



EMANUEL EUGÊNIO SILVA MARTINS

BSc in Mechanical Engineering Sciences

ENHANCING DENSITY CURRENT ANALYSIS THROUGH MACHINE LEARNING AND IMAGE PROCESSING TECHNIQUES

MASTER IN MECHANICAL ENGINEERING

NOVA University Lisbon
December, 2024



ENHANCING DENSITY CURRENT ANALYSIS THROUGH MACHINE LEARNING AND IMAGE PROCESSING TECHNIQUES

EMANUEL EUGÊNIO SILVA MARTINS

BSc in Mechanical Engineering Sciences

Adviser: Doctor Moisés Gonçalves de Brito

Assistant Professor, DEMI, FCT-NOVA

Examination Committee

Chair: Doctor Luís Miguel Chagas Costa Gil

Assistant Professor, DEMI, FCT-NOVA

Rapporteur: Doctor Duarte Manuel Salvador Freire Silva de Albuquerque

Assistant Professor, DEMI, FCT-NOVA

Enhancing Density Current Analysis Through Machine Learning and Image Processing Techniques

Copyright © Emanuel Eugênio Silva Martins, NOVA School of Science and Technology, NOVA University Lisbon.

The NOVA School of Science and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

For the honor and glory of the Almighty God.

ACKNOWLEDGEMENTS

Firstly, I'd like to thank Nova School of Science and Technology. As one of the most prestigious and revered educational institutions in the country, it has provided me with great opportunities in both my academic and personal lives. I'd like to thank all of the academic body of the university who have helped in my journey, in particular Professor Moisés Brito, for being my adviser during the writing of this dissertation.

Since I started this journey many years ago, my life has utterly changed for the better. I've had great experiences, some of them weren't so pleasant, but most were wonderful. Aside from all the knowledge I have acquired, I'll be forever grateful for all of the friends I've made along the way.

To my nine precious companions, who have shared many academic burdens with me, I'll never forget all we have lived through: our lunches on the grass; the study sessions in the library; the brownies and coffees; the essays we have written together; the struggles during the lockdowns; the hardships and the parties, the pain and the laughter, the sorrow and the joy. One day, the waters of Heaven will put out the fire of my heart. . . and *Tchhh!* But your names will be written in it forever.

Rita, the only thing that brings me more joy than annoying you is the fact that you still choose to be my friend despite it all. As if your friendship wasn't enough (which it absolutely is), you have also made me fall in love with the beautiful land of Monção! Thank you for putting up with me and for always making me feel at home.

Basílio, our conversations are always incredibly fun. From books to places, from movies to ideas, from music to people. You're interested in so many things that it's no surprise you're such an interesting person. I'm confident that's where your ability to so effortlessly connect with anyone comes from. I'm thankful for being one of those connections!

Catarina, you know so much about me and yet you never judge. It takes a very kind heart to do that. Your name is written all over this document, Miss Alves. Thank you for making this dissertation possible and for always being available to help me. Above all, thank you for worrying so much, for constantly calling just to know how I'm doing, and for giving me so much emotional support throughout these years.

Dinís, you are my most cherished nemesis. It seems that we never agree on anything, but our hearts are aligned on the absolutely critical importance of friendship, loyalty and adventure! Our long-winded discussions are always extremely entertaining, as are all the other experiences we share. Next time we meet, let's have a toast celebrating all the times I'm correct, and take the opportunity to leave everyone tired of listening to us, like we usually do. Cheers!

Diogo, you always pass the vibe check with flying colors! From our profound personal conversations, to our funny and silly moments, you always add so much depth to every scenario we're in. Thank you for always making what's good even better!

Mafalda, you bring so much joy to all of us! Never let anyone or anything bring that cheerful spirit down. Even going to Barreiro becomes exciting if it means being around your amazing energy. Thank you for being such a mood booster!

Margarida, you are our true founder! I gave the group its name, but you officially created it when I told you I'd stay at Nova, remember? You're one of the reasons I made that (correct) decision. I couldn't live without being a regular nuisance to you! Thank you for helping me choose to stay and for having the proactive attitude to make our group official!

Miguel, you saw me for who I am since day one, and I have felt your trust ever since. There's no greater display of strength than trust, which makes you an extremely powerful man. Thank you for always being there for me, and for consistently providing us with one of the best days of the year every August 7th!

Melo, how dare you be so humble? Smart, funny, amazing dance moves. . . couldn't you at least be a little arrogant sometimes? For real though, you came in later, but you're an indispensable part of our group. Thank you for making us complete.

We have asked for it, and we've gotten it: The *Tchhh* will be forever!

The people I took with me from college are much more valuable than any knowledge I have acquired. The ones I've met elsewhere have also been indispensable in this journey.

To my lovely Sarah: No matter how far apart, no matter the years without seeing each other, no matter where life takes us, you'll forever be my dear sister, the one who found me when I was but a mere toddler. I was young, but I'll never forget the first time I saw you. It changed my life. Kiss our mother and hug her in my stead, for she took such good care of me! I love you mom Elaine! You are my family, both of you, and Manu as well. I can only imagine the joy of reuniting with you someday. Thank you for all the love and care you have given me.

To my dear Teresa: You have seen me grow, and have grown a lot yourself! It's a pleasure to share this life with you. We walk, we talk and we become better together. You listen, you understand and you counsel me, with your patience and kindness, without ever letting me off the hook too easily. I take great delight in your company! Thank you for all the encouragement throughout these years.

To my sweet Leonor: You inspire me greatly. Everything I've done can't compare to anything you have already accomplished in your young life, and yet I never fail to feel

appreciated by you. Your confidence in me is much greater than my own. Lay a kiss upon each of your daughters and tell them it came with love, for their uncle misses them. Thank you for your friendship and your unrelenting trust.

To my first friend from this land, Pedro: You can't imagine how much I cherish the nostalgic childhood shared. The bike rides around the neighborhood; the parkour sessions where I could do little more than watch; the holidays at the beach and at the pool; the videogames we've played; the music we've listened to; the movies we've watched. I even cherish the struggles I've had for wanting to be like you! I've looked up to you from the moment we've met. Our childhood has shaped a lot of who I am, and even to this day, I see you as an older brother. Thank you for all the fond memories, and for inspiring me to always do better.

To my brother in arms, Leonardo: I have never met anyone as honorable and as loyal as you. If I had to pick someone to go to war with me, you'd be my first choice! Your trust is one of the most valuable things I have in my life, for I know you don't give it away too easily, and I will never take it for granted. Thank you for giving me the honor of being your friend.

To my fellow thinkers André, Malik and Miguel: You are the gardeners of my curiosity! Thanks to you, it has been able to grow and flourish, and give many good fruit. Our late night talks, walking through the streets; our dinners at Malik's house; the knowledge we share with each other constantly. It's a pleasure to be able to cultivate this garden with you. Thank you for not letting it die. Our quest for wisdom continues!

André, I also thank you for having so much empathy and kindness in your heart. You understand my way of thinking and my way of living better than anyone else. You always have my back and stand up for me, even when we disagree. Thank you for always making me feel heard and understood!

Miguel, a special thank you for all the help you have given me with the writing of this dissertation. Your knowledge of Python and Latex, and your unmatched intellect in general has made this work possible. I'm beyond grateful!

And Malik, we are now both free to pursue our crazy ambitions. You never fail to inspire me and to hype me up about what we can accomplish together. Thank you for believing in me!

To my fellow elect, my brothers and sisters in Christ: I thank my God upon every remembrance of you! João, Joana, Samuel, Ana, Rute, Lucas and all the others who are with me in this journey! Although it hasn't been that long since we've known each other, that which binds us is greater and stronger than anything in this world. Coming closer to God alongside you has been an unbelievable blessing. Thank you for all the prayers you have sent my way! God bless you all.

To my family in Brazil, in particular to my cousins Marcel, Mirela and Mikaella and to my uncle and aunt, Messias and Bernadeth: I have recently been with you, after fifteen years apart. It was like nothing had changed. I felt at home, as if I'd never left! I lovingly

cherish the fun we had in our youth, and the care I've always received whenever I'm with you. May our reunions be more often, and thank you for all the kindness!

Also, to my dear godmother, aunt Graça: Thank you for supporting me and for everything you did for me during my childhood. Your presence in my life was essential for me to become who I am today. Many blessings to you and to my little cousin!

I have received a lot of gifts during my life, but the best one was given to me by my mother. I had been asking for this gift since I was a little kid, and on the 24th of April of 2008 she finally gave birth to you, my dear brother. Gabriel, thank you for all the fun and joy you bring to me. It has been a pleasure to watch you grow, and to be able to share this amazing life with you. We may have our differences, but none of that matters, because we came from the same womb, and you are truly a blessing upon my life. For the honor of the woman who gave me such a precious gift, I'll never let anything or anyone drift us apart, and I'll cherish you forever!

To my borrowed father, Gonçalo: I thank you as well for giving me my brother, and for raising me as your own. I'll always be grateful for all that you did and still do for me, even to this day!

To the greatest woman in the world, Maria: My beloved mother, I could never thank you enough. You have brought me into this world, raised me and took care of me. For that alone, I'm beyond grateful. But you did so much more. You have taught me so many things: how to be independent; how to be strong; how to think for myself; how to fight for the ones I love! You have taught me respect, and I ask forgiveness for all the times I haven't shown it to you.

We have struggled a lot together, but I know you wouldn't change a thing. Neither would I. All of the pain and suffering has brought us closer together, and as we like to say: isn't suffering what life is all about?

Know that the life you have endured for my sake lives constantly in my mind, in my heart and in my soul. You are an admirable woman, a struggler and a true warrior, beyond all measures! I can only dream of one day being as fierce as you are.

You are my inspiration, and I want nothing more than to make you rejoice in being able to say: "I'm proud of my son"! Thank you for everything you have done for me. I love you.

Finally, and above all, to my Father, the Lord my God: How kind and generous You are to me! You have always been present, even when I turned my back on You. And as it went with the prodigal son, when I came back, You willingly received me with your comforting embrace. Thank You for blessing me with such a bountiful life! I pray that you stay with me, Lord, and as You've helped me win this small battle, give me also the strength to glorify You in the wars to come.

Now and forever, may Your will be done upon my life, for I trust in Your unfailing love and I know that no matter what, You will always take good care of me.

Amen.

” *“Be ye strong therefore, and let not your hands be weak:
for your work shall be rewarded.*”

— 2 Chronicles 15:7

ABSTRACT

Laboratory measurements of density current propagation are usually carried out through image acquisition methods. Processing these images allows for the extraction of fundamental flow parameters such as the current boundary, front velocity, head position, and density field. This processing generally requires complex computational routines, and the objective of this dissertation is to create more efficient and robust routines using the Python programming language.

This dissertation investigates the application of *Machine Learning (ML)* techniques in image processing for the study of density currents. The specific objectives of this dissertation include developing Python codes for distortion correction and density field estimation, and integrating machine learning techniques to improve image processing. The development of the code for density field estimation was carried out by translating a MATLAB code used by Alves [5].

With the Python code for density field estimation developed, its improvement through the implementation of *machine learning* could be addressed. The density profiles obtained through the implementation of a *Random Forest Regressor* showed smoother transitions than the original, and were also less sensitive to noise. Additionally, model evaluation resulted in low *Mean Squared Error (MSE)* values. These observations demonstrate potential improvements in the accuracy and robustness of the density profile calculations through the integration of *machine learning*.

Keywords: Image Processing, Density Currents, Lock Exchange Experiment, Machine Learning, Python

RESUMO

As medições laboratoriais da propagação de correntes de densidade são normalmente feitas através de métodos de aquisição de imagem. O processamento dessas imagens permite obter os parâmetros fundamentais do escoamento como o contorno da corrente, a velocidade da frente, a posição da cabeça e o campo de densidades. Este processamento geralmente requer rotinas computacionais complexas, sendo o objetivo desta dissertação criar rotinas mais eficientes e robustas com recurso à linguagem de programação Python.

Esta dissertação investiga a aplicação de técnicas de *Machine Learning* (ML) no processamento de imagens para o estudo de correntes de densidade. Os objetivos específicos desta dissertação incluem desenvolver códigos Python para correção de distorção e estimativa de campo de densidades, e integrar técnicas de machine learning para melhorar o processamento de imagem. O desenvolvimento do código para a estimativa do campo de densidades foi realizado através da tradução de um código MATLAB utilizado por Alves [5].

Com o código Python para a estimativa do campo de densidade desenvolvido, a sua melhoria através da implementação de *machine learning* pôde ser abordada. Os perfis de densidade obtidos através da implementação de um *Random Forest Regressor* mostraram transições mais suaves que o original, sendo também menos sensíveis ao ruído. Adicionalmente, a avaliação do modelo resultou na obtenção de valores baixos de *Mean Squared Error* (MSE). Estas observações demonstram potenciais melhorias na precisão e robustez do cálculo dos perfis de densidade através da integração de *machine learning*.

Palavras-chave: Tratamento de Imagens, Correntes de Densidade, Experiência de Corrente de Densidade, Aprendizagem Automática, Código Python

CONTENTS

List of Figures	xix
List of Tables	xxi
Acronyms	xxv
1 Introduction	1
1.1 Introduction	1
1.2 Motivation	1
1.3 Objectives	2
1.4 Methodology	2
1.5 Structure of the Dissertation	2
2 State of the Art Review	5
2.1 Introduction	5
2.2 Density Currents	5
2.2.1 Definition	5
2.2.2 Density Current Anatomy	7
2.2.3 Density Current Characterization	8
2.2.4 Lock-Exchange Method	9
2.2.5 Current and Obstacles Interaction	9
2.3 Image Processing Tools and Techniques	10
2.3.1 Preprocessing	11
2.3.2 Thresholding	11
2.3.3 Segmentation	11
2.3.4 Morphological Image Processing	11
2.3.5 Feature Detection	11
2.3.6 Transformations	11
2.3.7 Gaussian Image Processing	11
2.3.8 Edge Detection	12

2.4	Frameworks and Libraries	12
2.4.1	NumPy	12
2.4.2	SciPy	12
2.4.3	OpenCV	12
2.4.4	scikit-image	13
2.4.5	Matplotlib	14
2.5	Machine Learning and AI	14
2.5.1	Historical Perspective	14
2.5.2	Fundamental Concepts	17
2.6	Neural Networks	20
2.6.1	Convolutional Neural Networks (CNNs)	21
2.6.2	Recurrent Neural Networks (RNNs)	21
2.6.3	Generative Adversarial Networks (GANs)	22
2.7	Deep Learning Frameworks	23
2.7.1	TensorFlow	23
2.7.2	PyTorch	23
2.7.3	Keras	23
2.8	Machine Learning Applications in Fluid Mechanics	24
3	Experimental Tests	27
3.1	Introduction	27
3.2	Experiment Description	27
3.3	Image Analysis Technique	30
3.4	Density Calibration	32
4	Implemented Python Code	35
4.1	Introduction	35
4.2	Distortion Correction	35
4.2.1	Distortion Correction - Flowchart	35
4.2.2	Distortion Correction - Detailed Description	38
4.2.3	Distortion Correction - Results and Discussion	46
4.3	Density Estimation	48
4.3.1	Density Estimation - Flowchart	48
4.3.2	Density Estimation - Detailed Description	50
4.3.3	Density Estimation - Results and Discussion	57
5	Density Estimation: Machine Learning Improvement	63
5.1	Introduction	63
5.2	Random Forest Regressor	63
5.2.1	Random Forest Regressor: Flowchart	63
5.2.2	Random Forest Regressor: Detailed Description	67
5.2.3	Random Forest Regressor: Results and Discussion	70

5.3	Further Machine Learning Improvements	75
5.3.1	Spatial Consistency	75
5.3.2	Temporal Consistency	78
6	Conclusion	81
	Bibliography	85
	Appendices	
A	Distortion Correction Code	91
B	Density Estimation Code: Interpolation	95
C	Density Estimation Code: Random Forest	99

LIST OF FIGURES

2.1	Scheme of a bottom gravity current with density ρ_c intruding in the less dense ambient fluid with uniform density ρ_a [47]	6
2.2	Examples of gravity currents in the environment: (a) Copper River (upper center) spills light grey sediments into the Gulf of Alaska; (b) eruption of Soufri'ere Hills volcano, in the island of Montserrat; (c) avalanche from Hunza Peak, Karimabad, in Pakistan and (d) dust storm observed in the military Camp Fallujah, Iraq [47]	6
2.3	The general shape of a bottom-boundary gravity current of density ρ_c propagating over a horizontal surface in a less dense ambient fluid of density ρ_a [2]	7
2.4	Mixing process due to instabilities at the density current front: (a) Kelvin-Helmholtz billows; (b) Lobes and Clefts [68]	8
2.5	Gravity currents produced by lock-exchange where the darker and denser fluid is separated by a vertical gate from the lighter fluid, in the same height conditions [64]	10
2.6	Neural network architecture, with respective input, hidden and output layers identified [77]	20
2.7	Example of convolutional neural network architecture for classification [4] .	21
3.1	Configuration of the channel: a) top view and b) perspective view [5]	28
3.2	Lock zone: a) vaseline filling the gap between lock and channel wall and b) gate positioned [5]	28
3.3	Experiment Flowchart [5]	29
3.4	Camera Placement [5]	30
3.5	Raw images from StreamPix software: a) first half of the channel, lock view; b) second half of the channel [5]	31
3.6	Calibration steps and successive decrease of the density in the channel from a darker shade and denser fluid to a lighter shade and least dense one: a), b), c), respectively [5]	32
3.7	Sequence of Calibration Images [5]	33
3.8	Calibration Curve [5]	33

4.1	Distortion Correction Flowchart	37
4.2	Pinhole camera model [79]	38
4.3	Chessboard pattern used for camera calibration [5]	40
4.4	Radial distortion: a) Barrel distortion; b) Pincushion distortion [37]	44
4.5	a) Distorted Image; b) Undistorted Image	46
4.6	Corner detection on one of the calibration images	46
4.7	Density Estimation Flowchart	49
4.8	Calibration Area	52
4.9	Calibration Images: a) 1st Image b) 7th Image c) 14th Image	53
4.10	Density estimation plot	57
4.11	Density graphs: a) Obtained with the current code; b) Obtained by Alves, where the different colored lines represent the density profiles at different sections of the flow, with the green one representing the flow at section $x=0.7m$ [5] . . .	58
4.12	Density estimation plot with gaussian filter applied	59
4.13	Density graph at $x=0.7m$ with gaussian filter applied	59
4.14	Raw image with inverted rows	60
4.15	Density estimation plot with segmentation as done in the original MATLAB code [5]	61
5.1	Random Forest Model Structure [11]	64
5.2	Random Forest Regressor Code Flowchart	66
5.3	Density estimation plot with shorter channel view: a) Interpolation Function; b) Random Forest Regressor	71
5.4	Density estimation plot obtained using: a) Interpolation Function; b) Random Forests	72
5.5	Density estimation plot with gaussian filter applied: a) Interpolation Function; b) Random Forest	73
5.6	Density graphs with progressively more intense gaussian filters applied in the Random Forest model: a) $\sigma = 1$; b) $\sigma = 2$; c) $\sigma = 3$	74

LIST OF TABLES

5.1	Density values for each of the fourteen calibration images	79
5.2	Updated data calibration, with time as a variable	80

LIST OF LISTINGS

1	Distortion Correction - Libraries	39
2	Distortion Correction - Path to calibration images	39
3	Distortion Correction - Chessboard size	40
4	Distortion Correction - Arrays to store object points and image points . .	40
5	Distortion Correction - Prepare object points	41
6	Distortion Correction - Glob to get all file names that match the pattern .	41
7	Distortion Correction - Loop through calibration images	42
8	Distortion Correction - Calibrate the camera using all detected corners .	43
9	Distortion Correction - Set k_1 , k_2 , and k_3 to zero	43
10	Distortion Correction - Display the updated camera matrix and distortion coefficients	44
11	Distortion Correction - Glob to get all file names of images to undistort .	45
12	Distortion Correction - Loop through each image and undistort	45
13	Density Estimation - Libraries	50
14	Density Estimation - Input information	50
15	Density Estimation - Load calibration data	51
16	Density Estimation - Visualize calibration images	52
17	Density Estimation - Loop through experiment folders	54
18	Density Estimation - Process experiment images	54
19	Density Estimation - Loop through image pixels and perform density esti- mation via interpolation of the calibration data	55
20	Density Estimation - Plot and save Raw and Density images	56
21	Density Estimation - Write frames to video	57
22	Density Estimation - Print k and release video files	57
23	Random Forest - Libraries	67
24	Random Forest - Prepare training data for Random Forest	67
25	Random Forest - Split into training and validation sets	68
26	Random Forest - Train the Regressor	68
27	Random Forest - Evaluate the model	69

28	Random Forest - Loop through image pixels and perform density estimation via Random Forest Regression of the calibration data	69
29	Convolutional Neural Network - Model definition	76
30	Convolutional Neural Network - Model training	78
31	Distortion Correction - Full Code	93
32	Density Estimation - Full Code	98
33	Random Forest - Full Code	102

ACRONYMS

AI	Artificial Intelligence
BPTT	Backpropagation Through Time
CFD	Computational Fluid Dynamics
CNN	Convolutional Neural Network
CNNs	Convolutional Neural Networks
DL	Deep Learning
GANs	Generative Adversarial Networks
GRUs	Gated Recurrent Units
LSTM	Long Short-Term Memory
ML	Machine Learning
MSE	Mean Squared Error
NNs	Neural Networks
nRMSE	Normalized Root Mean Squared Error
RANS	Reynolds-averaged Navier–Stokes
RMSE	Root Mean Squared Error
RRN	Recurrent Neural Network
RRNs	Recurrent Neural Networks
SVMs	Support Vector Machines
TBNN	Tensor Basis Neural Network

INTRODUCTION

1.1 Introduction

This opening chapter serves to contextualize and give an overview of the subjects discussed in the present dissertation. The motivations for the current study will be addressed, as well as the main goals that are hoped to be achieved.

The methodologies applied will be evidenced and the chapter ends with the overall structure of the dissertation.

1.2 Motivation

Fluid mechanics is the study of fluids either in motion (fluid dynamics) or at rest (fluid statics). One particularly significant area of study within this field are density currents, which are primarily horizontal flows generated by a density difference between two fluids. These currents regularly occur in nature and are of considerable importance for various scientific and engineering applications [90, 42].

In order to effectively study these and other types of fluid flows, researchers oftentimes record their experiments and analyze the data using computational methods. The accuracy of the predictions, and the applicability of the findings to real-world scenarios as well, heavily depend on the robustness of these computational technologies. With the recent developments in Machine Learning (ML) and Artificial Intelligence (AI), it is important for engineers to integrate these advanced computational tools into their research in order to enhance the precision of their results.

In [5], a series of lock-exchange experiments were conducted to generate density currents, with subsequent density estimation performed using a MATLAB code. To pave the way for advancements in fluid flow image processing, this MATLAB code will be translated into Python, the predominant language for machine learning.

In addition to the density estimation code, a distortion correction MATLAB code was used to correct the slight distortion introduced by the camera during the experiment. For the present dissertation, a distortion correction code will also be developed into Python.

1.3 Objectives

The main objective of this dissertation is to enhance the analysis of density currents with machine learning and image processing techniques.

A crucial part of this study involves the development of the image distortion correction Python code and the translation of the density estimation MATLAB code used by Alves. Having both codes in Python will serve as a bridge, enabling their integration into new frameworks and potentially improving their functionality.

Furthermore, the dissertation seeks to explore the intersection of machine learning and image processing techniques with fluid mechanics. By researching these advanced computational methods, the study aims to understand their applicability to density estimation. The goal is to use machine learning and image processing to refine and improve the density estimation code, making it more robust and accurate. The dissertation will culminate with recommendations for future works, providing a roadmap for continued advancements in this field.

1.4 Methodology

After the relevant bibliography review is presented, and an overview of the work done by Alves is laid out, the two Python codes will be presented with their respective flowcharts and detailed descriptions. Their results will be analyzed and compared with the ones previously obtained.

To enhance the density estimation code, a Random Forest Regressor will be employed. Random Forests, a form of supervised learning, are instrumental for both classification and regression tasks. Additionally, other image processing techniques like gaussian filtering and segmentation will be explored and integrated as necessary.

However, due to constraints in computational resources, the dissertation will primarily focus on analysing the applicability of more sophisticated Deep Learning (DL) models rather than implementing them. While these advanced models will not be executed, a guide for their implementation will be provided, paving the way for future research and development.

1.5 Structure of the Dissertation

The dissertation is structured into six chapters, each aimed at addressing various aspects of the research.

Chapter 1, provides context and outlines the motivations behind the study. It introduces the main goals and methodologies, ending in a brief overview of the dissertation's structure.

Chapter 2 delves into an examination of existing knowledge and technologies relevant to the study. This chapter is further divided into several sub-sections including:

- Density Currents;
- Image Processing Tools and Techniques;
- Frameworks and Libraries;
- Machine Learning and Artificial Intelligence;
- Neural Networks;
- Deep Learning Frameworks;
- Machine Learning Applications in Fluid Mechanics.

These sections collectively set the foundation for the computational work to follow.

Chapter 3 focuses on the practical aspects of the research. It begins with an introduction to the experimental setup, followed by a description of the experiments conducted. The chapter then discusses the image analysis techniques employed and the methods used for density calibration.

In Chapter 4 the dissertation transitions to the implementation of the computational model. This chapter includes detailed descriptions for both codes, as well as their results. By translating the existing density estimation code into Python, the study aims to leverage the advanced computational capabilities of this programming language to enhance the analysis of the lock-exchange experiment.

Chapter 5 explores the integration of machine learning techniques to improve the density estimation process. To start, it describes the implementation of a Random Forest Regressor to replace an interpolation function. Additionally, the chapter discusses more sophisticated deep learning models like Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RRNs), providing a roadmap for their future implementation.

Finally, Chapter 6 summarizes the findings and contributions of the dissertation. It reflects on the research objectives, discusses the implications of the results, and offers recommendations for future work.

STATE OF THE ART REVIEW

2.1 Introduction

In this chapter, a bibliography review of the the different areas of investigation which are relevant to this dissertation will be conducted, starting with a review of density currents, which are the type of fluid flow we intend on studying. The lock-exchange experiment also comes into the picture as closely related to density currents.

After that, an analysis will be carried out of programming tools which might be useful for the development of the computational models.

Lastly, a research on Machine Learning (ML) and Artificial Intelligence (AI) will take place: their historical emergence, some fundamental topics and how they're applicable to fluid mechanics.

2.2 Density Currents

2.2.1 Definition

A density (or gravity) current occurs when a fluid of density ρ_c propagates into another fluid of a different density, ρ_a . These density gradients, which are primarily horizontal, may happen due to suspended materials within the fluids or temperature differences between the two. This phenomena generates a current with very particular characteristics, which we call a density current [76].

Depending on their density differences and the stratification of the ambient fluid, gravity currents can be classified as bottom currents, top currents or intermediate currents. Bottom currents occur when the current is denser than the ambient fluid, as shown in Figure 2.1. Sometimes, however, the ambient fluid can be denser than the current, generating thus a top current [47].

Gravity currents are present in the atmosphere, where they can take the form of thunderstorm outflows or sea-breeze fronts. They can also occur as atmospheric-suspension gravity currents, such as avalanches of airborne snow particles, or fiery avalanches and base surges formed from gases and solids issuing from volcanic eruptions [67].

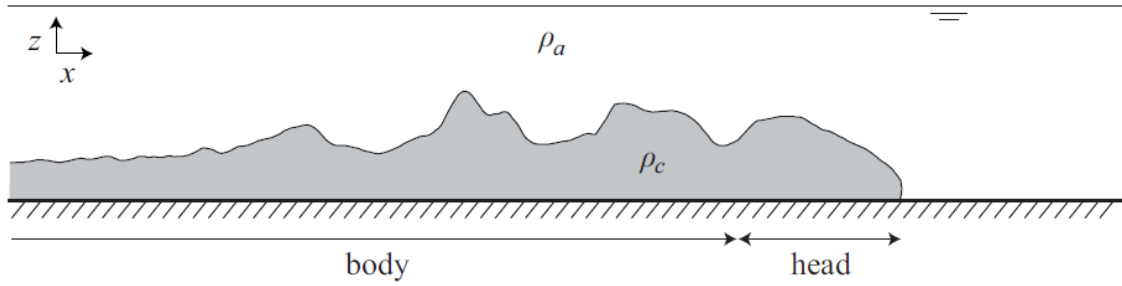


Figure 2.1: Scheme of a bottom gravity current with density ρ_c intruding in the less dense ambient fluid with uniform density ρ_a [47]

Some of the examples of gravity currents found in nature can be seen in Figure 2.2.



Figure 2.2: Examples of gravity currents in the environment: (a) Copper River (upper center) spills light grey sediments into the Gulf of Alaska; (b) eruption of Soufrière Hills volcano, in the island of Montserrat; (c) avalanche from Hunza Peak, Karimabad, in Pakistan and (d) dust storm observed in the military Camp Fallujah, Iraq [47]

These currents also occur in the ocean, and are caused by differences in temperature and salinity or by turbidity currents (turbidity currents are underwater avalanches or flows of sediment-laden water that move downslope from shallow to deeper waters, often triggered by earthquakes or other disturbances) with suspended mud or silt, and can be identified by lines of foam and debris on the surface of the ocean. These lines are often created by tidal processes, which also affect the behavior of gravity currents, such as river

plumes at the surface and salt wedges on a river bed [67].

Gravity currents have important practical uses in several areas, including aircraft safety, atmospheric pollution, pest control, and dense-gas technology. Additionally, they can be formed as a result of industrial accidents, such as the unintentional release of dense gas [67].

The wide range of environments in which gravity currents can manifest underscores the importance of investigating them and conducting controlled experiments. Such endeavors are essential for gaining deeper insights into the workings of natural phenomena and for preventing potential accidents.

2.2.2 Density Current Anatomy

Density currents exhibit a complex anatomy that influences their behavior and interactions with the surrounding environment. At the forefront of the current there is a distinct feature called the “head”, which is notably deeper than the following flow [67].

Figure 2.3 shows the typical density current anatomy, with the body and head.

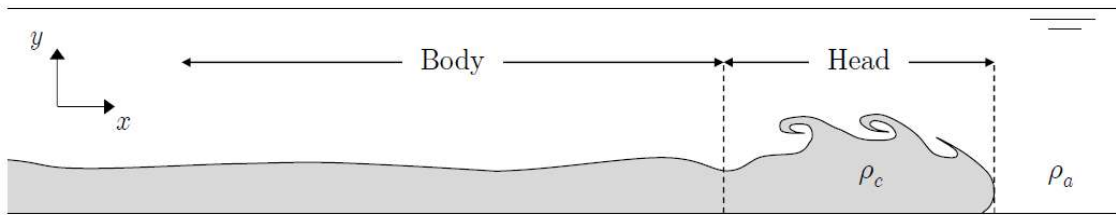


Figure 2.3: The general shape of a bottom-boundary gravity current of density ρ_c propagating over a horizontal surface in a less dense ambient fluid of density ρ_a [2]

The head of the current has a nose which is usually raised above the ground and the rest of the flow. The frontal region displays a complex and dynamic flow pattern, exhibiting three-dimensional unsteadiness originating from shear and gravitational instabilities. These instabilities are responsible for regulating the mixing processes within the zone. The two primary modes observed in this context are billows, which bear resemblance to Kelvin-Helmholtz instabilities, and an intricate, ever-shifting pattern characterized by lobes and clefts, as shown in Figure 2.4. These modes collectively contribute to the intricate behavior observed within the frontal zone [2].

The flow within the body region can be steady (or quasi-steady) or unsteady, depending on the way it has been initiated. Currents that are generated by instantaneous or short-duration release of a dense fluid typically have an unsteady flow in the body region. In this scenario, the characteristic variables that define the flow are both space and time dependent. On the other hand, currents that are continuously fed typically have a body with quasi-steady flow. This happens because there is the constant supply of denser fluid that balances the density reduction of the current due to mixing with the surrounding fluid. In this quasi-steady regime, the characteristic variables exhibit small variations in time [47].

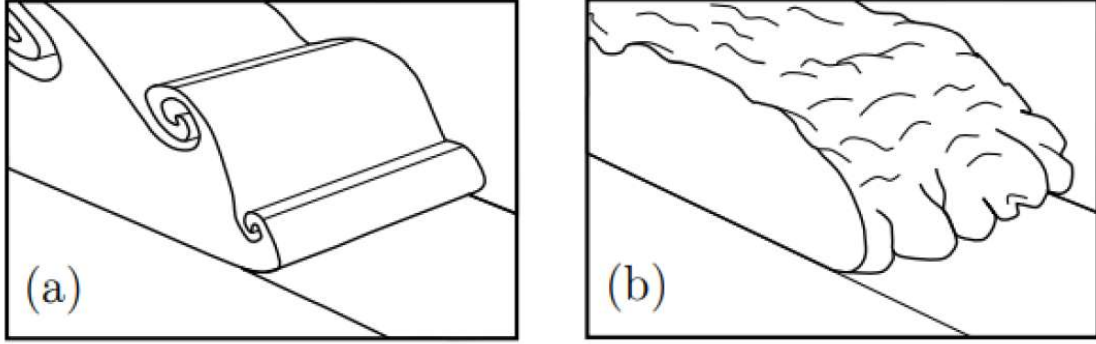


Figure 2.4: Mixing process due to instabilities at the density current front: (a) Kelvin-Helmholtz billows; (b) Lobes and Clefts [68]

2.2.3 Density Current Characterization

In order to better understand gravity currents, it's useful to define some important dimensionless parameters that govern their behavior, the first of which is the reduced gravity. If due to buoyancy forces a fluid comes into contact with another of different density, there will be an effective change in the acceleration acting on the first one. The reduced gravity pertains to this change in the acceleration and can be defined as follows:

$$g' = g \left(\frac{\Delta\rho}{\rho_0} \right) \quad (2.1)$$

Where $\Delta\rho$ is the average density difference, ρ_0 is the density of the less dense fluid and g is the gravitational acceleration [5, 76].

As is the case with many types of flows, the Reynolds number is another relevant dimensionless parameter. This number characterizes the relative importance of inertial and viscous forces and is defined as:

$$Re = \frac{UH}{\nu} \quad (2.2)$$

Where U is the depth-averaged current velocity, H is the depth of the current and ν is the kinematic viscosity of the current's fluid [2].

Another important parameter is the densimetric Froude number. This parameter represents the ratio of the current speed to the speed of a long wave propagating along the interface between the fluids of a density current. Furthermore, the Froude number can also be seen as the ratio between the destabilizing effects of velocity shearing inertial forces to the stabilizing effects of vertical density stratification gravity/buoyancy forces [62].

Since the action of gravity in density currents depends on the density differences between the two fluids, the Froude number can be defined with respect to the reduced gravity parameter [34]:

$$Fr' = \frac{\bar{U}}{\sqrt{g'd}} \quad (2.3)$$

Finally, the current propagation velocity can be obtained from simple energy conversion arguments and is defined as [76]:

$$U = \sqrt{g'h_0} \quad (2.4)$$

2.2.4 Lock-Exchange Method

The lock-exchange method is an experiment that consists in the sudden release of a gate that induces the flow of a denser fluid with fixed volume into a lighter fluid creating a density current. The denser fluid then flows from the gate towards the end of a channel, while the lighter fluid flows in the opposite direction.

For the purposes of the current dissertation, the lock-exchange experiment that will be studied is the one done by Alves [5], where the rectangular tank in which the experiment was conducted had a length that was much greater than its width. In this way, the flow can be considered two dimensional since the development of the current only happens in two directions, height and length.

This experiment was carried out with a full-depth release. This means that depth of both fluids is the same before the release of the gate.

Figure 2.5 shows an example of the development of a density current at different stages of the experiment.

Usually there are three different phases that can be identified during the experiment. The first one, which happens right after the gate release, is the slumping phase, wherein the forward movement of the current is hindered by the opposing flow of the ambient fluid. Subsequently, a purely inertial (or self similar) phase can ensue, where the buoyancy force exerted by the intruding fluid is counterbalanced by the inertial force acting upon it. This is followed by a viscous phase, wherein the buoyancy force is balanced by viscous forces. The inertial phase can be absent if the viscous effects become prevalent before the slumping phase is complete [28, 58].

2.2.5 Current and Obstacles Interaction

During these experiments it is common to introduce obstacles in the canal in order to simulate the propagation of a density current through vegetation and other natural obstacles.

There are three stages of the interaction between the flow and the obstacles that can be identified: the impact stage, the transition stage and the quasi-steady stage. During the impact stage the current flows upstream and encounters an obstacle, partially passing over it, while another portion is reflected back. The peak of the run up occurs in the transition stage, where a vortex is created upstream the obstacle due to the two fluids

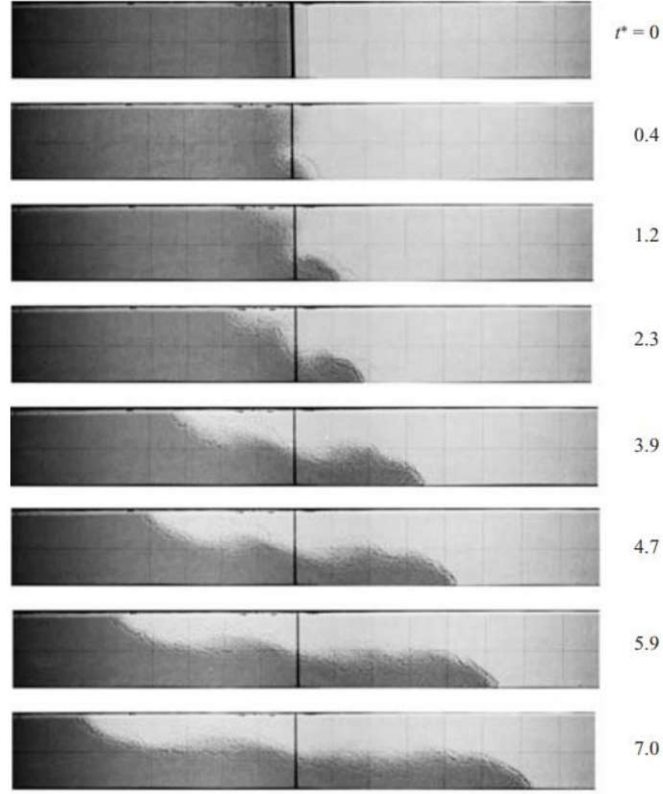


Figure 2.5: Gravity currents produced by lock-exchange where the darker and denser fluid is separated by a vertical gate from the lighter fluid, in the same height conditions [64]

having different velocities with different directions. Lastly, in the quasi-steady stage the current has already passed through the obstacle and the denser fluid flows with a quasi-steady behavior. The introduction of obstacles induces these stages in the current and serve to stimulate the mixing between the two fluids [45].

2.3 Image Processing Tools and Techniques

With an overview of density currents and the lock-exchange experiment complete, it's important to now delve into the state of the art of the image processing tools available for achieving the goal of the current dissertation.

It's important to remember that the main objective of this dissertation is to improve the density current analysis by leveraging machine learning and image processing techniques. In order to effectively achieve this, a thorough investigation of the technology available had to be conducted.

There are several image processing techniques which are widely used for different purposes. The following sub-chapters describe some of the most relevant techniques, with some of them being later used in the implemented Python code.

2.3.1 Preprocessing

Image preprocessing involves preparing the image for further processing and analysis. Techniques include noise reduction, contrast enhancement, and image sharpening. These steps can improve the quality of the image, making it easier to extract meaningful information in the later stages of analysis [70, 20].

2.3.2 Thresholding

Thresholding is a method used to segment an image into foreground and background. This is done by selecting a threshold value, and then classifying all pixels with values above this threshold as part of the foreground, and all other pixels as part of the background [63].

2.3.3 Segmentation

Image segmentation is the process of dividing an image into multiple segments or regions, where each segment corresponds to different objects or parts of the image. Techniques include thresholding, edge detection, and region growing [44].

2.3.4 Morphological Image Processing

Morphological image processing is a technique used in computer vision and image processing to analyze and manipulate the shapes and structures of objects within an image. It involves the application of mathematical operations, called morphological operators, to modify the geometric attributes of the image, such as its size, shape, and connectivity [46, 33].

2.3.5 Feature Detection

Feature detection involves identifying and locating relevant structures or objects in the image, such as edges, corners, and blobs. This information can be used for higher-level tasks like object recognition and image matching [74].

2.3.6 Transformations

Transformations involve changing the perspective or structure of an image. This includes operations like rotation, scaling, and translation [74].

2.3.7 Gaussian Image Processing

Gaussian image processing is a technique commonly used in computer vision and image processing to apply gaussian filters to images. The gaussian filter is a type of linear filter that is based on a gaussian distribution. It is used to smooth or blur an image, reducing noise and enhancing important features [18, 33].

2.3.8 Edge Detection

Edge detection is an essential technique in image processing that aims to identify and highlight boundaries or edges between different regions in an image. It plays a crucial role in various computer vision tasks, including object recognition, image segmentation, and feature extraction.

The goal of edge detection is to locate significant changes in pixel intensity values, which often correspond to transitions between different objects or regions in an image. These transitions can be caused by variations in color, texture, or brightness. By identifying and extracting these edges, we can obtain a simplified representation of the image that captures the most important structural information [74, 33].

2.4 Frameworks and Libraries

In order to implement the image processing techniques mentioned above, programming frameworks and libraries need to be used. The following sub-chapters describe a few of the most important frameworks and libraries in the image processing field.

2.4.1 NumPy

NumPy is a powerful Python library that not only excels in performing general mathematical processes and calculations but also offers capabilities for simple image processing techniques. It allows tasks such as flipping images, extracting features, and performing image analysis. NumPy is widely used in the field of image processing due to its efficient array operations and mathematical functions [48].

2.4.2 SciPy

SciPy is a free and open-source Python library used for scientific computing and technical computing. It is built on top of the NumPy array object and is part of the NumPy stack which includes tools like Matplotlib, Pandas and SymPy.

Overall, the SciPy library, with its powerful and efficient computation capabilities, is a valuable tool for image processing tasks and can be effectively used in conjunction with other Python-based scientific computing tools [48].

2.4.3 OpenCV

OpenCV (Open Source Computer Vision Library) is an open-source computer vision and machine learning software library. It provides a wide range of algorithms and functions for image and video processing, including feature detection, object recognition, and image enhancement. OpenCV is written in C++ and can be used with Python and other programming languages.

One of the key advantages of OpenCV is its extensive documentation and community support. It has a large user base and a wealth of online resources, tutorials, and examples. The library is constantly updated and improved, making it a reliable choice for computer vision tasks [49].

2.4.4 scikit-image

scikit-image (skimage) is a very popular collection of algorithms for image processing in Python. It provides a wide range of functions and tools that are essential for various image processing tasks.

Here are some key features and capabilities of scikit-image:

1. **Image manipulation:** scikit-image allows for various image manipulation techniques, such as resizing, cropping, rotating, and flipping images. These operations are essential for preprocessing and preparing images for further analysis or processing;
2. **Filters and enhancements:** It provides a rich set of filters and enhancements, including noise reduction, sharpening, contrast adjustment, and histogram equalization. These functions help in improving image quality and enhancing important features for further analysis;
3. **Segmentation:** scikit-image offers advanced image segmentation techniques, such as thresholding, region-based segmentation, and watershed segmentation. These methods are crucial for extracting meaningful regions or objects from images, which can be further analyzed or classified;
4. **Feature extraction:** It provides a variety of feature extraction methods, such as edge detection, corner detection, texture analysis, and blob detection. These techniques help in identifying and quantifying important visual features in images, which can be used for various tasks like object recognition or image classification;
5. **Image registration:** scikit-image includes algorithms for image registration, which is the process of aligning images from different sources or time points. This is particularly useful in scenarios where multiple images need to be compared or combined for analysis;
6. **Morphological operations:** It offers a range of morphological operations, including dilation, erosion, opening, and closing. These operations are fundamental for shape analysis, object extraction, and noise removal in images;
7. **Visualization:** scikit-image provides tools for visualizing images and results, including image plotting, overlays, and color maps. These capabilities aid in understanding and interpreting the processed images and results.

Overall, scikit-image is a powerful and versatile library for image processing in Python, offering a wide range of functions and algorithms that are crucial for the success of the current dissertation in achieving its image processing goals [88].

2.4.5 Matplotlib

Matplotlib is a popular Python library used for creating static, animated, and interactive visualizations. It provides an object-oriented API for embedding plots into applications. Matplotlib is also a great tool for visualizing data for better understanding. It's widely used for creating bar charts, line charts, scatter plots, and many other types of diagrams. In the context of image processing and computer vision, Matplotlib is often used to visualize images, draw shapes on images, and plot graphs with a large number of data points [27].

2.5 Machine Learning and AI

Machine learning tools work in tandem with traditional image processing techniques. Understanding the relevant libraries and frameworks used for image processing is critical for the effective study of machine learning models and their implementation.

With the overview of the foundational frameworks and libraries for image processing complete, an analysis of machine learning and AI can take place, starting with a historical perspective.

2.5.1 Historical Perspective

In 1950, Alan Turing published a paper entitled "Computing Machinery and Intelligence", in which he described the now famous "Turing Test". The test, which at the time Turing gave the name of "Imitation Game", was a way of evaluating a machine's capacity to think. For Turing, if a machine could trick humans into believing they were talking to another human, when in reality they were talking to the machine, for all intents and purposes that machine was a thinking agent. Before machines can think, however, they must first learn [75].

One of the earliest examples of machine learning is the perceptron, which was developed by Frank Rosenblatt in 1958. The perceptron is a type of artificial neural network that can be trained to recognize patterns in data. It consists of a single layer of neurons that are connected to each other, and each neuron is connected to a number of input neurons. The weights of these connections are adjusted during training to improve the performance of the network. The perceptron was a significant breakthrough in the field of artificial intelligence. While the perceptron is not as commonly used today as some other machine learning techniques, it remains an important part of the history of machine learning [57].

Arthur Samuel's self-learning program, developed in the 1960s, also made a significant contribution to the development of the field. The program played checkers against itself and improved its game over time, demonstrating the potential for computers to learn

and improve themselves. Samuel's program was one of the first examples of machine learning [60].

The emergence of backpropagation, a technique for training neural networks which greatly improved their performance, was an important development in the field. It is a method for calculating the error in a neural network and adjusting the weights of the network to reduce said error. Without backpropagation, it would be much more difficult to train neural networks and to use them effectively in a wide range of applications [59].

Support Vector Machines (SVMs) were introduced in the 1990s as a powerful classification tool for machine learning. They are a type of supervised learning algorithm that can be used for classification and regression analysis. SVMs are particularly useful when dealing with high-dimensional data and can be used to identify the hyperplane that best separates the data into different classes [61].

SVMs were first introduced in a paper by Corinna Cortes and Vladimir Vapnik in 1995. The paper, entitled "Support-Vector Networks", presented the concept of SVMs as a new approach to machine learning. It has since become one of the most widely cited papers in the field [15].

IBM's Watson computer system made headlines when in 2011 it won a game of *Jeopardy!* against two human champions. Watson was able to answer complex questions in natural language using a combination of machine learning, natural language processing, and other techniques. The development of Watson demonstrated the potential for machine learning to be applied in a wide range of fields, and it helped to raise public awareness of the possibilities of this technology [17].

In 2015, Geoffrey Hinton, Yoshua Bengio, and Yann LeCun published a paper entitled "Deep Learning". Deep learning makes use of several layers of neural networks to accomplish the most varied tasks. This approach has proven to be a powerful tool in a wide range of applications, including computer vision, speech recognition, and natural language processing [39].

Another historical achievement happened in 2015, when Google's AlphaGo program defeated Lee Sedol, a world champion Go player, in a five-game match. AlphaGo was able to defeat Sedol using a combination of machine learning and deep neural networks. Much like Watson, the development of AlphaGo demonstrated the potential for machine learning to be applied in games and other complex domains [65].

Although there has been immense progress in the field of artificial intelligence since Turing, none of the machines that have been developed so far could pass the Turing Test. However, the release of ChatGPT-3 in 2021 made some people think that the "imitation game" could finally be beaten.

ChatGPT-3 is a large language model developed by OpenAI. The model is one of the most advanced language models ever created, with the ability to generate human-like text and carry out a wide range of natural language processing tasks. The model is based on a deep neural network architecture known as a transformer, which was first introduced in a paper by Vaswani et al. in 2017. The transformer architecture has become increasingly

popular in the field of natural language processing because of its ability to process large amounts of text and generate high-quality language output [55, 86].

One of the key features of Chat GPT-3 is its ability to perform a wide range of language tasks, including language translation, text completion, and question answering, among others. The model is capable of generating coherent and natural-sounding text, and it has the ability to adapt to different tasks and contexts [3].

Despite ChatGPT-3's impressive performance, it could not yet pass the Turing Test. Although it can generate human-like text, it does not truly understand the meaning of the it's own words. This means that while it can answer questions and generate text that sounds like it was written by a human, it is not truly capable of thinking [3].

Thanks to ChatGPT, there has been increasing development and research in the field of machine learning and AI, probably more than ever. The amount of programs and tools that are being released is unprecedented and their ability to change the future is immense. A particular field that has seen an increasing development is computer vision.

Computer vision is a subfield of artificial intelligence that focuses on enabling machines to interpret and understand visual information from the world around them. Rosenblatt's perceptron was one of the earliest examples of computer vision, since its goal was to categorize simple shapes in images [57].

In the 1970s and 1980s, researchers began to develop more sophisticated computer vision algorithms that could recognize more complex patterns in images. These algorithms were based on a range of techniques, including template matching, feature extraction, and machine learning. With the development of faster computers and more powerful algorithms, computer vision began to be applied in a wide range of applications, from industrial automation to medical imaging [7].

One of the most significant breakthroughs in computer vision came in the late 1990s, with the development of the Viola-Jones algorithm for face detection. This algorithm used a technique called "Haar features" to detect faces in images, and it was able to achieve a high level of accuracy in real-time applications [87].

In recent years, deep learning has emerged as a powerful tool in computer vision, enabling machines to recognize and classify images with a high degree of accuracy. Deep learning algorithms are capable of learning complex patterns and relationships in data [39].

More recently, computer vision tools like Midjourney, Stable Diffusion and DALL-E 2 are poised to revolutionize the art field. These tools generate art based on a prompt given by the user and their resemblance to reality is becoming more and more accurate every day [9].

This historical perspective provides important context for understanding the development of machine learning and AI in fluid mechanics. By understanding the origins of these techniques, the strengths and weaknesses of their implementation becomes more clear. With this overview complete, the study of the fundamental concepts of machine learning and AI can be carried out.

2.5.2 Fundamental Concepts

Firstly, it is important to understand the differences between artificial intelligence, machine learning and deep learning.

Artificial intelligence refers to the development of intelligent machines that can perform tasks that typically require human intelligence. It involves creating algorithms and systems that can perceive, reason, learn, and make decisions. AI encompasses a broad range of techniques and applications, including natural language processing, computer vision, robotics, and expert systems [30].

Machine learning is a subset of AI that focuses on enabling machines to learn from data and improve their performance without being explicitly programmed. It involves developing algorithms and models that can automatically learn and make predictions or decisions based on patterns and relationships in the data. Machine learning techniques include supervised learning, unsupervised learning, reinforcement learning, and semi-supervised learning [30].

Deep learning is a sub-field of machine learning that is inspired by the structure and function of the human brain. It involves building neural networks with multiple layers of interconnected nodes, known as neurons. Deep learning algorithms can automatically learn hierarchical representations of data, extracting increasingly abstract and complex features [30].

2.5.2.1 Types of Machine Learning

There are several types of machine learning that can be used for different purposes, and their implementation largely depends on the availability of labeled data.

Supervised Learning is a machine learning approach where an algorithm is trained using labeled data to make predictions or classifications on new, unseen data. The algorithm learns from a set of training examples with known labels to infer a function that can predict or classify output variables. In supervised learning, the input dataset is divided into training and testing/validation sets, and the algorithm uses the training set to learn the relationship between input and output variables [35].

On the other hand, with **Unsupervised Learning**, the algorithm is trained to find patterns or relationships on unlabeled data. Unlike supervised learning, unsupervised learning does not require labeled examples for training. Instead, the algorithm explores the data on its own to discover hidden structures or clusters. This approach is particularly useful when the desired output or target variable is unknown or when the dataset is too large or complex to label manually [13].

Semi-Supervised Learning is a mixture of both. With this approach both labeled and unlabeled data are used to train the models. This is particularly useful when labeled data is scarce or expensive to obtain, as it allows leveraging the abundance of unlabeled data to improve model performance [13].

Lastly, **Reinforcement Learning** is a type of machine learning where an algorithm learns to make decisions by interacting with an environment. The algorithm receives feedback in the form of rewards or punishments based on its actions, and its goal is to maximize the cumulative reward over time. Unlike supervised learning, where the algorithm is provided with labeled examples, reinforcement learning relies on trial and error to discover the optimal actions in a given environment [73]. Arthur Samuel's self learning program was a type of reinforcement learning.

2.5.2.2 Data Handling

No matter the type of machine learning being employed, the data will have to be handled one way or another. For unsupervised learning and reinforcement learning, the manual handling of data is minimal since the algorithm often identifies patterns from raw or lightly processed data.

For supervised learning, however, the first steps of every model will almost always require some type **Feature Engineering**. This is a crucial step in the machine learning pipeline that involves selecting or transforming variables to improve the input of a model. It is the process of creating new features or modifying existing ones to enhance the performance and predictive power of an algorithm. Feature engineering plays a vital role in extracting meaningful information from raw data and capturing relevant patterns and relationships that may not be apparent initially [35].

Since supervised learning works on labeled data, creating a **Training/Validation Split** is important in order to evaluate the performance and generalization ability of the model on unseen data. To achieve this, a common practice is to divide the available dataset into two subsets: the training set and the validation set. The training set is used to train the model, while the validation set is used to assess the model's performance [24].

2.5.2.3 Model Training and Evaluation

With the data for the model prepared, the next crucial step is the training of the model. The training will be determined by the initial parameters defined, which are going to be updated during training.

These model's parameters are commonly adjusted by the **Gradient Descent** optimization algorithm. This method iteratively updates the parameters in the direction of the steepest descent of the loss function. This allows the model to find the optimal set of parameters that minimize the difference between predicted and actual values [59].

Aside from the parameters, which are updated during training, the hyperparameters of a model will also play a role in the training process. Hyperparameters are external configurations set before the training begins, and remain constant throughout the whole process.

One key hyperparameter is the **Learning Rate**, which determines the step size in the optimization process during training. It controls how much the model's parameters are adjusted in each iteration of the optimization algorithm. A high learning rate may lead to faster convergence but risk overshooting the optimal solution, while a low learning rate may result in slow convergence or getting stuck in local minima. Choosing an appropriate learning rate is crucial for training machine learning models effectively [69].

Additional hyperparameters include **batches** and **epochs**. Batches are smaller subsets of the training data, while epochs are to the number of times the entire training dataset is passed through the model. Each epoch consists of multiple iterations, where each batch is used to update the model's parameters. The batch size determines how many samples are processed in each iteration. Larger batch sizes can lead to faster training but require more memory, while smaller batch sizes may result in slower training but better generalization [12].

Another component of the model that affects the training is the **Loss/Cost Function**, which quantifies the discrepancy between predicted and actual values in a machine learning model. It represents the object that the model aims to minimize during training. Different types of loss functions may be used depending on the problem at hand, such as Mean Squared Error (MSE) for regression tasks or Cross-Entropy Loss for classification tasks. The choice of an appropriate loss function is crucial for guiding the model towards the desired outcome [8].

After the model is trained, a **Model Evaluation** must take place to assess its performance. This process involves applying the model to a separate test dataset and comparing its predictions to the actual values. Various evaluation metrics can be used, depending on the nature of the problem, such as accuracy, precision, recall, F1-score and MSE. Proper model evaluation ensures that the model's performance is reliable and can be generalized to unseen data [71].

2.5.2.4 Handling Model Complexity

When building a machine learning model, it's also important to keep in mind the **bias-variance tradeoff**. This fundamental concept involves finding the right balance between model flexibility and generalization. A model with high flexibility, such as a complex neural network with many parameters, can fit the training data very well, but it may not generalize well to unseen data. This problem is called **overfitting**. This is in contrast with **underfitting**, which happens when a model with low flexibility, such as a simple linear regression model, may have low variance but may not capture the complexity of the underlying data. It performs poorly both on training and test set. The goal is to find a model that minimizes both bias and variance, leading to good performance on both the training and validation data. Understanding and managing the bias-variance tradeoff is crucial for developing robust machine learning models [19].

This balance between bias and variance is especially important when working with neural networks, as these models are both highly flexible and powerful.

2.6 Neural Networks

Neural Networks (NNs) are models composed of interconnected nodes or "neurons". These neurons are organized into layers, with each layer responsible for performing specific computations. Input layers receive initial data or features, hidden layers perform weighed calculations and activations to capture patterns and relationships, and output layers produce the final prediction. Fig. 2.6 shows the basic architecture of a neural network [77].

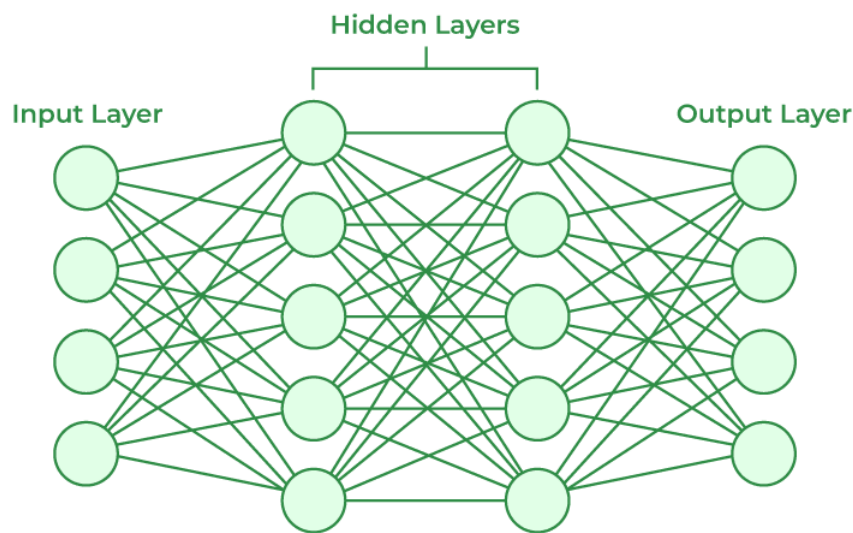


Figure 2.6: Neural network architecture, with respective input, hidden and output layers identified [77]

Neural networks have shown great success in various fields, including computer vision, natural language processing, and fluid mechanics. They have been applied to tasks such as image recognition, object detection, and flow prediction [7, 73, 8]

Activation functions are a crucial component of neural networks. By introducing non-linearity to the model, they allow them to establish complex relationships between inputs and outputs. Common activation functions include Rectified Linear Unit (ReLU), sigmoid, and tanh. Activation functions play a crucial role in determining the network's ability to learn and generalize from the data [73, 8].

As neural networks grow in complexity and depth, they evolve into deep neural networks, which are capable of modeling even more intricate patterns and representations in the data.

2.6.1 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are specialized deep neural networks designed for processing image data and are inspired by the visual cortex of animals, which is known for its ability to process visual information effectively. They consist of convolutional layers, pooling layers, and fully connected layers. Convolutional layers apply filters to input images, extracting features at different spatial scales. Pooling layers reduce the dimensionality of feature maps, preserving the most important information. Fully connected layers learn high-level representations from the extracted features. CNNs have revolutionized computer vision tasks such as image classification, object detection, and image segmentation [7, 40].

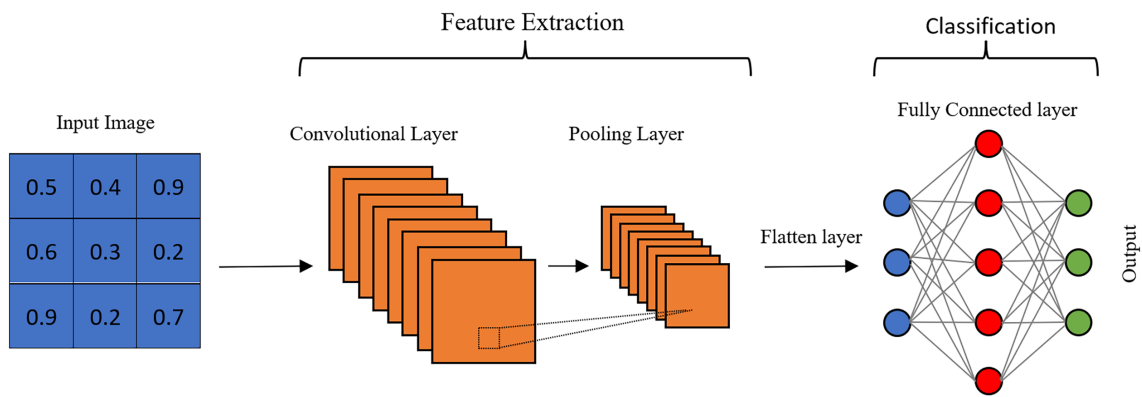


Figure 2.7: Example of convolutional neural network architecture for classification [4]

One of the key advantages of CNNs over traditional machine learning models is their ability to learn hierarchical representations of visual data. Given the right network structure and training data, CNNs can automatically learn to engineer the features they need to solve complex visual tasks. This eliminates the need for manual feature extraction, making CNNs more flexible and adaptable to different domains and datasets.

CNNs have achieved remarkable performance in various computer vision challenges, surpassing human-level accuracy in tasks such as image classification. They have also been instrumental in advancing the field of deep learning and enabling breakthroughs in areas like autonomous driving, medical image analysis, and facial recognition.

Krizhevsky et al introduced the concept of CNNs and demonstrated their effectiveness in large-scale image classification tasks. Their seminal paper showcases the architecture known as AlexNet, which achieved a significant improvement in performance on the ImageNet dataset and paved the way for subsequent advancements in the field [40, 36].

2.6.2 Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) are another type of deep neural network architecture specifically designed to handle sequential data, where the order of inputs is crucial. Unlike traditional feedforward neural networks, RNNs incorporate feedback connections that

allow information from previous steps in the sequence to influence the current step. This unique feature enables RRNs to capture and model temporal dependencies, making them exceptionally well-suited for tasks involving time-series or sequential data.

One of the most significant advantages of RRNs is their ability to process variable-length sequences. This flexibility is particularly beneficial in applications such as speech recognition, where the length of spoken phrases can vary significantly, or in language modeling, where sentences and paragraphs differ in length.

Despite their strengths, traditional RRNs can struggle with long-term dependencies due to issues like vanishing and exploding gradients. To address these challenges, advanced variants such as Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs) have been developed. These architectures include mechanisms to maintain and update information over long sequences, further enhancing the capabilities of RRNs.

Overall, RRNs and their advanced variants have become indispensable tools in the field of machine learning, enabling significant advancements in understanding and processing sequential data [73, 8, 29].

2.6.3 Generative Adversarial Networks (GANs)

Generative Adversarial Networks (GANs) are a class of deep learning models that are used for generating synthetic data, particularly images. GANs consist of two main components: a generator and a discriminator. The generator network learns to generate realistic-looking images, while the discriminator network learns to distinguish between real and generated images.

The generator network takes random noise as input and transforms it into an image that resembles the training data. It learns to create images that are indistinguishable from real images by minimizing the difference between the generated images and the real images. The discriminator network, on the other hand, learns to classify whether an image is real or generated. It is trained to maximize its accuracy in distinguishing between the two types of images.

During training, the generator and discriminator networks are trained in an adversarial manner. The generator tries to fool the discriminator by generating images that are realistic enough to be classified as real, while the discriminator tries to correctly classify real and generated images. This competition between the two networks leads to the improvement of both networks over time.

By utilizing GANs, researchers and practitioners can generate high-quality synthetic images that can be used for various applications, such as data augmentation, data synthesis, and creative image generation [22, 50].

2.7 Deep Learning Frameworks

Deep neural networks are an integral part of machine learning and specific deep learning frameworks are used for building and training these models.

2.7.1 TensorFlow

TensorFlow is an open-source machine learning framework developed by Google that has gained popularity for its ability to build and train neural network models, including CNNs and RNNs.

TensorFlow provides a flexible and comprehensive ecosystem for deep learning, allowing researchers and developers to efficiently design, implement, and train complex neural network architectures. It offers a rich set of functionalities, including automatic differentiation, model deployment, and distributed computing, making it suitable for both research and production-level projects.

The framework's popularity and extensive community support have led to the development of numerous resources, tutorials, and pre-trained models that can be leveraged to accelerate the development process. TensorFlow's documentation is extensive and regularly updated, providing detailed explanations and examples for various use cases.

By leveraging TensorFlow's capabilities, researchers and practitioners can build and train state of the art neural network models for various computer vision tasks [1, 51].

2.7.2 PyTorch

PyTorch is an open-source machine learning library developed by Facebook that has gained popularity for its flexibility and dynamic computational graph.

PyTorch distinguishes itself from other deep learning frameworks by its dynamic nature, allowing users to define and modify computational graphs on-the-fly. This flexibility makes PyTorch well-suited for research and experimentation, as it enables rapid prototyping and easy debugging of neural network architectures.

One of the key advantages of PyTorch is its intuitive and Pythonic interface, which makes it accessible to both beginners and experienced deep learning practitioners. It offers a high-level API that simplifies the process of building and training neural networks, while also providing low-level access for advanced customization [52].

2.7.3 Keras

Keras is a high-level neural networks API written in Python that is capable of running on top of TensorFlow, CNTK, or Theano. It provides a user-friendly interface for building and training neural network models, making it easier for researchers and developers to implement deep learning architectures.

With Keras, users can quickly prototype and experiment with different network structures, such as CNNs, RNNs, and GANs. Keras abstracts away the complexities of low-level implementation details, allowing users to focus on designing and fine-tuning their models.

One of the key advantages of Keras is its compatibility with multiple backend frameworks. Users have the flexibility to choose between TensorFlow, CNTK, or Theano as the underlying computational engine. This enables seamless integration with existing libraries and tools built on these frameworks.

Keras has gained widespread adoption in the deep learning community due to its simplicity and ease of use. It has a large and active user community, which contributes to its continuous development and improvement. The Keras documentation provides comprehensive resources and examples, making it accessible to beginners and experts alike [14].

2.8 Machine Learning Applications in Fluid Mechanics

The previous sub-chapters clearly highlight the potential of machine learning and deep learning to revolutionize various domains, including fluid mechanics. They have emerged as powerful tools in the field, enabling researchers to gain new insights and improve the accuracy of results and efficiency of simulations.

One area where machine learning has been applied is in the prediction and modeling of turbulent flows. Turbulent flows are highly complex and difficult to simulate accurately using traditional Computational Fluid Dynamics (CFD) methods. In [41], a special network architecture called Tensor Basis Neural Network (TBNN) was used as a deep learning approach to Reynolds-averaged Navier–Stokes (RANS) turbulence modeling.

Another application of machine learning in fluid dynamics is in the optimization of designs. Machine learning algorithms can be used to automatically search and optimize design parameters, leading to improved performance and efficiency. This can be particularly useful in areas such as aerodynamic design, where small changes in design parameters can have a significant impact on performance [53].

Machine learning has also been used in the analysis and interpretation of experimental data. By training machine learning models on large datasets of experimental measurements, researchers can develop models that can accurately predict and analyze complex physical phenomena. This can help in the understanding of flow behavior and guide the design of experiments [13].

One other example of a machine learning application in fluid dynamics is the use of Support Vector Machines (SVMs) for flow classification and pattern recognition. SVMs have been used to classify different flow regimes based on input features, such as velocity profiles or pressure distributions. This can help in the identification of flow characteristics and improve the understanding of complex flow phenomena [13].

Another example is the use of CNNs for image-based flow analysis. CNNs have been used to analyze flow visualization images and extract relevant flow features. This can help in the identification of flow structures and the understanding of flow behavior [38].

These are just a few examples of the many applications of machine learning in fluid mechanics. The field is rapidly evolving, with new techniques and approaches being developed continuously. As computing power and data availability continue to increase, machine learning is likely to play an even more significant role in the future of fluid dynamics research.

EXPERIMENTAL TESTS

3.1 Introduction

As was previously mentioned, the experimental tests of the lock-exchange method which is going to be studied in this dissertation were done by Alves [5].

This chapter aims at describing the setup and procedure of these experimental tests, including the image acquisition apparatus. It's important to pay special attention to any aspect of the procedure that might have an influence on the chosen computer vision model, particularly the calibration methods used for image acquisition.

3.2 Experiment Description

The purpose of the experiment was to test the lock-exchange method on a rectangular channel made of transparent glass, with the aim of measuring density differences using a mixture of ethanol and salt. The channel was supported by a steel structure with weights to ensure stability, and the reduced gravity targets were $g' = 0.09 \text{ m/s}^2$ and $g' = 0.36 \text{ m/s}^2$. For the former, 93 grams of salt and 1.8 liters of ethanol were required, while the latter required 357 grams of salt and 8.8 liters of ethanol.

A gate made of steel was positioned 0.3 meters from the left end of the channel and a 3 meter ruler was placed at its bottom. The first half of the channel contained an acrylic plate with an array configuration of three rows of thirty-one bulk cylinders each side by side, while the second half had a flat acrylic plate. The cylinders had a diameter of 0.2 meters and were separated by 0.25 meters from each other, as well as from each line of cylinders.

Alves' experiments differ from the typical setup, where the current has an unobstructed area to develop before reaching the part of the channel with obstacles. In this case, the current flows through an obstructed area first before reaching the flat part of the channel.

Figures 3.1 and 3.2 represent the channel configuration and a zoom of the lock area, respectively.

After cleaning the channel, the lock gate was placed and sealed with Vaseline to prevent

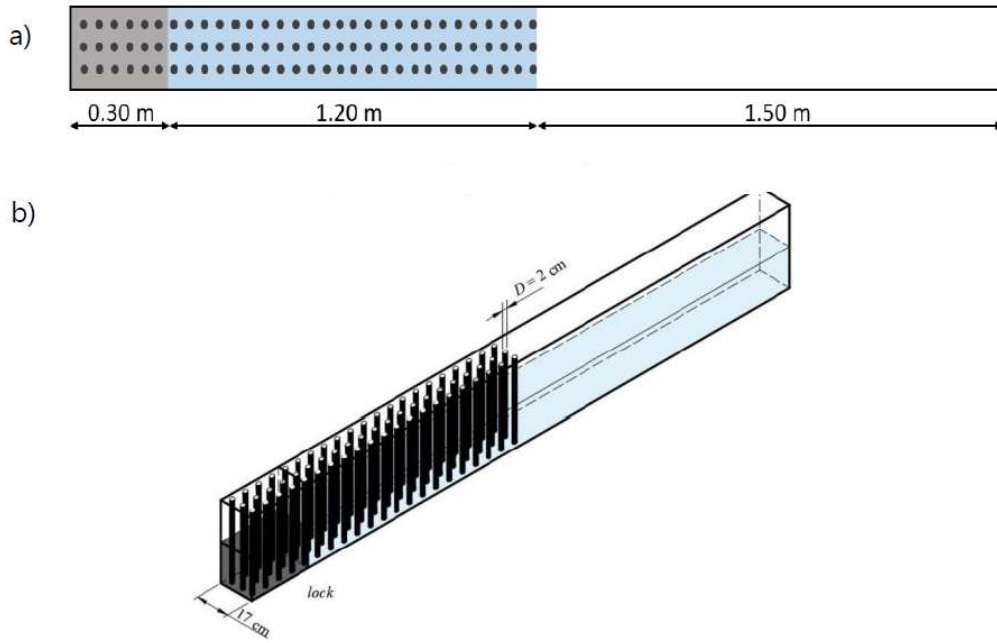


Figure 3.1: Configuration of the channel: a) top view and b) perspective view [5]

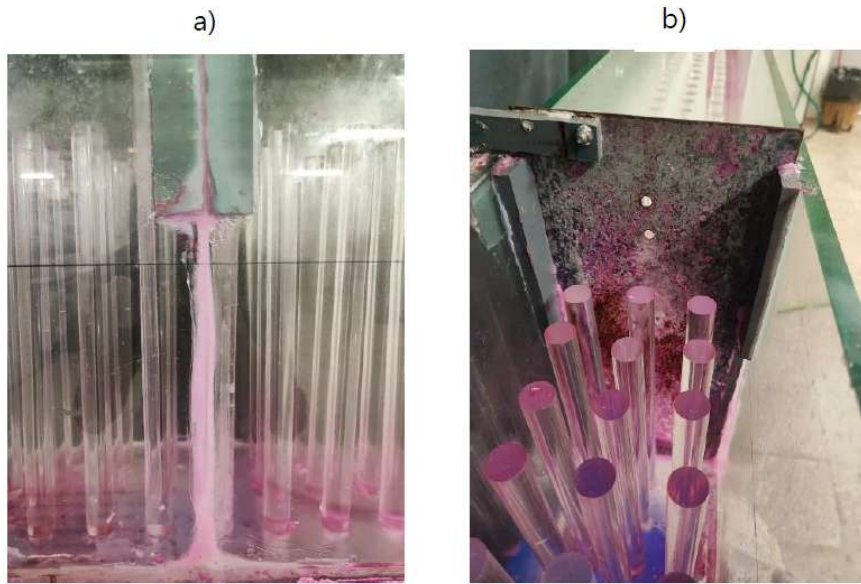


Figure 3.2: Lock zone: a) vaseline filling the gap between lock and channel wall and b) gate positioned [5]

leakage. The denser fluid, water and salt, and the ambient fluid, water and alcohol, were mixed in a proportion previously studied to achieve the target reduced gravity.

The mixture of water and alcohol for the ambient fluid was chosen because according to the literature it would be the best for refractive index matching. The refractive index is a physical property of materials that describes how much light is bent or refracted when it passes through a substance. For the purposes of this experiment it was important to have

refractive index matching in order for the image acquisition and processing to be done correctly [23, 16].

Both channel parts were filled simultaneously, ensuring similar fluid height on both sides to balance hydrostatic pressure, preventing the gate from shifting and causing the mixing of fluids ahead of time.

At a fluid height of 0.2 meters, the density of the denser fluid was measured in one point while the density of the ambient fluid was measured in three points to guarantee a good mixture. The fluids' temperature was also measured and the values of reduced gravity were validated.

Finally, the lights were turned off, image recording started, and the gate was removed to initiate the test. Figure 3.3 shows a flowchart of the experiment procedure.

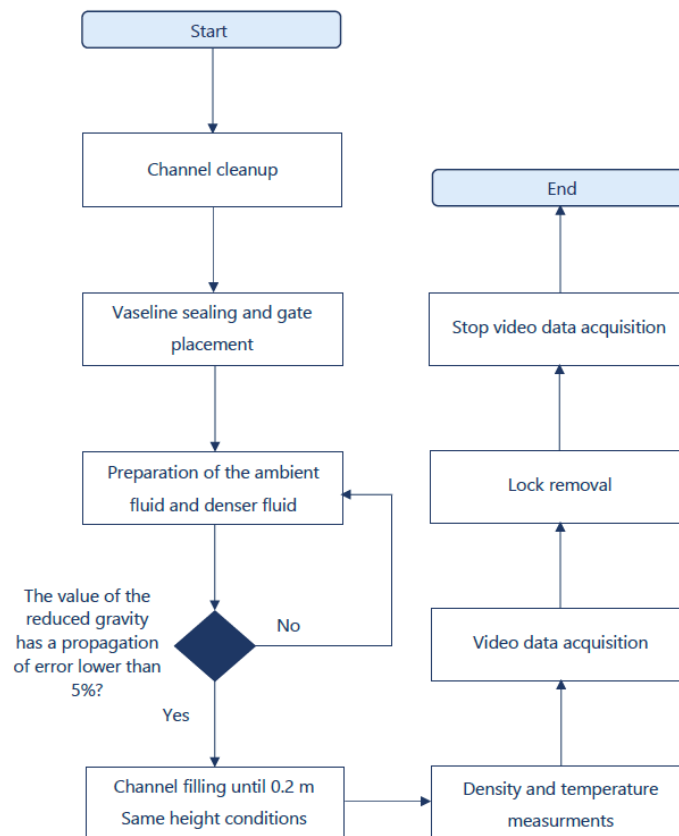


Figure 3.3: Experiment Flowchart [5]

3.3 Image Analysis Technique

The high-speed Allied Bonito CL-400B/C camera was used to capture images for the experiment. It had a high-definition image quality and was connected to the StreamPix software, which saved the sequence of images taken by the camera in a ".seq" format. Each image was saved in a ".bmp" format at a rate of 44 Hz. The camera had a frontal section of $800 \times 70 \text{ mm}^2$ and a width of 35 millimeters. Figures 3.4 and 3.5 show the mounted camera and the images as seen in the StreamPix software



Figure 3.4: Camera Placement [5]

The camera was fixed against the wall perpendicular to the side of the channel so that it would capture a side view of it. Since the camera couldn't capture the entire 3 meter channel, it was placed in two different positions: one that captured from 0 to 1.55 m and one that captured from 1.4 m to 2.95 m. This made it so there was an intersection area between the two acquisitions where both results were subsequently overlapped. For the purposes of this dissertation only the images taken from the first half of the channel were considered.

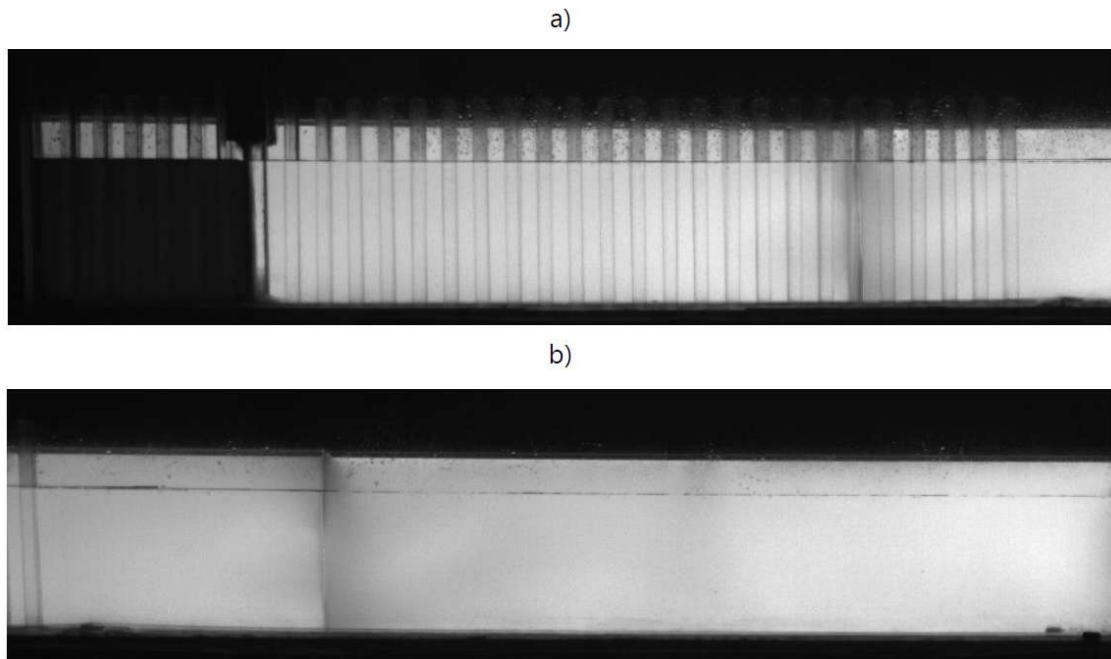


Figure 3.5: Raw images from StreamPix software: a) first half of the channel, lock view; b) second half of the channel [5]

The goal with the image acquisition process is to transform the captured images into density fields. In order to achieve this, a calibration process must be done before every experiment trial.

The camera that was used operates in a greyscale color ranging where the value 0 is the darkest shade of black and the value 255 is the lightest shade of white. In order to distinguish the fluids, the colorant rhodamine was used in the denser fluid. The darkest shade that can be reached by this colorant is a dark red, and the lighter shade is a soft pink. These values translate to a certain value of density and are influenced by the aperture of the lens, the exposure time of the camera, and the concentration of rhodamine.

Since the goal is to essentially transform RGB values into density values, the calibration process consisted in filling the channel with denser fluid, assigning a known value of density to a specified RGB value, then progressively adding lighter fluid in order to dilute the rhodamine, and map other known values of density to specified RGB values.

By doing this, there would be a correspondence between certain RGB values and the density values that they represent. This is what enabled the MATLAB program to translate images into density fields.

A geometric calibration that involved positioning and distortion mitigation was also done. When calibrating a camera for 3D reconstruction, it is important to consider the camera's geometric properties, including lens distortion. A patterned acrylic plate was used to map rays and image points, and the camera's external and internal parameters were determined, along with distortion coefficients, by taking images of the plate in different positions.

3.4 Density Calibration

The density calibration process is essential for converting greyscale images to density fields. As was previously stated, in the experiment, a series of pictures of the channel were taken with different concentrations of rhodamine. The calibration was essentially done pixel by pixel through controlled dilution, starting with the densest fluid and diluting to ambient fluid density, ensuring consistent camera conditions throughout.

Figure 3.6 shows the color shades during the calibration performed during the experiment.

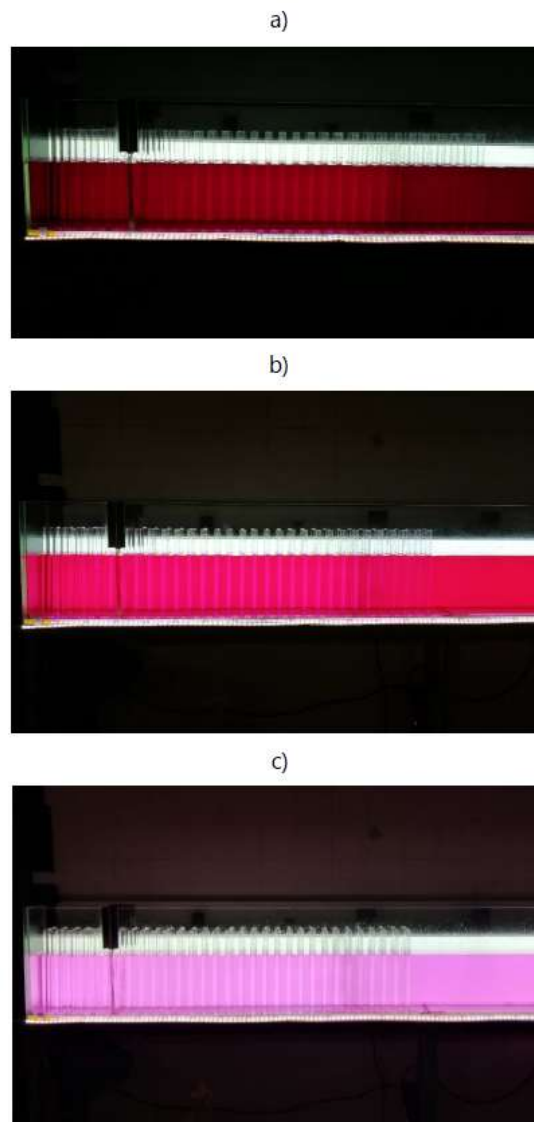


Figure 3.6: Calibration steps and successive decrease of the density in the channel from a darker shade and denser fluid to a lighter shade and least dense one: a), b), c), respectively [5]

Figure 3.7 shows the sequence of calibration images obtained.



Figure 3.7: Sequence of Calibration Images [5]

A total of fourteen calibration images were taken, and for each of these images, a pixel was chosen in order to establish a correspondence between RGB and density values before inserting the data in to a software. Thus, fourteen data points were created, and Figure 3.8 shows the calibration curve obtained after the reduced gravity of $g' = 0.36 \text{ m/s}^2$, which will be the one studied in this dissertation. This calibration curve will later be compared with the results obtained in the implemented Python code.

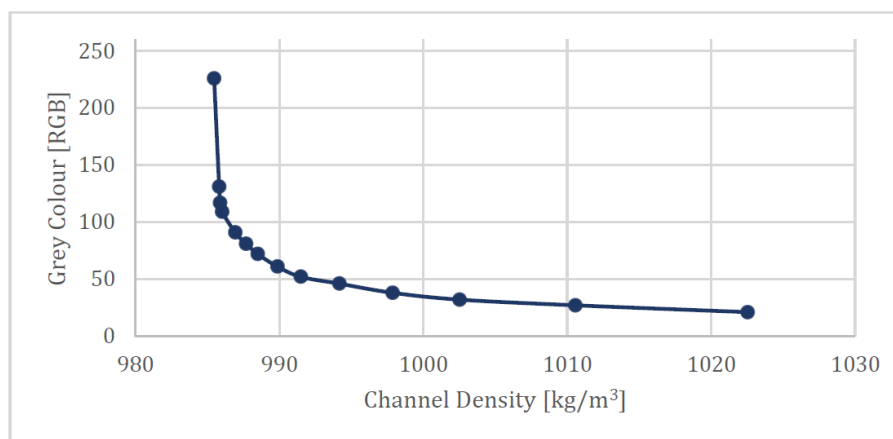


Figure 3.8: Calibration Curve [5]

IMPLEMENTED PYTHON CODE

4.1 Introduction

The present chapter aims at describing the Python model that was implemented in order to undistort the images gathered in the experiment described in Chapter 3, and their subsequent conversion into density fields.

To accomplish these tasks, two major Python codes were developed:

- Distortion Correction
- Density Estimation

As was previously stated the density estimation code was created based on the MATLAB code used by Alves.

For each of the codes, a flowchart was created along with a description of the libraries, tools and programming techniques that were used.

4.2 Distortion Correction

4.2.1 Distortion Correction - Flowchart

The first step of the programming pipeline consists in undistorting the images taken from the lock-exchange experiment done by Alves. For this, the chessboard camera calibration technique was used.

This technique is a method used to correct lens distortion in images. The process involves taking multiple pictures of a flat chessboard pattern at different angles and using these images to compute the distortion of the camera lens. Once the distortion is known, it can be corrected in subsequent images taken by the same camera. This process is critical in many computer vision tasks where the precise location of objects in an image is important [78].

The full code is available in Appendix A for detailed reference. The following steps give a high level overview of what the code does:

1. Import necessary libraries and modules;
2. Define the path to the folder where the calibration images (chessboard pattern pictures) are stored;
3. Define the size of the chessboard used for calibration;
4. Create arrays to store object points and image points;
5. Prepare the object points using *numpy* functions;
6. Use *glob* to get all file names that match the pattern of the calibration images;
7. Loop through all calibration images. Convert each image to grayscale and find the chessboard corners. If corners are found, refine them and add the object points and image points to the respective arrays;
8. After all corners are detected, calibrate the camera using *cv2*'s *calibrateCamera* function which returns the camera matrix and distortion coefficients;
9. Set the distortion coefficients $k1$, $k2$, and $k3$ to zero;
10. Display the updated camera matrix and distortion coefficients;
11. Use the camera matrix and distortion coefficients to undistort the images;
12. Display the original and undistorted images.

Fig. 4.1 shows the flowchart for the Distortion Correction code:

A detailed description of the code, with relevant code snippets can be found in the following sub-chapters.

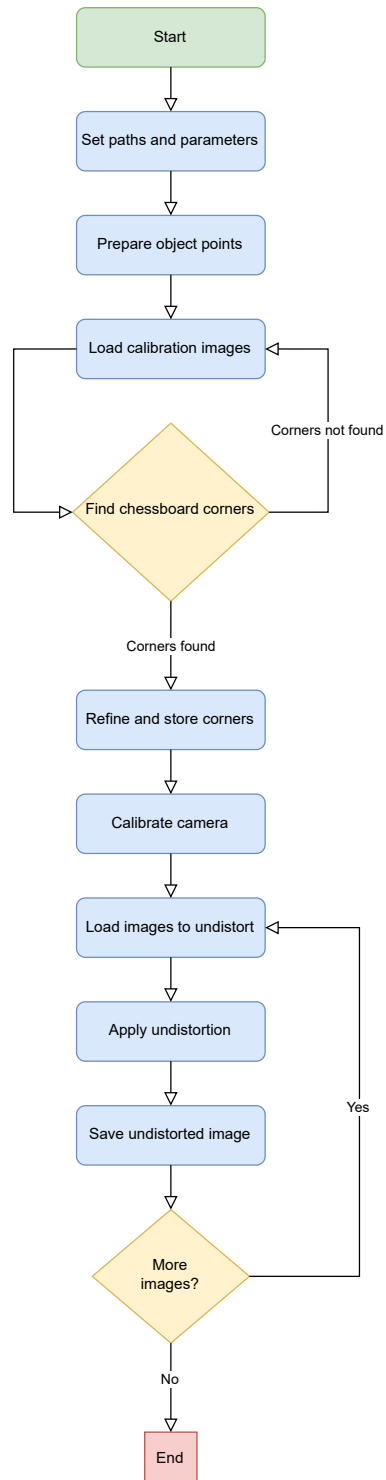


Figure 4.1: Distortion Correction Flowchart

4.2.2 Distortion Correction - Detailed Description

The goal of this code is to find the camera matrix and the distortion coefficients, which are presented in Eq. (4.1) and (4.2), respectively. This will be done through the calibration process in order to later undistort the distorted images.

$$\mathbf{K} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \quad (4.1)$$

$$\mathbf{D} = [k_1 \quad k_2 \quad p_1 \quad p_2 \quad k_3] \quad (4.2)$$

Where f_x and f_y are the focal lengths expressed in pixel units, and c_x and c_y represent a principal point that is usually at the center of the image. Both of these parameters are intrinsic to the camera. k_1 , k_2 and k_3 are the radial distortion coefficients and p_1 and p_2 are the tangential distortion coefficients [79].

Fig. 4.2 illustrates the pinhole camera model, which is used to obtain these equations:

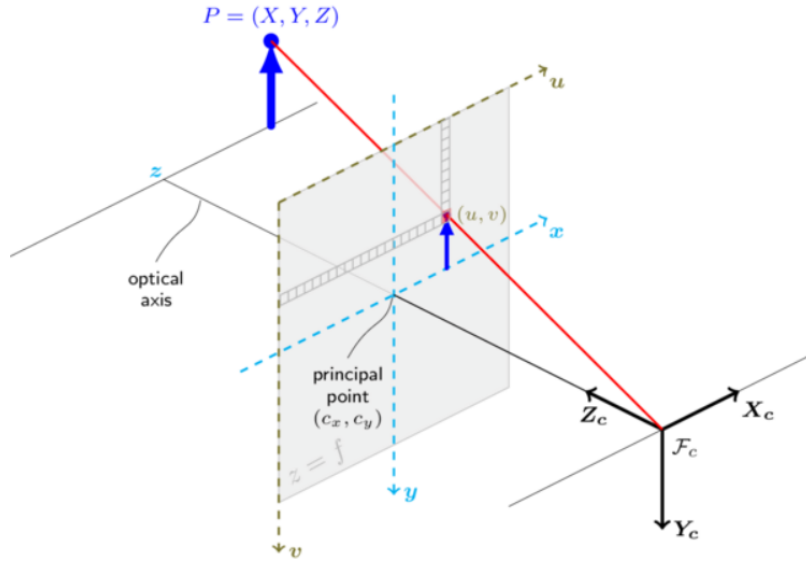


Figure 4.2: Pinhole camera model [79]

Radial distortion is a type of distortion that causes straight lines to appear curved in images. It occurs when light rays bend more near the edges of a lens than they do at the optical center.

Tangential distortion occurs when the lens and the image plane are not perfectly parallel. This type of distortion causes straight lines to appear as if they are bent or slanted towards one side of the image [78].

The code starts by importing the required libraries and modules (Listing 1).

```
import os
import cv2
import numpy as np
import glob
```

Listing 1: Distortion Correction - Libraries

In this case, only three modules and one library is needed:

- *cv2* is the primary module of the OpenCV library and includes a wide range of functionalities for image and video processing, computer vision tasks, and machine learning algorithms;
- *numpy* is a Python library which was described in the state of the art review;
- *glob* is a module that finds all the pathnames matching a specified pattern [83];
- *os* is a module that provides a way of using operating system dependent functionality [84].

Next, three file paths are defined (Listing 2).

- *calibration_pattern* holds the folder with the chessboard patterns;
- *images_to_undistort* holds the folder with the images to undistort;
- *save_directory* holds the folder where the undistorted images are to be stored;

```
# Path to calibration images
calibration_pattern = 'path_to_calibration_images*.bmp'

# Path to images to undistort
images_to_undistort_pattern = 'path_to_images_to_undistort*.bmp'

# Path to save undistorted images
save_directory = 'path_to_save_undistorted_images'
```

Listing 2: Distortion Correction - Path to calibration images

For the calibration function to work properly, the size of the chessboard pattern also has to be explicitly defined. This is done by indicating the number of inner corners in the chessboard pattern. The pictures of the pattern had been previously taken during Alves' experiment, in order to perform the MATLAB camera calibration for the experiment. As can be seen in Figure 4.3, the number of inner corners in the pattern are:

- Vertical: 9
- Horizontal: 78

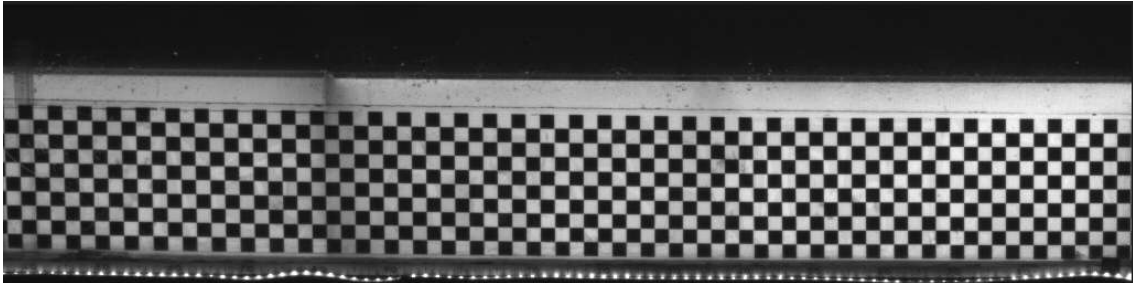


Figure 4.3: Chessboard pattern used for camera calibration [5]

Thus, the size of the chessboard pattern is defined in the Python code as follows (Listing 3).

```
# Chessboard size (number of inner corners)
chessboard_size = (9, 78)
```

Listing 3: Distortion Correction - Chessboard size

The following step is the creation of arrays to store object points and image points. Object points are 3D points in real-world space and image points are 2D points in the image plane (Listing 4).

```
# Arrays to store object points and image points
obj_points = [] # 3D points in real-world space
img_points = [] # 2D points in image plane
```

Listing 4: Distortion Correction - Arrays to store object points and image points

These lists will be populated with points for each calibration image processed.

The next section of the code creates an array that holds the fixed 3D coordinates of the chessboard corners. These coordinates are the same for every calibration image since they represent the actual physical layout of the chessboard (Listing 5).

The *numpy* function *np.zeros()* is used to create an array of zeros with the shape (*chessboard_size[0] * chessboard_size[1], 3*). In this case the shape will be (9 * 78, 3).

The *objp[:, :2] = np.mgrid[0:chessboard_size[0], 0:chessboard_size[1]].T.reshape(-1, 2)* line of code generates a grid of coordinates for the chessboard corners and reshapes the grid to a 2-column array where the first column represents x-coordinates and the second column represents y-coordinates of the corners. The z-coordinates are kept as zeros as the chessboard is planar.

```
# Prepare object points, like (0,0,0), (1,0,0), (2,0,0), ..., (8,5,0)
objp = np.zeros((chessboard_size[0] * chessboard_size[1], 3), np.float32)
objp[:, :2] = np.mgrid[0:chessboard_size[0],
↪ 0:chessboard_size[1]].T.reshape(-1, 2)
```

Listing 5: Distortion Correction - Prepare object points

After this operation, *objp* contains the 3D coordinates of the chessboard corners, with each point having coordinates $(x, y, 0)$. This array is then used to represent the real-world positions of the corners in each calibration image.

Afterwards, the *glob* module is used to get all file names that match the pattern of the calibration images (Listing 6).

```
# Glob to get all file names that match the pattern
calibration_images = glob.glob(calibration_pattern)
```

Listing 6: Distortion Correction - Glob to get all file names that match the pattern

This is important for processing all the calibration images in the defined folder, without having to specify each image individually.

The code then loops through the calibration images in order to obtain the parameters for the camera calibration (Listing 7).

For each calibration image:

- The image is read and converted to grayscale;
- *cv2.findChessboardCorners* is used to detect the 2D coordinates of the chessboard corners in the image;
- If corners are found:
 - The corners are refined;
 - *objp* which contains the fixed 3D coordinates of the chessboard corners, is appended to *img_points*;
 - The refined 2D coordinates (*corners*) are appended to *img_points*.

Refining the corners of the detected chessboard pattern improves the accuracy of the calibration and yields better results. The *criteria* defines when the iterative algorithm should stop refining the corner positions. In this case, the *criteria* are set to stop the algorithm if either 30 iterations have been completed or the corner positions change by less than 0.001 between iterations [81]:

- *cv2.TERM_CRITERIA_EPS* - stop the algorithm iteration if specified accuracy, *epsilon*, is reached;

```
# Loop through calibration images
for calibration_image_path in calibration_images:
    calibration_image = cv2.imread(calibration_image_path)
    gray_calibration = cv2.cvtColor(calibration_image,
    ↪ cv2.COLOR_BGR2GRAY)

    # Find chessboard corners and refine them
    ret, corners = cv2.findChessboardCorners(gray_calibration,
    ↪ chessboard_size, None)
    if ret:
        # Refine corners
        criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER,
        ↪ 30, 0.001)
        corners = cv2.cornerSubPix(gray_calibration, corners, (11, 11),
        ↪ (-1, -1), criteria)

        obj_points.append(objp)
        img_points.append(corners)

    cv2.drawChessboardCorners(calibration_image, chessboard_size,
    ↪ corners, ret)
    cv2.imshow('Chessboard Corners', calibration_image)
    cv2.waitKey(500) # Wait for a short duration to display each
    ↪ image

cv2.destroyAllWindows()
```

Listing 7: Distortion Correction - Loop through calibration images

- `cv2.TERM_CRITERIA_MAX_ITER` - stop the algorithm after the specified number of iterations, *max_iter*;
- `cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER` - stop the iteration when any of the above conditions is met;
- 30: The maximum number of iterations;
- 0.001: The epsilon value, which is the desired accuracy of the refined corners.

The `cv2.cornerSubPix` is a built in OpenCV function that uses the dot product trick to iteratively refine the corner locations until the termination criteria is reached. The parameters are as follows [80]:

- *gray_calibration*: The grayscale image where the corners were detected;
- *corners*: The initial detected corners from the `cv2.findChessboardCorners` function;

- *(11, 11)*: The size of the search window around each corner. The algorithm will search within this window for the optimal corner position;
- *(-1, -1)*: The zero zone in the search window. This means that the search window's central point is excluded from the search (a common practice to avoid self-influence);
- *criteria*: The termination criteria defined earlier.

Once all calibration images are processed and the chessboard corners refined, the variable *obj_points* contains the 3D coordinates of the chessboard corners for each image and *img_points* contains the corresponding 2D coordinates of the chessboard corners in the image plane.

After that, the *cv2.calibrateCamera* module uses these points to calculate the camera matrix (*mtx*) and distortion coefficients (*dist*) (Listing 8).

```
# Calibrate the camera using all detected corners
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(obj_points,
↪ img_points, gray_calibration.shape[:,-1], None, None)
```

Listing 8: Distortion Correction - Calibrate the camera using all detected corners

After careful examination of the undistorted image, it was determined that radial distortion was not at play. Radial distortion typically manifests as either barrel distortion or pincushion distortion, as shown in Fig. 4.4.

Barrel distortion occurs when straight lines appear to bulge outward from the center of the image, creating a "barrel" effect. Pincushion distortion happens when straight lines appear to curve inward towards the center of the image, creating a "pinched" or squeezed effect [37].

Since none of these effects are seen in the undistorted images, in order to prevent the code from erroneously correcting for a non-existent radial distortion, the radial distortion coefficients are forcibly set to zero (Listing 9).

The code then prints the camera matrix and the distortion coefficients, which in this case are the values in Eq. 4.3 and 4.4, respectively (Listing 10).

```
# Set k1, k2, and k3 to zero
dist[0, 0] = 0.0 # k1
dist[0, 1] = 0.0 # k2
dist[0, 4] = 0.0 # k3
```

Listing 9: Distortion Correction - Set k1, k2, and k3 to zero

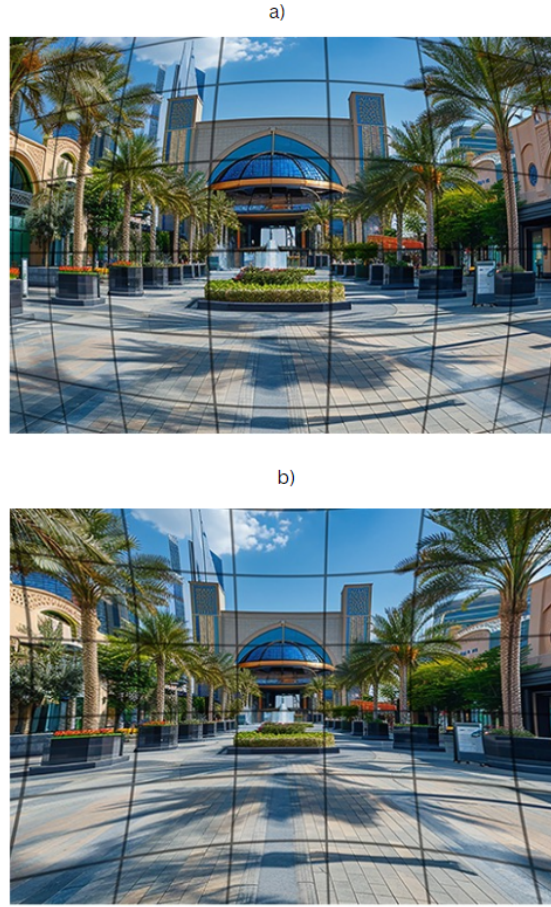


Figure 4.4: Radial distortion: a) Barrel distortion; b) Pincushion distortion [37]

```
# Display the updated camera matrix and distortion coefficients
print("Updated Camera Matrix:")
print(mtx)
print("\nUpdated Distortion Coefficients:")
print(dist)
```

Listing 10: Distortion Correction - Display the updated camera matrix and distortion coefficients

$$\mathbf{K} = \begin{bmatrix} 5.91e + 03 & 0 & 4.97e + 02 \\ 0 & 5.41e + 03 & 1.27e + 02 \\ 0 & 0 & 1 \end{bmatrix} \quad (4.3)$$

$$\mathbf{D} = \begin{bmatrix} 0 & 0 & -6.29e - 02 & 6.15e - 02 & 0 \end{bmatrix} \quad (4.4)$$

As can be observed, there is a slight difference between the f_x and f_y . This negligible difference likely indicates that the focal length in the x-direction is slightly larger than in the y-direction, which could result from the lens being asymmetric or how it's mounted.

Looking at the values of c_x and c_y , it's possible to infer that the principal point is slightly offset in the horizontal direction. The resolution of the Allied Bonito CL-400B/C high speed camera, which was used for the experiment, is 1024x256 pixels. Thus, the principal point should be located at $c_x = 512$ and $c_y = 128$.

With the actual values being $c_x = 497$ and $c_y = 127$, it means that there is an offset of 1 pixel in the y direction and 15 pixels in the x direction. While the offset in the vertical direction is negligible, the horizontal offset might create an impact on the results of the calibration.

The values of p_1 and p_2 , despite small, can have an important effect on the results. If there is tangential distortion, it must be minimal since by observing the images these are not noticeable. Thus, the small values obtained seem reasonable for the scenario at hand.

Once the camera matrix and the distortion coefficients are established, the undistortion phase begins. First, the *glob* function is used once again in order to get all file names of images to undistort (Listing 11).

```
# Glob to get all file names of images to undistort
images_to_undistort = glob.glob(images_to_undistort_pattern)
```

Listing 11: Distortion Correction - Glob to get all file names of images to undistort

Then, a loop with the *cv2.undistort* function is ran through all the distorted images in order to undistort them (Listing 12). Inside this loop, the files are also saved to the specified directory, and lastly, the original and undistorted images are displayed.

```
# Loop through each image and undistort
for image_path in images_to_undistort:
    image_to_undistort = cv2.imread(image_path)
    undistorted_image = cv2.undistort(image_to_undistort, mtx, dist,
    ↪ None, mtx)

    # Save the undistorted image
    base_name = os.path.basename(image_path)
    save_path = os.path.join(save_directory, f'undistorted_{base_name}.jpg')
    cv2.imwrite(save_path, undistorted_image)

    # Display the original and undistorted images
    cv2.imshow('Original Image', image_to_undistort)
    cv2.imshow('Undistorted Image', undistorted_image)
    cv2.waitKey(500) # Wait for a short duration to display each image

cv2.destroyAllWindows()
```

Listing 12: Distortion Correction - Loop through each image and undistort

4.2.3 Distortion Correction - Results and Discussion

Below, in Fig. 4.5, there is a comparison of the images before and after being undistorted.

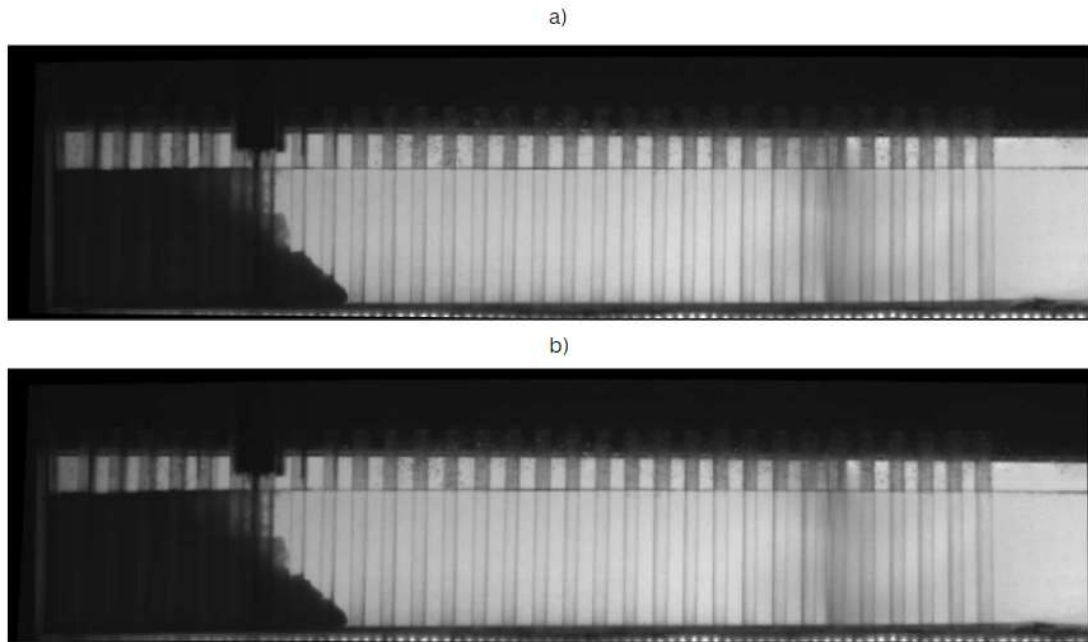


Figure 4.5: a) Distorted Image; b) Undistorted Image

Although the results were seemingly adequate to their intended purpose, there is one important consideration to be made regarding the implemented code.

It can be seen in Fig. 4.6 that during the calibration process, the chessboard corners were not detected 100% accurately, more specifically, the corners in the rightmost part of the pattern were particularly affected.

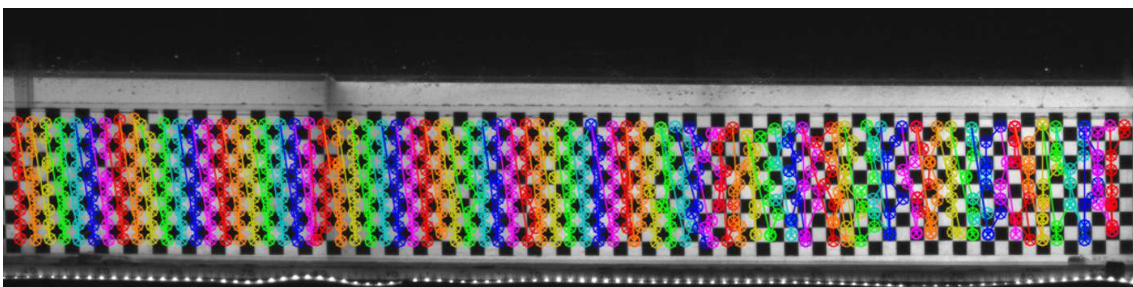


Figure 4.6: Corner detection on one of the calibration images

The inaccuracy of the calibration can be due to the aspect ratio of the chessboard pattern. The chessboard camera calibration technique is usually done with a roughly squared pattern. For the current setup, since the width of the pattern is much greater than its height, it might become more difficult for the program to accurately detect the corners and perform the calibration.

To solve this issue, one approach that might be taken is to have a chessboard pattern with a bigger height. The environment in which the calibration is done is important, but the chessboard pattern doesn't necessarily need to be the exact same length of the flow channel. As long as the lighting and natural filtering of both the calibration and recording of the experiment are the same, the code should be able to undistort the images effectively even with a pattern of different size.

It's also important to remember that the principal point of the camera was slightly offset in the horizontal direction, with the value of c_x being smaller than it was expected. This means that the principal point is offset to the left, which might be causing the camera to more accurately detect the corners in the left side of the pattern.

In order to solve for this issue, using a different camera or adjusting any intrinsic misalignment might be of help.

Additionally, in order to improve the accuracy of the camera calibration, the pictures taken from the chessboard pattern should be as varied as possible. The more images are taken, with the same camera, but from different angles and positions, the better the calibration process will be. Even if each corner detection loop fails at detecting all the corners correctly, having a varied set of calibration pictures can improve the calculation of the camera matrix and the distortion coefficients.

Thus, for future works, it's recommended to collect a larger and more varied set of pictures from the chessboard pattern, given the fact that the ones used for the current code are only twelve, taken from roughly the same position.

4.3 Density Estimation

4.3.1 Density Estimation - Flowchart

After the distortion correction is complete, and a folder with all the undistorted and preprocessed images is created, the density estimation can be calculated.

The density estimation code developed for this dissertation is based on the one used by Alves [5]. As was explained in Chapter 3, during Alves' experiment, in order to calibrate the data for the density estimation, a set of fourteen pictures of the channel were taken, with different concentrations of rhodamine. Thus, for each pixel in the fluid filled channel, it's possible to map pixel value to corresponding density value.

With this, for a given pixel in the undistorted image, a density value can be obtained by interpolating the values in the calibration data. This process is repeated for each pixel in the undistorted image, creating thus a density profile.

The full code is available in Appendix B for detailed reference. The following steps give a high level overview of what the code does:

1. Import necessary libraries and modules;
2. Define input variables;
3. Load calibration data;
4. Display calibration images;
5. Process current images;
6. Interpolate density values using calibration data;
7. Save images and displays plots;
8. Track progress;
9. Release video.

Fig. 4.7 shows the flowchart for the Density Estimation code. A detailed description of the code, with relevant code snippets can be found in the following sub-chapters.

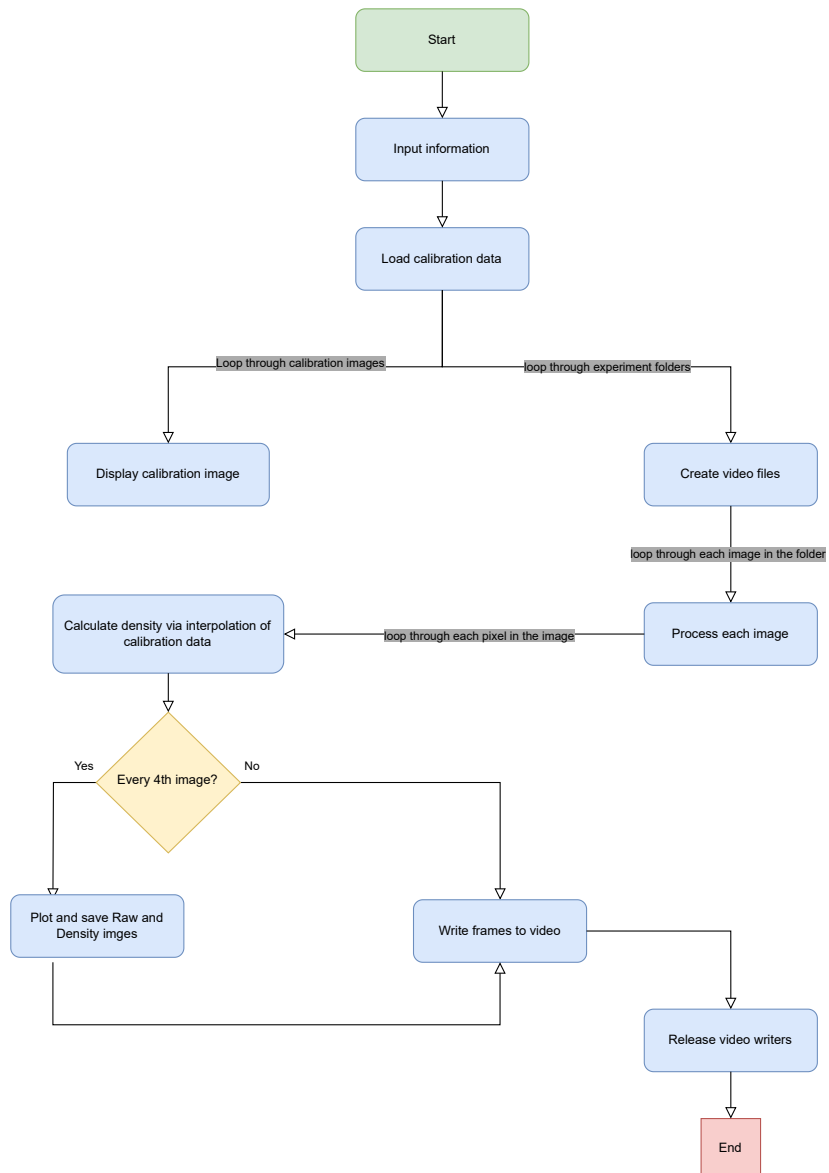


Figure 4.7: Density Estimation Flowchart

4.3.2 Density Estimation - Detailed Description

As usual, the code starts by importing the relevant libraries (Listing 13).

```
import os
import cv2
import numpy as np
from scipy.interpolate import interp1d
from natsort import natsorted
from matplotlib import pyplot as plt
from scipy.io import loadmat
```

Listing 13: Density Estimation - Libraries

Besides *os*, *cv2* and *numpy* which were also imported in the previous code, the density estimation requires several more modules and functions:

- *loadmat* is a function in the *scipy.io* module of the SciPy library in Python. It is used to load MATLAB files (*.mat* files) into Python;
- *natsorted* is a function from the *natsort* library in Python. It provides natural sorting, which means it sorts strings in a way that humans expect, especially when dealing with numbers embedded in text;
- *pyplot* is a module within the Matplotlib library which provides a MATLAB-like interface for creating static, animated, and interactive visualizations in Python;
- *interp1d* is a function within the *scipy.interpolate* module of the SciPy library.

All these modules and functions will be further explained in context whenever they're called in the code. Next, relevant input parameters are defined (Listing 14).

```
# Input information
ny = 18
Ly = 141
nx = 42
Lx = 1024
Lx_m = 1.6
Ly_m = 0.20
scale_x = Lx_m / Lx
scale_y = Ly_m / Ly
deltaT = 1 / (1100 / 25)
main_path = 'path_to_undistorted_images'
```

Listing 14: Density Estimation - Input information

Given the camera resolution, the pixel dimensions of the images which will be processed are:

- Height = 256
- Width = 1024

This is the case for both the undistorted images, and the calibration images that will subsequently be loaded. The parameters ny and Ly establish the range of the height of the calibration images that should be processed, while nx and Lx establish their width. In this case, it is established that the calibration images should only be read from a height of $ny = 18$ pixels through a height of $Ly = 141$ pixels, and from a width of $nx = 50$ pixels through a width of $Lx = 1024$. The reason for this will be explained further along in this chapter.

Lx_m and Ly_m are the actual lengths of the channel in meters, and $scale_x$ and $scale_y$ the respecting scaling factors from pixel to meters. These parameters will be used to plot the density estimation in meters instead of pixels later in the code.

ΔT defines the time interval between successive frames in the video sequence. This was kept from the previous MATLAB code used by Alves.

The *main_path* defines the path to the undistorted images from which the density estimation is to be obtained. These images should have come out from the distortion correction code.

Afterwards, the calibration data is loaded as intended (Listing 15).

```
# Load calibration data
calib_data = loadmat('path_to_calibration_data.mat')
pixel = calib_data['pixel'][ny:Ly, nx:Lx, :][::-1, :]
rho_values = calib_data['Rho']
```

Listing 15: Density Estimation - Load calibration data

Since the calibration data is stored in a MATLAB file, the *loadmat* function is used in order to load the data.

The calibration data is a dictionary with two keys:

- *pixel*: a three dimensional array of dimensions (256, 1024, 14), containing pixel values for the set of fourteen images;
- *rho_values*: a one dimensional array (14), containing the density values for each of the fourteen calibration setups.

The density values are extracted, and as for the pixel values, only the ones which are between ny , Ly , nx and Lx are extracted. This makes it so that only the area encircled by red in Fig. 4.8 is imported into the code.

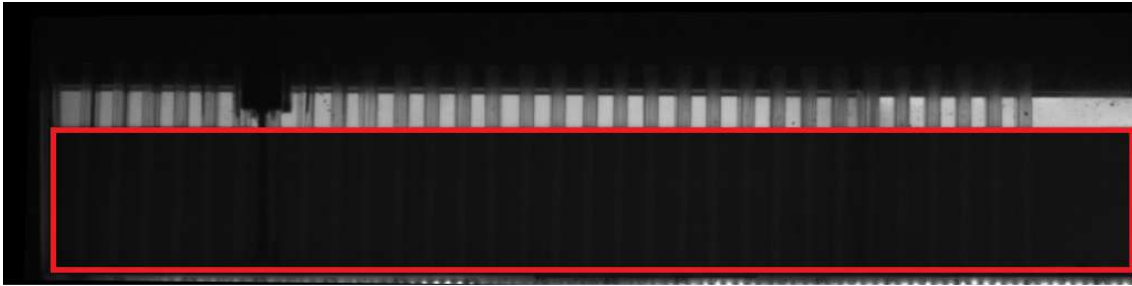


Figure 4.8: Calibration Area

This task is important in order to prevent the code from gathering undesirable data from the calibration data. The area above the red rectangle does not correspond to any fluid with a known density, so segmenting the image in this way creates an overall cleaner data.

The next part of the code displays the calibration images (Listing 16). This step is done in order to guarantee that the calibration data is being imported correctly into the code.

```
# Visualize calibration images
for w in range (0, 13):

    calibration_images = pixel[:, :, w]

    plt.figure(figsize=(10, 5))
    plt.imshow(calibration_images, cmap='viridis', aspect='auto')
    plt.colorbar(label='Pixel Intensity')
    plt.gca().set_aspect('equal')
    plt.title(f'Calibration Image {w+1}')
    plt.xlabel('Width (pixels)')
    plt.ylabel('Height (pixels)')
    plt.show()
```

Listing 16: Density Estimation - Visualize calibration images

Fig. 4.9 shows the 1st, 7th and 14th calibration images as imported into the code, and as can be seen, the data is being loaded correctly.

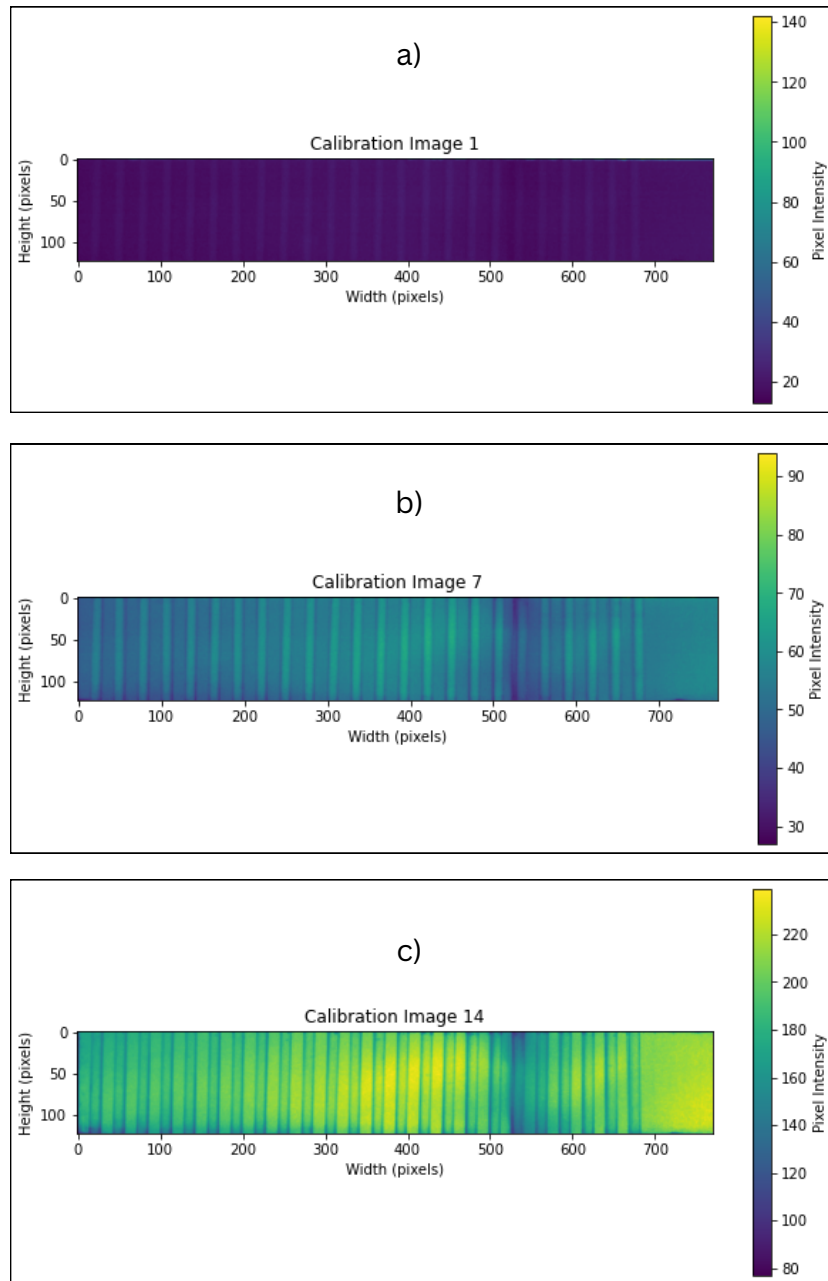


Figure 4.9: Calibration Images: a) 1st Image b) 7th Image c) 14th Image

Afterwards, the main loop is initialized. It starts by establishing the range of the loop, and reading the desired directory where the undistorted images of the experiment are located (Listing 17).

```
# Loop through experiment folders
for r in range(1, 3):
    # Read images
    files = natsorted(os.listdir(os.path.join(main_path, f'R{r}')))

    # Create video files and features
    vidObj =
    ↪ cv2.VideoWriter(f'path_to_results_folder/R{r}/Current_R{r}.avi',
    ↪ cv2.VideoWriter_fourcc(*'DIVX'), 4, (1024, 256))
    vidraw =
    ↪ cv2.VideoWriter(f'path_to_results_folder/R{r}/Rawimages_R{r}.avi',
    ↪ cv2.VideoWriter_fourcc(*'DIVX'), 4, (1024, 256))
```

Listing 17: Density Estimation - Loop through experiment folders

For this example, it is considered that there were three lock-exchange experiments done, and that the undistorted images of the experiment are stored in folders R1, R2 and R3 inside the *main_path* folder. The *natsorted* function is once again used, as in the previous code.

The video files are also created in order to store the image data and a nested loop is then created inside the previous one to process the current images (Listing 18).

```
for k, filename in enumerate(files, start=1):
    time = k * deltaT
    A = cv2.imread(os.path.join(main_path, f'R{r}', filename),
    ↪ cv2.IMREAD_GRAYSCALE)[(256-ny-(Ly-ny)):(256-ny), nx:Lx]

    density = np.zeros_like(A, dtype=float)[::-1, :]
```

Listing 18: Density Estimation - Process experiment images

This section of code performs the following actions:

- It initiates a loop over the *files* list, where *k* is the index starting from 1, and *filename* is the current file name in the iteration;
- It calculates the *time* variable by multiplying the index *k* by *deltaT*, which represents the time interval between successive frames;
- It reads the image file located at the path *os.path.join(main_path, f'R{r}', filename)* in grayscale mode using *cv2.imread*. The specific section of the image is cropped by

slicing the pixel array from $(256-ny-(Ly-ny))$ to $(256-ny)$ for the height and from nx to Lx for the width;

- It initializes a *density* array with the same shape as the cropped image A, filled with zeros and of type *float*.

Finally, one more loop is created in order to perform the density calculation pixel by pixel (Listing 19).

```
for j in range(A.shape[1]):
    for i in range(A.shape[0]):
        CAL = pixel[i, j, :]
        RGB = A[i, j]

        # Ensure RGB value is within calibration range
        RGB = min(max(RGB, CAL.min()), CAL.max())

        # print (CAL)

        # Linear interpolation to find density value
        interp_func = interp1d(CAL, rho_values[0,:],
                               ↪ kind='linear', fill_value="interpolation")
        density[i, j] = interp_func(RGB)
```

Listing 19: Density Estimation - Loop through image pixels and perform density estimation via interpolation of the calibration data

The main points of this section of the code are the following:

- The nested loop iterates over each pixel in the image A. This Variable is used as a placeholder for the undistorted experiment images;
- *CAL* is the calibration data for the pixel at position (i, j) ;
- *RGB* is the pixel value at position (i, j) in the image A;
- The RGB value is clamped to ensure it is within the calibration range;
- A linear interpolation function (*interp_func*) is created using the calibration data (*CAL*) and the corresponding density values (*rho_values*);
- The density value for the pixel is then found using the interpolation function and stored in the *density* array.

After the interpolation is done for all pixels of the undistorted experiment image, every 4th frame ($k \% 4 == 0$), the density image is plotted and saved. The same is done for the raw images (Listing 20).

```
if k % 4 == 0:
    # Plot density image
    plt.figure(figsize=(14, 4.5))
    plt.pcolor(density[::-1, :], cmap='jet', vmin=985, vmax=1025)
    plt.colorbar(label='Density (kg/m^3)')
    plt.gca().set_aspect('equal')
    x_ticks = np.arange(0, (Lx-nx) * scale_x, 0.2)
    y_ticks = np.arange(0, (Ly-ny) * scale_y, 0.05)
    plt.xticks(np.arange(0, (Lx-nx), 0.2 / scale_x),
        ↪ np.round(x_ticks, 2))
    plt.yticks(np.arange(0, (Ly-ny), 0.05 / scale_y),
        ↪ np.round(y_ticks, 2))
    plt.xlabel('$x \mathrm{(m)}$', fontsize=16)
    plt.ylabel('$z \mathrm{(m)}$', fontsize=16)
    plt.title(f't={time:.2f}s', fontsize=14)
    plt.savefig(f'path_to_results_folder/R{r}/Image_{k}.png')
    plt.close()

    # Plot raw image
    plt.figure(figsize=(14, 4.5))
    plt.pcolor(A[::-1, :], cmap='viridis')
    plt.colorbar(label='Pixel Intensity')
    plt.gca().set_aspect('equal')
    plt.xticks(np.arange(0, (Lx-nx), 0.2 / scale_x),
        ↪ np.round(x_ticks, 2))
    plt.yticks(np.arange(0, (Ly-ny), 0.05 / scale_y),
        ↪ np.round(y_ticks, 2))
    plt.xlabel('$x \mathrm{(m)}$', fontsize=16)
    plt.ylabel('$z \mathrm{(m)}$', fontsize=16)
    plt.title(f't={time:.2f}s', fontsize=14)
    plt.savefig(f'path_to_results_folder/R{r}/RawImage_{k}.png')
    plt.close()
```

Listing 20: Density Estimation - Plot and save Raw and Density images

The video files are written after the greyscale image is converted into RGB with the `cv2.cvtColor` function (Listing 21).

```
# Write frames to video
vidObj.write(cv2.cvtColor(A, cv2.COLOR_GRAY2BGR))
vidraw.write(cv2.cvtColor(density.astype(np.uint8),
    ↪ cv2.COLOR_GRAY2BGR))
```

Listing 21: Density Estimation - Write frames to video

Finally, the variable k is printed for tracking purposes, and the video files are released to ensure all data is written properly and resources are freed (Listing 22).

```
print(k)

vidObj.release()
vidraw.release()
```

Listing 22: Density Estimation - Print k and release video files

4.3.3 Density Estimation - Results and Discussion

The density estimation process yielded a set of density profiles for each experiment, which were stored as images and video files. These profiles provide a visual representation of the density distribution within the fluid channel at various time intervals.

Upon visual inspection, the density profiles show a clear and accurate representation of the fluid flow and density distribution. The calibration process appears to have successfully mapped the pixel values to the corresponding density values, and the density range is between the ones shown in Fig. 3.8, resulting in a coherent density profile, as can be seen in Fig. 4.10.

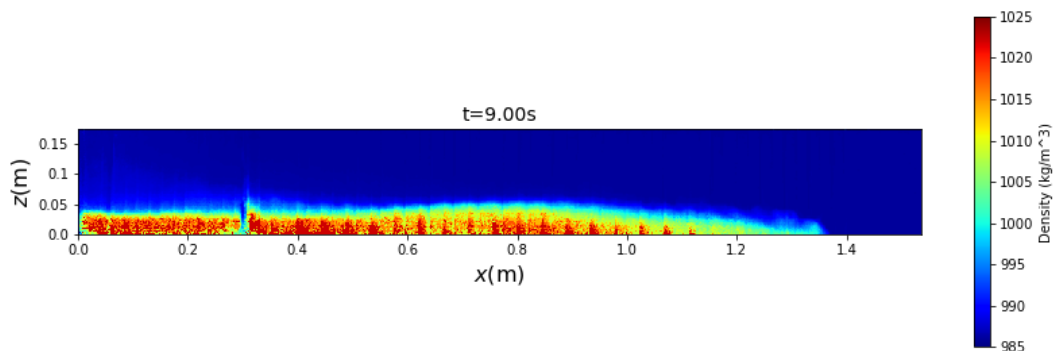


Figure 4.10: Density estimation plot

For a more quantitative analysis, the density profiles were displayed as a function of the concentration and the normalized height, as was done in the previous dissertation by Alves. Fig. 4.11 shows a comparison of the two results.

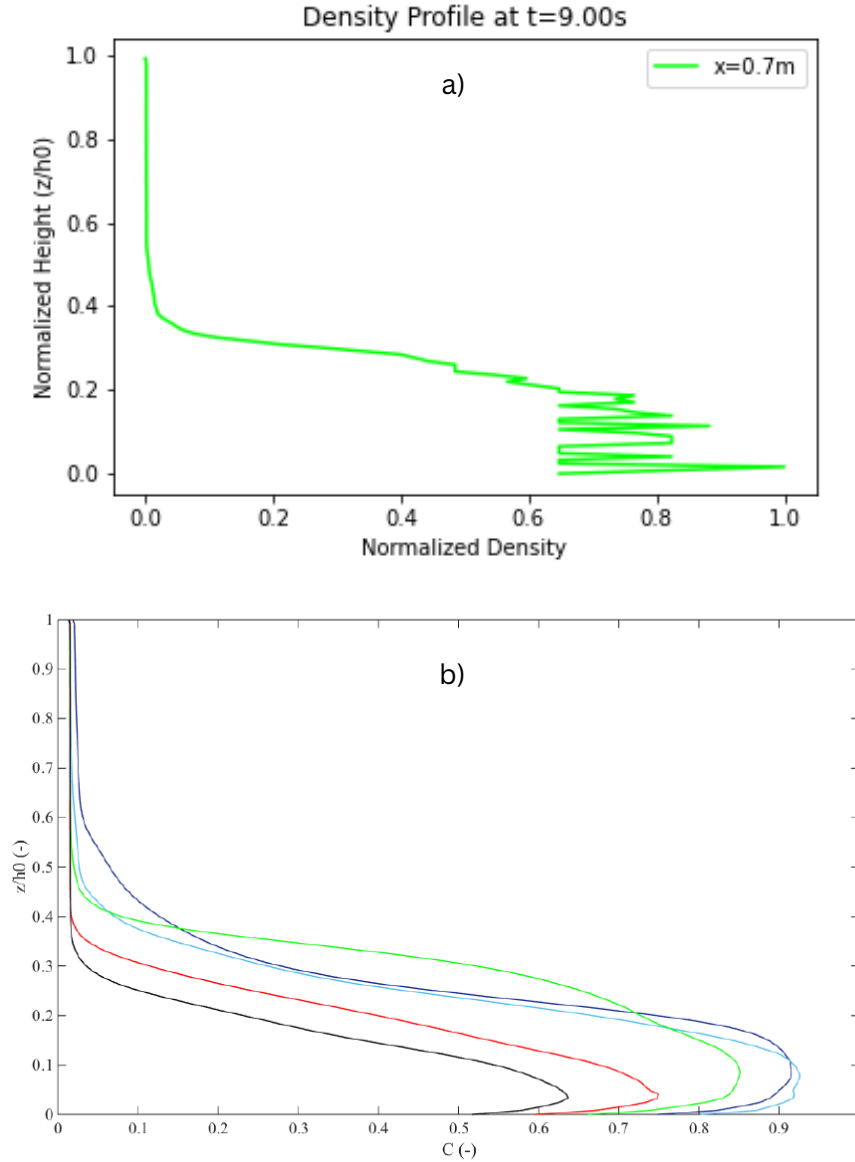


Figure 4.11: Density graphs: a) Obtained with the current code; b) Obtained by Alves, where the different colored lines represent the density profiles at different sections of the flow, with the green one representing the flow at section $x=0.7m$ [5]

Although not exactly equal, the line plot for the density profile at $x=0.7m$ has roughly the same shape. As can be seen, the graph obtained in the present dissertation is dealing with noise in the lower part of the channel. Alves' results don't have noise because the density estimation has a strong gaussian filter applied.

Therefore, in order to correct the noise in the new results, a gaussian filter was also applied to the present density estimation. However, since the filter smooths the pixel values based on it's neighbours' values, some data at the bottom of the channel was lost, as shown in Fig. 4.12.

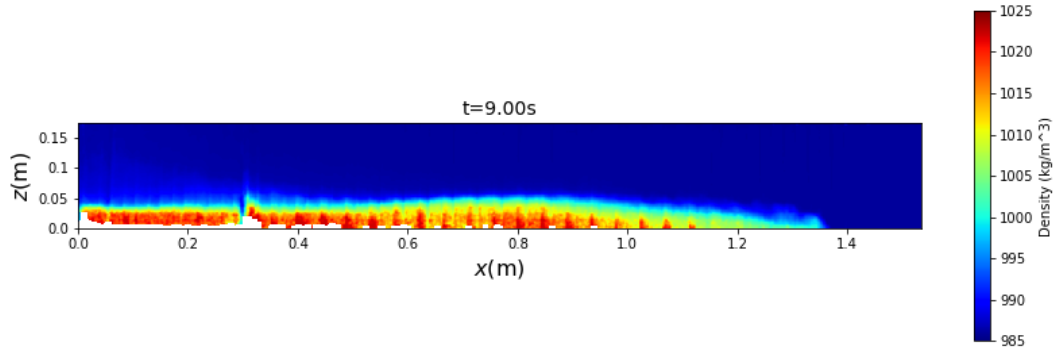


Figure 4.12: Density estimation plot with gaussian filter applied

Even with the missing values, a new and more appropriate plot for the density profile at $x=0.7m$ could be obtained, as is presented in Fig. 4.13. Although it showed less noise, the issue still persisted. The application of more intense gaussian filters led to even more loss in the data, and the density graphs could not be obtained.

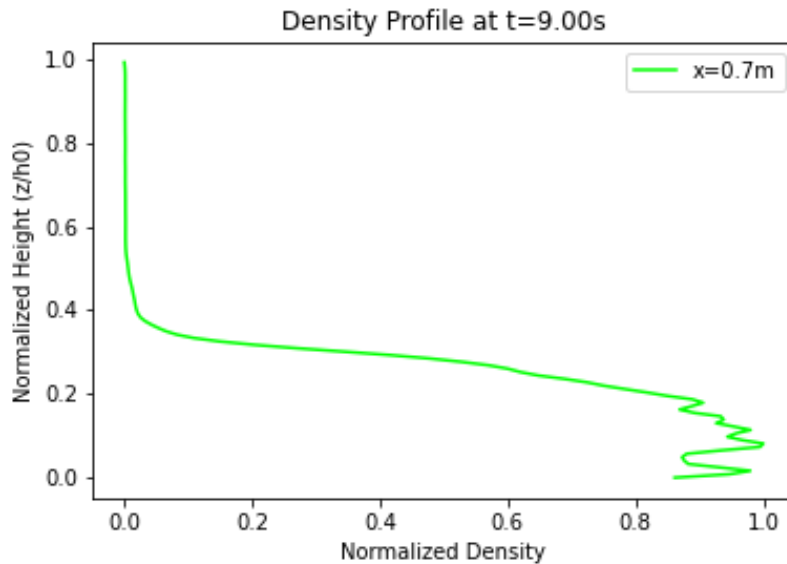


Figure 4.13: Density graph at $x=0.7m$ with gaussian filter applied

The density estimation process yielded coherent density profiles, which were visualized and compared to previous works. The application of a gaussian filter helped to reduce noise, albeit at the cost of losing some pixel values at the bottom of the channel. Future improvements could involve refining the filtering processes to preserve the integrity of the density profiles. The smoothness of the density graph will be resolved in the next chapter.

To close this chapter, it's important to make a few remarks about the translation of the MATLAB code into Python. Firstly, despite calling it a 'translation', the original code could not simply be written in another programming language in order to execute the same procedure. A thorough exploration of the required tasks, as well as dataset and parameters used, had to be conducted in order to write a compatible Python code.

Several debugging sessions had to be undertaken. One of the issues that was encountered was that a direct translation of the original code created an inverted representation of the generated images, as can be seen in Fig. 4.14.

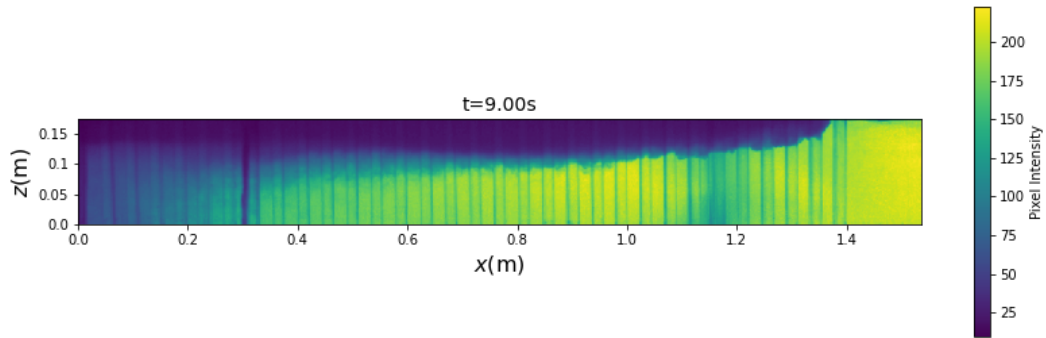


Figure 4.14: Raw image with inverted rows

This is due to the fact that when Python pulls the pixel values from the dataset, it assumes the columns are going from left to right, and the rows as going from top to bottom. MATLAB also assumes that the columns are going from left to right, but as far as the rows are concerned, it assumes they're going from bottom to top.

That's the reason why the Python code needs to have the slicing operation `[::-1, :]` when loading the calibration data and plotting the images. It reverses the order of the rows, as intended.

Also, the opportunity to improve the code upon translation was taken whenever possible. As an example, in the original MATLAB code, the only segmentation done was in the *y-axis*, and it created a sub-optimal window for the plotting of the density estimation, as can be seen in Fig. 4.15. In order to improve this, the segmentation in the *y-axis* was tightened and a segmentation in the *x-axis* was introduced.

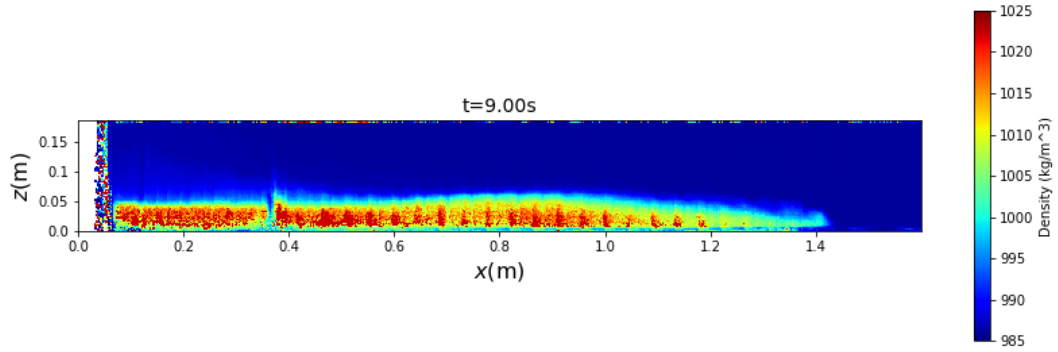


Figure 4.15: Density estimation plot with segmentation as done in the original MATLAB code [5]

The second important remark is that even without any machine learning applied, having a Python code that simply does what the original MATLAB code did, is extremely valuable.

As was previously stated, Python is the primary language used for machine learning and computer vision and MATLAB has its limitations in terms of available libraries. For instance, TensorFlow and scikit-learn, two of the most used machine learning libraries are not directly available in MATLAB.

Python's wider ecosystem and its open source nature makes it an extremely powerful language for machine learning tasks, and with the distortion correction and density estimation tasks now being performed in this programming language, a lot more can be done more easily during future research [72].

DENSITY ESTIMATION: MACHINE LEARNING IMPROVEMENT

5.1 Introduction

The present chapter aims at describing how machine learning can be used to improve the density estimation code.

To start, one major point of improvement of the code will be tackled with the help of a Random Forest Regressor. A simple machine learning model will be applied to the code, with the respective results being analyzed and compared with the ones previously obtained.

Afterwards, more complex deep learning models will be considered for other potential points of improvement, and their possible implementation will be studied. Due to lack of computational power, these models will only have their potential implementation researched, instead of actually being applied.

5.2 Random Forest Regressor

5.2.1 Random Forest Regressor: Flowchart

One potential point of improvement of the code relates to the data calibration interpolation. The calculation of density values are being done using linear interpolation. In this process, pixel intensity values from the images are matched to corresponding density values provided in the calibration data.

Linear interpolation assumes a straight-line relationship between pixel intensity and density values, which may not accurately capture more complex, non-linear relationships, being thus prone to underfitting.

On the other hand, the current method relies heavily on the specific calibration data provided. If this data is not representative of real-world variations, the system might overfit and fail to generalize well to new images.

Additionally, this type of interpolation does not inherently handle noise in the calibration data. If there are noisy or outlier data points, the interpolation can lead to erroneous density estimations.

To enhance the calculation of density values based on pixel intensities, the interpolation function was replaced by a supervised learning model called Random Forest. This model is a powerful ensemble learning method and can handle complex, non-linear relationships more effectively than linear interpolations.

Random Forests consist of multiple decision trees that work together to improve prediction accuracy. Each tree is trained on a random subset of the training data, and the final prediction is made by aggregating the predictions of all the individual trees. This aggregation helps to reduce both underfitting and overfitting and increases the model's robustness against noise and outliers in the data [10].

Random Forest models can be of two kinds. The Random Forest Classifier predicts categorical outcomes by taking a majority vote among the trees. For this dissertation, however, a Random Forest Regressor will be applied. This model predicts numerical values by averaging the outputs of multiple decision trees. Fig. 5.1 shows the general architecture of the Random Forest model [24].

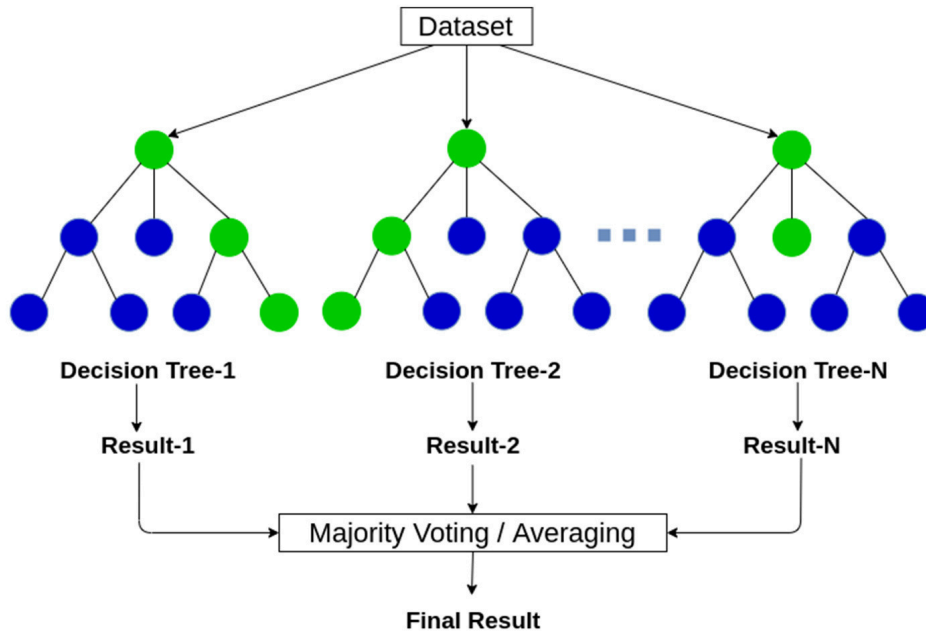


Figure 5.1: Random Forest Model Structure [11]

The full code is available in Appendix C for detailed reference. The following steps give a high level overview of what the code does:

1. Import necessary libraries and modules;
2. Define input variables;
3. Load calibration data;
4. Prepare Training Data;
5. Split the tests;
6. Train Random Forest Regressor;
7. Evaluate the model;
8. Displays calibration images;
9. Processes current images;
10. Predict density values using random forests and calibration data;
11. Saves images and displays plots;
12. Tracks progress;
13. Releases video.

Fig. 5.2 shows the flowchart for the Improved Density Estimation code.

A detailed description of the improvements done to the original density estimation code, with relevant code snippets, can be found in the following sub-chapters.

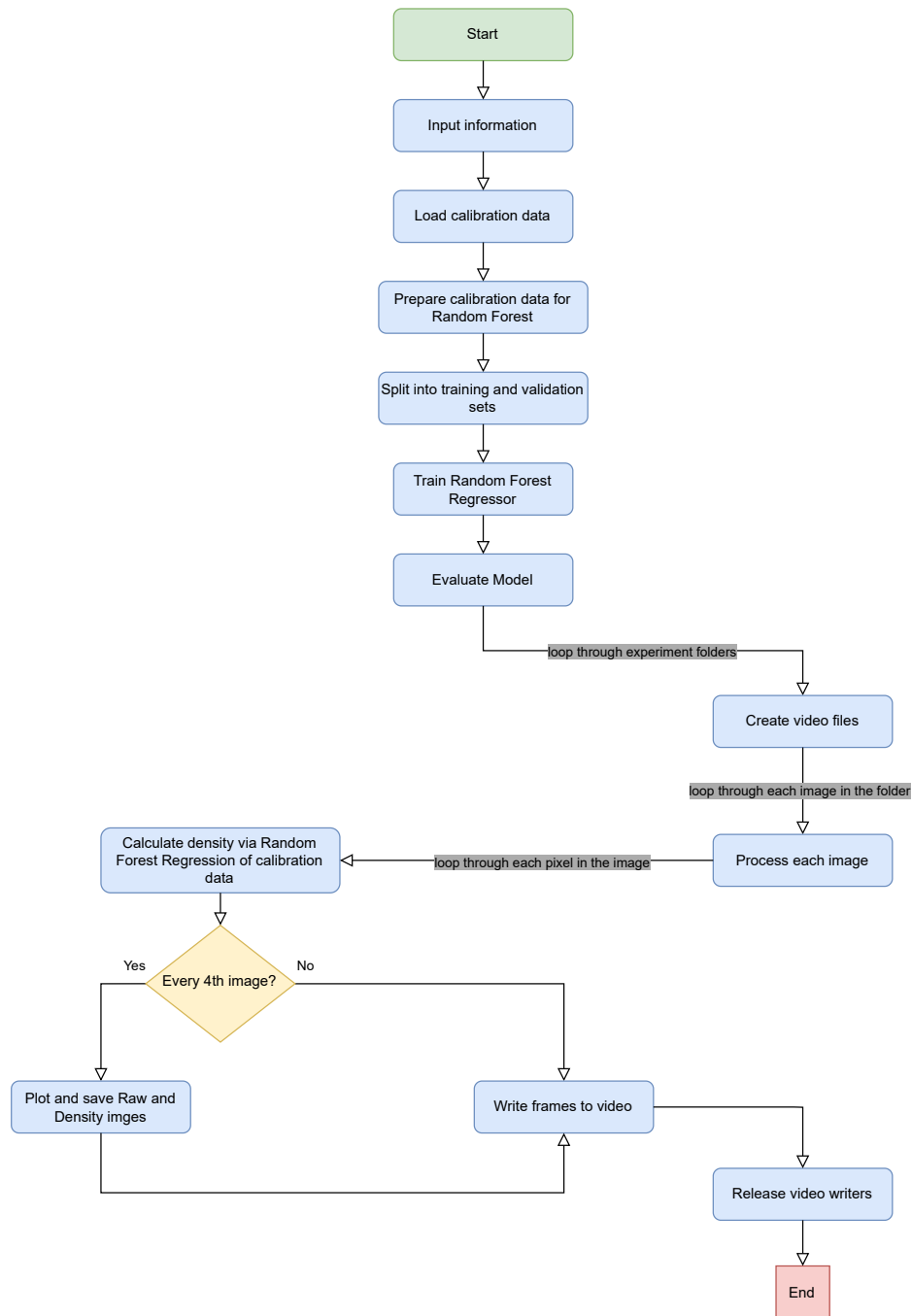


Figure 5.2: Random Forest Regressor Code Flowchart

5.2.2 Random Forest Regressor: Detailed Description

Besides the ones already imported in the previous density estimation code, the Random Forest Regressor code imports the following libraries (Listing 23):

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
```

Listing 23: Random Forest - Libraries

The libraries imported above serve the following purposes:

- The class *RandomForestRegressor* is used to create and train a Random Forest model for regression tasks;
- *train_test_split* is a function which is used to split a dataset into training and validation (or testing) sets. As was stated in Chapter 2.5.2.2, this is essential for evaluating the performance of machine learning models by ensuring that they are tested on data they haven't seen during training;
- *mean_squared_error* is a function used to evaluate the performance of regression models by calculating the average squared differences between actual and predicted values.

After importing the necessary libraries, defining input values and loading calibration data, the code proceeds to prepare the training data for the Random Forest model (Listing 24).

```
# Prepare training data for Random Forest
X_train = []
y_train = []
for i in range(pixel.shape[0]):
    for j in range(pixel.shape[1]):
        for k in range(pixel.shape[2]):
            X_train.append(pixel[i, j, k])
            y_train.append(rho_values[0, k])
X_train = np.array(X_train).reshape(-1, 1)
y_train = np.array(y_train)
```

Listing 24: Random Forest - Prepare training data for Random Forest

Two empty lists, *X_train* and *y_train*, are initialized. These lists will store the features (pixel intensity values) and the corresponding target values (density values), respectively, for training the model.

A nested loop that iterates over the calibration data is then created and:

- The code uses three nested loops to iterate over each element in the *pixel* array (*i*, *j* and *k*);
- The pixel intensity value is appended to the *X_train* list, and the corresponding density value is appended to the *y_train* list.

After populating the lists, they are converted to *numpy* arrays.

X_train is reshaped to have a shape of $(-1, 1)$, meaning it is converted to a 2D array with one column and as many rows as needed. This is necessary because the Random Forest model expects the input features to be in a 2D array format.

y_train is converted to a *numpy* array without any reshaping, as it naturally fits into the expected 1D array format for target values.

The training and validation sets are then split, in order for the model evaluation to be performed later (Listing 25).

```
# Split into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
↪ test_size=0.2, random_state=42)
```

Listing 25: Random Forest - Split into training and validation sets

The split is done by the *train_test_split()* function from scikit-learn. Since the test size is set to *test_size = 0.2*, this means that the validation set will correspond to 20% of the data while the remaining 80% is used to train the model.

The *random_state = 42* parameter sets the seed for the random number generator, ensuring that the results are reproducible. Using a fixed seed means that every time the code is run, the same results will be obtained.

Afterwards, the Random Forest Regressor is trained (Listing 26).

```
# Train Random Forest Regressor
rf_regressor = RandomForestRegressor(n_estimators=100, random_state=42)
rf_regressor.fit(X_train, y_train)
```

Listing 26: Random Forest - Train the Regressor

The *RandomForestRegressor* class is used to create a Random Forest model for regression tasks, in contrast with the *RandomForestClassifier* class, which is used for classification tasks.

The number of decision trees in the forest is determined by the *n_estimators=100* parameter. In this case, 100 trees will be created and used. More trees generally lead to better performance but also increase computational cost.

The model fitting is then performed by the `fit(X_train, y_train)` method, which trains the Random Forest model using the provided training data:

- `X_train`: This is the input feature set, where each element corresponds to a pixel intensity value from the calibration data;
- `y_train`: This is the target set, where each element corresponds to the density value associated with the pixel intensity in `X_train`.

During training, the Random Forest algorithm will:

- Create 100 decision trees using different subsets of the training data;
- Each tree will learn to predict the target density value based on the input feature;
- The final prediction for a given input will be the average of the predictions from all the individual trees.

After the training is complete, the model makes density values predictions and is evaluated by the Mean Squared Error (MSE) (Listing 27).

```
# Evaluate the model
val_predictions = rf_regressor.predict(X_val)
mse = mean_squared_error(y_val, val_predictions)
print(f'Validation MSE: {mse}')
```

Listing 27: Random Forest - Evaluate the model

The MSE is a measure of the average squared differences between the actual and predicted values. In this context, it is used to evaluate the performance of the Random Forest Regressor by quantifying how well the predicted density values match the actual density values in the validation set. A lower MSE indicates a better fit of the model to the data, meaning the predictions are closer to the true values [31].

Finally, the model is ready to predict the density values and create the density profiles (Listing 28).

```
for j in range(A.shape[1]):
    for i in range(A.shape[0]):
        RGB = A[i, j]
        RGB = min(max(RGB, X_train.min()), X_train.max())
        density[i, j] = rf_regressor.predict([[RGB]])[0]
```

Listing 28: Random Forest - Loop through image pixels and perform density estimation via Random Forest Regression of the calibration data

Inside the image processing loop, only two things have changed from the original:

1. $RGB = \min(\max(RGB, X_{train.min()}), X_{train.max()})$: In order to prevent the model from encountering out-of-range values, the pixel intensity RGB now needs to be clamped to ensure it falls within the range of values in the training data, instead of the calibration values in general;
2. $density[i, j] = rf_regressor.predict([[RGB]])[0]$: The clamped RGB value is passed to the trained Random Forest model to predict the corresponding density value. The prediction is stored in the $density$ array at position (i, j) .

It is clear that replacing the interpolation function with a Random Forest Regressor is a simple and straightforward process. Nonetheless, it is essential to analyze whether the resulting predictions are reliable and accurate.

5.2.3 Random Forest Regressor: Results and Discussion

Before analyzing the results, it's important to give a few remarks about the chosen model evaluation metric. For this case, the MSE was chosen since it is the most widely used for these types of machine learning tasks. Mathematically, the MSE is defined by Eq. (5.1) [32]:

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 \quad (5.1)$$

Where y_i is the real value of the target variable, $\hat{y}_i^2$ is it's predicted value and n is the number of data points.

The value of the MSE should be as low as possible, and documenting it is important to evaluate further improvements of the model. For the current results the mean squared error was $MSE = 13.40 \text{ kg}^2/\text{m}^6$.

By analyzing Eq. (5.1), and keeping in mind that the target variable is density, it can be observed that for this case, the units of the MSE are kg^2/m^6 . In order to have an clearer picture of the results, it is also common to calculate the Root Mean Squared Error (RMSE), which can be defined be Eq. (5.2) [26].

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2} \quad (5.2)$$

This metric is useful because it has the same units as the target variable, which in this case is kg/m^3 . The RMSE for this model was $RMSE = 3.66 \text{ kg}/\text{m}^3$.

Another useful metric is the Normalized Root Mean Squared Error (nRMSE), which expresses the RMSE relative to the range of the target variable. The nRMSE is defined by Eq. (5.3) [82].

$$\text{nRMSE} = \frac{\sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}}{y_{\max} - y_{\min}} \quad (5.3)$$

Given the range of densities at play in the experiment, a variance of 3.66 kg/m^3 yields a value of $\text{nRMSE} = 9.15\%$.

In order to more precisely capture the calibration data for the calculation, the value of nx was changed to 250. This makes it so that only the area after the gate of the lock-exchange experiment is taken into account.

By doing this, the MSE of the Random Forest model decreased to $\text{MSE} = 5.89 \text{ kg}^2/\text{m}^6$, and the RMSE decreased to $\text{RMSE} = 2.43 \text{ kg/m}^3$.

This simple segmentation decreased the error to $\text{nRMSE} = 6.07\%$, yielding the results displayed in Fig. 5.3.

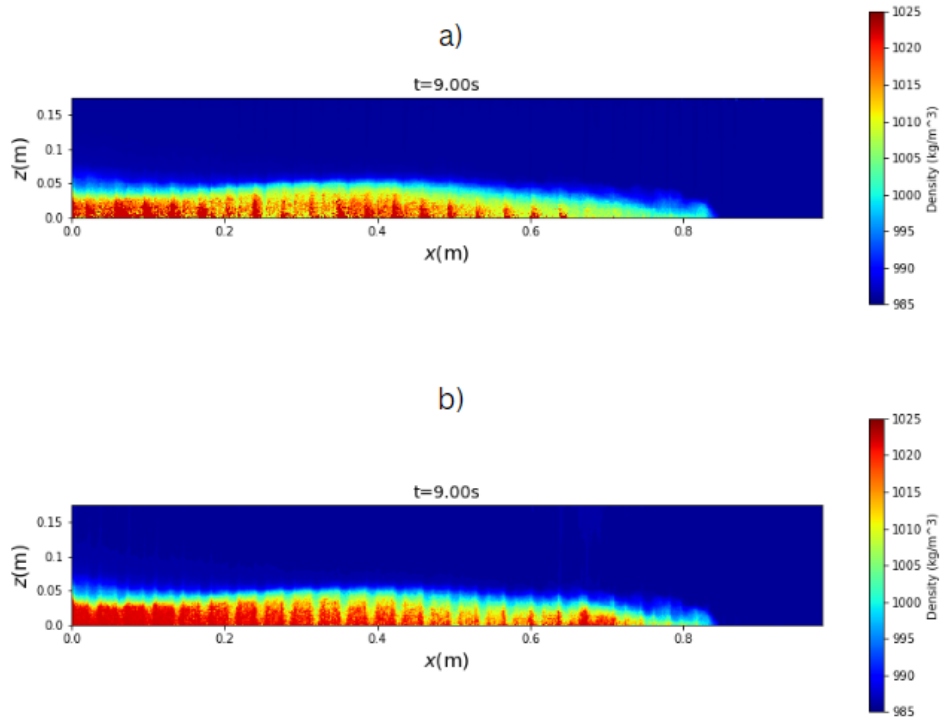


Figure 5.3: Density estimation plot with shorter channel view: a) Interpolation Function; b) Random Forest Regressor

In terms of how the Random Forest Regressor performed compared with the interpolation model, Fig. 5.4 shows a comparison between the density profile obtained using both models. It's noticeable that the results yielded by the Random Forest Regressor are smoother and, as expected, appear to be much less sensitive to noise.

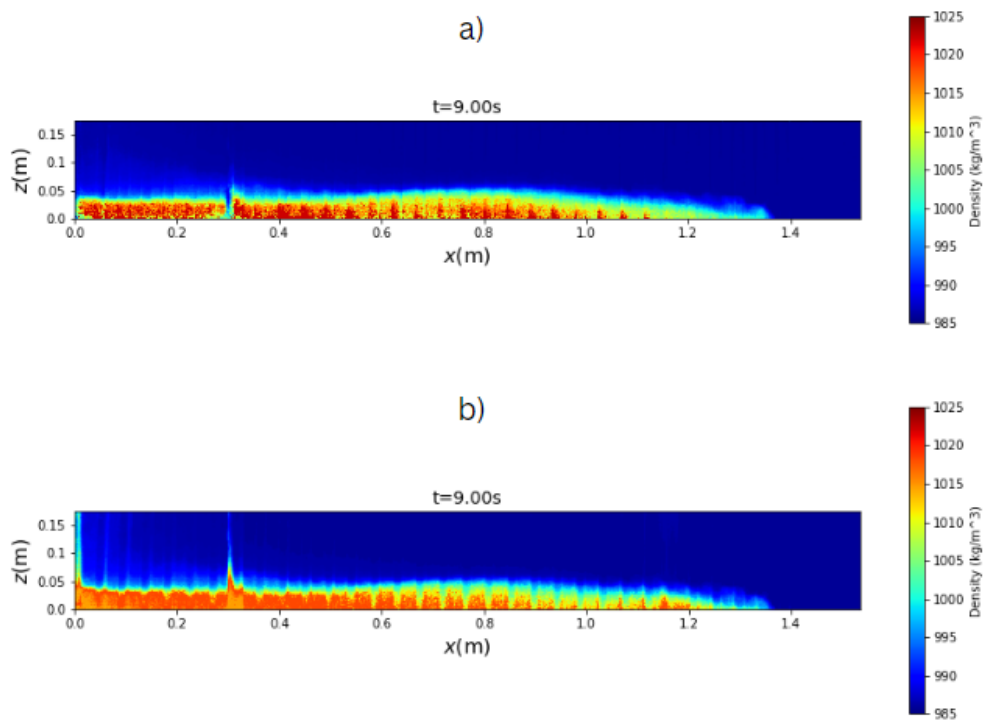


Figure 5.4: Density estimation plot obtained using: a) Interpolation Function; b) Random Forests

Another issue that the Random Forest model solved was that fact that in the previous code, because there was too much noise in the plot, when a gaussian filter was applied, some of the data near the wall was eliminated. With the Random Forest, this issue has been eliminated, as can be seen in Fig. 5.5.

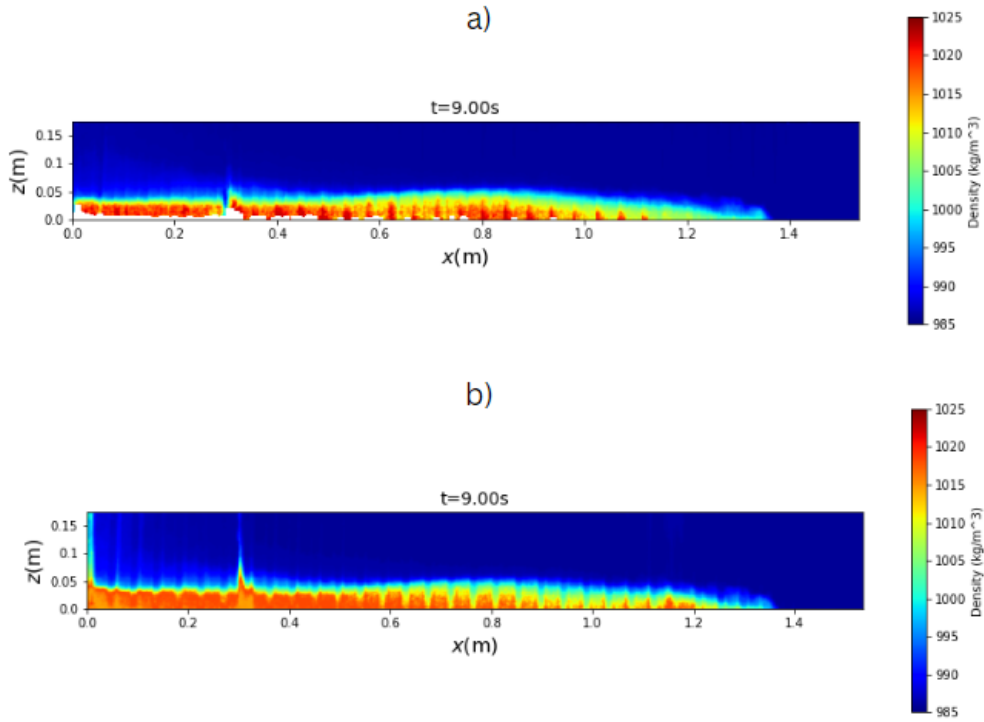


Figure 5.5: Density estimation plot with gaussian filter applied: a) Interpolation Function; b) Random Forest

This improvement made it possible to apply progressively more intense gaussian filters, yielding the density graphs shown in Fig. 5.6. These graphs are a lot more similar to the one obtained by Alves, with the effects of noise becoming negligible.

In order to further evaluate the robustness of the Random Forest Regressor, the model's parameters should be tweaked and it's metrics compared with the ones already obtained or with ones from other machine learning models.

Additionally, it's important to note that Random Forests are extremely powerful for multi-variable problems. This means that if a dataset can be created with additional features, this machine learning technique has the potential to drastically improve the results of the density estimation.

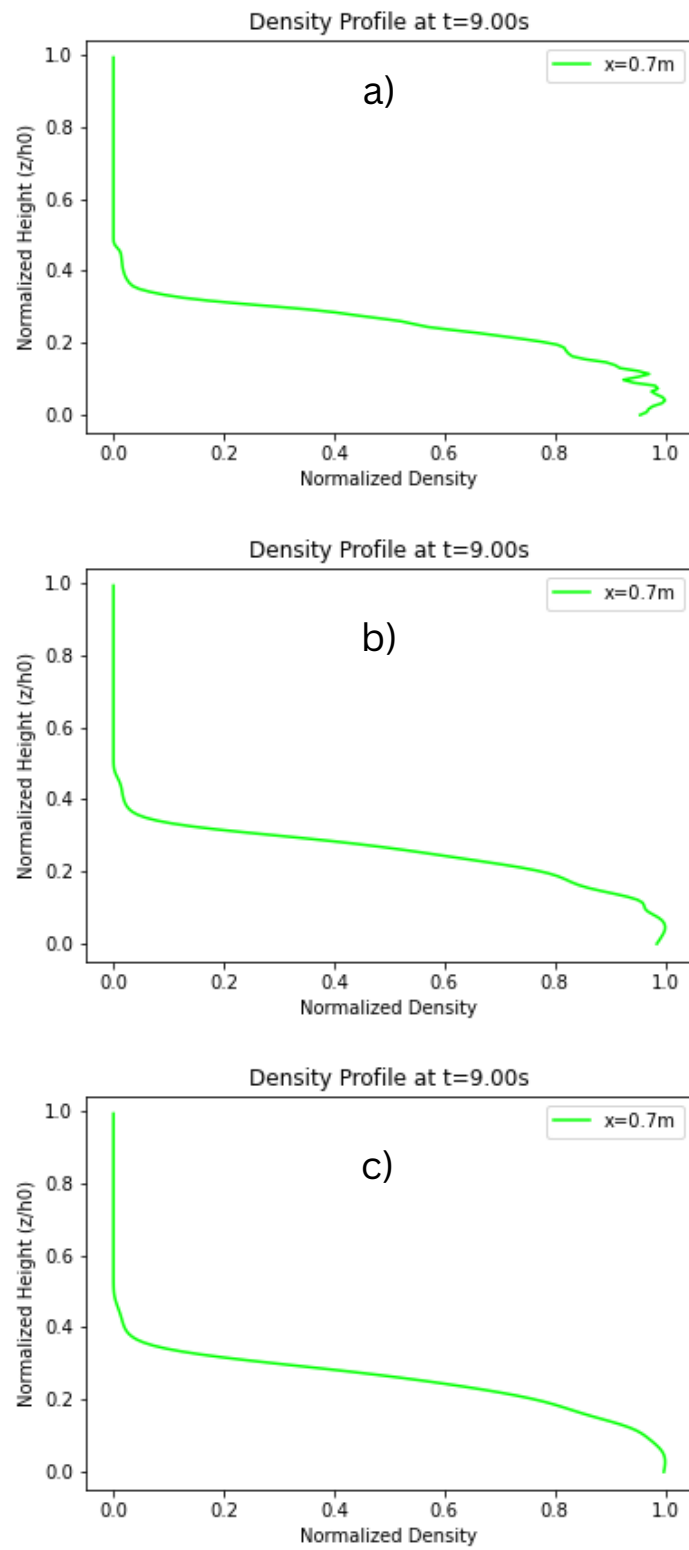


Figure 5.6: Density graphs with progressively more intense gaussian filters applied in the Random Forest model: a) $\sigma = 1$; b) $\sigma = 2$; c) $\sigma = 3$

5.3 Further Machine Learning Improvements

Even with the improved results obtained with the Random Forest Regressor, several other aspects of the density estimation code could be tackled using more advanced machine learning techniques.

For the purposes of this dissertation, two of these aspects were chosen to be further discussed, given their relevance to the problem at hand: the lack of both spatial and temporal consistency.

As was previously stated, due to lack of computational power, these more advanced machine learning techniques will be discussed but not implemented in the code itself. Nonetheless, the analysis of how they could be implemented can prove useful for future research.

5.3.1 Spatial Consistency

In the original code, density values are computed based on a single pixel's intensity, meaning that each pixel in an image is evaluated in isolation without considering the surrounding pixels. This approach uses the intensity value of a single pixel to determine the density at that specific point.

While this method can be straightforward, it may not capture the spatial context or patterns present in the image that could contribute to a more accurate density estimation. For instance, neighboring pixels often share similar characteristics, and their collective information can provide a richer, more reliable representation of the density.

To address these limitations, deep learning models that consider spatial context can be employed. Convolutional Neural Networks (CNNs) are particularly well-suited for this task. CNNs are designed to recognize spatial hierarchies in data by applying convolutional filters that capture local patterns and features [91].

As was previously stated, CNNs consist of convolutional layers, pooling layers and fully connected layers. Listing 29 shows how this layered model architecture should be structured [85].

`model = Sequential()` creates a neural network model where layers are stacked one after another. The convolutional layers of `Conv2D` will process the images and detect useful features using filters (or kernels). The first layer is responsible for detecting basic features. A 3x3 filter is applied in this layer, meaning that it looks at patches of 3x3 pixels in order to detect features.

```
# Define the CNN model
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(Ly-ny,
↪ Lx-nx, 1)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mean_squared_error')
```

Listing 29: Convolutional Neural Network - Model definition

The ReLU function introduces non-linearity. Without it, the model would behave like any other linear model, hence the need for an activation function. Mathematically, it is defined by Eq. (5.4) [6, 36].

$$\text{ReLU}(x) = \max(0, x) \quad (5.4)$$

The second convolution layer simply adds more filters, and by doing so, it focuses on detecting more complex features.

In the convolutional layers, neurons are associated with specific kernels. Each neuron receives input from local patch of pixels, performs the weighted sum, applies an activation function and outputs the result. The output of a neuron in a convolutional layer is a single number, which represents a feature detected by the filter in a specific region of the input [56].

The pooling layers of *MaxPooling2D* are used to reduce the dimensions of the feature maps, and should thus be added after each convolutional layer. In this case, since *pool_size=(2,2)*, for every 2x2 pixel patch in the feature map, the pooling layer takes the maximum value and discards the rest. With this, the feature map size is reduced while still maintaining the stronger feature in each patch [54].

The fully connected layers behave like traditional neural networks. Unlike in the convolution layers where neurons only connect to a small receptive field, in the fully connected layers every neuron is connected to every neuron in the previous layer. The features are then combined and the final predictions are made [66].

Flatten() converts the 2D feature maps into a 1D vector so it can be fed into the fully connected layers. *Dense(64, activation='relu')* defines the first fully connected layer, which has 64 neurons. *Dense(1)* is the final output layer, having thus 1 neuron which correspond to the density prediction value.

The *model.compile()* function prepares the model for training by defining an optimizer and a loss function, which in this case are *Adam* and the MSE, respectively. These are standard for Convolutional Neural Network (CNN) regression tasks.

This model will perform weight adjustments via backpropagation. Taking the results from the loss function at the final layer, backpropagation works backwards through the network, propagating the error to each preceding layer. For each layer, the gradient of the loss is calculated with respect to the weights of that layer. This tells the model how much and in which direction the weights need to be adjusted to reduce the error [56].

The weights of the network are updated via the optimizer *Adam*. This optimizer adjusts the learning rate for each parameter dynamically, speeding up the training process.

After the proper model architecture is implemented, the training of the model should be performed (Listing 30).

Unlike the previous code, for the CNN model, the number of epochs and the batch size need to be defined. Batch size defines how many images are processed at a time before the model updates its weights via backpropagation [12].

```
model.fit(X_train, y_train, epochs=20, batch_size=2,  
↪ validation_data=(X_val, y_val))
```

Listing 30: Convolutional Neural Network - Model training

Since the number of images is only 14, a small batch size of *batch_size* = 2 could be chosen. In order to account for the small number of images, the number of epochs could be set to *epochs* = 20, meaning the model goes through all 14 images 20 times. Each epoch will have 7 batches, and on each of these, a weight update will be made. Thus, after 20 epochs, the model will have performed a total of $7 \text{ batches} \times 20 \text{ epochs} = 140$ backpropagation steps.

Once trained, the model can perform the density estimation calculation. In this case, the *model.predict()* method should be used for the prediction of the density estimations instead of the *rf_regressor.predict()* used in the Random Forest model.

Given this rough overview and recommendations, future research should focus on implementing this model and making any necessary adjustments to the Random Forest Regressor code in order for it to properly work on the problem at hand.

Aside from the benefit of keeping the spacial consistency of the estimation, CNNs introduce feature detection to the implemented model. In a way, it transforms a bi-dimensional problem into a multidimensional one, since now the density values are not only dependant on the pixel values but also on the features detected by the CNN.

The final result should be a smoother and more spatially coherent density estimation, and an optimized code which is capable of being scaled to larger datasets without losing efficiency.

5.3.2 Temporal Consistency

As was mentioned in Chapter 2.2.2, the flow within the body region of a density current can be steady (or quasi-steady) or unsteady, depending on the way it has been initiated. Since for the lock-exchange experiment the current is generated by a short-duration release of the dense fluid, the flow within the body region will be unsteady. This means that the variables governing the flow are both space and time dependant. A precise density estimation model must take this into account.

Even with a CNN model, each image is processed independently, and there is no consideration for temporal consistency between frames. By not considering the temporal sequence, the model misses out on the contextual information that could help in smoothing out transitions and maintaining a more coherent and stable output over time.

To solve this issue, the most important aspect to take into account is that a time dependant dataset would have to be created, and the calibration process done differently. Assuming that the fourteen steps for the calibration methodology described in Chapter 3.4 are kept, Table 5.1 shows the density values that each of the fourteen images represent, from densest to less dense.

Table 5.1: Density values for each of the fourteen calibration images

Density	Values (kg/m^3)
ρ_1	1022.20
ρ_2	1009.40
ρ_3	1002.00
ρ_4	997.24
ρ_5	993.55
ρ_6	991.12
ρ_7	989.30
ρ_8	988.24
ρ_9	987.28
ρ_{10}	986.62
ρ_{11}	986.47
ρ_{12}	986.42
ρ_{13}	986.32
ρ_{14}	985.92

In order to create a time sensitive dataset, the data acquisition processes would need to have the following steps:

1. Fill the entire lock-exchange channel with the solution corresponding to ρ_{14} ;
2. Map RGB_{14} to ρ_{14} ;
3. Remove the solution from the channel;
4. Close the gate of the channel and seal it with Vaseline;
5. Refill the channel after the gate with the previously removed solution;
6. At the same time, fill the channel before the gate with the solution corresponding to ρ_{13} ;
7. Release the gate of the channel;
8. Record the resulting density current.

By doing this, the pixel intensity values could be directly measured as a function of time. For the density, since the fixed values of ρ_{14} and ρ_{13} are the only known variables, and the transition values can't be measured directly, a CNN model could still be used to predict density values based of pixel intensity.

With the transition from ρ_{14} to ρ_{13} recorded, and the CNN model implemented, it would be possible to create a dataset like the one in Table 5.2.

Table 5.2: Updated data calibration, with time as a variable

Time (t)	Pixel (i, j)	Pixel Intensity (RGB)	Density (kg/m^3)
t_i	(i_i, j_i)	RGB_{14}	ρ_{14}
...	...	...	...
t_x	(i_x, j_x)	RGB_x	ρ_x
...	...	...	...
t_f	(i_f, j_f)	RGB_{13}	ρ_{13}

This process should be repeated for the thirteen density transition creating a dataset which has density values as a function of both pixel intensity and time. While effectively creating this dataset might be challenging, the gathering and preparation of good quality data is arguably the most important part of any model implementation. Building such a dataset is essential for developing a model which takes into account the time dependency of the density current flow.

With this setup, the calibration data would be a series of density currents instead of a collection of fixed images, which lack contextual information about time and space.

After having such a dataset, applying Recurrent Neural Networks (RRNs) or Long Short-Term Memory (LSTM) networks can be highly beneficial. RRNs or LSTM networks are designed to handle sequential data by maintaining a hidden state that carries information from previous time steps to the current one [25].

Just as CNNs can learn spatial features through their convolutional layers, RRNs can also learn temporal dependencies between sequential inputs through their recurring layers. Both networks can thus perform feature learning, unlike Random Forests [21].

RRNs also use backpropagation to adjust weights, although these are of a special kind called Backpropagation Through Time (BPTT). BPTT adjusts weights time-step by time-step moving backward through the sequence of time. In order for the backpropagation to be performed, a loss function and an optimizer also have to be defined, along with batch size and number of epochs during the training process [29, 89].

It's important to notice that the CNN model itself could be improved with this new dataset, since it would have more spatially relevant data about the density current.

With the approach suggested, both a CNN and a Recurrent Neural Network (RRN) would be implemented simultaneously in order to capture both spatial and temporal dependencies for the density estimation. This general overview of how they both can be implemented should serve as a starting point for further improvements of the density estimation code.

CONCLUSION

This dissertation has explored various aspects of fluid mechanics and machine learning, specifically focusing on the computational aspect of the lock-exchange experiment. The present study aimed to build upon the work done by Alves [5] and to enhance density current analysis through machine learning and image processing techniques.

The key objectives of this dissertation were successfully achieved:

- A study of density currents was conducted, elucidating their significance in fluid mechanics;
- A thorough investigation of machine learning techniques took place in order to understand their applicability to the problem at hand;
- The distortion correction and density estimation Python codes were successfully developed, setting the stage for further computational enhancements;
- A Random Forest Regressor was implemented in order to improve the density estimation code.

Additionally, while constraints limited the implementation of sophisticated deep learning models, useful guides for their future development were provided.

The findings of this dissertation have significant implications for the field of fluid mechanics. By developing the codes in Python and enhancing the existing computational tools, the research contributes to more accurate and robust methods for studying density currents. The integration of machine learning techniques demonstrates the potential for advanced computational methods to revolutionize traditional fluid mechanics research.

Building on the work presented in this dissertation, several avenues for future research can be recommended:

- **Improve distortion correction code:** There were no machine learning improvements done to the distortion correction code. Future works could focus on implementing a few changes to this code based on the research done in this dissertation. Machine learning algorithms, such as Convolutional Neural Networks (CNNs), can be trained to identify and correct distortions in images by leveraging large datasets of distorted and corrected images;
- **Improve density estimation code translation:** Although the results were not unreasonable, there are still several issues that could be tackled in order to improve the translation of the density estimation code. It is important to investigate the reasons behind the failure of the gaussian filter in the initial analysis. This could involve examining the filter parameters, the nature of the fluid flow data, and any steps in the code that may have affected the filter's performance. Understanding these factors will help refine the filtering process and improve overall data quality. Another relevant aspect are the discrepancies between the vertical density progressions in Alves' dissertation and the present one. The graphs having different curves points to some issue in the translation which is yielding different results. A more thorough examination of the original code might be useful in discovering this particular issue. Additionally, although the present results were compared to the ones obtained by Alves, it's important to conduct experiments to further validate these results and analyze which ones more accurately represent reality and why;
- **Validation of improved density estimation results:** Efforts should be made to validate the results obtained by the Random Forest model implemented in Chapter 5.2.2. This could be done by fine tuning the model, by training it on different datasets or by implementing other machine learning techniques and comparing the results through an analysis of their Mean Squared Error (MSE);
- **Gathering of data:** The gathering of more extensive calibration data would be extremely helpful for the implementation of more complex deep learning models. This could be done by real world experiments and setups, like the one suggested in Chapter 5.3.2, by doing research on pre-trained models or even by methods of generating new synthetic data. As was mentioned in Chapter 2.6.3, Generative Adversarial Networks (GANs) are able to generate synthetic images which can serve to augment already available data;

- Implement suggested developments to the density estimation code: Chapter 5.3, outlines two deep learning techniques, CNNs and Recurrent Neural Networks (RRNs), that can improve the density estimation by maintaining spatial and temporal consistency. Future works should focus on creating the required datasets for their application, implementing the models and comparing their results with the Random Forest Regressor;
- Further research: Continuing research efforts should explore additional fluid mechanics phenomena and computational techniques. This can include studying different types of density currents, investigating new machine learning models, and developing novel methodologies for fluid flow analysis.

In conclusion, this dissertation was successful in it's goal of improving the analysis of density currents through the use of machine learning and image processing techniques.

While exploring the intersection of machine learning with fluid mechanics, the study has built upon existing knowledge and suggests possible directions for future research. The completion of this dissertation represents a modest step forward in the field, hopefully setting the stage for further research and development.

BIBLIOGRAPHY

- [1] M. Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. 2015. URL: www.tensorflow.org. (cit. on p. 23).
- [2] N. F. G. S. et al. *The structure of gravity currents propagating in finite domains Civil Engineering*. 2017 (cit. on pp. 7, 8).
- [3] T. B. B. et al. "Language Models are Few-Shot Learners". In: (2020-05). URL: <http://arxiv.org/abs/2005.14165> (cit. on p. 16).
- [4] A. Alsaleh and C. Perkgoz. "A space and time efficient convolutional neural network for age group estimation from facial images". In: *PeerJ Computer Science* 9 (2023). ISSN: 23765992. DOI: 10.7717/PEERJ-CS.1395 (cit. on p. 21).
- [5] C. C. W. Alves. *Experimental Study of Density Currents Within Array of Emergent Cylinders*. 2022 (cit. on pp. xi, xiii, 1, 8, 9, 27–33, 40, 48, 58, 61, 81).
- [6] R. Arora et al. "Understanding Deep Neural Networks with Rectified Linear Units". In: (2016-11). URL: <http://arxiv.org/abs/1611.01491> (cit. on p. 77).
- [7] D. H. Ballard and C. M. Brown. *Computer Vision*. 1982 (cit. on pp. 16, 20, 21).
- [8] C. M. Bishop. *Pattern Recognition and Machine Learning*. 2006 (cit. on pp. 19, 20, 22).
- [9] A. Borji. "Generated Faces in the Wild: Quantitative Comparison of Stable Diffusion, Midjourney and DALL-E 2". In: (2023-10). URL: <http://arxiv.org/abs/2210.00586> (cit. on p. 16).
- [10] L. Breiman. *Random Forests*. 2001 (cit. on p. 64).
- [11] A. Brital. *Random Forest Algorithm Explained*. 2021 (cit. on p. 64).
- [12] J. Brownlee. *What is the Difference Between a Batch and an Epoch in a Neural Network?* 2018. URL: <https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/> (cit. on pp. 19, 77).
- [13] S. L. Brunton, B. R. Noack, and P. Koumoutsakos. "Machine Learning for Fluid Mechanics". In: *Annu. Rev. Fluid Mech.* 2020 52 (2019), pp. 477–508. DOI: 10.1146/annurev-fluid-010719. URL: <https://doi.org/10.1146/annurev-fluid-010719> (cit. on pp. 17, 24).

- [14] F. Chollet. *Deep Learning with Python*. Manning Publications Company, 2017, 2017 (cit. on p. 24).
- [15] C. Cortes, V. Vapnik, and L. Saitta. *Support-Vector Networks* Editor. 1995 (cit. on p. 15).
- [16] G. J. Daviero, P. J. W. Roberts, and K. Maile. *Refractive index matching in large-scale stratified experiments*. 2001 (cit. on p. 29).
- [17] D. Ferrucci et al. *Watson: Beyond jeopardy!* 2013. DOI: 10.1016/j.artint.2012.06.009 (cit. on p. 15).
- [18] *Gaussian Smoothing*. URL: <https://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm> (cit. on p. 11).
- [19] S. Geman, E. Bienenstock, and R. Doursat. *VIEW Communicated by Lawrence Jackel Neural Networks and the Bias-Variance Dilemma*. 1992 (cit. on p. 19).
- [20] R. C. Gonzalez and R. E. (E. Woods. *Digital image processing*. 2018, p. 1168. ISBN: 9780133356724 (cit. on p. 11).
- [21] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. <http://www.deeplearningbook.org>. MIT Press, 2016 (cit. on p. 80).
- [22] I. Goodfellow et al. "Generative adversarial networks". In: *Communications of the ACM* 63 (11 2020-10), pp. 139–144. ISSN: 15577317. DOI: 10.1145/3422622 (cit. on p. 22).
- [23] I. A. Hannoun. "Matching the Refractive Index in Density Stratified Flows". In: 1985. URL: <https://api.semanticscholar.org/CorpusID:92890680> (cit. on p. 29).
- [24] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning Data Mining, Inference, and Prediction*. 2009 (cit. on pp. 18, 64).
- [25] S. Hochreiter and J. ". U. Schmidhuber. *Long Short-Term Memory*. 1997 (cit. on p. 80).
- [26] T. O. Hodson. *Root-mean-square error (RMSE) or mean absolute error (MAE): when to use them or not*. 2022-07. DOI: 10.5194/gmd-15-5481-2022 (cit. on p. 70).
- [27] J. Hunter and D. Dale. *The Matplotlib User's Guide*. 2007 (cit. on p. 14).
- [28] H. E. Huppert and J. E. Simpson. *The slumping of gravity currents*. 1980 (cit. on p. 9).
- [29] H. Jaeger and H. Jaeger. *A tutorial on training recurrent neural networks, covering BPTT, RTRL, EKF and the "echo state network" approach*. 2002 (cit. on pp. 22, 80).
- [30] D. Jakhar and I. Kaur. *Artificial intelligence, machine learning and deep learning: definitions and differences*. 2020-01. DOI: 10.1111/ced.14029 (cit. on p. 17).
- [31] G. James et al. *An Introduction to Statistical Learning*. 2023 (cit. on p. 69).
- [32] G. James et al. *Springer Texts in Statistics*. 2021. URL: <http://www.springer.com/series/417> (cit. on p. 70).

- [45] K. M. Mok, C. S. Leong, and K. I. Hoi. *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on : date, 27-29 May, 2011*. IEEE, 2011. ISBN: 9781612844862 (cit. on p. 10).
- [46] *Morphological Image Processing*. URL: <https://www.cs.auckland.ac.nz/courses/compsci773s1c/lectures/ImageProcessing-html/topic4.htm> (cit. on p. 11).
- [47] H. I. d. S. Nogueira. *Experimental Chacaracterization of Unsteady Gravity Currents Developing Over Smooth and Rough Beds*. 2013 (cit. on pp. 5–7).
- [48] T. E. Oliphant. *Guide to NumPy*. 2006. URL: <http://www.trelgol.com> (cit. on p. 12).
- [49] *OpenCV - About*. URL: <https://opencv.org/about/> (cit. on p. 13).
- [50] Z. Pan et al. “Recent Progress on Generative Adversarial Networks (GANs): A Survey”. In: *IEEE Access* 7 (2019), pp. 36322–36333. ISSN: 21693536. DOI: 10.1109/ACCESS.2019.2905015 (cit. on p. 22).
- [51] B. Pang, E. Nijkamp, and Y. N. Wu. *Deep Learning With TensorFlow: A Review*. 2020-04. DOI: 10.3102/1076998619872761 (cit. on p. 23).
- [52] A. Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019 (cit. on p. 23).
- [53] J. Rabault et al. “Artificial Neural Networks Trained Through Deep Reinforcement Learning Discover Control Strategies for Active Flow Control”. In: *Journal of Fluid Mechanics* 865 (2019-04), pp. 281–302. ISSN: 14697645. DOI: 10.1017/jfm.2019.62 (cit. on p. 24).
- [54] A. Ranzato et al. *Unsupervised Learning of Invariant Feature Hierarchies with Applications to Object Recognition*. 2007. URL: <http://www.cs.nyu.edu/%E2%88%BCyann> (cit. on p. 77).
- [55] S. Raschka. *Understanding Large Language Models - A Transformative Reading List*. 2023. URL: <https://sebastianraschka.com/blog/2023/llm-reading-list.html> (visited on 2023-02-07) (cit. on p. 16).
- [56] W. Rawat and Z. Wang. *Deep convolutional neural networks for image classification: A comprehensive review*. 2017-09. DOI: 10.1162/NECO_a_00990 (cit. on p. 77).
- [57] F. Rosenblatt. *The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain*. 1958 (cit. on pp. 14, 16).
- [58] J. W. Rottman and J. E. Simpson. *Gravity currents produced by instantaneous releases of a heavy fluid in a rectangular channel*. 1983 (cit. on p. 9).
- [59] D. E. Rumelhart. “Learning Representations by Back-Propagation Errors”. In: (1987) (cit. on pp. 15, 18).
- [60] A. L. Samuel. *Some Studies in Machine Learning Using the Game of Checkers. II-Recent Progress*. 1967 (cit. on p. 15).
- [61] B. Schölkopf et al. *Support Vector Machines and Kernel Algorithms*. 2002 (cit. on p. 15).

-
- [62] O. E. Sequeiros and et al. "Characteristics of Velocity and Excess Density Profiles of Saline Underflows and Turbidity Currents Flowing over a Mobile Bed Modeling". In: *Article in Journal of Hydraulic Engineering* (2010). URL: <https://www.researchgate.net/publication/259299694> (cit. on p. 8).
 - [63] M. Sezgin and B. Sankur. "Survey over image thresholding techniques and quantitative performance evaluation". In: *Journal of Electronic Imaging* 13 (1 2004-01), p. 146. ISSN: 1017-9909. DOI: 10.1117/1.1631315 (cit. on p. 11).
 - [64] J. O. Shin, S. B. Dalziel, and P. F. Linden. "Gravity currents produced by lock exchange". In: *Journal of Fluid Mechanics* 521 (2004-12), pp. 1–34. ISSN: 00221120. DOI: 10.1017/S002211200400165X (cit. on p. 10).
 - [65] D. Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529 (7587 2016-01), pp. 484–489. ISSN: 14764687. DOI: 10.1038/nature16961 (cit. on p. 15).
 - [66] K. Simonyan and A. Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: (2014-09). URL: <http://arxiv.org/abs/1409.1556> (cit. on p. 77).
 - [67] J. E. Simpson. *Gravity Currents in the Laboratory, Atmosphere and Ocean*. 1982 (cit. on pp. 5, 7).
 - [68] J. E. Simpson. *Gravity Currents: In the Environment and the Laboratory*. 2nd Edition. Cambridge University Press, 1997 (cit. on p. 8).
 - [69] L. N. Smith. "Cyclical Learning Rates for Training Neural Networks". In: (2017-06). URL: <http://arxiv.org/abs/1506.01186> (cit. on p. 19).
 - [70] W. E. Snyder and H. Qi. *Fundamentals of Computer Vision*. Cambridge University Press, 2017 (cit. on p. 11).
 - [71] M. Sokolova and G. Lapalme. "A systematic analysis of performance measures for classification tasks". In: *Information Processing and Management* 45 (4 2009-07), pp. 427–437. ISSN: 03064573. DOI: 10.1016/j.ipm.2009.03.002 (cit. on p. 19).
 - [72] Z. Strout. *MatLab vs Python*. 2023. URL: <https://www.sagemotion.com/blog/matlab-vs-python> (cit. on p. 61).
 - [73] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. 2018, p. 526. ISBN: 9780262039246 (cit. on pp. 18, 20, 22).
 - [74] R. Szeliski. *Computer Vision: Algorithms and Applications 2nd Edition*. 2022. URL: <https://szeliski.org/Book> (cit. on pp. 11, 12).
 - [75] A. M. Turing. *Computing Machinery and Intelligence*. 1950. URL: <https://academic.oup.com/mind/article/LIX/236/433/986238> (cit. on p. 14).
 - [76] M. Ungarish. *An Introduction to Gravity Currents and Intrusions*. Chapman and Hall/CRC, 2009-04. DOI: 10.1201/9781584889045 (cit. on pp. 5, 8, 9).

- [77] Unknown. *Artificial Neural Networks and its Applications*. 2024. URL: [rtificial%20Neural%20Networks%20and%20its%20Applications](#) (cit. on p. 20).
- [78] Unknown. *Camera Calibration*. URL: https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html (cit. on pp. 35, 38).
- [79] Unknown. *Camera Calibration and 3D Reconstruction*. URL: https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html (cit. on p. 38).
- [80] Unknown. *Finding Corners with SubPixel Accuracy*. URL: <https://theailearner.com/tag/cv2-cornersubpix/> (cit. on p. 42).
- [81] Unknown. *K-Means Clustering in OpenCV*. URL: [K-Means%20Clustering%20in%20OpenCV](#) (cit. on p. 41).
- [82] Unknown. *Normalized Root Mean Squared Error*. 2023. URL: <https://computer-nerd-zettelkasten.tistory.com/entry/NRMSE-Normalized-Root-Mean-Squared-Error> (cit. on p. 71).
- [83] Unknown. *Python Glob*. URL: <https://docs.python.org/3/library/glob.html> (cit. on p. 39).
- [84] Unknown. *Python OS*. URL: <https://docs.python.org/3/library/os.html> (cit. on p. 39).
- [85] Unknown. *TensorFlow - Convolutional Neural Network (CNN)*. URL: <https://www.tensorflow.org/tutorials/images/cnn> (cit. on p. 75).
- [86] A. Vaswani et al. "Attention Is All You Need". In: (2017-06). URL: <http://arxiv.org/abs/1706.03762> (cit. on p. 16).
- [87] P. Viola and M. Jones. *Rapid Object Detection using a Boosted Cascade of Simple Features*. 2001 (cit. on p. 16).
- [88] S. V. D. Walt et al. "Scikit-image: Image Processing in Python". In: *PeerJ* 2014 (1 2014). ISSN: 21678359. DOI: 10.7717/peerj.453 (cit. on p. 14).
- [89] P. J. Werbos. *Backpropagation Through Time: What It Does and How to Do It*. 1990 (cit. on p. 80).
- [90] F. M. White. *Fluid Mechanics*. 2009 (cit. on p. 1).
- [91] M. D. Zeiler and R. Fergus. *LNCS 8689 - Visualizing and Understanding Convolutional Networks*. 2014 (cit. on p. 75).

DISTORTION CORRECTION CODE

```
import os
import cv2
import numpy as np
import glob

# Path to calibration images
calibration_pattern = 'path_to_calibration_images*.bmp'

# Path to images to undistort
images_to_undistort_pattern = 'path_to_images_to_undistort*.bmp'

# Path to save undistorted images
save_directory = 'path_to_save_undistorted_images'

# Chessboard size (number of inner corners)
chessboard_size = (9, 78)

# Arrays to store object points and image points
obj_points = [] # 3D points in real-world space
img_points = [] # 2D points in image plane

# Prepare object points, like (0,0,0), (1,0,0), (2,0,0), ..., (8,5,0)
objp = np.zeros((chessboard_size[0] * chessboard_size[1], 3), np.float32)
objp[:, :2] = np.mgrid[0:chessboard_size[0],
↪ 0:chessboard_size[1]].T.reshape(-1, 2)

# Glob to get all file names that match the pattern
calibration_images = glob.glob(calibration_pattern)
```

```
# Loop through calibration images
for calibration_image_path in calibration_images:
    calibration_image = cv2.imread(calibration_image_path)
    gray_calibration = cv2.cvtColor(calibration_image,
    ↪ cv2.COLOR_BGR2GRAY)

    # Find chessboard corners and refine them
    ret, corners = cv2.findChessboardCorners(gray_calibration,
    ↪ chessboard_size, None)
    if ret:
        # Refine corners
        criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER,
        ↪ 30, 0.001)
        corners = cv2.cornerSubPix(gray_calibration, corners, (11, 11),
        ↪ (-1, -1), criteria)

        obj_points.append(objp)
        img_points.append(corners)

        cv2.drawChessboardCorners(calibration_image, chessboard_size,
        ↪ corners, ret)
        cv2.imshow('Chessboard Corners', calibration_image)
        cv2.waitKey(500) # Wait for a short duration to display each
        ↪ image

cv2.destroyAllWindows()

# Calibrate the camera using all detected corners
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(obj_points,
    ↪ img_points, gray_calibration.shape[::-1], None, None)

# Set k1, k2, and k3 to zero
dist[0, 0] = 0.0 # k1
dist[0, 1] = 0.0 # k2
dist[0, 4] = 0.0 # k3

# Display the updated camera matrix and distortion coefficients
print("Updated Camera Matrix:")
print(mtx)
```

```
print("\nUpdated Distortion Coefficients:")
print(dist)

# Glob to get all file names of images to undistort
images_to_undistort = glob.glob(images_to_undistort_pattern)

# Loop through each image and undistort
for image_path in images_to_undistort:
    image_to_undistort = cv2.imread(image_path)
    undistorted_image = cv2.undistort(image_to_undistort, mtx, dist,
    ↪ None, mtx)

    # Save the undistorted image
    base_name = os.path.basename(image_path)
    save_path = os.path.join(save_directory, f'undistorted_{base_name}')
    cv2.imwrite(save_path, undistorted_image)

    # Display the original and undistorted images
    cv2.imshow('Original Image', image_to_undistort)
    cv2.imshow('Undistorted Image', undistorted_image)
    cv2.waitKey(500) # Wait for a short duration to display each image

cv2.destroyAllWindows()
```

Listing 31: Distortion Correction - Full Code

DENSITY ESTIMATION CODE: INTERPOLATION

```
import os
import cv2
import numpy as np
from scipy.interpolate import interp1d
from natsort import natsorted
from matplotlib import pyplot as plt
from scipy.io import loadmat

# Input information
ny = 18
Ly = 141
nx = 42
Lx = 1024
Lx_m = 1.6
Ly_m = 0.20
scale_x = Lx_m / Lx
scale_y = Ly_m / Ly
deltaT = 1 / (1100 / 25)
main_path = 'path_to_undistorted_images'

# Load calibration data
calib_data = loadmat('path_to_calibration_data.mat')
pixel = calib_data['pixel'][ny:Ly, nx:Lx, :][::-1, :]
rho_values = calib_data['Rho']

# Visualize calibration images
```

```
for w in range(0, 13):

    calibration_images = pixel[:, :, w]

    plt.figure(figsize=(10, 5))
    plt.imshow(calibration_images, cmap='viridis', aspect='auto')
    plt.colorbar(label='Pixel Intensity')
    plt.gca().set_aspect('equal')
    plt.title(f'Calibration Image {w+1}')
    plt.xlabel('Width (pixels)')
    plt.ylabel('Height (pixels)')
    plt.show()

# Loop through experiment folders
for r in range(1, 3):
    # Read images
    files = natsorted(os.listdir(os.path.join(main_path, f'R{r}')))

    # Create video files and features
    vidObj =
    ↪ cv2.VideoWriter(f'path_to_results_folder/R{r}/Current_R{r}.avi',
    ↪ cv2.VideoWriter_fourcc(*'DIVX'), 4, (1024, 256))
    vidraw =
    ↪ cv2.VideoWriter(f'path_to_results_folder/R{r}/Rawimages_R{r}.avi',
    ↪ cv2.VideoWriter_fourcc(*'DIVX'), 4, (1024, 256))

    for k, filename in enumerate(files, start=1):
        time = k * deltaT
        A = cv2.imread(os.path.join(main_path, f'R{r}', filename),
            ↪ cv2.IMREAD_GRAYSCALE)[(256-ny-(Ly-ny)):(256-ny), nx:Lx]

        density = np.zeros_like(A, dtype=float)[::-1, :]

        for j in range(A.shape[1]):
            for i in range(A.shape[0]):
                CAL = pixel[i, j, :]
                RGB = A[i, j]

                # Ensure RGB value is within calibration range
                RGB = min(max(RGB, CAL.min()), CAL.max())
```

```

        # print (CAL)

        # Linear interpolation to find density value
        interp_func = interp1d(CAL, rho_values[0,:],
        ↪ kind='linear', fill_value="interpolation")
        density[i, j] = interp_func(RGB)

if k % 4 == 0:
    # Plot density image
    plt.figure(figsize=(14, 4.5))
    plt.pcolor(density[:, :-1, :], cmap='jet', vmin=985, vmax=1025)
    plt.colorbar(label='Density (kg/m^3)')
    plt.gca().set_aspect('equal')
    x_ticks = np.arange(0, (Lx-nx) * scale_x, 0.2)
    y_ticks = np.arange(0, (Ly-ny) * scale_y, 0.05)
    plt.xticks(np.arange(0, (Lx-nx), 0.2 / scale_x),
    ↪ np.round(x_ticks, 2))
    plt.yticks(np.arange(0, (Ly-ny), 0.05 / scale_y),
    ↪ np.round(y_ticks, 2))
    plt.xlabel('$x \mathrm{(m)}$', fontsize=16)
    plt.ylabel('$z \mathrm{(m)}$', fontsize=16)
    plt.title(f't={time:.2f}s', fontsize=14)
    plt.savefig(f'path_to_results_folder/R{r}/Image_{k}.png')
    plt.close()

    # Plot raw image
    plt.figure(figsize=(14, 4.5))
    plt.pcolor(A[:, :-1, :], cmap='viridis')
    plt.colorbar(label='Pixel Intensity')
    plt.gca().set_aspect('equal')
    plt.xticks(np.arange(0, (Lx-nx), 0.2 / scale_x),
    ↪ np.round(x_ticks, 2))
    plt.yticks(np.arange(0, (Ly-ny), 0.05 / scale_y),
    ↪ np.round(y_ticks, 2))
    plt.xlabel('$x \mathrm{(m)}$', fontsize=16)
    plt.ylabel('$z \mathrm{(m)}$', fontsize=16)
    plt.title(f't={time:.2f}s', fontsize=14)
    plt.savefig(f'path_to_results_folder/R{r}/RawImage_{k}.png')

```

```
plt.close()

# Write frames to video
vidObj.write(cv2.cvtColor(A, cv2.COLOR_GRAY2BGR))
vidraw.write(cv2.cvtColor(density.astype(np.uint8),
    ↪ cv2.COLOR_GRAY2BGR))

print(k)

vidObj.release()
vidraw.release()
```

Listing 32: Density Estimation - Full Code

DENSITY ESTIMATION CODE: RANDOM FOREST

```
import os
import cv2
import numpy as np
from natsort import natsorted
from matplotlib import pyplot as plt
from scipy.io import loadmat
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# Input information
ny = 18
Ly = 141
nx = 42
Lx = 1020
Lx_m = 1.6
Ly_m = 0.20
scale_x = Lx_m / Lx
scale_y = Ly_m / Ly
deltaT = 1 / (1100 / 25)
main_path = 'path_to_undistorted_images'

# Load calibration data
calib_data = loadmat('path_to_calibration_data.mat')
pixel = calib_data['pixel'][ny:Ly, nx:Lx, :][::-1, :]
rho_values = calib_data['Rho']
```

```
# Prepare training data for Random Forest
X_train = []
y_train = []
for i in range(pixel.shape[0]):
    for j in range(pixel.shape[1]):
        for k in range(pixel.shape[2]):
            X_train.append(pixel[i, j, k])
            y_train.append(rho_values[0, k])
X_train = np.array(X_train).reshape(-1, 1)
y_train = np.array(y_train)

# Split into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
    ↪ test_size=0.2, random_state=42)

# Train Random Forest Regressor
rf_regressor = RandomForestRegressor(n_estimators=100, random_state=42)
rf_regressor.fit(X_train, y_train)

# Evaluate the model
val_predictions = rf_regressor.predict(X_val)
mse = mean_squared_error(y_val, val_predictions)
print(f'Validation MSE: {mse}')

# Process current images
for r in range(1, 3):
    # Read images
    files = natsorted(os.listdir(os.path.join(main_path, f'R{r}'))))

    # Create video files and features
    vidObj =
    ↪ cv2.VideoWriter(f'path_to_results_folder/R{r}/Current_R{r}.avi',
                    cv2.VideoWriter_fourcc(*'DIVX'), 4, (1024,
    ↪ 256))

    vidraw =
    ↪ cv2.VideoWriter(f'path_to_results_folder/R{r}/Rawimages_R{r}.avi',
                    cv2.VideoWriter_fourcc(*'DIVX'), 4, (1024,
    ↪ 256))
```

```

for k, filename in enumerate(files, start=1):
    time = k * deltaT
    A = cv2.imread(os.path.join(main_path, f'R{r}', filename),
        ↪ cv2.IMREAD_GRAYSCALE)[(256-ny-(Ly-ny)):(256-ny), nx:Lx]

    density = np.zeros_like(A, dtype=float)[::-1, :]

    for j in range(A.shape[1]):
        for i in range(A.shape[0]):
            RGB = A[i, j]
            RGB = min(max(RGB, X_train.min()), X_train.max())
            density[i, j] = rf_regressor.predict([[RGB]])[0]

    if k % 4 == 0:
        # Plot density image
        plt.figure(figsize=(14, 4.5))
        plt.pcolor(density[::-1, :], cmap='jet', vmin=985, vmax=1025)
        plt.colorbar(label='Density (kg/m^3)')
        plt.gca().set_aspect('equal')

        # Update axis ticks and labels to meters
        x_ticks = np.arange(0, (Lx-nx) * scale_x, 0.2)
        y_ticks = np.arange(0, (Ly-ny) * scale_y, 0.05)
        plt.xticks(np.arange(0, (Lx-nx), 0.2 / scale_x),
            ↪ np.round(x_ticks, 2))
        plt.yticks(np.arange(0, (Ly-ny), 0.05 / scale_y),
            ↪ np.round(y_ticks, 2))
        plt.xlabel('$x$ \mathrm{(m)}$', fontsize=16)
        plt.ylabel('$z$ \mathrm{(m)}$', fontsize=16)
        plt.title(f't={time:.2f}s', fontsize=14)
        plt.savefig(f'path_to_results_folder/R{r}/Image_{k}.png')
        plt.close()

        # Plot raw image
        plt.figure(figsize=(14, 4.5))
        plt.pcolor(A[::-1, :], cmap='viridis')
        plt.colorbar(label='Pixel Intensity')
        plt.gca().set_aspect('equal')

        # Update axis ticks and labels to meters

```

```
plt.xticks(np.arange(0, (Lx-nx), 0.2 / scale_x),
            ↪ np.round(x_ticks, 2))
plt.yticks(np.arange(0, (Ly-ny), 0.05 / scale_y),
            ↪ np.round(y_ticks, 2))
plt.xlabel('$x \mathrm{(m)}$', fontsize=16)
plt.ylabel('$z \mathrm{(m)}$', fontsize=16)
plt.title(f't={time:.2f}s', fontsize=14)
plt.savefig(f'path_to_results_folder/R{r}/RawImage_{k}.png')
plt.close()

# Write frames to video
vidObj.write(cv2.cvtColor(A, cv2.COLOR_GRAY2BGR))
vidraw.write(cv2.cvtColor(density.astype(np.uint8),
                        ↪ cv2.COLOR_GRAY2BGR))

print(k)

vidObj.release()
vidraw.release()
```

Listing 33: Random Forest - Full Code



2024 Enhancing Density Current Analysis Through Machine Learning and Image Processing Techniques Eman Martin