes flow simulations as estimated with the magnitude of the pressure field. Second, this dissertation asks, "How do we encapsulate the emergence of complex behaviors from interactions between different components?" Chapter 5 proposes reframing this question as an error regression, using graph neural networks to adjust for the "error" — *i.e.*, emergent behavior — incurred by composing multiple basis Navier-Stokes simulations into one large simulation. Lastly, given solution field data, this dissertation asks, "Under what conditions can we detect novel device behaviors through computer-driven simulation and exploration?" Chapter 6 proposes a boundary representation method and modified a hierarchical clustering approach, called Silhouette-optimized Hierarchical Density-Based Spatial Clustering of Applications with Noise (SHDBSCAN), to identify clusters of fluidic devices with similar behaviors. This chapter shows that the solution field representation has a significantly stronger impact on detecting novel device behaviors than the clustering algorithm used, but that a significant challenge lies in capturing "interesting" behavior in the design space in the first place.

Overall, this dissertation illuminates promising simulation methods for automating functional discovery and initial work on using data-driven methods to analyze such data. It also highlights several challenges, including the curse of dimensionality, that plague such approaches.

Automated Simulation and the Discovery of Mechanical Devices

by

Kevin Chiu

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2024

Advisory Committee:
Professor Mark Fuge, Chair/Advisor
Professor Shapour Azarm
Professor Jeffrey Herrmann
Professor Johan Larsson
Professor William Regli

Table of Contents

List of Tables

List of Figures

List of Terminology and Abbreviations

Word	**Definition**
acquisition function	In GPR, a function that determines the next point in the domain to sample.
activation function	In NNs, a usually non-linear function to transform values between HLs. See Fig. 2.2.
adjusted Rand score	Metric for calculating clustering quality using ratio of correct and incorrect labels. Requires ground truth labels. See §2.1.2.5.
autoencoder (AE)	Class of NN used for dimensionality reduction. Made of an encoder and a decoder, they use reconstruction error to calculate gradients. See §2.3.5.3.
backpropagation	Computational method of calculating exact gradients by creating a ”tape” of the forward calculations and applying the Chain Rule in reverse. Often used in NNs for training.
basis geometry	Unit shapes that are composed to form bigger geometries. See Fig. 5.5.
basis simulation	Stand-alone simulation run using basis geometry. Unlike small geometries, not physically connected to other simulations. See §5.5.2.
behavior	In the Function-Behavior-Structure scheme, the physical phenomena that occur. For example, the pressure and flow field from a fluid simulation.
behavior space	The (often high-dimensional) space into which a vectorization of behavior (*e.g.*, BVecs) points.
behavior vector (BVec)	Method of vectorizing behaviors. Contains SF values on the borders, but none inside, of the domain. See §2.3.3.
big geometry	Geometries formed from multiple basis geometries. In this work, consist of 4x4 grids of basis geometries. See §5.5.1.

big simulation	Simulation run using big geometry. Here, Navier-Stokes with FEM. The target of our simulation composition work is to reproduce this. See §5.5.1.
boundary condition (BC)	Generally, a prescribed value or derivative value in a boundary value problem. Used interchangeably with DBC in this dissertation unless otherwise specified. See §2.2.2.
boundary value problem	Differential equation subject to BCs. Solutions to these must satisfy the equations and the BCs.
cluster tree	Tree-like structure that represents different levels of splitting data into clusters in HDBSCAN. Used in SHDBSCAN as well. See §6.3.
complete simulation sets	Simulation set that has a VF with appropriate SFs and sufficient BCs to define the solution. See §3.3.4.
core points	In density-based clustering methods, data points with sufficiently many neighbors.
creeping flow	Fluid flows with a Reynolds number ¡¡ 1. Also known as Stokes flow. Very thick, small, or slow flows (*e.g.*, corn syrup or microfluidic channels).
cross validation	Category of model quality assessment methods that use data splitting and resampling to predict out-of-sample performance. K-Folds is a common method. See §2.1.3.
data leakage	Information from the training dataset bleeding into the testing dataset through, *e.g.*, overlapping data points or shared hyperparameters. Usually inflates performance.
Density-Based Spatial Clustering of Applications with Noise (DBSCAN)	Clustering algorithm that crawls through regions of dense data to form clusters. See §2.1.2.2.
Dirichlet boundary conditions (DBC)	Specific class of BC that prescribes field values to locations. Used interchangeably with BC in this dissertation unless otherwise specified.
discrete Fourier transform (DFT)	Method of reconstructing, in this case, 2-D images using combinations of basis sinusoidal images. Can be used for dimensionality reduction. See §2.3.5.2.
edge	Connection between two nodes in a graph.
feature/feature vector	Values used as inputs to a ML model.

Finite Element Method (FEM)	Method for solving PDEs using combinations of basis functions. See §2.2.
folds	Single partition of a K-fold cross validation scheme.
function	In the Function-Behavior-Structure scheme, the use for which we assign a certain physical phenomena. For example, low pressure drop in one direction of flow in a device and high pressure drop in the other can be used as a diode.
Gaussian process regression (GPR)	Probabilistic regression method that uses observed data to improve an approximation of the true function. Uses a surrogate model based on kernels. See §2.1.1.4.
geometry (geometry representation, Geo. Rep.)	The domain on which the simulation will be run. In this dissertation, this means the region in which there is fluid.
graph	Mathematical construct that shows connections called edges between nodes. Used in GNNs.
graph neural network (GNN)	Generalization of convolutional NNs to work on arbitrarily shaped graphs. Often implements message passing between nodes. Can predict node-, edge-, and graph-level features. See §5.3.
grid search	Hyperparameter tuning method based on combinatorially searching sets of hyperparameters. See §2.1.4.
grouping/grouped cross validation	Modification of cross validation where sets of data points are kept together within either training or testing data sets. See §2.1.3.
hidden layer (HL)	Internal layer of neurons in deep NNs that are neither inputs nor outputs. See §2.1.1.3.
Hierarchical DBSCAN (HDBSCAN)	Extension of DBSCAN to account for varying cluster densities using hierarchical structures. See §2.1.2.3.
hyperparameter	Non-tunable, non-learned value that guides how a ML model learns. See §2.1.4.
incomplete simulation sets	Simulation set that does not have a VF with appropriate SFs and sufficient BCs to define the solution. See §3.3.4.
K-Nearest Neighbors (KNN)	Supervised learning algorithm for classification (KNNC) and regression (KNNR). Uses closest neighboring points to predict new labels. See §2.1.1.5 and §2.1.2.4.

kernel	Covariance function used as an internal model in GPR. Parameters are learned during training.
Kernel Search	Process of finding most appropriate kernels for a given application.
latent space	Embedding of data from a feature space to a (often lower-dimensional) manifold. Used in dimensionality reduction.
location	For BCs, the coordinates on which the BCs are applied.
loss	Error between the predicted and true values in regression. Used to optimize ML models. One common example is MSE. See §2.1.1.6.
machine learning (ML)	Statistical models used to learn patterns from data to generalize and predict on new data.
mean absolute error (MAE)	Possible loss metric. See Eq. 2.46.
mean percentage error (MPE)	Possible loss metric, but scaled. See Eq. 2.47.
mean squared error (MSE)	Possible loss metric. Mean of squared errors between predicted and true values. See §2.1.1.6.
message passing	Technique used in GNNs that uses information from neighboring nodes to inform predictions. See §5.3.
Navier-Stokes equations	Set of PDEs that govern fluid flows. Consists of conservation of mass and conservation of momentum. See §2.2.3.
Neural Network (NN)	ML models based on neural connections in animal brains. Uses tunable weights and non-linear activation functions to produce complex predictions. See §2.1.1.3.
node	Vertex in a graph. Connected to other nodes with edges.
ordinary least squares	Method of optimizing linear regression models that minimizes the sum of squared errors between predictions and true values. See §2.1.1.1.
parameter	In a ML model, a value that can be learned during the training process.

partial differential equation (PDE)	Equation that computes a function between partial derivatives of a multivariable function. Often difficult to solve analytically, so simulations are used. See §2.2.3.
partially complete simulation sets	Simulation set that has a VF with appropriate SFs and excessive or unrelated BCs to define the solution. See §3.3.4.
performance space	Manually defined space in which certain types of behavior are maximally separated from each other. See §6.5.2.
pipeline	Collection of common steps in ML model usage. Often consists of data collection, preprocessing, model training, and postprocessing.
port	Specified inlet or outlet in our geometry randomization. Locations of interaction with other devices in Ch. 6. See §2.2.1 for geometry generation or Fig. 2.18 where the ports are circled.
Principal Component Analysis (PCA)	Linear method that projects data into the axes of maximum variance. Often used in dimensionality reduction. See §2.3.5.1.
quantity of interest (QoI)	Calculated or measured value that are important in some way. Used in Ch. 4 as a surrogate for physical realizability.
radial basis function (RBF)	Commonly used kernel. See Eq. 2.14.
randomized search	Hyperparameter tuning method based on choosing sets of randomized hyperparameters. See §2.1.4.
regression	Class of supervised ML models that attempt to predict some unknown quantity from given inputs. See §2.1.1.
regularization	Category of methods to reduce overfitting in ML models by penalizing extreme parameter values.
silhouette score (SS)	Metric for calculating clustering quality based on intra- and inter-cluster similarities. Does not require ground truth labels. See §2.1.2.5.
Silhouette-optimized HDBSCAN (SHDBSCAN)	Extension of HDBSCAN with modified cluster persistence and hyperparameters optimized using SS. See §6.3.
simulation set	Group of BCs, SFs, and a VF that may define a simulation setup.

small simulation	Single decomposed portion of a big simulation. Unlike basis geometries, physically connected to other simulations.
solution field (SF)	Function that gives the field values for each point in a simulation's domain after the simulation has been run. Somewhat interchangeable with simulation results. See §2.4.2.
Stokes equations	PDE that governs creeping flow. Linearization and simplification of the Navier-Stokes equations by ignoring advective inertial forces. See §2.2.3.
structure	In the Function-Behavior-Structure scheme, the physical shape of a device. For example, the walls and cavities of a fluidic oscillator.
surrogate model	Function used to approximate another, more complex, function. Often some sort of ML model.
train-test split	Class of methods for partioning some data for ML training and some data for evaluation on out-of-sample data. See §2.1.3.
value	For BCs, the numerical quantity that is assigned.
variational form (VF)	Formulation of a PDE into a form usable in FEM.
viable	Describes simulation sets that contain enough information to define a simulation fully; includes both complete and partially complete simulation sets. See §3.3.4.

Chapter 1: Introduction

When humanity first started using tools and building machines, most were made using trial-and-error approaches. Realizing that some materials were harder than others, one can use harder materials to shape softer ones. From these observations and trial-and-error approaches, the design process as we know it today was started.

As we learned more about our physical world, we developed models and approximations to predict how things we had never seen before would behave. From Isaac Newton's Laws of Motion to machine-learning-based models of multi-physics fluid motion, we derived equations and computational procedures that predicted and governed everything we could see around us. Although some of these equations begin to fall apart in modern applications, many of the same fundamentals are still valid. However, manually tuning our models for these modern applications limits the range of devices we discover to those that we, as human engineers, can conceive or have the time to explore. Automating this discovery process could expand the domain of what we are able to explore to find novel devices that take advantage of unique, yet unintuitive, physical interactions, leaving human engineers free to work on higher-level problems.

In essence, we want to explore the question of "What device functionality emerges organically from knowledge of various physical laws?" In this dissertation, we use techniques drawing from machine learning (ML) to enhance our ability to explore novel designs automatically dis-

covered from physics simulations. This is split into three main topics: automated simulation, composing multiple simulations, and automated device discovery from simulation data.

1.1 Topic 1: Automating Physics Simulation for Exploration

Q1: How do we enumerate possible boundary conditions for a given physical law that can lead to well-defined solutions to a given partial differential equation? In this section, we want to automate the process of simulating different physical phenomena. We started by wanting to develop methods to allow fully automatic Finite Element Method (FEM) simulations to be set up and run, such that a machine could autonomously explore the physical fields that result from a given design, without requiring human intervention. However, we ran into several critical obstacles during this process; Ch. 3 and Ch. 4 are our responses to some of those obstacles, and consisted of two main technical challenges:

Chapter 3: Checking Types of Boundary Conditions in Finite Element Simulations While the heavy lifting of solving enormous systems of linear equations for simulations has been relegated to GPUs and high-performance computing clusters, setting up simulations is still relatively manual. As many engineering students learn early in their careers, setting up simulations correctly is much more difficult than it seems. Even intuitive and powerful libraries like the FEniCS library [1] come with a steep learning curve, as the author has personally experienced when trying to apply a 3-D vector boundary condition (BC) onto a scalar electrostatic potential field. Ensuring that a FEM simulation can run before attempting to run it is difficult to automate because this process is usually done by human experts with prior knowledge and intuition about the type of physics that will be simulated.

Chapter 3 tackles part of this problem with a type-based indexing scheme for ensuring the various parts of a FEM simulation match appropriately. We show two properties that are required for all *complete* simulations, which — combined with the proposed type-based indexing — provide some steps that can ultimately lead to a framework to set up and run simulations automatically once given appropriate BCs [2]. However, this approach assumes we are given a set of valid BCs; thus, we also need to define what constitutes appropriate sets of BCs. Exploring what makes BCs "valid" is the purpose of Ch. 4.

Although in this dissertation we only test our contributions on a limited set of types of simulations, in principle this approach can extend to many classical FEM simulations or physics, including heat transfer, solid mechanics, and turbulent fluid flows. Specifically, the contributions in Ch. 3 should be extensible to other simulation methods with moderate modifications as long as the simulation solves an initial value problem through a system of linear equations in some form.

Chapter 4: Physically Realizable Boundary Conditions Choosing sets of BCs that are valid is not trivial; a simple example of a *physically unrealizable* simulation is a pipe with two inlets. A physically unrealizable setup for such a simulation could be difficult for a human expert to identify, even though intuitively, we often know what will or will not work. Thus, even if a simulation runs with a given set of physically unrealizable BCs, we should not be confident that such a simulation is trustworthy.

To address this, we train five types of regression algorithms on numerical information about the simulation setups and low-dimensional representations of the fluid domain to predict the magnitude of the velocity field in Ch. 4. For simplicity, we tested only 2-D Stokes flow simulations with a discrete subset of possible BCs on randomly generated geometries as discussed in §2.2.1.

We show that applying Resolution Selection — as described in §2.3.3.2 — on the BCs and an autoencoder on the geometry representation, combined with a fully connected Neural Network predicts the pressure field magnitudes with the least mean squared error (MSE) in our experiments. Predicting the pressure field magnitudes as a surrogate for physical realizability is a single test bed for our contributions, and the chapter discusses extensions to other types of physics as avenues of future work.

1.2 Topic 2: Composition of Multiple Simulations

Q2: How do we encapsulate the emergence of complex behaviors from interactions between different components? Complex devices and behaviors are, unsurprisingly, more complex and expensive to simulate. This is due in part to the existence of long-distance and emergent phenomena — where certain interesting or useful patterns only appear at certain scales — and the increasingly complex BCs.

We propose that, rather than simulating an entire system, we simulate several small "basis simulations" ahead of time, then later compose them into one large surrogate of the larger system after correcting for long-distance phenomena. We expand on our previous work in [3] on composing pipe simulations and use Graph Neural Networks (GNN) [4] to predict the error that arises from composing smaller simulations. From preliminary exploratory work, we know that simulations must include some sense of the global geometry. Graphs provide an intuitive representation of big simulations as a collection of small simulations, so we take inspiration from various fields [5, 6, 7, 8] and use GNNs in this chapter, though to our knowledge, there has not been published work using GNNs to reason and compose physical systems on whole images. We

limit our testing to fluid flows near $Re = 100$ range, as this region of Reynolds numbers provides nontrivial flows throughout the environment. We end the chapter with a discussion of extending our approach to other types of simulations.

1.3 Topic 3: Discovering Device Function from Simulations

Q3: Under what conditions can we detect novel device behaviors through computer-driven simulation and exploration? Although the advances from the prior chapters may allow us to simulate many possible devices automatically, this does not mean that it will be straightforward to discover or determine unknown or interesting new functions from those simulation results. We hypothesize that, from a sufficiently large number of simulations, we can discover groups of devices that produce similar behaviors, even if they do so in different ways and even if we do not know ahead of time exactly what specific behaviors we are searching for.

In Ch. 6, we show that, given enough devices' simulation results, we can discover some groups of behavior without necessarily knowing ahead of time the behaviors for which we are searching. Our contributions in this chapter are two-fold: first, we propose *behavior vectors* and heuristics for dimensionality reduction as a method of removing irrelevant simulation information while retaining discriminative information. Second, we modify a clustering algorithm to produce more intuitive and usable clustering on our behavior vectors. Combined with the new vector representation, we can cluster our data with an adjusted Rand score of approximately 0.55 on some applications. However, we find that the type of task has the most affect on clustering performance, while the preprocessing and clustering algorithms provide some improvements in clustering, though not universally.

1.4 Outline and Summary of Contributions

Chapter 2 first provides background information and definitions of various concepts used throughout the dissertation that should help readers better understand the experiments and scientific contributions of this dissertation. Eventual fully automated ML frameworks for automated design will need structures to allow flexible, yet robust, multi-physics simulations while ensuring that such simulations follow the laws of physics that govern our universe. We draw on the field of ML to provide automated techniques to fulfill these needs. We begin with Chapters 3 and 4 focusing on allowing flexibility in the physics being simulated while keeping such simulations grounded in reality. These chapters encode some of the human intuition for checking that given simulations follow the laws of physics. We then look to Ch. 5 to predict emergent phenomena from connectivity information between simulations, though with the additional cost of training a ML model. With ML models, the more times a model is used after training, the cheaper the amortized cost of ML training is. As such, the large quantity of simulations used in Ch. 6 will eventually take advantage of composing multiple multi-physics simulations to render a fully automated pipeline that can explore and discover new devices without human intervention. We end with a summary of the contributions and a synthesis of this dissertation in Ch. 7.

In summary, the scientific contributions of this dissertation are:

1. We propose a type-based indexing scheme to connect various parts of FEM simulations robustly, subsequently allowing categorizing simulation setups using that indexing scheme. We test this scheme on three types of equations and show that our categorizing exactly matches empirical tests for constructing fully defined FEM simulations. This contribution is only tested on Dirichlet BCs, and extending this approach to new types of simulations

requires some manual setup.

2. We formulate predicting physical realizability in Stokes flow simulations as a regression problem for a quantity of interest (QoI) derived from the simulation results. We compare several preprocessing and regression methods and find that our Resolution Selection heuristic, an autoencoder, and a fully connected NN predict the QoI with the lowest MSE. Although we test these contributions on Stokes flows, we believe this approach can extend to other types of simulations through careful specification of the QoI.

3. We generalize the compositional method from [3] by developing a heuristic to assign BCs to intermediate simulations in Navier-Stokes simulations. We compare several architectures and find that a GCN with 1 hidden layer and 819 neurons per layer gives us the smallest MSE of the GNNs we test on this type of task, but that Linear Regression gives us about half the MSE of any GNNs we test. We discuss the limitations and assumptions of this method in extension to other types of physics.

4. We modify a popular clustering algorithm and propose a dimensionality reduction heuristic to improve clustering on device discovery tasks. We show that applying our dimensionality reduction heuristic generally improves clustering, even if the modified clustering algorithm does not recapture the calculated labels as well through an adjusted Rand score. However, our current randomization does not necessarily guarantee that many "interesting" devices are captured, even if they theoretically exist within the single physics of our design space. The methods we employ are agnostic to the type of simulation, so we expect that this approach is easily extensible to other types of FEM simulations with minimal modifications.

The next chapter introduces some useful background material that helps set the stage for the main

contributions that we present in later chapters.

Chapter 2: Background

By necessity, this dissertation discusses a variety of tools, algorithms, methods, and techniques, spanning a variety of fields. Experts in any of these fields individually are likely familiar with any one of these fields, but complete knowledge across all of them is less common. This chapter hopes to bridge that divide by providing some background information for a variety of these fields. The methods discussed here are generally widely known in their various fields, if not necessarily the "state-of-the-art"; a deeper dive into more advanced methods, when needed, is provided in the corresponding later chapters.

This chapter contains four main parts: Machine Learning (ML), Simulation, Representation (of information), and a Summary of Common Terms and Abbreviations. Many of the algorithms in these parts are used in various later chapters.

2.1 Machine Learning

Broadly speaking, there are three main types of ML: supervised, unsupervised, and reinforcement learning. This dissertation focuses on a subset of supervised and unsupervised methods, but these paradigms differ by their acquisition of the "ground truth," or objectively labeled, data. As the names imply, supervised algorithms require a "supervisor" to provide ground truth labels, while unsupervised does not; reinforcement learning derives its own labels through trial

and error and prompting its environment.

Regardless of the learning paradigm, most ML implementations have a fairly general *pipeline* consisting of four steps: data collection, preprocessing, ML model training, and postprocessing. In the first step, the researcher generates or collects data, either from experimentation or simulation, providing the dataset from which the algorithm will be trained. The researcher then preprocesses the data, often through scaling, cleaning up missing or faulty points, applying dimensionality reduction, and separating the data into different categories for training and testing. With the training data, the researcher trains the model, tuning the model's parameters using an optimization algorithm, based on some sort of convergence metric or other measure of performance. Finally, the researcher postprocesses the data, either by undoing the preprocessing steps or by calculating other *quantities of interest* (QoI) from the model's output.

In general, the most computationally expensive steps of this pipeline are either the data collection phase — *e.g.*, when using computationally expensive fluid dynamics simulations to generate data — or the training phase — *e.g.*, when performing backpropagation steps to tune a Neural Network's (NN) weights. If an additional ML model is used for preprocessing (*e.g.*, autoencoders for dimensionality reduction, as in §2.3.5.3), then the preprocessing step may be computationally expensive as well. However, most of these steps are programmatically automated with the exception of data collection. Even now, data collection and generation often require a researcher to specify the parameters for such a process, requiring both experience and intuition to generate high-quality data.

2.1.1 Supervised Learning: Regression

In ML, supervised models require ground truth labels on their data. Because these ground truth labels are often expensive or difficult to acquire, ML models are used to produce "good enough" approximations of the labels in pipelines where many labels are needed, such as future extensions of our work in Ch. 5 and Ch. 6. In this section, we focus on a subclass of supervised models called *regression*. In regression problems, the model attempts to predict a numerical value (or values, in multidimensional cases) from the input data. Many different types of regression algorithms exist, but we discuss five of them in this chapter.

2.1.1.1 Linear Regression

Linear Regression is likely the most widely known and used regression model. Although Linear Regression is a linear model, its flexibility, speed, easy implementation, and intuitiveness make it a powerful model for a wide range of problems. The base implementation of Linear Regression, often called *ordinary least squares*, minimizes the least squares error, as seen in Eq. 2.1.

$$L = \sum_{n=1}^{N} (y_n - \hat{y}_n)^2 \tag{2.1}$$

In this equation, L — for Loss — is the least squares error, N is the number of data points, y_n is the ground truth value of the n-th data point, and $\hat{y}_n$ is the predicted value of the n-th data point. In matrix form, this is equivalent to

$$L = ||y - X\beta||^2 \tag{2.2}$$

where y is the $N \times 1$ matrix of outputs, X is the $N \times m$ matrix of inputs, and β is the $m \times 1$ matrix of coefficients.

There are many methods to calculate the coefficients of a Linear Regression model, and we encourage curious readers to implement such methods as an exercise. Here, we will discuss one method that uses intuitive, if expensive, matrix inversion steps to calculate these coefficients.

Consider a m-input single-output Linear Regression Model. For a single data point, a Linear Regression model is a matrix multiplication

$$X\beta = y \tag{2.3}$$

where y is the 1×1 sized output value, X is the $1 \times m$ input matrix, and β is the $m \times 1$ matrix of coefficients. We know X and y, so our goal is to solve for β. Multiplying both sides by X^T ensures the matrix in front of β is square.

$$(X^T X)\beta = X^T y \tag{2.4}$$

We now invert $X^T X$ and multiply this on both sides to get β.

$$\beta = (X^T X)^{-1} X^T y \tag{2.5}$$

In practice, such models are rarely implemented from scratch in ML applications. For example, the following code snippet uses the implementation of Linear Regression within the Python Scikit-learn library.

```
# Assuming that xTrain, yTrain, xTest, and yTest,
# representing the training data, training labels,
# testing data, and testing labels, exist, respectively.

from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit(xTrain)
yTestPred = regressor.predict(xTest)
```

`yTestPred` can now be compared to `yTest` via a scoring metric, as in §2.1.1.6.

2.1.1.2 Elastic Net

Elastic Net is an extension of Linear Regression that includes penalization terms for the coefficients. Specifically, it penalizes the L^1 and L^2 norms of the coefficients, as in Eq. 2.6,

$$L = ||y - X\beta||^2 + \alpha R||\beta||_1 + 0.5\alpha(1 - R)||\beta||_2^2 \tag{2.6}$$

where $||\beta||_1$ and $||\beta||_2$ denote the L^1 and L^2 norm of β, respectively, α is the relative weight of the penalties to the error terms, and R is a relative weight of the L^1 and L^2 norms to each other between 0 and 1, inclusive. In Ch. 4, α and R are set using a cross validation scheme (§2.1.3), as described in §2.1.4. The Scikit-learn library packages this model selection, as shown in the code snippet below, by combining the cross validation and the Elastic Net training into one class.

```
# Assuming that xTrain, yTrain, xTest, and yTest,
# representing the training data, training labels,
```

```
    # testing data, and testing labels, exist, respectively.


    from sklearn.linear_model import ElasticNetCV
    regressor = ElasticNetCV()
    regressor.fit(xTrain)
    yTestPred = regressor.predict(xTest)
```

`yTestPred` can now be compared to `yTest` via a scoring metric, as in §2.1.1.6.

2.1.1.3 Neural Network Regression

NN models are inspired by biological neurons by modeling neurons as nodes and connections between neurons as weights between nodes. At its base level, NN models use an alternating series of linear matrix multiplications and non-linear function activations to create complex non-linear behavior. For a pedagogical example, consider the network shown in Fig. 2.1, representing a NN model with three inputs (X_0, X_1, and X_2), two *hidden layers* (HL) with two and four nodes per layer, and one output value (y). Each of the HLs multiplies the output of the previous layer

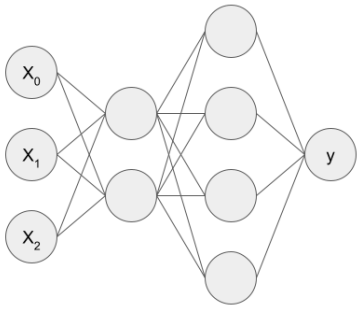


Figure 2.1: Pedagogical NN architecture.

by a tunable weighting matrix, W_i, and then applies an *activation function* to produce the output

for the next layer. Thus, Eq. 2.7 summarizes the calculation of a single layer of the network

$$h_i = \sigma(h_{i-1}W_i + b_i) \tag{2.7}$$

where h_i is the output of the i-th layer, h_{i-1} is the output from the previous layer, b_i is a tunable scalar that shifts the outputs by a scalar value, and σ is the activation function. Equation 2.8 calculates the overall NN model's output.

$$y = \sigma(\sigma(XW_0 + b_0)W_1 + b_1)W_2 + b_2 \tag{2.8}$$

The activation functions take a variety of forms; we refer readers to §5.1.2 Activation Functions (at the time of this writing) of [9] for a deeper discussion of three common types of activation functions: ReLU, Sigmoid, and Tanh. These activation functions are shown in Fig. 2.2 for the curious.

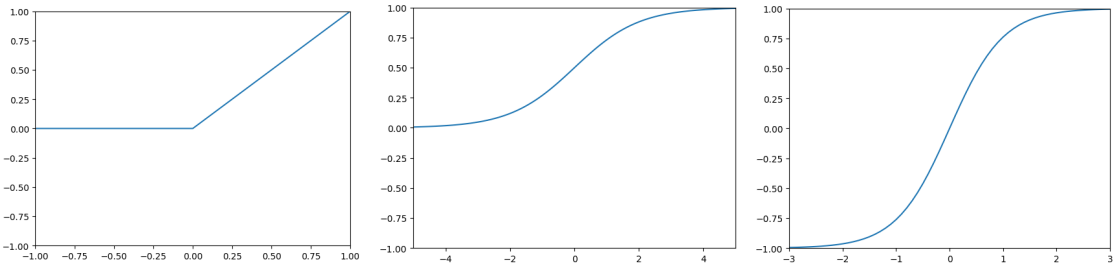


Figure 2.2: (A) ReLU, (B) Sigmoid, and (C) Tanh activation functions. Note that these are plotted across different x-ranges for visual clarity.

Training NNs is unsurprisingly difficult given the quantity of model parameters and calculations present. However, many implementations frame this training phase as an optimization problem where the objective function is the error on the predicted values, also called the *loss*. More complex models that use *regularization* may include other penalty terms, such as the mag-

nitude of the weights; a deeper discussion of such methods can be found in, *e.g.*, [10]. Computationally, the gradient of NN models is often calculated with *backpropagation*, a computational method of calculating exact gradients by creating a "tape" of the forward calculations, but such discussions are beyond the scope of this chapter. For a deeper discussion of NN models, we refer the reader to, *e.g.*, Ch. 3 and Ch. 5 of [9].

Practical implementation of NN models rest on setting the model's architecture, various hyperparameters, and training loops; the inner workings of training, matrix multiplication, and the like are abstracted away in most usage. We give one example implementation and training of a NN model using the PyTorch [11] and skorch [12] libraries in §A.1.

2.1.1.4 Gaussian Process Regression

Gaussian Process Regression (GPR) is a probabilistic regression method that uses observed data to improve an approximation of the true function. As more data is observed, the distribution of possible functions that fit the observed data tightens, thus more closely approximating the true function.

Mathematically, consider a Gaussian Process function f with a mean $m(x)$ and a covariance function $k(x, x')$ such that

$$f \sim GP(m, k) \tag{2.9}$$

Thus, any collection of function values queried from f has a mean vector μ and a covariance matrix K, such that

$$f(x_1), f(x_2), ..., f(x_n) \sim \mathcal{N}(\mu, K) \tag{2.10}$$

with the means $\mu_i = E(f(x_i)) = m(x_i)$ and covariances $K_{ij} = k(x_i, x_j)$. To predict the value

$\bar{f}_*$ and variance $V(f_*)$ on a new point x_*, we use Eq. 2.11

$$\bar{f}_* = K(x, x_*)^T (K(x, x))^{-1} y \tag{2.11}$$

and Eq. 2.12.

$$V(f_*) = K(x_*, x_*) - K(x, x_*)^T (K(x, x))^{-1} K(x, x_*) \tag{2.12}$$

However, K has hyperparameters, usually denoted θ, that must be learned during training. Learning θ usually amounts to maximizing the marginal log likelihood with respect to θ, as in Eq. 2.13.

$$\log p(y|X) = -\frac{1}{2} y^T (K(x, x))^{-1} - \frac{1}{2} \log |K(x, x)| - \frac{n}{2} \log 2\pi \tag{2.13}$$

Additionally, choosing the appropriate covariance function K, also known as the *kernel*, has a strong effect on the model prediction and represents a currently open problem within the ML community referred to as *Kernel Search*. Covering Kernel Search extensively is out of the scope of this section, but readers should be aware that optimizing over choices of possible kernels is non-trivial. Given this difficulty, a common kernel choice is the *radial basis function* (RBF), which takes the form in Eq. 2.14

$$k(x_i, x_j) = a^2 exp\left(-\frac{1}{2l^2} \|x - x'\|^2\right) \tag{2.14}$$

where a and l are learnable hyperparameters. Matérn kernels, as in Eq. 2.15 and elaborated in

Ch. 4, §4.2 of [13], are a generalization of RBFs.

$$k(x_i, x_j) = \frac{1}{\Gamma(\nu)2^{\nu-1}} \left(\frac{\sqrt{2\nu}}{l} d(x_i, x_j) \right)^{\nu} K_\nu \left(\frac{\sqrt{2\nu}}{l} d(x_i, x_j) \right) \tag{2.15}$$

Here, $d(x_i, x_j)$ is the Euclidean distance, Γ is the gamma function, and K_ν is a modified Bessel function. As $\nu \to \infty$, the Matérn kernel converges to the common RBF kernel. We refer interested readers to Rasmussen and Williams [13] and Zhang *et al.* [9] for a more in-depth discussion of GPR.

In the example Scikit-learn implementation below, the GPR uses a RBF kernel.

```
# Assuming that xTrain, yTrain, xTest, and yTest,
# representing the training data, training labels,
# testing data, and testing labels, exist, respectively.


from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF
kernel = RBF()
regressor = GaussianProcessRegressor(kernel = kernel)
regressor.fit(xTrain)
yTestPred = regressor.predict(xTest)
```

`yTestPred` can now be compared to `yTest` via a scoring metric, as in §2.1.1.6.

2.1.1.5 K-Nearest Neighbors Regression

K-Nearest Neighbors Regression (KNNR) interpolates between nearby points to predict a new data point's output. As we can see in Eq. 2.16,

$$\hat{y}(X) = \frac{1}{K} \sum_{i=1}^{K} f(NN(X, i)) \tag{2.16}$$

where $\hat{y}(X)$ is the prediction for a new data point X, K is the number of neighbors, $NN(X, i)$ is the i-th nearest neighbor to X, and $f(x)$ is the true output of x, this model requires a method to calculate the nearest neighbors of X. Naïve implementations of this operation can be costly. Additionally, though not shown in Eq. 2.16, these neighbors can be weighted, *e.g.*, by inverse distance. Given this model's simplicity, KNNR models serve as a useful baseline for many ML regression comparisons.

For an example implementation, the following code snippet implements K-Nearest Neighbors Regression using the Scikit-learn library.

```
# Assuming that xTrain, yTrain, xTest, and yTest,
# representing the training data, training labels,
# testing data, and testing labels, exist, respectively.


from sklearn.neighbors import KNeighborsRegressor
regressor = KNeighborsRegressor(
    n_neighbors = 5,
    weights = 'uniform'
```

```
)
regressor.fit(xTrain)
yTestPred = regressor.predict(xTest)
```

`yTestPred` can now be compared to `yTest` via a scoring metric, as in §2.1.1.6.

2.1.1.6 Regression Metrics

Although various regression metrics exist, this section focuses on calculating the *mean squared error* (MSE). As its name suggests, the MSE is the mean of the squared errors between a prediction and the true values, as seen in Eq. 2.17.

$$MSE = \frac{1}{N}\sum_{n=1}^{N}(y_n - \hat{y}_n)^2 \tag{2.17}$$

In this equation, N is the number of data points, y_n is the true value, and $\hat{y}_n$ is the predicted value. Based on this formulation, the MSE has a lower bound of 0 but no upper bound. Many standard ML and statistics libraries include a standard implementation of the MSE.

2.1.2 Unsupervised Learning: Clustering

In contrast, unsupervised models do not require ground truth labels on their data. In Ch. 6, we use a type of unsupervised learning called clustering, which aims to group data points together; Fig. 2.3 shows a pedagogical dataset and the results of several clustering algorithms.

Many different types of clustering algorithms exist, but we discuss three of them in this chapter.

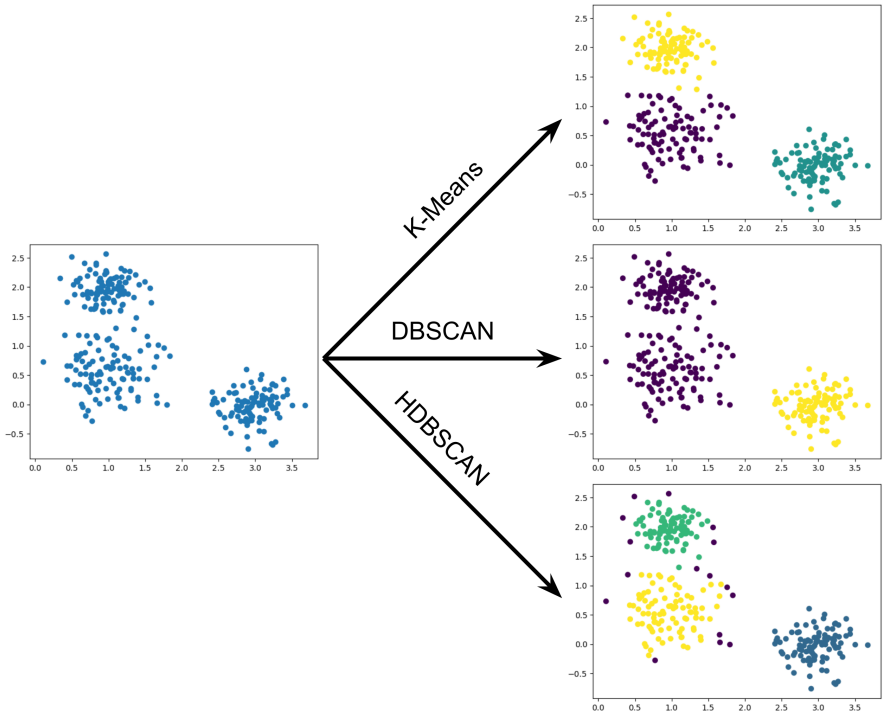


Figure 2.3: Example clustering on pedagogical data using K-Means (§2.1.2.1), DBSCAN (§2.1.2.2), and HDBSCAN (§2.1.2.3).

2.1.2.1 K-Means Clustering

K-Means clustering is a widely used linear clustering method that iteratively updates predicted cluster centers. The naïve implementation, also called Lloyd's algorithm [14], alternatively assigns each data point to the closest cluster center and moves the predicted cluster centers to the centroid of the data points currently assigned to that center. Given an initial set of k predicted cluster centers, named $m_1^{(1)}, m_2^{(1)}, ..., m_k^{(1)}$, Lloyd's algorithm assigns each data point to a cluster in a given iteration through Eq. 2.18,

$$S_i^{(t)} = \left\{ x_p : \left|\left| x_p - m_i^{(t)} \right|\right|^2 \leq \left|\left| x_p - m_j^{(t)} \right|\right|^2 \quad \forall \quad j, \quad 1 \leq j \leq k \right\} \tag{2.18}$$

where $S_i^{(t)}$ is the set of points in the i-th cluster at the t-th iteration, and x_p is a data point. Lloyd's algorithm then updates the predicted cluster centers by calculating the centroid of the points in each cluster through Eq. 2.19.

$$m_i^{(t+1)} = \frac{1}{\left|S_i^{(t)}\right|} \sum_{x_j \in S_i^{(t)}} x_j \tag{2.19}$$

We use the Scikit-learn implementation, shown in the code snippet below.

```
# Assuming that xTrain, yTrain, xTest, and yTest,
# representing the training data, training labels,
# testing data, and testing labels, exist, respectively.
# Also assuming a given number of clusters, K.


from sklearn.cluster import KMeans
clusterer = KMeans(n_clusters = K)
clusterer.fit(xTrain)
yTestPred = clusterer.predict(xTest)
```

`yTestPred` can now be compared to `yTest` via a scoring metric, as in §2.1.2.5.

2.1.2.2 Density-based Spatial Clustering of Applications with Noise

Density-based Spatial Clustering of Applications with Noise (DBSCAN) [15] is a clustering algorithm that crawls through data points, labeling those in dense areas as clusters while labeling those in sparse areas as noise. In general, DBSCAN can be broken into three main steps [16].

1. Finding points with neighbors within ϵ distance, and labeling those with at least minPts number of neighbors as *core points*.

2. Identifying the core points within ϵ distance of each other, and labeling those "close enough" core points as part of the same cluster.

3. Labeling the remaining (non-core) points with the closest core point labels within ϵ distance, or labeling the non-core point as noise if no core points are close enough.

In this algorithm, ϵ is a hyperparameter that controls the distance within which to data points are considered "neighbors," while minPts is the number of neighbors a point must have to be considered a core point. We refer readers to Ester *et al.* [15] and Schubert *et al.* [16] for more details of the algorithm. In Ch. 6, we use the Scikit-learn implementation, shown in the code snippet below.

```
# Assuming that xTrain, yTrain, xTest, and yTest,
# representing the training data, training labels,
# testing data, and testing labels, exist, respectively.
# Also assuming appropriate hyperparameters,
# eps and minPts, are  given.


from sklearn.cluster import DBSCAN
clusterer = DBSCAN(eps = eps, min_samples = minPts)
clusterer.fit(xTrain)
```

Because DBSCAN does not have a native "predict" function, we use a K-Nearest Neighbors Classifier from § 2.1.2.4 to implement out-of-sample prediction. While DBSCAN can detect

clusters of non-convex or "blob-like" shapes, it is restricted to finding clusters of a single density. Additionally, two ground truth separate clusters, if connected by a sufficiently dense "bridge" of data points, can be mistakenly labeled a single cluster.

2.1.2.3 Hierarchical Density-based Spatial Clustering of Applications with Noise

Hierarchical Density-based Spatial Clustering of Applications with Noise (HDBSCAN) [17] modifies and extends DBSCAN. Rather than separating clusters with a static density, HDBSCAN builds a hierarchical *cluster tree* of the data points and labels sufficiently persistent subclusters in the tree as a cluster. In general, HDBSCAN can be broken into five main steps [18].

1. Calculate mutual reachability by spreading out data points in regions with low density while condensing regions with high density.
2. Generate a minimum spanning tree of the transformed data points.
3. Build the hierarchy of connected subclusters in the cluster tree.
4. Condense the hierarchy such that all clusters contain at least the minimum cluster size number of points.
5. Extract persistent clusters from the hierarchy using a stability calculation.

We refer readers to the work by Campello, Moulavi, and Sander [17] for details on the algorithm and the library [18] for details on the implementation.

Although the Scikit-learn library now has HDBSCAN natively, this implementation does not include an out-of-sample prediction function. Instead, we use McInnes's implementation [18], which includes a prediction function for out-of-sample data points.

```
# Assuming that xTrain, yTrain, xTest, and yTest,
# representing the training data, training labels,
# testing data, and testing labels, exist, respectively.
# Also assuming appropriate hyperparameters,
# eps and minPts, are  given.

from hdbscan import HDBSCAN
clusterer = HDBSCAN(eps = eps, min_samples = minPts)
clusterer.fit(xTrain)

# Approximate prediction on new data points
import hdbscan
yTestPred = hdbscan.approximate_predict(clusterer, xTest)
```

`yTestPred` can now be compared to `yTest` via a scoring metric, as in §2.1.2.5.

2.1.2.4 K-Nearest Neighbors Classifier

K-Nearest Neighbors Classification (KNNC) is a vote-based classification algorithm that labels new data using the nearest previous points' labels. Although not an unsupervised clustering algorithm, it can be used to implement prediction on out-of-sample data in other clustering algorithms.

The most computationally expensive, and thus most critical, step of KNNC is the calculation of the K nearest neighbors. Many implementations exist, but we use the Scikit-learn library's

implementation of this classifier, using five neighbors inversely weighted by distance, as seen in the code snippet below.

```
def customPredict(xTrain, yTrainPred, xTest):
    # xTrain, the training data
    # yTrainPred, the predicted labels for the training data
    # xTest, the testing data

    # Returned variables:
    # Predicted labels for the testing data

    from sklearn.neighbors import KNeighborsClassifier
    knn = KNeighborsClassifier(
        n_neighbors = 5,
        weights = 'distance'
    )
    knn.fit(xTrain, yTrainPred)
    return knn.predict(xTest)
```

2.1.2.5 Clustering Metrics

Two broad classes of scoring metrics are used with clustering: those that require ground truth labels, and those that do not. In Ch. 6, we use the Adjusted Rand Index [19] — which requires ground truth labels — and the Silhouette Score [20] — which does not.

Given a set of predicted labels S_{pred} and a set of true labels S_{true}, the original Rand Index intuitively measures the proportion of labels that both sets agree are in the same (or different) partitions. Equation 2.20 measures the Rand Index.

$$RI = \frac{TS + TD}{TS + FS + TD + FD} \tag{2.20}$$

TS is the number of pairs of elements that have the **same** labels in both S_{pred} and S_{true}. FS is the number of pairs of elements that have the **same** labels in S_{pred} but have **different** labels in S_{true}. TD is the number of pairs of elements that have **different** labels in both S_{pred} and S_{true}. FD is the number of pairs of elements that have **different** labels in S_{pred} but have the **same** labels in S_{true}. Mathematically, the Rand Index is also the probability that both partitions will agree on a randomly chosen pair of elements.

However a given model can potentially "guess" some of the partitioning correctly and achieve an artificially inflated performance. The Adjusted Rand Index accounts for these "lucky guesses" by normalizing by the expected values of the Rand Index, as in Eq. 2.21.

$$ARI = \frac{2 * (TS * TD - FS * FD)}{(TS + FD) * (FD + TD) + (TS + FS) * (FS + TD)} \tag{2.21}$$

Thus it is common to use the Adjusted Rand Index in place of the Rand Index and this is the measure Ch. 6 uses to report its results.

In contrast to the Adjusted Rand Index, the Silhouette Score (SS) does not require ground truth labels. The SS is a metric that compares how similar points are to others in the same cluster

(Eq. 2.22) and how different points are to points in other clusters (Eq. 2.23)

$$a(i) = \frac{1}{|C_i| - 1} \sum_{j \in C_i, i \neq j} d(i, j) \tag{2.22}$$

where i is a point in cluster C_i, $|C_i|$ is the number of points in C_i, and $d(i, j)$ is a distance function between point i and point j, such as an L^2 norm. Note that in Eq. 2.23, the $\min$ operator looks for the closest other cluster to i.

$$b(i) = \min_{j \neq i} \frac{1}{|C_j|} \sum_{j \in C_j} d(i, j) \tag{2.23}$$

These two metrics are combined in Eq. 2.24 to provide the SS of a single point.

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}} \tag{2.24}$$

The overall SS of a given clustering C is the mean of the SS of every point, as in Eq. 2.25.

$$SS(C) = \frac{1}{|C|} \sum_{i \in C} s(i) \tag{2.25}$$

In this dissertation, all future references to SS refer to the overall SS of a given clustering, not to the SS of a single point in a given clustering.

2.1.3 Assessing Model Quality

Given that high-quality data is available, various methods are used to assess model quality. To start, data is usually split into two sets: training and testing. The training data is used to

improve a model's performance, hence its name. The model "sees" this data and "learns" from it by updating its parameters to capture trends or make predictions. The testing data is used to measure a model's performance after training. This is data that the model has never seen before the testing phase, and the model is not supposed to adjust itself in response to the testing data. Metaphorically, training data are the homework assignments and practice exams that students use to study, and testing data are the final exam questions used to assess learning.

Various methods to split a dataset into training and testing sets exist, but the simplest is to assign, *e.g.*, the first 75% of data to the training set and the remaining 25% to the testing set. Other methods can improve this split by, *e.g.*, splitting the data such that both the training and testing sets have similar distributions. These methods are referred to generally as *train-test splits*. Regardless of how the data is split, critically, none of the data is used in both batches. When information from the training dataset bleeds into the testing phase, even implicitly, the model has *data leakage*. Data leakage can come in various forms, but the most obvious is using data in both the training and testing sets. This error causes the model to overestimate its performance on new data; the worse the leakage, the higher the overestimation.

One common family of methods to split the data is *cross validation*. In this dissertation, we specifically use K-fold cross validation, which scales well with varying dataset sizes, as opposed to, *e.g.*, Leave-P-Out cross validation. K-fold cross validation has a hyperparameter, K, that defines the number of *folds* that are used. For each fold, $\frac{1}{K}$ of the dataset is held for validation, and the remainder is used for training, as shown in Fig. 2.4. After training K different models, each with a different distinct subset of data held out for testing, the average performance of the model is reported. When performed on a training dataset, K-fold cross validation gives a more accurate estimate of the out-of-sample (*i.e.*, testing) performance than the training performance.

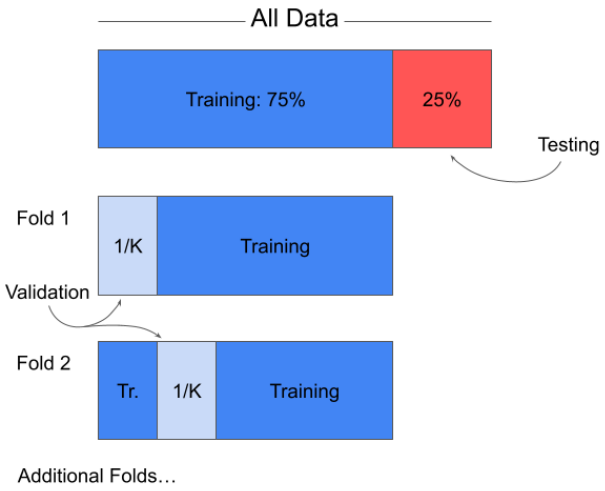


Figure 2.4: 5-fold cross validation on a dataset with a 75%-25% train-test split.

Chapters 4, 5, and 6 use a 75%-25% train-test split, as is common in this field. However, Ch. 5 splits the data by big simulations; in other words, all of the small simulations that make a single big simulation are assigned to training or testing together, and thus no big simulation will have part of its small simulations in training and part in testing. This practice is referred to as *grouping* or *grouped cross validation* and acts as a stronger test of generalization performance since test devices are grouped on a per-device level. Chapters 4 and 6 use 5-fold cross validation, as elaborated within the chapters.

The code snippet below shows a sample model being assessed with a 75%-25% train-test split and 5-fold cross validation implemented using the Scikit-learn library on a generic ML model called "model."

```
# Assuming that X, an N x M matrix of N
# data points, each with M features, and
# its corresponding labels Y, an N x 1 vector, already exist
# Also assuming that a supervised ML model, called "model"
```

```
# has been initialized


# 75%-25% train-test split
from sklearn.model_selection import train_test_split
xTrain, xTest, yTrain, yTest =
    train_test_split(X, y, test_size = 0.25)


# Calculating cross validation scores of "model"
import numpy as np
from sklearn.model_selection cross_val_score
CVScores = cross_val_score(model, xTrain, yTrain, cv = 5)
meanCV = np.mean(CVScores) # Mean score
```

2.1.4 Hyperparameter Tuning

Most ML models have some variables called *hyperparameters* that determine how the model functions. These are separate from *parameters*, which are adjusted in the training process, *e.g.*, with an ADAM optimizer [21] during gradient descent. Instead, hyperparameters are not changed during the pipeline; they are set *a priori* and held constant. However, hyperparameters can have drastic impact on a model's performance, and thus optimizing hyperparameters is an important step in the ML pipeline. For example, the number of predicted cluster centers in K-Means clustering is a hyperparameter, while the locations of those centers are parameters. Figure 2.5 shows an example clustering with K-Means clustering with three and two cluster cen-

ters, respectively, resulting in very different cluster qualities. While the centers are moved and updated in the training phase, the number of centers does not change. Basic implementations

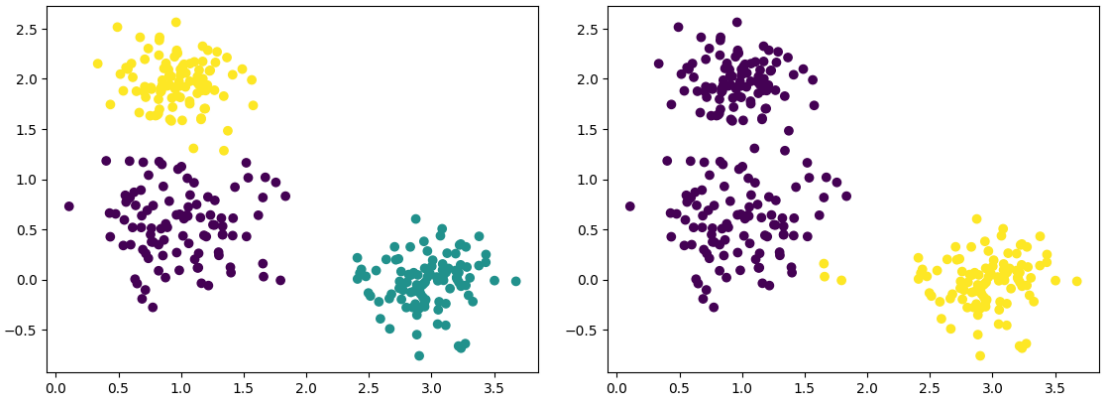


Figure 2.5: K-Means clustering on a pedagogical example with (A) an appropriate and (B) inappropriate number of predicted cluster centers. Different colors represent different clusters.

of hyperparameter tuning are often based on a *Grid Search*, where the possible space of hyperparameters is searched combinatorically and exhaustively. A modified implementation is a *Randomized Search*, where random sets of hyperparameters in the possible space are selected and tested. Randomized Search generally provides improved results over a Grid Search [22]. Algorithm 1 gives the pseudocode for one possible implementation of Grid Search, and Alg. 2 gives the same for Randomized Search with an equivalent number of iterations.

Algorithm 1 Sample Pseudocode for Grid Search

```
Set hyperparameter bounds and/or values
for each possible combination of hyperparameter values do
    Run algorithm with chosen hyperparameter values
    if the current performance is better than the previous best then
        Save the current hyperparameters
    end if
end for
Train the model with the best found hyperparameters
```

In practice, both Grid and Randomized Search may be inefficient for a larger number of hyperparameters or in applications where evaluating the objective function is expensive; in such

Algorithm 2 Sample Pseudocode for Randomized Search

```
Set hyperparameter bounds and/or values
Set total number of hyperparameter iterations
for the total number of hyperparameter iterations do
    Randomly choose a set of valid hyperparameter values
    Run algorithm with chosen hyperparameter values
    if the current performance is better than the previous best then
        Save the current hyperparameters
    end if
end for
Train the model with the best found hyperparameters
```

cases, researchers typically use Bayesian Optimization instead. Bayesian Optimization creates a *surrogate model*, or function to approximate another function, to be optimized using points sampled from the real function. The surrogate model in Bayesian Optimization is GPR, and a common choice is to use a Matérn kernel ($\nu = 2.5$), as discussed in §2.1.1.4. As more points are sampled, the posterior distribution of the model is updated to accommodate the newly acquired information. Points to be sampled are selected through an *acquisition function*. In this work, the acquisition function is an Upper Confidence Bound — where the acquisition function believes the best values will be, as in Eq. 2.26.

$$a(x; \lambda) = \mu(x) + \lambda\sigma(x) \tag{2.26}$$

The Upper Confidence Bound is a weighted sum between the mean and standard deviation of the Gaussian Process used in the surrogate model, with λ the relative weight between them. Higher values of λ weigh the uncertainty more, thus encouraging exploration, while lower values of λ weigh the known values more, encouraging exploitation. As one example, Nogueira [23] sets $\lambda = 2.576$, and we refer the reader to this implementation for details. Internally for the

experiments in this dissertation, the GPR uses a Matérn kernel with $\nu = 2.5$ and with parameters set to $\alpha = 1 \times 10^{-6}$, with y-value normalizing (*i.e.*, scaling outputs by removing the mean and scaling to unit variance), and five restarts of the optimizer. The specific implementation of Bayesian Optimization is beyond the scope of this chapter, but we refer readers to Nogueira [23] for further details. We integrate Nogueira's implementation of Bayesian Optimization with a general cross validation scheme from Scikit-learn, shown in §A.2.

2.2 Simulation

In this dissertation, we make generous use of physics simulations. Specifically, our work uses *Finite Element Method* (FEM) simulations to model physical phenomena, both as static fields and as dynamic motion. While these simulations are not perfect models of physical phenomena, they are significantly faster and cheaper to use than physical prototyping, especially in the exploratory work discussed in, *e.g.*, Ch. 6 of this dissertation.

The FEM approximates solutions to *partial differential equations* (PDE) that govern physical phenomena, such as the Navier-Stokes and continuity equations for fluid flows. We can represent the solutions to these equations with mathematical constructs (later dubbed *solution fields* (SF)) that, when combined with the PDEs (§2.2.3) and boundary conditions (BC, §2.2.2), portray how the physical fields would behave in a given domain (§2.2.1). At its core, the FEM is a structure for setting up a linear system of equations of the form:

$$Ax = b \tag{2.27}$$

By convention, A is called the stiffness matrix, b is the vector of solutions, and x is the vector

of variables. By this convention, the values in A and b are fully known; only the values in x are unknown. PDEs, SFs, and BCs provide a structured method to determine the dimensions of and coefficients in A and b. In practice, there are several types of BCs; this dissertation specifically uses *Dirichlet Boundary Conditions* (DBC), where the value of the SF at a certain point is prescribed. Unless noted otherwise, future uses of DBC and BC are interchangeable; both refer to Dirichlet Boundary Conditions and are elaborated in §2.2.2. We refer readers to [24] for a more in-depth description of the FEM.

In general, the simulation method is not critical to the work in this dissertation. We use the FEM as an approachable and easily implemented method of simulation, but any other simulation method (Finite Differences, Finite Volumes, mesh-free methods, *etc.*) are also valid as long as QoIs are calculable from the simulation results. For the rest of this dissertation, we will be assuming the FEM is used for simulations. The remaining parts of this section will introduce the specific ways we modeled the geometry, BCs, and specific PDEs and applied post-processing.

2.2.1 Geometry Generation

For Ch. 4 and Ch. 6, we use randomly generated geometries in our simulations. The method by which we generate these geometries can vary, but we detail the approach used in this dissertation below. Broadly speaking, our geometries reside in a square domain with either two or four *ports* (*i.e.*, inlets or outlets) and are formed from a randomized set of nodes and connectors, as mimics common features seen in microfluidic devices. Although unlikely, we can theoretically reproduce some existing fluidic devices with this approach.

We begin with a square domain between (0, 0) and (1, 1), as in Fig. 2.6. We represent

potential port locations as short black lines on the edge of our domain and potential node locations as blue dots. For four-port devices, we choose a subset of four of the potential port locations and a subset of the potential nodes.

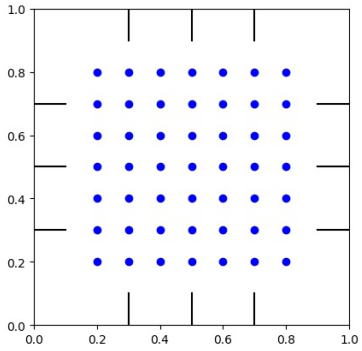


Figure 2.6: Potential node and port locations.

We start our list of nodes with the nodes for the ports. Before a potential node is added, we draw a random number between 0 and 1. If that number is below $\frac{7}{numNodes}$, we add another node and repeat the process. Else, we stop adding nodes.

This process leads to an interesting distribution of the number of nodes in a mesh. Figure 2.7 shows the number of nodes out of 100,000 samples. Note that the lower bound is 8 nodes, and the upper bound is 49 nodes.

With these potential nodes, we apply a Delaunay triangulation [25] on our nodes as in Fig. 2.8 to generate connections between nodes.

From initial testing, replacing these connections with fluid domains directly would result in a domain that is almost entirely filled with fluid, which we expect will not capture interesting phenomena. We thus take only one randomly selected edge of every triangle from the Delaunay triangulation, as in Fig. 2.9, to prune the connections. However, we ensure that every port is connected to each other, but any node that is not connected to the ports is removed.

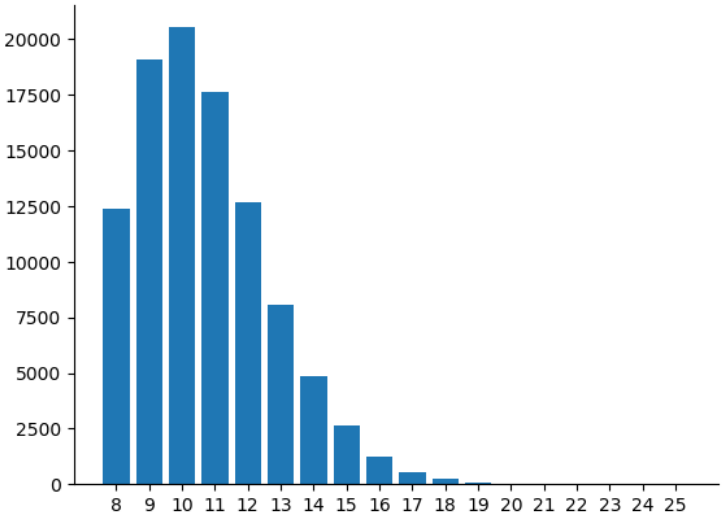


Figure 2.7: Distribution of number of nodes in a mesh over 100,000 samples.

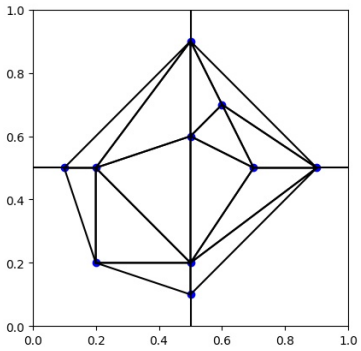


Figure 2.8: Delaunay triangulation on potential nodes on a sample 4-port device. Note that ports are always connected directly the edge.

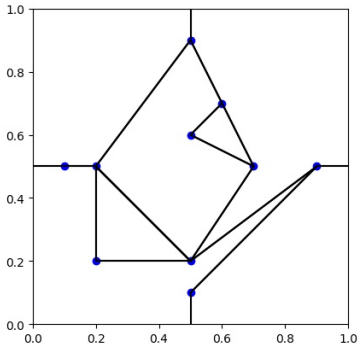


Figure 2.9: We keep one edge of every triangle and ensure that every port is connected.

We replace each node with a circular domain ($r = 0.05$) and each of these remaining connections with either a straight pipe ($width = 0.1$) or a semicircular arc (centered equidistant

from the two nodes, with an angle between $\frac{\pi}{2}$ and π, $width = 0.1$). With a 20% chance on each node, we also add a circular domain (with a radius $0.1 \leq r \leq 0.15$, jittered $0.9 \cdot r$ from the node) to allow for internal fluid circulation. Figure 2.10 shows one example simulation domain

Figure 2.10: Straight and curved pipes and circular domains between connected nodes.

that results from this process. We can see examples of straight pipes (bottom left), a semicircular pipe (bottom right), and circular domain (center top) in this domain. The simulation domain is then meshed through a software of choice (*e.g.*, pygmsh [26], gmsh [27], and meshio [28]). We include some example 4-port meshes in Figs. 4.2, 6.2, and 6.3.

In Ch. 4, we force all devices to have four ports, one at the center of each side of the initial square domain, as in Fig. 4.2. In Ch. 6, we relax this restriction to allow any of the potential port locations in Fig. 2.6 and to include 2-port devices; we refer the reader to Figs. 6.2 and 6.3. The process to generate 2-port meshes is identical except that only two ports are selected at the beginning.

2.2.2 Boundary Conditions

The simulations we use in this dissertation are classed as *boundary value problems*. As such, they require a set of *boundary conditions* (BC) that determine the phenomena along the

boundary of the simulation domain. While these BCs can take a variety of shapes and dimensions, we generally restrict this dissertation to fluid velocity, fluid pressure, and Poisson BCs. Although this does restrict some of the generalizability testing we could have done, the main contributions of this dissertation are generally applicable to different types of physics with minimal adjustments with the exception of Ch. 4, which we discuss in that chapter.

BCs have two critical parameters: *location* and *value*. The location is the region along the border at which a given BC is applied. The values are the numerical magnitude of the field at the corresponding BC locations. In the libraries we use, the location and values can both be functions of the coordinates, as exemplified in the code snippet below.

```
fluidVelocityBC = DirichletBC(
    V, # corresponding SF
    Expression(
        ('3*x[1]', 'x[0]/2'),
        degree = 2
    ), # value as function of coordinates
    'near(x[0], 0.5) && x[1] <= 2 && on_boundary' # location
)
```

In this example, we are applying a fluid velocity BC on points in the domain (as stored in the SF `V`) that are 1. on the boundary of the domain (`on_boundary`, as defined by the library), 2. have an x-coordinate (`x[0]`) near 0.5, and 3. have a y-coordinate (`x[1]`) less than or equal to 2. The values of the BC scale with the coordinates: $xVel(x, y) = 3y$ and $yVel(x, y) = \frac{x}{2}$. Implicitly, BCs have a dimensionality based on the number of value terms present. If the dimensionality of

the BC does not match that of the SF, the code throws an error stating that the dimensions do not match.

With the exception of Ch. 3, this dissertation only uses fluid velocity and fluid pressure BCs. Several of these BCs are common across the dissertation, so we discuss them in this chapter.

The most universally used BC in this dissertation is the "no-slip" condition. This BC assumes that fluid along the walls of the simulation are "stuck" on the wall. This means that the fluid does not move relative to the wall, nor does the fluid pass through the wall. In a 2-D space (as is used throughout this dissertation), the values of a no-slip BC are $(0, 0)$ and is denoted V_0. We apply this condition on all of the walls of our simulations unless specified otherwise.

In contrast to a BC for no fluid movement, we also have BCs for fluid movement, specifically for fluid entering our simulation. These BCs, called V_{in}, are applied on inlets to the system and force a given flow rate to enter the system. Although the maximum velocities vary across chapters, the velocity profile is generally either uniform or parabolic with the two ends of the parabola reaching $(0, 0)$ velocity flow. The parabolic shape closely matches the steady state velocity profile of laminar flow in an enclosed pipe, which matches our simulations. Increasing the maximum velocity of V_{in} profiles can result in higher Reynolds numbers, eventually reaching turbulent flow, which we do not address in this dissertation; thus, we generally use V_{in} velocities that result in relatively low Reynolds number flows. Both V_0 and V_{in} are fluid velocity BCs; the difference is only their values.

Aside from fluid velocity BCs, we also use fluid pressure BCs. These BCs are generally used for setting ambient pressure at outlets in the system and are denoted P_0. Because pressure is a scalar value, the value of P_0 is (0). Adding fluid pressure BCs with non-zero values can cause pressure differences that never reach steady state; as such, our fluid pressure BCs always have a

value of (0).

In addition, Ch. 3 also uses Poisson BCs. These BCs are scalar values and are implemented by the library similarly; we describe the issues that can arise in more detail in the relevant chapter.

2.2.3 Equations

Our simulations throughout this dissertation are governed by one of four types of PDEs: Navier-Stokes equations, Stokes flow equations, Poisson equation, or a synthetic equation of our design.

The Navier-Stokes equations govern fluid flows across an expanse of conditions and are simulated in Ch. 3, Ch. 5, and Ch. 6. They are a combination of conservation of momentum (Eq. 2.28) and continuity (*i.e.*, conservation of mass, Eq. 2.29) equations.

$$\frac{\partial u}{\partial t} + u \cdot \nabla u + \nabla p - \nu \Delta u = f \tag{2.28}$$

$$\nabla \cdot u = 0 \tag{2.29}$$

In this work, we use the Navier-Stokes equations for incompressible flows. We solve this iteratively using Chorin's method [29], which first calculates a tentative velocity by ignoring pressure in the momentum equation (Eq. 2.30) before calculating the pressure (Eq. 2.31) and velocity (Eq. 2.32).

$$\left\langle \frac{(u_h^* - u_h^{n-1})}{\Delta t_n}, v \right\rangle + \left\langle \nabla u_h^{n-1} \cdot u_h^{n-1}, v \right\rangle + \langle \nu \nabla u_h^n, \nabla v \rangle = \langle f, v \rangle \tag{2.30}$$

$$\langle \nabla p^n, \nabla q \rangle = -\frac{\langle \nabla \cdot u_h^*, q \rangle}{\Delta t_n} \tag{2.31}$$

$$\langle u_h^n, v \rangle = \langle u_h^*, v \rangle - \Delta t_n \langle \nabla p^n, v \rangle \tag{2.32}$$

In these equations, u and p are the unknown velocity and pressure fields, with test functions v and q, respectively. The tentative velocity in Chorin's method is denoted by u_h^*. Δt_n is the time step at step n, ν is the kinematic viscosity, and f an external force on the fluid. $\langle \cdot, \cdot \rangle$ denotes an inner product. These equations are iteratively solved — incrementing the time each iteration — until a stopping criterion is reached, usually as a maximum time or steady state criterion.

The Stokes equations are a linearization and simplification of the Navier-Stokes equations by ignoring advective inertial forces, and are simulated in Ch. 4. These equations, while similar to the Navier-Stokes equations, are only used in *creeping flow* for fluids, *i.e.*, when the Reynolds number is much lower than 1, as in very thick, very slow, or very small flows (*e.g.*, corn syrup or microfluidic channels).

$$-\nabla \cdot (\nabla u + pI) = f \tag{2.33}$$

$$\nabla \cdot u = 0 \tag{2.34}$$

The mixed variational form of the equations is

$$\int_\Omega \nabla u \cdot \nabla v + \nabla \cdot vp + \nabla \cdot uq \, dx = \int_\Omega f \cdot v \, dx + \int_{\delta\Omega_N} g \cdot v \, ds \tag{2.35}$$

and can be solved by approximating both the velocity and pressure simultaneously using linear solvers. Again, u and p are the unknown velocity and pressure fields with test functions v and q, respectively. Ω and $\delta\Omega$ are the domain and boundary, respectively, with respective differentials x and s.

The Poisson equation, in Eq. 2.36, has a variety of physical interpretations, such as electro-

static potential, graviational, and fluid pressure fields, and are used in Ch. 3.

$$\Delta^2 u = f \tag{2.36}$$

Here, Δ^2 is the Laplace, or second-order differential, operator; u and f are functions, though f is usually known and u is sought. Equation 2.37 gives the weak formulation of the Poisson equation.

$$\int_\Omega \nabla u \cdot \nabla v \, dx = \int_\Omega f v \, dx \tag{2.37}$$

where v is again the test function.

Finally, we simulate a synthetic variational form of an equation in Ch. 3. This equation (Eq. 2.38) does not have any physical interpretation and is constructed purely for experimental purposes to include fluid velocity, fluid pressure, and Poisson terms.

$$\langle u, v \rangle + \langle p, q \rangle + \langle s, t \rangle + f \cdot t + g \cdot t = 0 \tag{2.38}$$

Similar to previous equations, u, p, and s are the unknown velocity, pressure, and Poisson fields with test functions v, q, and t, respectively.

2.2.4 Post-Processing

Running a simulation is, in effect, defining a function for the field values at any point in the domain at any time. Thus, for our example fluid simulation, we can calculate the pressure and velocity at any point. This is critical in many of our ML applications because of the need to calculate QoIs that for the labels of our dataset.

In Ch. 4, we calculate the L^2 norm of the pressure field as a step toward the QoI. Given the pressure values at the mesh points in the domain are given in a vector p and each individual value can be identified with an index n as in p_n, the L^2 norm of p is

$$|p| = \sqrt{\sum_{n=1}^{N} p_n^2} \tag{2.39}$$

where N is the total number of points in the mesh. However, initial testing shows the L^2 norm directly scales with then number of V_{in} BCs we apply. To counter this, for a given simulation we divide the L^2 norm by the number of V_{in} BCs applied and use this value as the QoI.

In Ch. 6, we calculate diodicity of generated devices during ground truth labeling. The diodicity of a fluidic device is the ratio of its pressure loss with fluid flow in one direction to the other direction. The flow resistance in a single direction is the ratio of pressure loss to flow rate, as in Eq. 2.40,

$$R = \frac{\Delta p}{Q} \tag{2.40}$$

where R is the resistance, Δp is the pressure drop, and Q is the fluid flow rate. Calculating this resistance for both the forward and backward directions, we get Eq. 2.41,

$$Di = \frac{R_{forward}}{R_{reverse}} = \frac{\frac{\Delta p_{forward}}{Q_{forward}}}{\frac{\Delta p_{reverse}}{Q_{reverse}}} = \frac{\Delta p_{forward}}{\Delta p_{reverse}} \tag{2.41}$$

assuming that the flow rates are equivalent. However, our simulations give us a function for pressure, not a single pressure loss. Assuming that the pressures along the port are given in a vector p_{port}, we can calculate the pressure loss taking the difference in mean pressure across the

ports, as in Eq. 2.42,

$$\Delta p = \frac{1}{I}\sum_{i=1}^{I} p_i - \frac{1}{J}\sum_{j=1}^{J} p_j \tag{2.42}$$

where I refers to mesh points along the inlet port and J to mesh points along the outlet port.

2.3 Representation of Simulation Data for Machine Learning Applications

Data is generally formed into *features* or *feature vectors* before it is used in ML models. In general, data does not come to a ML model immediately ready for use, unless it is naturally represented by a single numerical value, like number of rooms in a house. Instead, it must be reformatted, usually into a vector or matrix, before it can be used in ML models. This section discusses various methods used in this dissertation of vectorizing data.

2.3.1 Related Work: Geometry and Simulation Representation

Although various methods to represent the simulation domains and results exist, unsurprisingly, many treat 2-D fields and geometries as images. This dissertation draws from these works; for example, many ([3, 30, 31, 32, 33, 34, 35, 36, 37]) use some method of pixelization to convert simulation data into vectors. We draw from these works in Ch. 4 and Ch. 5 in converting our geometries and simulation results into vectors.

Briefly, converting geometry and simulation data generally amount to the same process with only a difference of dimensionality. Geometries are generally treated as 1-D scalar fields; the numerical value, usually 0 or 1, indicates the presence or absence of solid material. For elasticity simulations, the solid material becomes "in" the domain; for fluid simulations, as in this dissertation, the gaps between solid material is "in." In Ch. 4, we discretize the square bounding

box of our geometries into a 101×101 grid and query if each point is inside or outside the fluid domain. Points that are inside the fluid domain are assigned a value of 1; those outside are assigned 0. We use 101 discretizations per edge as, from preliminary work, fewer discretizations lose features such as small cavities in the domain, while more discretizations quadratically increase the dimensionality of our inputs seemingly without providing significant additional information. Figure 2.11 shows this qualitatively.

(a) Original geometry (b) 5001×5001 (c) 101×101 (d) 21×21

Figure 2.11: Original geometry and representations at various discretization levels.

Simulation results, however, may be scalar or vector fields, depending on the type of simulation. This dissertation generally uses 2-D fluid simulations, which produces a 2-D vector fields (x- and y-velocity) and a scalar field (pressure). One common method to represent these fields is with images where a pixel's RGB values reflect the field magnitude. For instance, we have previously used the RGB channels of an image to represent the x- and y-components of fluid velocity flow while composing multiple fluid flows [3], similar to how Usman *et al.* [30] used images and deep learning to predict time-varying fluid fields. Similar representation methods have been used from atmospheric modeling [31] to wheel designs [32], among others [33, 34]. In Ch. 5, we ignore the pressure fields of our fluid simulations and focus only on the 2-D velocity fields, thus only using the red and green channels of the pixels. Extending our work to incorporate pressure is almost trivial, but doing so unnecessarily muddies visualization clarity.

2.3.2 Function Representation

An engineer's job, to some degree, is to manipulate the placement of material and geometries to create a *structure* that then predictably modifies some physical phenomena or state, creating a *behavior* that, when applied appropriately, performs a *function* that benefits humanity in some specific way. In other words, a structure is the physical shape of a device, like the curve of an airplane wing; a behavior is how physical fields change in response to that structure, like increased air pressure on one side of an airplane wing; and a function is using such behavior for a specific task, like flying across the country. This breakdown of a design into its Function-Behavior-Structure elements is a common decomposition in Design Methodology, first proposed by Gero [38].

Although representing behaviors can be done using images described in §2.3.1, we want to consider alternative methods of representation that may be more efficient or discriminative. Doing so could potentially improve model performance across a variety of applications, but to do so, we want to dig deeper into the most important facets of our representations from an engineering point of view. At a high level, we only care that a device performs a given *function* that benefits humanity, *i.e.*, performing boolean logic operations. How this function is achieved is less important than that the function is achieved. For design discovery, if we were able to describe a desired function in words, then we could use text-based approaches. These approaches generally embed text into some vector space, *e.g.*, with Word2vec [39, 40]. This approach has been used by, *e.g.*, Zhang *et al.* [41] to describe potential novel ideas for new products and by Park and Kim [42] to describe desired functions of existing products. However, others, like Ahmed, Fuge, and Gorbunov [43], use a Latent Dirichlet allocation instead to represent topics, rather than individual

words. These works use the cosine similarity metric [44] to compare vectors with each other. Taking inspiration from this, we use the dot product in Ch. 6 in assigning true labels to devices because the magnitude of vectors, not just their direction, is informative in our work. However, while our work does involve grouping devices of different functions, we do not have text descriptions of these functions; instead, we need to look for representations that do not require *a priori* knowledge of the functions we may want to discover. This restricts us to representations based on the devices' behavior or structure.

2.3.3 Behavior Representation: Behavior Vectors

In this section, we present the *behavior vector* (BVec), a lower-dimensional representation of behaviors than the pixel approaches from §2.3.1. When creating BVecs, we attempted to answer three questions:

1. What is behavior, and how do we represent it?
2. How do we avoid non-discriminatory values?
3. How do we minimize redundancy in our BVecs?

As an illustrative example, we discuss a 4-port fluidic device (Fig. 2.12).

First, we need to define and represent device behavior. If we consider that the movement of mass, *e.g.*, fluid velocity, differentiates between fluidic devices, then we recognize that the behavior must be related to the fluid velocity. As such, our method of representing the behavior must take into account the fluid velocity. One naïve way to represent the behavior of a device is to sample the velocity throughout the domain — *e.g.*, along a grid — and append the velocities into

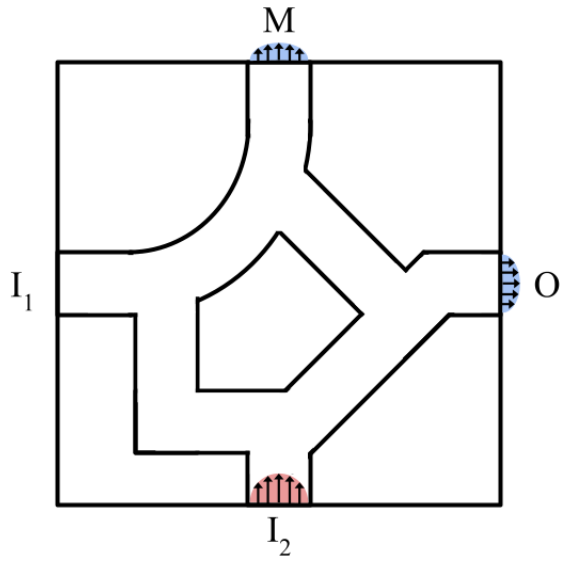


Figure 2.12: Example 4-port fluidic device.

a vector. This provides a large, high-dimensional vector that contains *all* of the information about the behavior. However, this method includes information that may be unnecessary in representing the device's behavior, such as the inside of the domain, which may not be relevant if we only care about the behavior at the ports.

Second, we want to avoid non-discriminatory values in our representation. To understand why the aforementioned representation would be inefficient, consider what a device's behavior means in terms of its function within a larger system. We, as engineers, usually care about how a device affects physical fields on the system boundary. In other words, does it matter if a motor spins because of changing magnetic fields inside the motor or because an elf inside is riding a small bike? Provided that the inputs and outputs of the motor viewed from the outside are identical (*e.g.*, same heat signature, electromagnetic fields, identical impedance matching on the voltage/current input, *etc.*), then not really. Do we need to know what is happening *everywhere*, or can we more compactly — yet comprehensively — represent a device's relationship to its environment by looking only at its interfaces with its environment? Following this line of thought,

we sampled points only along the *boundaries* of a device — *i.e.*, the ports where the fluidic device interfaces with the rest of the system. This approach seems promising as it generates a significantly smaller BVec. However, we also need to generate these BVecs in a consistent and well-defined manner.

To answer the third question, we want to minimize redundancy in our representation. To begin, we look at the sampling resolution along the border. Sampling more points — *i.e.*, a higher resolution — would give more detailed information about the device, but at the cost of a larger BVec. However, it is much easier to prune down a redundant BVec than it is to expand an uninformative BVec. If a device's BVec has slowly changing values, then spatially proximate points would not provide significant additional information about a device, so we can effectively reduce the sampling resolution by removing points from the BVec.

This discussion gives us an approach by which we can convert the continuous results of a simulation into a vector of discrete values on which we can apply ML methods. The sections below go into detail about the implementation of these steps.

2.3.3.1 Initial Boundary Resolution

To choose boundary points to include in the BVec, we consider the edges of the spatial domain. In theory, we can sample any number of points from the simulation data. However, including too many points increases the dimensionality of the clustering problem without increasing our ability to differentiate between clusters. The goal, then, is to include the fewest number of points that still contain enough information to differentiate between different devices.

Choosing the initial sampling resolution is not trivial, though it does correlate strongly

with the length scale of phenomena in the device. If the length scale is unknown, then the initial sampling resolution is little more than an arbitrary guess. As a simple heuristic, we suggest a visual inspection of some data points and choosing a resolution twenty times the frequency of the shortest "important" phenomenon, as in the pedagogical phenomena in Fig. 2.13. Drawing from

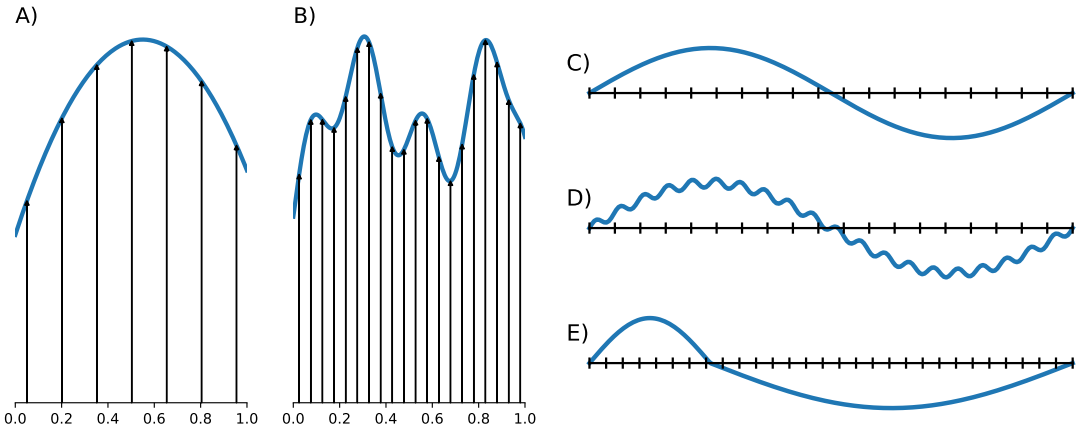


Figure 2.13: A) Pedagogical phenomena with longer length scales and B) shorter length scales use different sampling resolutions. C) Initial sampling resolution scales with the length scale of "important" phenomena, unlike D) where short-length noise is present. E) Using the shortest of varying length phenomenon to determine sampling resolution.

the field of signal processing, such a resolution would be above the analogue of the Nyquist rate and minimize potential aliasing effects. If the length scale is known, we suggest twenty times its minimum frequency as the initial sampling resolution. The specific choice of initial sampling frequency is not critical, so long as it is sufficiently high, since we propose methods to remove redundant dimensions in §2.3.3.2. In this dissertation, we assert that V_{in} BCs will be parabolic, so we use an initial resolution of 20 points per port.

2.3.3.2 Resolution Selection Preprocessing

We want to capture phenomena without knowing ahead of time the length scale of the phenomena. In brief, we estimate the length scale using differences between adjacent points and

use this estimation to determine the sampling resolution. This Resolution Selection preprocessing step shortens a current BVec by taking every m^*-th point, as defined below.

First, we linearly scale our BVecs so each value is between 0 and 1, stratified by dimension. For example, in our fluidic device, all of the values related to pressure are scaled together, all of the values related to x-velocity are scaled together, and all of the values related to y-velocity are scaled together. If we directly scale the entire BVec, then the pressure field, which may have a significantly higher magnitude than the velocity field, could dominate the scaling and essentially negate the discriminatory power of the velocity field.

Next, for every pair of adjacent points, we calculate the absolute value of their difference, which is guaranteed to be less than 1. We take the largest difference — which approximates the most rapidly changing region — and reciprocate this value, giving a number greater than 1. We call the floor of this value m, indicating the possible reduction for this single BVec.

We find the minimum m across our dataset, signifying the fastest-changing phenomena, and call this m^*. We take every m^*-th point from the original BVecs and append these values to form our new BVec. A sample implementation of this process is given in §A.3.

We include two example calculations of this process for a case where BVecs are not and are shortened.

Example 1: $m^* = 1$ Consider the following 3 BVecs.

- $\{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$
- $\{0.6, 0.4, 0.2, 0.0, 0.1, 0.3, 0.5, 0.7, 0.9, 1.0, 0.8\}$
- $\{0.9, 0.8, 0.7, 0.7, 0.6, 0.7, 0.0, 0.0, 0.3, 0.4, 0.3\}$

By taking differences between adjacent points, we get:

- $\{0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1\}$
- $\{0.2, 0.2, 0.2, 0.1, 0.2, 0.2, 0.2, 0.2, 0.1, 0.2\}$
- $\{0.1, 0.1, 0.0, 0.1, 0.1, 0.7, 0.0, 0.3, 0.1, 0.1\}$

Taking the maximum of each difference vector, we get: [0.1, 0.2, 0.7].

Reciprocating these values, we get: [10, 5, 1.43].

Taking the floor of these values, we get: [10, 5, 1]

These values correspond to the m values for our three example BVecs. The minimum of these is m^*. For this example, we would then take every (1-th) point to form the "shortened" Bvecs.

Thus, the BVecs after this preprocessing are:

- $\{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$
- $\{0.6, 0.4, 0.2, 0.0, 0.1, 0.3, 0.5, 0.7, 0.9, 1.0, 0.8\}$
- $\{0.9, 0.8, 0.7, 0.7, 0.6, 0.7, 0.0, 0.0, 0.3, 0.4, 0.3\}$

Example 2: $m^* = 2$ Consider this second example with the following 3 BVecs of length 11. They are similar to the previous example with a minor change in the third vector.

- $\{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$
- $\{0.6, 0.4, 0.2, 0.0, 0.1, 0.3, 0.5, 0.7, 0.9, 1.0, 0.8\}$
- $\{0.9, 0.8, 0.7, 0.7, 0.6, 0.7, \underline{0.2}, 0.0, 0.3, 0.4, 0.3\}$

By taking differences between adjacent points, we get:

- $\{0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1\}$
- $\{0.2, 0.2, 0.2, 0.1, 0.2, 0.2, 0.2, 0.2, 0.1, 0.2\}$
- $\{0.1, 0.1, 0.0, 0.1, 0.1, 0.5, 0.2, 0.3, 0.1, 0.1\}$

Taking the maximum of each difference vector, reciprocating these values, and taking the floor, we get: [10, 5, 2].

The minimum of these values is 2, so for this example, $m^* = 2$. We then take every 2nd point to form the shortened BVecs.

Thus, the BVecs after this preprocessing are:

- $\{0.0, 0.2, 0.4, 0.6, 0.8, 1.0\}$
- $\{0.6, 0.2, 0.1, 0.5, 0.9, 0.8\}$
- $\{0.9, 0.7, 0.6, 0.0, 0.3, 0.3\}$

2.3.4 Boundary Condition Representation

BCs define the physics applied to a given simulation; however, to the author's knowledge, the treatment of these BCs is still an open question [35], though Alguacil also notes the parallels between BCs and padding in Convolutional NNs. For physics-informed networks, some authors encode the BC information in the model [45], while others forgo representing the BCs and instead penalize deviation from them in the loss function [46]. However, our work differs by being agnostic to the type of information. Instead, we can treat BCs similarly to the BVecs described

in §2.3.3 and directly represent BC values in a BVec. The main difference is that we know BC values before the simulation is run, whereas BVec values are only known afterwards. In other words, BCs are a subset of BVecs because both concern values on the boundary of the simulation, but BVecs also include field values that are not predetermined before the simulation is run.

2.3.5 Dimensionality Reduction

In addition to appropriate simulation data representation, we use *dimensionality reduction* methods on our representations in the hopes of countering the *curse of dimensionality* [47]. This so called "curse" is the exponentially growing difficulty as ML problems become higher dimensional. For example, consider data collection in a new ML application. If we say assume that sampling two end points and a mid point of a given dimension is sufficient, then for a 1-D problem, we would need to collect data three times. For a 2-D problem, this becomes nine data collection steps; for 3-D, 27 steps; and as the dimensionality increases, the number of data points needed to "fill" the space increases exponentially. This "curse" is why many ML pipelines include some dimensionality reduction steps before the data is fed into the ML model.

Dimensionality reduction is an unsupervised class of algorithms and generally can be split into two approaches: feature *selection* and feature *extraction*. Feature selection algorithms, as the name implies, select important features from the original data set; our resolution selection heuristic from §2.3.3.2 is a feature selection algorithm. We do not discuss or use any other feature selection algorithms in this dissertation. Feature extraction, on the other hand, projects high-dimensional data into lower-dimensional manifolds while maintaining as much information as possible. Many dimensionality reduction methods exist; here, we discuss three that are used

in this dissertation.

2.3.5.1 Principal Component Analysis

Principal Component Analysis (PCA) [48] is a widely used linear dimensionality reduction method that projects data into a new coordinate system ("principal components") that maximally explains variance in the data in fewer dimensions than the original dataset. To use PCA to reduce dimensionality, we need to calculate the linear transformation matrix; this matrix can multiply later data to project the data into the new coordinate system. Visually, Fig. 2.14 shows the two principal components of a pedagogical data set in red. The majority of the variance is clearly explained in the direction of the longer axis.

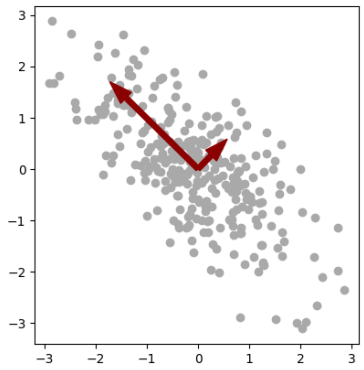


Figure 2.14: Pedagogical example of PCA in 2-D data.

To calculate the transformation matrix, the covariance matrix of the data is first calculated. The covariance matrix is the $m \times m$ matrix, with m being the number of dimensions (*i.e.*, features) the data has, that represents how the dimensions vary with each other. For example, 2-D data would have a 2×2 covariance matrix, while 10-D data would have a 10×10 covariance matrix.

To calculate the (i, j) element in the covariance matrix, we use Eq. 2.43

$$\mathrm{K}_{X_i X_j} = \mathrm{cov}[X_i, X_j] = \mathrm{E}[(X_i - \mathrm{E}[X_i])(X_j - \mathrm{E}[X_j])] \tag{2.43}$$

where K_{X_i,X_j} denotes the (i, j) element in the matrix, X_i is the i-th data point, and E is the expected value function.

We then calculate the eigenvalues and eigenvectors of the covariance matrix, which amounts to solving Eq. 2.44,

$$Kv = \lambda v \tag{2.44}$$

where K is the covariance matrix, v is the eigenvectors, and λ is the eigenvalues. Reordering the eigenvectors by decreasing eigenvalue gives us the principal components in decreasing order of variance explained. In other words, the eigenvectors with the largest eigenvalues project the data into the new coordinate system that maximally explains variance. We can then choose a subset of these eigenvectors to form the transformation matrix for projecting new data. The number of eigenvectors to keep is a hyperparameter that can be tuned, *e.g.*, by one of the methods in §2.1.4, or set based on other work in the field.

In practice, we implement PCA using the Scikit-learn library, as shown in the code snippet below. In this particular example, we keep enough components to explain 95% of the variance in the data.

```
# Assuming an n x m matrix, X, is already defined,
# with n data points and m features.
```

```
from sklearn.decomposition import PCA

model = PCA(n_components = 0.95)

model.fit(X)
```

As new data comes in, it can be transformed using `model.transform(xNew)`.

2.3.5.2 Discrete Fourier Transform

Discrete Fourier transforms (DFT) [49] are a method of reconstructing a given signal using a linear combination of basis functions. For 2-D images, as in Ch. 4, these basis functions are also 2-D images of varying sinusoidal patterns. To calculate the $M \times N$ DFT of a $M \times N$ image X, we can use Eq. 2.45.

$$\begin{aligned} \text{DFT}(X_{h,w}) = \frac{1}{\sqrt{MN}} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} X_{m,n} \exp\left(-2\pi i \left(\frac{mh}{M} + \frac{nw}{N}\right)\right) \\ \forall h \in \{0, 1, ..., M-1\}, \forall w \in \{0, 1, ..., N-1\} \end{aligned} \tag{2.45}$$

However, this gives a representation that is still $M \times N$. To reduce this, we base our DFT dimensionality reduction on [50] and truncate the borders of the representation, as shown in Fig. 2.15. The degree to which the representation is truncated is a hyperparameter of this method. In Ch. 4, we set this truncation such that the *mean percentage error* (MPE) of the reconstruction of the original image, which uses the *mean absolute error* (MAE) between the reconstructed and original image (Eq. 2.46), is less than 5% (Eq. 2.47).

$$\text{MAE}(x, \hat{x}) = \text{mean}(|x - \hat{x}|) \tag{2.46}$$

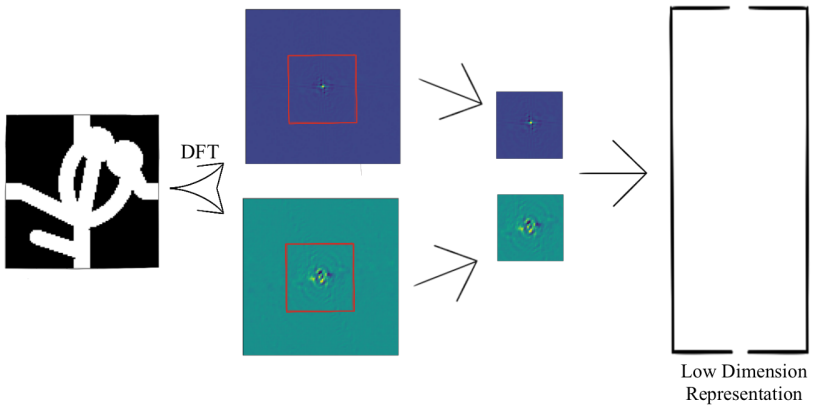


Figure 2.15: Truncating a DFT to reduce dimensionality.

$$\mathrm{MPE}(x, \hat{x}) = \frac{\mathrm{MAE}(x, \hat{x})}{\max(|x|)} \tag{2.47}$$

The level of truncation is set through the training data, and the same level is used for the testing data, to avoid data leakage.

2.3.5.3 Autoencoders

Autoencoders (AE) [51] are a class of unsupervised NNs made up of two parts: an *encoder* and a *decoder*. The encoder reduces the input dimensionality, encoding its information into a low-dimensional *latent space*. This is paired with a decoder, which expands the latent representation into the original input shape. The encoder and decoder usually have identical but reversed architectures; in other words, if the encoder has HLs of size (6, 3, 2), then the decoder would have HLs of size (2, 3, 6).

Generally, AEs use reconstruction loss — *i.e.*, the error incurred by this encoding-decoding process, often a variant of MSE — as an objective function, which is minimized using backpropagation and a gradient descent solver. This method is commonly used for unsupervised dimen-

sionality reduction (*e.g.*, [52, 53, 54]) at the cost of an expensive training process.

We use an AEs with architectures drawn from [53] and [54] in Ch. 4 to encode geometry information. We give an example implementation in §A.4. An interesting, if unintuitive, note is that the inputs and outputs of an AE are identical since the AE is intended to reconstruct the input data as closely as possible.

2.4 Summary of Common Terms and Abbreviations

Throughout this dissertation, we use a variety of abbreviations and terminology that are specific here. Although they have been defined in their respective sections, we aggregate some of the common ones here for convenience. We split this section into two parts: quantities known *Before Simulation* and *After Simulation*.

2.4.1 Before Simulation

The following quantities are known before a simulation is run:

Dirichlet boundary conditions (DBC, BC)	Values prescribed on the SF that must be met. In Fig. 2.16, these are blue arrows and red lines, along with the walls of the simulation, and are also denoted as functions of the x-y coordinates.
geometry (geometry representation, Geo. Rep.)	The domain on which the simulation will be run; in this dissertation, the region in which there is fluid. This correlates to the gray portions of Fig. 2.16.
simulation parameters	Values related to the simulation setup. In Ch. 4, these are the mesh cell size and the Lipschitz number of the BCs.

Table 2.1: Known SF quantities before simulation.

Location	Velocity	Pressure
Left Port	✓	
Bottom Port		✓
Right Port		✓
Top Port	✓	
Walls	✓	
Everywhere Else		

Saving BCs is similar to saving BVecs (see Fig. 2.18) wherein field values along the borders are saved. However, only known field values are saved; unknown values are either ignored — which could potentially cause varying size representations if different BCs are applied — or zero-padded — at the cost of additional dimensionality.

2.4.2 After Simulation

After a simulation is run, we know the additional following quantities:

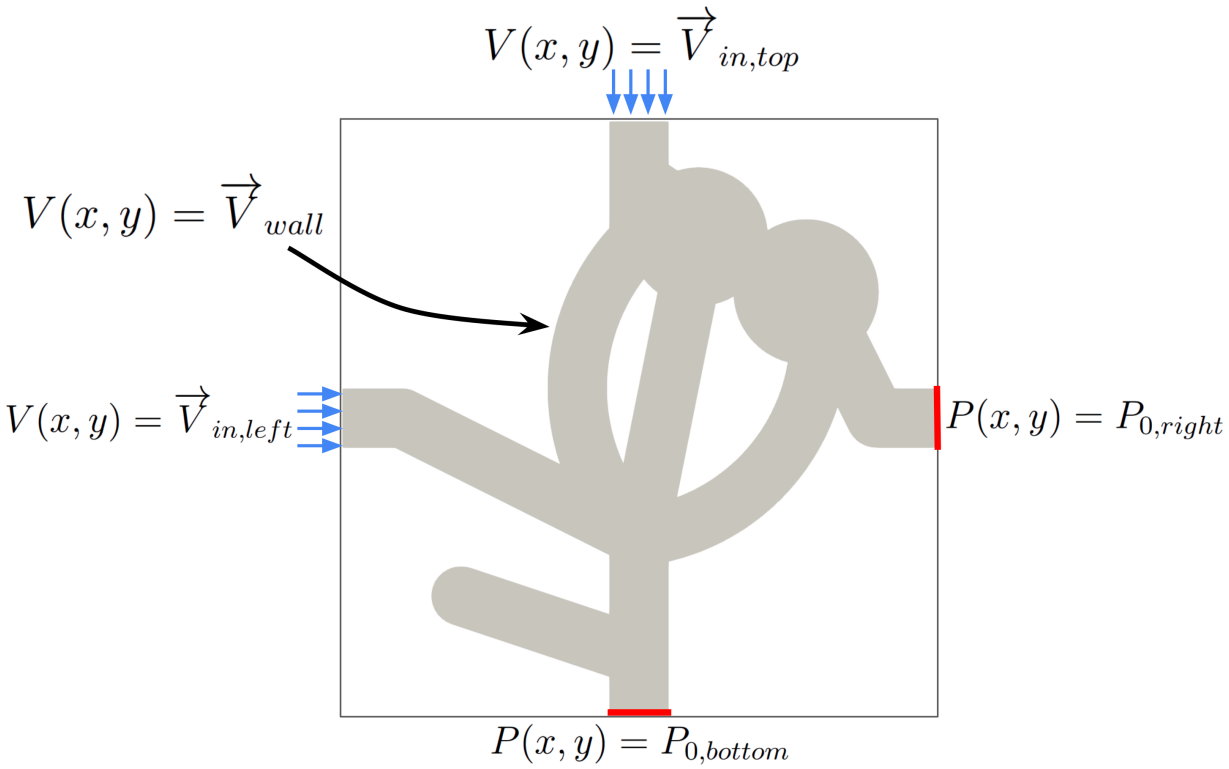


Figure 2.16: Before simulation, we know simulation parameters and functions for BCs.

solution field (SF)	The numerical results of the simulation. Generally, some function of the coordinates. Multiple can exist, and can be scalar, vector, or tensor, depending on the physics. Additional QoI (*e.g.*, drag from pressure SFs in a fluid simulation) can be calculated from these. This correlates to the color portions of Fig. 2.17.

As we can see in Fig. 2.17, we can save the values in a SF to a matrix for use in ML pipelines. This method is applied in this dissertation to Stokes and Navier-Stokes simulations but is applicable to any type of simulation that produces data that can be queried at arbitrary points within the domain. Of note, we use Navier-Stokes simulations in Ch. 5. The Navier-Stokes

simulation results we use have a time component that we ignore in this approach; that is, we look at only the last time step of our simulation results. Our method below of saving simulation data to matrices can be extended to higher-dimensional matrices (*i.e.*, tensors) if additional time steps are needed, or if more SF dimensions are to be saved. Here, we have used a 3-D matrix.

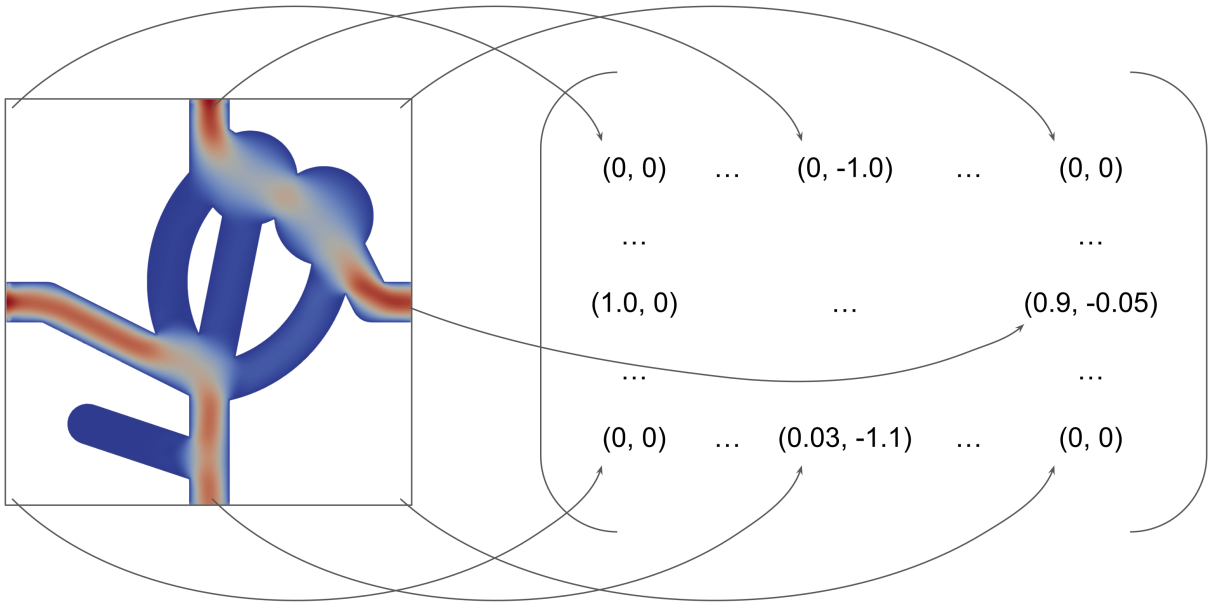


Figure 2.17: After simulation, we know all SF values anywhere in the domain. We can save, *e.g.*, velocity as a $k \times k \times 2$ matrix.

From the SFs, we calculate the following quantities in this dissertation:

behavior vector (BVec)	Low-dimensional representations of simulation data. These are made by appending SF values along edges, specifically ports in this dissertation, as shown in Fig. 2.18. This is used in Ch. 6 as representations of device behavior.
pressure field magnitude	Magnitude of the vector that represents the pressure field values in a fluid simulation. This is used in Ch. 4 as a surrogate for physical realizability.

Table 2.2: Known SF quantities after simulation.

Location	Velocity	Pressure
Left Port	✓	✓
Bottom Port	✓	✓
Right Port	✓	✓
Top Port	✓	✓
Walls	✓	✓
Everywhere Else	✓	✓

For a more comprehensive list of terms defined in this dissertation, we refer readers to the Glossary on pg. viii.

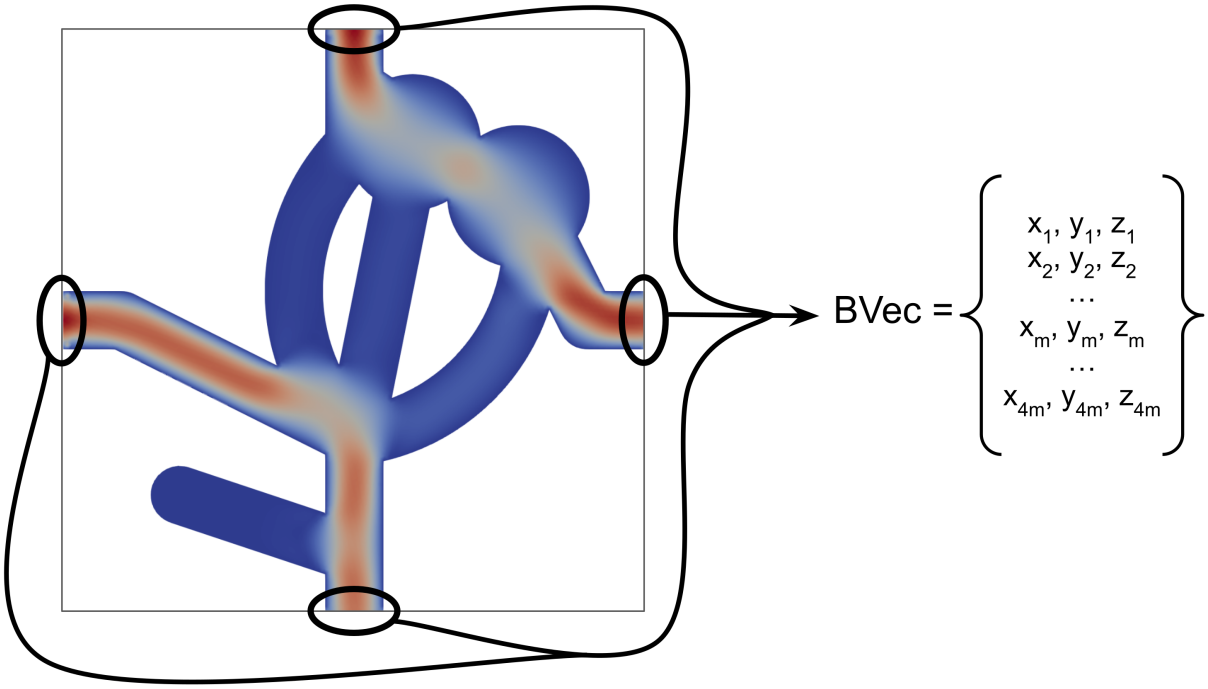


Figure 2.18: To make BVecs, we assume each port has m points along it and save the SF values along each port in a matrix.

Chapter 3: Type-Based Indexing in Finite Element Simulations

This chapter initially set out to explore the first half of the question posed in the introduction: **How do we enumerate possible boundary conditions for a given physical law** that can lead to well-defined solutions to a given partial differential equation? During the exploration of this question, we created an automated method of checking that a given set of boundary conditions (BC) and a given physical law are compatible through a type-based indexing scheme. We also delineate two properties of that scheme that can generate valid Finite Element Method (FEM) formulations, resulting in a three-fold increase in the number of simulations generated from a limited set of BCs, providing a stepping stone toward BC enumeration.

This work was published and presented in the American Society of Mechanical Engineers (ASME) 2019 International Design Engineering Technical Conferences & Computers and Information in Engineering (IDETC/CIE) Conference in August 2019.

3.1 Introduction

Modern advances in computation provide pathways for data-intensive processes in mechanical design. For example, current state of the art approaches to engineering analysis, topological optimization, generative design, and ML can benefit from large amounts of training data or methods to generate that data. Generating this data manually — *e.g.*, by having an employee manually

run various test cases — is infeasible due to the large quantity needed, so a method to automate the generation of such data is crucial to these data-driven methods. To acquire this data, we wish to automate the simulation of physical phenomena using the FEM to solve the partial differential equations (PDE) that govern many physical phenomena. From these simulations, we can calculate, *e.g.*, forces acting on objects from fluid stress fields or differences in electrostatic potential. To build these simulations manually is time-consuming and impractical for the large number of simulations needed in ML model training, so this chapter proposes a method to check that an automatically generated simulation will run given Dirichlet BCs (DBC) that satisfy certain properties.

As an illustrative example, consider a fluid flowing through a pipe in the presence of an electrostatic field (*e.g.*, as is common in biological microfluidic devices) shown in Fig. 3.1. The fluid in the pipe is governed by the Navier-Stokes equations, whereas the electrostatic field is governed by the Poisson equation. If the fluid and electric field are not coupled, then two simulations can be run separately. However, if they are coupled — that is, the electrostatic field affects the fluid flow while the fluid flow affects the electrostatic field — then both fluid and electrostatic BCs are needed to run this coupled simulation. If not enough BCs are provided, the coupled simulation cannot run, but some subset of uncoupled simulations may.

From a slightly different direction, the equations that govern our fluid in an electrostatic field require the interaction of three vectors: fluid velocity, fluid pressure, and electrostatic potential. Each of these vectors needs a mathematical construct, which we call the *solution field* (SF), that contains the vectors' values. Additionally, each SF needs an "anchor" value against which quantities (especially relative quantities, *e.g.*, pressure, electrostatic potential, *etc.*) can be measured; thus, each SF requires a BC that sets some part of the SF to a prescribed value.

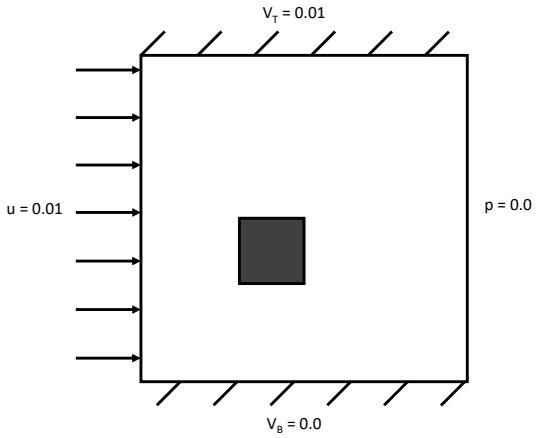


Figure 3.1: Example problem of fluid in a pipe in an electrostatic field.

If an algorithm wanted to automatically construct each of the possible subsets of simulations in this example, it would need to overcome several crucial problems:

- Choosing an equation that can be fully defined from the potential BCs, defining appropriate SFs to solve the simulation, and applying the correct BCs to ensure convergence of the solution.

- Guaranteeing the viability of a simulation given the BCs, SFs, and variational form (VF).

- Ensuring robustness of simulations to permutations in VF and BCs — *i.e.*, so long as there is sufficient information to construct the simulation, the order of that information should not matter.

A human engineer or scientist implicitly overcomes many of these problem (*e.g.*, we choose to put 2-D fluid velocity into 2-D matrices, not 1-D vectors); we are not currently aware of any work that formalizes this intuition.

Specifically, the key contributions of this chapter are:

- We propose the use of type-based indexing to generate appropriate SFs from BCs without

knowing ahead of time the types (scalar, vector, or tensor) or dimensions of the BCs. This contrasts previous implementations where we have prior knowledge of the BC types and manually build SFs based on that knowledge.

- We discuss properties of *simulation sets* necessary to run FEM simulations, labeling them as what we call *complete*, *partially complete*, and *incomplete* simulations, and show that these labels can be generated automatically. Under certain conditions, these labels guarantee simulation viability.

- We apply type-based indexing in VFs to improve robustness of simulations to ordering permutations of SFs, BCs, and VF terms. This gives us the ability to construct simulations more robustly than previous implementations in which, *e.g.*, the order of BCs can determine whether a simulation runs or not.

To demonstrate our above contributions, we use the FEniCS library [1] to implement the FEM; however, these contributions are applicable to other implementations of the FEM as well. We test our contributions on three types of equations, as discussed in detail in §2.2.3. These are the Navier-Stokes equations, Poisson equation, and a synthetic equation we formulate specifically to stress-test our contributions. Notably, the synthetic equation uses fluid velocity, fluid pressure, and Poisson terms; however, it has no physical interpretation.

3.2 Related Work

Past approaches to automating the construction of FEM simulations falls roughly into two camps: automating mesh generation and languages for general purpose, multi-physics solvers.

Much of the previous work in automating FEM focuses on the mesh generation process. For example, [55] and [56] describe basic methods of automating the meshing process, leading to a variety of applications, *e.g.*, from bio-medical [57] to environmental [58]. Such approaches have led to major advances in how to discretize FEM problems, assuming that the VF of the governing equation and BCs are known. This work is important to the field but orthogonal to the contributions of this chapter, which focuses on assembling appropriate BCs, SFs, and a VF. As such, we assume an appropriate quality mesh is provided or can be computed using past methods because our concerns lie with guaranteeing simulation viability and setting up the FEM problem definition.

More interesting to this chapter are programmatic languages for writing and simulating PDEs using FEM. Specifically, our work extends the work of [59] and [60], which laid the framework for the FEniCS library [1] — a programmatic language for writing and simulating FEM solutions to PDEs, such as the Unified Form Language [61]. Some similar packages, such as SfePy [62], also provide an application programming interface (API) for implementing complex FEM simulations, acting either as a black-box solver or a framework for custom implementations. While such approaches have helped standardize how one can describe and implement various types of single and multi-physics analysis (*e.g.*, [63]), they still require researchers to write customized code for implementing specific physics, particularly in coupled models. The closest work to this chapter is the work of Xia [64], who implements a multi-physics solver that can generate FEniCS code automatically by assuming that BCs exist for every portion of the domain. Our approach differs from [64] in that we make no assumptions regarding the types or number of BCs given, and can automatically generate and check for multiple subsets of physics models that are consistent with the BCs.

3.3 Proposed Implementation of Type-based Indexing

In most FEM implementations, one knows the type of physics or equation being solved ahead of time and, consequently, can build SFs that match the VF exactly. However, this approach is limited in that each code is specific to a single type of physics; to simulate a different type of physics, an entirely new simulation must be coded from scratch. The goal of this chapter is to develop a generalizable system that can check that multiple single- and multi-physics FEM simulations can run without the need of checking each simulation manually, with the eventual goal of generating large datasets of simulations completely automatically.

To this end, we propose the use of type-based indexing to derive the corresponding SFs from a list of BCs and to map SFs to the corresponding variables in VFs. This also facilitates robustness to the order of terms in the *VF*.

3.3.1 Type-Based Indexing for Boundary Conditions

As we discussed in §2.2.2, a valid BC requires a value and a location. In addition, we add a *type* to each BC object that we manually define. For this chapter, these types include fluid velocity, fluid pressure, and Poisson. We present one simple possible implementation in §A.5.

3.3.2 Type-Based Indexing for Solution Fields

We propose type-based indexing to mitigate the disadvantages of name-based or number-based indexing. While this approach has some overhead in encoding more information into the VF, it provides an unambiguous label for each unique type of BC and removes the issue of ordering the SFs as they are referred to only by their type, rather than a name or index.

To build the SFs, we can use an implementation like in Algorithm 3. This algorithm takes a

Algorithm 3 Builder

```
1:  Initialize dictionary of SFs
2:  for each BC do
3:      Add BC to SF Dictionary with the key of type(BC)
4:  end for
5:  Initialize empty mixed finite element
6:  for each key in the SF dictionary do
7:      Add the dimensioned element to the mixed finite element
8:  end for
9:  Build the function space from the mixed finite element
10: for each key in the SF dictionary do
11:     for for each BC with a type that matches the key do
12:         Create library's BC object on the corresponding SF and append to list of BCs
13:     end for
14: end for
15: Return function space and BC objects
```

list of BCs as input. It stores each unique type of BC in a dictionary since it only matters that each *type* has a SF. Once we have extracted the types, we add appropriate elements to a mixed element placeholder. Each of the added elements has dimensions according to the type it represents (*e.g.*, fluid pressure fields are scalars and use scalar elements, whereas fluid velocity fields in 2-D have 2-D vector elements). These dimensions can be derived from the value of each BC. With this mixed element, the FEniCS library can build a function space that contains SFs corresponding to the different elements in the mixed element placeholder. Finally, the algorithm creates BC objects, which require information about the SF to be used, value, and location of each BC. The mixed function space and list of BCs are returned. The FEniCS library has specific objects that represent BCs, but any implementation-specific object or function to apply BCs on the stiffness matrix can be used.

We would like to draw special attention to lines 3 and 11. In line 3, we use a key that, rather than a number or a name, is the *type* of BC being applied. This ensures that the function mapping

from the BCs to the SFs is surjective, thus meeting one of the criteria for a complete simulation set from §3.3.4.

In line 11, we again refer to a dictionary key that is the *type* of a BC. Applying each BC to the corresponding SF is fairly straightforward: we find the SF whose key matches the BC's type and apply the BC onto that SF.

Also of note are lines 6 and 10, where we iterate over the keys matching each type of BC, thus ensuring BCs of the same types are kept on the same SF and those of different types are separated.

3.3.3 Type-Based Indexing for Variational Forms

Similar to the SFs, we add some additional information to the VFs to encode the *needed types* into the abstraction. Although setting the VF abstractions does have some overhead cost, this process is only needed once for each type of simulation. Including the needed types additionally provides the information needed for completeness labeling in §3.3.4. We show a sample implementation for the Navier-Stokes equation in §A.6.

3.3.4 Completeness of Simulation Sets

Each simulation requires what we call a *simulation set* of BCs, SFs, and a VF. To solve a PDE, BCs are applied on SFs, whose values are manipulated to satisfy the VF. Because of these interwoven relations, a viable simulation has a simulation set with two important properties between its different members.

First, we define a *viable* simulation set as a simulation set whose members fully define a

simulation that: (1) runs to completion (*e.g.*, no shape mismatches or undefined terms) and (2) converges to a solution (*e.g.*, not a singular matrix, no infinite or NAN values).

In this section, we assume that, for a given VF, the SFs in the simulation set match the VF exactly; we discuss generating these SFs in §3.3.2. We define a simulation set with at least one BC for every variable type in the VF without extra as *complete*; if there are unused/extra BCs, that simulation set is called *partially complete*, and extra BCs can be trimmed off to make a complete simulation set (see §3.3.6. In both of these cases, the simulation set can define a viable simulation. If the VF contains variables whose types do not have corresponding BCs, then that VF is considered *incomplete*, leading to a simulation set that cannot run. With this definition, we can decide whether a simulation set is viable or not by making several observations about its members and their relationships to each other.

3.3.4.1 Solution Fields and Variational Forms

For a viable simulation set, the first condition is that SFs and VFs must have a one-to-one correlation. That is, every unique term in the VF must have exactly one SF that relates to it, and every SF must be used in the VF.

To see why a VF must use all SFs, consider an unused SF. Because the SF is unused, then the coefficients of the corresponding rows in the stiffness matrix are 0. Because at least one row of the stiffness matrix is all 0s, the stiffness matrix is singular and cannot be inverted, thus leading to an unsolvable problem (see §2.2 for FEM and solving linear systems).

To see that every unique term in the VF must have an SF associated to it, consider a term in the VF that does not have a SF. Since there is a variable/mathematical construct missing from

the equation, the equation cannot be solved numerically.

In a third case, multiple SFs could be associated to a single VF term; for example, we will consider two Poisson SFs for a single Poisson VF. Since both SFs are effectively identical, then the same BCs must be applied to both, and the same governing PDE must hold true across both SFs. To ensure this is the case, the coefficients of the corresponding rows of both SFs must be identical, which in turn makes the stiffness matrix singular and non-invertible.

The only way for the stiffness matrix not to be singular is for each SF to correspond to a unique term in the VF, and for every unique term in the VF to have a corresponding SF.

Intuitively, this is analogous to solving multiple equations with multiple unknowns quantities. Each quantity must have its own variable (*i.e.*, SF), and having more variables than equations renders the probleum unsolvable.

3.3.4.2 Boundary Conditions and Solution Fields

The second condition of a viable simulation set is that the each SF must have at least one BC applied, and every BC is applied once. It is fairly straightforward that any unused BC can be removed with no impact on the simulation results. By [65], having a SF with no applied BCs results in a singular stiff matrix, rendering the simulation unsolvable. Since we only are looking at simulations that are solvable, then each SF must have at least one BC applied. Although other types of BCs (Neumann, Robin, etc) may not match this criterion exactly, for DBCs, this condition must hold.

3.3.5 Labeling Simulation Sets for Completeness

After applying type-based indexing for the BCs, SFs, and VFs, we now have a more robust method of connecting various parts of a simulation set together. Labeling for completeness is simplified with our type-based indexing and consists effectively of checking whether the list of required types is a subset of the given BC types and vice versa. The code snippet below shows one sample implementation of completeness labeling. Note that checking if two sets are subsets of each other is equivalent to checking that two sets are equivalent.

```
# Assuming that the types of BCs are in a list BCTypes,
# and that the types in the VF are in a list VFTypes


# Testing for completeness
completeTest = set(VFTypes).issubset(BCTypes) and
               set(BCTypes).issubset(VFTypes)


# Testing for partial completeness
partCompTest = set(VFTypes).issubset(BCTypes) and
               not set(BCTypes).issubset(VFTypes)


# Testing for incompleteness
incompTest = not (completeTest or partCompTest)
```

`completeTest` is `True` when the simulation set is complete, and `partCompTest` is `True` when the simulation set is partially complete. If neither is true, then the simulation set must be

incomplete.

3.3.6 Selection of Applicable Boundary Conditions for Viable Simulation Set Generation

By using the method in §3.3.5, we can label a simulation set with its completeness. However, partially complete simulation sets still include BCs that are irrelevant to a given VF; these extra BCs should be removed before the simulation is run. We implement this in Algorithm 4, which returns the set of BCs that are applicable to the simulation set. This algorithm takes in a

Algorithm 4 Selector

```
1:  if The given simulation set is incomplete. then
2:      The simulation set in not viable. Do not run a simulation.
3:  end if
4:  if The given simulation set is complete. then
5:      Nothing to prune. Return the BCs as-is.
6:  end if
7:  if The given simulation set is partially complete. then
8:      for each BC do
9:          if The VF does not use the BC then
10:             Remove the BC
11:         end if
12:     end for
13:     Return the remaining BCs
14: end if
```

set of BCs and a given VF and returns the pruned set of BCs. The resulting simulation set with the pruned BCs is thus either complete or known to be incomplete.

3.4 Experimental Setup

In this chapter, we assume that only physically realizable BCs are given as inputs. For example, fluid flowing through a pipe can have one V_{in} BC and one P_0 BC. In contrast, if both ends

mandate velocities leaving the domain with no inputs, then fluid must be generated somewhere within the pipe to conserve mass (*i.e.*, maintain a divergence of 0). If there is no source of fluid, then the simulation will break due to non-physicality, not necessarily because the FEM cannot "solve" the PDE. In other words, solutions might be mathematically possible but not physically realizable. Physical realizability of BCs is explored more in Ch. 4.

Throughout the rest of this chapter, we use our example of a fluid flowing through a pipe in an electrostatic potential field to demonstrate the core contributions, as seen in Fig. 3.1. This example provides us with two different types of physics — governed individually by the Navier-Stokes and Poisson equations — and with a potential coupled example, governed by a coupled form in [66]. For the purposes of this chapter, we do not consider the fully coupled form of the equations; instead, we choose a "synthetic" VF that requires the same types of BCs and SFs as the fully coupled form to make computation simpler and faster to replicate.

Specifically, we use the following three VFs:

- Quasi-static Incompressible Navier-Stokes (Pressure, Velocity)

$$F = \langle u, v \rangle + \langle \nabla u, \nabla v \rangle + \langle \nabla p, v \rangle + \langle \nabla \cdot u, q \rangle \tag{3.1}$$

- Poisson

$$F = \langle \nabla u, \nabla v \rangle - f \cdot v - g \cdot v \tag{3.2}$$

- "Synthetic" (Pressure, Velocity, Poisson)

$$F = \langle u, v \rangle + \langle p, q \rangle + \langle s, t \rangle + f \cdot t + g \cdot t \tag{3.3}$$

In the Navier-Stokes equations, u and p are the velocity and pressure, respectively, with v and q the respective test functions. In the Poisson equation, u is the variable of interest with v as the test function. In the Synthetic VF, u, p, and s are velocity, pressure, and Poisson terms, respectively, with v, q, and t the respective test functions. In both the Poisson and Synthetic VFs, f and g are internal and boundary source terms, respectively.

While the Navier-Stokes and Poisson VF are fairly standard, the "Synthetic" VF we use is constructed for the purposes of this chapter. This form has no physical meaning; rather, it is used merely as a test equation that requires multiple and different BCs and combines a pressure field, a velocity field, and a Poisson field. The only criteria of the Synthetic VF are that it requires different BCs (and subsequently SFs) and that it converges to a solution, regardless of the physical interpretation of that solution.

We also use three types of BCs: fluid pressure, fluid velocity, and Poisson. We separate them into the following sets for our experiments:

0. $\{\}$
1. $\{$Velocity L$\}$
2. $\{$Pressure R$\}$
3. $\{$Poisson T$\}$
4. $\{$Poisson T, Poisson B$\}$
5. $\{$Poisson T, Pressure R$\}$
6. $\{$Velocity L, Pressure R, Poisson T$\}$

7. {Velocity T, Velocity L, Poisson B}

8. {Velocity B, Velocity T, Velocity L, Pressure R}

9. {Velocity L, Pressure R, Poisson T, Poisson B}

10. {Velocity B, Velocity T, Velocity L, Pressure R, Poisson T}

11. {Velocity B, Velocity T, Velocity L, Poisson T, Poisson B}

12. {Velocity B, Velocity T, Velocity L, Pressure R, Poisson T, Poisson B}

Here, "B", "T", "L", and "R" refer to the bottom, top, left, and right edges of the domain, respectively, and as seen in Fig. 3.1. Velocity L is a small, positive, uniform, horizontal flow, similar to V_{in}, while Velocity B and Velocity T are both V_0, the "no-slip" condition from §2.2.2. Pressure R is a P_0 BC. Poisson T and Poisson B are uniform positive and zero values, respectively.

These sets of BCs are heuristically chosen by the authors to maximize diversity in the following traits in the aforementioned sample problem:

- Number of BCs — how many BCs are applied
- Types of BCs — fluid velocity, fluid pressure, and Poisson
- Combinations of types — *e.g.*, fluid velocity with Poisson, fluid pressure by itself, *etc.*

Although many other possible test problems are possible, we combinatorically choose among these sets to simplify testing in our experiments. While a more exhaustive test set would include all $2^6 = 64$ possible combinations of 6 BCs, many of these tests would be redundant for testing our approach (*e.g.*, two sets of BCs where the only difference between them is removing one V_0 and adding in the other).

This chapter refers only to simulations with DBCs. While Neumann, Robin, and other types of BCs are important, they are not considered in this work due to the differences in their implementation in FEM, specifically because they are added as terms in the VF. In this work, we assume the VF is already given. It is expected that this chapter is compatible with Neumann and Robin BCs with minimal conceptual changes, though the specific implementation may differ if there are no DBCs acting on the same fields.

Our experiment in §3.5 tests different methods of generating SFs. We use the same sets of BCs in §3.6 to explore the accuracy with which we can automatically label a simulation set given the VF of the governing equation and set of BCs while assuming the associated SFs are correct. With the now fully defined and complete simulation sets, we use the VFs from above and the same sets of BCs to build and run viable simulations to test the implementation of our approach compared to standard methods in §3.7.

3.5 Experiment 1: Generating Solution Fields from Boundary Conditions

In this section, we test if we can generate SFs from BCs without intervention. We compare various baseline methods of automated SF generation against our type-based method from §3.3.2.

3.5.1 Baseline Methods for Solution Field Generation

Arbitrary The most naïve approach to generate SFs is to initialize some arbitrarily large number of SFs of multiple dimensions and types (scalar, vector, and tensor). This approach assumes the BCs provide no information about the required SFs. However, this is computationally expensive and makes no guarantees on the viability of the simulation. In essence, this can produce too

few or too many SFs, and we cannot know which without more information.

Minimal-Maximum As a small improvement on this naïve approach, we can use the number of BCs as an upper bound to the number of SFs. In other words, if there are N BCs, we would initialize N scalar fields, N vector fields, and N tensor fields. This would, of course, be many more SFs than are needed.

Unique BC Rather than gathering no information from the BCs, we can assign each BC to its own appropriately dimensioned SF (*e.g.*, a 1-D BC to a scalar SF). This imposes an upper limit on the number of SFs that can possibly be required in a specified VF because each SF requires at least one BC for convergence. However, this approach does not link BCs of the same type together; *e.g.*, if two fluid velocity BCs are given for a two-inlet pipe system, then two distinct fluid velocity SFs would be generated, leading to ambiguity as to which fluid velocity SF the VF "should" use.

Unique Dimension A third approach is to build a SF for each BC of a different dimension. For example, a fluid simulation would have pressure BCs and velocity BCs. Because pressure BCs affect scalar SFs and velocity BCs affect 2-D vector SFs, then one scalar SF and one 2-D vector SF would be created. Unfortunately, this approach breaks down when different BCs require SFs of the same dimension, *e.g.*, an electrostatics-coupled Navier-Stokes simulation [66]. In this case, electrostatic potential is a scalar quantity and requires a scalar SF, and while pressure is also a scalar quantity and requires a scalar SF, electrostatic potential and pressure should not be treated as the same variable. This approach gives a lower bound on the number of SFs required because BCs of different dimensions must have distinct SFs.

Unique Names Instead of extracting information from the BCs, we can try to derive necessary SFs from the unique variable names in the VF. However, each variable's name in a given VF is somewhat arbitrary, and a simple difference in convention (*e.g.*, ϕ, electrostatic potential, vs. u, temperature in the heat equation, though both are applications of the Poisson equation) should not necessarily imply a different simulation type entirely (*e.g.*, a general Poisson simulation vs. electrostatic potential or heat simulations), only a different physical interpretation of the solution. Additionally, while there is some sort of convention for naming variables, ordering the SFs that correspond to those variables is loosely alphabetical at best and entirely random at worst. This ambiguity with both naming and ordering can lead to uncertainties in applying BCs to the correct SFs.

Indexed Names A slight modification to the name-based indexing stated above replaces names with numerical indices. For example, a Navier-Stokes simulation, which requires two SFs, could assign, *e.g.*, velocity to SF_0 and pressure to SF_1. The VF then would also use SF_0 for velocity terms and SF_1 for pressure terms. Numerical indices are easily extensible to an (almost) arbitrarily large number of SFs, but the same issue arises with ordering the SFs in some unambiguous manner, *e.g.*, the indices of the velocity and pressure SFs should not change the viability of the simulation set. Table 3.1 shows a potential case where the ordering of the BCs affects the SF references.

3.5.2 Results

Table 3.2 compares our type-based approach with the baseline methods using the sets of BCs discussed in §3.4. For every ordering permutation of the BCs in each given set of BCs, we

Table 3.1: Different orderings of BCs can result in inconsistently referenced SFs, even if the same type of simulation is run.

BCs Applied	SF_0 Expected	SF_0 Actual	SF_1 Expected	SF_1 Actual
$\{V_0, P_0\}$	2-D Vector	2-D Vector	Scalar	Scalar
$\{V_0, V_{in}, P_0\}$	2-D Vector	2-D Vector	Scalar	Scalar
$\{V_0, P_0, V_{in}\}$	2-D Vector	2-D Vector	Scalar	Scalar
$\{P_0, V_0\}$	2-D Vector	Scalar	Scalar	2-D Vector

Table 3.2: Ratios of correctly generated SFs. Dashes indicate no successes.

BC Set	Arbitrary	Min-Max	Unique BC	Unique Dimension	Unique Names	Indexed Names	Types
0	-	1.0	1.0	1.0	1.0	1.0	1.0
1	-	-	1.0	1.0	1.0	1.0	1.0
2	-	-	1.0	1.0	1.0	1.0	1.0
3	-	-	1.0	1.0	1.0	1.0	1.0
4	-	-	-	1.0	1.0	1.0	1.0
5	-	-	1.0	-	0.5	0.5	1.0
6	-	-	1.0	-	0.167	0.167	1.0
7	-	-	-	1.0	0.333	0.333	1.0
8	-	-	-	1.0	0.75	0.75	1.0
9	-	-	-	-	0.083	0.083	1.0
10	-	-	-	-	0.3	0.3	1.0
11	-	-	-	1.0	0.3	0.3	1.0
12	-	-	-	-	0.0083	0.0083	1.0
Average	0.0	0.077	0.462	0.615	0.572	0.572	1.0
Generalizable	✓	✓	✓	✓	-	✓	✓

use each baseline and the proposed approach to generate SFs and compare the generated SFs to those we generate manually. A correctly generated set of SFs would be ones where the algorithm builds the correct number of SFs of the correct size (scalar, vector, or tensor) and dimensionality. We report the ratio of correctly generated SFs in Table 3.2. Additionally, we note whether each method is generalizable, or extendable to different VFs without additional adjustments.

We see that as we move to more sophisticated methods, we see an increase in the number of BCs that may have correctly generated SFs. We also note that every method is generalizable except method "Unique Names," which uses names of variables to differentiate fields. Even so, methods "Unique Names," "Indexed Names," and "Types" all generate at least some correct SFs. With the switch to indexed names, method "Indexed Names" seems to generalize method "Unique Names." However, these two methods perform poorly as the possible number of order-

Table 3.3: Number of labels based on VF and ground truth labels.

	Poisson	Navier-Stokes	Synthetic
Incomplete	4	8	9
Part. Complete	7	4	0
Complete	2	1	4
# Correct	**13**	**13**	**13**
Total	13	13	13

ings increases.

The "Unique BC" and "Unique Dimension" approaches correctly generate SFs regardless of BC ordering, but in general, these approaches are not robust to the combinations of BCs that may be encountered. While the "Unique Names" and "Indexed Names" approaches generated the correct SFs at least sometimes in all of our test cases, Table 3.2 shows they lack of robustness when the BCs are ordered differently, especially as the number and types of BCs increase. Our type-indexed approach "Types" provides the correct references to the corresponding SFs regardless of the order of the BCs in all of our test cases.

3.6 Experiment 2: Labeling Completeness

To ensure completeness labels can be generated automatically, we compare automatically generated completeness labels with the ground truth labels manually assigned by the author. We take each set of BCs from §3.4 and generate its corresponding SFs, as in §3.5. This set of BCs and SFs is then combined with each of the VFs from §3.4 to form a simulation set, whose completeness is calculated. As can be seen from Table 3.3, our approach correctly labels the simulation sets for every test case we attempted, regardless of the type of VF or its completeness. That there are no partially complete Synthetic tests is due to the Synthetic VF requiring all three

possible types of BCs. The total number of tests matches the number of unique BC sets.

3.7 Experiment 3: Completeness Labels and Simulation Viability

In this section, we test if our type-based and completeness labels are indicative of simulation viability. Specifically, we want to test if applying our completeness labels guarantees simulation viability. Figure 3.2 shows the overall pipeline of our methods. We begin with the BCs and a potential VF, forming an initial simulation set. This simulation set is labeled with the method in §3.3.5; at this point, incomplete simulation sets are stopped from running. Partially complete simulation sets have extra BCs pruned by Algorithm 4, after which they become complete. The complete simulation sets are passed into Algorithm 3, which builds the appropriate SFs. Finally, the simulation set and an appropriate mesh are ready to be fed into a FEM simulator.

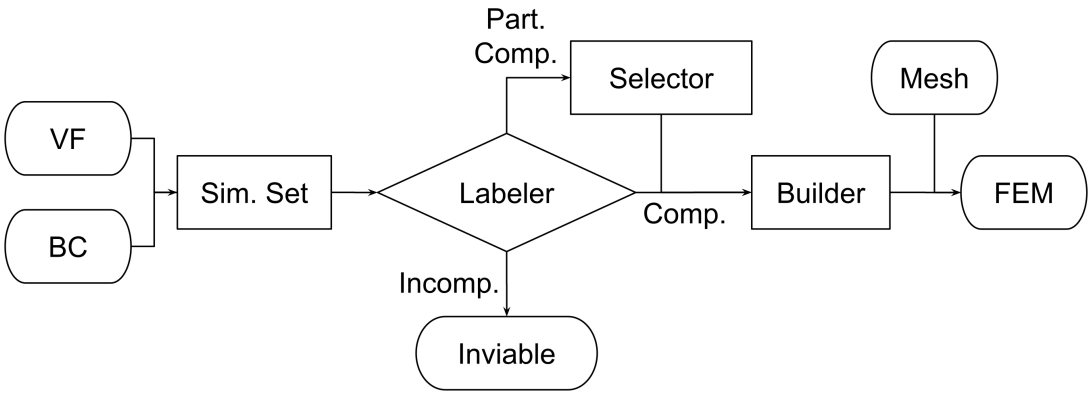


Figure 3.2: Proposed pipeline in §3.7.

To determine our pipeline's success, we run each combination of BC and VF from §3.4 through our pipeline and through a FEM simulation with manually created SFs and properly applied BCs. Our pipeline is successful when it says a simulation set is viable and the FEM simulation runs to completion. Conversely, a failure of our pipeline would be when our pipeline

Table 3.4: Results of non-shuffled simulation testing with and without completeness labels.

	Attempted	Success	% Success
Complete	7	7	100
Viable	18	18	100
All	39	8	20.5

expects a simulation set to be viable but the FEM simulation does not run. The following two subsections show the results of these tests initially where the BCs are not shuffled, and then when the BCs are shuffled, thus testing robustness to order permutations.

3.7.1 Use of Completeness Labels to Determine Simulation Viability

We first run three different groups of simulation sets: "Complete," "Viable" (*i.e.*, both Complete and Partially Complete), and "All" sets. We tabulate the number of attempted simulations and the number of successful simulations to calculate the percent of successful simulations each group attempts. With 3 VFs and 13 sets of BCs, we run up to $3 \times 13 = 39$ total simulations for each group through our FEM simulator. Table 3.4 shows the results of this experiment.

First, we see that the "Complete" and "Viable" groups have fewer attempted simulations than the "All" group, as expected. There are fewer complete and partially complete simulations than there are total possible simulations. Not all of the simulations in the "All" group ran successfully, but all of the simulations in both the "Complete" and "Viable" groups ran successfully. Additionally, the "Viable" group had more attempted (and consequently, successful) simulations because Algorithm 4 was able to prune unnecessary BCs from the simulation set.

In addition, we note there are 10 simulations that "Viable" runs successfully but "All" does not. This seems odd initially, but this behavior is explained from our conditions in §3.3.4.1. Be-

cause some of the simulation sets in "All" are partially complete, they include extra BC types that are not used in the VF. "Viable" simulation sets, on the other hand, contain completeness labels, which informs whether extraneous BCs should be pruned before constructing the simulation. Thus, when an "All" simulation set is used to construct a simulation, extraneous *BC*s cause the construction to fail, whereas "Viable" simulation sets have extra *BC*s stripped before construction.

Even taking out Partially Complete simulation sets, one should expect that "All" runs the same number of successful simulations as "Complete" does since both should run only Complete sets. However, Table 3.4 shows this is not the case: "All" contains eight successfully run simulation sets, whereas "Complete" only contains seven simulation sets. This extra viable simulation set actually comes from BC set 8 (BC8), which contains three fluid velocity and one fluid pressure BCs. Specifically, in the "Complete" pass through the BCs, our approach checks BC8 against the requirements of the Poisson VF. Finding that there is no Poisson BC, this simulation set (of fluid velocity/pressure BCs and Poisson VF) does not make a viable simulation. However, the "All" pass does not check the BC against the VF; thus, when the "All" algorithm encounters a set of fluid velocity and pressure conditions, a Navier-Stokes simulation is performed, regardless of the intended Poisson simulation. Because the Navier-Stokes VF is technically viable for BC8, the simulation completes without any problems, despite the fact that an entirely different set of equations is solved. A similar reasoning applies to the 897 "Complete" vs. 921 "All" simulations in the experiment in Table 3.5 of §3.7.1.

In summary, this experiment shows that our pipeline chooses only to attempt fully defined simulations, and it can do so without human intervention; we only set the BCs and VFs initially, and the pipeline decides which combinations of BCs and VFs to run, adjusting them as needed

Table 3.5: Results of shuffled simulation testing with and without completeness labels.

	# Attempted	# Succeeded	% Succeeded
Complete	897	897	100
Viable	**2765**	**2765**	**100**
All	3084	921	29.9

by removing extra BCs. Next, we want to make this problem more challenging by shuffling the order of the BCs.

3.7.2 Robustness of Type-Based Indexing to Boundary Condition Order Permutations

We run a similar setup as §3.7.1, but we combinatorically permute the order of the BCs. Again, we calculate the number of simulations attempted by each method and compare it to the number of simulations that were run to completion. These results are shown in Table 3.5.

From Table 3.5, we can see that both "Complete" and "Viable" simulations run with 100% success in our tested cases. However, fewer simulations (in this case, less than 1/3) are run when the "Complete" criterion is used as compared to the "Viable" criterion. "All" simulations encompass 3084 possible cases, of which only 921 (29.9%) are able to run to completion. In this case, "Viable" covers 89.7% of "All" simulations while running to completion more than triple the number of simulations. From this, we can see that the "Viable" set attempts fewer simulations but runs more successful simulations than previous implementations, thus saving in computational time wasted on failed simulation attempts.

Our type-based indexing allow both a larger number of simulations to be run without additional manual processing and that more of the attempted simulations run successfully, especially when combined with completeness labels. While it may seem that many of the simulations being

run are redundant or repetitions of other simulations (*e.g.*, BC3 and BC5 differ in the addition of a Poisson BC), many of these cases include BCs whose addition does not fully define other types of simulation sets, so running simulations with each of these BCs *should* result in the same exact simulations. In addition, our type-based indexing allow the simulations to run without regard to the order of the BCs, providing the framework for automated simulation construction from BCs. Because our approach allows more successful runs with fewer attempts, we claim that implementing the completeness and type-based indexing allows for more robust automatic simulation setup.

However, our work is limited in several aspects. We previously mentioned that each set of BCs is assumed to be physically realizable; providing these BCs is still a manual operation at this point. We discuss physically realizable BCs in Ch. 4 further. For example, given two fluid velocity inlet and a pressure BC in a pipe, our work in this chapter will consider that set as viable, even though it is physically impossible for a pipe to have only inlets and no outlets for an incompressible fluid.

Secondly, we assume that VFs are given. The derivation of these equations can be done manually, but the process is difficult to automate while keeping the scope of this chapter reasonably limited. However, this would be a potential avenue of future work.

Third, we restricted our tests to only three types of VFs and three types of BCs to make experimentally testing our claims more straight-forward. Implementing more types of physics and VFs would extend the practical functionality of this approach, though that is separate from the intellectual contributions of the chapter.

Fourth, our work only discusses DBCs without Neumann or Robin BCs. Neumann and Robin BCs are implemented differently than DBCs in that they are additional terms in the VF,

while DBCs are enforced by manipulating values in the stiffness matrix. However, simulation sets would still need to be defined appropriately to create viable simulations, even in cases with pure Neumann BCs or simulations that do not require DBCs, *e.g.*, those using inertial relief methods.

Finally, we have used only boundary value problems with no time dependence in this work. Initial value problems can be solved with similar setups of simulation sets, *e.g.*, assuming each time step in the simulation is quasi-static but depends on the previous state to determine the next state (*e.g.*, the Runge-Kutta method). We expect that extending our contributions to time-dependent equations is orthogonal to this chapter.

Although our initial goal in this chapter was the autonomous generation of meaningful FEM simulations, we realized that several critical issues arose. This chapter is in response to one of those issues, namely that ensuring the viability of a simulation is critical to generating such simulations autonomously.

3.8 Conclusion

In this chapter, we proposed three improvements to the algorithmic construction of FEM simulations. First, we proposed a type-based indexing method to generate SFs from BCs without human intervention. Second, we discussed conditions between different aspects of a simulation to ensure viability and show that these conditions can be checked automatically. Third, we show that our implementation of completeness labels guarantees simulation viability and, when combined with type-based indexing for SFs and VFs, provides the framework needed for constructing FEM simulations automatically from BCs.

Although we initially set out to enumerate possible BCs for a given physical law to produce

well-defined solutions to PDEs, we ran into several critical obstacles that this chapter attempts to mitigate. With the contributions from this chapter, we are able to generate and check for viable physics simulations automatically from collections of physically realizable BCs. With extension to a larger number of PDEs, we lay the foundations of automatically constructing single- and multi-physics FEM models, allowing future implementations to build and run simulations autonomously. These automatic simulations can then lead to physics- and data-driven design optimization, ML, analysis, and topology optimization, allowing us to apply new computational methods to mechanical design and to provide a path forward for future data-driven design methods. In this chapter, we assumed that physically realizable BCs were given; the next chapter explores how physically realizable a given set of BCs is.

Chapter 4: Physical Realizability of Dirichlet Boundary Conditions

This chapter explores the second half of the question posed in the introduction: How do we enumerate possible boundary conditions for a given physical law **that can lead to well-defined solutions to a given partial differential equation**? In particular, we attempt to generalize "well-defined solutions" by reframing this problem as a regression problem in which we predict a quantity of interest (QoI) that is a surrogate for well-defined-ness. We concentrate specifically on Stokes flow simulations and discuss generalizing to other physical laws in §4.10.

4.1 Introduction

With the development of more advanced and accurate numerical methods and the seemingly endless increase in computing power, simulations are more viable than ever in the design process. They can provide a relatively quick, cheap, and accurate method of testing before physical prototyping. However, the ubiquity of simulation work in design comes at a cost: the expertise needed to set up appropriate simulations.

As discussed in §2.2, each simulation requires a domain on which the simulation is run, boundary conditions (BC) that direct simulation behavior, and governing equations that model physical phenomena. Generally, the domain and governing equations are determined manually based on the simulation's purpose, and Ch. 3 discusses a proposed method for setting up the

mathematical constructs needed for the simulation.

However, as we also mention in Ch. 3, choosing BCs that lead to well-defined simulations is not trivial. A simple example of a *physically unrealizable* situation is a pipe with two inlets and no outlets. Physically, we would expect the pipe to burst, water to stop flowing in, or *something* to go wrong, even if we do not know exactly what that failure mode is. However, a simulation of that same pipe does not necessarily fail in the same way; often, a Finite Element Method (FEM) simulation of a pipe with two inlets will run to completion with no obvious issues. Thus, even if a simulation runs to completion, if it uses a set of physically unrealizable BCs, we should not be confident that such a simulation is trustworthy.

A physically unrealizable set of BCs for such a simulation could be difficult for a human expert to detect, even though intuitively, we often know what will or will not work. Manually checking every set of BCs would not be feasible for a fully automated system that sets up and runs its own simulations to explore different designs. This is the motivation behind our work: a fully computer-driven design tool that can choose appropriate simulations as part of an iterative design process.

In light of these difficulties, we predict a simulation's solution field (SF) magnitude as a heuristic to classifying its physical realizability, or whether the applied BCs and resulting simulation match what would happen in reality. Specifically, this chapter has the following contributions:

- We compare the preprocessing methods needed for predicting the magnitude of a velocity SF in Stokes flow. We find that using the Resolution Selection heuristic on the BCs and an autoencoder on the geometry gives us the smallest mean squared error (MSE) in our

regressions, regardless of the type of regression algorithm.

- We compare multiple types of regression algorithms for this type of prediction task. Combined with the previous contribution, we find that a fully connected Neural Network (NN) resulted in the lowest MSE when predicting the pressure SF magnitudes.

Although we specifically test our contributions on Stokes flow simulations — as we discuss in §2.2.3 — we believe that these contributions can be extended to other types of simulations as a direction of future work.

4.2 Related Work

We consider two main avenues of thought in applying machine learning (ML) to simulation viability prediction: well-posedness of problems and error prediction using ML.

4.2.1 Well-posed and Well-conditioned Problems

How well we can set up a problem depends on if the problem is *well-posed* and *well-conditioned*. These two ideas, while related, assert slightly different properties on a problem.

A *well-posed* problem is one that "has a unique solution which depends continuously on the initial data" [67], implying the existence of a solution. Our work in this paper uses Stokes flow simulations, which are known to have solutions; however, these solutions are not always physically realizabile, even if they are well-posed. For example, a pipe with two inlets and no outlets can be well-posed and "solved," *e.g.*, by FEM simulations. Such a solution would not mirror reality, though.

For comparison, a *well-conditioned* problem is one where small changes to the input lead to

small changes in the output, or, *i.e.*, has a small condition number [68]. It is possible for a question to be well-posed, but ill-conditioned. For example, double pendulum systems have drastically different behaviors with small perturbations of their initial conditions, making them well-posed but ill-conditioned. From preliminary work, we have found that slightly adjusting the BCs of a physically realizable Stokes flow simulation — *e.g.*, by reducing an inlet velocity slightly — does not drastically change the solution, which suggests the problem is not ill-conditioned. However, turbulent flows, which are ill-conditioned, are common in design problems involving fluids.

4.2.2 Regression

Regression algorithms predict QoIs from given inputs, as we discuss in §2.1.1. These QoIs can be the flow fields themselves, as in [69], though many other applications, such as lift and drag coefficient [70] or stress-strain curves [71], are possible. Previous work generally assumes that the simulation setups are valid and physically realizable and attempts to predict the flow field (for one of many examples, [72]). Others have used ML to predict errors from, *e.g.*, approximate solutions to partial differential equations (PDE) [73, 74]; Beck *et al.* [75] review some of the mathematical results and proofs related to deep learning-based approximation methods for solving PDEs. Our work is similar in that we try to predict a QoI that can be a surrogate for model error, though our QoI, as explained in §2.2.4, is not an error in solving the simulation but rather in the setup. Additionally, we specifically propose methods to handle a wider variety of geometries than other work has previously explored.

4.3 Data Generation

We break down Data Generation into Inputs and Outputs. In total, we run $15,000$ simulations to ensure that our 75% train-test split has more data points than features.

4.3.1 Inputs

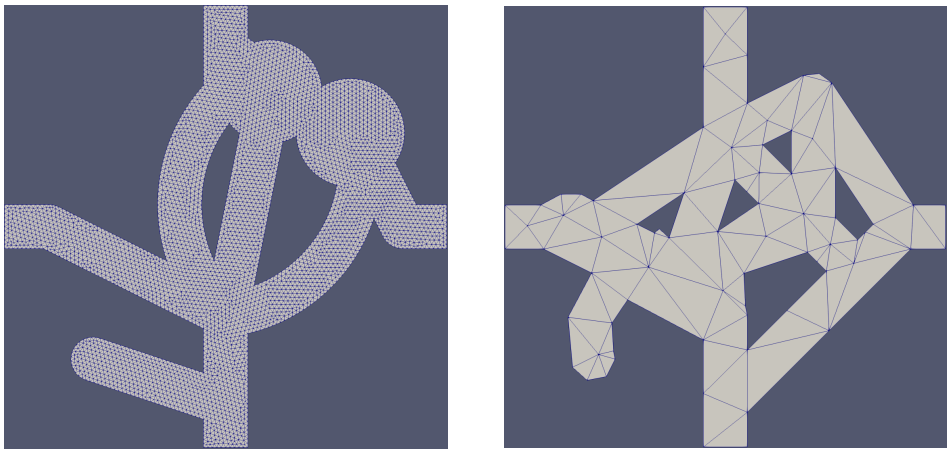
For our Inputs, we have three distinct parts that we calculate before running any simulations: simulation parameters, BCs, and geometry representation (Geo. Rep.). Each of these components includes raw data generation followed by preprocessing.

4.3.1.1 Simulation Parameters

The first component is numerical information about the simulation setup. These are a parameter that guides mesh cell size and the Lipschitz number of the BCs.

The mesh cell size is a value used by the meshing software pygmsh [26]. Previous experience has shown that overly coarse meshes cause numerical instabilities in simulation. We pass a mesh resolution of 10^k for $k \in \{-2, -1, 0, 1, 2\}$. Internally, pygmsh then refines the mesh using the given mesh resolution. We store k as the first term in our simulation parameters. We refer readers to Fig. 4.1 for two example geometries with different mesh resolutions.

Second, we calculate the Lipschitz number [76] to approximate how quickly the BCs change. Intuitively, more rapidly changing BCs have more potential for divergences, though our work limits the possible BCs that can be applied; the Lipschitz number is a way to quantify how rapidly a BC changes. In this work, we approximate the Lipschitz number with the maximum finite difference derivative of our BCs with respect to position. We scale the Lipschitz



(a) Mesh resolution of 10^{-2}. (b) Mesh resolution of 10^{1}.

Figure 4.1: Two example meshes of varying resolutions.

numbers across all of our simulations to values between -1 and 1.

4.3.1.2 Boundary Conditions Representation

The second component is a representation of the BCs. The BCs seem to have the greatest impact on the pressure field magnitude, so including them in the inputs to our ML model is critical for this application. We choose from three possible BCs, as discussed in §2.2.2: a no-slip condition V_0, a velocity BC V_{in}, or a pressure BC P_0. Every port has exactly one of these BCs.

We discretize the BCs with 20 points along each of the 4 ports, as we did in [77], and append the x- and y-velocities to our inputs. We exclude pressure from our inputs because all of our applied pressure BCs are set to 0. With two velocities at each point, twenty points along each port, and four ports in a device, this adds a total of $2 \times 20 \times 4 = 160$ dimensions to the input vector. We scale the x- and y-velocities between -1 and 1.

As a comparison, we replace the scaled BC values with three additionally preprocessed alternatives: a Principal Component Analysis (PCA) representation, set to keep enough components to explain 95% of the variance; a Resolution-Selected representation of the BCs as per [77];

and a combination of both PCA and Resolution Selection.

We refer the reader to §2.3.4, §2.3.5, and §2.4.1 for more explanation of BCs, dimensionality reduction, and saving behavior vectors, which are treated similarly to the BCs here.

4.3.1.3 Geometry Representation

The third component is our Geo. Rep. From experience, geometries that are overly complex or have very small channels causes FEM simulations to diverge. Including the Geo. Rep. in our inputs increases the dimensionality significantly but also may provide discriminatory information for our regression models. We first randomize our fluid domain as in §2.2.1. Some sample geometries are show in Fig. 4.2.

Figure 4.2: Sample geometries with 0/1 encoding.

We compare our baseline 0-1 encoding with a PCA representation, as discussed in §2.3.5.1. Others have used between 1 and 20 dimensions [70, 78, 79, 80], a minimum eigenvalue [81], or 98-99% explained variance [82, 83]. However, there did not appear to be consensus on the method, which may imply that the appropriate level of explained variance is domain-specific. In this work, we keep enough components to explain 95% of the variance.

As a second comparison, we apply an autoencoder (AE, [51], see §2.3.5.3) with a structure similar to [53] to our 0-1 encoding and as shown in Fig. 4.3. Specifically, the encoding half of the AE has 5000, 2500, 1000, 500, 100, and finally, 50 nodes in each layer before embedding the representation in 10 dimensions; the decoding half has the same number of nodes per layer in reverse. Similar to [53], we use a latent space of 10 dimensions and a ReLU activation function on every step except for the last layer of the decoding network, where we use a Sigmoid function to discretize the representation. We also apply an average pooling function (kernel size of 3, stride and padding of 1) on the final forward step of the decoder to reduce checkerboarding in the output. Many different types of AEs exist (*e.g.*, convolutional AEs [84]), but we limit our work to a basic AE and acknowledge that testing additional types of AEs could be an avenue for future work. We train over 200000 epochs with an early stopping patience of 5000 iterations and a threshold of 1e-4 on an Adam optimizer [21] using a learning rate of 1e-3 and betas = $(0.9, 0.999)$).

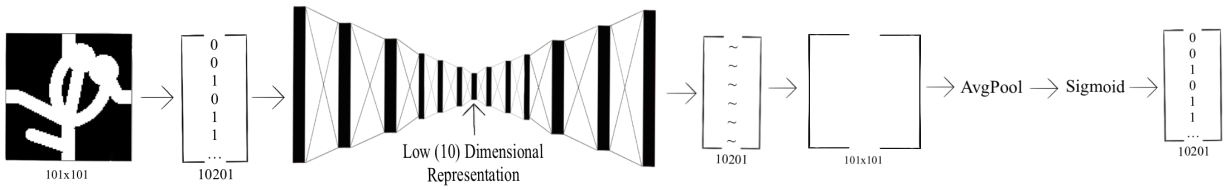


Figure 4.3: Architecture of the AE used to encode geometries.

For our third comparison, we apply a center-shifted discrete Fourier transform (DFT) to our 0-1 encoding to create a spectral representation based on [50]. Other methods exist, but we limit our work to this method and acknowledge that testing additional types of image compression — such as JPEG compression algorithms [85] — could be an avenue for future work. We refer the reader to §2.3.5.2 for more details on using DFT for dimensionality reduction. In this particular work, this reduces our Geo. Rep. to a latent space of around 480, or about 47% of the original Geo. Rep.'s dimensionality.

4.3.2 Outputs

We use the BCs described in §2.2.2 and meshes from §2.2.1 to simulate Stokes flow, as in §2.2.3.

After running these simulations, we can calculate the magnitude (*i.e.*, L^2 norm) of the vector representing the pressure SF; §2.2.4 elaborates the calculation of this magnitude. From preliminary work, these magnitudes distinguish between physically realizable and unrealizable simulation. Figure 4.4 shows three distinct groups of magnitudes. The first group, with norms equaling 0, contains trivial simulations that have no V_{in}. The stagnant fluid "flow" results in the pressure being 0 everywhere in the domain.

The second group, with moderate L^2 norms, contains physically realizable simulations that have converged appropriately. These are simulations with at least one V_{in} and at least one P_0. V_{in} ensures that fluid will flow into the device, while P_0 guarantees an exit for the fluid to maintain conservation of mass. These are simulations that are non-trivial and physically realizable.

The third group, with large L^2 norms, contains physically unrealizable simulations. These

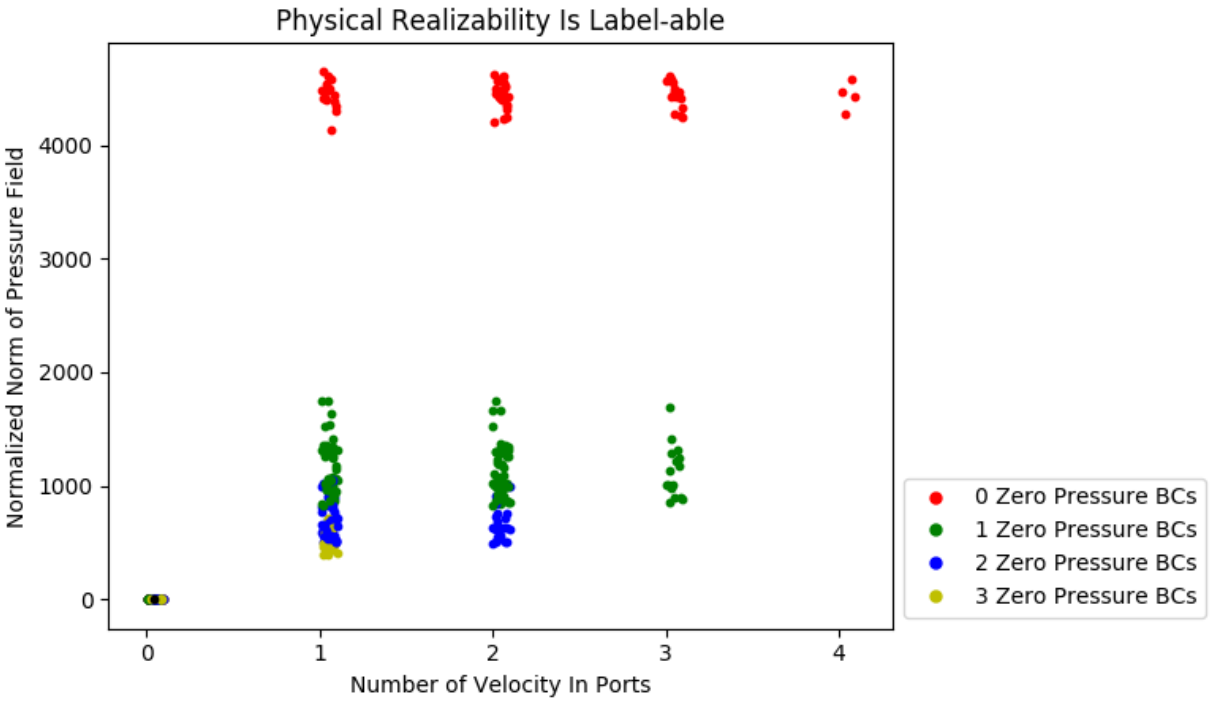


Figure 4.4: Physical realizability is distinguishable by the pressure SF magnitudes. Each point is a simulation. Jitter has been added to x-coordinates for visibility.

are simulations with at least one V_{in} but no P_0, similar to a blocked pipe. Despite being physically unrealizable, these simulations will run to completion.

4.4 Regression Across Preprocessing Methods

For our three types of inputs, we use corresponding preprocessing methods. Simulation parameters are scaled, as discussed in §4.3.1.1. BC representations are scaled and have PCA and Resolution Selection applied, as discussed in §4.3.1.2. Our Geo. Rep. has PCA and an AE applied, as discussed in §4.3.1.3.

We use Linear Regression, K-Nearest Neighbors Regression (KNNR), and Gaussian Process Regression (GPR). Briefly, Linear Regression is a widely used, simple, yet powerful model that linearly weights and sums its inputs to minimize the sum of the squared errors; KNNR

weights close neighboring points to predict new outputs; and GPR adapts an internal function using kernel functions to approximate the underlying physics. We refer the reader to §2.1.1 for deeper discussion of these models. We report the MSE between the predicted and the simulated pressure SF magnitude and refer the reader to §2.1.1.6 for more details on the MSE.

4.5 Results Across Preprocessing Methods

We combinatorically run each BC preprocessing, Geo. Rep. preprocessing, and regression algorithm. The MSE of these experiments is shown in Tables 4.1, 4.2, and 4.3. We split the results by regression algorithm.

From Table 4.1, we see that applying no preprocessing on the Geo. Rep. results in the highest Test MSE. However, applying either AE or DFT increases the Train MSE while decreasing the Test MSE by many orders of magnitude. However, the MSE variation across BC preprocessing is much smaller than across Geo. Rep. preprocessing. The Train MSE for applying AE is greater than the Test MSE; it is more common to see a lower Train MSE than Test MSE. Overall, applying Resolution Selection on the BCs and an AE on the Geo. Rep. gives the lowest Test MSE for Linear Regression.

Table 4.2 shows some different trends. Most notably, the Train MSE of every algorithm is 0. The Test MSEs are lower than that for Linear Regression, except when AE is applied to Geo. Rep. We see again that the type of BC preprocessing has a small effect, if any, on the Test MSE. Overall, applying Resolution Selection or Resolution Selection + PCA on the BCs and an AE on the Geo. Rep. gives the lowest Test MSE for KNNR.

Table 4.3 shows the Train MSEs being 0, as we saw in KNNR, but with lower Test MSEs.

Table 4.1: MSE of Linear Regression

Geo. Rep.	Preproc.	Train MSE	Test MSE	Train % Error	Test % Error
None	None	1.2522e5	1.308e26	95.8	3.1e12
	PCA	1.2065e5	330.5e26	94.0	49e12
	Res	1.3538e5	75.1e26	99.6	23e12
	ResPCA	1.1803e5	106.9e26	93.0	28e12
AE	None	7.5357e5	6.8026e5	234.9	223.4
	PCA	7.1634e5	6.5261e5	229.1	218.8
	Res	**7.1634e5**	**6.5260e5**	**229.1**	**218.8**
	ResPCA	7.1634e5	6.5261e5	229.1	218.8
DFT	None	9.2102e5	1147.7e5	259.7	2901.3
	PCA	5.3608e5	1906.5e5	198.2	3739.4
	Res	6.0804e5	1857.9e5	211.1	3691.4
	ResPCA	5.3608e5	1909.4e5	198.2	3742.2

Table 4.2: MSE of KNNR

Geo. Rep.	Preproc.	Train MSE	Test MSE	Train % Error	Test % Error
None	None	0	10.837e5	0	281.9
	PCA	0	10.837e5	0	281.9
	Res	0	10.950e5	0	283.4
	ResPCA	0	10.950e5	0	283.4
AE	None	0	7.6976e5	0	237.6
	PCA	0	7.6976e5	0	237.6
	Res	**0**	**7.1414e5**	**0**	**228.9**
	ResPCA	**0**	**7.1414e5**	**0**	**228.9**
DFT	None	0	11.023e5	0	284.3
	PCA	0	11.023e5	0	284.3
	Res	0	11.023e5	0	284.3
	ResPCA	0	11.023e5	0	284.3

Either no preprocessing or applying DFT on the Geo. Rep. results in approximately the same Test MSEs. However, applying AE to the Geo. Rep. gives a lower Test MSE than any of the other experiments. The best performance is achieved by again either Resolution Selection or Resolution Selection + PCA on the BCs along with an AE on the Geo. Rep.

Table 4.3: MSE of GPR

Geo. Rep.	Preproc.	Train MSE	Test MSE	Train % Error	Test % Error
None	None	0	9.2004e5	0	259.8
	PCA	0	9.2004e5	0	259.8
	Res	0	9.2004e5	0	259.8
	ResPCA	0	9.2004e5	0	259.8
AE	None	0	5.8630e5	0	207.4
	PCA	0	5.8630e5	0	207.4
	Res	**0**	**5.6205e5**	**0**	**203.0**
	ResPCA	**0**	**5.6205e5**	**0**	**203.0**
DFT	None	0	9.2004e5	0	259.8
	PCA	0	9.2004e5	0	259.8
	Res	0	9.2004e5	0	259.8
	ResPCA	0	9.2004e5	0	259.8

4.6 Discussion Across Preprocessing Methods

Most apparently, we see from Tables 4.2 and 4.3 that our Train MSEs for all KNNR and GPR algorithms are 0, while the Test MSEs remain large. A small Train MSE with a large Test MSE is often a sign of overfitting in the model; that is, the model is learning to reproduce the data on which we train, but has low generalizability to data it has not seen. We expect that our model is overfitting because the number of inputs is very large relative to the number of data points, and that the model is overly complex for the amount of training data we use. We also note potential overfitting in Linear Regression with no Geo. Rep. preprocessing, where the Test MSE is much higher than the Train MSE from Table 4.1.

Unexpectedly, we find from Table 4.1 that Linear Regression and AE, across all BC preprocessing methods, has a smaller Test MSE than Train MSE. The Train MSE is usually less than the Test MSE because the Train MSE is the objective function being minimized. A lower Test MSE can be a sign of differences in data set difficulties or a fluke of the randomization. However, we use $11,250$ data points in our training set and $3,750$ in our test set, which we expect to make

differences in data set difficulties and flukes of randomization unlikely; the data distribution is shown in Fig. 4.5.

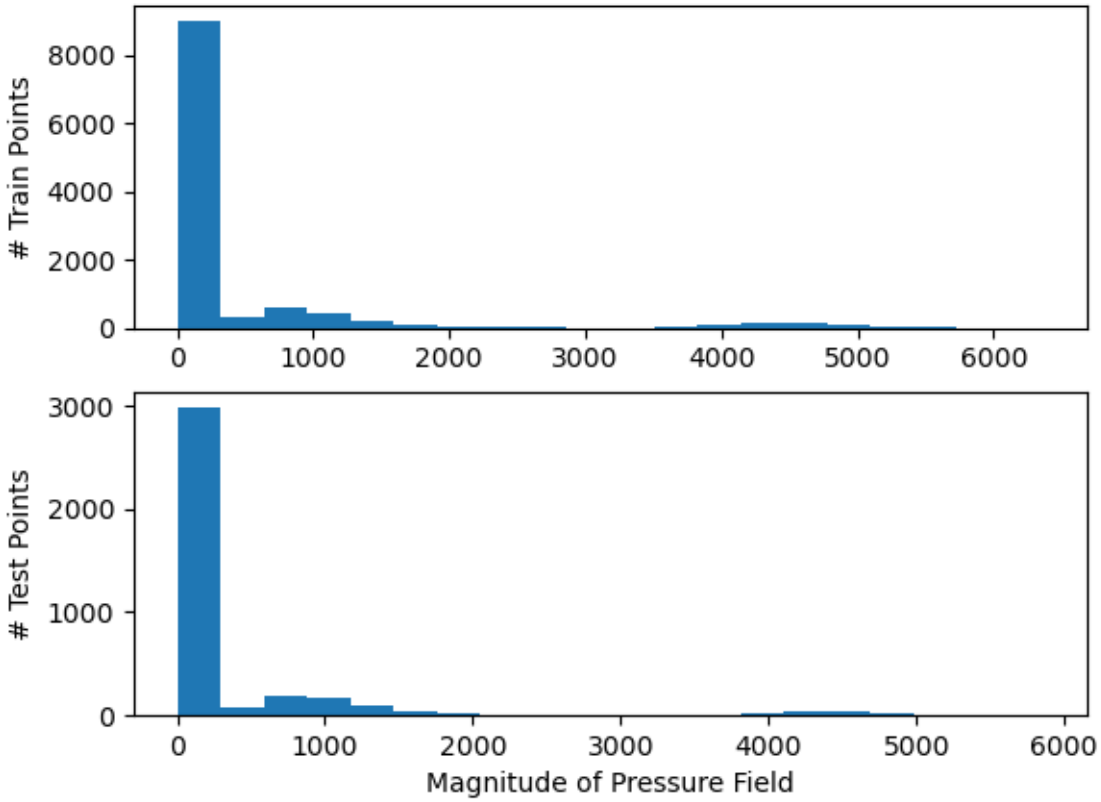


Figure 4.5: Training and Testing data have similar distributions of pressure magnitudes.

In general, we find that applying AE to the Geo. Rep. is the most effective preprocessing, regardless of regression algorithm and BC preprocessing. Although Train MSE on some combinations of regression and BC preprocessing are comparable, AE on the Geo. Rep. gives a lower Test MSE across any given regression algorithm. The Geo. Rep. is, dimensionally, the largest component of our input, and AE reduces it from 10201 dimensions to 10 latent dimensions, or about 0.1% of its original dimensionality. However, AE is the most computationally expensive Geo. Rep. preprocessing. We have not compared our AE against other types of AEs — *e.g.*,

Variational AEs [86] — or by varying the amount of training we apply on the AE preprocessor, but we suspect that more effective dimensionality reduction methods exist for this application. Intuitively, we believe that the Geo. Rep. should be dimensionality reducible by a significant amount; however, we are unsure to what degree or the most effective method for doing so.

From this first experiment, we find that GPR is the most accurate regression method that we studied. GPR has a lower Test MSE than either KNNR or Linear Regression for any given Geo. Rep. and BC preprocessing method, though it is computationally more expensive than either KNNR and Linear Regression. However, this is somewhat surprising because GPR generally does not scale well to high-dimensional problems; a common rule of thumb is that GPR works until 15-20 dimensions, but beyond that, the curse of dimensionality impedes its performance.

Based on this experiment, we suspect that some of our inputs are not discriminative for this type of problem. Going another step further, we hypothesize that the Geo. Rep. may, in fact, be detrimental to the regression performance due to its high dimensionality and potentially low impact on the pressure field magnitude. We also believe that the Lipschitz number is not helpful in our work in this chapter because of the limited range of BCs we use. Since all of the BCs are limited in magnitude and shape, the Lipschitz number takes a limited range of values and may be more discriminative in datasets with a wider variety of BCs. However, we do not formally test or analyze the discriminative value each input has, so a more thorough exploration of simulation parameter representation is an avenue of future work.

4.7 Deeper Dive into Regression Methods

From the results in §4.5, we surmise that a deeper dive into regression methods could produce better predictions of pressure SF magnitudes. Most notably, we want to turn our attention toward both more complex regression methods as well as tuning hyperparameters. To this end, we tune and compare seven types of regression algorithms.

We begin again with Linear Regression, though we use a 5-fold cross validation scheme to train five different models. Of these, we choose the model with the lowest validation error as our final model, which we then compare to our test data.

As another linear model, we train an Elastic Net regressor [87], which is similar to Linear Regression but with the addition of L^1 and L^2 penalties on the coefficients. We additionally use a 5-fold cross validation scheme to set the weight of the penalty terms relative to the error and the weighting between the L^1 and L^2 terms.

Our third model is KNNR. We use a randomized 5-fold cross-validation search to set the number of neighbors and whether the neighbors are uniformly weighted or inversely weighted by distance.

Our fourth model is a GPR, similar to above. However, we modify the kernels to follow Eq. 4.1:

$$Y = C_1 * K(x) + C_2 + \epsilon \tag{4.1}$$

where C_1 and C_2 are tunable constants, ϵ is tunable white noise, and $K(x)$ is one of a RBF kernel or a Matern kernel with $\nu = 0.5$, $\nu = 1.5$, or $\nu = 2.5$, all with length scales bounded between 1 and 1e3. We then train our regressor with all four types of kernels and use the kernel with the

least MSE as our GPR model.

Our fifth model is a fully connected NN. As part of the hyperparameter tuning, we consider a variety of numbers and sizes of hidden layers (HL), types of activation functions, and optimizer parameters. We use a randomized 5-fold cross-validation search to set these parameters, similar to previous regressors, and search over $10,000$ iterations. The hyperparameter bounds we use are as follows:

- Activation Functions: Sigmoid, Tanh, ReLU
- Alpha: log-uniform distribution between $1\mathrm{e}{-5}$ and $1\mathrm{e}2$
- Beta 1: uniform distribution between 0 and 1
- Beta 2: uniform distribution between 0 and 1

We choose a number of HLs — between 1 and 4, inclusive — and a random number of nodes independently for each layer — between 10 and 50, by multiples of 5, inclusive. We train each NN using the Adam optimizer [21] for the number of epochs needed for at least 1e6 total optimizer steps (*i.e.*, $ceil(\frac{1e6}{numData})$ epochs), and the corresponding parameters chosen from the above lists. We use the models with the architecture and hyperparameters that give us the least Validation MSE.

Finally, we compare these to a "mean" regression, where the predicted value is simply the mean of the training outputs. That is, we take the mean of the outputs of our training data and use that as the predicted value for all of our testing data. Intuitively, this gives us some idea of whether advanced algorithms are worth the more complex implementation and cost to train, or whether the problem in its current form is too simple or difficult for these types of methods.

We again refer the reader to §2.1.1 for deeper discussion of these models and the MSE metric and to §2.1.4 for more details on hyperparameter tuning.

4.8 Results Across Regression Algorithms

We summarize the results of our experiments in Table 4.4. Most notably, we see that KNNR has 0 Train MSE, even though its Test MSE is not comparatively close to 0. We see that NNs have the next lowest Train MSE, at $3.732\text{e}5$, while maintaining the lowest Test MSE at $3.880\text{e}5$. However, GPR has a similar Train MSE at $3.757\text{e}5$ with a Test MSE that is slightly higher at $3.904\text{e}5$. Both Linear Regression and Elastic Net Regression have relatively close Train and Test MSEs to each other, though higher than any of the other regression algorithms. Compared to using Mean regression, every tested regression algorithm improves on the Train and Test MSE, though to varying degrees. NN and GPR provide the lowest Test MSEs — around 42% of the Mean Regression MSE — though KNNR is not far behind with a Test MSE at 48% of Mean Regression's MSE. As a note, our hyperparameter tuning gave a GPR kernel of `constant(1)*rbf(length = 1) + constant(1) + noise(level = 1)` and a NN architecture of 45-40-15-30 HLs, a ReLU activation function, $\alpha = 0.003249756$, initial learning rate of $lr = 0.091874356$, $\beta_1 = 0.8454759977$, and $\beta_2 = 0.70782938$.

Table 4.4: MSE of Regression Algorithms using Resolution Selection and AE preprocessing.

Regression Algorithm	**Train MSE**	**Test MSE**	**Train % Error**	**Test % Error**
Linear Regression	7.196e5	6.508e5	229.6	218.5
Elastic Net	7.614e5	6.832e5	236.2	223.8
KNNR	0	4.442e5	0	180.5
GPR	3.757e5	3.904e5	165.9	169.2
NN	**3.732e5**	**3.880e5**	**165.3**	**168.7**
Mean	10.11e5	9.225e5	272.1	260.1

4.9 Discussion Across Regression Algorithms

Despite the low Train MSE, neither KNNR does not have the lowest Test MSE. Specifically, KNNR has the third lowest Test MSE, which is about 15% higher than that of NN, while GPR's Test MSE is on the same level as NN. GPRs generally do not scale well, so applying it to our data — which has more than the 15-20 dimensions GPR generally can handle — initially results in GPR more or less blindly guessing. However, by forcing its length scales to respect a minimum value of 1, GPR overfits less on the Training data and thus lowers its Test MSE. Even so, we do not believe GPR is an appropriate model for this type of problem based on the chosen kernel.

We surmise that because KNNR is more flexible than the linear models, it is able to reach a lower MSE than the linear models. In fact, if the neighbors in a KNNR model are inversely weighted by distance — as it is in this case — the Train MSE should be exactly 0. The nearest point in the training set will be exactly the point at which we are predicting, implying that its distance from the point-to-be-predicted is 0, and thus making its weight infinitely high. In practice, when predicting on one of the training points, the model exactly predicts the training point's true output, thus making the MSE 0. Doing so does not necessarily result in the best Test MSE, though, as the test data may be relatively far and in sparse regions of space, especially with low numbers of data and high dimensional inputs.

The two linear models have relatively close Train and Test MSEs, which is unsurprising given that Elastic Net is effectively Linear Regression but with a slightly modified penalty term. We suspect that if Linear Regression were able to overfit more severely, either through a simpler problem or more independent inputs, then Elastic Net would regularize the model more effectively and reduce overfitting. That said, Linear Regression already seems unable to overfit to

our data, so applying additional regularization in the form of an Elastic Net does not improve performance.

Overall, we find that NN has the lowest Test MSE while maintaining a relatively low Train MSE. Because both the Train and Test MSE are relatively close, we expect that this model has converged without overfitting to the training data. However, the high percentage error in both our Train and Test datasets leads us to believe that there is room for improving the model by modifying the input and output preprocessing, increasing the amount of training data, and performing a more thorough hyperparameter search.

4.10 Conclusion

In this chapter, we set out to explore the conditions for generating well-defined solutions to PDEs. To this end, we proposed a novel approach to use ML to predict a simulation's physical realizability, *i.e.*, whether or not such a simulation would mimic reality. We tested various preprocessing methods and found that applying the Resolution Selection preprocessing on BCs and using an AE on Geo. Reps. gives the lowest MSE in predicting pressure SF magnitudes, which we assert are a surrogate for simulation convergence. Additionally, we found that a NN provides the lowest MSE when combined with the given preprocessing methods.

AE preprocessing resulted in the lowest MSE, but we found that the AE preprocessing still resulted in significant reconstruction loss, even after training. In addition, the computational cost of training an additional NN may not be worth any benefits to the regression. We thus consider that a potential avenue of future work could be to engineer the inputs more carefully, *e.g.*, by removing the Geo. Rep. entirely from the inputs. Doing so may reduce the problem

to a mass balance integration over boundaries, though doing so would not catch cases where the simulation crashes because of, *e.g.*, time steps being too large or meshes being too coarse, causing divergences.

When considering the extensions of our contributions within the realm of FEM simulations, we expect that extension to the Navier-Stokes equations is straightforward. The Stokes equations are a form of the Navier-Stokes equations, so we do foresee any major difficulties in transferring our contributions between the types of simulation. The only possibility we see is that the simulations encompass more chaotic and fast-moving flows, which may result in blurrier distinctions between physically realizable and unrealizable simulations. Within continuum mechanics, we generally expect slower, less chaotic phenomena — such as heat dissipation and Poisson simulations — are within the scope of our contributions, while more rapid or chaotic phenomena — such as turbulent flow or buckling in solids — are more difficult to predict, even if they are physically realizable. We are not able to claim that our contributions are currently compatible with other classes of simulations, such as n-body simulations or Monte Carlo-based simulations. However, we hope that our methods may provide some insight to other types of problems for future researchers.

If we want to extend our contributions to other types of physics, the specific QoI to be predicted and most informative inputs likely must be revised as physics-dependent parameters. In this chapter, we concentrate on the L^2 norm of the pressure SF in Stokes flow simulations as the QoI based in part on intuition and past experience. We have previously explored fluid simulations, and when physically unrealizable simulations were attempted, we would note that the SF magnitudes would spike. We directly tested several physically unrealizable simulations of pipes with multiple inlets but no outlets, and we observed that the pressure SF specifically saw

the greatest increase in magnitude. Since a fluid simulation's outputs are effectively just a list of pressure and velocity SFs, our QoI should be derived from these SFs, both in this case and in general. However, using pressure magnitude as the predictive value on, *e.g.*, a heat diffusion simulation would not be reasonable, so field-specific expertise is needed to determine the most appropriate surrogate QoIs for physical realizability for differing physics. Even so, we expect that our process of calculating SF magnitude is directly applicable to simulations with a single SF; the only difference is that the corresponding SF should be used (*e.g.*, temperature in a heat diffusion simulation). For physics where physically unrealizable BCs may not exist, the contributions in this chapter may focus more on using simulation setup parameters rather than BC representation to inform the algorithm. For example, the authors have not come across a setup for a heat dissipation simulation that was physically unrealizable, but the error compared to analytical solutions of such simulations scales with, *e.g.*, the mesh resolution. Models using the approach detailed in this chapter to predict heat dissipation simulations may, then, place more weight on the mesh resolution than the BCs being applied. As such, generalizing to different types of physics and simulations through the specification of QoIs and the specific inputs on which models implementing the approach discussed in this chapter is an avenue of future work.

The difficulty arises when considering simulations with multiple SFs. Methods to combine, project, or otherwise analyze multiple QoIs (*e.g.*, weighted sums or multidimensional approaches) are possible, and exploring more robust methods of choosing QoIs are a potential avenue of future work, including whether physically unrealizable simulations exist within certain physical laws. In combination with Ch. 3, this chapter helps us move toward fully automatic FEM simulations being set up and run by machines to autonomously explore new designs.

Chapter 5: Composition of Multiple Simulations

This chapter explores the second question posed in the introduction: **How do we encapsulate the emergence of complex behaviors from interactions between different components?** In this chapter, we combine small fluid simulations to form big simulations, knowing that such composition causes error. We use Graph Neural Networks as a method to predict the error that arises from composing smaller simulations in fluidic simulations, and we discuss generalizability to other physics in §5.9.

5.1 Introduction

One of the difficulties of composing multiple smaller simulations into a larger one is the existence of long-distance and emergent phenomena. Much of the current work has focused on predicting behavior in a single simulation. However, these long-distance phenomena that may span multiple small simulations do not necessarily appear except in length scales beyond a single small simulation. As such, this chapter focuses on breaking down a large, complex simulation into small simulations, each distinct from the others, and then composing them by correcting for long-distance phenomena. In theory, a relatively small set of basis simulations, with an appropriate composition function, can allow prediction of a combinatorially large set of big simulations.

For a pedagogical example, consider the static fluidic oscillator shown in Seiber *et al.* [88]. This device can be split into three distinct parts: a central chamber, an upper tube, and a lower tube. Each of these three parts can be simulated separately, but doing so would not show oscillatory sweeping flow with static boundary conditions (BC) like the full device has. Only when all three parts are simulated together does the flow oscillate. As such, the oscillatory phenomenon only occurs in the big simulation as an emergent phenomenon. This example motivates finding an appropriate composition function that can predict emergent phenomena, even when the small simulations that comprise it do not show such behavior.

This chapter extends our previous conference paper [3] that used Graph Neural Networks (GNN) [4] to predict the composition of multiple smaller simulations. In particular, this chapter expands the domain and graph from a two-pipe connection to a 4×4 grid, including more types of geometries. The specific contributions of this chapter are:

1. We generalize the compositional method from [3] — which considered single-direction pipe networks with known BCs — to pipes with unspecified flow directions. Here, we propose a heuristic for assigning BCs to subsections with *a priori* unknown BCs. We demonstrate this method on computing pipe flows in networks with Reynolds numbers $Re = 100$. When combined with GNNs, our method can predict the fluid flow fields with a mean squared error (MSE) of 0.02029, compared to a naïve composition with a MSE of 0.07140. This corresponds to up to a 1.5x lower percent error when BCs are selected using our heuristic as opposed to using other BCs.

2. We compare GNN architectures using Graph Convolution Networks (GCN) [89] and Graph Attention Networks [90] with one and six hidden layers (HL). We find that a GCN with 1

HL and 819 neurons per layer gives us the smallest MSE of the GNNs we test on this type of task, but that Linear Regression gives us about half the MSE of any GNNs we test.

We concentrate testing these contribution on Navier-Stokes simulations, as described in §2.2.3, with a Reynolds number $Re = 100$ using geometries we generate in §5.5. We focus specifically on a limited class of fluid simulations to provide a scientifically interesting test bed for our approach while minimizing extraneous difficulties with simulation, but we expect that this work can be extended to other types of simulations. We discuss the conditions of these extensions in §5.9.

5.2 Related Work

This section reviews the current state-of-the-art pertinent to (1) compositional machine learning and (2) GNN models for physical system modeling and reasoning.

5.2.1 Compositional Machine Learning

While a single machine learning (ML) model (*e.g.*, a Neural Network (NN)) can be good at accomplishing single types of tasks (*i.e.*, abstractions) for a small set of phenomena — *e.g.*, using a single NN to predict a flow velocity field [91] or classifying an image into dogs or cats [92] — one single model cannot be good at tasks with different sub-tasks. Compositional ML combines several NNs to perform segments of a bigger task and assumes the compositional model as a whole performs better at the intricate sub-tasks while maintaining reasonable computing space [93].

In contrast to state-of-the-art ML techniques focused on training distinct ML models on different types of data and aggregating the results into a single global model for a single target

problem — *e.g.*, *Federated Learning* [94] or *Ensemble Learning* [95] — compositional learning concentrates on different models for different types of sub-tasks. For example, Ivanov *et al.* [96] decomposed the complete car navigation task into a sequence of sub-tasks (*i.e.*, segments of the track) and developed separate NN controllers for each of the sub-tasks (*e.g.*, go straight or turn) and ensured the correct transitions between arbitrary compositions of the sub-tasks. Similarly, Jagtap and Karniadakis [97] propose a framework for space and time decomposition of simulations where individual physics-informed NNs are employed in each subdomain. Modeling language and thought is another challenge for NNs that have been criticized for lacking compositionally [98]. Recent work has proposed the sequence-to-sequence (seq2seq) models [99, 100] including the SCAN (Simplified version of the CommAI Navigation) data [101] for compositional learning [102]. Data science practitioners have a pressing need to enable this compositionality, especially those outside the IT industry, because of the lack of training data [103]. The mechanical engineering community has been increasingly dissatisfied with NNs because simulated, optimized, or manufactured data (*e.g.*, computational fluid dynamics simulation and airfoil design optimization) for training are both computationally and physically expensive [104] to obtain. As a typical example, high Reynolds numbers and scale-resolving simulations using Navier-Stokes equations are not attainable with existing computational resources, and the alternative approximated simulations or experiments are also expensive and slow. ML provides new avenues for flow modeling by supporting a more concise framework in both kinematics and dynamics [105].

In this chapter, we further enhance our prior work [3] by taking the fluid flow modeling as a pedagogical example and attempting to fill the gap by breaking the heavy-duty task (*i.e.*, big fluid simulations) into lighter sub-tasks (*i.e.*, small simulations) and substituting the time-consuming heavy task with our compositional ML model, which corrects the transitions between sub-tasks

using GNNs. The related work to GNN models in system composition will be reviewed in the following section.

5.2.2 Graph Neural Network Models for Physical System Modeling and Reasoning

GNNs [106] are connectionist models that capture the dependence of graphs via message passing between the nodes of graphs. Zhou *et al.* [107] defined GNN applications in physical system modeling as a structural scenario where the data has an explicit relational structure and the applications in image analysis as a non-structural scenario where the relational structure is not explicit. In this chapter, we investigate the physical system composition of fluid pipe flows relying on non-structural image data, thus reviewing the relevant work below.

By representing objects as nodes and relations as edges, a GNN can reason about objects, relations, and physics in a simplified but effective way. Inspired by the GNN model [106], Battaglia *et al.* [108] introduced an interaction network (IN), which they propose as a general-purpose, learnable physics engine and a framework for reasoning about objects and relations in a wide variety of complex real-world domains. Sanchez-Gonzalez *et al.* [109] then extended and improved the GNN model to support accurate prediction, inference, and control across eight distinct physical systems. Rather than INs that tackle fully observable systems, Li *et al.* [110] further developed propagation networks (PropNet), a differentiable, learnable dynamics model that handles partially observable situations and enables instantaneous propagation of signals beyond pairwise interactions. The DeepMind AI group has made available a *Graph Nets*[1] library

[1] https://github.com/deepmind/graph_nets

that enables building GNNs in *TensorFlow*[2] and *Sonnet*[3] based on the work of Battaglia *et al.* [4]. Their library provides the flexibility to design GNNs for different applications, including physical system prediction (*e.g.*, mass-spring systems). Instead of using static and dynamic parameters of a physical system, Watters *et al.* [111] proposed a visual interaction network (VIN) that predicts future physical states from input image frames (pixels from video data).

Overall, GNN models provide an avenue for using a graph to summarize human knowledge and experience, while the reasoning mechanism can discover potentially unknown knowledge based on existing knowledge in the graph [112]. This stretches the capacity of ML models in compositional learning when concentrating on multiple sub-tasks with known knowledge when system composition needs unknown knowledge.

In this chapter, we adopt the PyTorch Geometric library [113] to compose fluid pipe flows to construct a GNN model depending on non-structural images of the pipe flow velocity field. To the best of our knowledge, GNN models have never been used to reason and compose physical systems using whole images.

5.3 Background: Graph Neural Networks

To understand what a GNN is and how it works, it is essential first to understand what a *graph* is. An in-depth discussion of graphs is not the purpose of this chapter, but for the purposes of this dissertation, it is sufficient to know that graphs are data abstractions that show relations between *nodes* via connections called *edges*. We refer readers to, *e.g.*, *Introduction to Graph Theory* [114] for more details on graphs.

[2]https://www.tensorflow.org/
[3]https://sonnet.readthedocs.io/en/latest/

GNNs are a specialized type of NN that implements *message passing* between nodes of a graph for predictive tasks. In general, GNNs can predict either node-, edge-, or graph-level features and properties. This dissertation focuses on node-level regression tasks, *i.e.*, tasks that predict values specific to each node in a given graph.

Broadly speaking, messages passing for a given node consists of three steps. First, node features for all neighboring nodes are collected and aggregated using a function that can take varying numbers of inputs, forming a "message." This aggregation function is commonly a sum, mean, or max function. Second, the aggregated message is transformed with a function common across all nodes. This is often a relatively simple NN. Finally, the original node updates its own values using the transformed message. Repeating this message passing across every node in the graph constitutes a single iteration of the message passing step. Vectorizing this implementation gives rise to Eq. 5.1 for a GNN with l HLs,

$$f(H^{(l)}, A) = \sigma(AH^{(l)}W^{(l)}) \tag{5.1}$$

where f is the output of the next HL in the GNN, $H^{(l)}$ is the output of the previous HL, A is the adjacency matrix without self-loops, σ is a non-linear activation function (see §2.1.1.3), and $W^{(l)}$ is the weight matrix for the l-th HL. We refer readers to, *e.g.*, *Graph Representation Learning* [115] for more details on GNNs. We will briefly introduce the two types of GNNs we use in this chapter: GCNs [89] and GATs [90]. However, we refer readers to the original papers for a more in-depth discussion of these models.

GCNs apply a symmetric normalization step when calculating the output of each HL, as

seen in Eq.5.2,

$$f(H^{(l)}, A) = \sigma(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}) \tag{5.2}$$

where $\hat{D}$ is the node degree matrix of $\hat{A}$ — *i.e.*, a diagonal matrix representing how many neighbors each node has — and $\hat{A}$ is the adjacency matrix with self-loops — *i.e.*, $\hat{A} = A + I$, where I is the identity matrix. In addition to Kipf and Welling [89], we refer readers to Kipf's *blog post*[4] for an intuitive and digestible walkthrough of GCNs.

GATs implement an attentional mechanism such that each node pays more "attention" to "important" neighbors than to less important neighbors. As seen in Eq. 5.3, this effectively becomes a weighted sum of the neighbors' messages, as opposed to GCN's equally weighted neighbors.

$$f(H^{(l)}, A)_i = \sum_{j \in N(i)} \alpha_{i,j} W H_j^{(l)} \tag{5.3}$$

$\alpha_{i,j}$ is the weight calculated by the graph-normalized attention mechanism in Eq. 5.4, W is a learnable weight matrix, and $N(i)$ is the set of neighbors of node i, including i itself. Although A is not explicitly used in this equation, its information is incorporated in $N(i)$.

$$\alpha_{i,j} = \frac{\exp(e_{ij}^{(l)})}{\sum_{k \in N(i)} \exp(e_{ik}^{(l)})} = \text{softmax}(e_{ij}^{(l)}) \tag{5.4}$$

$e_{ij}^{(l)}$ is the attention mechanism and, in this case, is a single layer NN calculated using Eq. 5.5.

$$e_{ij}^{(l)} = \text{LeakyReLU}(a_j W H_j^{(l)}) \tag{5.5}$$

[4] https://tkipf.github.io/graph-convolutional-networks/

In addition to Veličković *et al.* [90], we refer readers to Veličković's *blog post*[5] for an in-depth explanation of GATs.

5.4 Converting Big Simulations into Graphs

Our approach to simulating big simulations is to break the big simulation into multiple small simulations, substitute in the results for simulations, and correct the difference using a GNN. Figure 5.1 shows an overview of the classic method of simulating a big simulation by simulating the entire domain at once on the left and our approach of breaking the big simulation into small simulations and adjusting the output afterwards on the right. However, converting simulations to graphs is not trivial.

In general, GNNs can use node, edge, and global features, as we discuss in §5.3, as inputs. Giving the model an idea of how the flows within a given geometry could behave is the most informative input we could conceive; as such, we want to encode what we later dub basis simulations into the inputs. We consider other types of inputs, such as the BCs and geometry, as mentioned in §2.4.1; however, that information is already implicitly encoded in the basis simulation results.

The degree to which a big simulation should be decomposed is not necessarily obvious. In this chapter, each big simulation is composed of a 4×4 grid of basis simulations, as described in §5.5.1 and §5.5.2; as such, we know the relevant and appropriate decomposition. We discuss potential results of different decomposition scales in §5.8 and recognize that this is an avenue for future work. We generate a graph where each node is a small simulation, and each edge connects physically adjacent and connecting small simulations. In Fig. 5.1, we show an example big simulation that we are trying to run. The right-hand path shows the big simulation divided into

[5] https://petar-v.com/GAT/

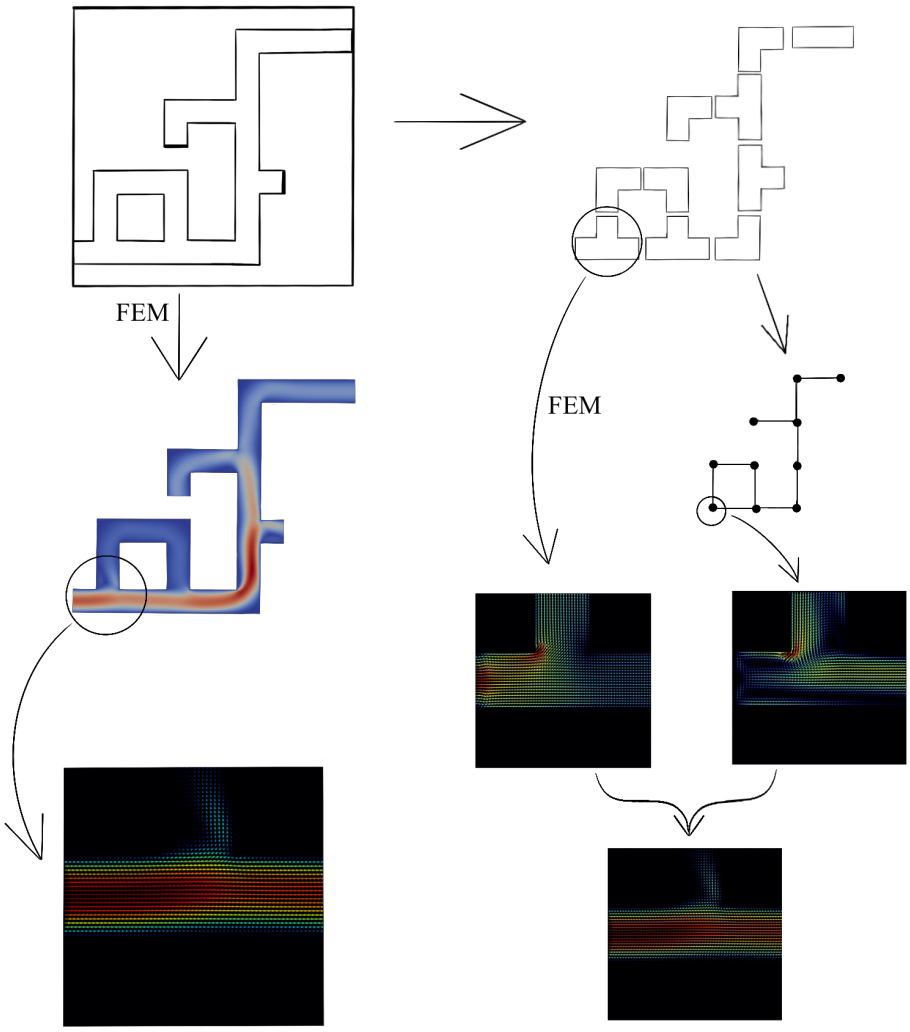


Figure 5.1: Classical (left) and our (right) method of running big simulations.

small simulations and the corresponding graph that these small simulations make. The predicted behavior, *i.e.*, the sum of the small simulation behavior and the correction term, are given by the GNN regressor at the bottom of the figure. We compare this output to the classical method on the left-hand side, where the entire fluid domain is meshed and simulated simultaneously.

Each node has features that represent the velocity fields of the corresponding small simulation. Specifically, we unravel the $64 \times 64 \times 2$ matrices from §5.5.2 into a vector of length 8192.

We create edges by connecting physically connected small simulations. That is, if fluid flows directly between two small simulations, or equivalently that they have touching fluid domains, we connect their corresponding nodes with an edge in our graph. However, two adjacent small simulations are not necessarily connected in the graph. In the second row, for example, we see a top-left corner adjacent to a top-right corner in the third row, but because the two do not have touching fluid domains, their corresponding nodes are not connected by an edge in the graph. Additionally, we tag each edge with an edge feature of either a 0 or 1, flagging edges that connect small simulations horizontally or vertically, respectively.

Our outputs exactly match the size of the inputs because they are a correction term added element-wise to the inputs. GNNs can predict node-level outputs, which naturally fit this type of regression problem where there are a varying number of inputs and outputs. However, our approach does not use pressure field values, either in the inputs or in the outputs. This, in part, is due to the curse of dimensionality. As more inputs/outputs are added, an exponentially increasing amount of data is needed to explore the full space of values to the same density. However, our GNN architectures already seemed to struggle with the problem only with fluid velocities; initial testing with including pressure, even with the additional available information, did not show promising results given our GNN architectures. A deeper dive into tuning the inputs, outputs, architectures, or even preprocessing are potential avenues of future work.

5.4.1 Choosing Boundary Conditions for Small Simulations

While choosing the small simulation with the correct geometry is trivial in this chapter, assigning the appropriate BCs to each small simulation is not. If we consider the T-shaped pipe

in Fig. 5.2 for example, the fluid is not guaranteed always to flow in from the left port; fluid can flow from the right as well, or from the top, or from multiple ports. We hypothesize that choosing basis simulations with BCs that most closely match the (decomposed) big simulation behavior would improve prediction of the correction term, and we test this hypothesis in §5.7.

From experience, we generally see that as distance from an inlet increases, the pressure decreases. Given that fluids generally move from areas of high pressure toward areas of low pressure, this implies that areas closer to inlets are more likely to be inlets in a given subdomain. Our proposed method, then, is to assign small simulation borders closest to inlets as inlets themselves, and elsewhere along the small simulation border as outlets. We thus calculate the distance along which water can flow from the big simulation inlet, as in Fig. 5.2, indicated by small black numbers. In this figure, the left T-shaped geometry has one port that is 0 m and two ports that are 3 m from the inlet. Thus, the port closest to the inlet is the most likely to be this small simulation's inlet, so we assign the shortest distance from the big simulation's inlet as the small simulation's inlet (*i.e.*, V_{in}) and every other port as outlets (*i.e.*, P_0). Only the port(s) with the shortest distance is the inlet; all others, even if they have differing distances, would be considered outlets in the small simulation. Additionally, the distance is measured by the distance through the pipe network that fluid can flow through, not, *e.g.*, by a Minkowski distance from the inlet.

By using this distance metric to determine BCs, some geometries, as in the right side of Fig. 5.2, will result in small simulations with ambiguous inlet locations. The T-shaped geometry on the left side has unambiguous inlets, but both ports in the L-shaped geometry on the right are equidistant from the inlet. However, based on our work from Ch. 4 and our intuition, both ports in a 2-port device cannot be inlets. Although human engineers can sometimes intuit the appropriate BCs, our distance heuristic does not provide enough information to automate such

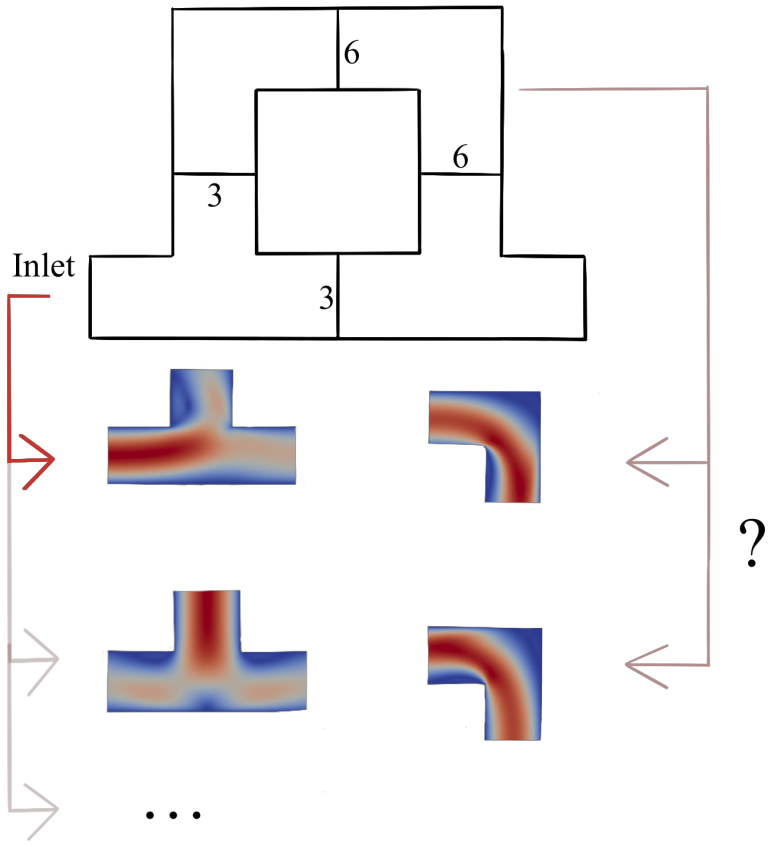


Figure 5.2: Our distance heuristic provides some guidance when guessing appropriate BCs (left), but some cases still generate ambiguous BCs (right).

intuition. For these cases, we set the velocity at every point in the domain to 1 m/s in both the x- and y-directions, which provides information about the geometry since we do not have *a priori* insight about the inlet locations based on our approach.

5.4.2 Graph Neural Network Architecture

To perform regression, we train multiple GNN architectures as implemented by the PyTorch Geometric library [113]. Specifically, we use either GCNs [89] or GATs [90]. Each architecture has either 1 or 6 HLs, each with a ReLU activation function, and either 10% or 15% of the input size number of nodes.

Since we use a 4×4 grid of small simulations, the farthest possible path between any two small simulations is 14 steps. Such a design is unlikely, though, and most big simulations are between eight and twelve small simulations long for this example problem. See Fig. 5.4 for the number of small simulations in each big simulation in our data set. We want to ensure that the message passing between nodes can reach the inner nodes, so choosing half the number of steps between small simulations — *i.e.*, six — is sufficient for any node to pass a message to either the inlet or the outlet. We compare this to a network with only one HL, which we expect is not sufficient for messages to reach the innermost nodes. Intuitively, if the network only does one round of message passing, then nodes should only be able to communicate with nodes within one step away. We expect that long-distance phenomena will need more than one round of message passing, and thus more than one HL should be used.

We show an example implementation of a GCN with one HL and a GAT with six HLs in §A.7.

5.5 Data Generation

Our data generation comes in two distinct parts: big simulations and small simulations. In the first, we discuss how we generate big simulations using a combination of basis geometries to form relatively complex networks of pipes to simulate target simulation data. In the second, we discuss how we generate small simulation results that form basis simulations for our composition. We discuss the decomposition scale in §5.8 and acknowledge that alternative methods of decomposition are an avenue for future work.

5.5.1 Big Simulations

In this chapter, big simulations are made of a combination of a 4×4 grid of small simulations, as in Fig. 5.3. The bottom left corner has the only inlet, while the top right corner has the only guaranteed outlet. We apply V_{in} to the bottom left corner. All other ports (red in Fig. 5.3) have P_0 applied, with V_0 conditions along the walls.

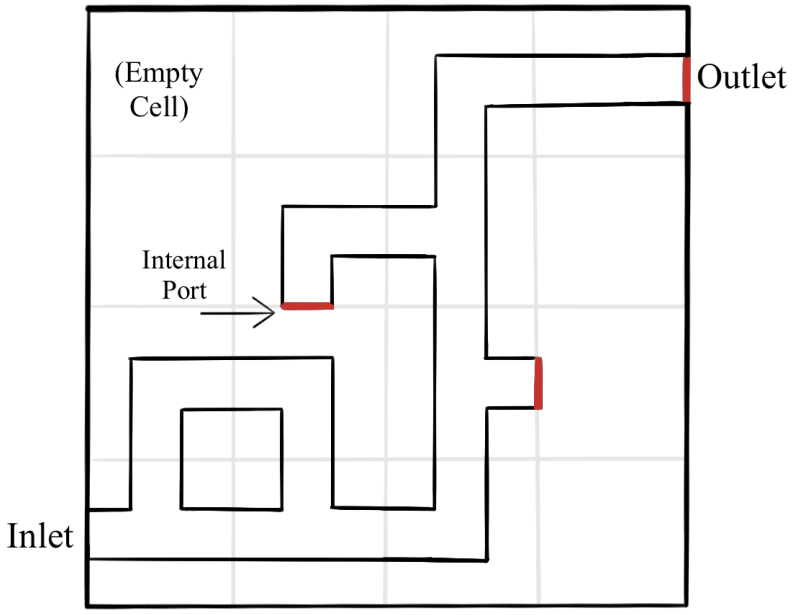


Figure 5.3: A sample potential big simulation made of a 4×4 grid of small simulations. Outlets are denoted in red.

To generate big simulation geometries, we begin with an empty 4×4 grid. We fill each small square with a randomly chosen small simulation geometry (see §5.5, and their $90°$, $180°$, and $270°$ rotations). After ensuring that the inlet and guaranteed outlet are connected, we remove any geometry that is not connected to the inlet and outlet. Each pipe has a channel width of 1 m; thus, the big simulation has a 12 m $\times$ 12 m bounding box.

We use the FEniCS library [1] to run Finite Element Method (FEM) simulation of the

Navier-Stokes equations (see §2.2 for more details). In theory, the longest route between the inlet and outlet is 42 m. Since our inlet velocity is 1 m/s, we run the simulations for a maximum of 42 seconds with time steps of 0.005 seconds. However, if the velocity field changes slowly enough — *i.e.*, the L^2 norm of the difference between corresponding $64 \times 64 \times 2$ velocity field matrices between time steps 0.5 seconds apart is less than a tolerance of 1e-4 — then we assume the simulation has reached steady state and stop the simulation at that time step. Doing so reduces the total time to run simulations while capturing the field at the steady state flow (or end of the simulation period, whichever comes first). Any big simulation that does not converge (*e.g.*, infinite velocity or pressure) is removed from our dataset.

We use a kinematic viscosity of $\nu = 0.01\ m^2/s$ which, given our geometry and BCs, gives us a Reynolds number of $Re = \frac{u*L}{\nu} = \frac{1*1}{0.01} = 100$.

We separately store the velocity fields of the last time step of each of the small domains that comprise the big simulation as $64 \times 64 \times 2$ matrices, as long as they are not empty cells. For example, Fig. 5.3 has 10 non-empty cells, so 10 $64 \times 64 \times 2$ matrices would be stored for this big simulation. This results in generally between eight and twelve matrices for each big simulation in our data set, as seen in Fig. 5.4. See §2.4.2 for more details on storing this simulation data.

5.5.2 Small Simulations

We generate small simulations by exhaustively simulating fluid flows in 2-, 3-, and 4-port devices with basis geometries, shown in Fig. 5.5, using the Navier-Stokes equations (see §2.2.3). We include rotations of the basis geometries as well, resulting in 2 "straight" configurations, 4 "corner" configurations, 4 "T" configurations, and 1 "plus" configuration for a total of 11

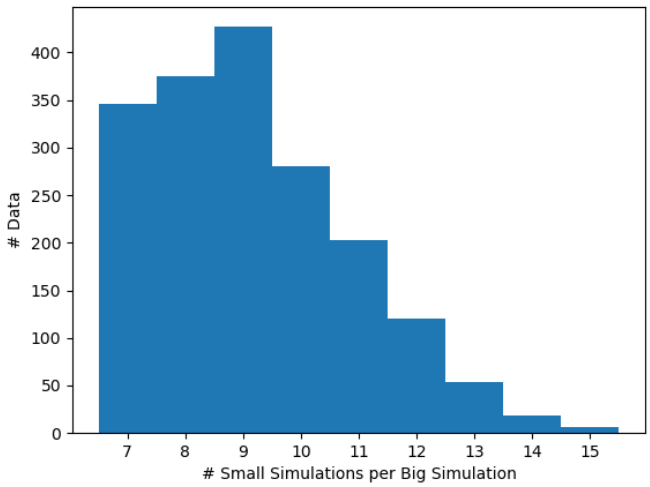


Figure 5.4: Distribution of number of small simulations per big simulation; guaranteed to be between 7 and 16.

geometries. Similar to the big simulations, each pipe has a channel width of 1 m, thus fitting in a 3 m $\times$ 3 m bounding box.

(a) Straight. (b) "L"-type. (c) "T"-type. (d) "+"-type.

Figure 5.5: 2-, 3-, and 4-port basis geometries.

We use two types of BCs, as described in §2.2.2: V_{in} and P_0. All walls that are not a port have a V_0 condition. We exhaustively apply every possible combination of V_{in} and P_0 as long as there is at least one P_0 for each geometry. This results in 3 sets of BCs for each of the "straight" and "corner" configurations, 7 for the "T" configurations, and 15 for the "plus" configuration, for a total of 61 small simulations.

We use the FEniCS library [1] to run FEM simulations of the Navier-Stokes equations over

a 3-second interval in increments of 0.005 seconds for 2- and 3-port configurations and 0.001 seconds for 4-port configurations because of convergence issues.

We store the velocity field in the last simulated time step as a $64 \times 64 \times 2$ matrix. We again refer the reader to §2.4.2 for more details on storing this simulation data.

5.6 Experiment 1: Regression

This section details our experiments applying regression to the graph representations of our big simulations. We discuss the specific inputs, outputs and error metric for the regression, and compare the GNN approaches with Linear Regression as a baseline method.

5.6.1 Regression Setup

The inputs to our regression algorithms contain two pieces of information. The first piece is the $64 \times 64 \times 2$ matrices representing the basis simulation velocity fields from §5.5.2. The appropriate small simulation is chosen in §5.4.1. The second piece is only used in GNNs and is the graph connectivity from §5.4. This graph is often represented as a $N \times N$ adjacency matrix, though this is inefficient in sparsely connected graphs. For the PyTorch Geometric library [113], connectivity is represented as a $2 \times e$ array of indices, where e is the number of edges in the graph and each index refers to a node in the graph.

The outputs to our regression algorithms are the correction terms needed to reproduce the separated parts of the big simulations we saved in §5.5.1. In other words, each output is a $64 \times 64 \times 2$ matrix that, when added to the basis simulation's velocity field — *i.e.*, the inputs — produces the separated velocity fields of the big simulation.

We report the element-wise MSE between the predicted output — *i.e.*, the sum of the predicted correction term and the basis simulation — and the separated velocity fields of the big simulation. Additionally, we mask values outside the corresponding small simulation's domain to simplify the regression task. For example, consider a horizontal straight geometry. Only the velocity field in the horizontal middle of the small simulation are considered when calculating the MSE; the rest of the domain is excluded from the error calculations.

5.6.2 Baseline Method: Linear Regression

As a baseline comparison model, we use Linear Regression, which performs a weighted sum of the inputs to determine the outputs while minimizing the squared error. We refer readers to §2.1.1.1 for more details on this model. Unlike GNN models, Linear Regression does not use the connectivity of nodes or edge features as a predictive value; we only give it the basis simulation velocity fields and predict the corresponding correction term, regardless of the small simulation's connectivity to other small simulations. As such, rather than each of the 1,829 data points each containing approximately 9 small simulations on average, we have 16,782 data points. However, we still split these data into training and testing sets based on the big simulations; that is, all of the small simulations that would comprise a big simulation are kept together, either all in training or all in testing.

5.6.3 Results

We summarize the results of our experiments in Table 5.1. We calculate the percent error by dividing the error by the maximum velocity in either the x- or y- directions, as in Eq. 5.6.

$$\%Err = \frac{\sqrt{MSE}}{\max(\text{abs}(\overrightarrow{v}))} \tag{5.6}$$

Here, $\overrightarrow{v}$ is the vector of all fluid velocities, including both x- and y-directions.

Table 5.1: MSE for various GNN architectures.

GNN Type	# Layers	# Nodes	Train MSE	Test MSE	Train % Error	Test % Error
GCN	1	819	0.01963	0.02029	7.3	7.5
		1228	0.01972	0.02051	7.4	7.5
	6	819	0.02469	0.03225	8.2	9.4
		1228	0.02614	0.02672	8.5	8.6
GAT	1	819	0.03512	0.03552	9.8	9.9
		1228	0.03444	0.03485	9.7	9.8
	6	819	0.03512	0.03552	9.8	9.9
		1228	0.03512	0.03552	9.8	9.9
Linear Regression			**0.01025**	**0.01035**	**5.3**	**5.3**
Naïve			0.04507		11.1	

Compared to naïve simulation composition, all of our tested architectures have at least a 20% reduction to the MSE. In general, we see that the GCNs have a lower test MSE than GATs; in some cases, GATs have around a 1.75x higher MSE.

Our Test MSE is always larger than our Train MSE, as is expected for regression problems. The inverse generally indicates some error with the methods or type of model being used, *e.g.*, if a model is underfit. However, we see that the Train and Test MSE are also relatively close in most cases. Using a GCN with 6 layers and 819 nodes gives the highest discrepancy between Train and Test MSE with a value of 0.00756.

Within GCNs, we see that networks with only one HL have lower Train and Test MSEs than networks with six HLs. However, the difference between 819 nodes and 1228 nodes is generally minimal across any number of HLs or GNN types. Of the architectures that we tested, we find that using 1 GCN HL with 819 nodes per layer gives the lowest Test MSE at 0.02029. However, a network with 1 GCN HL and 1228 nodes per layer gives a similar Test MSE at 0.02051.

Finally, we find that Linear Regression has both lower Train and Test MSEs than the GNNs. Its Test MSE of 0.01035 is about half that of the best-performing GCN; equivalently, its Test percent error is about 40% lower than that of the best-performing GCN.

5.6.4 Discussion

From Table 5.1, we find that GCNs are more effective at simulation composition tasks than GATs are, though both are better than no composition at all. However, neither performs as well as Linear Regression.

It is unexpected that the discrepancy between Train MSE and Test MSE is so much higher for a 6-layer GCN with 819 nodes per layer. It is possible that the training was stopped before convergence. Alternatively, the 6-layer-819-node GCN could be overfitting to the training data, though that would be unexpected because 1-layer GCNs — which have similar Train and Test MSEs and thus are unlikely to overfit — have even smaller Train MSEs. If the 6-layer-819-node GCN were overfit, we would expect its Train MSE to be lower than the 1-layer GCNs'.

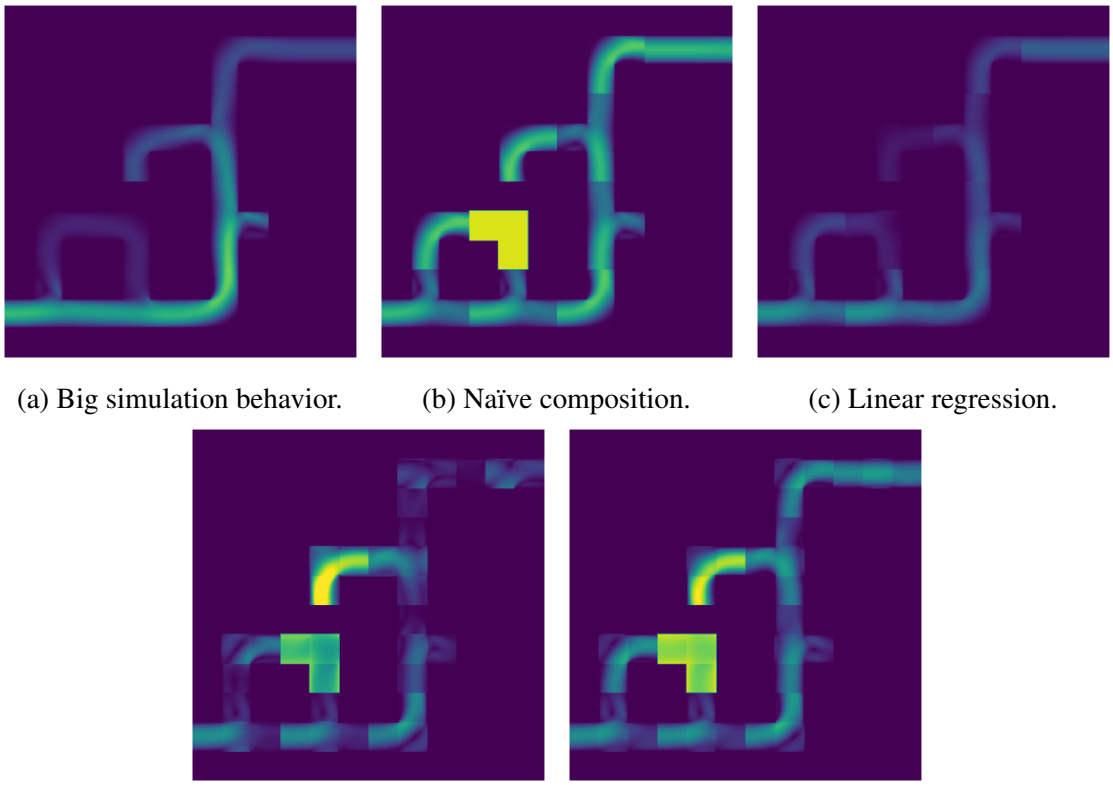
In general, we see that the GCNs with fewer layers have lower MSEs, both in training and testing. This is somewhat counterintuitive, as we expect that the depth of the graphs would require deeper networks to pass messages fully through the network. However, it is possible that

networks with one HL are complex enough because the messages do not need to be passed from the inlet through the entire pipe system. Instead, because of the geometries to which we restrict this work — which do not have dead ends or capped pipes, for example — it is possible that such deep networks are unneeded. With visual inspection of our data, we qualitatively see that, in many of our networks, the fluid often does not flow strongly throughout the entire domain and instead exits through the first few outlets it encounters, thus making the flow in later areas weaker. Consequently, the velocity field may be simpler to predict farther from the inlet. The long-distance phenomena in our dataset may only reach across one or two small simulations, thus not requiring significant message passing at all.

Additionally, the architecture of the networks may not be optimized for this type of problem. However, an in-depth architecture search is not the purpose or scope of this paper. Others who have used images as inputs to NNs, such as Gladstone *et al.* [53] and Wang, Chiu, and Fuge [3], structure their NNs such that each consequent layer is about half the number of nodes as the previous layer. In contrast, our network directly jumps from 8192 dimensions to either 819 or 1228 dimensions (about 10% and 15%, respectively) then stays constant through all of the HLs. This large jump may cause difficulties in converging the hyperparameters for future prediction, as the final output is also a 6-10x increase in dimensionality.

An additional explanation for the differences between our 1- and 6-layer GNNs is the number of training epochs. Specifically, we train all of our GNNs the same number of iterations, regardless of the number of layers. However, deeper networks may need more iterations to reach convergence due to the larger number of hyperparameters in the network. Because we train every network for the same total number of iterations, the deeper networks may not have enough iterations for convergence. In a similar vein, we only train on a few thousand data points, which

is much fewer than the design space covers. It is possible that increasing the number of data points would also increase the generalizability and predictive power with more training. Figure 5.6 shows clear sharp borders between small simulations and even within small simulations, thus furthering our belief that the models have not reached convergence yet.



(a) Big simulation behavior. (b) Naïve composition. (c) Linear regression.

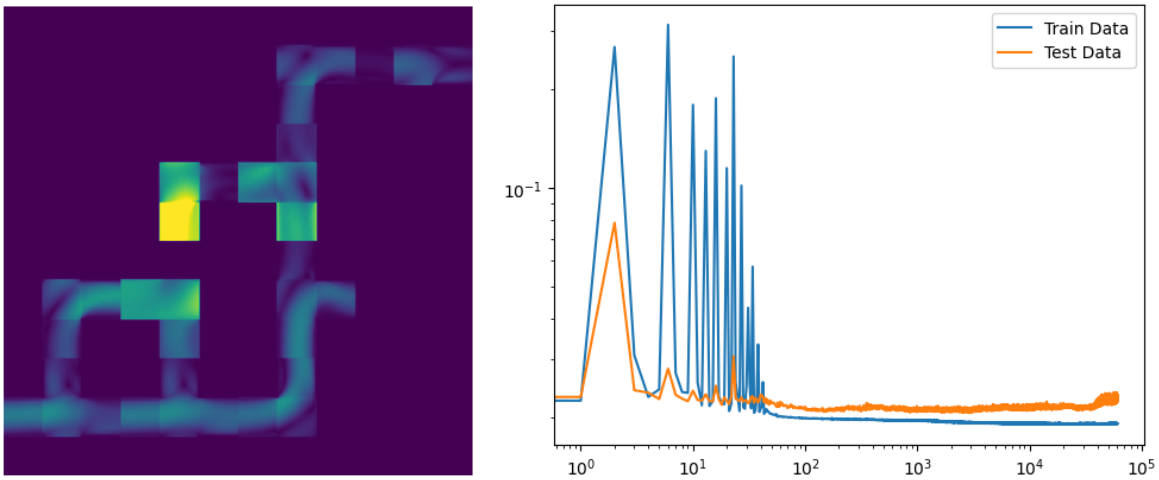
(d) GCN, 1 layer, 819 nodes. (e) GAT, 6 layers, 1228 nodes.

Figure 5.6: Sample composition of a subset of regression algorithms.

Finally, we see that Linear Regression has a lower MSE than any of the GNNs. Despite having no sense of surrounding simulation behaviors, Linear Regression is still able to predict the correction term. This may also be a result of the fluid flowing only through the first few small simulations and dwindling farther in the pipe network. As such, a strong baseline guess for the correction values could be just reducing the velocities of the basis simulation, which would be easy for an algorithm like Linear Regression to discover. In fact, we do see this behavior in

Fig. 5.6c, where the magnitude of the fluid flow is effectively damped to smaller values but left as-is otherwise.

We examine the convergence of the 1-layer-819-node network in Fig. 5.7 on a log-log scale. However, since we see the training error and testing error — in blue and orange, respectively — reach fairly steady values, we do not see evidence that this is a convergence issue. In fact, we surmise that the model is overfit in this particular case and that more robust methods to set convergence criteria would improve this model. These plots suggest that the issue may be the architecture, as we discuss above, rather than the convergence of the models.



(a) Predicted output.

(b) Convergence testing on a log-log scale.

Figure 5.7: Training a GCN, 1 layer, 819 nodes with 1e8 training iterations.

5.7 Experiment 2: Effect of Small Simulation Boundary Conditions

To separate the effect of the BC selection heuristic from the regression algorithms, we consider the case where we choose other sets of BCs for the small simulations. We run a subset of the experiments summarized in §5.6.3, but we rather than using our heuristic to choose small simulation BCs, we explicitly choose other BCs. For example, if a small simulation is of a

horizontal pipe and our heuristic selects a flow from left to right, then in this experiment, we use a flow from right to left for this small simulation. We choose the two GNN architectures with the lowest Train MSEs, Linear Regression, and Naïve composition in this experiment; we do not expect any of these results to change drastically across GNN architectures. Other than the choice of small simulation BCs and only using a subset of the GNN architectures, all other aspects of the regression process remain identical to §5.6.3.

We summarize the results of this experiment in Table 5.2, with corresponding results from Table 5.1 replicated for convenience.

Table 5.2: MSE for various GNN architectures, comparing application of our heuristic for choosing BCs.

GNN Type	**# Layers**	**# Nodes/Layer**	**With Our Heuristic**		**Without Our Heuristic**	
			Train % Error	**Test % Error**	**Train % Error**	**Test % Error**
GCN	1	819	7.3	8.9	12.7	12.8
		1228	7.4	7.5	12.7	12.8
Linear Regression			**5.3**	**5.3**	**5.6**	**5.6**
Naïve			11.1		22.2	

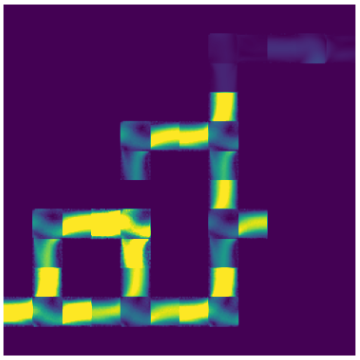
Compared to the corresponding results in Table 5.1, we see that using our heuristic lowers the percent error by about 74%. This is most apparent in the Naïve composition, where there is a 2x decrease in the percent error when our heuristic is used. However, the Train and Test percent errors for all of the tested GNN architectures also show between a 43-74% increase without our heuristic. Because Naïve composition shows a higher increase in error than GNNs, the GNNs are an improvement over Naïve composition, though the misleading BCs still confuse GNNs to a significant degree. This is a strong case that our BC choosing heuristic is helpful in this type of problem, especially with GNNs.

Unexpectedly, Linear Regression's Test MSE only increases by 5.7% when our heuristic is not used, and thus still has the lowest Test percent error in this experiment. We believe this

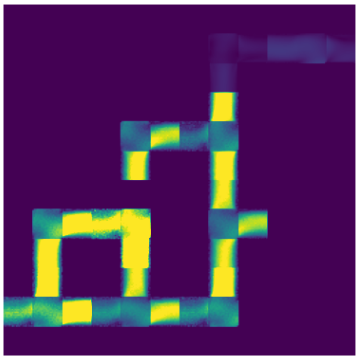
(a) Big simulation behavior.

(b) Naïve composition.

(c) Linear regression.



(d) GCN, 1 layer, 819 nodes.



(e) GAT, 6 layers, 1228 nodes.

Figure 5.8: Sample composition of a subset of regression algorithms. Note that the color scale is four times larger for visual clarity.

strengthens our case that the physics of the problem are such that large portions of the behavior are low velocity flows, and implies that simpler models like Linear Regression are less easily "confused" by incorrect BCs while more complex models, whether they perform well with the correct BCs, are more easily "confused." That said, our BC choosing heuristic still improves Linear Regression's performance on this task.

5.8 Discussion: Effect of Decomposition Scale

In this section, we discuss what we hypothesize would happen by varying the scale of the decomposition. We do not present any experimental data, but instead, base our discussion on

intuition in case it helps guide future work on this problem. One of the biggest limitations of this work is that we choose by fiat the scale of the decomposition. As we state in §5.5.1 and is implied in Fig. 5.3, each big simulation has a 12 m $\times$ 12 m bounding box. In this chapter, we decompose this to a 4×4 grid of 3 m $\times$ 3 m squares, in large part due to how we generate the small simulations in the first place. However, we are not implying that this is the best or only way to decompose this simulation; on the contrary, we expect this to be an interesting direction of research, possibly drawing from past work on automated meshing (*e.g.*, see §3.2) or hierarchical models (*e.g.*, see §5.2.1).

To take the decomposition to one extreme, we consider how basis simulations compare as they approach the size of the big simulation. If the basis simulation is exactly the same size of the big simulation, our method would produce a graph that consists of a singular node, so the "graph" part of our work is effectively irrelevant. However, to maintain performance, a larger variety of geometries and BCs would need to be run. Our current basis geometries are designed to replicate any big simulation geometry in our design space exactly, but combinatorically more basis geometries are needed as they approach the size of the big geometry to reproduce big geometries. If the geometries do not require perfect reproduction from the basis geometries, then fewer basis geometries are needed, but this makes the regression problem harder because the regression algorithm would have less information about the big simulation. BCs follow similar reasoning: bigger basis geometries means that either more BCs are needed to continue matching the big simulations, or the regression problem becomes more difficult. Alternatively, the regression problem can be simplified by simulating more varieties of BCs, but this comes at the risk of not matching the big simulation's BCs as closely and the cost of more manual setup, such as with BC locations. The combined effect of having basis simulations the same size as big simulations

is likely that the error is similar to that of non-graph-based models that directly predict simulation results such as, *e.g.*, [72, 116]. The regression problem is made more difficult by the basis geometries and BCs not matching the big simulation as closely, but at the same time, it is made easier by the GNN not needing to approximate physical laws as much. This tradeoff is likely affected significantly by the architecture of the GNN; that is, architectures that are effective at manipulating geometries and BCs, but not at approximating physical laws, would likely perform better when the basis simulations approach the size of the big simulation.

Taking the decomposition to the opposing extreme, we consider how basis simulations compare when they become extremely small. Taken to the extreme, basis simulations become like pixels in the domain; if these pixels are squares, then the only geometry would be a small square with fluid entering or leaving on each of its sides. While this simplifies generating the basis simulations significantly, this complicates the graph. Our work in this chapter do not explore whether including regions outside the fluid domain help or hinder the regression. In other words, we are unsure if including regions where there is no fluid improves performance in this type of problem. Currently, we do not include entirely empty cells in the GNNs; however, including these nodes may provide some information about distances between nodes that can help the regression problem. An exploration of the inputs to the GNNs may be a direction of future work. In any case, having very small basis geometries leads to closer approximations of the big geometry, especially when the big geometry is not made directly from the basis geometries, as we did in this chapter. With sufficiently small basis geometries, almost any arbitrary big geometry can be recreated, with the degree only limited effectively by the resolution of the basis geometries. Additionally, it is possible that small basis simulations can more closely reproduce the BCs of the big simulation as well. Modifying the possible BCs to remove points with no

flow may be helpful to improve matching the big simulations' BCs to avoid multiplication by 0 in the GNN. While smaller basis geometries do give better reproductions of big geometries, they also increase the order and size (*i.e.*, number of nodes and number of edges, respectively) of the corresponding graph. We note that, by reframing the problem slightly with specific basis simulations, GNN architectures, and squinting a bit, using smaller basis simulations begins to look similar to a FEM simulation. We believe that such an analogue would have each node as a cell in the mesh, and each edge a connection between adjacent cells. In such a framework, we expect that decreasing the size of the basis simulations would decrease the MSE, similar to how finer FEM meshes tend toward the "true" solution. However, similar to FEM, we expect that smaller basis simulations increase computational time (*e.g.*, to train the GNN), and similar to bigger basis simulations in the preceding paragraph, this approach effectively becomes the current method of simulating the big simulation in its entirety, but through a GNN rather than through basis simulations. If we can show that both extremes of basis simulation sizes — *i.e.*, where basis simulations approach the size of big simulations and where basis simulations become much smaller than the big simulation — then we can deduce that there is some size of basis simulation where the GNN is computationally cheapest and able to predict the error that arises from composing basis simulations. One one side, the ideal basis simulation size would minimize the computational cost of big basis simulations and the prediction error associated with inaccurate BCs and geometries. Simultaneously, this basis simulation size would minimize the computational cost of training on extensive graphs and deep GNNs as well as the computational error of many different compositions.

5.9 Conclusion

In this chapter, we expanded on our previous work in [3] by generalizing the compositional method to larger and more complex networks of pipes. To this end, we proposed a heuristic for assigning appropriate BCs to small simulations, as well as comparing various GNN architectures using GCNs and GATs. Specifically, we use the distance from the inlet of each small simulation's ports to determine which of the ports are inlets and outlets in the decomposed small simulations and show that this heuristic improves regression performance, though the effect is less pronounced on some types of regression models than others. Additionally, we find that, of the architectures we explore, GCNs with one HL are the most effective of the architectures we tested in predicting correction terms for composing small simulations into big simulations, but that Linear Regression is both cheaper and gives a lower MSE, which we infer results from how we set up our simulations.

However, we believe that the results shown in this chapter can be improved. Because previous work (*e.g.*, [3, 53]) have NN architectures that more slowly reduce the side of each HL, a more in-depth architecture search is likely able to improve on the results we show here. The increased complexity of our model and problem likely requires a subsequent increase in the amount of data used in training as well. The classes of geometries explored in this work are also specifically limited for the scope of this chapter; however, an even larger, more descriptive set of basis shapes to describe more complex big simulations is an avenue to explore. Doing so also opens the door to hierarchical versions of composition, wherein one model composes basis simulations together, another model composes compositions of basis simulations, another model composes different big simulations with different types of physics, *etc.*.

We began this chapter by asking how we can encapsulate the emergence of complex behaviors from interactions between different components. In this process, we concentrated on the specific example of fluid simulations, which are often complex and which we know, from prior experience, can show long-distance and emergent phenomena. We believe that problems that are "easier" are thus also within the scope of this chapter. Many factors go into how "easy" or "hard" a given problem is, but generally, factors that make the behavior more complex or chaotic make the prediction task more difficult. For example, fluids with larger length scales can develop turbulent flows, which are chaotic and difficult to predict. Decreasing viscosity or increasing velocity, both of which also have the effect of increasing the Reynolds number, likely make the problem more difficult as well. However, slow-moving dynamics, such as Stokes flows, should be easier to predict. Similarly, Poisson simulations, such as electrostatic potential, should also be easier to predict. Some physics, like fluids, can encompass a variety of difficulties: continuum mechanics ranges from small deformations, which should be relatively easy to predict, to beam buckling, which may be much harder. We also expect that increasing the scale of a simulation, whether that's through finer decompositions that increase the number of nodes in our decomposition graph, larger domains, or increasing the time scale likely all hinder our ability to predict behavior accurately. However, problems that are on a smaller scale should all be within the same difficulty scale for our contributions.

As we claimed in §5.1 and as we show in §5.6.4 and §5.7, this chapter advances our initial research question by showing GNN performance on predicting emergent behaviors in fluid simulations. We hope that this chapter, combined with Chs. 3 and 4, informs future research in using GNNs to capture the interactions across composed simulations across a variety of physics. In Ch. 6, we assume that complex simulations, eventually including ones this chapter aspires to

tackle and more, can be set up and run automatically for the purposes of generating data on which we train our ML models.

Chapter 6: Discovering Devices from Computer-Driven Simulations

This chapter explores the third question posed in the introduction: **Under what conditions can we detect novel device behaviors through computer-driven simulation and exploration?** An exhaustive search over all possible behaviors and physics is well beyond the scope of this chapter, so here, we focus specifically on simple static fluidic devices with a small set of boundary conditions (BC); we discuss generalizing to other types of physics in §6.7. For this problem, this chapter proposes a method for identifying groups of devices with similar behaviors, which presumably coincide with devices with similar functions, from simulations of myriad random devices; we discuss the difference between behaviors and functions in §6.7. We explore methods to represent our device behavior more compactly than current common practice, propose a modification to a popular clustering algorithm, and compare various clustering algorithms to find these groups of devices.

6.1 Introduction

One output of engineering design is to manipulate geometry and material in space-time (often called a device's *structure*) to modify something about nature's current state via physical fields (*i.e.*, a device's *behavior*) in service of benefiting humanity in a specific way (*i.e.*, a device's *function*). For example, devices called logic gates all perform the same useful *func-*

tion — *e.g.*, performing a Boolean operations (such as AND, OR, and XOR) between one or more inputs. Logic gates can vary wildly in their physical *behavior*, however, while achieving this same *function*. For example, while familiar AND gates (*e.g.*, using MOSFETs) manipulate electrostatic fields to alter a material's conductivity, there are a wide range of other physical behaviors that can produce this same function: (1) *fluidic* logic gates can use dye flowing into different channels [117], pressure-activated gates [118], or even bubble-based gates [119] to create fluidic circuits used in sensor or lab-on-a-chip applications; (2) *photonic* gates can guide waves through crystals [120]; (3) *chemical* diffusion can drive AND behavior in synthetic biology circuits [121]; and (4) *trapped ions* [122] or *laser* pulses [123] can combine to form Toffoli gates, useful for higher qubit quantum circuits.

In principle, all of these devices can have the same function — performing Boolean operations — though they implement that function via drastically different physical behaviors. This chapter addresses how one might discover a variety of similarly functioning devices automatically. We seek algorithms that can identify "interesting" physical behaviors that emerge from a set of physics and a given design space, but without requiring human guidance, intervention, or explicit direction regarding what behaviors or functions to look for. For example, without telling an algorithm that we want something that acts like a logic gate, can it nevertheless automatically discover the existence of such devices by only directly observing the physical behavior of various designs? Can it derive novel families of devices that perform functions that the algorithm does not know about *a priori*? Answering such questions would enable the automated discovery of engineered components for new physics regimes where humans have yet to explore rigorously or even via trial-and-error — *e.g.*, in spacecraft components in extreme environments where existing devices break down such as in landers on Venus [124] or hyper-sonic flight regimes, or in adapt-

ing engineering design methods to novel domains like synthetic biology or quantum computing.

This chapter addresses such concerns by studying two inter-related questions: (1) how do we discover groups of designs whose behavior is "similar enough" to constitute a family of related functions, and (2) how do we compactly represent a design's behavior? This chapter's key idea is that by combining large-scale, automated simulation of different physics across (appropriately randomized) design spaces with new ways of preprocessing and clustering the resulting simulation data, we can automatically identify unique functional families of devices. Specifically, the key contributions of this chapter are:

1. We modify the Hierarchical Density-based Spatial Clustering of Applications with Noise (HDBSCAN) algorithm's persistence measure and use the Silhouette Score (SS) [20] to select hyperparameters. This method, called the Silhouette-modified HDBSCAN (SHDBSCAN), generally outperforms its predecessor algorithms on behavior clustering tasks for simple fluidic devices, but we find that KMeans matches or outperforms density-based algorithms in general.

2. We propose a heuristic to reduce dimensionality of behavior vectors to provide better clustering of simple fluidic devices than un-preprocessed data when applied to simulation data of these devices. We compare various preprocessing steps through adjusted Rand scores on our tested data and find that the type of behavior (*e.g.*, diodes vs. logic gates) has a higher impact on the clustering performance than the preprocessing.

To demonstrate the value of these contributions, we show that we group fluidic devices with similar behavior together without explicitly stating objectives or optimizing for such behavior. Stated more plainly, interesting logic-gate behavior "falls out naturally" from our above behavior

representation and clustering, even though we do not explicitly instruct the algorithm to find those specific behaviors.

We simulate the fluidic devices in this chapter using Chorin's method — an iterative solving scheme — for the Navier-Stokes equations, as discussed in §2.2.3. We discuss how these methods may be generalized in §6.7.

Building off our work in the previous chapters, we assume in Ch. 6 that we have simulation results for any device we want to simulate. That is, we assume we have simulation inputs (*e.g.*, BCs, governing equations, meshes, mesh resolution, *etc.*) and simulation outputs (*e.g.*, SF values, such as fluid velocity, fluid pressure, and electrostatic potential), and use both the simulation inputs and/or simulation outputs as *inputs* to the *machine learning (ML) models* used in Ch. 6. We call these inputs to our ML model *behavior vectors* (BVec), as they capture device behavior only on the edges of the domain, rather than throughout the domain. We elaborate on the reasoning of this choice in §2.3.3, but briefly, we hypothesize that, since fields are continuous phenomena that only interact with their surroundings through their borders, using only their borders as inputs to our model should be sufficient to capture their interactions to their surroundings fully. Because this chapter focuses on finding groups of devices, we framed this problem as a clustering problem; thus, our outputs are cluster labels for each device, which usually take the form of an integer.

Although our methods in this chapter may discover new behaviors, we cannot quantitatively measure the presence of these new behaviors. Instead, we can only compare our findings against known ground truths. We detail our approach to calculating our "True Labels" in §6.5.2.

6.2 Related Work

One question we explore is what it means for devices to have a *function*. Intuitively, a group of devices that all perform the same *function* should be close to each other in some abstract *behavior space*. Assuming we have an appropriate representation of device behaviors, we want to automate identifying these groups of devices.

In ML, this type of problem is called *clustering*. Clustering algorithms identify groups of similar items without knowing *a priori* data membership labels. Although a comprehensive review of clustering algorithms is not in the scope of this chapter, we have previously found that the specific clustering algorithm does not significantly affect our results [77]. Centroid-based clustering, as exemplified by KMeans clustering [14], iteratively shifts representative vector and reassigns cluster membership. Density-based clustering, as exemplified by Density-based Spatial Clustering of Applications with Noise (DBSCAN)[1] [15], crawls through the space and labels regions of dense points as clusters. An extension of DBSCAN, called HDBSCAN [17], combines DBSCAN with connectivity-based strategies.

Generally, we use clustering to find representatives of a group. For example, Zhang *et al.* [41] use hierarchical clustering to group design ideas to discover new functions, while Ahmed, Fuge, and Gorbunov [43] use Spectral Clustering to find a diverse — yet high-quality — group of representative design ideas. Park and Kim [42] come from the consumers' side by analyzing what consumers want in a device to extract a set of desired features using HDBSCAN and Spectral Clustering. However, it is possible to use clustering to discover voids in the design space as well [33]. In this chapter, we compare KMeans, DBSCAN, HDBSCAN, and a new clustering algorithm we

[1] DBSCAN is the predecessor to HDBSCAN, which inspired SHDBSCAN.

call SHDBSCAN to identify groups of devices. Unlike others who search for a specific function with, *e.g.*, genetic algorithms, as in [125], our work is more similar to, *e.g.*, [126, 127, 128], where function is deduced from physical interactions, simulated interactions, and videos, respectively. However, our work differs in that we use clustering on randomized objects, rather than the pre-existing objects that other research has used.

6.3 Proposed Clustering Algorithm: Silhouette-optimized HDBSCAN

We treat function identification as essentially a clustering problem. That is, we expect that different devices with similar behaviors exist and that grouping those similar behaviors will identify the underlying functions those devices perform. Preliminary results with off-the-shelf clustering algorithms either resulted in clusters of unrelated devices — while related ones were separated into different clusters — or numerous tiny clusters — some including only single devices. Many of these tiny clusters included similar behaviors; as such, we posited that modifying an existing algorithm to force these small clusters to stay together could improve the clustering results. Density-based algorithms, such as HDBSCAN, seemed promising from our initial results. Although it produced many small clusters, HDBSCAN did not usually cluster together unrelated devices.

The original HDBSCAN algorithm [17] has the following high-level steps:

1. Compute the mutual reachability metric, in effect spreading out regions of low density and compressing regions of high density.

2. Build the minimum spanning tree of the dataset using, *e.g.*, Prim's algorithm [129].

3. Build a hierarchy of how clusters are connected, merging smaller clusters into larger ones

as the hierarchy goes up.

4. Condense the cluster tree such that it represents large, persistent clusters that lose points, splitting only when the subsequent clusters are larger than the minimum cluster size.

5. Finally, extract the longest-lived and most persistent clusters by a stability metric, keeping a cluster when its stability is greater than that of its children.

We modify the HDBSCAN algorithm in two ways. First, we change its measure of cluster persistence to a new measure governed by a hyperparameter called the *maximum cluster size*. Secondly, to select this hyperparameter, we select the value that maximizes the SS, which measures the inter- and intra-cluster distances to approximate how well-separated and tightly clustered the resulting clusters are. We call our approach the Silhouette-optimized HDBSCAN, or SHDBSCAN.

First, we consider some value c that we denote the *maximum cluster size*. We start at the root of the cluster tree, representing a clustering where all points are in one cluster. We can thus step down through the cluster tree, clustering our data into smaller and smaller clusters without dropping any data from our clusters. Figure 6.1 demonstrates how we step down a single-linkage tree for a pedagogically illustrative example.

At each node, we calculate the size of that cluster by counting the number of leaves in the subtree rooted at that node; if that cluster size is less than c, we stop at that node and label all leaves of the subtree to be in the same cluster. We then move onto the next node with unlabelled leaves and repeat the process. We continue this process until all points are labeled. This is a different measure of "cluster persistence" than is used in HDBSCAN. However, since our proposed persistence-modified HDBSCAN depends on an appropriate value for c, our algorithm needs to

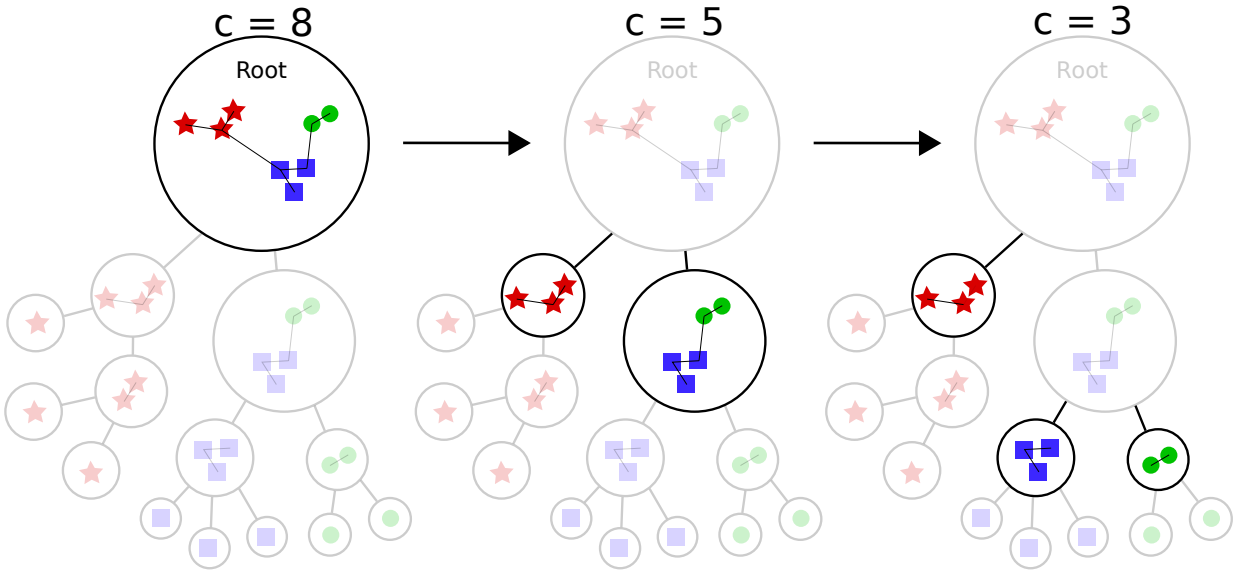


Figure 6.1: At the root node, all nodes are in the same cluster. As we set c smaller, we work our way down through the clusters. By setting $c = 8$, we end our clustering at the root node. If we set $c = 5$, we move down to the next level, where each cluster includes at most five nodes. Setting $c = 3$, the left-side cluster is smaller than c, but the right-side cluster must be split once more.

choose this value.

To choose the appropriate max cluster size c, we use the SS. Specifically, we run the persistence-modified HDBSCAN for every integer between $c = 1$ (where every data point is its own cluster) to $c = N$ (where N is the number of points in our dataset, representing all points in a single cluster), calculating the SS for each value of c. We then set c to the value that maximizes the SS. Algorithm 5 shows the pseudocode for this algorithm, and §A.8 shows one implementation of this method.

With SHDBSCAN thus defined, the usage is similar to that of other Scikit-learn models.

```
# Assuming that X, a matrix containing the data,
# of the shape (numSamples, numFeatures) is defined.
# Also assuming that yTrue, a vector of size
# numSamples that contains the true labels, is defined.
```

Algorithm 5 Pseudocode for Fitting SHDBSCAN

```
Fit HDBSCAN to generate useful intermediate objects
Extract the HDBSCAN condensed tree as a numpy array
Convert this numpy array to a tree where each key is a label and the values are arrays of keys of that node's children
Find roots of the tree by checking if every key is the child of another key
Build an accumulated tree that counts how many leaf nodes each node has
Build a flat tree that contains the leaf nodes, as an array of values, of every node
for each possible value of c do
    Use the accumulated tree to find nodes with at most c leaves
    Give each leaf a corresponding label using the flat tree
    Calculate the SS of the clustering with the given labels
end for
Return the clustering with the highest SS
```

```
import shdbscan

clusterer = shdbscan.SHDBSCAN(verbose=True, levels=20)

yPred = clusterer.fit_predict(X)
```

`yPred` can now be compared to the true labels, `yTrue`, via a scoring metric, as in §2.1.1.6. We note that, in this implementation, there is no `clusterer.predict(xTest)` for new data; as such, predicting on out-of-sample data would require using, *e.g.*, a K-Nearest Neighbors Classifier, as in §2.1.2.4.

6.4 Proposed Dimensionality Reduction Heuristic: Resolution Selection

We detailed the resolution selection preprocessing in §2.3.3.2. To briefly summarize it again here, we use the reciprocal of the maximum difference between adjacent points to calculate the degree to which we can shorten BVecs. We refer readers to §2.3.3.2 for details on this heuristic and two examples of the resolution preprocessing step: one where the BVecs are not shortened

($m^* = 1$), and one where they are ($m^* = 2$).

6.5 Behavior Clustering on Fluidic Devices

This experiment evaluates our SHDBSCAN algorithm and preprocessing steps against several baseline clustering algorithms on two- and four-port fluidic devices with calculated true labels. In summary, we take the following broad steps:

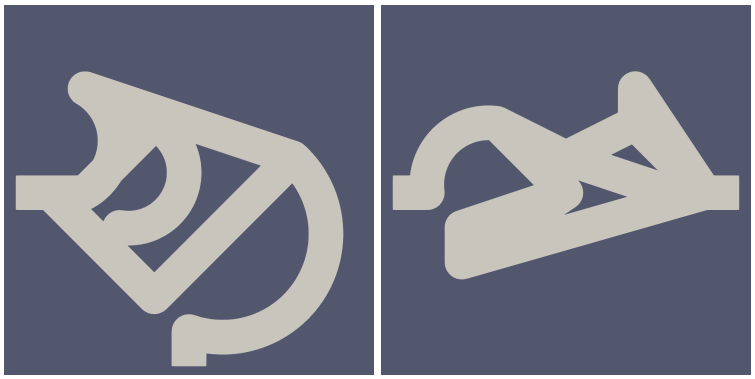
1. Data Generation, forming the inputs to our clustering algorithm.
   (a) Run Navier-Stokes simulations, using meshes from §2.2.1 and equations from §2.2.3.
   (b) Generate BVecs using methods discussed in §2.4.2.
2. Calculate True Labels, forming the outputs to our clustering algorithm.
3. Run the clustering algorithms and compare the algorithm's outputs to the True Labels.

6.5.1 Data Generation

To generate data for clustering, we run fluid simulations, create BVecs from those simulations, label the BVecs, then run clustering algorithms on the BVecs. This allows us to compare the performance of various clustering algorithms against a common metric.

6.5.1.1 Geometry Generation

We use methods in §2.2.1 to generate geometries for our simulations in this chapter. We use a variety of port locations in our mesh generation that cannot be mimicked through rotations



(a) “2A”-style fluid device (b) “2B”-style fluid device

Figure 6.2: Example fluid domains for two types of 2-port devices.

or flipping. Ports are located at ratios of 0.3, 0.5, or 0.7 the distance along the bounding box. Figures 6.2 and 6.3 gives one example mesh with each type of port locations.

6.5.1.2 Boundary Conditions

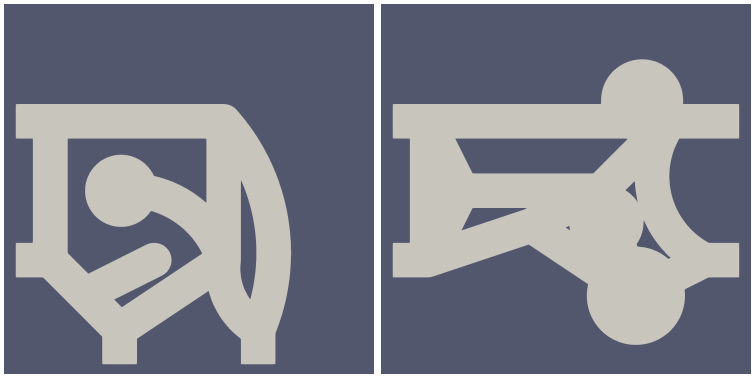
On each of the ports on our meshes, we can apply three types of BCs: no slip, or effectively blocking up the port; velocity coming in, with a given profile and magnitude; or zero pressure, or effectively leaving the port open (see §2.2.2). We combinatorially apply all three of these BCs in every simulation as long as there is at least one velocity inlet and one zero pressure port in the simulation.

Velocity at inlets is defined with a parabolic profile perpendicular to the port. On each side of the port, the velocity is 0 m/s; in the middle, it is 0.1 m/s. These are denoted as V_{in} in this chapter.

No-slip conditions are velocity conditions where the velocity is set to 0. These are denoted ⊘ in this chapter. On all of the walls that are not a port, we apply a no-slip condition.

We use the FEniCS library [1] to run Chorin’s method [29] to solve Navier-Stokes simula-

(a) “4A”-style fluid device (b) “4B”-style fluid device (c) “4C”-style fluid device



(d) “4D”-style fluid device (e) “4E”-style fluid device

Figure 6.3: Example fluid domains for five types of 4-port devices.

tions using the Finite Element Method. If, during the time stepping of these simulations, either the velocity or pressure exceeds $100\,\mathrm{m/s}$ or $1e6\,\mathrm{Pa}$, respectively, we consider the simulation to have diverged and remove it from our data set. Divergence typically occurs when sharp changes in geometry cause rapidly changing velocity fields, leading to unstable solution fields (SF).

For fluid flows, the Reynolds number is defined as:

$$Re = \frac{\rho u D_h}{\mu} \tag{6.1}$$

For our simulations, we set $\rho = 1\,\mathrm{kg/m^3}$, $u = 0.1\,\mathrm{m/s}$, and $\mu = 0.01\,\mathrm{kg/(m \cdot s)}$. The hydraulic diameter, D_h, is approximated as for a square duct; *i.e.*, we set it to the width of the pipe: $D_h = 0.1\,\mathrm{m}$. This gives us a Reynolds number of $Re = 1$.

6.5.1.3 Simulation and Behavior Vector Generation

During each time step in the simulation, we calculate the BVec and compare it to the previous time step's BVec. If the maximum relative difference between the two is less than 0.001, we assume the simulation has reached steady state and store the BVec from the last simulated time step. Otherwise, we let the simulation run until it has simulated one second of real time and store the BVec from the last time step.

Many useful behaviors do not arise from simulating a single BC but rather as differences between simulating multiple sets of BCs. Combinatorially selecting all possible sets of BCs results in too many potential BVecs, thus obscuring the potential devices that may exist in our design space. As a consequence, randomly choosing a subset of these BC sets also does not give a data set that has sufficiently many "interesting" devices; they end up being too rare to find. We

attempted methods such as Maximal Marginal Relevance to choose sets of BCs efficiently, but with limited results. In this chapter, we choose sets of BCs that are valid for various types of devices, *e.g.*, choosing forward and backward flows to test for diodes, or "(0, 0), (0, 1), (1, 0), (1, 1)" inputs for testing logic gates. We elaborate on these sets more in §6.5.2 for each device separately.

6.5.2 Calculating True Labels

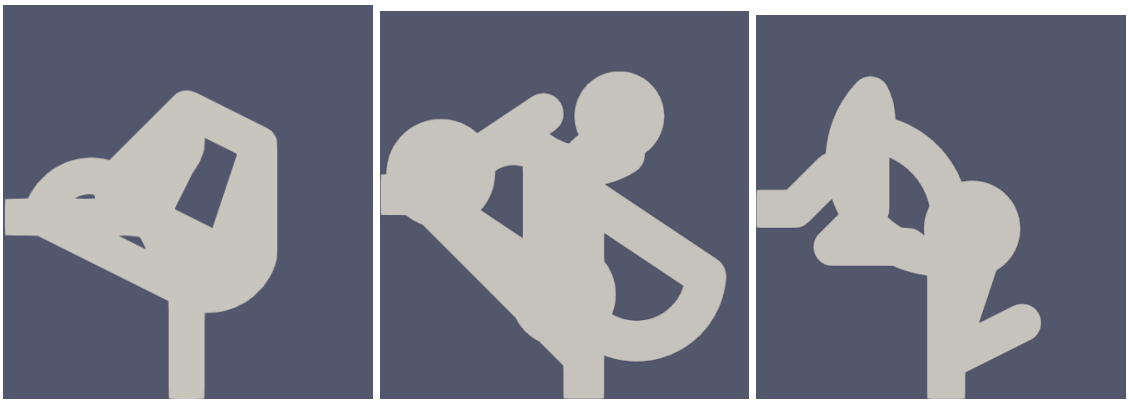
With the BCs we set, we would normally expect to find three potential types of devices if a human were to review the device behaviors manually: Diodes, Velocity-based Logic Gates, and Flow Splitters.

To measure the performance of our clustering algorithms, we need ground truth labels against which to compare using the adjusted Rand score. Our simulations do not give us inherent true labels. Instead, we assign our best guess of true labels by projecting each device into a *performance space* and grouping them by their performance as various known devices. For example, we can measure the head loss in the forward and backward directions of any two-port device and find its diodicity. This gives us a measure of how diode-like a device is. However, it is not always obvious how "diode-like" is sufficient to be labeled a diode. It may be that a diodicity of 1.5 is sufficient for some types of diodes—though not for others—depending on the type of physics. For example, electric diodes tend to have much higher diodicities than fluidic diodes; a cutoff diodicity would vary based on the type of physics.

Rather than setting differing cutoffs, we use a Gaussian Mixture Model to generate the appropriate labels. We project each device into a performance space for the possible devices

we could find, then cluster using our Gaussian Mixture Model. We use the theoretical maximum number of devices that can exist in each performance space as the number of mixtures in our model, and we optimize the model parameters using the Expectation Maximization algorithm [130].

We include some example devices with their calculated true labels in Figs. 6.4, 6.5, 6.6, and 6.7. The labels' values for the diode clustering do not have inherent meaning; any of them may mean forward diodes, backward diodes, or non-diodes. Consequently, permutations of these values may correspond to identical clustering. For example, a set of labels [0, 1, 1, 2] is equivalent to [1, 2, 2, 0].



(a) True label: 0 (b) True label: 1 (c) True label: 2

Figure 6.4: Sample 2-port diodes.

6.5.2.1 Fluidic Diodes

Fluidic diodes are generally two-port devices that, when fluid flows one way, results in a low head loss, while fluid that flows in the reverse direction results in a large head loss. For two-port devices, we set one port to an inlet and one to an outlet ("forward flow"). We then reverse the inlet and outlet ("backward flow"). This gives us two BCs used in fluidic diodes. For four-port

(a) Open ports: Right, Top
True label: 0

(b) Open ports: Left, Bottom
True label: 1

(c) Open ports: Left, Top
True label: 2

Figure 6.5: Sample 4-port diodes. Two ports are open, and two have V_0 BCs to block them.

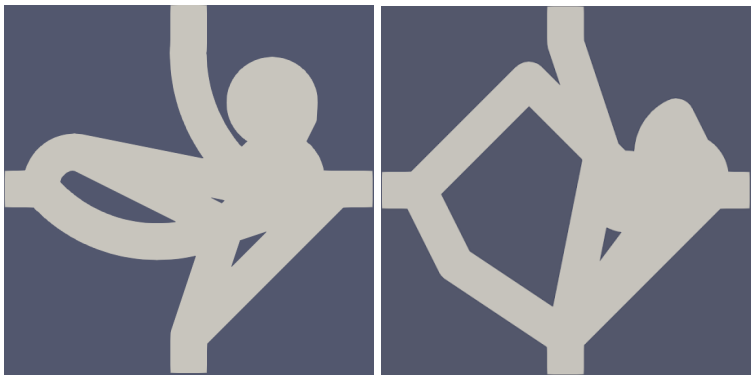
(a) Input ports: Left, Bottom
True label: Q

(b) Input ports: Left, Bottom
True label: AND

(c) Input ports: Left, Top
True label: XOR

(d) Input ports: Bottom, Right
True label: P

Figure 6.6: Sample 4-port logic gates. Because we can only measure 3 of the 4 possible input conditions, only 8 of 16 possible logic gates are theoretically discoverable.



(a) Inlet: Bottom
True label: Even split

(b) Inlet: Left
True label: Leaves from Bottom

Figure 6.7: Sample 4-port flow splitters.

devices, we set two ports to no-slip conditions; the remaining two ports are treated as above.

Potential fluidic diodes can either be forward diodes, backward diodes, or pipes; thus, there are three possible groups of devices.

To project fluidic diode data into performance space, we calculate the diodicity of a device. We first find the average pressure across each of the ports for every set of applied BCs. The difference between the inlet and outlet pressures for a given set of BCs gives us the head loss for that BC. The ratio of the head losses between different BCs is the diodicity. Algorithms 6 and 7 show the pseudocode for this process for 2- and 4-port diodes, respectively, and we refer the reader to §A.9 for an implementation of these calculations.

Algorithm 6 Pseudocode to Project 2-port Diode Behavior Vector to Performance Space

1: Remove velocity values across all ports from the query BVec
2: Average pressure values across ports of the query BVec
3: Calculate differences in pressure drop for both sets of BCs
4: Calculate ratio of pressure drops

Algorithm 7 Pseudocode to Project 4-port Diode Behavior Vector to Performance Space

```
Remove values not associated with the two relevant ports from the query BVec
Remove velocity values across all ports from the query BVec
Average pressure values across ports of the query BVec
Calculate differences in pressure drop for both sets of BCs
Calculate ratio of pressure drops
```

6.5.2.2 Velocity Logic Gates

Velocity logic gates are four-port devices that can perform binary operations using outlet flow velocity to define output values. These devices have two zero-pressure ports (*i.e.*, outlets) and two ports designated as input conditions. Due to the conservation of mass, the velocities at the two outlets are intimately linked as a single degree of freedom, so measuring either outlet is sufficient for labeling.

Two-input binary logic gates have four possible input conditions, each of which can result in either a "0" or "1" output. A "0" input is a no-slip condition ($\oslash$), whereas a "1" input is a velocity input condition (V_{in}). The ($\oslash$, $\oslash$) input always results in a "0" output since no fluid is flowing into the device, rendering it trivial. Since each inlet port can have one of two velocities (either $\oslash$ or V_{in}), and we neglect the ($\oslash$, $\oslash$) condition, velocity logic gates have three possible inlet conditions, each with one of two outlet conditions. Thus, there are $2^3 = 8$ possible groups of devices.

To project velocity logic gates into performance space, we calculate an 8-D vector that represents how well a device aligns with a particular logic gate behavior. We start by taking the velocity magnitudes of our output port under all three inlet condition — *i.e.*, ($\oslash$, V_{in}), (V_{in}, $\oslash$), and (V_{in}, V_{in}) — creating a 3-D vector. We use a dot product to compare this to the theoretical "perfect" logic gate, *i.e.*, one that has exactly outputs of (0, 0, 0), (0, 0, 1), (0, 1, 0), etc. up to (1,

1, 1) for a total of eight comparisons. These ideal BVecs are listed in Table 6.1. These vectors

Table 6.1: Each row denotes one ideal BVec. Column headings denote the input conditions of the logic gate.

$(\oslash, V_{in})$	$(V_{in}, \oslash)$	(V_{in}, V_{in})
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

are then normalized to have a magnitude of 1 before each device's BVec is compared to each row separately using the dot product. The values of these dot products project a given device into its performance space. Algorithm 8 shows the pseudocode for this process, and we refer the reader to §A.10 for an implementation of this calculation.

Algorithm 8 Pseudocode to Project Velocity Logic Gate Behavior Vector to Performance Space

```
Set the ideal BVecs according to Table 6.1
Remove pressure values across all ports from the query BVec
Calculate magnitude (L^2 norm) of each velocity of the query BVec
Average magnitudes of the query BVec
Extract only the outlet port magnitudes from the query BVec
for each possible ideal BVec do
    Calculate and store dot product with the query BVec
end for
Return all dot products
```

6.5.2.3 Flow Splitters

What we call flow splitters are devices that take a single inlet velocity and either redirect it or split it into multiple flows evenly. These devices have one inlet velocity and zero-pressure outlets everywhere else. Because of this, they only have one BC associated with their behavior.

For a four-port splitter, the possible outputs are an even split into all three other ports, an even split into two other ports, or redirecting all flow into one other port. Thus, there are seven possible groups of devices.

To project flow splitters into performance space, we calculate a 7-D vector that represents how well a device aligns with a particular behavior. We start by taking the average of the velocity magnitudes across our output ports. This gives us a 3-D vector of magnitudes. We use a dot product to compare this to the theoretical "perfect" flow splitters, *i.e.*, ones that split the flow into three even flows, two even flows, or simply redirect the flow into one port. For example, a three-way splitter would have output magnitudes of approximately (0.33, 0.33, 0.33), while a two-way splitter could be (0.5, 0.0, 0.5). The seven dot products against "perfect" flow splitters gives us a 7-D vector in the performance space. These ideal BVecs are listed in Table 6.2. These

Table 6.2: Each row denotes one ideal BVec. The sum of each vector should be 1.0.

Proportion of Flow		
0.33	0.33	0.33
0	0.5	0.5
0.5	0	0.5
0.5	0.5	0
0	0	1
0	1	0
1	0	0

vectors are then normalized to have a magnitude of 1 before each device's BVec is compared to each row separately using the dot product. Algorithm 9 shows the pseudocode for this process, and we refer the reader to §A.11 for an implementation of this calculation.

Algorithm 9 Pseudocode to Project Flow Splitter Behavior Vector to Performance Space

```
Set the ideal BVecs according to Table 6.2
Remove pressure values across all ports from the query BVec
Remove inlet port velocities from the query BVec
Calculate magnitude ($L^2$ norm) of each velocity of the query BVec
Average magnitudes of the query BVec
for each possible ideal BVec do
    Calculate and store dot product with the query BVec
end for
Return all dot products
```

6.5.3 Baseline Methods

We explore the Scikit-learn implementations of KMeans clustering (KMeans) and DBSCAN [131]. We also test HDBSCAN [18]. We have chosen these algorithms because they showed promising results in our previous work [77], and we found that the choice of clustering algorithm was not critical to performance on this type of problem.

We perform 5-fold cross validation with Scikit-learn and hyperparameter optimization using Bayesian Optimization (α = 1e-4, $n_{iter} = 5 + 30 \times$number of parameters) [23] with the ranges in Tables 6.3, 6.4, and 6.5. For example, KMeans gets 35 iterations ($5 + 30 \cdot 1 = 35$), while DBSCAN gets 95 ($5 + 30 \cdot 3 = 95$). Each hyperparameter also includes its associated code nickname. Note that SHDBSCAN does not have any free parameters, so it does not have a table here associated with it. We do not expect any significant improvements in performance by optimizing the hyperparameters further.

Table 6.3: Hyperparameter Ranges for KMeans Clustering.

KMeans	Type	Low	High
Number of Clusters (n_clusters)	int	2	20

To measure performance in the hyperparameter optimization, we use the SS from Scikit-

Table 6.4: Hyperparameter Ranges for DBSCAN Clustering.

DBSCAN	Type	Low	High
Epsilon (eps)	float	1e-5	5
Minimum Samples (min_samples)	int	2	100
Leaf Size (leaf_size)	int	2	20

Table 6.5: Hyperparameter Ranges for HDBSCAN Clustering.

HDBSCAN	Type	Low	High
Minimum Cluster Size (min_cluster_size)	int	2	100
Minimum Samples (min_samples)	int	2	100
Cluster Selection Epsilon (cluster_selection_epsilon)	float	1e-5	5

learn. We train on 75% of our data, without stratification, and use the remaining 25% for testing. We measure performance on the test data using the adjusted Rand score [19] and calculated true labels.

6.6 Results

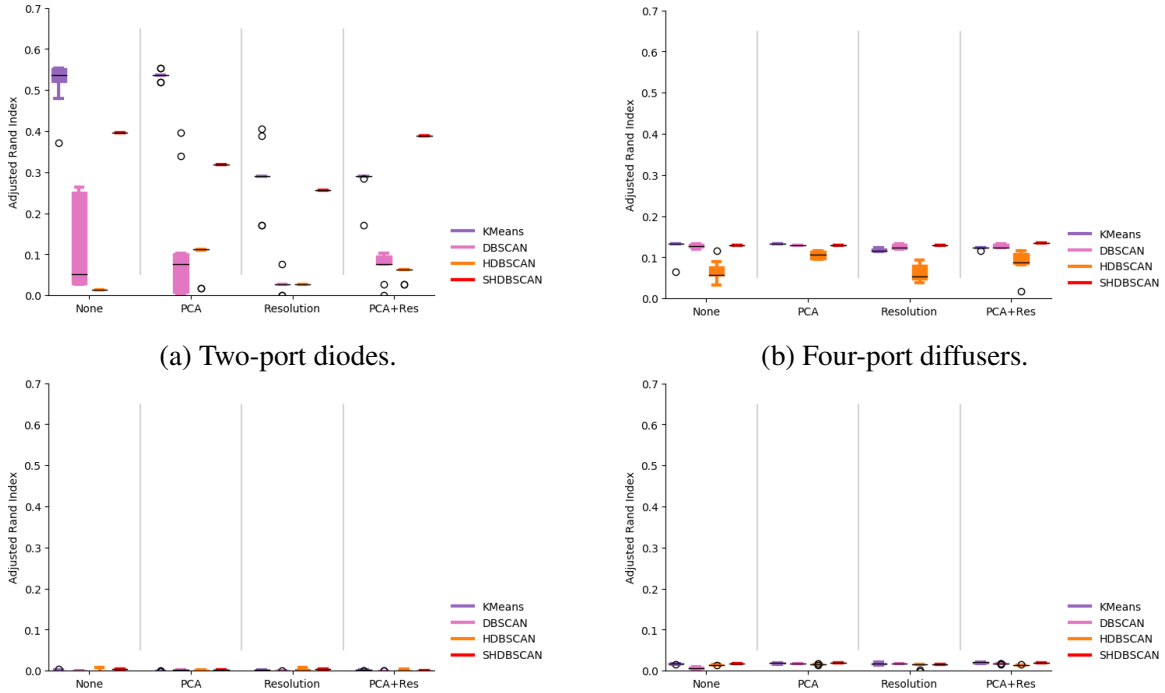
After running the experiments outlined in §6.5.3, we find that the preprocessing method, clustering algorithm, and device application all have strong impacts on the clustering performance. Figure 6.8 shows the results of our clustering. Black lines indicate median scores, with thick colored rectangles indicating the first and third quartiles.

The adjusted Rand score for four-port diodes and velocity logic gates hovers around 0.0 across the clustering and preprocessing methods, as seen in Fig. 6.8c and 6.8d. Figure 6.8b shows that for diffusers, the adjusted Rand score increases to around 0.14 for most clustering and

preprocessing applications with the exception of HDBSCAN; its score is somewhat lower than the other algorithm.

We find the highest adjusted Rand scores in Fig. 6.8a, which shows that clustering of two-port diodes using KMeans clustering rises to about 0.55 using no preprocessing or using Principal Component Analysis (PCA); other preprocessing methods decrease the clustering quality. We also see that DBSCAN and HDBSCAN both perform relatively poorly (medians around 0.1 at best), regardless of preprocessing, but that SHDBSCAN has adjusted Rand scores that peak around 0.4 when using no preprocessing or using both PCA and resolution selection.

We find that DBSCAN has the highest variability in performance; the other clustering algorithms have tighter spreads of performance.



(a) Two-port diodes.

(b) Four-port diffusers.

(c) Four-port diodes

(d) Four-port velocity logic gates.

Figure 6.8: Adjusted Rand score of clustering algorithms on various applications.

Figures 6.9, 6.10, 6.11, and 6.12 show devices from different clusters found by KMeans

with no preprocessing.

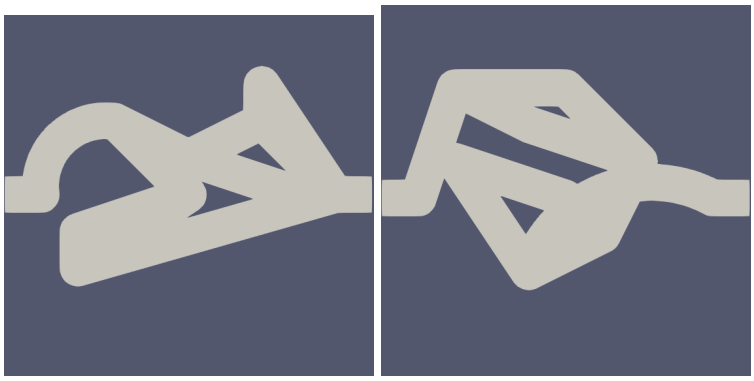


Figure 6.9: Potential diode-like 2-port devices found by KMeans with no preprocessing.

6.7 Discussion

The best performance we see in this chapter is with diode-like devices, specifically using KMeans clustering, even without preprocessing. PCA in this case seems only to decrease the clustering performance variability.

We see that DBSCAN performs most poorly across most of our experiments, though it is surprising to see HDBSCAN perform worse on the diffuser application while DBSCAN performs on-par with the other algorithms. In general, KMeans performs best across the board, though SHDBSCAN slightly outperforms it when applying both PCA and resolution selection in the two-port diodes and the diffuser applications. Despite this, however, more complex density-based algorithms do not make not make a compelling case for these types of problem over simpler KMeans clustering.

One area where automated clustering struggles, in general, is for more complex behaviors, such as four port devices. The device geometries we use in these experiments are randomly generated; as such, it is extremely unlikely that we will produce devices that are high-performance

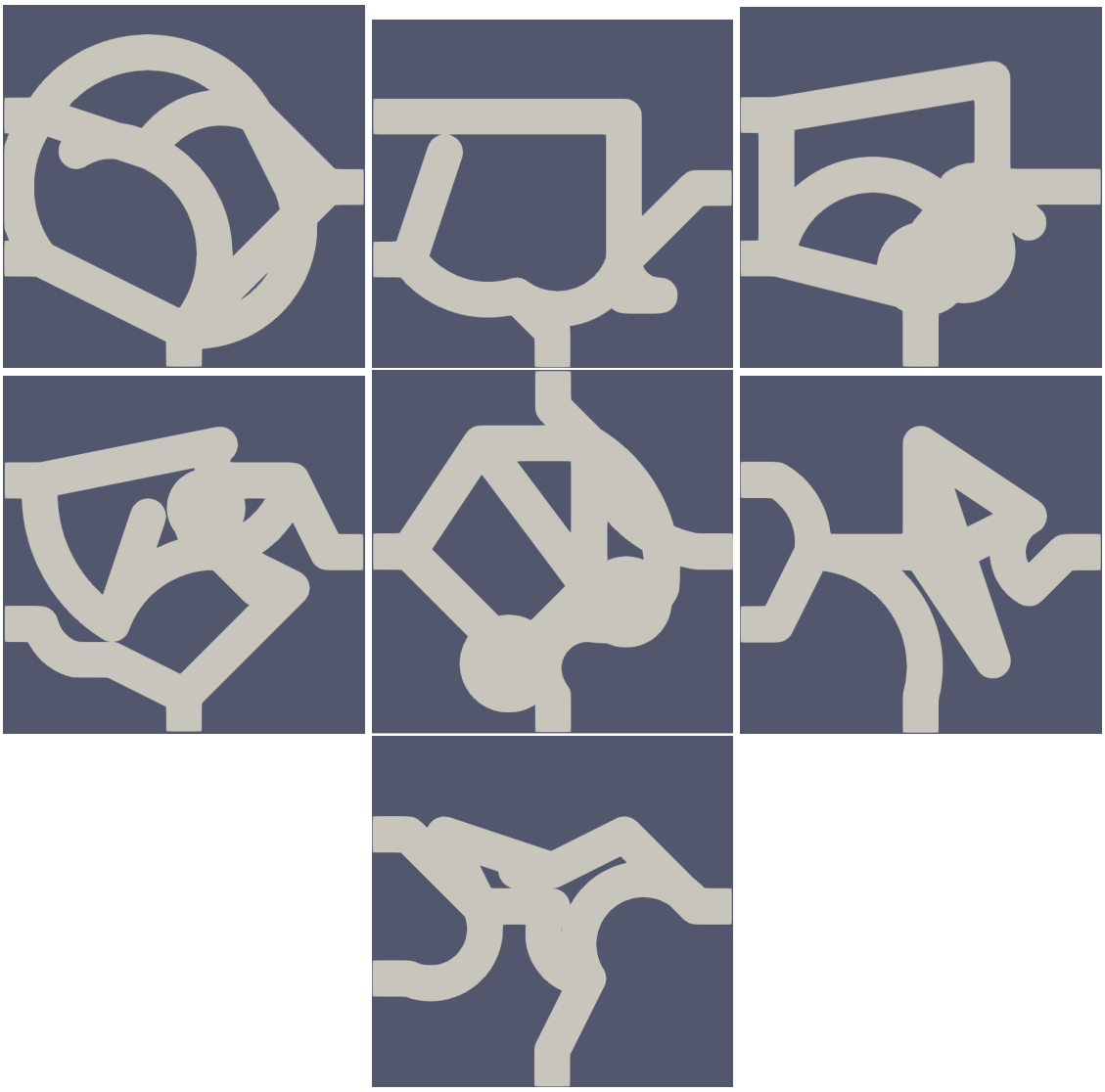


Figure 6.10: Potential diode-like 4-port devices found by KMeans with no preprocessing.

fluidic devices, *e.g.*, a fluidic diode or logic gate. Although it is possible that *some* devices will perform specialized functions well, they are extremely rare, so it is unlikely that they are included in sufficient quantities in our data set to produce a meaningful "cluster." In other words, this type of problem may be too imbalanced for clustering methods to capture those functions well.

One potential mitigation of these challenges would be to include simulation data from a variety of "good" devices — *e.g.*, devices that we know can perform specialized functions well. For example, including a significant quantity of various designs for Tesla valves, with the appropriate

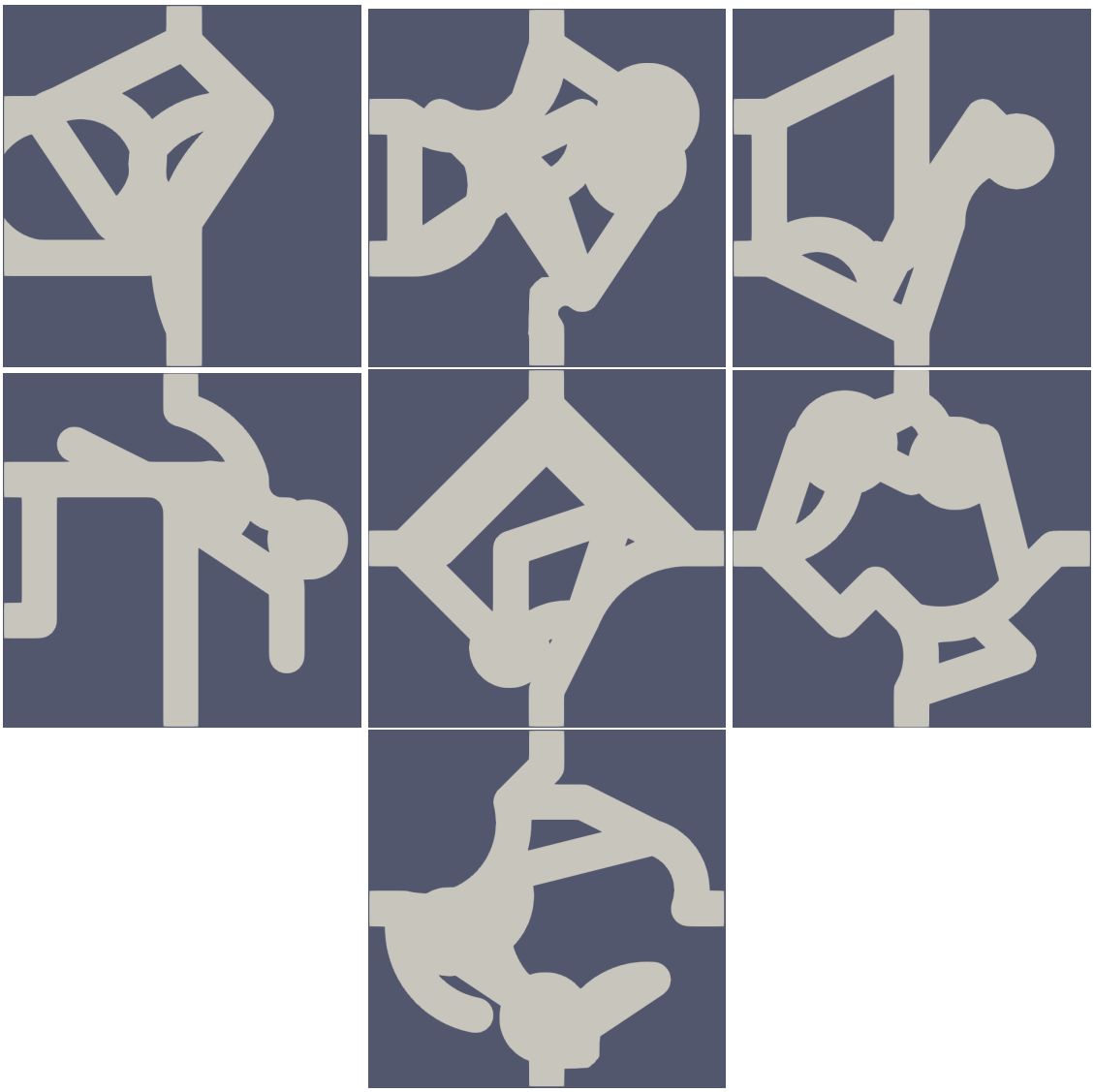


Figure 6.11: Potential logic gate-like 4-port devices found by KMeans with no preprocessing.

BCs, could force a "good diode" cluster to form in our data set, which our clustering algorithms could then potentially find. Similarly, including a significant quantity of various designs for fluidic AND, OR, and XOR gates, again with appropriate BCs, could also force clusters of "good" fluidic logic gates to form. However, in this chapter we wanted to assess automated clustering for functional discovery directly, rather than biasing the possible cluster toward known solutions.

Because of the many sources of randomness in our current experimental setup, we expected

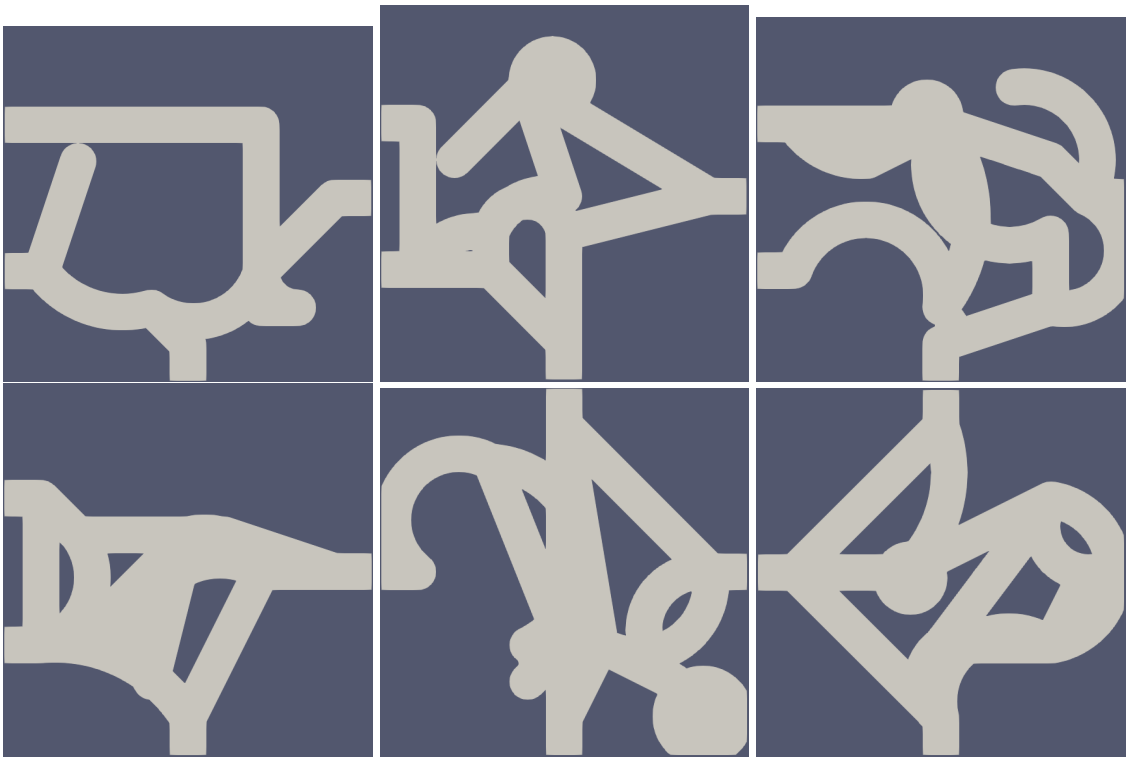


Figure 6.12: Potential flow splitter-like 4-port devices found by KMeans with no preprocessing.

the number of "good" devices to be low, but it is possible that it was lower than expected and thus was not able to be caught using clustering algorithms. It may be worth trying anomaly detection algorithms rather than clustering algorithms, especially if only a handful (*e.g.*, on the order of 1% of data or less being "good" devices) are present in the data set.

One potential concern with our setup is that we set, by fiat, the calculation of true labels. Specifically, we specify the equations and calculations used to project a device's behavior into the performance space for a certain type of device, *e.g.*, by using the code provided in §6.5.2.1 to calculate diodicity from fluid pressure fields. For clarity, a device's *behavior space* is the N-D space in which its BVecs lie. This dimension changes between experiments based on the number of BCs that are applied, but for an example 2-port fluidic diode, this is a space across 2 ports, with 20 points per port, and 2 sets of applied BCs, resulting in a $2 \times 20 \times 2 = 80$-D behavior space.

For an example 4-port logic gate, this space spans 4 ports, with 20 points per port, and 3 sets of applied BCs, resulting in a $4 \times 20 \times 3 = 240$-D behavior space. To calculate the true labels of these devices, we project them into their corresponding *performance space*. For the 2-port fluidic diode,this means using a method like the code snippet in §6.5.2.1 to calculate the diodicity, which is a 1-D value; thus, the performance space is this device's diodicity and only spans one dimension. for the 4-port logic gate, we instead use a method like the code snippet in §6.5.2.2 to calculate the projection against eight possible "ideal" BVecs; thus, the performance space for the 4-port logic gate is 8-D. Projecting from the behavior space to the performance space for labeling necessarily limits the possible clusters of devices we can identify to those that we can calculate in our performance spaces, even if other clusters inherently exist. We do this because, for the purposes of scientific inquiry, we need some metric to present. Although some metrics exist that do not require ground truth labels — *e.g.*, the SS, which intuitively measures how well a point matches other points in its cluster and how different it is from points in other clusters — those metrics do not necessarily give a reader a sense of how "good" a score is. Additionally, we use the SS to optimize our hyperparameters; presenting the SS as an objective metric would not be scientifically rigorous and would artificially inflate our models' performances. In summary, while novel devices may exist within our data, we are unable to measure them scientifically.

One of the unfortunate side effects of our labeling process is that novel devices, which do not project into our performance space well, are forced to hold a label that does not match their behavior. Currently, our algorithm and labeling methods only handle hard labeling — *i.e.*, each device is assigned exactly one label, and each device is put into exactly one cluster, without any fuzziness or soft borders. Because a novel device would not match any of the behaviors that the performance space is meant to differentiate, it may be projected far away from any of

the other devices and effectively be assigned an arbitrary label in the projection space. This reduces clustering quality according to our performance metric (the adjusted Rand score), so, somewhat ironically, the more novel devices exist in the design space, the worse our algorithms will perform. Additionally, while our design space theoretically may contain a wide variety of devices, the vast majority of the randomly generated designs will be effectively pipes, albeit with extra bits. High-performance devices with complex behaviors, such as logic gates, are unlikely to appear by chance through our randomization scheme; as such, most of our devices likely form a large blob in the performance space as part of the "glorified pipe" cluster. Even so, our labeling algorithm may still forcibly label these glorified pipes with some other behavior if they show slight tendencies towards known interesting behaviors.

Throughout this chapter, we have alluded to the Function-Behavior-Structure scheme first proposed in [38]; however, we have not discussed why we focus on *behaviors* instead of *functions*. On one hand, it would make sense for us to cluster based on function, as that is an end goal of engineering design. We tend to agree, and would argue that, for single-physics devices with relatively simple behaviors, the behavior and function are effectively identical. The distinction is more apparent when considering simulations across a variety of physics; as we noted in §6.1, many different types of physics can produce the same functions. That said, in this chapter, we cluster specifically on behaviors, not functions.

We expect our contributions in this chapter to be conceptually generalizable to other types of physics with relatively small changes. As long as the borders of other simulations allow for generating BVecs of equal size, we do not foresee any potential issues with technical implementation of these contributions. The effectiveness of such extensions is less clear. We expect our algorithms are most effective when devices have disparate, well-separated locations in the

behavior space; that is, when several groups of behaviors distinctly exist. We used fluidic devices in this chapter as our test bed problem because we expected that the complexity would provide an interesting application. Instead, we believe that this is too difficult a test problem and that it would be prudent to conduct our tests on a simpler set of problems. We expect a design space based on compliant mechanisms or rigid link structures would make a more appropriate test problem. Presuming that displacements are the SFs of interest, we expect these behaviors are more discrete, disparate, and time-independent than that of fluidic devices. With more advanced simulation capabilities, we expect that multi-physics devices may also provide a wide array of possible devices. However, we imagine that the design space (*i.e.*, structure) would need to be restricted — or at least modified — to encourage more varied behaviors, rather than the blob of glorified pipes that we see in our experiments.

6.8 Conclusion

This chapter explored a new avenue of computationally discovering groups of fluidic devices from unlabeled data, even when the device's behavior is not known *a priori*. Specifically, we investigated the performance of several clustering algorithms and preprocessing heuristics in the discovery of fluidic devices, leading to the following two conclusions:

1. Our modifications on the HDBSCAN algorithm, which we call the SHDBSCAN algorithm, generally outperforms its predecessors on behavior clustering tasks, but is outperformed by KMeans clustering, at least for the tasks we consider in this chapter.

2. Of the dimensionality reduction and preprocessing methods we tested, the clustering algorithm used and the type of application are both more important than the type of preprocess-

ing.

Exploring expanded design spaces computationally can uncover novel and diverse behaviors and devices that have been previously unexplored, even without the need for human guidance. This has the potential to invent families of devices that have been overlooked by human designers.

However, the methods presented here rely on some assumptions about the data. First, we approach the task of function discovery as a clustering problem. If clusters do not exist in the behavior space, then these methods will not discover clusters of similar behavior, regardless of preprocessing or clustering algorithm. The lack of clusters can come from a lack of data capturing "interesting" phenomena or behavior (*e.g.*, the phenomena are so rare that they are not captured by our randomized designs), too much similarity between data (*e.g.*, every single device behaves almost identically), or a performance space that is inherently uniformly dense. In our preprocessing, we assume that the data points are physically adjacent; removing this assumption renders our Resolution Selection heuristic inapplicable. Similarly, selecting an initial sampling resolution that is too low renders BVec formulation ineffective because the BVec would not capture information about the phenomena in the first place. We also test a necessarily limited set of preprocessing and clustering algorithms. In this chapter, we use PCA as one potential preprocessing step; however, PCA is a linear model, whereas the data used in our experiments is not necessarily linear. Potential other preprocessing methods that handle nonlinear data more effectively (*e.g.*, kernel PCA [132]) could potentially reduce more dimensionality with less impact on information loss. Finally, we only study limited classes of fluidic devices that exist in our design space within our simulation criteria. Increasing the flexibility of our design space could

potentially result in more classes of devices, and widening our simulation criteria could push into either other types of physics or more advanced phenomena in fluidic devices, *e.g.*, devices that have time-dependent or turbulent behaviors.

We started this chapter by asking under what conditions we can detect novel device behaviors through computer-driven simulation and exploration. Towards answering this question, we studied fluidic devices using various clustering algorithms to group device behavior from simulation data while also exploring various types of preprocessing that could be used to improve clustering performance. Although we were limited by our simulation capabilities, we hope that future work will incorporate our contributions from Chs. 3, 4, and 5 to allow a wider range of complex, multi-physics simulations.

Chapter 7: Conclusion

This dissertation originally set out to tackle the question of "What device functionality emerges organically from knowledge of various physical laws?" Although this topic remains an active area of research, we have made the following contributions to answer this question:

1. In Ch. 3, we detailed a type-based indexing scheme for robustly defining Finite Element Method (FEM) simulations. This indexing scheme was tested on three types of equations and predicted viability of FEM simulations accurately. We recognized that this contribution currently handles Dirichlet boundary conditions (BC) only and that extension to other types of BCs and physics is an avenue of future work.

2. In Ch. 4, we predicted a surrogate quantity of interest that substituted physical realizability of Stokes flow simulations using regression methods. Our experiments suggested that Resolution Selection — a heuristic we developed and demonstrated in §2.3.3.2 — an autoencoder, and a fully connected Neural Network predict physical realizability most accurately. We hope that extending this to other types of physics provides more practical benefit in the future.

3. In Ch. 5, we extended our previous work in [3] to include a wider variety of geometries. This extension led to a heuristic that improved predictive accuracy across all of the models

we tested. We additionally hypothesized on various limitations of our current method, including decomposition scale and types of physics.

4. In Ch. 6, we proposed a modification to a popular clustering algorithm called Silhouette-optimized Hierarchical Density-based Spatial Clustering of Applications with Noise (SHDBSCAN) and a dimensionality reduction heuristic to shorten behavior vectors. While our dimensionality reduction heuristic increased adjusted Rand scores in our clustering, the potential classes of devices had a stronger affect on the results, likely due to shortcomings in our randomization methods. We noted that extending this work to other types of physics may broaden the types of devices that could be discovered.

We hoped to expand the use of data-driven design in mechanical engineering by automating parts of the process to free engineers to work on scientifically more creative avenues. Unsurprisingly, data-driven design assumes that "good" data is readily available, but we have found that acquiring such data is not trivial because of the expertise and human creativity needed to acquire that data. Throughout this dissertation, we have learned four critical aspects that impact the ability of computers to learn how to automate design discovery. We summarize these briefly below.

Identifying "bad" boundary conditions is difficult. Simulations for a specific type of physics, broadly speaking, have three components: geometry, solver parameters, and BCs. Inappropriate simulation parameters (*e.g.*, low viscosity, large time steps, *etc.*) and inappropriate geometries (*e.g.*, non-continuous, overlapping, or extremely thin geometries or overly coarse meshes) were usually easily identified from the outputs; these issues cause the simulation to diverge and be unsolvable. Meshes can also be visually inspected for those flaws, though doing so at scale

is infeasible. However, in FEM solvers, BCs are effectively constants within a few rows of a linear system. As we discuss in Ch. 4, even when physically unrealizable BCs are applied in a FEM simulation, the solver often will still "solve" the system, even though the results do not match reality. A simulation of a pipe with two inlets and no outlets, in many solvers, will run to completion. Such simulations are not difficult to define by accident; a missing negative sign on an inlet velocity BC is sufficient. Additionally, misapplying BCs in a FEM simulation is a simple mistake to make. While initially simulating the fluid in an electrostatic field from Fig. 3.1, we often accidentally applied electrostatic BCs on pressure fields, or pressure BCs on velocity fields, or velocity BCs on electrostatic fields. Although our particular solver raised errors when we tried to apply scalar BCs — like pressure — onto vector fields — like velocity — the solver overlooked our mistakes when the BC and field were both scalars, even if one was for pressure and the other for electrostatic potential. This is the inspiration behind Ch. 3, where we develop a scheme for more robust simulation setup, especially when working with multiple types of physics.

Appropriately randomized geometry is critical to data generation. In Ch. 6, we struggle to find strong groups of devices that show certain behaviors. We believe this is, in large part, due to the geometry randomization that we use in this chapter. Although we can theoretically reproduce devices like the fluidic oscillator from [88] or Tesla valves [133], the probability of such geometries appearing in our data set in large enough quantities to discover the cluster is low. Unfortunately, such restrictions diminish our ability to find devices with novel behaviors; the design space is too large for our current experiments to cover. We partially mitigate this issue in Ch. 5, where we restrict the basis simulations to a handful of basis geometries. This, however, may have restricted the geometries inappropriately; some of the geometries unrealistically

allowed fluid to leave the system too early, causing unexpected flow magnitudes throughout the system. Finding the balance between geometry comprehensiveness and manageability is critical for data generation, and we are not sure where that line falls, especially since that balance depends on the specific application. Based on our work in this dissertation, we believe that creating a randomization scheme for automated design discovery based on "good" designs, rather than a comprehensive design space, may be more viable, at least to start. Our approach was generally to encompass as many possible designs as possible; that is, we focused on covering a comprehensive design space. As a result, we were skeptical of any "good" designs we came across in our analysis (*e.g.*, a diode with a diodicity greater than 5), believing these may be a fluke of the simulation procedures rather than a good design. Instead, we suggest that future researchers begin with designs for devices that show interesting behaviors to start and modify those designs to create a design space. This will bias any automated search toward those specific behaviors initially, but we believe this is a reasonable compromise for developing automated search methods. If a search method cannot rediscover obvious behaviors in a simplified behavior space, we do not imagine it would discover novel behaviors in an expansive behavior space.

But geometry is not as crucial as an input to regression models as expected. For multiple chapters, we attempt to embed some part of the geometry into the models' inputs. In Ch. 4, we use a 0/1 encoding of the domain, along with several preprocessing methods. In Ch. 5, we instead implicitly encode the overarching geometry through connected basis geometries. However, we found that model performance generally improved in Ch. 4 as the number of dimensions devoted to geometry representation decreased, and in Ch. 5, the best performing model did not have the overarching geometry implicitly at all. Chapter 6 specifically ignores the internal geometry,

though we are unable to draw any conclusions based on it specifically. This is surprising because we expected the geometry to contribute significant predictive power. We hypothesize that geometry inputs may not provide as much information value per dimension added as other types of inputs, and thus only confuse the machine learning (ML) models with tangentially relevant, if not critically useful, information.

Broadening this idea further, the best type of information representation for a model's inputs depends on the specific application and the goals for a particular model. For cases where a domain's internal behavior is unimportant, as we hypothesized in Ch. 6, using information from the boundaries of the domain can be sufficient. In fact, we even reduce the dimensionality of the behavior vectors used because behavior clustering tasks only need some way to differentiate between devices, not a full understanding of the internal behavior as image-based methods provide. However, when a domain's internal behavior is important, as we conjectured in Ch. 5, using a method that respects physical proximity between different parts of the system can be necessary. We saw that flows in one small simulation affect flows in its neighbors, though not quite to the degree that we initially hypothesized. We cannot conclude whether this is a consequence of the specific phenomena we explored or inherent in compositional work where long-distance phenomena can be completely captured through more local models. That said, we can still use images to represent domains in cases where only the domain shape is needed, as we explored in Ch. 4. Images are often high-dimensional, so reducing the dimensionality through any of the many high-density-low-loss image compression algorithms that already exist can be a strong starting point. Applying complex ML models that require costly training, like autoencoders, may not always be the best choice; at some point, the cost to train the model is not worth the benefits the model provides. Such a trade-off is specific to the application and type of model used,

though, and we do not have quantitative guidelines for when such a trade-off tips the other way. For example, autoencoders in §4.5 were most effective for that application, but we question if it was helpful at all, while Ch. 6 used a simple algorithm called Resolution Selection explained in §2.3.3.2, and Ch. 5 did not use any preprocessing at all. Being cognizant of why we add an image to the inputs of our models can drastically change the dimensionality of the problem, whether it provides helpful information or actively distracts.

Fluidic devices may be too difficult for initially developing automated design search methods. Throughout this dissertation, we have simulated the Navier-Stokes and Stokes flows for fluidic device discovery. We hoped, perhaps naïvely, that our algorithms would be able to analyze Navier-Stokes simulation results and rediscover a wide variety of fluidic logic gates, multiple designs for fluidic diodes, and perhaps even devices that combined multiple physical phenomena to produce entirely new and robust behaviors. Over time, we have realized that designing fluidic devices is difficult, in part due to the scale of phenomena described by the Navier-Stokes equations. Fluid flows range from creeping to chaotic turbulence, which, while able to encompass a variety of phenomena, makes automated design search difficult. Instead, for future researchers we suggest starting with a simpler equation, such as the heat equation, as a test bed for the initial development of automated design search methods. A simpler equation should still provide an appropriate level of difficulty such that interesting behaviors are not obvious to baseline models, but not so difficult that the majority of difficulties stem from simulation, rather than the scientific contributions being tested. Physics with only one solution field, as in heat diffusion, also have a lower dimensionality than the multiple fields in fluid flows and a likely shorter simulation computational cost, allowing more data to be collected in the same amount of time. Although the

behaviors are likely more straightforward, we believe that this is a reasonable trade-off to focus on the aspects of most scientific interest.

7.1 Future Work

We see several potential directions for future work building off this dissertation. Throughout this dissertation, we see the importance that "good" BCs have on producing simulations that model reality and on guiding automated design search. In Ch. 4, we attempt to evaluate a given simulation setup's quality, implicitly differentiating "good" and "bad" BCs, and Ch. 3 selects sets of viable BCs from a given set of possible BCs. However, all of the BCs we use are, at their core, manually defined; we do not generate new BCs automatically. We believe that developing methods for generating physically realizable BCs that fully define a simulation and explore potentially novel ways to "use" a device are a promising avenue of future work in automated design search. Such approaches can build on our contributions toward measuring BC quality, though we expect that more sophisticated and targeted methods may additionally prove beneficial (*e.g.*, models for a specific application or type of physics).

One aspect we consistently struggled with in this dissertation is setting the most appropriate inputs, outputs, and information representation for our models. We attempt to choose inputs to our ML models that are informative, discriminative, and low-dimensional — *i.e.*, "good" inputs. However, we do not have a specific metric for what constitutes a good input to our models. Instead, we decide, based on experience and intuition, the types of values that should be good inputs, such as domain geometry and BC values. We use full geometries and BCs in Ch. 4, then forgo geometries for full simulation results in Ch. 5, then eschew full simulation results for

border behaviors in Ch. 6, but with limited success. For example, our expectation that geometry representations would be valuable to predicting physical realizability was contradicted by our experiments showing that models with fewer geometry-related inputs generally performed better. This brings to mind several questions: Do our models need more inputs? Fewer? Are there inputs we could have included that would have been extremely informative? Were the inputs we included helpful, or at least not misleading? Were the inputs we included actively harming our model performance? Towards this, we expect that there is some metric of quality or "information density" — the amount of predictive power a single dimension of an input gives — that can guide which inputs are used in ML models for automated design search. We are not currently aware of any research that quantifies the quality of different inputs for automated design discovery, but we believe this is a potential avenue of future work.

Another potential direction for future work is the implementation of automatically directed search. Our contributions in this dissertation general revolve a randomized approach to discovery; *i.e.*, Ch. 6 uses quantity to look for rare and interesting behaviors, which is what Chs. 3, 4, and 5 are meant to facilitate. In a way, our approach is the metaphorical infinitely many monkeys with typewriters: if we try enough random simulations, we expect that we will eventually find something interesting. Instead, we suggest that there may be other methods by which such search can be performed. For example, Ch. 6 clusters devices by behaviors, generally finding clusters in areas of high density. Could an algorithm look specifically in low-density regions in the same space to discover devices that have not been included in the design space? And in doing so, could such an algorithm then push to make certain behaviors even more extreme? For example, one of our labels in Ch. 6 refers to diodes, where the forward pressure drop is much less than the backward pressure drop. Could an algorithm find the diode cluster, then search for designs

with stronger diodicity without a human specifying that diode-like behavior is interesting or that diodicity is an interesting function?

Taking a different, if related, direction for automatically directed search, we also wonder about methods to compose various simpler behavior to produce more complex behaviors in a directed way. We discuss composing basis simulations to form big, more complex simulations in Ch. 5; we wonder, then, if doing so is also possible in a directed way. We introduced a fluidic oscillator from [88] as an interesting combination of complex behaviors from simple pieces in §5.1, as well as the idea of fluidic logic gates in Ch. 6. In theory, our contributions in Ch. 5 can be set set to take basis simulations for some specific set of fluidic oscillators and basic fluidic logic gates and simulate a counting clock. However, doing so would require fine-tuning by a human engineer to ensure that the logic gates are connected to form such complex behaviors. Would it, then, be conceivable that an algorithm could analyze some space of designs, recognize areas of low density, and pull from a library of known devices, such as fluidic logic gates and oscillators, to synthesize a network that performs behaviors in those areas of low density?

Through this dissertation, we have discussed a variety of contributions toward the goal of uncovering device functionality through understanding physical laws. Although this work does not completely automate all future design search, we hope that it provides another step on the path toward automating design discovery. We are hopeful that, by pushing toward more automated simulation and design search, we can free engineers — and humanity as a whole — to explore higher-level questions and understand our future in the universe.

Appendix A: Code Implementations

A.1 Neural Network Implementation

The implemented example Neural network has 3 fully connected hidden layers with 5, 10, and 15 nodes, respectively.

```
# Assuming that xTrain, yTrain, xTest, and yTest,
# representing the training data, training labels,
# testing data, and testing labels, exist, respectively.
# Also assuming that appropriate optimizer hyperparameters,
# namely initLearningRate, alpha, beta1, and beta2,
are already defined.


import torch
from torch import nn
from skorch import NeuralNetRegressor


numInputs = xTrain.shape[1] # Number of features
numData = xTrain.shape[0] # Number of data points
```

```
numEpochs = 1e6 # Number of training loops to run


# Defining a Neural Network regressor object
class NNRegressor(nn.Module):
    def __init__(self,
        numInputs,
        hiddenLayerSizeTuple,
        activationFunction):
        super(NNRegressor, self).__init__()
        self.numInputs = numInputs
        self.numOutputs = 1
        self.activationFunction = nn.Relu()


        self.allLinearLayers = [
            nn.Linear(self.numInputs, 5), # Inputs to 1st HL
            nn.Linear(5, 10), # 1st to 2nd HL
            nn.Linear(10, 15), # 2nd to 3rd HL
            nn.Linear(15, self.numOutputs) # 3rd HL to output
        ]


        self.allLinearLayers =
            nn.ModuleList(self.allLinearLayers)
```

```
        def forward(self, x):
            for linearLayer in self.allLinearLayers:
                x = linearLayer(x)
                x = self.activationFunction(x)
            return x


    # Initializing a skorch object to handle training loops
    skorchRegressor = NeuralNetRegressor(
            NNRegressor,
            batch_size = 1000,
            optimizer__lr = initLearningRate,
            optimizer__weight_decay = alpha,
            optimizer__betas = (beta1, beta2),
            # Using an ADAM optimizer for gradient descent
            optimizer = torch.optim.Adam,
            device = 'cuda', # Using GPU acceleration
            )


    skorchRegressor.fit(xTrain, yTrain)
    yTestPred = skorchRegressor.predict(X_test)
```

`yTestPred` can now be compared to `yTest` via a scoring metric, as in §2.1.1.6.

A.2 Implementation of Cross Validation with Bayesian Optimization

We have excluded some code related to data importing for brevity.

```
# Assuming that xTrain, yTrain, xTest, and yTest,
# representing the training data, training labels,
# testing data, and testing labels, exist, respectively.


# Assuming that clusterer is an instantiated clusterer.
# Using cv number of folds;
# in this case, 5-fold cross validation
# Using the Silhouette Score (SS) as the scoring
# metric for cross validation
def general_crossval(**kwargs):
    # Set the clusterer's parameters
    # given a dictionary of parameters
    clusterer.set_params(**kwargs)


    # Calculating the cross validation score
    cvScore = np.mean(
        cross_val_score(
            clusterer,
            xTrain,
            cv = 5,
```

```
            scoring = SS
        )
    )

    return cvScore

# Instantiating Bayesian Optimization object
# with cross validation.
# paramDict is a dictionary of parameter bounds
optimizer = BayesianOptimization(
    f = general_crossval,
    pbounds = paramDict)

# Variable number of optimizer iterations
# based on how many parameters are needed
n_iter = 30 * len(paramDict)

# The optimization step
optimizer.maximize(
    n_iter = n_iter,
    alpha = 1e-4
)
```

```
# Printing the optimal parameters
print(optimizer.max['params'])
```

A.3 Resolution Selection Implementation

To calculate the reduction in a single behavior vector (BVec), we can use the following code snippet.

```
# Assuming we have a N x 1 vector, bvec, that represents the
# field values for a single dimension of the simulation
# along a single port, e.g. x-velocity on the left edge.

def calculateM(bvec):
    bv = bvec/np.ptp(bvec) # Scaling between 0 and 1
    bv = np.diff(bv) # Calculating neighbor differences
    bv = np.abs(bv) # Forcing positive values
    bv = np.max(bv) # Finding largest jump
    bv = 1.0/bv # Reciprocating
    return np.floor(bv) # Truncating to an integer

    # In a single line, this can be written as:
    # return np.floor(1.0/np.max(
    #     np.abs(np.diff(bvec/np.ptp(bvec)))
    # ))
```

Every BVec in the data set is looped through this function to calculate m^*; this step is not shown. Once m^* is calculated, we can use the following code snippet to shorten a singular BVec. Note that this function starts extracting points at $\frac{m^*}{2}$ to avoid potentially uninformative values on the edge of the ports due to, *e.g.*, the no-slip condition in fluid flows.

```
# Assuming we have a N x 1 vector, bvec, that represents the
# field values for a single dimension of the simulation
# along a single port, e.g. x-velocity on the left edge,
# and the corresponding m*, mStar, for the dataset.


def shortenBVec(bvec, mStar):
    startIdx = int(mStar/2)
    return bvec[startIdx::mStar]
```

A.4 Autoencoder Implementation

We begin with importing libraries used for our sample autoencoder (AE). We use the PyTorch [11] and skorch [12] libraries to set up and train our model, respectively.

```
import numpy as np
import os
from sklearn.decomposition import PCA
import torch
import torch.nn as nn
import torch.nn.functional as F
```

```
from skorch import NeuralNetRegressor
from skorch.callbacks import EarlyStopping


# Using GPU acceleration
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

With the appropriate libraries imported, we define the encoder, decoder, and AE classes.

```
# Defining the encoder architecture
class Encoder(nn.Module):
    def __init__(self, numInputs = 10201):
        super(Encoder, self).__init__()
        self.numInputs = numInputs
        self.linear1 = nn.Linear(numInputs, 5000)
        self.linear2 = nn.Linear(5000, 2500)
        self.linear3 = nn.Linear(2500, 1000)
        self.linear4 = nn.Linear(1000, 500)
        self.linear5 = nn.Linear(500, 100)
        self.linear6 = nn.Linear(100, 50)
        self.linearOut = nn.Linear(50, 10)


    def forward(self, x):
        x = torch.flatten(x, start_dim = 1)
        x = F.relu(self.linear1(x))
```

```
        x = F.relu(self.linear2(x))
        x = F.relu(self.linear3(x))
        x = F.relu(self.linear4(x))
        x = F.relu(self.linear5(x))
        x = F.relu(self.linear6(x))
        x = self.linearOut(x)
        return x


# Defining the decoder architecture
class Decoder(nn.Module):
    def __init__(self, numInputs = 10201):
        super(Decoder, self).__init__()
        self.numInputs = numInputs
        self.linearLatent = nn.Linear(10, 50)
        self.linear1 = nn.Linear(50, 100)
        self.linear2 = nn.Linear(100, 500)
        self.linear3 = nn.Linear(500, 1000)
        self.linear4 = nn.Linear(1000, 2500)
        self.linear5 = nn.Linear(2500, 5000)
        self.linear6 = nn.Linear(5000, numInputs)
        self.avgPool = nn.AvgPool2d(
            3, stride = 1, padding = 1
        )
```

```
        self.sigmoidOut = nn.Sigmoid()


    def forward(self, z):
        z = F.relu(self.linearLatent(z))
        z = F.relu(self.linear1(z))
        z = F.relu(self.linear2(z))
        z = F.relu(self.linear3(z))
        z = F.relu(self.linear4(z))
        z = F.relu(self.linear5(z))
        z = self.linear6(z)
        z = torch.flatten(
            self.avgPool(
                z.reshape((-1, 101, 101))
            ),
            start_dim = 1
        )
        z = self.sigmoidOut(z)
        return z


# Combining the encoder and decoder
class Autoencoder(nn.Module):
    def __init__(self, latent_dims):
        super(Autoencoder, self).__init__()
```

```
        self.encoder = Encoder(latent_dims)
        self.decoder = Decoder(latent_dims)

    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded, encoded
```

We train the AE using the skorch library with an early stopping criterion of 500 epochs without improvement.

```
class AutoEncoderNet(NeuralNetRegressor):
    def get_loss(self, y_pred, y_true, *args, **kwargs):
        decoded, encoded = y_pred
        loss_reconstruction = super().get_loss(
            decoded, y_true, *args, **kwargs
        )
        return loss_reconstruction

# Assuming that the training data, xTrain, exists.

skorchRegressor = AutoEncoderNet(
    Autoencoder,
    device = device,
```

```
    callbacks = [EarlyStopping(patience = 500)],
)

skorchRegressor.fit(xTrain, xTrain)
```

Although it may seem unintuitive, `skorchRegressor.fit(xTrain, xTrain)` uses the input data as both the input data and the target labels because the AE is meant to reconstruct the input data.

Once `skorchRegressor` is fit, new data can be encoded with

```
xReconstructed, xEncoded = skorchRegressor.forward(xNew)
```

with the added bonus of getting the reconstructed data as well.

A.5 Typed Boundary Condition Object Implementation

```
import numpy as np


class TypedBCObject(object):
    # Need a value, location, and type
    def __init__(self, **kwargs):
        self.value = kwargs.get('value')
        self.location = kwargs.get('location')
        self.type = kwargs.get('type')


if __name__ == '__main__':
```

```
# Creating sample coordinates (location)
xCoords = np.zeros(11)
yCoords = np.linspace(0, 1, 11)
location = list(zip(xCoords, yCoords))

# Creating sample fluid velocity value
value = yCoords*(1 - yCoords)

# Creating sample type
type = 'fluidVelocity'

# Creating the sample object
exampleBC = TypedBCObject(
    value = value,
    location = location,
    type = type
)
```

A.6 Typed Variational Forms Object Implementation

We have excluded some helper functions for brevity.

```
from fenics import *
```

```
class navierStokesVF(object):
    def __init__(self, **kwargs):
        self.VFType = 'NavierStokes'
        self.SFTypesRequired = (
            'fluidVelocity',
            'fluidPressure'
        )

        # A check can be performed here to ensure
        # that the given SFs cover all of the
        # required SF types.

        # Storing SFs and TFs (test functions)
        self.SF = kwargs.get('SF')
        self.TF = kwargs.get('TF')

    def returnVF():
        # Renaming variables for brevity
        u = self.SF['fluidVelocity']
        v = self.TF['fluidVelocity']
        p = self.SF['fluidPressure']
        q = self.TF['fluidPressure']
```

```
        # Calculating VF for this PDE
        self.VF =
            inner(u, v)*dx \
            + inner(grad(u), grad(v))*dx \
            + inner(grad(p), v)*dx \
            + inner(div(u), q)*dx

        return self.VF
```

A.7 Graph Neural Networks Implementation

Note that the GCN takes edge connectivity as input, but not edge features, whereas the GAT uses both edge connectivity and edge features.

```
import torch
import torch.nn.functional as F
from torch_geometric.nn import GCNConv
from torch_geometric.nn import GATConv


class GCN1HL(torch.nn.Module):
    def __init__(self, numInputs, numNodesPerLayer):
        super().__init__()
        self.numInputs = numInputs
        self.numNodesPerLayer = numNodesPerLayer
```

```
        # Define the layers to be used
        self.convIn = GCNConv(
            self.numInputs, self.numNodesPerLayer
        )
        self.convOut = GCNConv(
            self.numNodesPerLayer, self.numInputs
        )

    def forward(self, x, edgeConnectivity):
        x = self.convIn(x, edgeConnectivity)
        x = F.relu(x)
        x = self.convOut(x, edgeConnectivity)
        return x

class GAT6HL(torch.nn.Module):
    def __init__(self, numInputs, numNodesPerLayer):
        super().__init__()
        self.numInputs = numInputs
        self.numNodesPerLayer = numNodesPerLayer

        # Define the layers to be used
        self.convIn = GATConv(
```

```
            self.numInputs, self.numNodesPerLayer
        )
        self.conv1 = GATConv(
            self.numNodesPerLayer, self.numNodesPerLayer
        )
        self.conv2 = GATConv(
            self.numNodesPerLayer, self.numNodesPerLayer
        )
        self.conv3 = GATConv(
            self.numNodesPerLayer, self.numNodesPerLayer
        )
        self.conv4 = GATConv(
            self.numNodesPerLayer, self.numNodesPerLayer
        )
        self.convOut = GATConv(
            self.numNodesPerLayer, self.numInputs
        )

    def forward(self, x, edgeConnectivity, edgeFeature):
        x = self.convIn(x, edgeConnectivity, edgeFeature)
        x = F.relu(x)
        x = self.conv1(x, edgeConnectivity, edgeFeature)
        x = F.relu(x)
```

```
        x = self.conv2(x, edgeConnectivity, edgeFeature)
        x = F.relu(x)
        x = self.conv3(x, edgeConnectivity, edgeFeature)
        x = F.relu(x)
        x = self.conv4(x, edgeConnectivity, edgeFeature)
        x = F.relu(x)
        x = self.convOut(x, edgeConnectivity, edgeFeature)
        return x
```

A.8 Silhouette-optimized Hierarchical Density-Based Spatial Clustering on Applications with Noise (SHDBSCAN)

pseudocode

```
from sklearn import cluster
import numpy as np
import hdbscan
from sklearn import metrics
from sklearn.metrics import pairwise_distances,
    silhouette_score, adjusted_rand_score
import sklearn.decomposition
import copy
from sklearn.neighbors import KNeighborsClassifier
try:
```

```
    import tqdm
    has_tqdm = True
except:
    has_tqdm = False


def print_tree(tree, i, depth, acc_tree=None):
    for k in tree[i]:
        if acc_tree!=None:
            print(
                '-'*depth+' '
                +'{}'.format(k)
                +': {}'.format(acc_tree[k])
            )
        else:
            print('-'*depth+' '+'{}'.format(k))
        if k in tree:
            if acc_tree!=None:
                print_tree(
                    tree, k, depth+1,
                    acc_tree=acc_tree
                )
            else:
                print_tree(tree, k, depth+1)
```

```
class SHDBSCAN():
    def __init__(self, verbose=False, levels=20):
        self.verbose = verbose
        self.levels = levels
        self.best_params_ = "N/A"

    def map_keys(X, labels):
        pass

    def set_params(self, **kwargs):
        pass

    def get_params(self, deep = False):
        params = {
            'verbose': self.verbose,
            'levels' : self.levels
        }
        return params

    def fit(self, X):
        """Takes an array of shape (n_samples, n_features)
        and generates cluster labels, stored in the
```

```
.labels_ property"""
self.X = X
self.origX = X
clusterer = hdbscan.HDBSCAN(
    min_cluster_size=2
).fit(X)

# Here's where we build the intermediate data
# structures to perform the clustering
# tree-> convert the condensed tree (as a numpy
#        array) to a dictionary of lists, where
#        dictionary keys correspond to nodes in the
#        tree, and the contents are a list of child
#        nodes. Note that only leaf nodes
#        correspond to actual data; the other nodes
#        are intermediate nodes in the tree
# root-> The topmost node in the tree. If there
#        are multiple roots, we create a new,
#        higher-level node that links them into a
#        single tree.
# acc_tree-> Store the size of the tree under each
#        node. This is used to generate clusters,
#        by walking down the tree until the size of
```

```
#        the sub tree is less than the max size
# flat_m-> "flatten" the tree. Same structure as
#        "tree", but the list of child nodes
#        contains only the leaf nodes for that sub
#        tree, in other words the actual data
#        points contained in that sub-tree

# list of tuples;
# 1st element is parent index
# 2nd is child index
# 3rd/4th are probability data (unused)
nc = clusterer.condensed_tree_.to_numpy()

# accumulate the numpy array into a tree structure
if self.verbose:
    print("Clusterer Shape: {}".format(nc.shape))
root = None
# each key (int) corresponds to label for node in
# condensed tree in HDBSCAN; value is array of keys
# (ints) of children of that node; leaf has just
# itself as child
tree = dict()
for i in range(0, nc.shape[0]):
```

```
        if nc[i][0]==i:
            root = i
            continue
        if nc[i][0] in tree:
            tree[nc[i][0]] = tree[nc[i][0]]+[nc[i][1]]
        else:
            tree[nc[i][0]] = [nc[i][1]]
    if self.verbose:
        print("Number of nodes: {}".format(len(tree)))

    # walk up the tree, until you hit the top;
    # finding roots, for each key, check every other
    # key if this key is 2nd key's child
    roots = []
    for i in tree.keys():
        child = False
        for j in tree.keys():
            if i in tree[j]:
                child = True
                break
        if not child:
            roots.append(i)
    root = roots[0]
```

```
# merge all roots under fake key -1
tree[-1] = roots
root = -1
self.tree_ = tree
if self.verbose:
    print("Roots: {}".format(roots))



# build accumulated tree; count how many leaf nodes
# are under each node
# keys are node indices, values are number of leafs
# in subtree
acc_tree = dict()
parents = [root]
while len(parents)>0:
    k = parents[-1]
    if k in tree:
        skip = False
        for c in tree[k]:
            if c not in acc_tree:
                parents = parents+[c]
                skip = True
```

```
                break
        if skip:
            continue
        acc_tree[k] = sum(
            [acc_tree[c] for c in tree[k]]
        )
    else:
        acc_tree[c] = 1
    parents.remove(k)

if self.verbose:
    print_tree(tree, -1, 0, acc_tree=acc_tree)

# contains indices (values) of all leaves under a
# given node (key)
flat_m = dict()
parents = [root]
while len(parents)>0:
    k = parents[-1]
    if k in tree:
        skip = False
        for c in tree[k]:
            if c not in flat_m:
```

```
                parents = parents+[c]
                skip = True
                break
        if skip:
            continue
        out = []
        for c in tree[k]:
            out = out+flat_m[c]
        flat_m[k] = copy.deepcopy(out)
    else:
        flat_m[k] = [k]
    parents.remove(k)

if self.verbose:
    print("Tree Size: {}".format(len(flat_m)))

# With the above data structures set up, we
# explore a range of max cluster sizes, and compute
# the silhouette score at each size.
num_c = []
scores = []
thresh=[]
# starting point of max cluster sizes
```

```
t=int(len(flat_m)/self.levels)
if t < 1:
    t = 1

thirdParam = int(len(flat_m)/float(self.levels))
if thirdParam < 1:
    thirdParam = 1

# array of max cluster sizes; 1:range/levels:N
tests = list(
            range(t, len(flat_m), thirdParam)
        ) + [len(flat_m)]

if self.verbose and has_tqdm:
    for _t in tqdm.tqdm(tests):
        res = calc_silhouette(
            X, _t,
            flat_m,
            tree,
            acc_tree,
            root
        )
        num_c.append(res[0])
```

```
            scores.append(res[1])
            thresh.append(_t)
    else:
        for _t in tests:
            res = calc_silhouette(
                X, _t,
                flat_m,
                tree,
                acc_tree,
                root
            )
            num_c.append(res[0])
            scores.append(res[1])
            thresh.append(_t)

    # array of results of each level
    sizes = list(zip(thresh, num_c, scores))
    if self.verbose:
        print(sizes)
    # search for maximum silhouette score
    sizes = [
        s for s in sizes
        if s[2]==max(sizes, key=lambda x:x[2])[2]
```

```
        ]
        # grab maximum cluster size for top scoring result
        size = sizes[0][0]
        clusters = get_clusters(root, size, tree, acc_tree)
        # set results internally
        self.clusters_ = clusters
        self.labels_ = gen_labels_for_k(
            flat_m,
            clusters,
            X.shape[0]
        )

    def fit_predict(self, X):
        self.fit(X)
        return self.labels_

#utility functions for computing silhouette scores
def gen_labels_for_k(flat_tree, clusters, size):
    """given the current clusterings and total number of
    points, generate a 1-D array that contains a label for
    each datum. -1 corresponds to points without a cluster,
    which are noise"""
    labs = [-1 for _ in range(0, size)]
```

```
    labs = np.array(labs)
    # iterate over list of clusters
    for i, c in enumerate(clusters):
        for p in flat_tree[c]:
            # label of each leaf in a cluster is that
            # cluster's index in list of clusters
            labs[p] = i
    return labs


def calc_silhouette(
    data,
    size,
    flat_tree,
    tree,
    acc_tree,
    root
):
    """compute the silhouette score for
    a given clustering"""
    # generate the clusters for a
    # given maximum cluster size
    clusters = []
    nodes = [root]
```

```
while len(nodes)>0: # walking down the tree
    n = nodes[-1]
    for t in tree[n]:
        # if the current node (aka a cluster) is too
        # big, iterate over it again later
        if acc_tree[t]>size:
            nodes = nodes+[t]
        else:
            # else it's small enough
            # so save it as a cluster
            clusters.append(t)
    nodes.remove(n)
# compute labels for that clustering
total_size = data.shape[0]
labels = gen_labels_for_k(
    flat_tree,
    clusters,
    total_size
) # add labels to each leaf node
# building temp set of data and
# labels that filter out noise; mostly for safety
_data = [
    data[i, :] for i, l in
```

```
        enumerate(labels) if l != -1
    ]
    _labels = [l for i, l in enumerate(labels) if l != -1]
    # compute score
    # if only a single label, return worst possible score
    # (silhouete score is ill-defined)
    if len(set(_labels))<=1:
        return (len(clusters), -1)

    try:
        score = metrics.silhouette_score(
            _data,
            _labels,
            metric='euclidean'
        )
    except:
        score = -1

    # return number of clusters and silhouette score
    return (len(clusters), score)

# same as top part of calc_silhouette
def get_clusters(root, size, tree, acc_tree):
```

```
    clusters = []
    nodes = [root]
    while len(nodes)>0:
        n = nodes[-1]
        for t in tree[n]:
            if acc_tree[t]>size:
                nodes = nodes+[t]
            else:
                clusters.append(t)
        nodes.remove(n)
    return clusters
```

A.9 Implementation of Diodicity Calculations

The following code snippet defines three functions. The first is a helpful utility function. The second, `calc2PortDiodicityFromBVec(BVec)`, takes a BVec from a 2-port device and calculates its diodicity, assuming `resolution` number of points along each port. The third, `calc4PortDiodicityFromBVec(BVec, BCs)`, does the same but for a 4-port device. This function requires the BCs as well to determine the correct port locations.

```
import numpy as np
resolution = 20 # number of points along each port


# Utility function for calculating averages across groups of
```

```
# values
def groupedAvg(myArray, N = resolution):
    result = np.cumsum(myArray, 0)[N-1::N]/float(N)
    result[1:] = result[1:] - result[:-1]
    return result

# Calculates diodicity of 2-port devices from the behavior
# vector, which is a resolution*numPorts*numBCs=20*2*2=80
# by 3 matrix for 2-port diodes. The first two columns
# contain velocity information; the third contains pressure.
# Diodicity is a single value greater than 0.
def calc2PortDiodicityFromBVec(BVec):
    numPorts = 2
    BVec = BVec[:, -1] # Only use pressure values

    # Averaging across the port
    ga = groupedAvg(BVec)

    # Fluids flow from areas of high pressure to areas of
    # low pressure. The 0-1 and 3-2 indices here, rather
    # than 0-1 and 2-3, ensure that the larger pressure
    # is always on the same side of the subtraction, thus
    # ensuring that the diodicity is always positive.
```

```
    forwardPressureDrop = ga[0] - ga[1]
    backwardPressureDrop = ga[3] - ga[2]

    diodicity = forwardPressureDrop/backwardPressureDrop \
                if backwardPressureDrop != 0 else np.nan
    return diodicity

# Calculates diodicity of 4-port devices from the behavior
# vector, BVec, which is a
# resolution*numPorts*numBCs = 20*4*2 = 160
# by 3 matrix for 4-port diodes. The first two columns
# contain velocity information; the third contains pressure.
# Diodicity is a single value greater than 0.
# Assuming the parameter 'BCs' is a string, like '0012',
# to denote the applied BCs.
# 0 is a blocked port (i.e., no slip condition).
# 1 is a Velocity In condition.
# 2 is a Zero Pressure condition.
def calc4PortDiodicityFromBVec(BVec, BCs):
    numPorts = 4

    wheres1 = BCs[0].find('1') # Inlet
    wheres2 = BCs[0].find('2') # Outlet
```

```
# Averaging across the port
ga = groupedAvg(BVec)[:, -1]

# Finding the indices of the not-blocked ports
# First set of BCs
BCIdx = 0
BC0Wheres1 = BCIdx*numPorts + wheres1
BC0Wheres2 = BCIdx*numPorts + wheres2

# Second set of BCs
BCIdx = 1
BC1Wheres1 = BCIdx*numPorts + wheres1
BC1Wheres2 = BCIdx*numPorts + wheres2

# Explicitly ensuring that the diodicity is
# positive.
forwardPressDiff = np.abs(ga[BC0Wheres1]
                          - ga[BC0Wheres2])
backwardPressDiff = np.abs(ga[BC1Wheres1]
                           - ga[BC1Wheres2])
diodicity = forwardPressDiff/backwardPressDiff
```

```
    return diodicity
```

A.10 Projecting Behavior Vectors into Velocity Logic Gate Performance Space

Note that this code snippet uses the `groupedAvg` function from §A.9.

```
import numpy as np


# Calculates similarity to 8 ideal BVecs for different
# potential logic gates for 4-port devices from the behavior
# vector BVec. BVec is a
# resolution*numPorts*3 = 20*4*3 = 240
# by 3 matrix. The first two columns
# contain velocity information; the third contains pressure.
# Ideal BVec similarities is an 8 by 1 vector.
# Assuming the parameter 'BCs' is a string, like '0012',
# to denote the applied BCs.
# 0 is a blocked port (i.e., no slip condition).
# 1 is a Velocity In condition.
# 2 is a Zero Pressure condition.
def calcVelLogicFromBVec(BVec, BCs):
    numPorts = 4


    outletIdx = BCs[0].find('2')
```

```
# Setting ideal BVs
allIdealBV = np.zeros((8, 3))
for seed in range(allIdealBV.shape[0]):
    allIdealBV[seed, :] = np.array(
            list(
                np.binary_repr(
                    seed, width = 3
                )
            )
        )
    if seed > 0:
        allIdealBV[seed, :] =
            allIdealBV[seed, :]/np.linalg.norm(
                allIdealBV[seed, :]
            )

# Only using velocity values to calculate
# velocity magnitudes.
magnitude = BVec[:, :2]
for idx, bv in enumerate(magnitude):
    magnitude[idx, 0] = np.linalg.norm(bv)
magnitude = magnitude[:, 0]
```

```
    magnitude = groupedAvg(magnitude)


    # Only looking at the outlet port's velocities
    thisBV = magnitude[outletIdx::numPorts]


    # Calculate similarity to all ideal BVecs
    allDotProducts = []
    for idx, idealBV in enumerate(allIdealBV):
        thisDotProduct = np.dot(thisBV, idealBV)
        allDotProducts.append(thisDotProduct)


    return allDotProducts
```

A.11 Projecting Behavior Vectors into Flow Splitter Performance Space

Note that this code snippet uses the `groupedAvg` function from §A.9.

```
import numpy as np


# Calculates similarity to 7 ideal BVecs for different
# potential flow splitters for 4-port devices
# from the BVec. BVec is a
# resolution*numPorts*1 = 20*4*1 = 80
# by 3 matrix. The first two columns
```

```
# contain velocity information; the third contains pressure.
# Ideal BVec similarities is a 7 by 1 vector.
# Assuming the parameter 'BCs' is a string, like '0012',
# to denote the applied BCs.
# 0 is a blocked port (i.e., no slip condition).
# 1 is a Velocity In condition.
# 2 is a Zero Pressure condition.
def calcDiffuserFromBVec(BVec, BCs):
    numPorts = 4

    # Only using velocities values to calculate
    magnitude = BVec[:, :2]
    for idx, bv in enumerate(magnitude):
        magnitude[idx, 0] = np.linalg.norm(bv)
    magnitude = magnitude[:, 0]
    magnitude = groupedAvg(magnitude)

    # Removing the inlet magnitude
    thisBV = np.delete(magnitude, BCs[0].find('1'))

    # Setting up the ideal BVecs
    allIdealBV = np.zeros((7, 3))
    allIdealBV[0, :] = np.array([1, 1, 1]) \
```

```
                    /np.linalg.norm([1, 1, 1])
allIdealBV[1, :] = np.array([0, 0.5, 0.5]) \
                    /np.linalg.norm([0, 0.5, 0.5])
allIdealBV[2, :] = np.array([0.5, 0, 0.5]) \
                    /np.linalg.norm([0.5, 0, 0.5])
allIdealBV[3, :] = np.array([0.5, 0.5, 0]) \
                    /np.linalg.norm([0.5, 0.5, 0])
allIdealBV[4, 0] = 1
allIdealBV[5, 1] = 1
allIdealBV[6, 2] = 1

allDotProducts = []
for idx, idealBV in enumerate(allIdealBV):
    thisDotProduct = np.dot(thisBV, idealBV)
    allDotProducts.append(thisDotProduct)

return allDotProducts
```

Bibliography

[1] Martin S. Alnæs, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris Richardson, Johannes Ring, Marie E. Rognes, and Garth N. Wells. The FEniCS project version 1.5. *Archive of Numerical Software*, 3(100), 2015.

[2] Kevin Chiu and Mark Fuge. Checking the Automated Construction of Finite Element Simulations from Dirichlet Boundary Conditions. *ASME International Design Engineering Technical Conferences*, Anaheim, CA, August 18-21, 2019 2019.

[3] Jun Wang, Kevin Chiu, and Mark Fuge. Learning to abstract and compose mechanical device function and behavior. In *ASME 2020 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. American Society of Mechanical Engineers Digital Collection, 2020.

[4] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andrew Ballard, Justin Gilmer, George Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks, 2018.

[5] Filipe De Avila Belbute-Peres, Thomas Economon, and Zico Kolter. Combining differentiable PDE solvers and graph neural networks for fluid flow prediction. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 2402–2411. PMLR, 13–18 Jul 2020.

[6] Zijie Li and Amir Barati Farimani. Graph neural network-accelerated lagrangian fluid simulation. *Computers & Graphics*, 103:201–211, 2022.

[7] M. Raissi, P. Perdikaris, and G.E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

[8] Sai Gokul Subraveti, Zukui Li, Vinay Prasad, and Arvind Rajendran. Physics-based neural networks for simulation and synthesis of cyclic adsorption processes. *Industrial & Engineering Chemistry Research*, 61(11):4095–4113, 2022.

[9] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. Cambridge University Press, 2023. `https://D2L.ai`.

[10] Jan Kukacka, Vladimir Golkov, and Daniel Cremers. Regularization for deep learning: A taxonomy. *CoRR*, abs/1710.10686, 2017.

[11] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[12] Marian Tietz, Thomas J. Fan, Daniel Nouri, Benjamin Bossan, and skorch Developers. *skorch: A scikit-learn compatible neural network library that wraps PyTorch*, July 2017.

[13] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning*. The MIT Press, 11 2005.

[14] S. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, March 1982.

[15] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, pages 226–231, 1996.

[16] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. Dbscan revisited, revisited: Why and how you should (still) use dbscan. *ACM Trans. Database Syst.*, 42(3), jul 2017.

[17] Ricardo J. G. B. Campello, Davoud Moulavi, and Joerg Sander. Density-Based Clustering Based on Hierarchical Density Estimates. *Advances in Knowledge Discovery and Data Mining*, pages 160–172, 2013.

[18] Leland McInnes, John Healy, and Steve Astels. HDBSCAN: Hierarchical density based clustering. *The Journal of Open Source Software*, 2(11), March, 2017 2017.

[19] L. Hubert and P. Arabie. Comparing partitions. *Journal of Classification*, 2:193–218, 1985.

[20] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65, 1987.

[21] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[22] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13(null):281–305, feb 2012.

[23] Fernando Nogueira. Bayesian Optimization: Open source constrained global optimization tool for Python, 2014–.

[24] J.D. CLAYTON and P. Chung. *Applied Finite Element Methods: Lecture Notes on Principles and Procedures*. CreateSpace Independent Publishing Platform, 2018.

[25] Boris Delaunay. Sur la sphère vide. *Bulletin de l'Académie des Sciences de l'URSS, Classe des Sciences Mathématiques et Naturelles*, 6:793–800, 1934.

[26] Nico Schlömer. pygmsh: A Python frontend for Gmsh (v7.1.17). *Zenodo*, 2022.

[27] Christophe Geuzaine1 and Jean-Françoise Remacl. Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009.

[28] Nico Schlömer. meshio: Tools for mesh files (v5.3.4). *Zenodo*, 2022.

[29] Alexandre Joel Chorin. Numerical solution of the Navier-Stokes equations. *Mathematics of Computation*, 22:745–762, 1968.

[30] Ali Usman, Muhammad Rafiq, Muhammad Saeed, Ali Nauman, Andreas Almqvist, and Marcus Liwicki. Machine learning computational fluid dynamics. In *2021 Swedish Artificial Intelligence Society Workshop (SAIS)*, pages 1–4, 2021.

[31] N. D. Brenowitz and C. S. Bretherton. Prognostic validation of a neural network unified physics parameterization. *Geophysical Research Letters*, 45(12):6289–6298, 2018.

[32] Seowoo Jang, Soyoung Yoo, and Namwoo Kang. Generative design by reinforcement learning: Enhancing the diversity of topology optimization designs. *Computer-Aided Design*, 146:103225, 2022.

[33] Kikuo Fujita, Kazuki Minowa, Yutaka Nomaguchi, Shintaro Yamasaki, and Kentaro Yaji. Design Concept Generation With Variational Deep Embedding Over Comprehensive Optimization. In *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, volume Volume 3B: 47th Design Automation Conference (DAC), 08 2021. V03BT03A038.

[34] Amin Heyrani Nobari, Muhammad Fathy Rashad, and Faez Ahmed. CreativeGAN: Editing Generative Adversarial Networks for Creative Design Synthesis. In *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, volume Volume 3A: 47th Design Automation Conference (DAC), 08 2021. V03AT03A002.

[35] Antonio Alguacil, Wagner Gonçalves Pinto, Michael Bauerheim, Marc C. Jacob, and Stéphane Moreau. Effects of boundary conditions in fully convolutional networks for learning spatio-temporal dynamics, 2021.

[36] Pranshu Pant, Ruchit Doshi, Pranav Bahl, and Amir Barati Farimani. Deep learning for reduced order modelling and efficient temporal evolution of fluid simulations. *Physics of Fluids*, 33(10):107101, 10 2021.

[37] Octavi Obiols-Sales, Abhinav Vishnu, Nicholas Malaya, and Aparna Chandramowliswharan. Cfdnet: A deep learning-based accelerator for fluid simulations. In *Proceedings of the 34th ACM International Conference on Supercomputing*, ICS '20, New York, NY, USA, 2020. Association for Computing Machinery.

[38] John S Gero. Design prototypes: a knowledge representation schema for design. *AI magazine*, 11(4):26–26, 1990.

[39] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[40] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality, 2013.

[41] Chengwei Zhang, Youngwook Paul Kwon, Julia Kramer, Euiyoung Kim, and Alice M. Agogino. Concept Clustering in Design Teams: A Comparison of Human and Machine Clustering. *J. Mech. Des.*, 139(11):111414, October 2017.

[42] Seyoung Park and Harrison M. Kim. Phrase Embedding and Clustering for Sub-Feature Extraction From Online Data. *Journal of Mechanical Design*, 144(5), 12 2021. 054501.

[43] Faez Ahmed, Mark Fuge, and Lev D. Gorbunov. Discovering Diverse, High Quality Design Ideas From a Large Corpus. In *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, volume Volume 7: 28th International Conference on Design Theory and Methodology, 08 2016. V007T06A008.

[44] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. Introduction to Information Retrieval. *Cambridge University Press*, 2008.

[45] Masanobu Horie and Naoto Mitsume. Physics-embedded neural networks: Graph neural pde solvers with mixed boundary conditions, 2023.

[46] Grant M. Rotskoff, Andrew R. Mitchell, and Eric Vanden-Eijnden. Active importance sampling for variational objectives dominated by rare events: Consequences for optimization and generalization, 2021.

[47] R.E. Bellman. *Dynamic programming*. Princeton University Press, Princeton, NY, 1957.

[48] Harold Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of Educational Psychology*, 24:417–441, 1933.

[49] Shmuel Winograd. On computing the discrete fourier transform. *Proceedings of the National Academy of Sciences*, 73(4):1005–1006, 1976.

[50] Oren Rippel, Jasper Snoek, and Ryan P. Adams. Spectral representations for convolutional neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, page 2449–2457, Cambridge, MA, USA, 2015. MIT Press.

[51] Mark A. Kramer. Nonlinear principal component analysis using autoassociative neural networks. *AIChE Journal*, 37(2):233–243, 1991.

[52] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[53] Rini Jasmine Gladstone, Mohammad Amin Nabian, Vahid Keshavarzzadeh, and Hadi Meidani. Robust topology optimization using variational autoencoders, 2021.

[54] Tinghao Guo, Danny J. Lohan, Ruijin Cang, Max Yi Ren, and James T. Allison. *An Indirect Design Representation for Topology Optimization Using Variational Autoencoder and Style Transfer*. American Institute of Aeronautics and Astronautics, 2018.

[55] K. Ho-Le. Finite element mesh generation methods: a review and classification. *Computer-Aided Design*, 20(1):27–38, 1988.

[56] P. M. Finnigan, A. Kela, and J. E. Davis. Geometry as a basis for finite element automation. *Engineering with Computers*, 5(3):147–160, Jun 1989.

[57] M. Viceconti, M. Davinelli, F. Taddei, and A. Cappello. Automatic generation of accurate subject-specific bone finite element models to be used in clinical studies. *Journal of Biomechanics*, 37(10):1597–1605, Oct 2004.

[58] Lu Sun, Guoqun Zhao, and Gour-Tsyh Yeh. An automatic quadrilateral mesh generation algorithm applied to 2-d overland flow simulations. *Computational Geosciences*, 22(5):1283–1303, Oct 2018.

[59] Anders Logg. Automating the finite element method. *Archives of Computational Methods in Engineering*, 14(2):93–138, Jun 2007.

[60] Anders Logg and Garth N. Wells. DOLFIN: automated finite element computing. *CoRR*, abs/1103.6248, 2011.

[61] Martin S Alnæs, Anders Logg, Kristian B Ølgaard, Marie E Rognes, and Garth N Wells. Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Transactions on Mathematical Software (TOMS)*, 40(2):9, 2014.

[62] Robert Cimrman. SfePy - write your own FE application. In Pierre de Buyl and Nelle Varoquaux, editors, *Proceedings of the 6th European Conference on Python in Science (EuroSciPy 2013)*, pages 65–70, 2014. http://arxiv.org/abs/1404.6391.

[63] Python FEM and Multiphysics Simulations with FEniCS and FEATool. On the WWW, June 2017. URL `www.featool.com/`.

[64] Qingfeng Xia. Automated Mechanical Engineering Design using Open Source CAE Software Packages. In *FEniCS '18, Oxford, UK, June*, 2017. URL `github.com/qingfengxia/FenicsSolver`.

[65] Daryl L. Logan. *A First Course in the Finite Element Method*. Cengage Learning, Boston, MA, 2014.

[66] Nehzat Emamy, Martin Karcher, Roozbeh Mousavi, and Martin Oberlack. A high-order fully coupled electro-fluid-dynamics solver for multiphase flow simulations. In *Proceedings of the VI international conference on coupled problems in science and engineering*, pages 753–9, San Servolo, Venice, Italy, 05 2015.

[67] *McGraw-Hill Dictionary of Scientific and Technical Terms*. McGraw-Hill Dictionary of Scientific & Technical Terms. McGraw-Hill Education, 2003.

[68] *Chapter 1: Numerical Algorithms*, chapter 1, pages 1–15. 2011.

[69] L.H. Queiroz, F.P. Santos, J.P. Oliveira, and M.B. Souza. Physics-informed deep learning to predict flow fields in cyclone separators. *Digital Chemical Engineering*, 1:100002, 2021.

[70] Wei Chen, Kevin Chiu, and Mark D. Fuge. Airfoil design parameterization and optimization using bézier generative adversarial networks. *AIAA Journal*, 58(11):4723–4735, 2020.

[71] Charles Yang, Youngsoo Kim, Seunghwa Ryu, and Grace X. Gu. Prediction of composite microstructure stress-strain curves using convolutional neural networks. *Materials & Design*, 189:108509, 2020.

[72] Vinothkumar Sekar, Qinghua Jiang, Chang Shu, and Boo Cheong Khoo. Accurate near wall steady flow field prediction using physics informed neural network (pinn), 2022.

[73] Brian A. Freno and Kevin T. Carlberg. Machine-learning error models for approximate solutions to parameterized systems of nonlinear equations. *Computer Methods in Applied Mechanics and Engineering*, 348:250–296, 2019.

[74] Eric J. Parish and Kevin T. Carlberg. Time-series machine-learning error models for approximate solutions to parameterized dynamical systems. *Computer Methods in Applied Mechanics and Engineering*, 365:112990, 2020.

[75] Christian Beck, Martin Hutzenthaler, Arnulf Jentzen, and Benno Kuckuck. An overview on deep learning-based approximation methods for partial differential equations. *Discrete and Continuous Dynamical Systems - B*, 28(6):3697–3746, 2023.

[76] Iman Makaremi, Alireza Fatehi, and Babak Nadjar Araabi. Lipschitz numbers: A medium for delay estimation. *IFAC Proceedings Volumes*, 41(2):7468–7473, 2008. 17th IFAC World Congress.

[77] Kevin Chiu, David Anderson, and Mark Fuge. Automatically discovering mechanical functions from physical behaviors via clustering. *ASME IDETC*, 2021.

[78] Manyu Xiao, Dongchen Lu, Piotr Breitkopf, Balaji Raghavan, Subhrajit Dutta, and Weihong Zhang. On-the-fly model reduction for large-scale structural topology optimization using principal components analysis. *Structural and Multidisciplinary Optimization*, 62:209–230, 2020.

[79] Alec Van Slooten and Mark Fuge. Effect of optimal geometries and performance parameters on airfoil latent space dimension. In *ASME 2022 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. American Society of Mechanical Engineers Digital Collection, 2022.

[80] Wei Chen, Kevin Chiu, and Mark Fuge. Aerodynamic design optimization and shape exploration using generative adversarial networks. In *AIAA SciTech Forum*, San Diego, USA, Jan 2019. AIAA.

[81] Andrea Montanino, Gianluca Alaimo, and Ettore Lanzarone. A gradient-based optimization method with functional principal component analysis for efficient structural topology optimization. *Structural and Multidisciplinary Optimization*, 64:177–188, 2021.

[82] Min Li, Zhibao Cheng, Gaofeng Jia, and Zhifei Shi. Dimension reduction and surrogate based topology optimization of periodic structures. *Composite Structures*, 229:111385, 2019.

[83] Steven H. Berguin and Dimitri N. Mavris. *Dimensionality Reduction In Aerodynamic Design Using Principal Component Analysis With Gradient Information*. American Institute of Aeronautics and Astronautics, 2014.

[84] Xiangrui Zeng, Miguel Ricardo Leung, Tzviya Zeev-Ben-Mordehai, and Min Xu. A convolutional autoencoder approach for mining features in cellular electron cryo-tomograms and weakly supervised coarse segmentation, 2018.

[85] Muzhir Al-Ani and Fouad Awad. The jpeg image compression algorithm. *International Journal of Advances in Engineering & Technology*, 6:1055–1062, 05 2013.

[86] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2013.

[87] Hui Zou and Trevor Hastie. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(2):301–320, 2005.

[88] Moritz Sieber, Florian Ostermann, Rene Woszidlo, Kilian Oberleithner, and C. Oliver Paschereit. Lagrangian coherent structures in the flow field of a fluidic oscillator. *Phys. Rev. Fluids*, 1:050509, Sep 2016.

[89] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.

[90] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2018.

[91] Ayya Alieva, Dmitrii Kochkov, Jamie Alexander Smith, Michael Brenner, Qing Wang, and Stephan Hoyer. Machine learning accelerated computational fluid dynamics. *Proceedings of the National Academy of Sciences USA*, 2021.

[92] Shagun Sharma and Kalpna Guleria. Deep learning models for image classification: Comparison and applications. In *2022 2nd International Conference on Advance Computing and Innovative Technologies in Engineering (ICACITE)*, pages 1733–1738, 2022.

[93] Sven Apel, Christian Kästner, and Eunsuk Kang. Feature interactions on steroids: On the composition of ml models. *IEEE Software*, 39(3):120–124, 2022.

[94] Badra Souhila Guendouzi, Samir Ouchani, Hiba EL Assaad, and Madeleine EL Zaher. A systematic review of federated learning: Challenges, aggregation methods, and development tools. *Journal of Network and Computer Applications*, page 103714, 2023.

[95] Ammar Mohammed and Rania Kora. A comprehensive review on ensemble deep learning: Opportunities and challenges. *Journal of King Saud University-Computer and Information Sciences*, 2023.

[96] Radoslav Ivanov, Kishor Jothimurugan, Steve Hsu, Shaan Vaidya, Rajeev Alur, and Osbert Bastani. Compositional learning and verification of neural network controllers. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(5s):1–26, 2021.

[97] Ameya Dilip Jagtap and George E. Karniadakis. Extended physics-informed neural networks (xpinns): A generalized space-time domain decomposition based deep learning framework for nonlinear partial differential equations. *Communications in Computational Physics*, 2020.

[98] Gary F Marcus. Rethinking eliminative connectionism. *Cognitive psychology*, 37(3):243–282, 1998.

[99] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.

[100] Brenden M Lake. Compositional generalization through meta sequence-to-sequence learning. *Advances in neural information processing systems*, 32, 2019.

[101] Tomas Mikolov, Armand Joulin, and Marco Baroni. A roadmap towards machine intelligence, 2016.

[102] Brenden Lake and Marco Baroni. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *International conference on machine learning*, pages 2873–2882. PMLR, 2018.

[103] Christian Bauckhage, César Ojeda, Jannis Schücker, Rafet Sifa, and Stefan Wrobel. Informed machine learning through functional composition. In *LWDA*, pages 33–37, 2018.

[104] Qiuyi Chen, Jun Wang, Phillip Pope, Wei Chen, and Mark Fuge. Inverse design of two-dimensional airfoils using conditional generative models and surrogate log-likelihoods. *Journal of Mechanical Design*, 144(2):021712, 2022.

[105] Steven L Brunton, Bernd R Noack, and Petros Koumoutsakos. Machine learning for fluid mechanics. *Annual review of fluid mechanics*, 52:477–508, 2020.

[106] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.

[107] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.

[108] Peter Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, et al. Interaction networks for learning about objects, relations and physics. In *Advances in neural information processing systems*, pages 4502–4510, 2016.

[109] Alvaro Sanchez-Gonzalez, Nicolas Heess, Jost Tobias Springenberg, Josh Merel, Martin Riedmiller, Raia Hadsell, and Peter Battaglia. Graph networks as learnable physics engines for inference and control. *arXiv preprint arXiv:1806.01242*, 2018.

[110] Yunzhu Li, Jiajun Wu, Jun-Yan Zhu, Joshua B Tenenbaum, Antonio Torralba, and Russ Tedrake. Propagation networks for model-based control under partial observation. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 1205–1211. IEEE, 2019.

[111] Nicholas Watters, Daniel Zoran, Theophane Weber, Peter Battaglia, Razvan Pascanu, and Andrea Tacchetti. Visual interaction networks: Learning a physics simulator from video. In *Advances in neural information processing systems*, pages 4539–4547, 2017.

[112] Yonghong Chen, Hao Li, Han Li, Wenhao Liu, Yirui Wu, Qian Huang, and Shaohua Wan. An overview of knowledge graph reasoning: key technologies and applications. *Journal of Sensor and Actuator Networks*, 11(4):78, 2022.

[113] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[114] R.J. Trudeau. *Introduction to Graph Theory*. Dover Books, 2001.

[115] W.L. Hamilton. *Graph Representation Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2020.

[116] Yubiao Sun, Ushnish Sengupta, and Matthew Juniper. Physics-informed deep learning for simultaneous surrogate modeling and pde-constrained optimization of an airfoil geometry. *Computer Methods in Applied Mechanics and Engineering*, 411:116042, 2023.

[117] Tor Vestad, David W. M. Marr, and Toshinori Munakata. Flow resistance for microfluidic logic operations. *Applied Physics Letters*, 84(25):5074–5075, 2004.

[118] James Weaver, Jessica Melin, Don Stark, Stephen Quake, and Mark Horowitz. Static control logic for microfluidic devices using pressure-gain valves. *Nature Physics*, 6, 01 2010.

[119] Manu Prakash and Neil Gershenfeld. Microfluidic Bubble Logic. *Science*, 315(5813):832–835, 2007.

[120] Vakhtang Jandieri, Ramaz Khomeriki, and Daniel Erni. Realization of true all-optical AND logic gate based on nonlinear coupled air-hole type photonic crystal waveguides. *Optics express*, 26(16):19845–19853, 2018.

[121] Liqiang Wang, Kun Qian, Yan Huang, Nana Jin, Hongyan Lai, Ting Zhang, Chunhua Li, Chunrui Zhang, Xiaoman Bi, Deng Wu, et al. SynBioLGDB: a resource for experimentally validated logic gates in synthetic biology. *Scientific reports*, 5:80–90, 2015.

[122] T Monz, K Kim, W Hänsel, M Riebe, AS Villar, P Schindler, M Chwalla, M Hennrich, and R Blatt. Realization of the quantum Toffoli gate with trapped ions. *Physical review letters*, 102(4):040501, 2009.

[123] Xiao-Feng Shi. Deutsch, Toffoli, and CNOT Gates via Rydberg Blockade of Neutral Atoms. *Physical Review Applied*, 9(5):051001, 2018.

[124] Philip G. Neudeck, Roger D. Meredith, Liangyu Chen, David J. Spry, Leah M. Nakley, and Gary W. Hunter. Prolonged silicon carbide integrated circuit operation in Venus surface atmospheric conditions. *AIP Advances*, 6(12):125119, 2016.

[125] Derek S. Linden. Antenna design using genetic algorithms. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, GECCO'02, page 1133–1140, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[126] Ruizhen Hu, Oliver van Kaick, Bojian Wu, Hui Huang, Ariel Shamir, and Hao Zhang. Learning how objects function via co-analysis of interactions. *ACM Trans. Graph.*, 35(4), jul 2016.

[127] Yining Hong, Kaichun Mo, Li Yi, Leonidas Guibas, Antonio Torralba, Joshua Tenenbaum, and Chuang Gan. Fixing malfunctional objects with learned physical simulation and functional prediction. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2022.

[128] He Wang, S"oren Pirk, Ersin Yumer, Vladimir Kim, Ozan Sener, Srinath Sridhar, and Leonidas Guibas. Learning a generative model for multi-step human-object interactions from videos. In *Computer Graphics Forum*, 2019.

[129] R. C. Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, 1957.

[130] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):1–38, 1977.

[131] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[132] Bernhard Schölkopf, Alexander Smola, and Klaus-Robert Müller. Nonlinear Component Analysis as a Kernel Eigenvalue Problem. *Neural Computation*, 10(5):1299–1319, 07 1998.

[133] Quynh M. Nguyen, Joanna Abouezzi, and Leif Ristroph. Early turbulence and pulsatile flows enhance diodicity of Tesla's macrofluidic valve. *Nat. Commun.*, 12:2884, May 2021.