

THE PENNSYLVANIA STATE UNIVERSITY  
SCHREYER HONORS COLLEGE

DEPARTMENT OF COMPUTER SCIENCE

Using Physics-Informed Generative Adversarial Networks to Enhance Multiphase Fluid  
Simulation

MATTHEW LI  
SPRING 2021

A thesis  
submitted in partial fulfillment  
of the requirements  
for a baccalaureate degree  
in Computer Science  
with honors in Computer Science

Reviewed and approved\* by the following:

Christopher McComb  
Assistant Professor of Engineering Design  
Thesis Supervisor

Jesse Barlow  
Professor of Computer Science  
Honors Adviser

\* Electronic approvals are on file.



## ABSTRACT

Machine learning methods have been shown to demonstrate the ability to reconstruct single-phase turbulent fluid flow from low-resolution inputs, with potential applications in many industries, but especially in engineering design. However, no work thus far has explored the application of machine learning image super-resolution methods to multiphase fluid flow. In this work, we apply the Super-Resolution Generative Adversarial Network (SRGAN) model to a multiphase turbulent fluid flow problem, specifically to reconstruct fluid phase fraction at a higher resolution. Two models were created in this work, one with a simple physics-constrained loss function and one without, and the results are discussed and analyzed. We found that both models were able to significantly outperform non-machine learning upsampling methods and can preserve an impressive amount of detail and nuance, showing the versatility of the SRGAN model for upsampling fluid simulations. But, the difference in accuracy between the two models is quite minimal, whereas physics-informed models have shown better results than non-physics-informed models in past work with single-phase fluid flow. This result leads to some important points of discussion and room for future research on the topic.



## TABLE OF CONTENTS

Introduction.....	1
Background.....	3
The importance of CFD .....	3
Image Super-Resolution.....	4
Generative Adversarial Networks .....	5
Image Super-Resolution Applied to CFD .....	6
Methods.....	8
Problem Description.....	8
Using the OpenFOAM Software Library.....	9
Time Continuity Divergence Problems.....	14
Training Data Generation.....	16
Converting OpenFOAM data to a usable format .....	19
Network Architecture.....	23
Network Implementation .....	25
Loss Function Design.....	25
Results and Discussion .....	27
Conclusions and Future Work .....	37
Appendix A Code used to assemble OpenFOAM data into matrix format .....	39
Appendix B Code used to distribute computation of OpenFOAM simulations to multiple CPUs .....	43
Appendix C Code used to define the GAN model.....	45
Appendix D Code used to train GAN model.....	49
Appendix E Foam Case Class code; helper class for training GAN model.....	55
Appendix F Code used to evaluate GAN model.....	56
Appendix G Code used to implement loss function .....	61
Appendix H Optimized matrix functions and benchmark Code.....	62
Bibliography .....	65



## Acknowledgements

I would like to thank Professor Christopher McComb for his guidance and patience for the past few years, as well as my friends and family who have stuck with me for my entire undergraduate career.



## LIST OF FIGURES

FIGURE 1: ILLUSTRATION OF PROBLEM STATEMENT: UPSAMPLING FROM A $16 \times 16$ RECTILINEAR MESH TO A $64 \times 64$ RECTILINEAR MESH.....	8
FIGURE 2: EXCERPT OF DAMBREAK BLOCKMESHDICT FILE .....	10
FIGURE 3: REGIONS OF THE GRID MESH IN THE DAMBREAK SIMULATION .....	11
FIGURE 4: EXCERPT OF DAMBREAK CONTROLDICT FILE.....	12
FIGURE 5: EXCERPT OF DAMBREAK SETFIELDSDICT FILE .....	13
Figure 6: DAMBREAK SNAPSHOT AT 1.45 SECONDS AT $256 \times 256$ RESOLUTION.....	14
Figure 7: DAMBREAK SNAPSHOT AT 1.45 SECONDS AT $512 \times 512$ RESOLUTION.....	15
FIGURE 8: LOW-RESOLUTION AND HIGH RESOLUTION DAMBREAK CASE AT SAME TIME STEP. THE DAM IS DENOTED BY THE WHITE SQUARE. (YELLOW IS WATER, PURPLE IS AIR).....	17
FIGURE 9: FOUR DIFFERENT DAMBREAK CASES AT TIME = 0 SECONDS (YELLOW IS WATER, PURPLE IS AIR).....	18
FIGURE 10: PRE-GROUND-IMPACT DATA VS. POST-GROUND-IMPACT DATA (YELLOW IS WATER, PURPLE IS AIR) .....	18
FIGURE 11: EXCERPT FROM SAMPLE ALPHA.WATER FILE .....	19
FIGURE 12: ILLUSTRATION OF ARRANGEMENT OF OPENFOAM DATA POINTS .	20
FIGURE 13: SAMPLE OF ASSEMBLED 3D OPENFOAM DATA.....	22
FIGURE 14: NETWORK LAYER DIAGRAM. REPEATED BLOCKS ARE CONDENSED TO SAVE SPACE.....	24
FIGURE 15: TRAINING LOSSES FOR MSE-ONLY MODEL, INCLUDING DISCRIMINATOR LOSS (LEFT) AND GENERATOR LOSS (RIGHT).....	27
FIGURE 16: TRAINING LOSSES FOR MSE+PHYSICS MODEL, INCLUDING DISCRIMINATOR LOSS (LEFT) AND GENERATOR LOSS (RIGHT).....	28
FIGURE 17: EXAMPLES OF MODEL INFERENCE COMPARED TO LINEAR UPSAMPLING, BICUBIC UPSAMPLING, AND GROUND TRUTH HIGH-RESOLUTION SIMULATION.....	30
FIGURE 18: FIG17 DD ENHANCED .....	31
FIGURE 19: FIG 17 DE ENHANCED .....	31



FIGURE 20: FIGURE & DF ENHANCED .....	32
FIGURE 21: COMPARISON OF MAGNITUDE AND DISTRIBUTION OF LOSS VALUES ON A PURE MSE LOSS FUNCTION ON A LOG SCALE FOR THE DIFFERENT UPSAMPLING METHODS. THESE VALUES WERE TAKEN FROM A RANDOM SAMPLE OF 10000 DATA POINTS FROM THE DATA SET.....	33
FIGURE 22: COMPARISON OF MAGNITUDE AND DISTRIBUTION OF LOSS VALUES ON THE CUSTOM MSE + PHYSICS LOSS FUNCTION ON A LOG SCALE FOR THE DIFFERENT UPSAMPLING METHODS. THESE VALUES WERE TAKEN FROM A RANDOM SAMPLE OF 10000 DATA POINTS FROM THE DATA SET.....	34



## Introduction

Computational Fluid Dynamics, or CFD has been a crucial pillar of the engineering design process for decades. By dramatically reducing the number of physical tests and prototypes needed to fine-tune the design of a part or a system, CFD simulations can greatly reduce the cost for many engineering design tasks. They also provide engineers with much more flexibility, as fixing a failed simulation is much easier than fixing a failed physical prototype. CFD tools have become invaluable to many engineers, but current methods are not without their limitations.

The field of CFD is very broad, as fluid flow behaviors can manifest in a multitude of ways. In general, CFD simulations model fluid flow by splitting the simulation space into a mesh of discrete points over time and space and creating systems of differential equations which are solved or approximated using numerical computing methods. [1] To control the granularity of the CFD simulation, the density of the discrete cells can be changed. Having more points means more detail is captured at the cost of longer computation times, which increase with the number of points, forcing users to make a tradeoff between detail and computation time.

Deep learning methods have been applied to the field of CFD, and have been used to tradeoff with success. Deep neural networks can be used to up-sample CFD simulations with good results [2-8] showing that deep-learning accelerated CFD could be a promising field of research. We add to this burgeoning field of research by introducing a model with a new capability.



The majority of Super Resolution in CFD (SRCFD) papers published to date have been for single phase fluid simulations. The addition of an additional fluid phase into a CFD simulation can dramatically increase the complexity of the problem with a new set of equations, as well as adding phase fraction as a fluid property [2].

In this paper, we demonstrate the feasibility of the super-resolution of multiphase CFD simulations with a deep neural network. More specifically, we demonstrate the feasibility of using a deep neural network to approximate the Volume Of Fluids (VOF) numerical method of modelling multiphase flow. We present both a physics-informed and a non-physics-informed deep neural networks, and compare these against naïve upsampling approaches. The remainder of the paper is organized as follows: the background, which contains the motivation and some prerequisite information, followed by the methods and results.



## **Background**

### **The Importance of CFD**

Computational Fluid Dynamics, or CFD, is a field of study that combines numerical analysis and fluid mechanics, using numerical methods to solve problems that involve fluids. This is still a burgeoning new field, and thousands of scientists and engineers continue to actively study and create new research in CFD. Even though the fundamental fluid equations, such as the Navier-Stokes equation, had been derived during the 19th century, it wasn't until computers became reasonably fast in the 1950's that the age of modern CFD could truly begin [3].

The importance of CFD stems its ability to simulate physical experiments. For example, iterative airfoil design could be done very cheaply using CFD instead of a physical wind tunnel [4]. CFD is widely used in a variety of different industries, such as mechanical design [5,6], food science [7–9] and materials science [10,11]. Obviously, physical experiments can never be completely replaced by CFD simulations, but they are still immensely useful. CFD simulations can be performed on rougher prototypes, and these results of these simulations can be used to fine-tune the design until a physical prototype can be built for a real-world test [1].

By dramatically reducing the number of physical tests and prototypes needed to fine-tune the design of a part or a system, CFD simulations can greatly reduce the cost for many engineering design tasks. Over the decades, CFD has become an invaluable tool to engineers and designers, and any advancements made in the field will surely benefit the entire field of engineering.



## Image Super-Resolution

Image super-resolution has been a well-researched problem in the realm of Computer Vision for many decades now and is still actively being researched to this day. The problem is this: can additional fidelity and detail be inferred from a low-resolution image? Many models and methods have been developed so far with varying degrees of success, and progress is still being made.

A plethora of novel methods and architectures have been experimented with, such as deep Laplacian pyramid networks [13], dense skip connections [14], deep residual channel attention networks [15], etc., all with their own advantages and disadvantages. One of the most famous models developed, and the standard benchmark for models in this field of research, is the Super Resolution Convolutional Neural Network (SRCNN). This model by Dong [16] uses a convolutional neural network to create a mapping between low-resolution and high-resolution images. This model shows promising results for up-sampling images, but it fails to capture higher-level features and instead minimizes a lower-level pixel-wise loss. Further refinements to this model have been made with improvements to efficiency in the form of FSRCNN (Fast SRCNN) [17], as well as an increase in depth with VDSR (Very Deep Super Resolution) [18]. Both these works showed some improvements over SRCNN.

A much more powerful model, using a generative adversarial network architecture, is the Super Resolution GAN (SRGAN), created by Ledig [19]. The SRGAN architecture will serve as the cornerstone for our work. Compared to SRCNN, SRGAN is much more powerful in capability and can infer very high-level features, leading to much more natural and realistic-looking images.



This also comes at the cost of having many times more weights, which makes training and inference much less efficient.

## **Generative Adversarial Networks**

The Generative Adversarial Network (GAN) is a novel architecture that involves training two separate deep neural networks against each other. One network, the generator, is responsible for generating the actual output. The other network, the discriminator, is responsible for distinguishing between the generator's output and the ground truth data. During the training process, the generator will get better at producing outputs that resemble the ground truth, and the discriminator will become better at discerning them. [20] Ideally, the generator should become strong enough to fool the discriminator. The resulting generator network can then be used for inference, while the discriminator is discarded in the final model.

The versatility of the GAN architecture has led to its application to a multitude of different problems. GANs have been demonstrated to produce remarkably good results in novel tasks, such as generating human faces [21] and transforming photographs into paintings [22]. They have also proven to be useful for many specific and practical engineering applications, such as designing airfoils [23], predicting stress distribution in structures [24], and estimating leakage parameters for liquid pipelines [25]. However, GANs can be large in size and difficult to train, and are vulnerable to mode collapse and vanishing gradients [26]. Their large number of parameters also leads to slower inference, making them less suitable for time sensitive tasks [27].



## Image Super-Resolution Applied to CFD

Image super-resolution techniques can be applied somewhat directly to CFD simulations. The resolution of a CFD simulation lies in the density of points, which is functionally analogous to pixels in an image. Researchers have recently begun applying machine-learning based image SR techniques to a variety of different types of CFD simulations, with varying degrees of success. The work of Fukami [28] was one of the earlier works in this field, and showed that turbulent flows were able to be reconstructed with some success using a relatively simple convolutional neural network as well as a hybrid downsampled skip-connection multi-scale model (DSC/MS). This work was one of the first to show that certain types of fluid flows can be reconstructed with machine learning methods. Another work by Liu [29], produced further results with CNNs as well as a multiple temporal paths convolutional network (MTPC). They were able to show that deep learning methods outperformed traditional upsampling methods like bicubic interpolation, but were still limited in their capabilities to produce results that follow the physical constraints of fluids. [29] The generative adversarial network architecture (GAN) was also applied to the CFD super-resolution problem with great results, in the work by Xie [30] with a novel application in 3D smoke diffusion.

An even more recent innovation is the use of physics-informed networks, i.e.- explicitly using physical properties in the loss functions to provide more context to the models. We have already seen that physics-constrained neural networks produce good results for other applications, such as the modelling of materials [31]. Can this same philosophy be applied to CFD super-resolution? Two very recent models that are designed to solve this problem for single-phase



turbulent flow CFD simulations are MeshfreeFlowNet by Jiang et. al. and TEGAN by Subramaniam et. al.

TEGAN is similar to our model, as it is also based on SRGAN, but was designed to solve a single-phase fluid flow super-resolution problem. It was trained to up-sample instances of the incompressible forced isotropic homogeneous turbulence problem in 3D and is constrained by the time dependent Navier-Stokes equations and the Poisson equation. [32] This model can up-sample both pressure and velocity for four total fields in 3D (pressure, and x, y, z velocity), and provides impressive results across all four fields. This paper also provides another interesting result, which involves experimentation with the value of the weight given to the physics-based loss functions. The paper shows that adding a non-zero weight to the physics loss greatly improved the results but too high of a weight may be detrimental. This result is extremely important, not only legitimizing the use of physics-based loss functions for this field, but also gives some data points for other researchers to use while tuning the hyperparameters of their own models.



## Methods

### Problem Description

The goal of the current work is to use image SR techniques to increase the resolution of the fluid phase fraction generated by a solver that uses the Volume Of Fluids (VOF) method of modelling two-phase incompressible turbulent fluid flow. We make use of the InterFoam solver from OpenFOAM [35], an open-source and highly performant CFD software library commonly used in both industry and research [36]. Upsampling the output approximates the results of increasing the density of the discretized mesh, and ideally will produce additional detail without incurring the cost of computation that comes with direct numerical solutions. To make our CFD simulations suitable for image SR models, we chose to discretize them with a uniform rectilinear mesh, which ensures the computational analogy to pixels typically used in SR. In this work only the fluid phase fraction is considered. The model trained in this work specifically upsamples from a mesh resolution of  $16 \times 16$  to  $64 \times 64$  (see Figure 1).

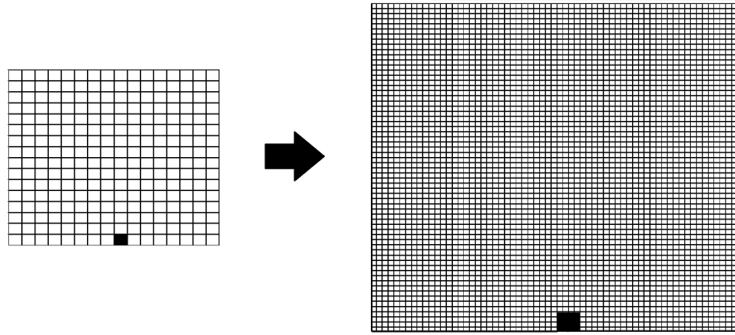


FIGURE 1: ILLUSTRATION OF PROBLEM STATEMENT: UPSAMPLING FROM A  $16 \times 16$  RECTILINEAR MESH TO A  $64 \times 64$  RECTILINEAR MESH



## Using the OpenFOAM Software Library

The Open-Source Field Operation and Manipulation library, or OpenFOAM library is an open-source software library that provides an object-oriented and highly efficient family of solvers for fluid dynamics problems and an easy-to-use command-line interface. [34] This provides and easily scriptable and automatable CFD environment for efficient data generation. For this work, the “interFoam” solver was used. This solver utilizes an algorithm based on the volume of fluid method (VOF) to calculate the ratio of two fluid phases at points in the mesh. This ratio is called the phase fraction, or  $\alpha$ , and predicting  $\alpha$  at upscaled resolutions is the aim of this work. The phase fraction essentially keeps track of the position of water and air in the simulation.

Each solver is an executable program that can be called using the command-line, and configuration of OpenFOAM simulations is done by modifying several different config files. The directory structure of an OpenFOAM simulation will begin with the “constant”, “postprocessing”, and “system” folders, as well as a folder for each iterative timestep of the simulation. The training data for the model will be extracted from these timestep directories. All the configuration files to set the initial conditions are in the “constant” and “system” folders, with the most important being “blockMeshDict”, “controlDict” and “setFieldsDict” from the system folder. The constant folder contains settings that did not need to be modified for this work, such as the acceleration of gravity and some other properties.

The “blockMeshDict” file contains the dimensions and density of the mesh grid of the simulation. The general file format contains a section containing 3-tuples representing the vertices, and other sections containing the arrangement of blocks, edges, and boundaries. Other more



complex geometries can be specified in this file, such as 3-dimensional surfaces, but these are not needed in this work. An excerpt of a sample “blockMeshDict” file is shown below.

```

1  /*-----* C++ -*-----*\
2  | ===== |
3  | \ \ / F i e l d | OpenFOAM: The Open Source CFD Toolbox |
4  | \ \ / O p e r a t i o n | Version: v1812 |
5  | \ \ / A n d | Web: www.OpenFOAM.com |
6  | \ \ / M a n i p u l a t i o n | |
7  \*-----*/
8  FoamFile
9  {
10     version      2.0;
11     format        ascii;
12     class         dictionary;
13     object        blockMeshDict;
14 }
15 // *****
16
17 scale 0.146;
18
19 vertices
20 (
21     (0 0 0)
22     (2 0 0)
23     (2.25 0 0)
24     (4 0 0)
25     (0 0.25 0)
26     (2 0.25 0)
27     (2.25 0.25 0)
28     (4 0.25 0)
29     (0 4 0)
30     (2 4 0)
31     (2.25 4 0)
32     (4 4 0)
33     (0 0 0.1)
34     (2 0 0.1)
35     (2.25 0 0.1)
36     (4 0 0.1)
37     (0 0.25 0.1)
38     (2 0.25 0.1)
39     (2.25 0.25 0.1)
40     (4 0.25 0.1)
41     (0 4 0.1)
42     (2 4 0.1)
43     (2.25 4 0.1)
44     (4 4 0.1)
45 );
46
47 blocks
48 (
49     hex (0 1 5 4 12 13 17 16) (16 2 1) simpleGrading (1 1 1)
50     hex (2 3 7 6 14 15 19 18) (14 2 1) simpleGrading (1 1 1)
51     hex (4 5 9 8 16 17 21 20) (16 30 1) simpleGrading (1 1 1)
52     hex (5 6 10 9 17 18 22 21) (2 30 1) simpleGrading (1 1 1)
53     hex (6 7 11 10 18 19 23 22) (14 30 1) simpleGrading (1 1 1)
54 );

```

FIGURE 2: EXCERPT OF DAMBREAK BLOCKMESHDICT FILE





Each entry of the blocks section contains three fields. The first is a list of the eight vertices from the vertices section that make up the vertices of the block, the second is the density of mesh cells per axis (x, y, and z), and the third field specifies the cell expansion ratios for each axis, which was left unmodified. The “simpleGrading” type means that cells will be uniformly distributed across each axis, which is ideal.

In the case of the DamBreak simulation, “blockMeshDict” contains the information for a 2-dimensional grid with an immovable dam in the center. To account for this, the mesh is split into five separate blocks and each one is configured independently, as shown in the figure below. Each of the five blocks shown in the below figure has an entry in the blocks section of the blockMeshDict file.

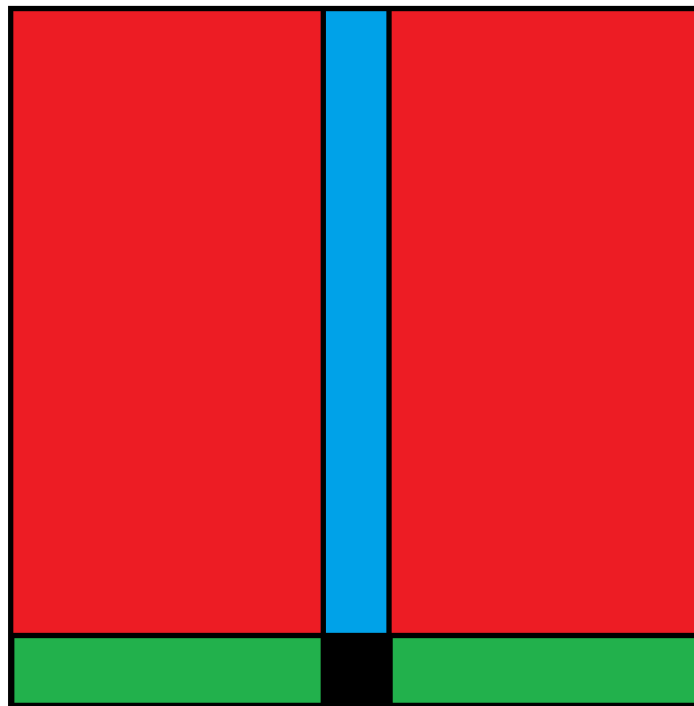


FIGURE 3: REGIONS OF THE GRID MESH IN THE DAMBREAK SIMULATION



FIGURE 3: REGIONS OF THE GRID MESH IN THE DAMBREAK SIMULATION

Next, the “controlDict” file contains all the configuration settings for the time and I/O control of a given simulation. An excerpt is included below. Essentially, this file is used to control the time length of the simulation, the write interval, and the numerical precision of these operations. There are finer settings and functions available, but they are not quite as important for this work. In this case shown in the figure, the simulation is ran for 3 seconds, with observations recorded every 0.05 seconds, resulting in a total of 60 timesteps worth of data.

```

2 | ===== |
3 | \ \      / F i e l d      | OpenFOAM: The Open Source CFD Toolbox |
4 | \ \      / O p e r a t i o n | Version: v1812 |
5 | \ \      / A n d      | Web: www.OpenFOAM.com |
6 | \ \      / M a n i p u l a t i o n |
7 | *-----*/
8 FoamFile
9 {
10     version      2.0;
11     format        ascii;
12     class         dictionary;
13     location      "system";
14     object        controlDict;
15 }
16 // * * * * *
17
18 application      interFoam;
19
20 startFrom        startTime;
21
22 startTime        0;
23
24 stopAt           endTime;
25
26 endTime          3;
27
28 deltaT           0.001;
29
30 writeControl      adjustableRunTime;
31
32 writeInterval     0.05;

```

FIGURE 4: EXCERPT OF DAMBREAK CONTROLDICT FILE



Finally, we have the “setFieldsDict” file, which sets the initial conditions of the fluids in the simulation. Different solvers may have different versions of this file, and the excerpt shown below is specific to the interFoam solver. Here, the water phase is defined as a box with two corners. Other geometries can be specified, but for this work only rectangular blocks are used. The rectangular block is defined by two points in 3D space, and its “volScalarFieldValue” is defined as 1, meaning the block of fluid is purely water. The default “volScalarFieldValue” is 0, meaning everything besides the water block is purely air.

```

1  /*-----*- C++ -*-----*/
2  | ===== |
3  | \ \ / F i e l d | OpenFOAM: The Open Source CFD Toolbox |
4  | \ \ / O p e r a t i o n | Version: v1812 |
5  | \ \ / A n d | Web: www.OpenFOAM.com |
6  | \ \ / M a n i p u l a t i o n | |
7  /*-----*/
8  FoamFile
9  {
10     version      2.0;
11     format        ascii;
12     class          dictionary;
13     location       "system";
14     object          setFieldsDict;
15 }
16 // *****
17
18 defaultFieldValues
19 (
20     volScalarFieldValue alpha.water 0
21 );
22
23 regions
24 (
25     boxToCell
26     {
27         box (0 25 -1) (19 27 1);
28         fieldValues
29         (
30             volScalarFieldValue alpha.water 1
31         );
32     }
33 );
34
35
36 // *****

```

FIGURE 5: EXCERPT OF DAMBREAK SETFIELSDICT FILE



## Time Continuity Divergence Problems

Originally, training data generation was achieved by running each DamBreak case in both the low and high resolutions, with the same starting conditions. It was thought that doing this would create identical fluid data for each resolution. However, when using this data for training, the performance was abysmal, and the resulting models were practically worthless. Upon further investigation, we found that for data points of the same case and at identical timesteps differed quite significantly. Included below is an example of this phenomenon, with two identical data points (run with different resolutions) rendered in ParaView.

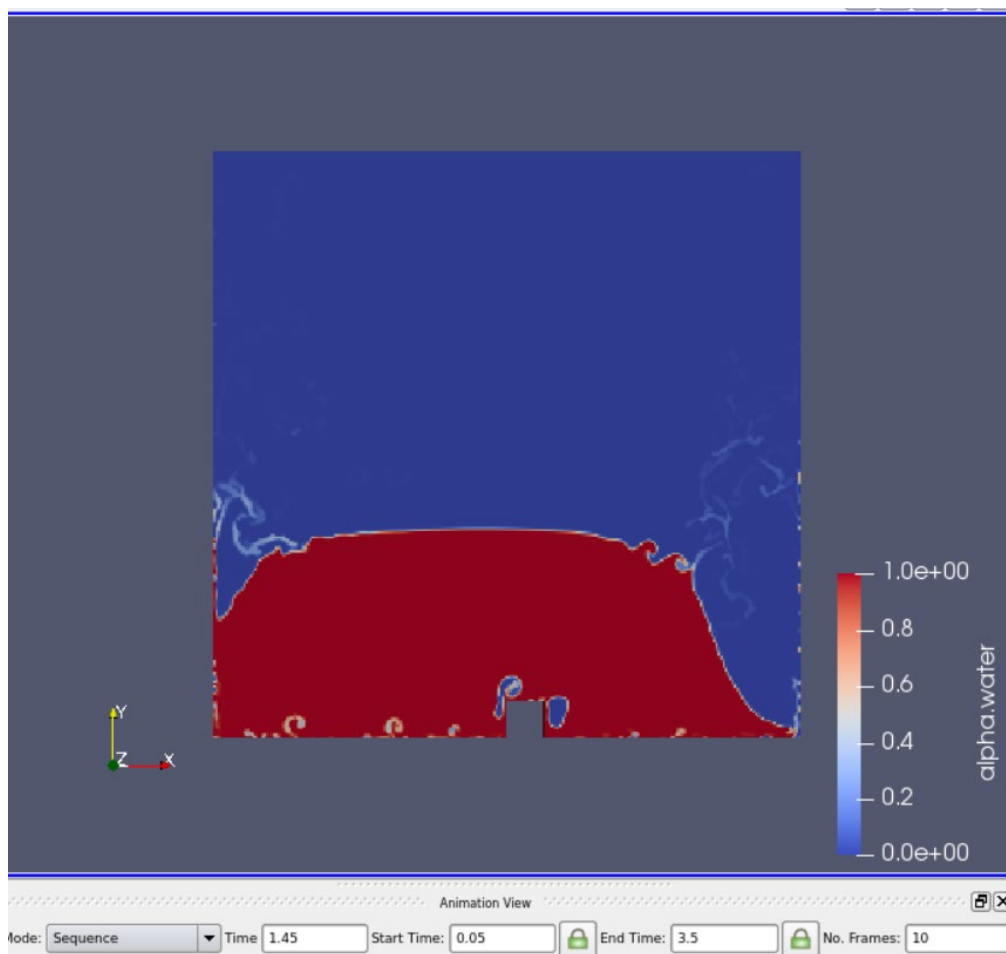


Figure 6: DAMBREAK SNAPSHOT AT 1.45 SECONDS AT 256x256 RESOLUTION

FIGURE 6: DAMBREAK SNAPSHOT AT 1.45 SECONDS AT 256x256 RESOLUTION



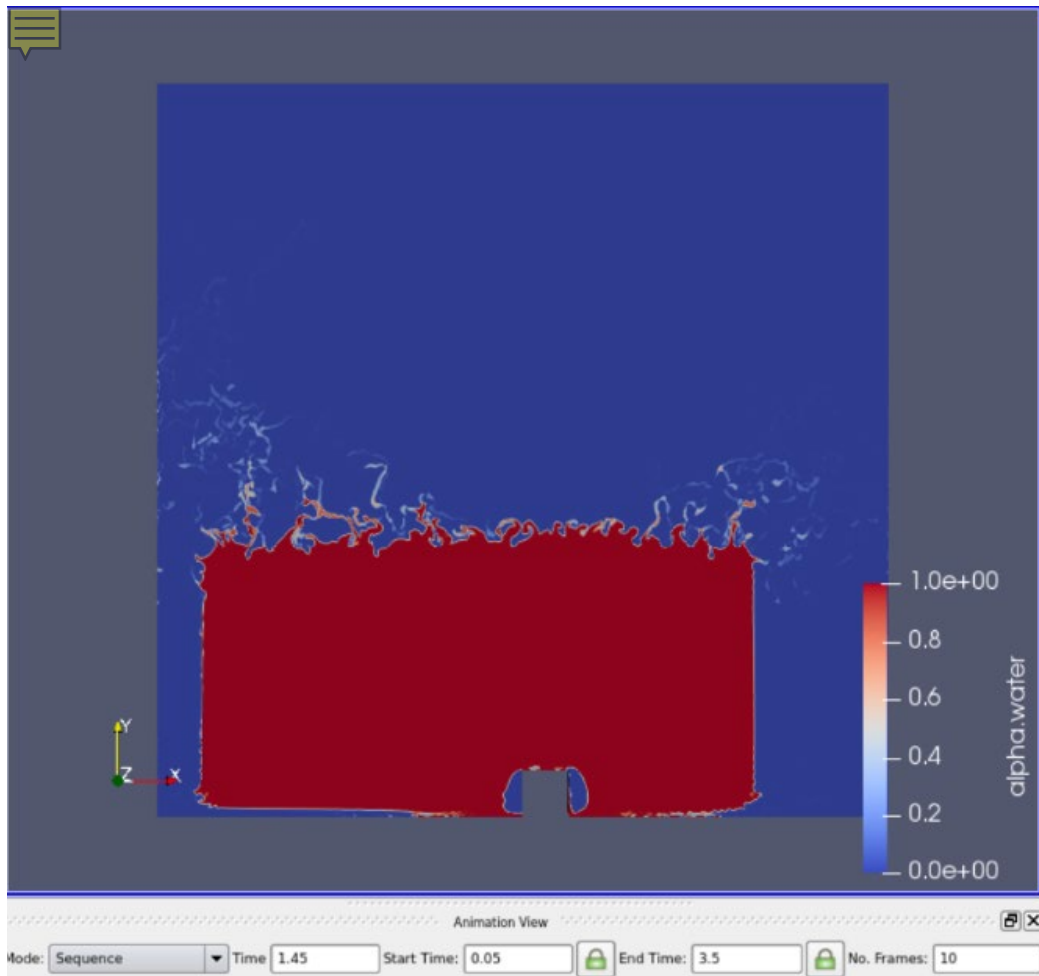


Figure 7: DAMBREAK SNAPSHOT AT 1.45 SECONDS AT 512x512 RESOLUTION

FIGURE 7: DAMBREAK SNAPSHOT AT 1.45 SECONDS AT 512x512 RESOLUTION

As seen in the figures, there appears to be a time continuity error. More specifically, the two simulations experience a sort of time divergence and are unsynchronized at the same time steps. This seemed to be an issue inherent to CFD itself, and the current hypothesis is that changing the mesh cell density (resolution) of the simulation correlates with an increase in the residual error accumulated during each iteration of the interFoam solver. This issue was bypassed by running only the high-resolution simulations and then creating the low-resolution data points by down-sampling. This is described in more detail in the following section.



## Training Data Generation

Training data was generated using OpenFOAM using the “DamBreak” case, a 2-dimensional interphase laminar flow simulation which depicts a mass of water falling from the air and onto the ground, then colliding with a solid immovable dam in the center of the floor. This creates a large amount of fluid movement for the model to learn and predict.

A total of 800 cases were simulated with a uniform mesh density of  $64 \times 64$  for 3 seconds with 60 discrete timesteps each (0.05s increments), with the fluid data at each timestep serving as a data point. Every timestep constitutes a unique training sample, resulting in 48,000 total data training samples. The initial position, size and shape of the water mass is randomized between cases, as shown in Figure 3. This was done by randomly changing the two defining points in the “setFieldsDict” file for each respective case directory. The rationale for this was to create a more varied set of training samples, as changing these conditions changes the amount of kinetic energy in the system, which could drastically change the fluid behavior between cases.

Each of the 800 cases were run using the interFoam solver as described in the previous section and were run using a multiprocessing queue in python for maximum throughput. OpenFOAM has options for multi-threaded solvers, such as domain decomposition, but they were not used for this work. Assigning each of the 800 cases to a single CPU core is sufficient to maximally utilize the hardware, and for throughput to be 100%. The code used to distribute these tasks to CPU cores is included in appendix B.



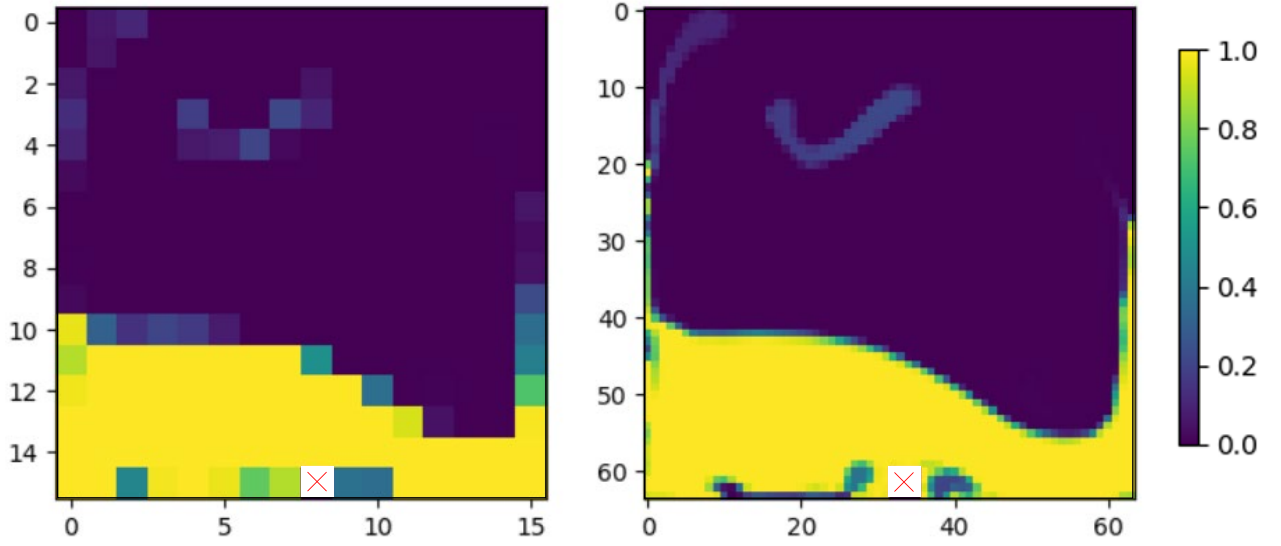


FIGURE 8: LOW-RESOLUTION AND HIGH RESOLUTION DAMBREAK CASE AT SAME TIME STEP. THE DAM IS DENOTED BY THE WHITE SQUARE. (YELLOW IS WATER, PURPLE IS AIR)

Each of these samples was then downsampled using linear interpolation, creating high- and low-resolution samples pairs (see Figure 8). This dataset was then split into two categories: pre-ground-impact and post-ground-impact (see Figure 10). In this work we focus on the post-ground impact dataset since most of the fluid deformation occurs after the fluid collides with the ground. The pre-ground-impact dataset can likely be predicted with much simpler methods (i.e., projectile motion) and is not used in this paper. The model presented is trained completely on the post-ground data. This post-ground dataset has 19,664 total data points, and 5000 were randomly samples for training and testing.



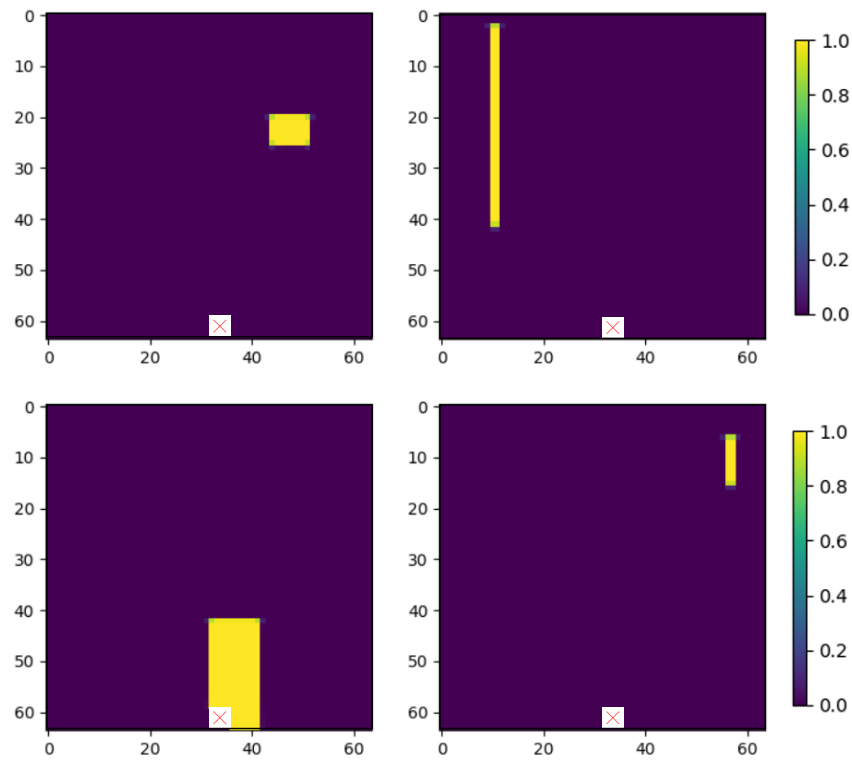


FIGURE 9: FOUR DIFFERENT DAMBREAK CASES AT TIME = 0 SECONDS (YELLOW IS WATER, PURPLE IS AIR)

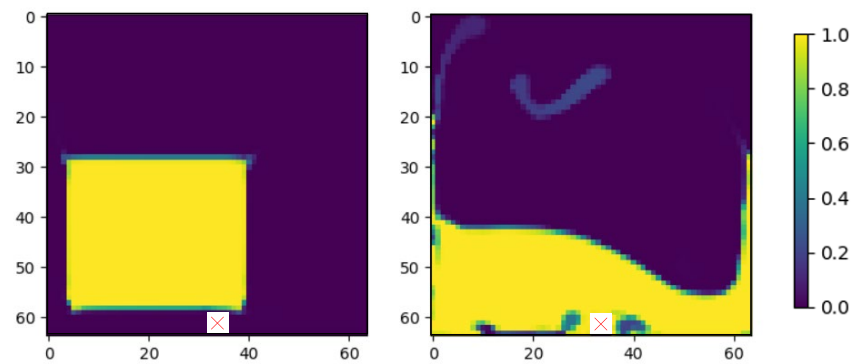


FIGURE 10: PRE-GROUND-IMPACT DATA VS. POST-GROUND-IMPACT DATA (YELLOW IS WATER, PURPLE IS AIR)



## Converting OpenFOAM data to a usable format

The OpenFOAM file format cannot be immediately substituted into image super resolution tasks. Images are stored as matrices of numbers, while the OpenFOAM data format stores data in simple lists. The OpenFOAM data needs to be converted into a matrix format to be useful. An excerpt of a sample “alpha.water” file is shown below. The “alpha.water” file contains the phase fraction values, with an entry for every mesh cell.

```

1  /*-----*-- C++ --*-----*\
2  | ===== |
3  | \ \      / F i e l d      | OpenFOAM: The Open Source CFD Toolbox |
4  | \ \      / O p e r a t i o n | Version: v1812 |
5  | \ \      / A n d      | Web: www.OpenFOAM.com |
6  | \ \ /      M a n i p u l a t i o n | |
7  \*-----*--\
8  FoamFile
9  {
10     version      2.0;
11     format        ascii;
12     class         volScalarField;
13     location      "0.05";
14     object        alpha.water;
15 }
16 // * * * * *
17
18 dimensions      [0 0 0 0 0 0 0];
19
20
21 internalField    nonuniform List<scalar>
22 4080
23 (
24 2.79584e-75
25 1.84191e-71
26 6.99637e-68
27 1.60097e-66
28 8.98945e-61
29 1.69403e-60
30 -1.16949e-63
31 -4.60252e-63

```

FIGURE 11: EXCERPT FROM SAMPLE ALPHA.WATER FILE



The points in the “alpha.water” file needs to be arranged in a matrix format, as these points alone do not have any positional information, and are of little use. They must be reconstructed using information from their blockMeshDict file, and follow a very specific format, which is better explained using the following figure.

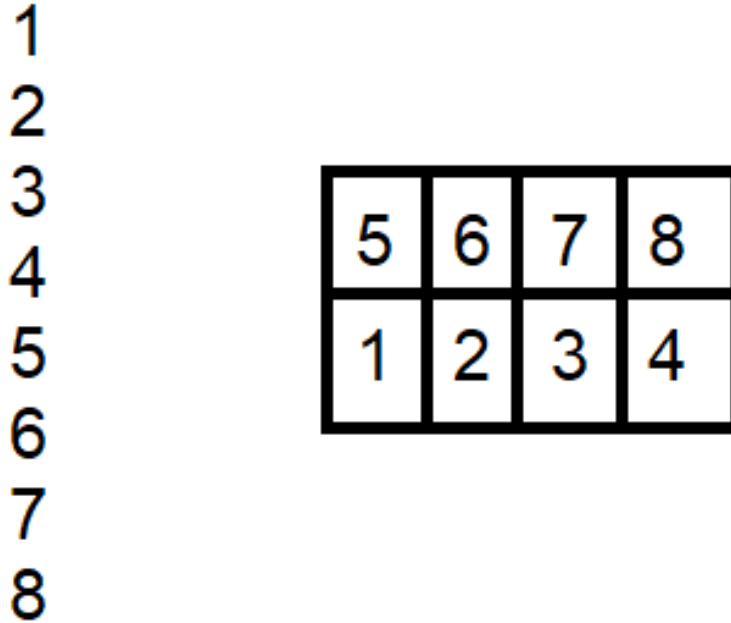


FIGURE 12: ILLUSTRATION OF ARRANGEMENT OF OPENFOAM DATA POINTS

The list of numbers on the left represents the OpenFOAM data format, and the grid on the right represents the reconstructed matrix. The data points need to be assembled from left to right, bottom to top, block by block, in the order defined in the “blockMeshDict” file. A small software library was created to help assemble OpenFOAM data into numpy files with a matrix format. A sample excerpt of this code is shown below, and the rest is included in the appendix.



```

#builds all the blocks in a given case into a single array, works for both 2d
#and 3d cases; does array arithmetic based on xyzmax and res (point density)
def buildarr(res, datafile, meshfile, channel=0):
    final = np.zeros(res)
    data = open(datafile, "r")
    mesh = open(meshfile, "r")
    points = getVertices(mesh)
    XYZmax = np.array(xyzmax(points))
    mul = list(map(int, np.divide(res, XYZmax)))
    blocks = getBlocks(mesh, points)
    for i in range(23):
        discard = data.readline()
    for i in range(len(blocks)):
        st = blocks[i][0][0]
        st = (st[1]*mul[1], st[0]*mul[0], st[2]*mul[2])
        st = list(map(int, st))
        blocks[i] = buildBlock(blocks[i], data, channel)
        shp = blocks[i].shape
        final[res[0]-(st[0]+shp[0]):res[0]-st[0], st[1]:st[1]+shp[1], \
                st[2]:st[2]+shp[2]] = blocks[i]

    return final

```

```

#builds all the blocks in a given case into a single array, works for both 2d
#and 3d cases; does array arithmetic based on xyzmax and res (point density)
def buildarr(res, datafile, meshfile, channel=0):
    final = np.zeros(res)
    data = open(datafile, "r")
    mesh = open(meshfile, "r")
    points = getVertices(mesh)
    XYZmax = np.array(xyzmax(points))
    mul = list(map(int, np.divide(res, XYZmax)))
    blocks = getBlocks(mesh, points)
    for i in range(23):
        discard = data.readline()
    for i in range(len(blocks)):
        st = blocks[i][0][0]
        st = (st[1]*mul[1], st[0]*mul[0], st[2]*mul[2])
        st = list(map(int, st))
        blocks[i] = buildBlock(blocks[i], data, channel)
        shp = blocks[i].shape
        final[res[0]-(st[0]+shp[0]):res[0]-st[0], st[1]:st[1]+shp[1], \
                st[2]:st[2]+shp[2]] = blocks[i]

    return final

```



The first function, called “buildarr”, is the function used to assemble a matrix from a “blockMeshDict” file and a data file. Essentially, this function opens the mesh file and data file, and then assembles the matrix using the data file. First, an empty matrix of the desired resolution is created, and then the individual block components are assembled using the “buildBlock” function, and then placed into the empty matrix. This matrix is then returned by the function.

The “buildBlock” function takes a block, a data file, and another argument called “channel” to assemble a single block from the data file, exactly as shown in in figure 12. Essentially, given the block information, the function traverses through all three possible dimensions of the block axes, and then inserts the points into this block matrix iteratively. The channel argument is used specifically for assembling the velocity matrix, which has three separate values for each entry (x velocity, y velocity, z velocity). The channel argument can take on the value of 0,1, or 2, and each value corresponds to either the x, y, or z velocity, and all three must be constructed as separate matrices. This capability was not used for this project but may be useful for future endeavors. Another capability of this code that was unused was the scalability to three-dimensional data. This work only focused on the two-dimensional DamBreak simulation, but future work on 3D OpenFOAM simulations will be able to reuse this code.

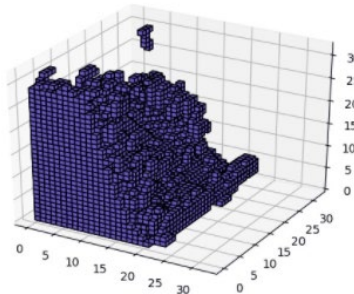


FIGURE 13: SAMPLE OF ASSEMBLED 3D OPENFOAM DATA



## Network Architecture

The network used here is based heavily on the SRGAN architecture, created by Ledig [19], and the full network layout is shown in Figure 14. This is a generative adversarial network designed to solve the general image super-resolution problem.

The generator begins with a convolutional layer with a  $9 \times 9$  kernel size with 64 filters and a PReLU activation, followed by 16 residual blocks with skip connections, which consist of two convolutional layers with 64 filters and a  $3 \times 3$  kernel size each with a batch normalization layer, as well as two up-sampling blocks which contain a convolutional layer with 256 filters and a  $3 \times 3$  kernel size, and an 2D up-sampling layer. Then, the last layer is a convolutional layer with one filter and a kernel size of  $9 \times 9$ , with a sigmoid activation layer. The sigmoid activation function is chosen for the last layer instead of the hyperbolic tangent because the fluid phase fraction values must be between zero and one since it is the ratio of the presence of two fluids at a point in space. So, a sigmoid activation makes more sense here.

The discriminator consists of eight convolutional layers with a kernel size of  $3 \times 3$ , split into pairs of 64 filters, 128 filters, 256 filters, and 512 filters. Each convolutional layer uses the leaky ReLU activation function. The last few layers consist of two dense layers and a sigmoid activation function.



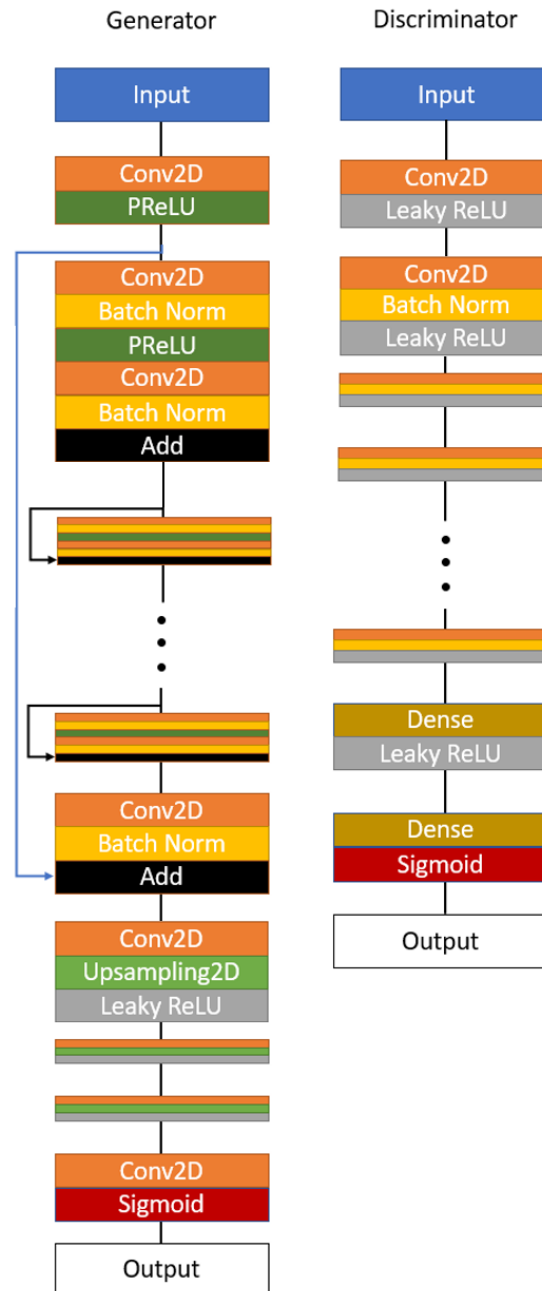



FIGURE 14: NETWORK LAYER DIAGRAM. REPEATED BLOCKS ARE CONDENSED TO SAVE SPACE.



## Network Implementation

The neural network was written in python using the TensorFlow software library. The base code was borrowed from Deepak Birla's Keras implementation of SRGAN [36] and modified to have a sigmoid activation function as well as a custom loss function suited to the task. This code was also adapted slightly to run on  Tensorflow.Keras rather than the original Keras library. The full code is included in appendices C, D, and E.

## Loss Function Design

While the discriminator network was compiled and trained with the original binary cross-entropy loss function, the generator was trained differently from the original SRGAN implementation. The generator was originally constrained with both a pixel-wise loss function in MSE as well as a higher-level content-wise loss function, using a pre-trained VGG network. The work of Ledig et. al. cleverly used VGG loss (perceptual loss) because SRGAN and VGG were designed to operate on the ImageNet dataset. VGG loss cannot be applied to the dataset presented here because it this data is divergent from the training data for VGG (e.g., photographs of animals and plants). However, this dataset follows a set of governing equations, and so additional physics-based constraints to the can be used to train the model.

$$\mathcal{L}_{gen}(Y, \hat{Y}) = \lambda_1 \cdot \frac{1}{mn} \sum_{i=1}^m \sum_{j=1}^n (y_{ij} - \hat{y}_{ij})^2 + \lambda_2 \cdot \frac{1}{mn} \left( \sum_{i=1}^m \sum_{j=1}^n y_{ij} - \sum_{i=1}^m \sum_{j=1}^n \hat{y}_{ij} \right)^2 \quad (1)$$



Physics-based loss functions for SRCFD networks have demonstrated success, as shown by Subramaniam [34] and Jiang [33]. For models that predict fluid velocity and pressure, loss functions are based on the Navier-Stokes equations, and a model that minimizes its loss should stay as consistent with these governing equations as possible, but implementing them can be difficult. However, the phase fraction follows the interphase equations, which are simpler in form. Essentially, our loss function aims to minimize the difference of the volume of liquid phase between the input and the output, in addition to minimizing pixel-wise loss. The interFoam solver models incompressible fluids, so this volume should remain consistent. The full loss function is shown below in Equation 1. The left summand is the MSE with a weighting of  $\lambda_1$  and the right summand is the normalized squared difference of total water volume with a weighting of  $\lambda_2$ . The  $m$  and  $n$  terms represent the height and width of the rectilinear mesh.



## Results and Discussion

Training was performed on a computer with an NVIDIA RTX 3070 GPU on TensorFlow 2.5.0. The network was trained using 5000 data points from the dataset, which was split into 80% for training and 20% for testing. The network was trained with a batch size of 16 for 1000 epochs, and the Adam optimizer was used with a learning rate of  $1 \times 10^{-4}$ . Two SR models were trained – one in which the loss function consisted only of the MSE term, and the other consisting of a combination of the MSE term and the physics-informed term.

Figure 15 shows the training loss for the generator and discriminator of the MSE-only model, and Figure 16 shows the training loss for the generator and discriminator of the MSE + physics model.

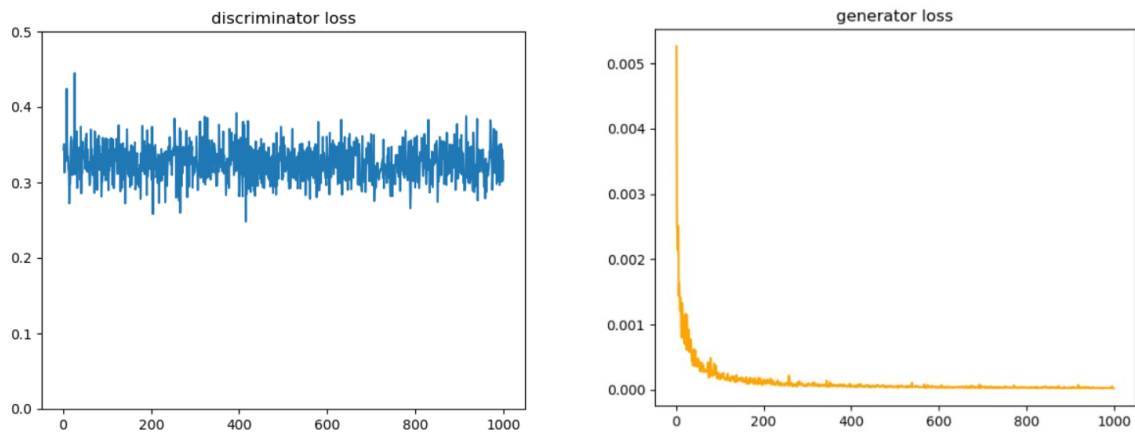


FIGURE 15: TRAINING LOSSES FOR MSE-ONLY MODEL, INCLUDING DISCRIMINATOR LOSS (LEFT) AND GENERATOR LOSS (RIGHT)



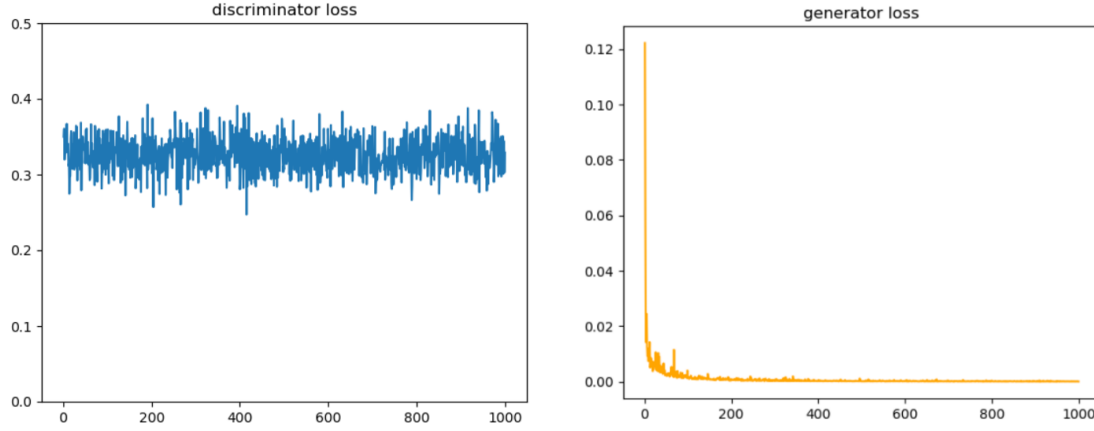


FIGURE 16: TRAINING LOSSES FOR MSE+PHYSICS MODEL, INCLUDING DISCRIMINATOR LOSS (LEFT) AND GENERATOR LOSS (RIGHT)

Figures 15 and 16 demonstrate that the discriminator did not actually converge to an ideal value in this case. Ideally, the discriminator loss should converge to 0.5, which signifies the inability of the discriminator to differentiate between the data created by the generator and data in the ground truth dataset better than chance. However, in this case the discriminator loss was less than 0.5 in both models, indicating some discriminatory accuracy. This should be addressed in future work through careful hyperparameter tuning.

Despite this suboptimal training, the generator was able to converge successfully. The failure of the discriminator to fully converge may be explained by the nature of the problem. The dataset presented here contains quite a diverse set of fluid interactions, but the space of this dataset is still quite limited compared to the ImageNet database, which is what the SRGAN was designed to operate on. The discriminator architecture from SRGAN may have been too powerful for the task, and perhaps a simpler discriminator network may have been more ideal.



Figure 17 provides several comparisons between different upsampling approaches, namely linear upsampling, bicubic upsampling, and the two models presented in this work. These samples were not present in the original training data. The furthest left column in the figure shows the low-resolution input, and the furthest right column shows the ground truth high-resolution simulation. The GAN models were able to capture a high degree of detail, with many high-level features preserved and visible. It seems that the generator was able to learn many of the turbulent and chaotic fluid behaviors present in the dataset. There seems to be very little difference visually between the MSE-only model and the MSE+physics model.

An examination of Figure 17 also indicates that the GAN model outputs have much more clarity and detail than the traditional up-sampling techniques. Turbulent trails of fluid were captured in the upsampled results of these models which are not present in the linear or bicubic results. Some details that are almost imperceptible in the low-resolution input were inferred by the GAN models. For instance, thin, wispy trails of water splashing in the air that are almost completely absent from the input and are only a few pixels in size in the high resolution simulation are captured (see Figures 17AE and 17CE). In addition, small pockets of air resulting from the collision of the water with the bottom boundary were somewhat preserved by the model (see like in Figures 17BE and 17DE). In some cases, like Figures 17BE and 17DE, these features are more pronounced in the MSE+physics model than in the MSE-only model. A more extensive qualitative analysis should be conducted to assess the extent to which this statement holds. These details do differ slightly from the ground truth, but the ability of the model to learn these details is still noteworthy.



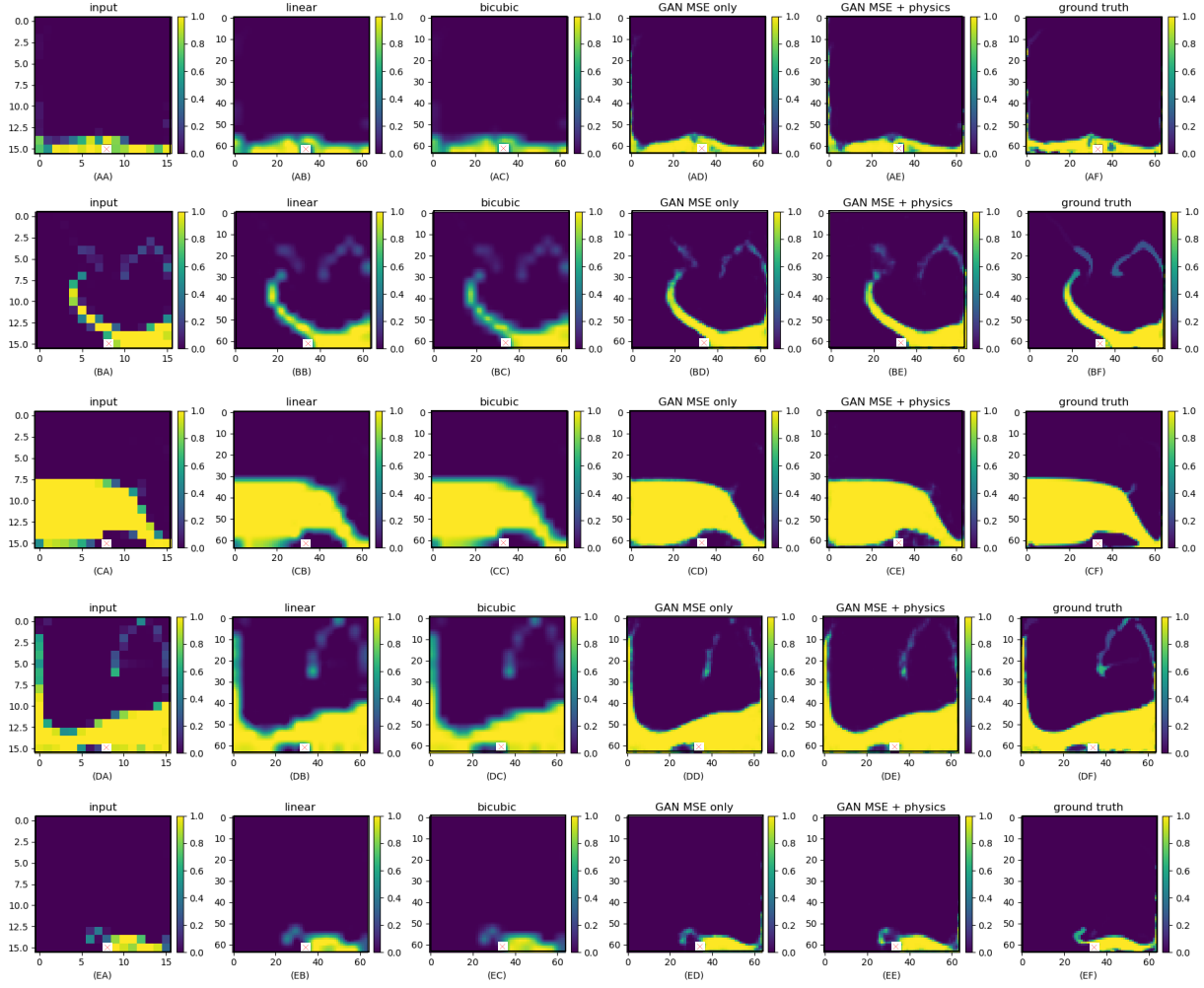


FIGURE 17: EXAMPLES OF MODEL INFERENCE COMPARED TO LINEAR UPSAMPLING, BICUBIC UPSAMPLING, AND GROUND TRUTH HIGH-RESOLUTION SIMULATION.

The fine levels of detail can really be appreciated when looking closer at the snapshots in figure 17.



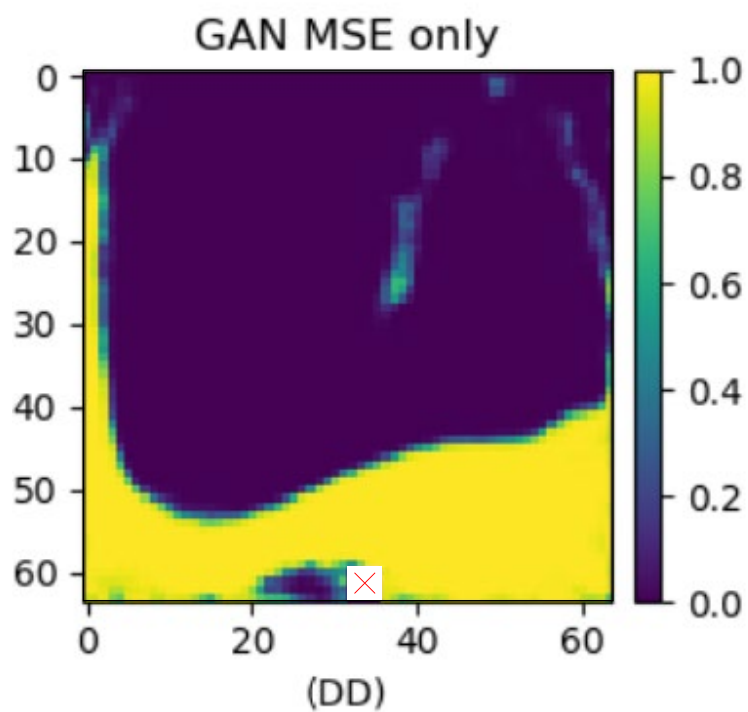


FIGURE 18: FIG17 DD ENHANCED

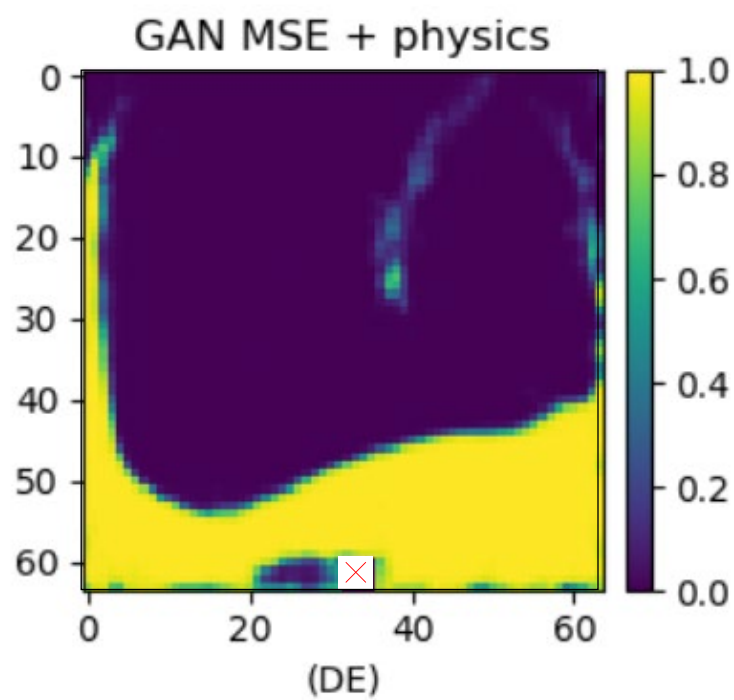


FIGURE 19: FIG 17 DE ENHANCED



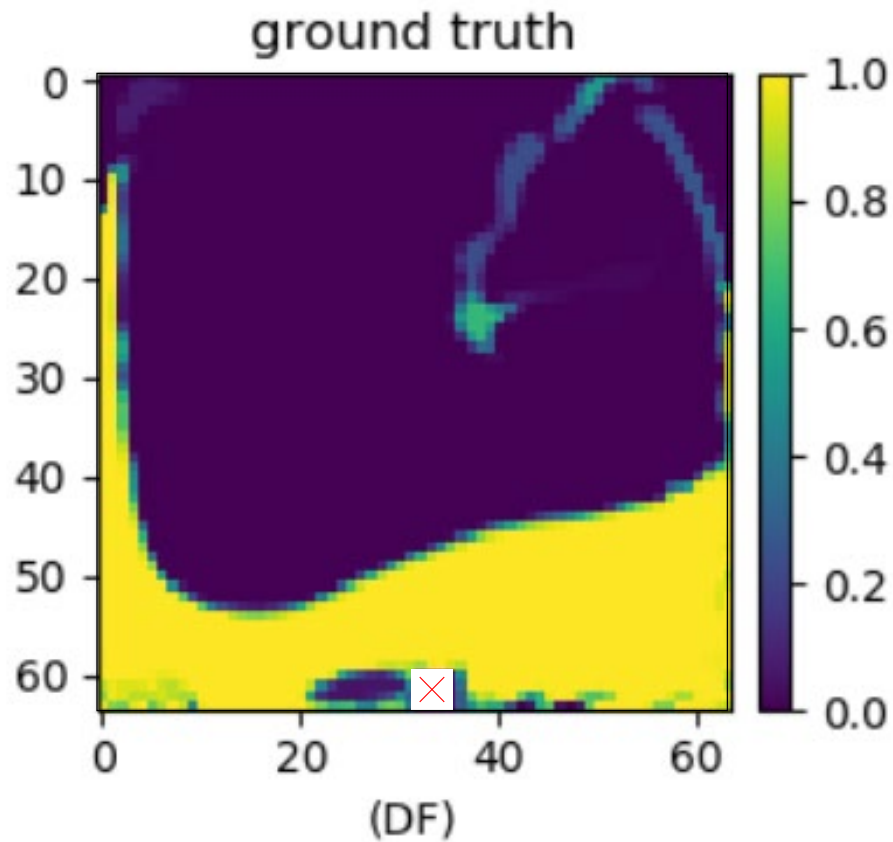


FIGURE 20: FIGURE &amp; DF ENHANCED

Here, the small details can be seen more clearly. The large trail of water created from the water collision with the right wall has been reproduced by the model quite well, as well as the small air pockets underneath the mass of water. Another point to notice is that the MSE-only and MSE + physics models do have slightly different outputs that can be seen more clearly in figures 18 and 19, even if these differences are very small. In figure 19, it seems that the MSE + physics model seems to be closer to the ground truth, especially when looking at the trail on the right as well as the positions of the air pockets, but in other cases the MSE-only model is better. No definitive conclusion about model quality can be made from looking at these figures visually.



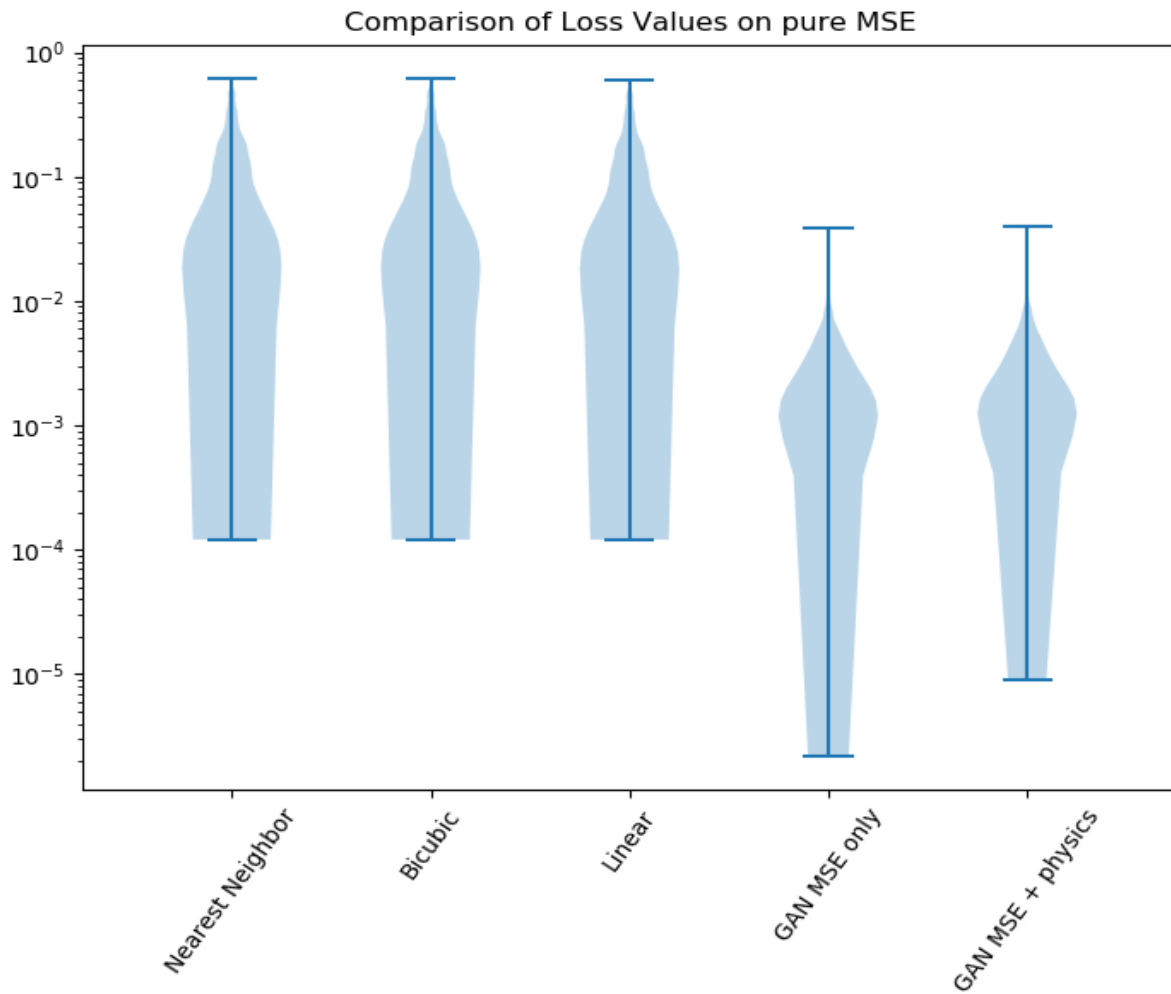


FIGURE 21: COMPARISON OF MAGNITUDE AND DISTRIBUTION OF LOSS VALUES ON A PURE MSE LOSS FUNCTION ON A LOG SCALE FOR THE DIFFERENT UPSAMPLING METHODS. THESE VALUES WERE TAKEN FROM A RANDOM SAMPLE OF 10000 DATA POINTS FROM THE DATA SET.



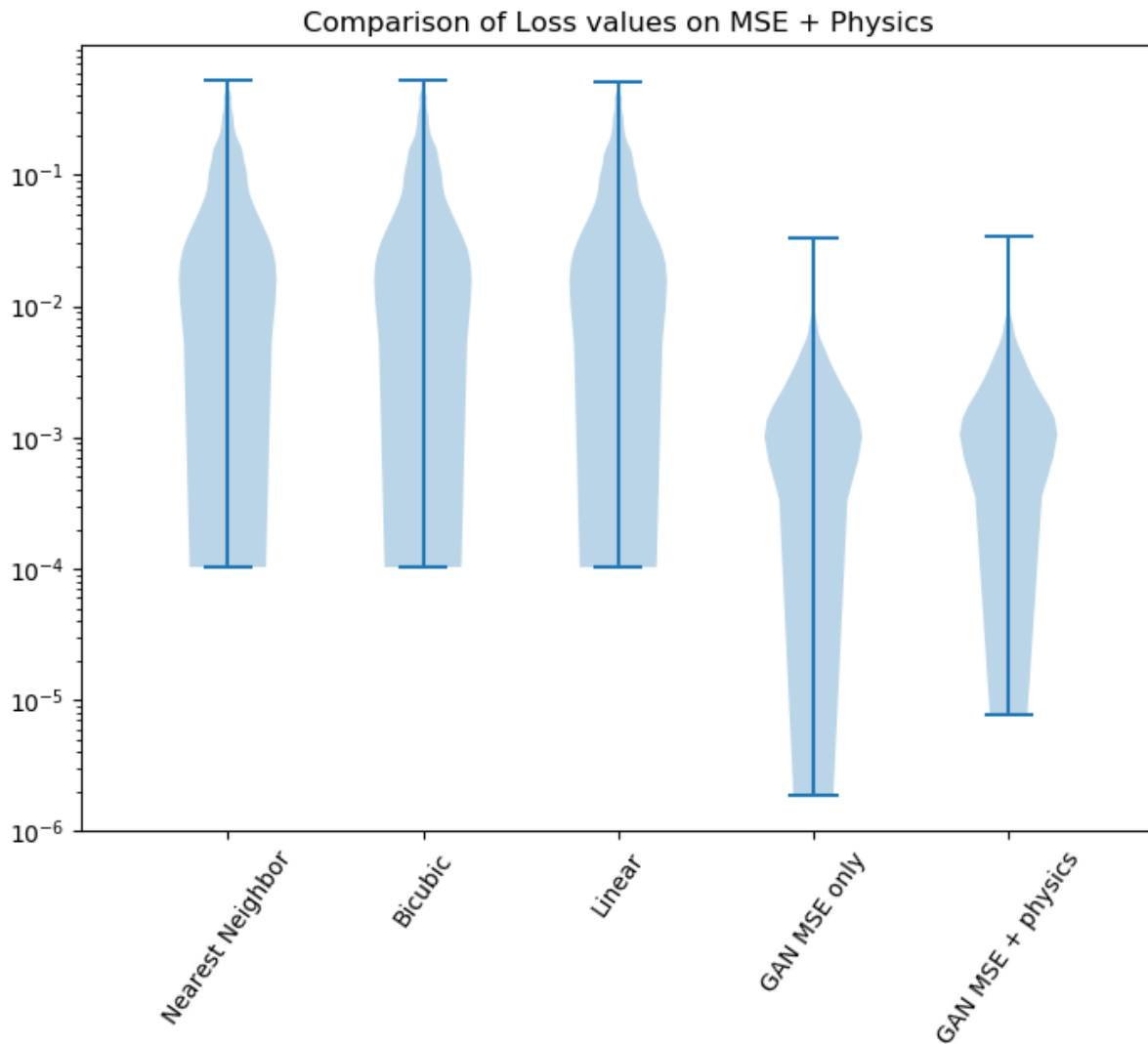


FIGURE 22: COMPARISON OF MAGNITUDE AND DISTRIBUTION OF LOSS VALUES ON THE CUSTOM MSE + PHYSICS LOSS FUNCTION ON A LOG SCALE FOR THE DIFFERENT UPSAMPLING METHODS. THESE VALUES WERE TAKEN FROM A RANDOM SAMPLE OF 10000 DATA POINTS FROM THE DATA SET.



Figure 21 shows the MSE loss values measured across all models with a log axis, and figure 22 shows the same values evaluated using the MSE + physics loss function. This shows that the GAN models achieve loss that is substantially lower than nearest-neighbor, bicubic, and linear upsampling approaches. These figures reveal quantitative evidence to back up several of the observations from Figure 17. For instance, there is little difference between the loss values of the MSE-only and MSE + physics models. The primary difference is that the MSE + physics model seems to have smaller error bounds than the MSE-only model. However, the MSE-only model does achieve some lower loss values. Both of these models do significantly outperform bicubic and linear upsampling. The variance is significant (crossing orders of magnitude for all upsampling approaches), which may be explained by the variability of the dataset, as demonstrated in Figure 9.

As measured by loss values only, the MSE-only model does appear to out-perform the physics-informed model. It could be that MSE alone provides enough information to guide the training process on this dataset, and the scope of the output space is not further reduced by adding the physical constraint. In other words, minimizing the MSE value may already result in fluid volume consistency. More comprehensive models that reconstruct other fluid properties like pressure and velocity have a much larger output space, and therefore may benefit more from physical constraints. A more complex physics-based constraint could yield better results for the current multi-phase problem.

The capabilities of the model presented here should help make the argument for further research into the field of SRCFD, because the computational resources saved could potentially be



immense. Up-sampling low-resolution CFD simulations to an acceptable quality using a neural network instead of directly running high-resolution CFD simulations with traditional numerical methods can save a lot of computation time, potentially leading to the streamlining of many workflows. The results presented here are not perfect, but the success found here definitely warrants further research and experimentation nonetheless.



## Conclusions and Future Work

This work presented a physics-informed neural network model for the super-resolution of multi-phase CFD simulations. The current scope of the field of SRCFD research has focused predominantly on single phase fluid flow simulations. In this work, we demonstrated that those methods work on multiphase fluid flow as well. The model presented here can reconstruct turbulent multiphase flow at a higher resolution with high accuracy, far exceeding bicubic and linear upsampling. We observed that the discriminator may have failed to converge during training, but this may be a consequence of directly applying the SRGAN discriminator architecture to a more limited dataset. This is an issue that should be investigated in future work, and perhaps a weaker discriminator could be experimented with to improve loss values.

Future work should further investigate the application of refined and modified SRCFD models to multiphase turbulent flow. CFD simulations include a time component, and subsequent timesteps are dependent on one another. Recurrent models could possibly provide improved results by taking advantage of the information contains in subsequent simulation frames. This work also focused solely on the fluid phase fraction, but a comprehensive model that reconstructs additional fluid properties like pressure and velocity for multiphase flow would potentially benefit more significantly from physics-based constraints. Finally, this work was conducted using two-dimensional CFD simulations only. Three-dimensional simulations incur even greater computational penalties when resolution is increased, so super-resolution should be investigated for those cases as well.



Future work could also focus on investigating the scalability and adaptability of SRCFD. The scope of this project only included rectangular geometries, and it will be worth investigating if the capabilities of the model presented here can transfer to more complex geometries. For instance, more complex obstacles can be added to the simulations instead of a single dam, or more bodies of water can be added at the start. Future work could also try to replicated the success found in this work with other simulations using other numerical solvers besides interFoam.



## Appendix A

### Code used to assemble OpenFOAM data into matrix format

Full source code for this project can be found at:

<https://github.com/matthewli125/SRCFD/releases/tag/v1.0>

```
import numpy as np
import re

#gets the vertices from the blockMeshDict file and makes a list of tuples, the
#list index of each vertex will correspond to its vertex number
def getVertices(File):
    points = []
    line = ""
    while line != "vertices\n":
        line = File.readline()
    line = File.readline()
    while line != ");\n":
        line = File.readline()
        if len(line)>4: points.append(eval(line[4:].replace(" ", ",")))
    return points

#gets the blocks from the blockMeshDict file, creates a list of blocks, which
#are sets of numbers corresponding to vertices
def getBlocks(File, points):
    # blocks = np.array([])
    blocks = []
    line = ""
    while line != "blocks\n":
        line = File.readline()
    line = File.readline()
    while line != ");\n":
        line = File.readline()
        filtered = re.split('\(|\|', line[4:])
        if len(filtered)<4: break
        Vertices = eval "[" + filtered[1].replace(" ", ",") + "]"
        Vertices = list(map(lambda x: points[x], Vertices))
        Density = eval "[" + filtered[3].replace(" ", ",") + "]"
        # blocks = np.append(blocks, (Vertices, Density))
        blocks.append((Vertices, Density))
```



```

return blocks

#gets the largest xyz values from all vertices
def xyzmax(pointList):
    x = max(pointList, key = lambda x: x[0])[0]
    y = max(pointList, key = lambda x: x[1])[1]
    z = max(pointList, key = lambda x: x[2])[2]
    return [x,y,z]

#builds a block from data points in a file
def buildBlock(block, File, channel):
    finalblock = np.empty(block[1])
    finalblock = finalblock.reshape(block[1][1], block[1][0], block[1][2])
    for k in range(block[1][2]):
        for i in reversed(range(block[1][1])):
            for j in range(block[1][0]):
                data = File.readline()
                if data[0] == "(":
                    points = [float(i) for i in data[1:-2].split(" ")]
                    point = points[channel]
                else:
                    point = float(data)
                # finalblock[i][j][k] = point if point > 0.00001 else 0
                finalblock[i][j][k] = point
    return finalblock

#builds all the blocks in a given case into a single array, works for both 2d
#and 3d cases; does array arithmetic based on xyzmax and res (point density)
def buildarr(res, datafile, meshfile, channel=0):
    final = np.zeros(res)
    data = open(datafile, "r")
    mesh = open(meshfile, "r")
    points = getVertices(mesh)
    XYZmax = np.array(xyzmax(points))
    mul = list(map(int, np.divide(res, XYZmax)))
    blocks = getBlocks(mesh, points)
    for i in range(23):
        discard = data.readline()
    for i in range(len(blocks)):
        st = blocks[i][0][0]
        st = (st[1]*mul[1], st[0]*mul[0], st[2]*mul[2])

```



```

    st = list(map(int, st))
    blocks[i] = buildBlock(blocks[i], data, channel)
    shp = blocks[i].shape
    final[res[0]-(st[0]+shp[0]):res[0]-st[0], st[1]:st[1]+shp[1], \
            st[2]:st[2]+shp[2]] = blocks[i]

    return final

#performs buildarr on all cases in a directory, but sorts each level based on
#filename so they are saved and named in the correct time and case order. Uses
#global path vars. Set high to true if doing highres to follow file naming
#convention.
def buildall(res, high, savepth):
    FILES = ["alpha.water"]
    filFunc = lambda x: "highres" in x if high else "highres" not in x
    for i in sorted(filter(filFunc, PTHd), key= lambda x: int(x.split("_")[0])):
        # for i in [h + "_highres" for h in brokencases]:
            print("doing case {} {}".format(i, "highres" if high else "lowres"))
            for j in tqdm(list(filter(lambda x:x[0].isdigit(),listdir(PTH+"/"+i)))):
                for k in filter(lambda x: x in FILES, listdir(PTH+"/"+i+"/"+j)):
                    try:
                        arr = buildarr(res, PTH+"/"+i+"/"+j+"/"+k, PTH+"/"+i+blockMes
hDictPath)

                        np.save(savepth+"{}-{}x{}x{}-{}-{}.npy".format \
                                (i, *res, k, j), arr)

                    except:
                        print("fail")

@cached()
def getBroken(cases):
    brokenCasesLR = {str(i[0]) for i in [(y,len([x for x in listdir(LRDATAPTH)
        if x.startswith("{}-
".format(y))])) for y in tqdm(range(num))] if i[1] < cases}

    brokenCasesHR = {str(i[0]) for i in [(y,len([x for x in listdir(HRDATAPTH)
        if x.startswith("{}_highres-
".format(y))])) for y in tqdm(range(num))] if i[1] < cases}

    print(list(brokenCasesLR.union(brokenCasesHR)))
    return list(brokenCasesLR.union(brokenCasesHR))

```



```

def buildpartial(res, high, savepth, cases):
    filFunc = lambda x: "highres" in x if high else "highres" not in x
    for i in cases:
        print("doing case {} {}\n".format(i, "highres" if high else "lowres"))
        for j in tqdm(list(filter(lambda x: x[0].isdigit(), listdir(PTH+"/"+i)))):
            for k in filter(lambda x: x in FILES, listdir(PTH+"/"+i+"/"+j)):
                try:
                    arr = buildarr(res, PTH+"/"+i+"/"+j+"/"+k, PTH+"/"+i+blockMes
hDictPath)
                    np.save(savepth+"{}_highres-{}x{}x{}-{}-{}.npy".format \
                        (i, *res, k, j), arr)
                except:
                    print("fail")

```



## Appendix B

### Code used to distribute computation of OpenFOAM simulations to multiple CPUs

```
import subprocess
from os import listdir
from tqdm import tqdm
import numpy as np
import multiprocessing as mp
from checkComplete import getBroken
from paths import PTH

num = 800 #number of cases to run

# calls the openFoam executable Allrun sequentially for a given list of
# directories
def Run(cases):
    failedCases = []
    for i in tqdm(cases):
        # result1 = subprocess.call([PTH + "/" + i + "/Allclean", ">", "/dev/nul
1"])
        result2 = subprocess.call([PTH + "/" + i + "/Allrun", ">", "/dev/null"])
        if result2 != 0:
            failedCases.append(i)

# finds all the folders that have incomplete or empty timesteps and puts them
# all in a list for easier handling
def GetIncomplete(correct_count):
    sum = 0
    incomplete = []
    print("searching for incomplete cases")
    for i in tqdm(list(listdir(PTH))):
        for j in listdir(PTH+"/"+i):
            sum+=1
        if sum < correct_count:
            incomplete.append(i)
        sum = 0
    return incomplete

def Distribute(cases):
    nextLargest = len(cases)
    while nextLargest % numCores > 0:
        nextLargest-=1
```



```
distributedCases = list(np.split(np.array(cases)[:nextLargest], numCores))
distributedCases = [list(i) for i in distributedCases]

for i in range(len(cases[nextLargest:])):
    distributedCases[i % numCores].append(cases[nextLargest:][i])

return distributedCases

if __name__ == "__main__":
    numCores = mp.cpu_count()-2
    print(str(numCores) + "cores available for use\n")
    incomplete = GetIncomplete(65) #correct number of items per folder
    print(incomplete)
    print(str(len(incomplete)) + " cases to be handled\n")
    pool = mp.Pool(numCores)
    pool.map(Run, Distribute(incomplete))
```



## Appendix C

### Code used to define the GAN model

```
#title      :Network.py
#description :Architecture file(Generator and Discriminator)
#author     :Deepak Birla (modified by Matthew Li)
#date      :2018/10/30
#usage     :from Network import Generator, Discriminator
#python_version :3.5.4

# Modules
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Activation
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.layers import UpSampling2D
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Input
from tensorflow.keras.layers import Conv2D, Conv2DTranspose
from tensorflow.keras.models import Model
from tensorflow.keras.layers import LeakyReLU, PReLU
from tensorflow.keras.layers import Add

# Residual block
def res_block_gen(model, kernel_size, filters, strides):

    gen = model

    model = Conv2D(filters = filters, kernel_size = kernel_size, strides = stride
s, padding = "same")(model)
    model = BatchNormalization(momentum = 0.5)(model)
    # Using Parametric ReLU
    model = PReLU(alpha_initializer='zeros', alpha_regularizer=None, alpha_constraint=None, shared_axes=[1,2])(model)
    model = Conv2D(filters = filters, kernel_size = kernel_size, strides = stride
s, padding = "same")(model)
    model = BatchNormalization(momentum = 0.5)(model)

    model = Add()([gen, model])

    return model
```



```

def up_sampling_block(model, kernal_size, filters, strides):

    # In place of Conv2D and UpSampling2D we can also use Conv2DTranspose (Both are used for Deconvolution)
    # Even we can have our own function for deconvolution (i.e one made in Utils.py)
    #model = Conv2DTranspose(filters = filters, kernel_size = kernal_size, strides = strides, padding = "same")(model)
    model = Conv2D(filters = filters, kernel_size = kernal_size, strides = strides, padding = "same")(model)
    model = UpSampling2D(size = 2)(model)
    model = LeakyReLU(alpha = 0.2)(model)

    return model


def discriminator_block(model, filters, kernel_size, strides):

    model = Conv2D(filters = filters, kernel_size = kernel_size, strides = strides, padding = "same")(model)
    model = BatchNormalization(momentum = 0.5)(model)
    model = LeakyReLU(alpha = 0.2)(model)

    return model


# Network Architecture is same as given in Paper https://arxiv.org/pdf/1609.04802.pdf
class Generator(object):

    def __init__(self, noise_shape):

        self.noise_shape = noise_shape

    def generator(self):

        gen_input = Input(shape = self.noise_shape)

        model = Conv2D(filters = 64, kernel_size = 9, strides = 1, padding = "same")(gen_input)
        model = PReLU(alpha_initializer='zeros', alpha_regularizer=None, alpha_constraint=None, shared_axes=[1,2])(model)

        gen_model = model

```



```

# Using 16 Residual Blocks
for index in range(16):
    model = res_block_gen(model, 3, 64, 1)

    model = Conv2D(filters = 64, kernel_size = 3, strides = 1, padding = "same")(model)
    model = BatchNormalization(momentum = 0.5)(model)
    model = Add()([gen_model, model])

# Using 2 UpSampling Blocks
for index in range(2):
    model = up_sampling_block(model, 3, 256, 1)

    model = Conv2D(filters = 1, kernel_size = 9, strides = 1, padding = "same")(model)
    model = Activation('sigmoid')(model)

    generator_model = Model(inputs = gen_input, outputs = model)

    return generator_model

# Network Architecture is same as given in Paper https://arxiv.org/pdf/1609.04802.pdf
class Discriminator(object):

    def __init__(self, image_shape):

        self.image_shape = image_shape

    def discriminator(self):

        dis_input = Input(shape = self.image_shape)

        model = Conv2D(filters = 64, kernel_size = 3, strides = 1, padding = "same")(dis_input)
        model = LeakyReLU(alpha = 0.2)(model)

        model = discriminator_block(model, 64, 3, 2)
        model = discriminator_block(model, 128, 3, 1)
        model = discriminator_block(model, 128, 3, 2)
        model = discriminator_block(model, 256, 3, 1)
        model = discriminator_block(model, 256, 3, 2)
        model = discriminator_block(model, 512, 3, 1)
        model = discriminator_block(model, 512, 3, 2)

```



```
model = Flatten()(model)
model = Dense(1024)(model)
model = LeakyReLU(alpha = 0.2)(model)

model = Dense(1)(model)
model = Activation('sigmoid')(model)

discriminator_model = Model(inputs = dis_input, outputs = model)

return discriminator_model
```



## Appendix D

### Code used to train GAN model

```

from gan import Generator, Discriminator
from foam_case_class import Foam_Case
import random
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input
from tensorflow.keras.optimizers import Adam, SGD, RMSprop
from tqdm import tqdm
from os import listdir
import numpy as np

from loss_functions import master_loss

import tensorflow as tf

gpus = tf.config.experimental.list_physical_devices('GPU')
if gpus:
    try:
        # Currently, memory growth needs to be the same across GPUs
        for gpu in gpus:
            tf.config.experimental.set_memory_growth(gpu, True)
        logical_gpus = tf.config.experimental.list_logical_devices('GPU')
        print(len(gpus), "Physical GPUs,", len(logical_gpus), "Logical GPUs")
    except RuntimeError as e:
        # Memory growth must be set before GPUs have been initialized
        print(e)

data_file = "gan_data.h5"
np.random.seed(10)
epochs = 1
model_save_dir = "gan models"

# Remember to change image shape if you are having different size of images
image_shape = (16,16,1)
image_shape_scaled = (64,64,1)

lr_res = (16,16,1)
hr_res = (64,64,1)

adam = Adam(lr=1E-4, beta_1=0.9, beta_2=0.999, epsilon=1e-08)

```



```

def get_data(ratio):
    lr_data_path = "E:/gan_data/ground_only/phase_only/lowres/"
    hr_data_path = "E:/gan_data/ground_only/phase_only/highres/"

    print("loading data into memory")

    lr_data = np.array([np.load(lr_data_path + i) for i in tqdm(listdir(lr_data_p
ath)[:5000])])
    hr_data = np.array([np.load(hr_data_path + i) for i in tqdm(listdir(hr_data_p
ath)[:5000])])

    print(hr_data[1].shape)

    index = int(len(lr_data) * ratio)

    return lr_data[:index], hr_data[:index], lr_data[index:], hr_data[index:]

def get_gan_network(discriminator, generator, optimizer):
    discriminator.trainable = False
    gan_input = Input(shape = image_shape)
    x = generator(gan_input)
    gan_output = discriminator(x)
    gan = Model(inputs=gan_input, outputs=[x, gan_output])
    gan.compile(loss=[master_loss, "binary_crossentropy"],
                loss_weights=[1., 1e-10],
                optimizer=optimizer)

    return gan

def train(epochs, batch_size, data_split_ratio):
    x_train_lr, x_train_hr, x_test_lr, x_test_hr = get_data(data_split_ratio)
    batch_count = int(x_train_hr.shape[0] / batch_size)

    generator = Generator(image_shape).generator()
    generator._name = "generator"
    discriminator = Discriminator(image_shape_scaled).discriminator()
    generator.compile(loss=master_loss, optimizer=adam)

    discriminator.compile(loss="binary_crossentropy", optimizer=adam)

    print(generator.summary())
    print(discriminator.summary())

```



```

gan = get_gan_network(discriminator, generator, adam)

for e in range(1, epochs+1):
    print ('-'*15, 'Epoch %d' % e, '-'*15)
    for batch in tqdm(range(batch_count)):

        rand_nums = np.random.randint(0, x_train_hr.shape[0], size=batch_size
)

        image_batch_hr = x_train_hr[rand_nums]
        image_batch_lr = x_train_lr[rand_nums]
        generated_images_sr = generator.predict(image_batch_lr)

        real_data_Y = np.ones(batch_size) - np.random.random_sample(batch_size
e)*0.2
        fake_data_Y = np.random.random_sample(batch_size)*0.2

        discriminator.trainable = True

        d_loss_real = discriminator.train_on_batch(image_batch_hr, real_data_
Y)
        d_loss_fake = discriminator.train_on_batch(generated_images_sr, fake_
data_Y)
        discriminator_loss = 0.5 * np.add(d_loss_fake, d_loss_real)

        rand_nums = np.random.randint(0, x_train_hr.shape[0], size=batch_size
)

        image_batch_hr = x_train_hr[rand_nums]
        image_batch_lr = x_train_lr[rand_nums]

        gan_Y = np.ones(batch_size) - np.random.random_sample(batch_size)*0.2
        discriminator.trainable = False
        gan_loss = gan.train_on_batch(image_batch_lr, [image_batch_hr, gan_Y])

        print("discriminator_loss : %f" % discriminator_loss)
        print("gan_loss :", gan_loss)
        gan_loss = str(gan_loss)

        loss_file = open(model_save_dir + 'losses.txt' , 'a')
        loss_file.write('epoch%d : gan_loss = %s ; discriminator_loss = %f\n' %(e
, gan_loss, discriminator_loss) )
        loss_file.close()

```



```

generator.save_weights("generator_weights_physics")
discriminator.save_weights("discriminator_weights_physics")

def get_data_paths(path):

    print("FETCHING DATA PATHS...")

    def sortbynum(nums):
        return sorted(nums, key = lambda x: float(x.split("_")[0]))

    #this sorts all the directories in the path; each of these directories
    #is an openfoam case that has subdirectories for timesteps
    lowres = sortbynum([i for i in listdir(path) if "highres" not in i])
    highres = sortbynum([i for i in listdir(path) if "highres" in i])
    data = list(zip(lowres, highres))

    def expand(paths):
        all_sub_paths = []
        isnum = lambda x: x[0].isdigit() and float(x) != 0
        for i in tqdm(paths):
            times = sortbynum(list(filter(isnum, listdir("".join([path,"/",i])))))
        )
            sub_paths = ["".join([path,"/",i,"/",timestep]) for timestep in times
        ]

            all_sub_paths+=sub_paths

        return all_sub_paths

    lowres_expanded = expand(lowres)
    highres_expanded = expand(highres)

    print("DATA PATHS FETCHED")

    return list(zip(lowres_expanded, highres_expanded))

#this helper function loads the data from a given list of file directories. This
#allows the data to be loaded and unloaded on the fly, making the operation more
#memory efficient.
def load_data_batch_unbuilt(batch, res, type):
    return np.array([Foam_Case(res, file_path, type).fetch().enum() for file_path
in tqdm(batch)])

```



```

# def load_data_batch(num, res, type):
#     if type == "lowres":
#         return np.load("E:/gan_data/lowres/%d.npy" % num[0])
#     else:
#         return np.load("E:/gan_data/highres/%d.npy" % num[0])

def load_data_batch(nums, res, type):
    if type == "lowres":
        return np.array([np.load("E:/gan_data/lowres_downsample/%d.npy" % i) for i in nums])
    else:
        return np.array([np.load("E:/gan_data/highres_single/%d.npy" % i) for i in nums])
#this helper function takes a loaded list of data files and deletes them, freeing
#memory.
def unload_data_batch(batch):
    [foam_case.crunch() for foam_case in batch]

def train_mem_efficient(epochs, batch_size, data_split_ratio):
    path = "E:/dambreak_cases4"
    lr_res = (16,16,1)
    hr_res = (64,64,1)

    x_train_hr = x_train_lr = np.array(range(48000))

    batch_count = int(x_train_hr.shape[0] / batch_size)

    generator = Generator(image_shape).generator()
    discriminator = Discriminator(image_shape_scaled).discriminator()
    generator.compile(loss=master_loss, optimizer=adam)
    discriminator.compile(loss="binary_crossentropy", optimizer=adam)

    gan = get_gan_network(discriminator, generator, adam)

    for e in range(1,epochs+1):
        print ('-'*15, 'Epoch %d' % e, '-'*15)
        for batch in tqdm(range(batch_count)):

            # rand_nums = np.random.randint(0, x_train_hr.shape[0], size=1) #keep
            size as 1 for now; each file has 60 data points
            rand_nums = np.random.randint(0, x_train_hr.shape[0], size=batch_size
)

```



```

        image_batch_hr = load_data_batch(x_train_hr[rand_nums], hr_res, "high
res")
        image_batch_lr = load_data_batch(x_train_lr[rand_nums], lr_res, "lowr
es")
        generated_images_sr = generator.predict_on_batch(image_batch_lr)

        real_data_Y = np.ones(batch_size) - np.random.random_sample(batch_siz
e)*0.2
        fake_data_Y = np.random.random_sample(batch_size)*0.2

        discriminator.trainable = True

        d_loss_real = discriminator.train_on_batch(image_batch_hr, real_data_
Y)
        d_loss_fake = discriminator.train_on_batch(generated_images_sr, fake_
data_Y)
        discriminator_loss = 0.5 * np.add(d_loss_fake, d_loss_real)

        rand_nums = np.random.randint(0, x_train_hr.shape[0], size=batch_size
)
        image_batch_hr = load_data_batch(x_train_hr[rand_nums], hr_res, "high
res")
        image_batch_lr = load_data_batch(x_train_lr[rand_nums], lr_res, "lowr
es")

        gan_Y = np.ones(batch_size) - np.random.random_sample(batch_size)*0.2
        discriminator.trainable = False
        gan_loss = gan.train_on_batch(image_batch_lr, [image_batch_hr, gan_Y],
return_dict=True)

        print("discriminator_loss : %f" % discriminator_loss)
        print("gan_loss :", gan_loss)
        gan_loss = str(gan_loss)

        loss_file = open(model_save_dir + 'losses.txt' , 'a')
        loss_file.write('epoch%d : gan_loss = %s ; discriminator_loss = %f\n' %(e
, gan_loss, discriminator_loss) )
        loss_file.close()

        generator.save_weights("E:/gan_data/generator_weights_phase_only")
        discriminator.save_weights("E:/gan_data/discriminator_weights_phase_only")

```



## Appendix E

### Foam Case Class code; helper class for training GAN model

```

import numpy as np
import os
from foamToPy import buildarr

class Foam_Case:
    def __init__(self, res, file_path, type):
        self.type = type #highres or lowres
        self.res = res
        self.mesh_file = os.path.dirname(file_path) + "/system/blockMeshDict"
        self.alpha_path = file_path + "/alpha.water"
        self.U_path = file_path + "/U"
        self.p_path = file_path + "/p"

    def fetch(self):
        self.alpha = np.squeeze(buildarr(self.res, self.alpha_path, self.mesh_file, channel = 0))
        self.p = np.squeeze(buildarr(self.res, self.p_path, self.mesh_file, channel = 0))
        self.Ux = np.squeeze(buildarr(self.res, self.U_path, self.mesh_file, channel = 1))
        self.Uy = np.squeeze(buildarr(self.res, self.U_path, self.mesh_file, channel = 1))
        self.Uz = np.squeeze(buildarr(self.res, self.U_path, self.mesh_file, channel = 2))

        return self

    def crunch(self):
        del self.alpha
        del self.p
        del self.Ux
        del self.Uy
        del self.Uz

    def enum(self):
        return np.stack([self.alpha, self.p, self.Ux, self.Uy, self.Uz], axis=-1)

```







```

cases = [
    "40552.npy",
    "22078.npy",
    "23555.npy",
    "22139.npy",
    "46791.npy"
]

labs = ["A", "B", "C", "D", "E"]
i = 0

for case in cases:

    input = np.array([np.load("E:/gan_data/ground_only/phase_only/lowres/%s"
% case)])
    outputMSEphysics = gen1.predict(input)
    outputMSEonly = gen2.predict(input)
    real = np.load("E:/gan_data/ground_only/phase_only/highres/%s" % case)

    cubic = cv2.resize(np.squeeze(input), (64,64), interpolation = cv2.INTER_
CUBIC)
    linear = cv2.resize(np.squeeze(input), (64,64), interpolation = cv2.INTER
_LINEAR)
    nearest = cv2.resize(np.squeeze(input), (64,64), interpolation = cv2.INTE
R_NEAREST)

    fig,ax = plt.subplots(nrows=1, ncols=6, figsize=(18,3))

    im1 = ax[0].imshow(np.squeeze(input), vmin=0, vmax=1)
    im2 = ax[1].imshow(cubic, vmin=0, vmax=1)
    im3 = ax[2].imshow(linear, vmin=0, vmax=1)
    im4 = ax[3].imshow(np.squeeze(outputMSEonly), vmin=0, vmax=1)
    im5 = ax[4].imshow(np.squeeze(outputMSEphysics), vmin=0, vmax=1)
    im6 = ax[5].imshow(np.squeeze(real), vmin=0, vmax=1)

    ax[0].set_title("input")
    ax[1].set_title("linear")
    ax[2].set_title("bicubic")
    ax[3].set_title("GAN MSE only")
    ax[4].set_title("GAN MSE + physics")
    ax[5].set_title("ground truth")

    ax[0].set_xlabel("(%sA)" % labs[i])
    ax[1].set_xlabel("(%sB)" % labs[i])

```



```

ax[2].set_xlabel("(%sC)" % labs[i])
ax[3].set_xlabel("(%sD)" % labs[i])
ax[4].set_xlabel("(%sE)" % labs[i])
ax[5].set_xlabel("(%sF)" % labs[i])

i+=1

fig.colorbar(im1, fraction=0.046, ax=ax[0], pad=0.04)
fig.colorbar(im2, fraction=0.046, ax=ax[1], pad=0.04)
fig.colorbar(im3, fraction=0.046, ax=ax[2], pad=0.04)
fig.colorbar(im4, fraction=0.046, ax=ax[3], pad=0.04)
fig.colorbar(im5, fraction=0.046, ax=ax[4], pad=0.04)
fig.colorbar(im6, fraction=0.046, ax=ax[5], pad=0.04)

fig.tight_layout()

plt.show()

def make_violinplot(gen1, gen2):
    cases = np.random.choice(listdir("E:/gan_data/ground_only/phase_only/lowres/")
                             )[5000:], 10000, replace=False)

    losses = {"nearest": [], "bicubic": [], "linear": [], "model MSE only": [], "model MSE + physics": []}
    losses_physics = {"nearest": [], "bicubic": [], "linear": [], "model MSE only": [], "model MSE + physics": []}

    for case in tqdm(cases):

        input = np.array([np.load("E:/gan_data/ground_only/phase_only/lowres/%s" % case)])

        outputMSEphysics = gen1.predict(input).astype("float64")
        outputMSEonly = gen2.predict(input).astype("float64")
        real = tf.convert_to_tensor(np.load("E:/gan_data/ground_only/phase_only/highres/%s" % case))

        cubic = cv2.resize(np.squeeze(input), (64,64), interpolation = cv2.INTER_CUBIC)
        linear = cv2.resize(np.squeeze(input), (64,64), interpolation = cv2.INTER_LINEAR)
        nearest = cv2.resize(np.squeeze(input), (64,64), interpolation = cv2.INTER_NEAREST)

```



```

losses["nearest"].append(MSE(nearest, real).numpy())
losses["bicubic"].append(MSE(cubic, real).numpy())
losses["linear"].append(MSE(linear, real).numpy())
losses["model MSE only"].append(MSE(outputMSEonly, real).numpy())
losses["model MSE + physics"].append(MSE(outputMSEphysics, real).numpy())

losses_physics["nearest"].append(master_loss(nearest, real).numpy())
losses_physics["bicubic"].append(master_loss(cubic, real).numpy())
losses_physics["linear"].append(master_loss(linear, real).numpy())
losses_physics["model MSE only"].append(master_loss(outputMSEonly, real).numpy())
losses_physics["model MSE + physics"].append(master_loss(outputMSEphysics, real).numpy())

fig, ax = plt.subplots(nrows = 2, ncols = 1, figsize =(6,12))

v1 = ax[0].violinplot([losses[i] for i in losses])
ax[0].set_yscale('log')
ax[0].set_title("Comparison of Loss Values on pure MSE")

v2 = ax[1].violinplot([losses_physics[i] for i in losses_physics])
ax[1].set_yscale('log')
ax[1].set_title("Comparison of Loss values on MSE + Physics")

labels = ['Nearest Neighbor', 'Bicubic', 'Linear', 'GAN MSE only', 'GAN MSE + physics']

def set_axis_style(ax, labels):
    ax.get_xaxis().set_tick_params(direction='out')
    ax.xaxis.set_ticks_position('bottom')
    ax.set_xticks(np.arange(1, len(labels) + 1))
    ax.set_xticklabels(labels, rotation = 55)
    ax.set_xlim(0.25, len(labels) + 0.75)

for i in [ax[0],ax[1]]:
    set_axis_style(i, labels)

plt.tight_layout()
plt.show()

```



```
if __name__ == "__main__":

    from os import listdir
    from tqdm import tqdm

    dir = "E:/gan_data/ground_only/phase_only/"
    save_dir = "E:/gan_data/outputs/"

    generatorMSE = Generator(image_shape).generator()
    generatorMSE.compile(loss="MSE", optimizer=adam)
    generatorMSE.load_weights("MSE_only_weights/generator_weights")

    generatorMSEphysics = Generator(image_shape).generator()
    generatorMSEphysics.compile(loss="MSE", optimizer=adam)
    generatorMSEphysics.load_weights("MSE+physics_weights/generator_weights_physics")

    plot_comparison(generatorMSEphysics, generatorMSE)
    make_violinplot(generatorMSEphysics, generatorMSE)
```



## Appendix G

### Code used to implement loss function

```
import numpy as np
import tensorflow.keras.backend as K
import tensorflow as tf

# this loss function calculates the difference between the sums of the phase
# fractions of two single dambreak frames. Because the interFoam solver solves
# for incompressible fluids, meaning the amount of fluid should remain constant
# with time.
def phase_fraction_loss(y_true, y_pred):
    alpha_true_sum = tf.math.reduce_sum(y_true)
    alpha_pred_sum = tf.math.reduce_sum(y_pred)

    size = 64 * 64

    return (alpha_true_sum/size - alpha_pred_sum/size)**2
```



## Appendix H

### Optimized matrix functions and benchmark Code

```
import numpy as np
from numba import jit, prange, njit
from timer import timeit_wrapper
from functools import lru_cache

##SEQUENTIAL

def matrixMax(arr): # finds max value of a 2d matrix
    return np.max([np.max(i) for i in arr])

def matrixMin(arr): # finds min value of a 2d matrix
    return np.min([np.min(i) for i in arr])

def matrixAvg(arr): # finds average value of a 2d matrix
    return np.mean([np.mean(i) for i in arr])

def matrixNormalize(arr, min, max): # normalizes values of a 2d matrix to 0 and 1
    return [(i - min)/(max - min) for i in arr]

##PARALLEL

def pmatrixMax(arr):
    prowMaxs(arr)
    return np.max(arr)

@njit(parallel=True)
def prowMaxs(arr):
    for i in prange(len(arr)):
        arr[i] = np.max(arr[i])

def pmatrixMin(arr):
    prowMins(arr)
    return np.min(arr)

@njit(parallel=True)
def prowMins(arr):
    for i in prange(len(arr)):
        arr[i] = np.min(arr[i])
```



```

def pmatrixAvg(arr):
    prowMeans(arr)
    return np.mean(arr)

@njit(parallel=True)
def prowMeans(arr):
    for i in prange(len(arr)):
        arr[i] = np.mean(arr[i])

@njit(parallel=True)
def pmatrixNormalize(arr, min, max):
    for i in prange(len(arr)):
        for j in prange(len(arr)):
            arr[i,j] = (arr[i][j] - min)/(max - min)

from time import perf_counter
from timeit import timeit, Timer
from decimal import Decimal
from foamToPy import buildarr
from matrix import *
from tqdm import tqdm
from functools import partial

def timeFunc(func, *args, **kwargs):
    # start = Decimal(perf_counter())
    t = Timer(partial(func, *args, **kwargs))
    retVal = t.timeit(number=100)
    # end = Decimal(perf_counter())
    # return end-start
    return retVal

if __name__ == "__main__":
    a = np.squeeze(buildarr((512,512,1), "D:/openfoamData/dambreak_cases5/1_highr
es/2./alpha.water", "D:/openfoamData/dambreak_cases5/0_highres/system/blockMeshDic
t"))

    seqTimes = {"min":0, "max":0, "avg":0, "normalize":0}
    parTimes = {"min":0, "max":0, "avg":0, "normalize":0}

    ##initialize jit functions
    pmatrixMin(a)

```



```

pmatrixMax(a)
pmatrixAvg(a)
pmatrixNormalize(a, pmatrixMin(a), pmatrixMax(a))

seqTimes["min"] += timeFunc(matrixMin, a)
seqTimes["max"] += timeFunc(matrixMax, a)
seqTimes["avg"] += timeFunc(matrixAvg, a)
seqTimes["normalize"] += timeFunc(matrixNormalize, a, matrixMin(a), matrixMax
(a))

parTimes["min"] += timeFunc(pmatrixMin, a)
parTimes["max"] += timeFunc(pmatrixMax, a)
parTimes["avg"] += timeFunc(pmatrixAvg, a)
parTimes["normalize"] += timeFunc(pmatrixNormalize, a, pmatrixMin(a), pmatrix
Max(a))

print("parallel matrix min:", "{0:1.2f}X improvement".format(seqTimes["min"]/
parTimes["min"]))
print("parallel matrix max:", "{0:1.2f}X improvement".format(seqTimes["max"]/
parTimes["max"]))
print("parallel matrix average:", "{0:1.2f}X improvement".format(seqTimes["av
g"]/parTimes["avg"]))
print("parallel matrix normalize:", "{0:1.2f}X improvement".format(seqTimes["
normalize"]/parTimes["normalize"]))

```



## Bibliography

- [1] Bhaskaran, R., and Collins, L., 2005, “Introduction to CFD Basics,” Cornell Univ. - Sibley Sch. Mech. Aerosp. Eng. Ithaca, USA.
- [2] Lane, G. L., Schwarz, M. P., and Evans, G. M., 2005, “Numerical Modelling of Gas-Liquid Flow in Stirred Tanks,” Chem. Eng. Sci., 60(8-9 SPEC. ISS.), pp. 2203–2214.
- [3] Chung, T. J., 2010, Computational Fluid Dynamics, Second Edition.
- [4] Ganesh Ram, R. K., Cooper, Y. N., Bhatia, V., Karthikeyan, R., and Periasamy, C., 2014, “Design Optimization and Analysis of NACA 0012 Airfoil Using Computational Fluid Dynamics and Genetic Algorithm,” Appl. Mech. Mater., 664(August), pp. 111–116.
- [5] Shah, S. R., Jain, S. V., Patel, R. N., and Lakhera, V. J., 2013, “CFD for Centrifugal Pumps: A Review of the State-of-the-Art,” Procedia Engineering.
- [6] Badra, J., Khaled, F., Tang, M., Pei, Y., Kodavasal, J., Pal, P., Owoyele, O., Fuetterer, C., Brenner, M., and Farooq, A., 2020, “Engine Combustion System Optimization Using CFD and Machine Learning: A Methodological Approach,” ASME 2019 Internal Combustion Engine Division Fall Technical Conference, ICEF 2019.
- [7] Norton, T., and Sun, D. W., 2006, “Computational Fluid Dynamics (CFD) - an Effective and Efficient Design and Analysis Tool for the Food Industry: A Review,” Trends Food Sci. Technol.
- [8] Sun, D. W., 2007, Computational Fluid Dynamics in Food Processing.
- [9] Malekjani, N., and Jafari, S. M., 2018, “Simulation of Food Drying Processes by Computational Fluid Dynamics (CFD); Recent Advances and Approaches,” Trends Food Sci. Technol.



- [10] Philo, A. M., Butcher, D., Sillars, S., Sutcliffe, C. J., Sienz, J., Brown, S. G. R., and Lavery, N. P., 2018, "A Multiphase CFD Model for the Prediction of Particulate Accumulation in a Laser Powder Bed Fusion Process," *Minerals, Metals and Materials Series*.
- [11] Li, G., and Li, S., 2015, "Physics-Based CFD Simulation of Lithium-Ion Battery under the FUDS Driving Cycle," *ECS Trans*.
- [12] Hilgenstock, A., and Ernst, R., 1996, "Analysis of Installation Effects by Means of Computational Fluid Dynamics - CFD vs Experiments?," *Flow Meas. Instrum.*
- [13] Lai, W. S., Huang, J. Bin, Ahuja, N., and Yang, M. H., 2019, "Fast and Accurate Image Super-Resolution with Deep Laplacian Pyramid Networks," *IEEE Trans. Pattern Anal. Mach. Intell.*
- [14] Tong, T., Li, G., Liu, X., and Gao, Q., 2017, "Image Super-Resolution Using Dense Skip Connections," *Proceedings of the IEEE International Conference on Computer Vision*.
- [15] Zhang, Y., Li, K., Li, K., Wang, L., Zhong, B., and Fu, Y., 2018, "Image Super-Resolution Using Very Deep Residual Channel Attention Networks," *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*.
- [16] Dong, C., Loy, C. C., He, K., and Tang, X., 2016, "Image Super-Resolution Using Deep Convolutional Networks," *IEEE Trans. Pattern Anal. Mach. Intell.*
- [17] Dong, C., Loy, C. C., and Tang, X., 2016, "Accelerating the Super-Resolution Convolutional Neural Network," *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*.



- [18] Kim, J., Lee, J. K., and Lee, K. M., 2016, “Accurate Image Super-Resolution Using Very Deep Convolutional Networks,” Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition.
- [19] Ledig, C., Theis, L., Huszár, F., Caballero, J., Cunningham, A., Acosta, A., Aitken, A., Tejani, A., Totz, J., Wang, Z., and Shi, W., 2017, “Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network,” Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017.
- [20] Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y., 2014, “Generative Adversarial Nets,” Advances in Neural Information Processing Systems.
- [21] Perez, L., and Wang, J., 2017, “The Effectiveness of Data Augmentation in Image Classification Using Deep Learning.”
- [22] Zhu, J. Y., Park, T., Isola, P., and Efros, A. A., 2017, “Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks,” Proceedings of the IEEE International Conference on Computer Vision.
- [23] Wang, Y., Shimada, K., and Farimani, A. B., “Airfoil GAN: Encoding and Synthesizing Airfoils for Aerodynamic-Aware Shape Optimization.”
- [24] Jiang, H., Yeo, R., Nie, Z., Farimani, A. B., and Kara, L. B., “S Tress GAN : A Generative Deep Learning Model for 2D S Tress Distribution Prediction,” pp. 1–14.
- [25] Zheng, J., 2020, “A Method of Leakage Parameters Estimation for Liquid Pipelines Based on Conditional Generative Adversarial Network,” Proceedings of the Biennial International Pipeline Conference, IPC.



- [26] Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., and Chen, X., 2016, “Improved Techniques for Training GANs,” *Advances in Neural Information Processing Systems*.
- [27] Pant, P., and Farimani, A. B., 2020, “Deep Learning for Efficient Reconstruction of High-Resolution Turbulent DNS Data,” (ML), pp. 1–12.
- [28] Fukami, K., Fukagata, K., and Taira, K., 2019, “Super-Resolution Reconstruction of Turbulent Flows with Machine Learning,” *J. Fluid Mech.*
- [29] Liu, B., Tang, J., Huang, H., and Lu, X. Y., 2020, “Deep Learning Methods for Super-Resolution Reconstruction of Turbulent Flows,” *Phys. Fluids*, 32(2).
- [30] Xie, Y., Franz, E., Chu, M., and Thuerey, N., 2018, “TempoGAN: A Temporally Coherent, Volumetric GAN for Super-Resolution Fluid Flow,” *ACM Trans. Graph.*
- [31] Liu, D., and Wang, Y., 2019, “Multi-Fidelity Physics-Constrained Neural Network and Its Application in Materials Modeling,” *Proceedings of the ASME Design Engineering Technical Conference*.
- [32] Subramaniam, A., Wong, M. L., Borker, R. D., Nimmagadda, S., and Lele, S. K., 2020, “Turbulence Enrichment Using Physics-Informed Generative Adversarial Networks,” pp. 1–14.
- [33] Science, C., 2012, “Evaluating the Performance of the Two-Phase Flow Solver InterFoam Evaluating the Performance of the Two-Phase Flow Solver InterFoam.”
- [34] Jasak, H., 2009, “OpenFOAM: Open Source CFD in Research and Industry,” *Int. J. Nav. Archit. Ocean Eng.*
- [35] Jiang, C. M., Esmaeilzadeh, S., Azizzadenesheli, K., Kashinath, K., Mustafa, M., Tchelepi, H. A., Marcus, P., Prabhat, and Anandkumar, A., 2020, “MeshfreeFlowNet: A Physics-Constrained Deep Continuous Space-Time Super-Resolution Framework.”



- [36] Birla, D. “Keras-SRGAN”, (2018), Github repository,  
<https://github.com/deepak112/Keras-SRGAN>.



# Matthew Li

*mwl5628@psu.edu*

---

## EDUCATION:

Bachelor of Science in Computer Science  
Pennsylvania State University, Schreyer Honors College, Class of 2021

---

## SKILLS:

**Programming Languages:** Python, C, C#, C++, Java (learning), MIPS Assembly

**Hardware Languages:** Verilog

**Other skills:** Computer building, Arduino, SolidWorks, Latex, Microsoft Office

---

## TECHNICAL EXPERIENCE:

**Microsoft: Software Engineering Intern for MSAI User Understanding Team** (Summer 2020)

- Performed data analysis on internal email dataset for intelligent email suggestion project
- Worked with the .NET environment and the Azure DevOps tech-stack

**HACKPSU Project: Lluvia, weather-based Spotify playlist web app** (Fall 2018)

- Wrote Python code to directly interact with the Spotify API to fetch and analyze track data
- integrated Python algorithm into Ruby on Rails project using RubyPython gem

**Undergraduate research for PSU THRED Lab** (Summer 2018 - )

(Technology and Human Research in Engineering Design)

- **Spring, Summer 2019:**
  - Accelerating and improving fluid dynamics simulations using neural networks (Keras w/ Tensorflow)
  - Generating a database of Fluid Dynamics simulations using the OpenFOAM CFD software
  - <https://github.com/matthewli125/SRCFD>

---

## AWARDS AND SCHOLARSHIPS

**Summer 2018 Multi-Campus REU (Research Experience for Undergraduates) -- \$4200**

- Research Stipend awarded to undergraduates across all Penn State Campuses
- Received for Summer 2018 Project in THRED Lab

**Dr. David and Mrs. Shirley Wormley Research Scholarship -- \$1500**

- Private scholarship awarded for student research
- Received for Spring 2019 Project in THRED Lab

---

## RELEVANT COURSEWORK:

CMPSC 201—C++ and MATLAB for engineers  
CMPSC 465—data structures and algorithms  
CMPEN 331—computer organization and design  
CMPSC 221—object-oriented programming with web apps  
EE 210—circuits and devices  
CMPSC 461—programming language concepts

CMPSC 311—intro to systems programming  
CMPSC 497—special topics: GPUs and deep learning  
CMPSC 473—operating systems  
CMPSC 464—theory of computation  
CMPEN 362—communication networks  
CMPSC 431W—database management