TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

Thesis no: M-59-MSMDE-2020-2023

Development of a Neural Network to Predict Path of an Object in a Two Dimensional Potential Flow

by

Rijan Niraula

A THESIS
SUBMITTED TO THE DEPARTMENT OF MECHANICAL AND AEROSPACE ENGINEERING
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE IN
MECHANICAL SYSTEMS DESIGN AND ENGINEERING

DEPARTMENT OF MECHANICAL AND AEROSPACE ENGINEERING
LALITPUR, NEPAL

APRIL, 2023

COPYRIGHT

The author has agreed that the library, Department of Mechanical and Aerospace Engineering, Pulchowk Campus, Institute of Engineering may make this thesis freely available for inspection. Moreover, the author has agreed that permission for extensive copying of this thesis for scholarly purpose may be granted by the professor(s) who supervised the work recorded herein or, in their absence, by the Head of the Department wherein the thesis was done. It is understood that the recognition will be given to the author of this thesis and to the Department of Mechanical and Aerospace Engineering, Pulchowk Campus, Institute of Engineering in any use of the material of the thesis. Copying or publication or the other use of this thesis for financial gain without approval of the Department of Mechanical and Aerospace Engineering, Pulchowk Campus, Institute of Engineering and author's written permission is prohibited.

Request for permission to copy or to make any other use of the material in this thesis in whole or in part should be addressed to:

Head
Department of Mechanical and Aerospace Engineering
Pulchowk Campus, Institute of Engineering
Lalitpur, Kathmandu
Nepal

TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
PULCHOWK CAMPUS
DEPARTMENT OF MECHANICAL ENGINEERING

The undersigned certify that they have read, and recommended to the Institute of Engineering for acceptance, a thesis entitled ”Development of a Neural Network to Predict Path of an Object in a Two Dimensional Potential Flow” submitted by Rijan Niraula in partial fulfillment of the requirements for the degree of Master of Science in Mechanical Systems Design and Engineering.

Supervisor, Dr. Laxman Poudel
Professor/Program coordinator (MSMDE)
Department of Mechanical and Aerospace Engineering

Supervisor, Dr. Rajendra Shrestha
Professor/Assistant Dean
Department of Mechanical and Aerospace Engineering

External Examiner, Er. Madan Timsina
Deputy Managing Director/Generation Directorate
Nepal Electricity Authority (NEA)

Committee Chairperson, Dr. Surya Prasad Adhikari
Assoc. Professor/Head of Department
Department of Mechanical and Aerospace Engineering

Date: April 10, 2023

ABSTRACT

Neural networks have been widely used in various fields, including fluid dynamics, to predict complex phenomena that are difficult to model analytically. In this research, a neural network is developed to predict the path taken by a circular body in a two-dimensional fluidic domain. The study involves simulating the potential flow over a rectangular domain inside which a circular body is placed. Fluctuations in different parameters such as pressure, forces, and velocity field during the motion of the body are studied. The Laplace equation is solved at each time step by applying the techniques of finite element method (FEM) to obtain accurate data, which is fed into the neural network. The neural network comprises of three layers input , middle and output layer. The study is carrried out using computational methods that relies on open-source software Python and its modules like NumPy. The results of the neural network's predictions are compared with accurate data to analyze the error. Fluctutaion of error with respect to different hyperparameters of the network is calculated and accordingly suitable hyperparameters of the network are determined.

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to Professor Dr. Laxman Poudel and Professor Dr. Rajendra Shrestha for supervising my thesis work. Without their invaluable advice, this research work would not have succeeded.

I also like to thank Pulchowk Campus for giving me the computational tools I required to complete my research.

I am also very thankful to Dr. Surya Adhikari, Head of the Department of Mechanical and Aerospace Engineering.

In addition, I want to sincerely thank my friend Abhishek Kafle for his assistance with my research. I want to thank my friends Krishna Bista, Spad Acharya, Dinesh Bhusal, and Ujjwal Dhakal, who helped me with this project in many ways.

I want to express my gratitude to everyone who has assisted me in this thesis work in different ways.

TABLE OF CONTENTS

Copyright . . . 2
Approval page . . . 3
Abstract . . . 4
Acknowledgement . . . 5
Table of Contents . . . 6
List of Tables . . . 9
List of Figures . . . 10
List of Symbols . . . 12
List of Acronyms . . . 13

1 INTRODUCTION **14**
1.1 Background . . . 14
1.2 Problem Statement . . . 15
1.3 Rationale . . . 16
1.4 Objectives . . . 17
1.4.1 Main objective . . . 17
1.4.2 Specific objectives . . . 17
1.5 Assumptions and Limitations . . . 17

2 LITERATURE REVIEW **18**
2.1 Common terminologies related to Fluid Flow . . . 18
2.1.1 Velocity Field . . . 18
2.1.2 Governing Equations . . . 19
2.1.3 Boundary Conditions . . . 19
2.2 Vector Calculus . . . 20
2.2.1 Potential function and theorem . . . 20
2.2.2 Laplace Equation . . . 20
2.2.3 Harmonic functions,properties and uniqueness . . . 20
2.3 Finite Element Method (FEM) . . . 21
2.3.1 Introduction and Methodology . . . 21

2.3.2 FEM and Laplace equation 23
2.3.2.1 Strong form 23
2.3.2.2 Weak form 23
2.3.2.3 Discretization 24
2.4 Neural Network 28
2.4.1 Introduction 28
2.4.2 Perceptron 30
2.4.3 Sigmoid Neurons 31
2.4.4 Architecture of Neural Network 33
2.4.5 Learning with Gradient Descent 34
2.5 Related Researches 36
2.6 Software Environment 38
2.6.1 Python 38
2.6.2 Python Modules 38
2.6.2.1 NumPy 38
2.6.2.2 Matplotlib 39

3 RESEARCH METHODOLOGY 41
3.1 Literature Review 41
3.2 Inputs and Mesh generation 42
3.2.1 Boundary Conditions and Initial Point 42
3.2.2 Create the flow domain 43
3.2.3 Generation of Mesh 44
3.3 Case Setup, Solution and Calculations 45
3.3.1 Case Setup and Solution 45
3.3.2 Calculations 45
3.3.2.1 Velocity Field Calculation 45
3.3.2.2 Pressure Calculation 45
3.3.2.3 Force Calculation 46
3.3.2.4 Trajectory Calculation 47
3.3.3 Store the Result 48
3.4 Development of Neural Network 49
3.5 Post Processing 50

3.6 Documentation/Conclusion 50

4 RESULTS AND DISCUSSION **51**

4.1 Development of the mesh 51

4.2 Potential Solution 53

4.3 Velocity Field 55

4.4 Path Trajectory 57

4.4.1 Via Calculation 57

4.4.2 Via Neural Network 58

4.5 Error Analysis and Comparison 59

4.5.1 Mean Square Error (MSE) 59

4.5.2 MSE vs Learning rate 60

4.5.3 MSE vs No of Neurons 62

4.5.4 MSE vs No of Dataset 64

5 CONCLUSIONS AND RECOMMENDATIONS **66**

5.1 Conclusions 66

5.2 Recommendations 67

References 68

Appendices 69

LIST OF TABLES

3.1 Methodology. 41

LIST OF FIGURES

1.1 Problem statement diagram 16
2.1 Velocity Field 18
2.2 Wall Boundary Condition 20
2.3 FEM Process 22
2.4 FEM Discretization 24
2.5 Triangular Element 25
2.6 Perceptron 30
2.7 Network of Perceptrons 31
2.8 Sigmoid Function 32
2.9 Example of a Network 33
2.10 Example of a Sophisticated Network 34
2.11 Double Variable Function 35
3.1 Platform for Input Condition in Sublime Text 42
3.2 Flow domain 43
3.3 Many flow domains 44
3.4 Mesh Region 44
3.5 Forces on the Body 46
3.6 Force Diagram 47
3.7 Net Force Diagram 47
4.1 Triangular meshes drawn in the region 51
4.2 Different meshes drawn in the region 52
4.3 Scatter plot of the solution 53
4.4 Scatter plot at various points 54
4.5 Velocity vector field 55
4.6 Velocity field at different object location 56
4.7 Path trajectory of the body 57
4.8 Path tajectory of the body by Network 58
4.9 MSE vs Learning Rate 60
4.10 MSE vs Learning Rate 61

4.11 MSE vs No of Neurons 62
4.12 MSE vs No of Neurons 63
4.13 MSE vs No of Dataset 64
4.14 MSE vs No of Dataset 65

LIST OF SYMBOLS

∇	Del Operator
$\nabla.$	Divergence Operator
$\nabla\times$	Curl Operator
∇^2	Laplacian Operator
$\mathbf{V}$	Fluid velocity
p	pressure
np	numpy
ρ	Density
t	time
$\vec{n}$	Unit Normal Vector
ϕ	Potential Function
$\frac{D}{Dt}$	Material Derivative

LIST OF ACRONYMS

CFD	Computational Fluid Dynamics
FDM	Finite Difference Method
PDE	Partial Differential Equation
FEM	Finite Element Method
NN	Neural Network
ML	Machine Learning
MSE	Mean Square Error

CHAPTER ONE: INTRODUCTION

1.1 Background

Since the time of the great Greek philosophers and scientists, people have been attempting to answer problems relating to fluid dynamics and its related mechanisms. As most problems in engineering deal with solving Partial Differential Equations (PDE), the first partial differential equation for fluid dynamics was created in 1752, by Euler, who focused primarily on inviscid fluids (hence, the name Euler equation for inviscid fluids). After that, Navier and Stokes independently added the viscous forces to the equation by using Newton's description of friction caused by velocity gradient and fluid viscosity; as a result, the equation is now known as the Navier-Stokes equation. There are several methods to solve such differential equations – Analytical, Experimental or Numerical methods. Apart from some simple flow problems, which can be solved using analytical methods, most of these problems are complicated and incredibly difficult to solve. In most cases, these equations cannot be solved analytically. On the other hand, there are experimental methods which are accurate and used quite often but it's limited and very expensive. Specialized facilities are required, that are expensive to develop and require high level of calibration and meticulous design. So, this method might not be feasible to explore new possibilities.

A third method involves the use of neural network to solve the PDE, which combines modern numerical techniques and high speed digital computers to solve the given physical problem. The importance and use of neural network has seen a substantial increase in different realms of modern technology be it digit recognition, speech recognition, data analysis and prediction of properties of physical system. The methods used are easier to set up, at low cost and offers a lot of flexibility. Computational methods like FEM, FDM transform complex differential equation into a set of algebraic equation, which then can be solved easily with help of computers. In this research we will be specifically solving laplace equation for a given set of boundary conditions in a two dimensional rectangular domain.

On a circular body in a two dimensional fluidic domain, the fluid exerts force and torque. The nature of the force and torque determine the nature of the path the body moves in. At any given instant, the presence of the body also contributes to the nature of flow, since it itself acts as a rigid boundary. Thus we need to solve the laplacian equation at each time step, since at every instant the nature of the problem is little bit different from its previous position. This obtained data is then fed into a neural network which then predicts the path of the body for a new starting point of the body.

1.2 Problem Statement

The significance of predicting the path the body takes, while interacting with fluid has been of paramount importance for a long time. Due to the unavailability of computational power we have today, the computational study of such a phenomenon was not possible a couple of decades ago. Furthermore, the experimental setup of such interaction also requires a large amount of cost and resources, and a high level of calibration, which is difficult to attain. A number of researches have been done in order to study this phenomenon since the development of computational power and advanced algorithms in recent times. However, a complete development of the system like neural network governing such phenomenon is still lacking

Consider a rigid circular body in a two dimensional flow in a rectangular domain as shown below in the figure. The flow is assumed to be incompressible and inviscid. If the boundary condition of the rectangular domain is specified, the fluid exerts a force and torque on the circular object. As a result of which the body moves in a trajectory within the flow. Also, notice that the presence of the body itself, is detrimental to the nature of the flow and thus the nature of force and torque acting on the body at each instant.

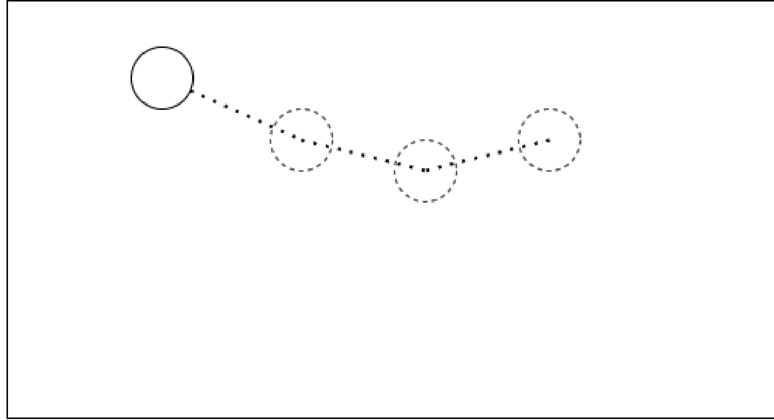

Figure 1.1: Problem statement diagram

1.3 Rationale

The main focus of this research is to increase our understanding of the impacts of the boundary condition and intial starting point on the potential flow of the body. The primary emphasis lies on the development of a neural network and how accurately it can predict when compared against accurate data. The study shall be done using computational methods that relies heavily in open source software like python and its module like numpy, matplotlib. This will advance our understanding of the impacts of different boundary condition and the subsequent nature of the flow of the body in two dimensional regime. Understanding the impacts of this phenomenon shall contribute to the design and faster computation time for similar problems. This is particularly helpful in situation where real time data is required for critical decision making process like automated driving etc.

1.4 Objectives

1.4.1 Main objective

The main objective of this research is to study and predict the various point of the path the body will take given the nature of the flow by developing a neural network.

1.4.2 Specific objectives

1. To simulate potential flow over a rectangular domain inside which a circular body is placed.
2. To study fluctuations in different parameters like pressure, forces, velcoity field during the motion of the body.
3. To figure out and analyze the error between the actual path and predicted path by the neural network .

1.5 Assumptions and Limitations

The assumptions and limitations of this research are listed as follows:

1. This research is carried out in a two-dimensional flow domain.
2. This research is limited to computational methods.
3. The properties of fluid remain constant.
4. Calculation is carried out under incompressible and inviscid assumptions.
5. The research can be further extended to the realm of three dimension and viscous flow.
6. Gradient descent algorithm is used for backpropagation purpose.

CHAPTER TWO: LITERATURE REVIEW

2.1 Common terminologies related to Fluid Flow

2.1.1 Velocity Field

The common method of describing fluid flow is the Eulerian description of fluid motion, named after the Swiss mathematician Leonhard Euler (1707–1783). In the Eulerian description of fluid flow, a finite volume called a flow domain or control volume is defined, through which fluid flows in and out. Instead of tracking individual fluid particles, we define field variables, functions of space and time, within the control volume. The field variable at a particular location at a particular time is the value of the variable for whichever fluid particle happens to occupy that location at that time. For example, the pressure field is a scalar field variable; for general unsteady two-dimensional fluid flow in Cartesian coordinates,

$$Pressure : p = p(x, y) \tag{2.1}$$

The velocity field vector is defined in similar fashion,

$$Velocity : V = \vec{V}(x, y)) \tag{2.2}$$

Figure 2.1 shows a typical example of two dimensional velocity field.

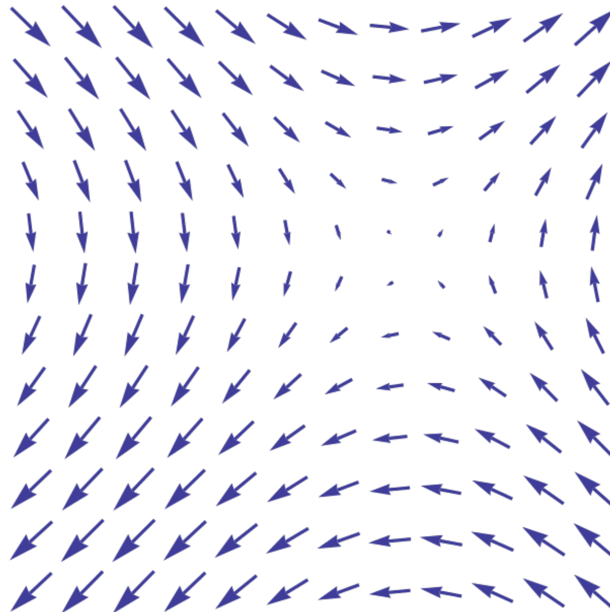

Figure 2.1: Velocity Field

2.1.2 Governing Equations

The basic governing equations that governs basic isothermal inviscid fluid flow are continuity equation and euler equation.

- Mass conservation

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{V}) = 0 \tag{2.3}$$

  where, for an steady incompressible flow, the equation reduces to,

$$\nabla \cdot (\tilde{\mathbf{V}}) = 0 \tag{2.4}$$

- Euler Equation

$$\rho \frac{D\vec{V}}{Dt} = -\nabla \mathrm{p} + \rho \mathbf{g} \tag{2.5}$$

  where, $\frac{D}{Dt}$ is the material derivative operator, ρ is density, p is the pressure and $\mathbf{g}$ is acceleration due to gravity.

2.1.3 Boundary Conditions

The boundary condition assumed in the project is that of wall boundary condition.The simplest boundary condition is that of a wall. Since, fluid cannot pass through a wall, the normal component of velocity is set to zero relative to the wall along a face on which the wall boundary condition is prescribed. Mathematically speaking,

$$\vec{V} \cdot \vec{n} = 0 \tag{2.6}$$

where, $\vec{n}$ is the normal vector along the face of the wall

Figure 2.2 shows velocity having no components along the normal direction of the wall.

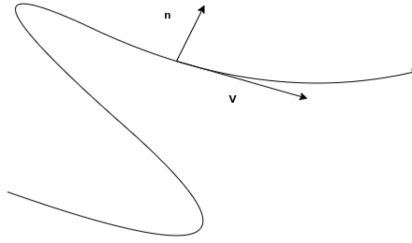

Figure 2.2: Wall Boundary Condition

2.2 Vector Calculus

2.2.1 Potential function and theorem

Any vector function can written as a sum of gradient of a scalar function and curl of a vector function.In particular, if it is a irrotational field i.e its curl vanishes , the the vector function can writtten as a gradient of scalar function.

$$\vec{V} = \nabla(\phi) + \nabla\times(\tilde{\mathbf{C}}) \tag{2.7}$$

In particular, for irrotational flow the equation reduces to,

$$\vec{V} = \nabla(\phi) \tag{2.8}$$

2.2.2 Laplace Equation

For an incompressible flow , we also have $\nabla \cdot (\tilde{\mathbf{V}}) = 0$, thus equation 2.8 reduces to laplace equation. Mathematically,

$$\nabla^2\phi = 0 \tag{2.9}$$

2.2.3 Harmonic functions,properties and uniqueness

The solutions of laplace equations are called as harmonic functions. The harmonic functions have two important properties which are listed down below.

- The value of the potential function ϕ at a point is the average of those around the point.

- Laplace equation toelrates no maximum or minima; extreme values of the ϕ occurs at the boundary.

Laplace's equation doesn not by itself determine ϕ; in addition, a suitable set of boundary conditions must be supplied. A proposed set of boundary conditons will suffice for a particular problem is ensured by uniqueness theorems presented below.

- First uniqueness theorem: The solution to Laplace's equation in some region is uniquely determined if ϕ or the normal derivative $\frac{\partial \phi}{\partial n}$ is specified on its boundary.

2.3 Finite Element Method (FEM)

2.3.1 Introduction and Methodology

In all disciplines of science and engineering, the focus is on determining what will happen when a physical system of interest is subjected to the effects of the environment, which may in turn lead to a response from the system.The set of mathematical equations allowing the determination of state variables and, in turn, the analytical investigation of any physical process, is sometimes referred to as the mathematical model of a process. In most physical problems of interest, the mathematical models typically correspond to differential equations. This means that we can formulate a set of equations involving the derivatives of functions with respect to spatial variables and/or time. One example differential equation, for the case that we want to find the spatial distribution of a state variable T(x), is the following:

$$\frac{dT}{dx} + s(x) = 0 \tag{2.10}$$

where, s(x) is some given function

For any problem, the use of the FEM can be assumed to consist of four basic conceptual steps, summarized in figure below. The first step is to formulate the differential equations that describe the problem. As mentioned above, these differential equations express the governing physics of the problem at hand and usually

comprise a mathematical statement of the conservation principle for a quantity such as mass, energy, or momentum. The differential equations for any problem are supplemented by boundary conditions, which merely provide statements about what happens at the boundary of the domain of the problem. For example, if we have a two-dimensional body, the boundary conditions will provide information on what happens on the bounding curve of the body.

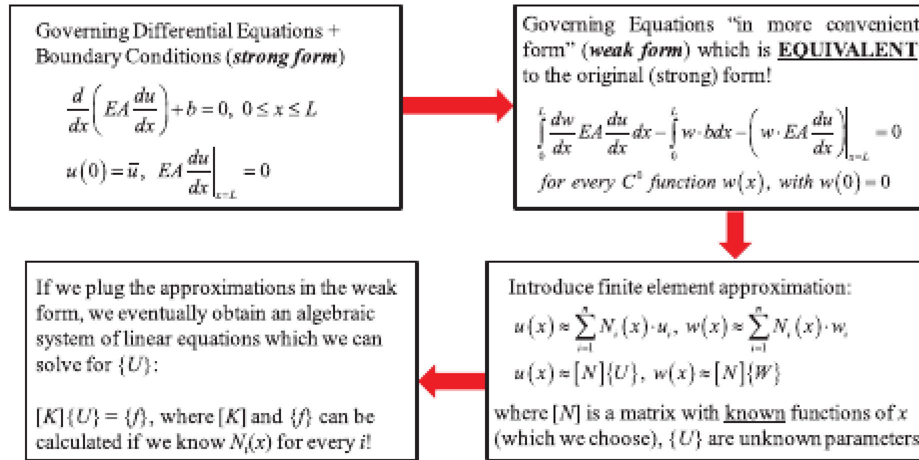

Figure 2.3: FEM Process

The differential equations and the boundary conditions collectively comprise the strong form of a physical problem. In simple cases, we may be able to solve the differential equation and obtain the exact solution to the strong form. However, in most cases (especially for multidimensional problems), it is not possible to analytically solve the differential equation. This,in turn, necessitates the use of numerical methods to obtain approximate solutions. The use of the FEM to solve such problem relies on a different approach, which requires the transformation of the strong form into an equivalent set of expressions called the weak form. The weak form of any problem typically involves integral equations. Eventually, we will plug the finite element approximation into the weak form, which will allow us to transform the problem from an integral equation, to a system of linear equations for a set of unknown constant quantities. This set of linear equations can easily be solved to obtain the numerical solution of given problem at hand.

2.3.2 FEM and Laplace equation

2.3.2.1 Strong form

The strong form of laplace equation constitutes the equation $\nabla^2\phi = 0$, augmented with boundary conditions. There are generally two different type of boundary condition imposed on the type of problem we are working with, essential boundary condition and neumann boundary condition.

- Essential boundary conditions:
  The value of the field function at hand i.e ϕ is specified at the boundary $\mathcal{T}$.

$$\phi = \bar{\phi} \text{ on } \mathcal{T}_\phi$$

- Neumann boundary conditions:
  The value of the normal derivative of the field is specified at the boundary $\mathcal{T}$.

$$\vec{V} \cdot \vec{n} \text{ on } \mathcal{T}_{\mathrm{II}}$$

Thus, the strong form consists of laplace equation augmented with relevant boundary condition.

$$\nabla \cdot (\nabla\phi) = 0 \tag{2.11}$$

$$\phi = \bar{\phi} \text{ on } \mathcal{T}_\phi \tag{2.12}$$

$$\vec{V} \cdot \vec{n} \text{ on } \mathcal{T}_{\mathrm{II}} \tag{2.13}$$

2.3.2.2 Weak form

We will now establish the weak form for two dimensional heat conduction. We consider an arbitrary function, w(x,y), which vanishes on the essential boundary: w = 0 at $\mathcal{T}_\phi$. We multiply the governing differential equation 2.11 by w(x,y) and then integrate over the two-dimensional domain, Ω:

$$\iint_\Omega w(\nabla \cdot (\nabla\phi))dV = 0 \tag{2.14}$$

Using Green's formula, we have,

$$\int_{\mathcal{T}} w \cdot (\nabla\phi \cdot \vec{n})dS - \iint_{\Omega} \nabla w \cdot \nabla\phi dV = 0 \tag{2.15}$$

The boundary integral in equation 2.15 can be separated into two parts, one over $\mathcal{T}_\phi$ (where we know that w vanishes) and the other over $\mathcal{T}_{\mathrm{II}}$ (for which we have the natural boundary condition):

$$\int_{\mathcal{T}} w \cdot (\nabla\phi \cdot \vec{n})dS = \int_{\mathcal{T}_\phi} w \cdot (\nabla\phi \cdot \vec{n})dS + \int_{\mathcal{T}_q} w \cdot (\nabla\phi \cdot \vec{n})dS \tag{2.16}$$

If we also account for the natural boundary condition of 2.13 becomes:

$$\int_{\mathcal{T}} w \cdot (\nabla\phi \cdot \vec{n})dS = -\int_{\mathcal{T}_q} w \cdot \bar{\phi}dS \tag{2.17}$$

Thus, we finally arrive at required weak form:

$$\iint_{\Omega} \nabla w \cdot \nabla\phi dV = -\int_{\mathcal{T}_q} w \cdot \bar{\phi}dS \tag{2.18}$$

for all w(x) with w = 0 at essential boundary.

2.3.2.3 Discretization

As shown in figure below, the salient feature of the finite element solution is that the two dimensional domain Ω is discretized (subdivided) into N_e subdomains, called elements, and each element consists of nodes. We will rely on a piecewise approximation, that is, we will stipulate the approximation of the field for each element separately.

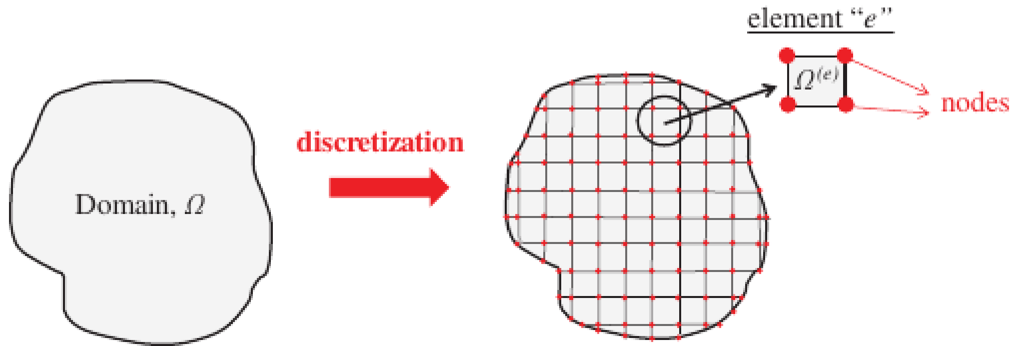

Figure 2.4: FEM Discretization

Consider a three noded traingular element as shown below.

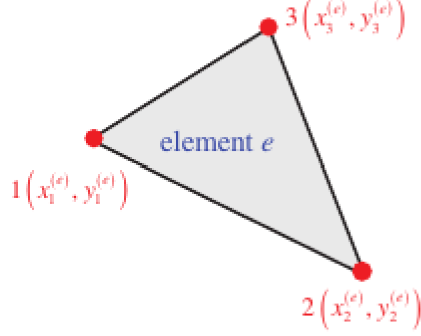

Figure 2.5: Triangular Element

We now approximate the nature of field function in each element as:

$$\phi^e(x,y) = a_0^e + a_1^e \cdot x + a_2^e \cdot y \tag{2.19}$$

$$\phi^e(x,y) = \begin{bmatrix} 1 & x & y \end{bmatrix} \cdot \begin{bmatrix} a_0^e \\ a_1^e \\ a_2^e \end{bmatrix} \tag{2.20}$$

putting values and solving for nodal values of ϕ, we have,

$$\phi^e(x,y) = \begin{bmatrix} N_1^e(x,y) & N_2^e(x,y) & N_3^e(x,y) \end{bmatrix} \cdot \begin{bmatrix} \phi_1^e \\ \phi_2^e \\ \phi_3^e \end{bmatrix} \tag{2.21}$$

$$\phi^e(x,y) = [N^e] \cdot [\phi^e] \tag{2.22}$$

where, $\begin{bmatrix} N_1^e(x,y) & N_2^e(x,y) & N_3^e(x,y) \end{bmatrix}$ is given by,

$$[N_1^e(x,y)] = \frac{1}{2 \cdot A^e} \cdot [x_2^e \cdot y_3^e - x_3^e \cdot y_2^e + (y_2^e - y_3^e) \cdot x + (x_3^e - x_2^e) \cdot y] \tag{2.23}$$

$$[N_2^e(x,y)] = \frac{1}{2 \cdot A^e} \cdot [x_3^e \cdot y_1^e - x_1^e \cdot y_3^e + (y_3^e - y_2^e) \cdot x + (x_1^e - x_3^e) \cdot y] \tag{2.24}$$

$$[N_3^e(x,y)] = \frac{1}{2 \cdot A^e} \cdot [x_1^e \cdot y_2^e - x_2^e \cdot y_1^e + (y_1^e - y_2^e) \cdot x + (x_2^e - x_1^e) \cdot y] \tag{2.25}$$

where, A_e represents area of the triangular element.

Similarly, we see that the gradient of the field function ϕ is given by,

$$\nabla\phi^e = \begin{bmatrix} \frac{\partial\phi^e}{\partial x} \\ \frac{\partial\phi^e}{\partial y} \end{bmatrix} = [B^e] \cdot [\phi^e] \tag{2.26}$$

where B^e is given by,

$$[B^e] = \frac{1}{2 \cdot A^e} \cdot \begin{bmatrix} y_2^e - y_3^e & y_3^e - y_1^e & y_1^e - y_2^e \\ x_3^e - x_2^e & x_1^e - x_3^e & x_2^e - x_1^e \end{bmatrix} \tag{2.27}$$

Similarly, the arbitrary function is also approximated in similar way.

$$w^e(x, y) = [N^e] \cdot [w^e] = [w^e]^T [N^e]^T \tag{2.28}$$

$$\nabla w^e = [B^e] \cdot [w^e] \tag{2.29}$$

$$[\nabla w^e]^T = [w^e]^T \cdot [B^e]^T \tag{2.30}$$

Where the equations has been transformed into transposed form.

Finally, we plug in the equation 2.30 and 2.26 into weak form of equation 2.18.The first part of equation is transformed as:

$$\iint_\Omega [\nabla w]^T \cdot \nabla\phi dV = \Sigma_{e=1}^{N_e} (\iint_{\Omega_e} [\nabla w]^T \cdot \nabla\phi dV) \tag{2.31}$$

where, the term occuring in brackets of second part can be simplified as:

$$\iint_{\Omega_e} [\nabla w]^T \cdot \nabla\phi dV = \iint_{\Omega_e} [w^e]^T \cdot [B^e]^T \cdot [B^e] \cdot [\phi^e] dV \tag{2.32}$$

$$\iint_{\Omega_e} [w^e]^T \cdot [B^e]^T \cdot [B^e] \cdot [\phi^e] dV = [w^e]^T \cdot (\iint_{\Omega_e} [B^e]^T \cdot [B^e] dV) \cdot [\phi^e] \tag{2.33}$$

$$\iint_{\Omega_e} [w^e]^T \cdot [B^e]^T \cdot [B^e] \cdot [\phi^e] dV = [w^e]^T \cdot [k^e] \cdot [\phi^e] \tag{2.34}$$

wher, k_e is called local stiffness matrix given by,

$$k_e = \iint_{\Omega_e} [B^e]^T \cdot [B^e] dV \tag{2.35}$$

If we write ϕ^e and w^e as:

$$[\phi^e] = [L^e] \cdot [\phi] \tag{2.36}$$

$$[w^e] = [L^e] \cdot [w] \tag{2.37}$$

Where, ϕ and w represents global value matrix for total domain in consideration. Thus, equation 2.32 looks like :

$$\iint_{\Omega_e} [\nabla w]^T \cdot \nabla \phi dV = [w]^T \cdot [L^e]^T \cdot \cdot [k^e] \cdot [L^e] \cdot [\phi] \tag{2.38}$$

Also the equation 2.31 transforms as :

$$\iint_{\Omega} [\nabla w]^T \cdot \nabla \phi dV = [w]^T \cdot [K] \cdot [\phi] \tag{2.39}$$

where [K], also called as global matrix is given by,

$$[K] = \Sigma_{e=1}^{N_e} ([L^e]^T \cdot [k^e] \cdot [L^e]) \tag{2.40}$$

Similarly, the second part of the equation 2.18 simplifies to :

$$\int_{\tau_q} w \cdot \bar{\phi} \cdot dS = \Sigma_{e=1}^{N_e} (\int_{\tau_q^e} w \cdot \bar{\phi}^e dS) \tag{2.41}$$

If we write,

$$[f^e_{\tau_\phi}] = \int_{\tau_q^e} w \cdot \bar{\phi}^e dS = \int_{\tau_q^e} [N^e]^T \cdot \bar{\phi}^e dS \tag{2.42}$$

we have,

$$-\int_{\tau_q} w \cdot \bar{\phi} \cdot dS = \Sigma_{e=1}^{N_e} \cdot ([w]^T \cdot [L^e]^T [f^e_{\tau_q}] \tag{2.43}$$

where, $[f^e_{\tau_q}]$ represents local force matrix.

Finally, equation 2.41 looks like:

$$-\int_{\tau_q} w \cdot \bar{\phi} \cdot dS = [w]^t \cdot [f] \tag{2.44}$$

where, [f] represents global force matrix.

Altogether we have then,

$$[w]^T \cdot [K] \cdot [\phi] = [w]^T \cdot [f] \ \forall \ w \tag{2.45}$$

$$[K] \cdot [\phi] = [f] \tag{2.46}$$

Thus obtained matrix equation can be solved for unknown values of the field function at different nodes to obtain the solution numerically.

2.4 Neural Network

2.4.1 Introduction

Many tasks involving intelligence or pattern recognition are extremely difficult to automate, but appear to be performed very easily by animals. For instance, animals recognize various objects and make sense out of the large amount of visual information in their surroundings, apparently requiring very little effort. It stands to reason that computing systems that attempt similar tasks will profit enormously from understanding how animals perform these tasks, and simulating these processes to the extent allowed by physical limitations. This necessitates the study and simulation of Neural Networks.

The roots of all work on neural networks are in neurobiological studies that date back to about a century ago. For many decades, biologists have speculated on exactly how the nervous system works. The following century old statement by James (1890) is particularly insightful, and is reflected in the subsequent work of many researchers. The amount of activity at any given point in the brain cortex is the sum of the tendencies of all other points to discharge into it, such tendencies being proportionate to :

- the number of times the excitement of other points may have accompanied that of the point in question
- to the intensities of such excitements
- to the absence of any rival point functionally disconnected with the first point, into which the discharges may be diverted.

How do nerves behave when stimulated by different magnitudes of electric current? Is there a minimal threshold (quantity of current) needed for nerves to be activated? Given that no single nerve cell is long enough, how do different nerve cells communicate electrical currents among one another? How do various nerve

cells differ in behavior? Although hypotheses could be formulated, reasonable answers to these questions could not be given and verified until the mid twentieth century, with the advance of neurology as a science.

Hayman (1999) is credited with developing the first mathematical model of a single neuron. This model has been modified and widely applied in subsequent work.System-builders are mainly concerned with questions as to whether a neuron model is sufficiently general to enable learning all kinds of functions, while being easy to implement, without requiring excessive computation within each neuron. Biological modelers, on the other hand, must also justify a neuron model by its biological plausibility.

Most neural network learning rules have their roots in statistical correlation analysis and in gradient descent search procedures. Hebb (1949) learning rule incrementally modifies connection weights by examining whether two connected nodes are simultaneously ON or OFF. Such a rule is still widely used, with some modifications. Rosenblatt (1958) "perceptron" neural model and the associated learning rule are based on gradient descent, rewarding or punishing a weight depending on the satisfactoriness of a neuron's behavior. The simplicity of this scheme was also its nemesis; there are certain simple pattern recognition tasks that individual perceptrons cannot accomplish, as shown by Minsky & Papert (1969). A similar problem was faced by the Widrow-Hoff (1960, 1962) learning rule, also based on gradient descent. Despite obvious limitations, accomplishments of these systems were exaggerated and incredible claims were asserted, saying that intelligent machines have come to exist. This discredited and discouraged neural network research among computer scientists and engineers.

In recent years, several other researchers (such as Amari, Grossberg, Hopfield, Kohonen, Bienenstock & von der Malsburg (1987), and Willshaw) have made major contributions to the field of neural networks; such as in self-organizing maps and in associative memories.

2.4.2 Perceptron

In this section , an artificial type of neuron called perceptron will be explained.artificial neuron called a perceptron. Perceptrons were developed in the 1950s and 1960s by the scientist Frank Rosenblatt, inspired by earlier work by Warren McCulloch and Walter Pitts. Today, it's more common to use other models of artificial neurons in this research, the main neuron model used is one called the sigmoid neuron.So how do perceptrons work? A perceptron takes several binary inputs, x_1, x_2, ... , and produces a single binary output.

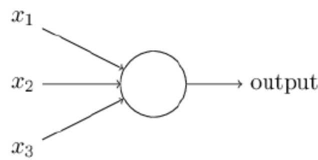

Figure 2.6: Perceptron

In the example shown the perceptron has three inputs,x_1, x_2 , x_3 . In general it could have more or fewer inputs.A simple rule is proposed to compute the output. Introduce weights, w_1 , w_2 , ... , real numbers expressing the importance of the respective inputs to the output. The neuron's output, 0 or 1 , is determined by whether the weighted sum Σ_j wj xj is less than or greater than some threshold value. Just like the weights, the threshold is a real number which is a parameter of the neuron.By varying the weights and the threshold, we can get different models of decisionmaking. To put it in more precise algebraic terms:

$$output = 0 \; if \; \Sigma_j w_j \cdot x_j \leq \; threshold \tag{2.47}$$

$$output = 1 \; if \; \Sigma_j w_j \cdot x_j > \; threshold \tag{2.48}$$

If we define bias = -threshold, we can rearrange above to arrive at following equation. The bias can be thought of as a measure how much likely is a particular neuron likely to fire.

$$output = 0 \; if \; \Sigma_j w_j \cdot x_j + bias \; \leq \; 0 \tag{2.49}$$

$$output = 1 \; if \; \Sigma_j w_j \cdot x_j + bias \; > \; 0 \tag{2.50}$$

2.4.3 Sigmoid Neurons

Suppose we have a network of perceptrons that we'd like to use to learn to solve some problem.And, we'd like the network to learn weights and biases so that the output from the network correctly classifies the digit. To see how learning might work, suppose we make a small change in some weight (or bias) in the network. What we'd like is for this small change in weight to cause only a small corresponding change in the output from the network. This property is what makes learning possible.Schematically,

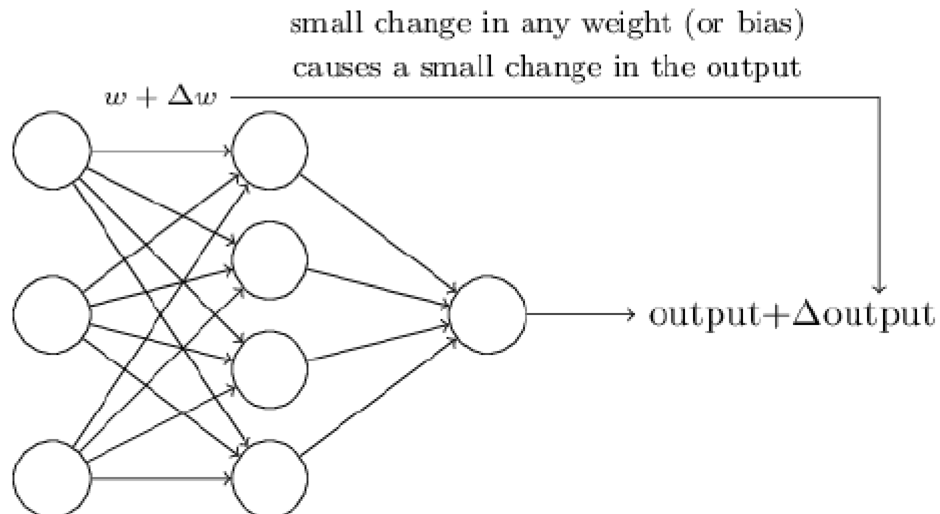

Figure 2.7: Network of Perceptrons

If it were true that a small change in a weight (or bias) causes only a small change in output, then we could use this fact to modify the weights and biases to get our network to behave more in the manner we want."9". And then we'd repeat this, changing the weights and biases over and over to produce better and better output. The network would be learning.The problem is that this isn't what happens when our network contains perceptrons. In fact, a small change in the weights or bias of any single perceptron in the network can sometimes cause the output of that perceptron to completely flip, say from 0 to 1 . That flip may then

cause the behaviour of the rest of the network to completely change in some very complicated way.

We can overcome this problem by introducing a new type of artificial neuron called a sigmoid neuron. Sigmoid neurons are similar to perceptrons, but modified so that small changes in their weights and bias cause only a small change in their output. That's the crucial fact which will allow a network of sigmoid neurons to learn.

Just like a perceptron, the sigmoid neuron has inputs, x_1, x_2 ,.. . But instead of being just 0 or 1 , these inputs can also take on any values between 0 and 1 . So, for instance, 0.638 is a valid input for a sigmoid neuron. Also just like a perceptron, the sigmoid neuron has weights for each input, w_1, w_2, and an overall bias, b. But the output is not 0 or 1 . Instead, its σ (w.x + b), where σ is called sigmoid function, and is defined by :

$$\sigma(z) = \frac{1}{1 + \exp(-z)} \tag{2.51}$$

Plotting the sigmoid function , it looks like :

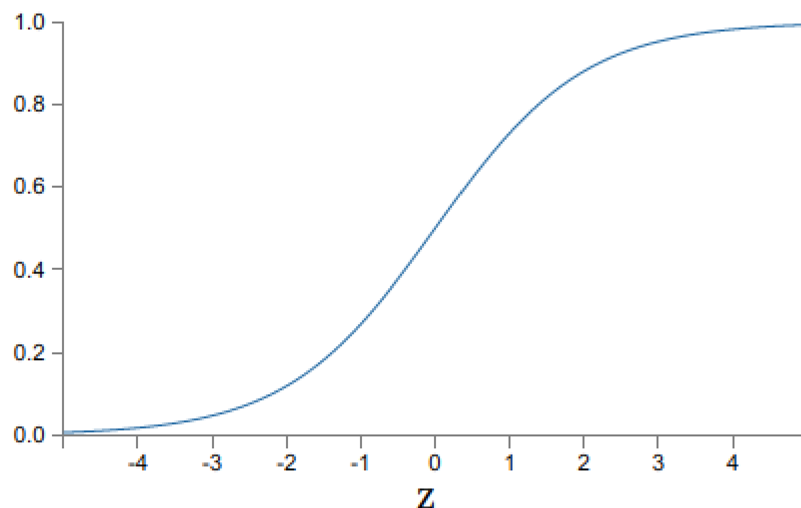

Figure 2.8: Sigmoid Function

If σ had in fact been a step function, then the sigmoid neuron would be a perceptron, since the output would be 1 or 0 depending on whether w . x + b was positive or negative. By using the actual σ function we get, as already implied

above, a smoothed out perceptron. Indeed, it's the smoothness of the *sigma* function that is the crucial fact, not its detailed form. The smoothness of σ means that small changes Δw_j in the weights and Δb in the bias will produce a small change Δoutput in the output from the neuron. In fact, calculus tells us that Δ output is well approximated by:

$$\Delta output = \Sigma_j \frac{\partial output}{\partial w_j} \cdot \Delta w_j + \frac{\partial output}{\partial b} \cdot \Delta b \tag{2.52}$$

2.4.4 Architecture of Neural Network

Consider a network of sigmoid neurons as shown below in the figure.Some terminologies of the network will be discussed below.

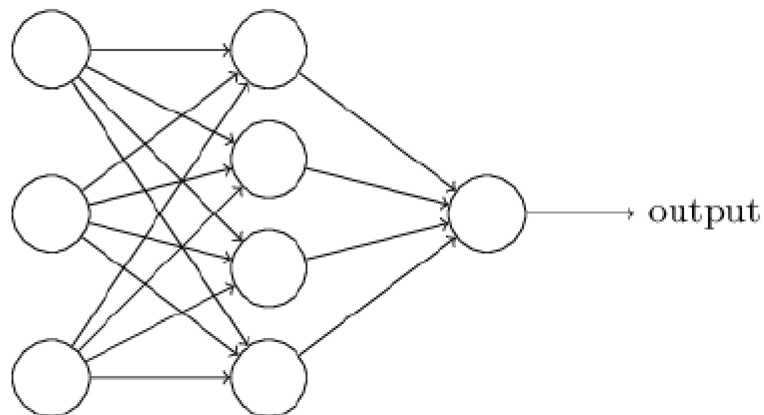

Figure 2.9: Example of a Network

As mentioned earlier, the leftmost layer in this network is called the input layer, and the neurons within the layer are called input neurons. The rightmost or output layer contains the output neurons, or, as in this case, a single output neuron. The middle layer is called a hidden layer, since the neurons in this layer are neither inputs nor outputs.The network above has just a single hidden layer, but some networks have multiple hidden layers. For example, the following four-layer network has two hidden layers. The actual design of the neural network we develop will be talked about in detail in next section called research methodology.

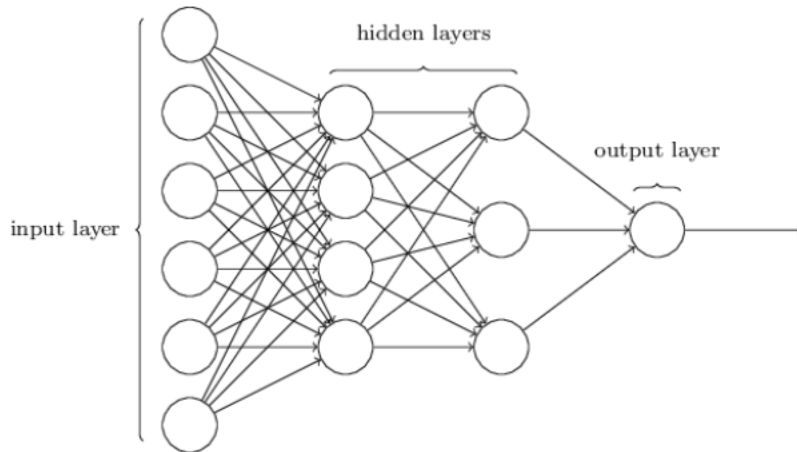

Figure 2.10: Example of a Sophisticated Network

2.4.5 Learning with Gradient Descent

To make a nerwork learn , what we'd like is an algorithm which lets us find weights and biases so that the output from the network approximates y(x) for all training inputs x. To quantify how well we're achieving this goal we define a cost function:

$$C(w, b) = \frac{1}{2n} \cdot \Sigma_x \; ||y(x) - a||^2 \tag{2.53}$$

Here, w denotes the collection of all weights in the network, b all the biases, n is the total number of training inputs, a is the vector of outputs from the network when x is input, and the sum is over all training inputs, x.Inspecting the form of the quadratic cost function, we see that C(w, b) is nonnegative, since every term in the sum is non negative. Furthermore, the cost C(w, b) becomes small, i.e., C(w, b) = 0 , precisely when y(x) is approximately equal to the output, a, for all training inputs, x. So our training algorithm has done a good job if it can find weights and biases so that C(w, b) = 0 . By contrast, it's not doing so well when C(w, b) is large that would mean that y(x) is not close to the output a for a large number of inputs. So the aim of our training algorithm will be to minimize the cost C(w, b) as a function of the weights and biases. In other words, we want to find a set of weights andbiases which make the cost as small as possible. It'll be done using an algorithm known as gradient descent.

let's suppose we're trying to minimize some function, C(v) . This could be any

real valued function of many variables,v = v_1 , v_2,... To minimize C(v) it helps to imagine C as a function of just two variables, which we'll call v_1 and v_2 :

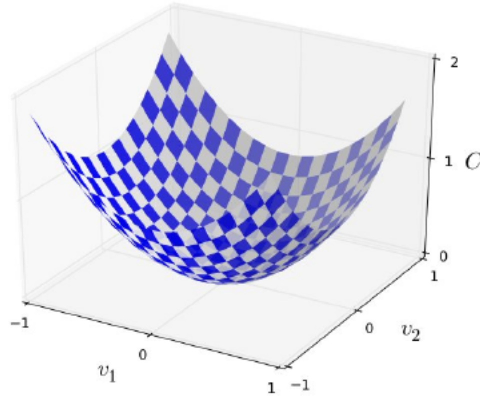

Figure 2.11: Double Variable Function

If we move a small amount Δv_1 in the v_1 direction, and a small amount Δv_2 in the v_2 direction. Calculus tells us that, C changes as follows:

$$\Delta C = \frac{\partial C}{\partial v_1} \cdot \Delta v_1 + \frac{\partial C}{\partial v_2} \cdot \Delta v_2 \tag{2.54}$$

We need to find a way of choosing Δv_1 and Δv_2 so as to make ΔC negative.We'll choose them so that a point is moving towards the valley of the function.To figure out how to make such a choice it helps to define Δv to be the vector of changes in v, $\Delta v = (\Delta v_1, \Delta v_2)^T$,where T is again the transpose operation, turning row vectors into column vectors. We'll also define the gradient of C to be the vector of partial derivatives,$(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2})^T$. We denote the gradient vector by ∇C, i.e.:

$$\nabla C = (\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2})^T \tag{2.55}$$

Thus equation 2.54 looks like :

$$\Delta C = \nabla C \cdot \Delta v \tag{2.56}$$

If we choose, Δv as :

$$\Delta v = -\eta \cdot \nabla C \tag{2.57}$$

Where, η is a small positive parameter called as learning rate.

Then, equation 2.56 will transforms as :

$$\Delta C = -\eta \nabla C \cdot \nabla C = -\eta \; ||\nabla C||^2 \tag{2.58}$$

Because $||\nabla C||^2 \geq 0$, this guarantees that $\Delta C \leq 0$, i.e., C will always decrease, never increase if we change v according to prescription in equation 2.57.Mathematically,

$$v \rightarrow v' = v - \eta \nabla C \tag{2.59}$$

Then we'll use this update rule again, to make another move. If we keep doing this, over and over, we'll keep decreasing C until we hope we reach a global minimum. g.Above section explains gradient descent when C is a function of just two variables. But, in fact, everything works just as well even when C is a function of many more variables. Suppose in particular that C is a function of m variables, v_1,..., v_m . Then the change ΔC in C produced by a small change Δv = $(\Delta v_1, ..., \Delta v_m)^T$ is: Thus equation 2.54 looks like :

$$\Delta C = \nabla C \cdot \Delta v \tag{2.60}$$

Where, the ∇C is the vector :

$$\nabla C = (\frac{\partial C}{\partial v_1}, ..., \frac{\partial C}{\partial v_m})^T \tag{2.61}$$

Just as for the two variable case, we can choose,

$$\Delta v = -\eta \cdot \nabla C \tag{2.62}$$

and we're guaranteed that our expression 2.60 for ΔC will be negative. This gives us a way of following the gradient to a minimum, even when C is a function of many variables, by repeatedly applying the update rule

$$v \rightarrow v' = v - \eta \nabla C \tag{2.63}$$

One can think of this update rule as defining the gradient descent algorithm. It gives us a way of repeatedly changing the position v in order to find a minimum of the function C .

2.5 Related Researches

The main focus of this research is to increase our understanding of the impacts of the boundary condition and intial starting point on the potential flow of the body.Understanding the impacts of this phenomenon shall contribute to the de-

sign and faster computation time for similar problems.

Cheng (2022) has given a technique of solving the poisson equation where both type of boundary conditions be it Dirchlet or Neumann boundary condition is specified on the domain of interest. The author introduces ghost points beyond the boundary domain which is neccesary to solve the poisson equation where neumann type of boundary condition is specified.

Koutromanos (2018) has given the fundamental details of solving laplace equation with neumann or dirchlet boundary conditon by using the technique of finite element method (FEM). The author clearly explains how the process of solving a laplace eqaution eveolves from strong form to weak form to discretization and finally solving the obtained algebraically constrained equations.

Nielsen (2015) has developed an neural network algorithm for accurately classifying and recognizing digits. Here the author talks about the basics of neural network which includes the idea from perceptrons to sigmoid neurons to gradient descent algorithm. The author also talks about how can an efficiency of such networks can be enhanced.

Santos et al. (2020) have developed a 3D Neural network that provides fast and accurate fluid flow prediction for three dimensional rock images. They trained their network to extract spatial relationships between the porous medium morphology and the fluid velocity field.Their results show that the extracted information is sufficient to obtain accurate flow field predictions in less than a second, without performing expensive numerical simulations providing a speed-up of several orders of magnitude.

Bishop (1994) has discussed about basic two type of models of neural network in use today. They describe these models in detail and explains about various techniques used to train them. They also discuss about various key issues that must be addressed when applying neural networks to practical problems. Finally, they survey various classes of problem which may be addressed using neural networks and illustrate variety of examples drawn from a range of fields.

2.6 Software Environment

2.6.1 Python

Python is a open source, high-level, general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation.Python is dynamically-typed and garbage-collected. It supports multiple programming paradigms, including structured (particularly procedural), object-oriented and functional programming. Guido van Rossum began working on Python in the late 1980s as a successor to the ABC programming language and first released it in 1991 as Python 0.9.0. Python 2.0 was released in 2000 and introduced new features such as list comprehensions, cycle-detecting garbage collection, reference counting, and Unicode support. Current version of Python 3.0, was released in 2008.

Rather than building all of its functionality into its core, Python was designed to be highly extensible via modules. This compact modularity has made it particularly popular as a means of adding programmable interfaces to existing applications.Python strives for a simpler, less-cluttered syntax and grammar while giving developers a choice in their coding methodology.Python's large standard library provides tools suited to many tasks and is commonly cited as one of its greatest strengths.

2.6.2 Python Modules

2.6.2.1 NumPy

NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. It also has functions for working in domain of linear algebra, fourier transform, and matrices..The ancestor of NumPy, Numeric, was originally created by Jim Hugunin with contributions from several other developers. In 2005, Travis Oliphant created NumPy by incorporating features of the competing Numarray into Numeric, with extensive modifications. NumPy is open-source software and has many contributors.

NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.The array object in NumPy is called ndarray, it provides a lot of supporting functions that make working with ndarray very easy.Arrays are very frequently used in data science, where speed and resources are very important.The major advantages of using NumPy over traditional lists are:

- Compact storage
- Fast array loops
- Slicing without copying
- Array operations
- Compatibility
- Memory and Execution efficiency

2.6.2.2 Matplotlib

Data visualization is key to understanding complex pattern in the data. It helps reveal underlying trends and patterns in data so the the end-users, including data scientists or analysts, make an informed decision based on it. Matplootlib is one of Python's most effective visualization libraries for data visualization.

Matplotlib is an open-source data visualization and graph plotting library built over Numpy arrays.John Hunter presented it in the year 2002. It is a two dimensional visualization library, but some extensions also allow it to plot three dimensional graphs. It provides several plotting functions to modify and fine-tune the graphs. Additionally, it supports various plots like scatter plots, bar charts, histograms, box plots, line charts, pie charts, etc. The major advantages of using Matplotlib are:

- It supports various kinds of graphs like Bar, histograms, Line-plots, Scatter-plots, etc.
- It can be used and accessed through Python Scripts, iPython shells and Jupyter Notebook.

- It can also allow us to create three dimensional plots.

CHAPTER THREE: RESEARCH METHODOLOGY

The research will primarily focus on developing a neural network capable of predicting path of a body in a potential flow. The result predicted by the network will be compared against the experimental data to reach a proper conclusion.

Table 3.1: Methodology.

Literature Review	• Review related researches
Take Inputs and Generate Mesh	• Take input boundary conditions • Create the rectangular flow domain • Generate thie mesh
Case Setup Solution and Calculations	• Apply boundary conditions • Solve using FEM • Store the result
Development of Neural Network	• Divide into feed and test data • Construct the neural network • Predict result for given input point
Post Processing	• Plot the predicted path • Compare against actual path • Analyze the observed errors
Documentation and Conclusion	• Draw proper conclusions.

3.1 Literature Review

The research works on the related topic of developing a neural network are collected from various sources. These researches are then reviewed to identify the problem statement. Other literary research papers are reviewed in order to find a suitable solution to the research gap identified.

3.2 Inputs and Mesh generation

The development of a neural network capable of predicting the path of a body in two dimensional potential flow will be the main focus of the research.The domain is considered to be rectangular so the inital step will be to take dimension of this domain. The platform to enter this variables will be provided in jupyter notebook.

3.2.1 Boundary Conditions and Initial Point

In this section, the user will input lenght(l) and breadth(b) of the rectangular domain. Furthermore, the boundary conditions imposed will be also taken as input. Similarly, the initial point where the body initally is present in domain is provided. This is important because later on, the neural network will take this point as an input and provide us with the plot of appropriate predicted path. A screen shot of Sublime Text platform where users can input such data is shown below:

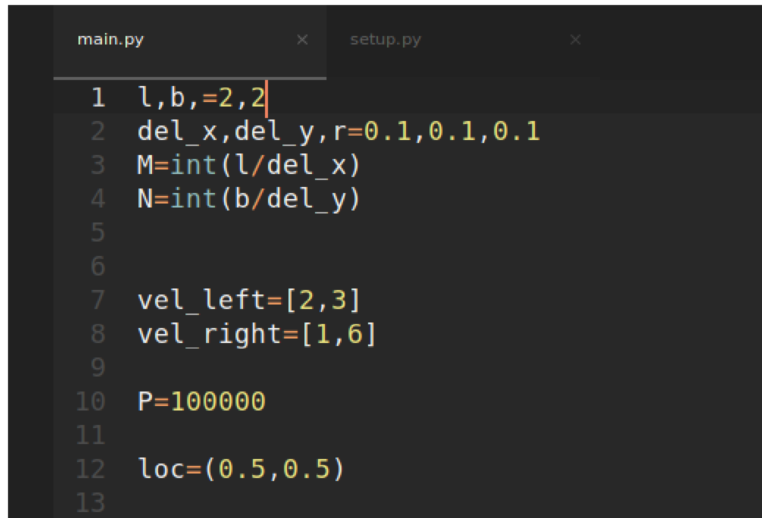

Figure 3.1: Platform for Input Condition in Sublime Text

Figure 3.1 shows the input platform in jupyter notebook.On close inspection, we see that we have to specify the co-ordinates of the two points in the particular part of the recttangular domain with magnitude and direction (angle in degrees) of the flow in between these points.

3.2.2 Create the flow domain

Once the input for the boundary condition as well as parameter of flow domain is obtained the program creates the recatnagular flow domain as shown below.

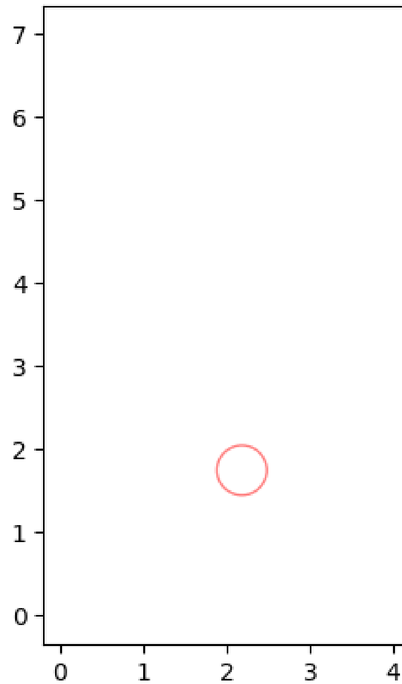

Figure 3.2: Flow domain

Figure 3.2 shows the rectangular flow domain with an object inserted in a position specified by the user. Infact, the program will have to create numerous such flow domains with the location of object placed at different point through out the domain. This is important because the solution of ϕ at this various point in the domain is required to calculate the actual trajectory of the object for given initial point.

Infact, numerous such domains will be created by the program where the initial point is present at different location within the domain. This is needed because we need to calculate the actual trajectory for given initial point.Some samples of the generated flow domains are shown below:

Figure 3.3: Many flow domains

3.2.3 Generation of Mesh

Once the flow domain is established the next step is to generate mesh over it.This process is performed by a module called setup which is written in numpy, which can be seen in appendix. The mesh is created in the grayed out domain between circle and the rectangle.

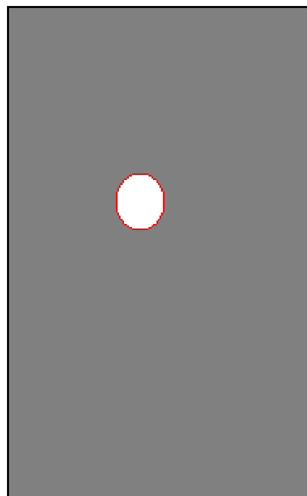

Figure 3.4: Mesh Region

Figure 3.3 shows the region wehere mesh structure of the rectangular flow domain with an object inserted in a position specified by the user is to be drawn.

3.3 Case Setup, Solution and Calculations

3.3.1 Case Setup and Solution

After the users inputs all the required information and after program prepares the mesh structure, the program applies the boundary condition to solve the problem at hand. As already mentioned, the solution of $\phi(x, y)$ is calculated by putting the value of the location of the object throught out this domain. This problem as already mentioned is solved by the techniques of FEM.

The code to solve this problem is attached in the appendix section of this report. After the solution is obtained, the obtained solutions are stored, which will be further used be used to calculate actual trajecory and later by a neural network for correcting its weights and biases for prediction purpose.

3.3.2 Calculations

3.3.2.1 Velocity Field Calculation

After obtaining the solution of laplace equation we apply equation 2.8 i.e $\vec{V} = \nabla\phi$ to obtain the field information. The techniques of FDM (Second Order) method is used to calculate this deirivatives. Specifically, the x and y component can be obtained as:

$$v_x = \frac{\partial \phi}{\partial x} \tag{3.1}$$

$$v_y = \frac{\partial \phi}{\partial y} \tag{3.2}$$

3.3.2.2 Pressure Calculation

After obtaining the velocity field, we need to calculate the distribution of pressure through out the domain. This is done via the euler equation 2.5. Expanding out this equation and neglecting the time components we have :

$$\rho \cdot (\ v_x \cdot \frac{\partial v_x}{\partial x} + v_y \cdot \frac{\partial v_x}{\partial y}\ ) = -\frac{\partial p}{\partial x} \tag{3.3}$$

$$\rho \cdot (\ v_x \cdot \frac{\partial v_y}{\partial x} + v_y \cdot \frac{\partial v_y}{\partial y} \ ) = -\frac{\partial p}{\partial y} \tag{3.4}$$

With the initial condition of the pressure specified as well this equations are used to calculate the distribution of pressure through out the domain.

3.3.2.3 Force Calculation

Now that the information of pressure distibution is known throught out the domain, we proceed to calculate the force experienced by this object. To do this we note the value of pressure through out the perimeter of the body. The force experienced by this body is compressive directed along the center. A Schematic figure of the force experienced by the body is shown below:

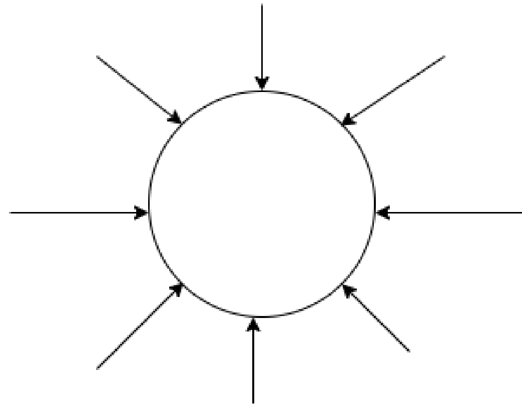

Figure 3.5: Forces on the Body

The net force experienced by the body is then given by vector sum of all this forces acting at its center.

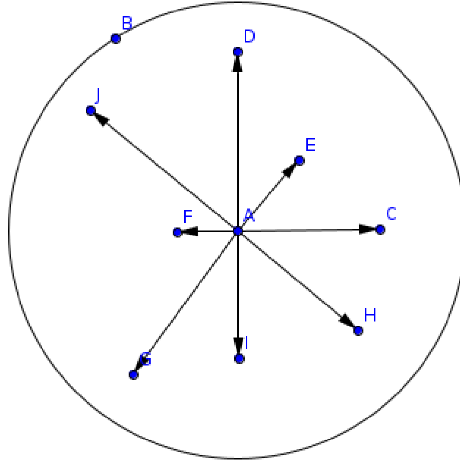

Figure 3.6: Force Diagram

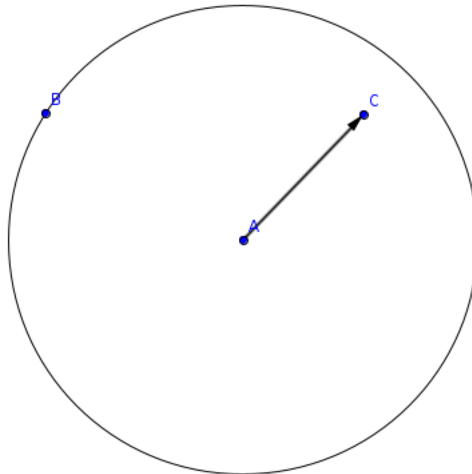

Figure 3.7: Net Force Diagram

Above figure shows the net force (magnitude and direction) experienced by the body. The amount of force and acceleration experienced by the body when placed at different location through out the domain is calculated and stored. This information is used to calculate the trajectory of the body with a given starting point in the next section.

3.3.2.4 Trajectory Calculation

Once we know the force experienced by the body at different location through out the body, we can now calculate the trajectory given the initial location. Suppose

(x_0, y_0) is the initial location with initial velocity $(v_{x_0} = 0, v_{y_0} = 0)$, then we can use following kinematics equation to know about its next location. We pick the travel time Δt to be very small.

$$x_1 = x_0 + v_{x_0} \cdot \Delta t + \frac{1}{2} \cdot a_x \cdot (\Delta t)^2 \tag{3.5}$$

$$y_1 = y_0 + v_{y_0} \cdot \Delta t + \frac{1}{2} \cdot a_y \cdot (\Delta t)^2 \tag{3.6}$$

Where, (x_1, y_1) represents the next point of the travelling body

Calculation of velocity at this location is caaried out via following equations:

$$v_{x_1} = v_{x_0} + a_x \cdot \Delta t \tag{3.7}$$

$$v_{y_1} = v_{y_0} + a_y \cdot \Delta t \tag{3.8}$$

where, (v_{x_1}, v_{y_1}) represents velocity of the body at this new location.

Since we also know the acceleration experienced by this body at this new location, we can use equation 3.5, 3.6, 3.7, 3.8 again to calulate new position of the body. In this way, we calculate trajectory of the body.

3.3.3 Store the Result

After velocity, pressure field and trajectory information is obtained using Python and Numpy, the data is processed to get accurate results, and then stored in a CSV file. CSV files are commonly used in data analysis and data science due to their ease of transfer between different software systems.

In the next phase, the data stored in the CSV file will be used to train a neural network. By training the neural network on the velocity,pressure field and trjectory information, it will be able to predict the behavior of path of an object in fluid flow in different scenarios.

3.4 Development of Neural Network

We will develop a neural network with 3 layers to solve the problem at hand. The input layer will have 2 neurons, which will receive the input data as a tuple of two values (x, y). The middle layer will have 30 neurons, which will extract features from the input data and pass it to the output layer. The output layer will have 10 neurons, representing 5 tuples (x, y), which will make predictions about the target values. To train the network, we will use the gradient descent algorithm and set the learning rate (eta) to 3. During each iteration of the training process, the network will compute the error between the predicted values and the actual values and adjust the weights and biases to minimize the error. This process will continue until the error is reduced to an acceptable level, or a specified number of iterations is reached. The final result of the training process will be a set of optimized weights and biases that produce accurate predictions for the target values.

Finally, after the training process is completed, the weights and biases of the neural network will be stored in either a CSV or NPY file for future use. This allows us to use the trained model for new, unseen input data. To compute the output for a special user input tuple (x, y), we will simply pass it through the network, using the stored weights and biases, to produce the prediction. The network will apply the activation function to the weighted sum of the inputs for each neuron in the middle layer and finally produce the output at the prediction layer. This output will be our prediction for the target values corresponding to the user input tuple (x, y). With this, we will have a trained neural network that can perform prediction tasks effectively, with a high degree of accuracy. Additionally, the computed output from the neural network will be compared against the actual data to evaluate the performance of the network. This will give us an idea of how well the network has learned the underlying relationship between the input and target values during the training process.

3.5 Post Processing

The results obtained from the network needs to be interpreted and visualized. To visualize and analyze the results, Matplotlib is used. Matplotlib is a free, cross-platform program for data analysis and visualization. It may be applied to generate visualizations for both qualitative and quantitative data analysis. Different types of plots like scatter plot, bar chart, graphs, mesh plot, contour plot will be used through out the visualisation process.

3.6 Documentation/Conclusion

Finally, with all the necessary simulations and analysis of the results, a conclusion shall be drawn and documented.

CHAPTER FOUR: RESULTS AND DISCUSSION

4.1 Development of the mesh

The mesh region will be meshed using triangular meshes, which can be represented in Python by a list of lists of three tuples. For example, each sublist can contain the vertices of a single triangle, specified as tuples of (x, y) coordinates. These meshes can be drawn at different locations for objects throughout the domain, allowing for more complex and accurate modeling of geometric shapes and their interactions.

In addition, the objects within the domain will be placed at different locations, and meshes will be created for each of these as well. This is necessary in order to accurately calculate the trajectory of each object, taking into account its position and movement within the meshed domain. By representing the domain and objects with triangular meshes, complex motions and interactions can be modeled and simulated with high accuracy.

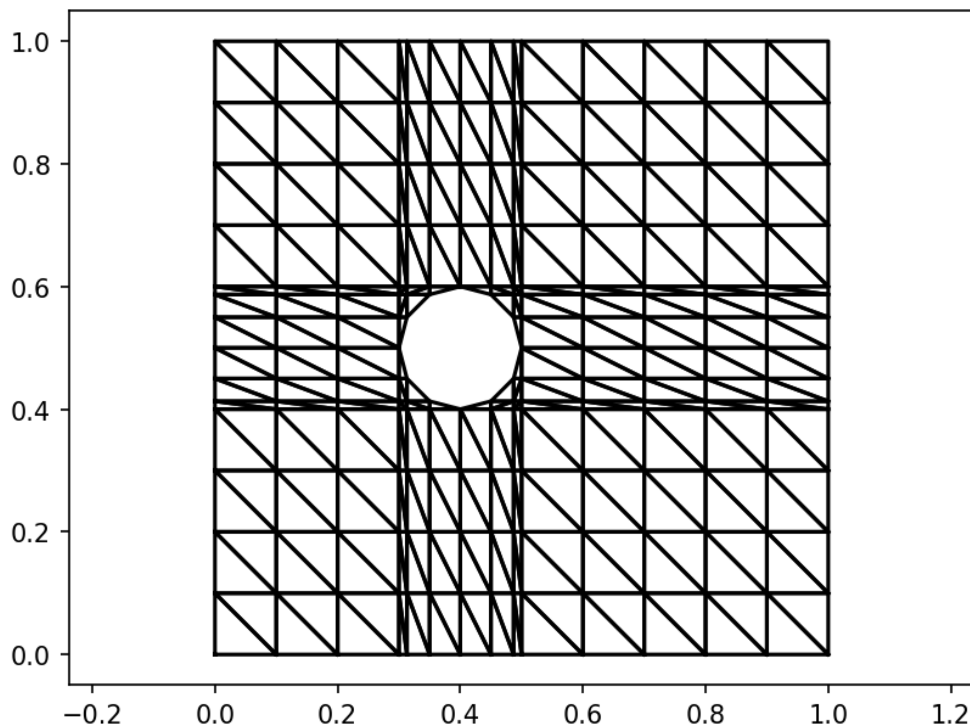

Figure 4.1: Triangular meshes drawn in the region

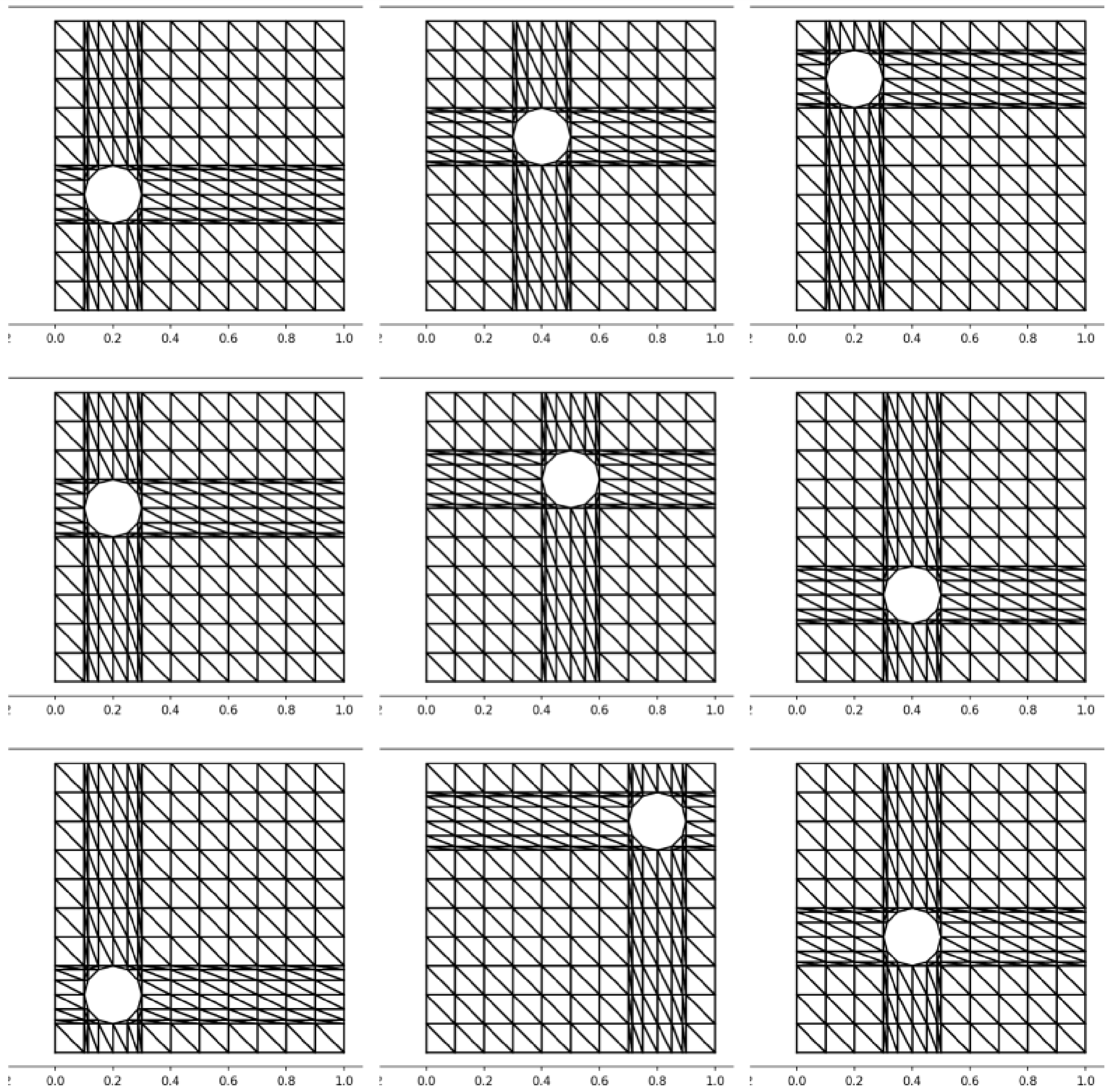

Figure 4.2: Different meshes drawn in the region

Figure 4.2 shows different mesh diagram developed and stored by placing objects through at differnet points throughout the rectangular domain.

4.2 Potential Solution

The FEM solution for the Laplace equation is implemented in a Python program, which generates a scatter plot of the solution. This plot provides a visual representation of the solution and can help in identifying any patterns or trends in the data. By analyzing the scatter plot, one can gain valuable insights into the Laplace equation solution and its behavior in different situations.

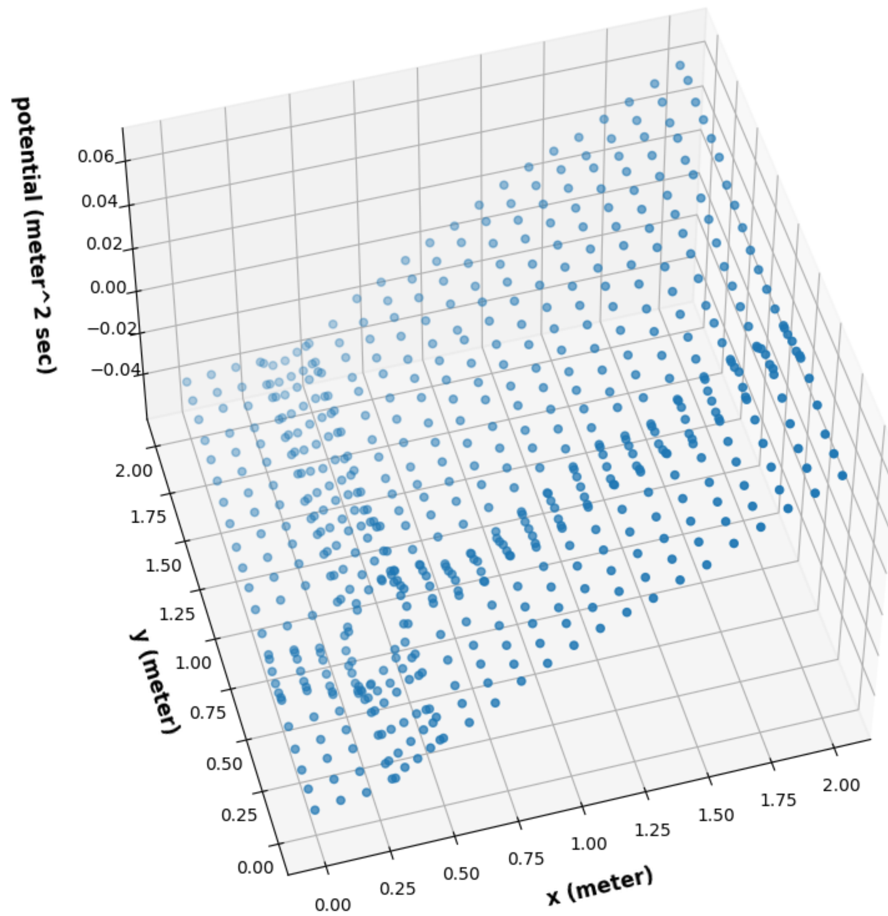

Figure 4.3: Scatter plot of the solution

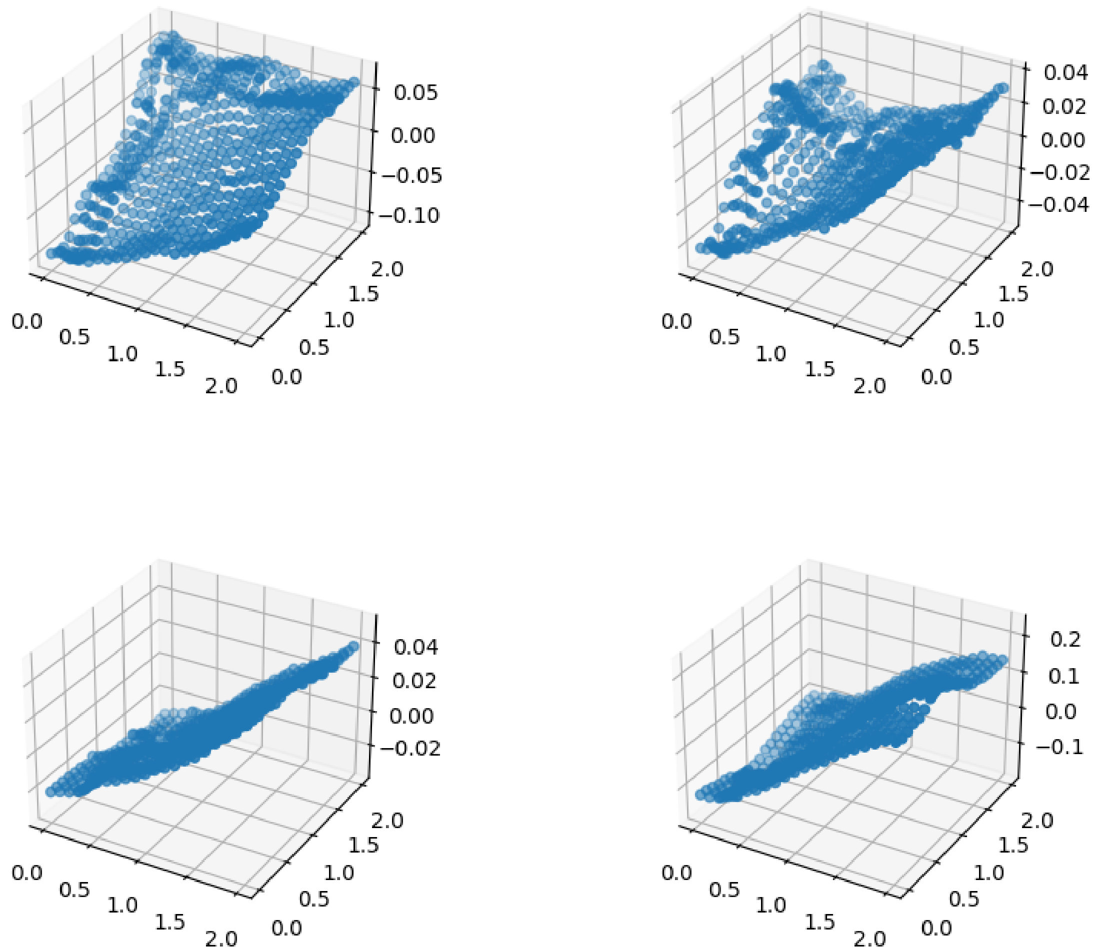

Figure 4.4: Scatter plot at various points

Figure 4.4 shows different scatter plot solution by placing object at various points throughout the rectangular domain.

4.3 Velocity Field

After applying the gradient to the obtained solution of laplace equation we get velocity field information.

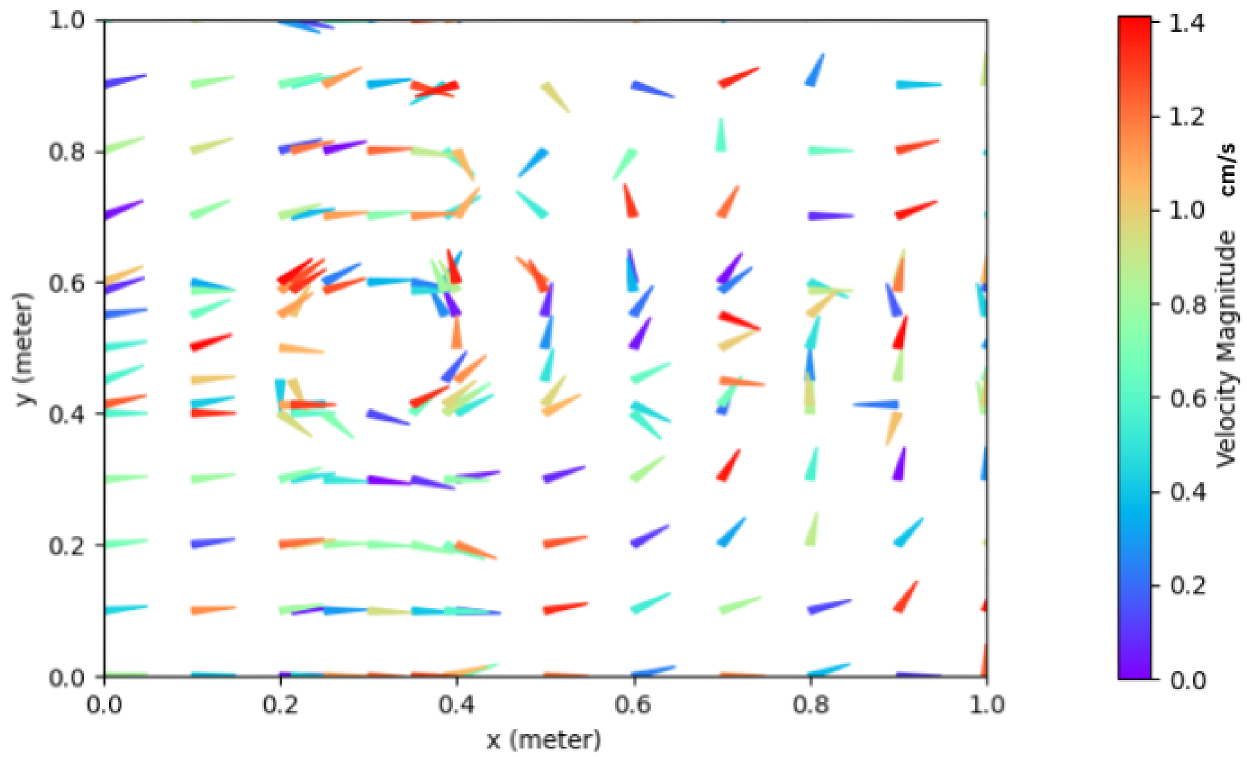

Figure 4.5: Velocity vector field

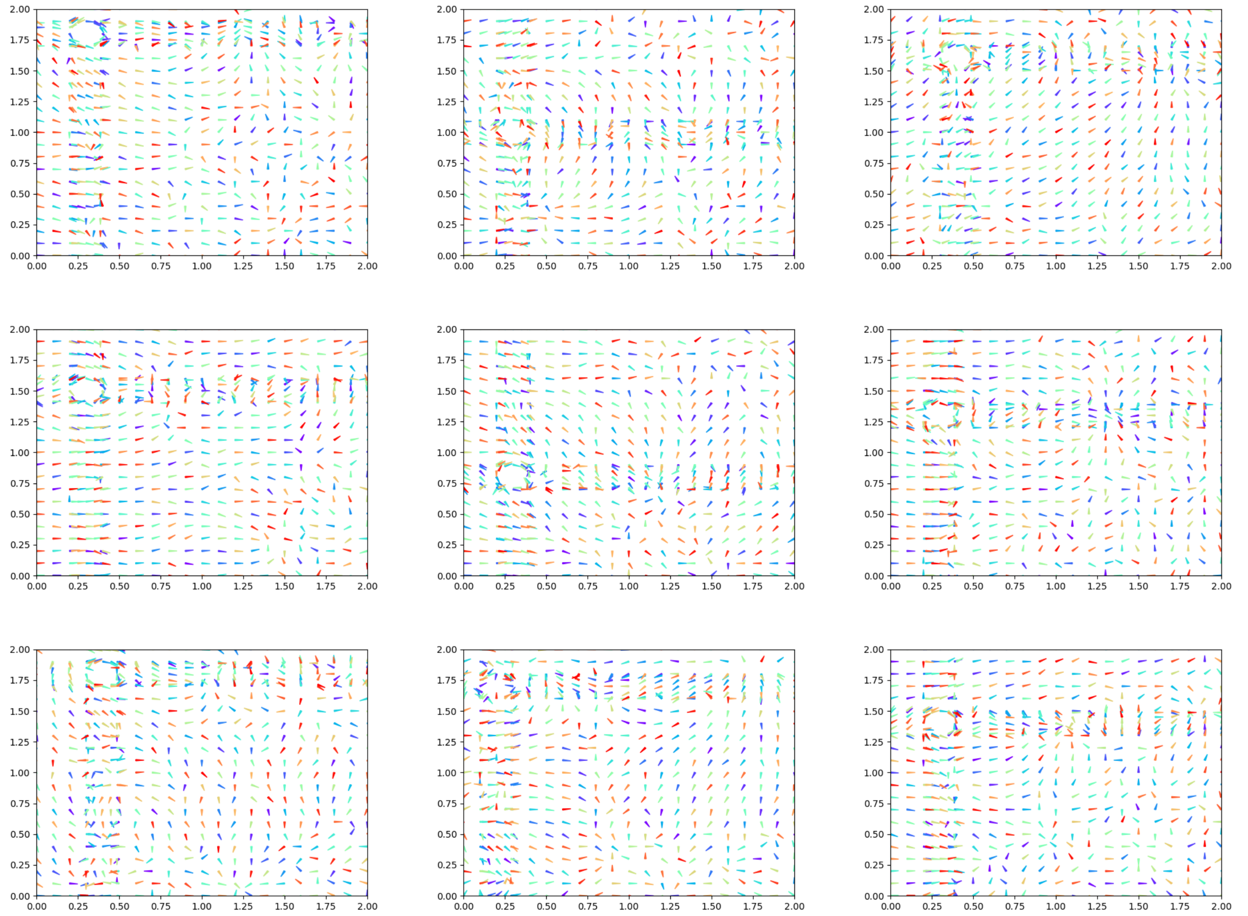

Figure 4.6: Velocity field at different object location

Figure 4.6 shows different velocity field of vector by placing object at various points throughout the rectangular domain.

4.4 Path Trajectory

4.4.1 Via Calculation

The path of the object is calculated using kinematic equation mentioned in previous section.

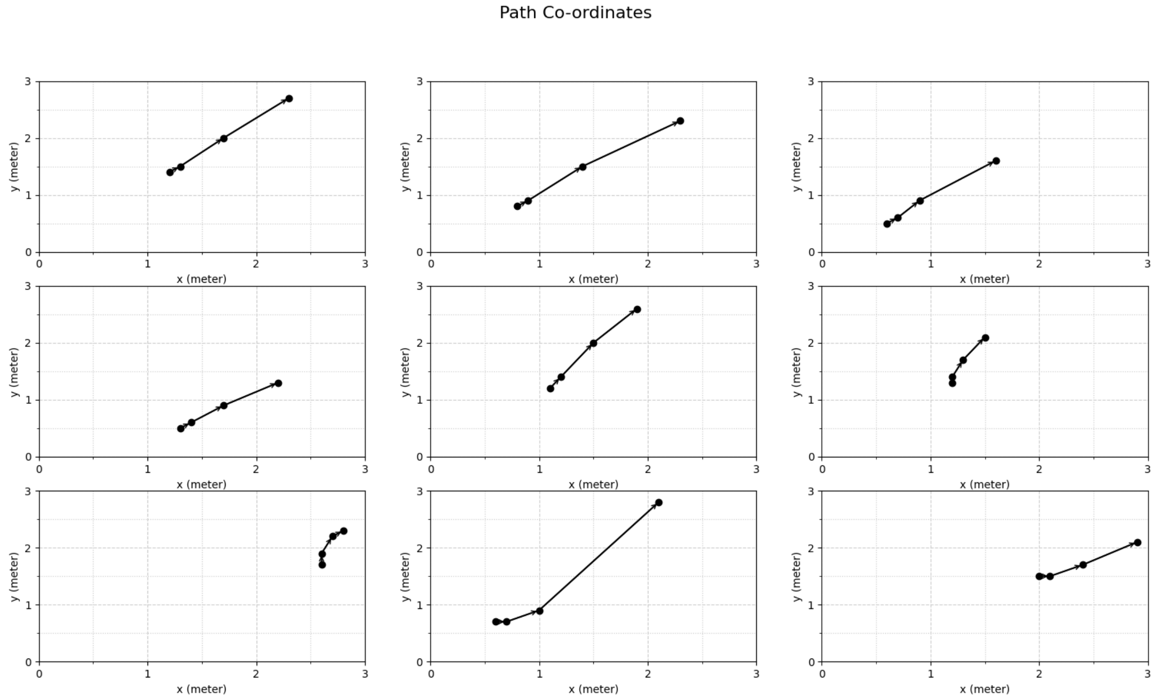

Figure 4.7: Path trajectory of the body

Figure 4.6 shows different path trajectory of the body at various starting points throughout the rectangular domain.

4.4.2 Via Neural Network

The predicted path of the object is calculated by feeding the initial point to the network and monitoring its output. For different initial points, we see following path predicted by the network.

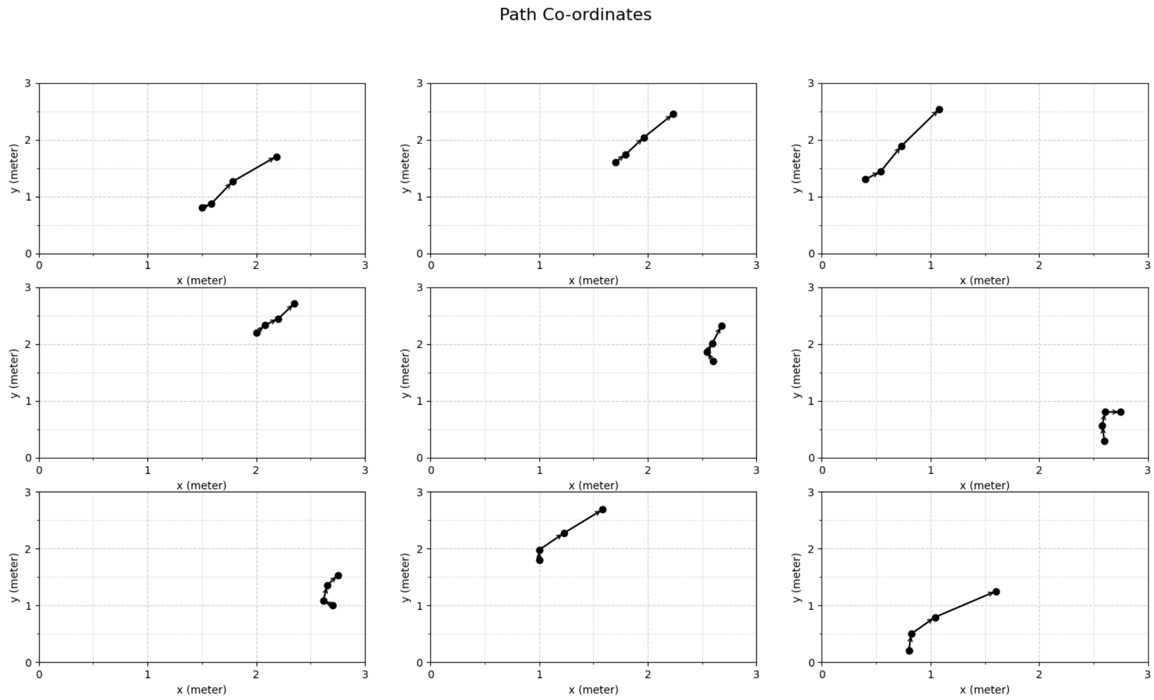

Figure 4.8: Path tajectory of the body by Network

Figure 4.6 shows different path trajectory of the body at various starting points throughout the rectangular domain.

4.5 Error Analysis and Comparison

4.5.1 Mean Square Error (MSE)

In neural networks, Mean Squared Error (MSE) is a commonly used metric to evaluate the performance of the model. It measures the average squared difference between the predicted output and the actual output over all the examples in the dataset. MSE is calculated by taking the sum of squared differences between the predicted and actual values and dividing it by the total number of examples.

In our neural network with two input nodes and six output nodes, the MSE will be calculated by comparing the predicted values for each of the six output nodes with their corresponding actual values. The predicted values will be obtained by feeding the input values through the network and obtaining the output values at the six output nodes. The actual values will be the known values from the dataset that correspond to the input values.

Once the predicted and actual values are obtained, the MSE can be calculated by taking the squared difference between each predicted and actual value, summing and averaging them up across all six output nodes, and dividing by the total number of examples. This will give us a measure of how well our neural network is performing on the given dataset, with lower MSE indicating better performance.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{4.1}$$

where,N represents the number of samples in the dataset y_i represents the actual value for the ith sample, and $\hat{y}_i$ represents the predicted value for the ith sample.

4.5.2 MSE vs Learning rate

Once the algorithm for calculating error is defined we need to see how MSE varies when we change the learning rate in our network. The learning rate determines the step size the optimization algorithm takes to update the weights of the network during training. Changing the learning rate can have a significant impact on the convergence of the network and its overall performance.

By monitoring the MSE during training with different learning rates, we can determine an optimal learning rate that leads to the best performance on the given dataset. This is an essential step in tuning the hyperparameters of the network to achieve the best results. We will also randomly shuffle the the entire dataset to figure out best learning rate range for our network.

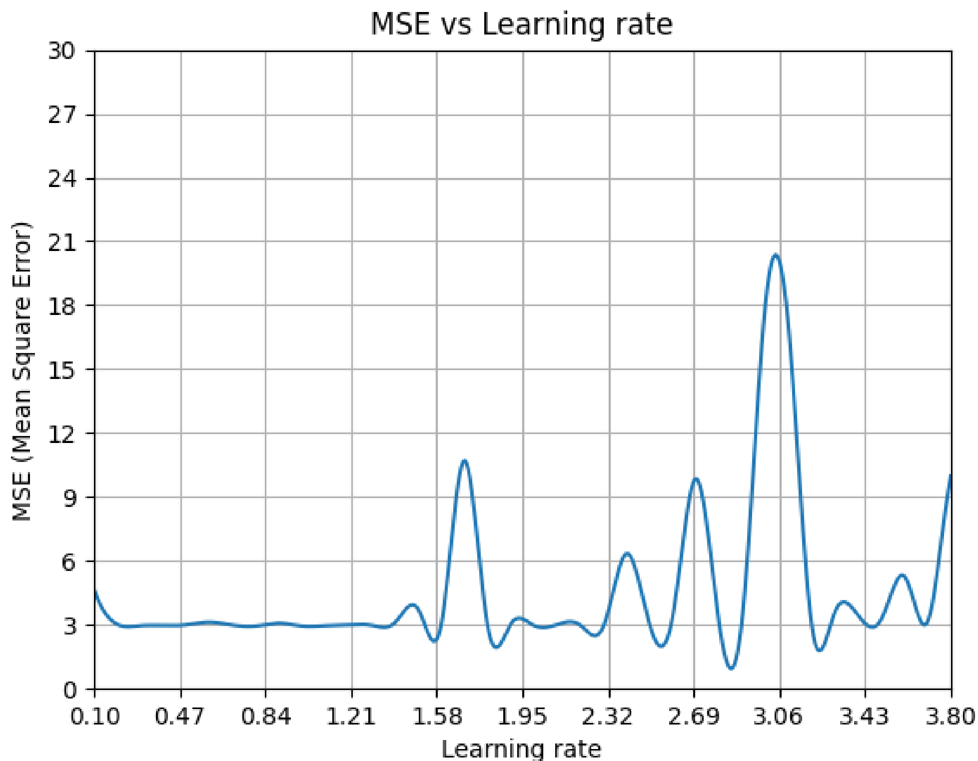

Figure 4.9: MSE vs Learning Rate

Figure 4.9 shows that MSE stabilizes for values of η around the region of 0.3 to 1. To confirm such is the case, we shuffle our dataset and see how MSE varies with learning rate for various set of shuffled dataset.

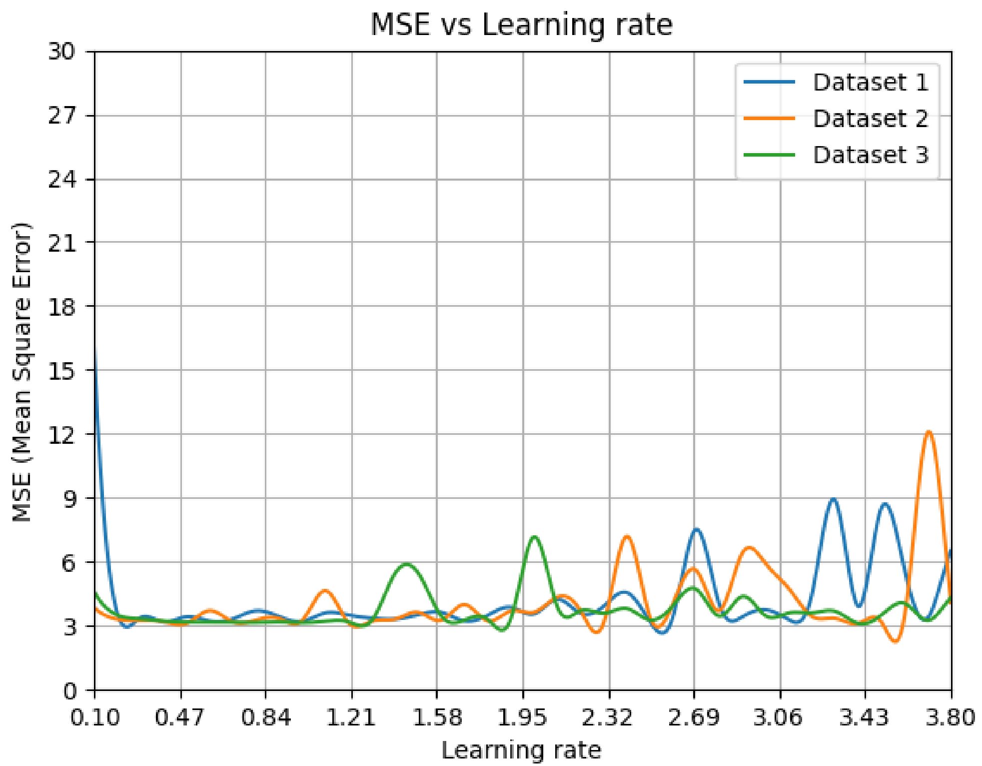

Figure 4.10: MSE vs Learning Rate

Figure 4.10 indeed confirms that the region of stability for learning rate η is around 0.3-1 for most of the dataset. Beyond this region, it can be seen that mean square error fluctuate too much and thus choosing learning rate from this region would imply that there is chance of occurrence of high value for mse for any random dataset.

4.5.3 MSE vs No of Neurons

To understand the impact of the number of neurons in the middle layer on model performance, we can examine how the MSE varies with different numbers of neurons. By monitoring the MSE while changing the number of neurons in the middle layer, we can determine the optimal number of neurons for the given problem. Once again we will randomly shuffle our data to figure out best range for number of neurons in our middle layer.

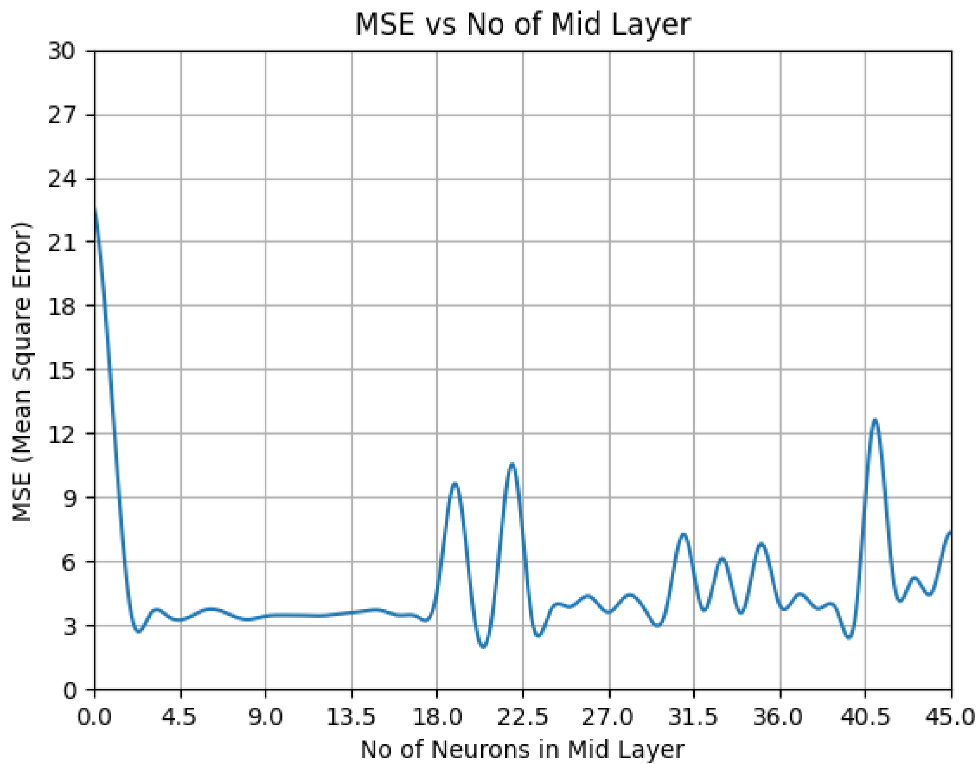

Figure 4.11: MSE vs No of Neurons

Figure 4.11 shows that MSE is stable for number of neurons in the region of 5-12. To confirm such is the case, we shuffle our dataset and see how MSE varies with number of neurons in middle layer for various set of shuffled dataset.

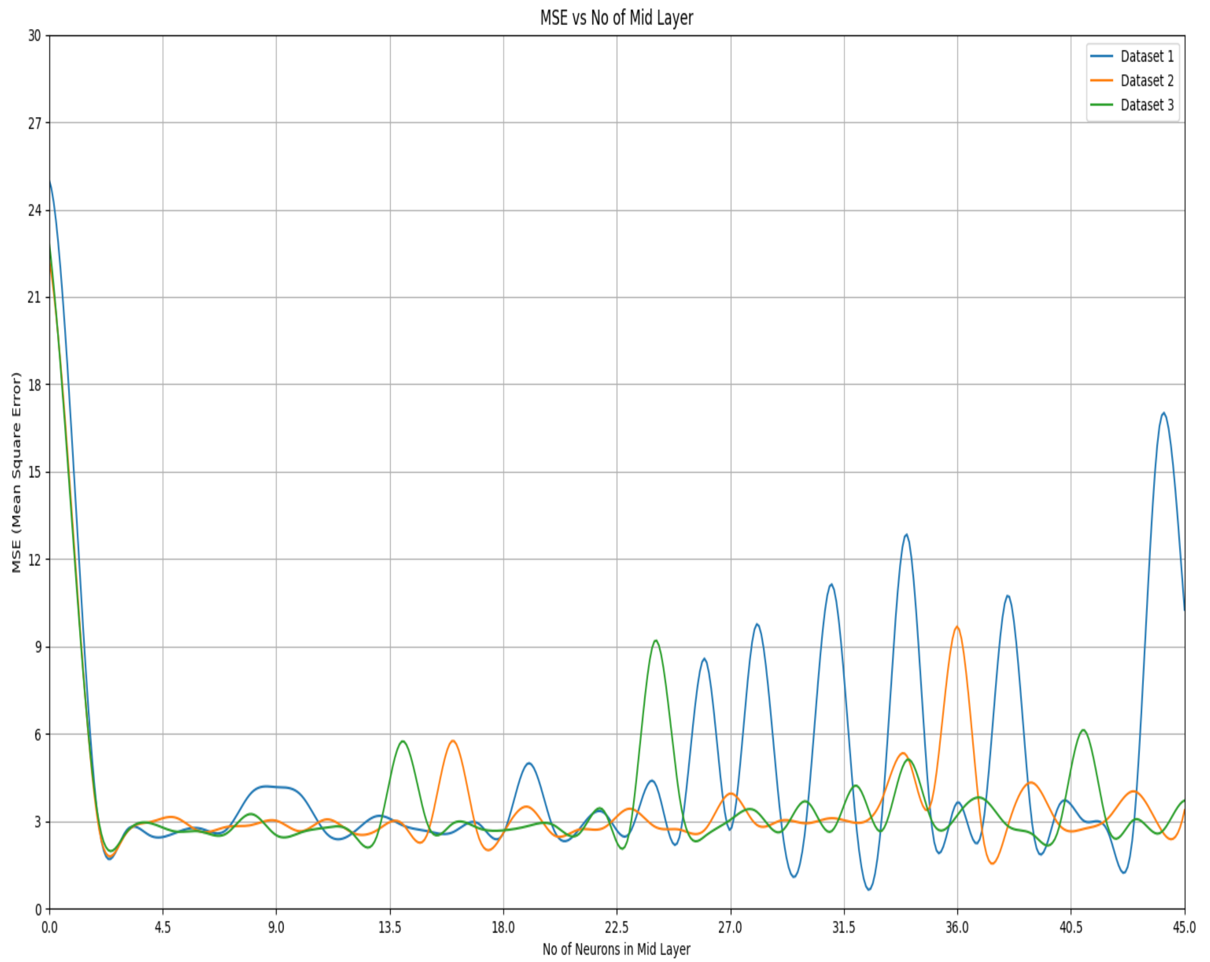

Figure 4.12: MSE vs No of Neurons

Figure 4.12 indeed confirms that the region of minimum MSE for number of neurons for most of the dataset occurs in between 5-12.It can also be seen that beyond this region, mean square error fluctuates too much.

4.5.4 MSE vs No of Dataset

To analyze the impact of dataset size on model performance, we can examine how the MSE varies with different numbers of data samples. By monitoring the MSE while changing the number of data samples, we can determine the minimum amount of data required for the model to achieve acceptable performance, as well as the effect of increasing dataset size beyond this minimum.

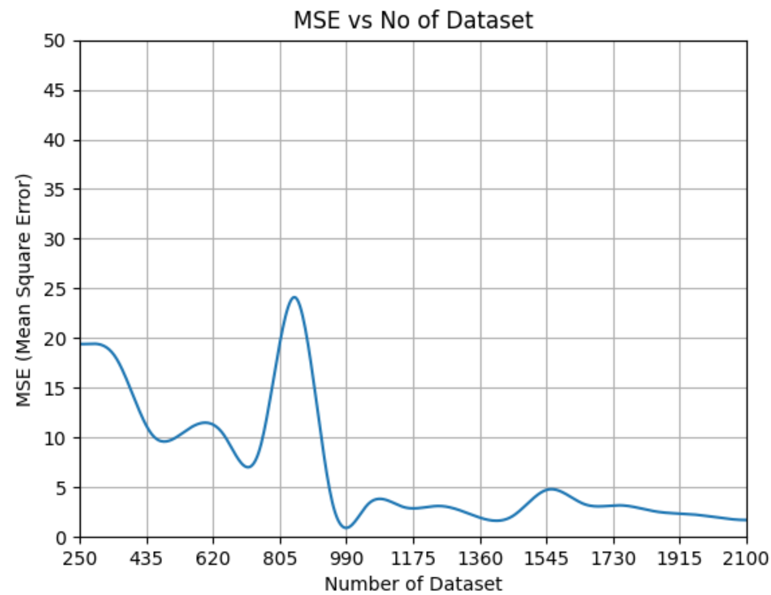

Figure 4.13: MSE vs No of Dataset

Figure 4.13 shows that MSE stabilises whent there is at least 1700 number of data in our dataset. We confirm such prediction by shuffling and see how MSE varies with number of data for various set of shuffled dataset.

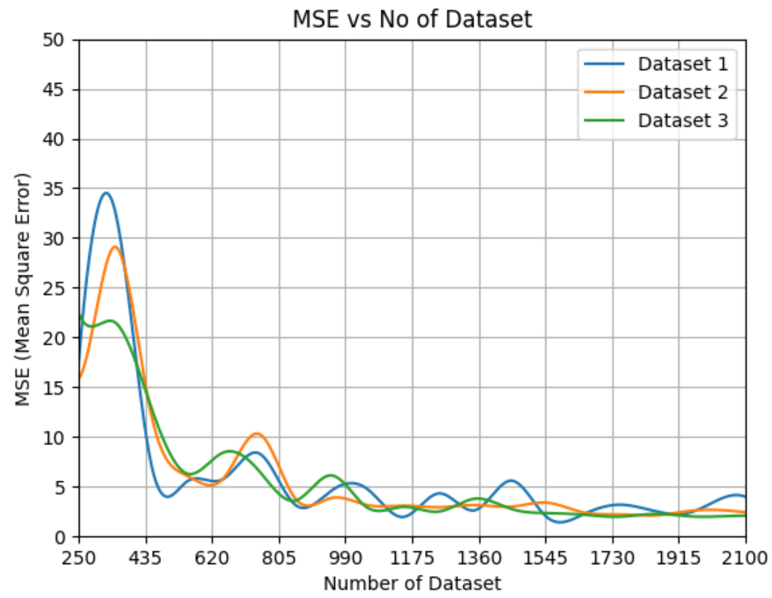

Figure 4.14: MSE vs No of Dataset

Figure 4.14 also shows that for most of the dataset,more than 1700 number of data is required so that mean square error doesn't fluctuate too much.

CHAPTER FIVE: CONCLUSIONS AND RECOMMENDATIONS

5.1 Conclusions

Following conclusions can be drawn from this research.

1. A neural network capable of predicting path in potential flow has been developed.

2. Through this research, we were able to determine the optimal values for key hyperparameters in our neural network model. Specifically, we found that a learning rate in the range of 0.3 to 1, a middle layer consisting of 7-10 neurons were necessary for achieving effective performance in our model. By systematically varying these hyperparameters and monitoring the resulting MSE values, we were able to identify these optimal values and refine our model for better results. These findings can be used to guide the design of future neural network models for similar problems.

3. The development of the neural network to predict the path of an object in a potential flow required numerical computation of the velocity and pressure fields, which provide information about the forces acting on the body and thus determine its path. These fields were then used as input features to the neural network, enabling it to learn the complex relationship between the object's motion and the fluid dynamics of the potential flow.

4. Using a neural network for such problems offers faster results compared to numerically solving the governing equations. This makes neural networks a valuable tool for accelerating the design and optimization of fluid dynamic systems.

5.2 Recommendations

The following suggestions are made for further research in this area:

1. The study could be extended to incorporate three dimensional potential flow systems.

2. An experimental setup to study such paths of body in potential flow is recommended .

3. A complicated and difficult extension of the study can be to incorporate viscosity and compressibility effects i.e Non potential flows.

4. It is recommended to develop a network with more than one middle layer, and see if performance of the network improves or declines.

References

Bienenstock, E., & von der Malsburg, C. (1987). A neural network for invariant pattern recognition. *EPL (Europhysics Letters)*, *4* (1), 121.

Bishop, C. M. (1994). Neural networks and their applications. *Review of scientific instruments*, *65* (6), 1803–1832.

Cheng, L. (2022). *Finite difference methods for poisson equation.*

Hayman, S. (1999). The mcculloch-pitts model. In *Ijcnn'99. international joint conference on neural networks. proceedings (cat. no. 99ch36339)* (Vol. 6, pp. 4438–4439).

Hebb, D. O. (1949). The first stage of perception: growth of the assembly. *The Organization of Behavior*, *4*, 60–78.

James, W. (1890). The perception of reality. *Principles of psychology*, *2*, 283–324.

Koutromanos, I. (2018). *Fundamentals of finite element analysis: Linear finite element analysis.* John Wiley & Sons.

Minsky, M., & Papert, S. (1969). An introduction to computational geometry. *Cambridge tiass., HIT*, *479*, 480.

Nielsen, M. A. (2015). *Neural networks and deep learning* (Vol. 25). Determination press San Francisco, CA, USA.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, *65* (6), 386.

Santos, J. E., Xu, D., Jo, H., Landry, C. J., Prodanović, M., & Pyrcz, M. J. (2020). Poreflow-net: A 3d convolutional neural network to predict fluid flow through porous media. *Advances in Water Resources*, *138*, 103539.

Appendix

main.py

```
l,b,=1,1
del_x,del_y,r=0.1,0.1,0.1
M=int(l/del_x)
N=int(b/del_y)

vel_left=[2/100,3/100]
vel_right=[3/100,6/100]

P=100000

loc=(0.5,0.5)

from setup.setup import setup

force_dict=setup(l,b,del_x,del_y,r,vel_left,vel_right,P,loc)
#print(force_dict)

from path import path
dict_for_NN=path(force_dict,1,3,3)

#saving to pickle file
import pickle
with open('dict_for_NN.pickle', 'wb') as f:
    pickle.dump(dict_for_NN, f)
```

setup.py

```
import math
import numpy as np

def points_on_circle(center, radius):
    points = []
    for i in range(0,360,30):
        angle = 2 * math.pi * i / 360
        x = round(center[0] + radius * math.cos(angle),3)
        y = round(center[1] + radius * math.sin(angle),3)
        points.append((x, y))
    return points

def setup(l,b,del_x,del_y,r,v_left,v_right,P,loc):
    from setup.extras import int_points,mesh_1,mesh_2,mesh_3,
                                       mesh_4,mesh_5,mesh_6,mesh_7,
                                       mesh_8
    from setup.solution_k import global_dict,k_global
    from setup.solution_f import f_global,val_dict
    from setup.csv_codes import dictionary_to_csv,force_to_csv
    from setup.solution_fdm import f_x,f_y,dp_dx,dp_dy,pressure,
                                       force
    from plots import plot_triangles,combine,velo_plot

    force_dict={}

    interior_pts=int_points(l,b,del_x,del_y,r)

    for obj in interior_pts:
        print(obj)

        bndry_circle_pts=points_on_circle(obj,r)

        bndry_circle_pts_y_cord=list(set([i[1] for i in
                                         bndry_circle_pts ]))
        bndry_circle_pts_y_cord.sort()

        bndry_circle_pts_x_cord=list(set([i[0] for i in
                                         bndry_circle_pts ]))
```

```
bndry_circle_pts_x_cord.sort()

mesh_lst_1=mesh_1(obj,l,b,del_x,del_y,r,
                                bndry_circle_pts_y_cord)
mesh_lst_2=mesh_2(obj,l,b,del_x,del_y,r,
                                bndry_circle_pts_x_cord)
mesh_lst_3=mesh_3(obj,l,b,del_x,del_y,r,
                                bndry_circle_pts_y_cord)
mesh_lst_4=mesh_4(obj,l,b,del_x,del_y,r,
                                bndry_circle_pts_x_cord)

mesh_lst_5=mesh_5(obj,l,b,del_x,del_y,r,
                                bndry_circle_pts_x_cord,
                                bndry_circle_pts_y_cord)
mesh_lst_6=mesh_6(obj,l,b,del_x,del_y,r,
                                bndry_circle_pts_x_cord,
                                bndry_circle_pts_y_cord)
mesh_lst_7=mesh_7(obj,l,b,del_x,del_y,r,
                                bndry_circle_pts_x_cord,
                                bndry_circle_pts_y_cord)
mesh_lst_8=mesh_8(obj,l,b,del_x,del_y,r,
                                bndry_circle_pts_x_cord,
                                bndry_circle_pts_y_cord)

mesh=mesh_lst_1+mesh_lst_2+mesh_lst_3+mesh_lst_4+
                                mesh_lst_5+mesh_lst_6+
                                mesh_lst_7+mesh_lst_8
plot_triangles(mesh, filename='mesh_png/'+'mesh_'+str(obj)
                                +'.png')


# print(mesh)

glob_dict=global_dict(mesh) # {():1}
# print(len(mesh))

K_glob=k_global(mesh,glob_dict)
np.savetxt('K_csv/'+'K_'+str(obj)+'.csv',K_glob,delimiter=
                                ',')
```

```
F_glob=f_global(mesh,glob_dict,v_left,v_right,l)
np.savetxt('F_csv/'+'F_'+str(obj)+'.csv',F_glob,delimiter=
                                  ',')

try:
    u=np.linalg.solve(K_glob,F_glob)
except: # K might be singular matrix, in that cas error so
                                  skip if error happens
    continue

value_dict= val_dict(u,glob_dict)
dictionary_to_csv(value_dict,'u_csv/'+'u_'+str(obj)+'.csv'
                                  )
# print(value_dict)

v_x_dict=f_x(obj,l,b,del_x,del_y,value_dict,
                                  bndry_circle_pts_y_cord)
dictionary_to_csv(v_x_dict,'v_x_csv/'+'v_x_'+str(obj)+'.
                                  csv')

v_y_dict=f_y(obj,l,b,del_x,del_y,value_dict,
                                  bndry_circle_pts_x_cord)
dictionary_to_csv(v_y_dict,'v_y_csv/'+'v_y_'+str(obj)+'.
                                  csv')

combine_v_dict=combine(v_x_dict,v_y_dict)
velo_plot(combine_v_dict,obj,l,b)

dv_x_dx_dict=f_x(obj,l,b,del_x,del_y,v_x_dict,
                                  bndry_circle_pts_y_cord)
dictionary_to_csv(dv_x_dx_dict,'dv_x_dx_csv/'+'dv_x_dx_'+
                                  str(obj)+'.csv')

dv_x_dy_dict=f_y(obj,l,b,del_x,del_y,v_x_dict,
                                  bndry_circle_pts_x_cord)
dictionary_to_csv(dv_x_dy_dict,'dv_x_dy_csv/'+'dv_x_dy_'+
                                  str(obj)+'.csv')

dv_y_dx_dict=f_x(obj,l,b,del_x,del_y,v_y_dict,
```

```
                                              bndry_circle_pts_y_cord)
            dictionary_to_csv(dv_y_dx_dict,'dv_y_dx_csv/'+'dv_y_dx_'+
                                              str(obj)+'.csv')

            dv_y_dy_dict=f_y(obj,l,b,del_x,del_y,v_y_dict,
                                              bndry_circle_pts_x_cord)
            dictionary_to_csv(dv_y_dy_dict,'dv_y_dy_csv/'+'dv_y_dy_'+
                                              str(obj)+'.csv')

            dp_dx_dict=dp_dx(v_x_dict,v_y_dict,dv_x_dx_dict,
                                              dv_x_dy_dict)
            dictionary_to_csv(dp_dx_dict,'dp_dx_csv/'+'dp_dx_'+str(obj
                                              )+'.csv')

            dp_dy_dict=dp_dy(v_x_dict,v_y_dict,dv_y_dx_dict,
                                              dv_y_dy_dict)
            dictionary_to_csv(dp_dy_dict,'dp_dy_csv/'+'dp_dy_'+str(obj
                                              )+'.csv')

            press_dict=pressure(obj,l,b,del_x,del_y,dp_dx_dict,
                                              dp_dy_dict,
                                              bndry_circle_pts,P)
            dictionary_to_csv(press_dict,'pressure_csv/'+'pressure_'+
                                              str(obj)+'.csv')

            force_lst=force(press_dict,r)
            force_to_csv(obj,force_lst,'force_csv/'+'force.csv')

            force_dict[obj]=force_lst
            # appending to force_dict {obj:[f_x,f_y]}

    return force_dict
```

mesh.py

```
import numpy as np

def int_points(l,b,del_x,del_y,r):
    M=int(l/del_x)
    N=int(b/del_y)
    interior_pts=[]
    for i in range (2,M-1):
        for j in range (2,N-1):
            interior_pts.append((round(i*del_x,1),round(j*del_y,1)
                                                  ))

    return interior_pts

def mesh_1(obj,l,b,del_x,del_y,r,bndry_circle_pts_y_cord):
    x_list=[round(i/10,1) for i in range(0,int(round(10*obj[0])),
                                    int(round(10*del_x)))]

    y_1=[round(i/10,1) for i in range(0,int(round(10*obj[1])),int(
                                    round(10*del_y)))]
    y_2=bndry_circle_pts_y_cord
    y_3=[round(i/10,1) for i in range(int(round(10*(obj[1]+del_y))
                                    ),int(10*(b+del_y)),int(round
                                    (10*del_y)))]

    y_list=list(set(y_1+y_2+y_3))
    y_list.sort()

    my_lst=[]
    for i in range(len(y_list) - 1):
        for j in range(len(x_list) - 1):
            tri1 = [(x_list[j], y_list[i]), (x_list[j+1], y_list[i
                                                  ]), (x_list[j],
                                                  y_list[i+1])]
            tri2 = [(x_list[j+1], y_list[i]), (x_list[j+1], y_list
                                                  [i+1]), (x_list[j],
                                                  y_list[i+1])]
            my_lst.append(tri1)
            my_lst.append(tri2)
```

```
    return my_lst

def mesh_2(obj,l,b,del_x,del_y,r,bndry_circle_pts_x_cord):
    x_list=bndry_circle_pts_x_cord

    y_list=[round(i/10,1) for i in range(0,int(round(10*obj[1])),
                                     int(round(10*del_y)))]

    my_lst=[]
    for i in range(len(y_list) - 1):
        for j in range(len(x_list) - 1):
            tri1 = [(x_list[j], y_list[i]), (x_list[j+1], y_list[i
                                         ]), (x_list[j],
                                         y_list[i+1])]
            tri2 = [(x_list[j+1], y_list[i]), (x_list[j+1], y_list
                                         [i+1]), (x_list[j],
                                         y_list[i+1])]
            my_lst.append(tri1)
            my_lst.append(tri2)
    return my_lst

def mesh_3(obj,l,b,del_x,del_y,r,bndry_circle_pts_y_cord):
    x_list=[round(i/10,1) for i in range(int(round(10*(obj[0]+
                                     del_x))),int(round(10*(l+
                                     del_x))),int(round(10*del_x))
                                     )]

    y_1=[round(i/10,1) for i in range(0,int(round(10*obj[1])),int(
                                     round(10*del_y)))]
    y_2=bndry_circle_pts_y_cord
    y_3=[round(i/10,1) for i in range(int(round(10*(obj[1]+del_y))
                                     ),int(10*(b+del_y)),int(round
                                     (10*del_y)))]

    y_list=list(set(y_1+y_2+y_3))
    y_list.sort()

    my_lst=[]
    for i in range(len(y_list) - 1):
```

```
        for j in range(len(x_list) - 1):
            tri1 = [(x_list[j], y_list[i]), (x_list[j+1], y_list[i
                                                ]), (x_list[j],
                                                y_list[i+1])]
            tri2 = [(x_list[j+1], y_list[i]), (x_list[j+1], y_list
                                                [i+1]), (x_list[j],
                                                y_list[i+1])]
            my_lst.append(tri1)
            my_lst.append(tri2)
    return my_lst



def mesh_4(obj,l,b,del_x,del_y,r,bndry_circle_pts_x_cord):
    x_list=bndry_circle_pts_x_cord

    y_list=[round(i/10,1) for i in range(int(round(10*(obj[1]+
                                    del_y))),int(round(10*(b+
                                    del_y))),int(round(10*del_y))
                                    )]

    my_lst=[]
    for i in range(len(y_list) - 1):
        for j in range(len(x_list) - 1):
            tri1 = [(x_list[j], y_list[i]), (x_list[j+1], y_list[i
                                                ]), (x_list[j],
                                                y_list[i+1])]
            tri2 = [(x_list[j+1], y_list[i]), (x_list[j+1], y_list
                                                [i+1]), (x_list[j],
                                                y_list[i+1])]
            my_lst.append(tri1)
            my_lst.append(tri2)

    return my_lst



def mesh_5(obj,l,b,del_x,del_y,r,bndry_circle_pts_x_cord,
                                bndry_circle_pts_y_cord):
    x_list=bndry_circle_pts_x_cord[0:4]
    y_list=bndry_circle_pts_y_cord[0:4]
```

```
    my_lst=[]
    k=0
    for i in range(len(y_list) - 1):
        for j in range(len(x_list)-1-k):
            if j == range(len(x_list)-1-k)[-1]:
                tri1 = [(x_list[j], y_list[i]), (x_list[j+1],
                                                  y_list[i]), (
                                                  x_list[j], y_list
                                                  [i+1])]
                my_lst.append(tri1)
            else:
                tri1 = [(x_list[j], y_list[i]), (x_list[j+1],
                                                  y_list[i]), (
                                                  x_list[j], y_list
                                                  [i+1])]
                my_lst.append(tri1)
                tri2 = [(x_list[j+1], y_list[i]), (x_list[j+1],
                                                  y_list[i+1]), (
                                                  x_list[j], y_list
                                                  [i+1])]
                my_lst.append(tri2)
        k=k+1
    return my_lst

def mesh_6(obj,l,b,del_x,del_y,r,bndry_circle_pts_x_cord,
                                  bndry_circle_pts_y_cord):
    x_list=bndry_circle_pts_x_cord[3:7]
    x_list.reverse()
    y_list=bndry_circle_pts_y_cord[0:4]

    my_lst=[]
    k=0
    for i in range(len(y_list) - 1):
        for j in range(len(x_list)-1-k):
            if j == range(len(x_list)-1-k)[-1]:
                tri1 = [(x_list[j], y_list[i]), (x_list[j+1],
                                                  y_list[i]), (
                                                  x_list[j], y_list
```

```
                                          [i+1])]
                my_lst.append(tri1)
            else:
                tri1 = [(x_list[j], y_list[i]), (x_list[j+1],
                                          y_list[i]), (
                                          x_list[j], y_list
                                          [i+1])]
                my_lst.append(tri1)
                tri2 = [(x_list[j+1], y_list[i]), (x_list[j+1],
                                          y_list[i+1]), (
                                          x_list[j], y_list
                                          [i+1])]
                my_lst.append(tri2)
        k=k+1
    return my_lst

def mesh_7(obj,l,b,del_x,del_y,r,bndry_circle_pts_x_cord,
                                bndry_circle_pts_y_cord):
    x_list=bndry_circle_pts_x_cord[0:4]
    # x_list.reverse()
    y_list=bndry_circle_pts_y_cord[3:7]
    y_list.reverse()

    my_lst=[]
    k=0
    for i in range(len(y_list) - 1):
        for j in range(len(x_list)-1-k):
            if j == range(len(x_list)-1-k)[-1]:
                tri1 = [(x_list[j], y_list[i]), (x_list[j+1],
                                          y_list[i]), (
                                          x_list[j], y_list
                                          [i+1])]
                my_lst.append(tri1)
            else:
                tri1 = [(x_list[j], y_list[i]), (x_list[j+1],
                                          y_list[i]), (
                                          x_list[j], y_list
                                          [i+1])]
                my_lst.append(tri1)
```

```
                tri2 = [(x_list[j+1], y_list[i]), (x_list[j+1],
                                                 y_list[i+1]), (
                                                 x_list[j], y_list
                                                 [i+1])]
                my_lst.append(tri2)
        k=k+1
    return my_lst

def mesh_8(obj,l,b,del_x,del_y,r,bndry_circle_pts_x_cord,
                                bndry_circle_pts_y_cord):
    x_list=bndry_circle_pts_x_cord[3:7]
    x_list.reverse()
    y_list=bndry_circle_pts_y_cord[3:7]
    y_list.reverse()

    my_lst=[]
    k=0
    for i in range(len(y_list) - 1):
        for j in range(len(x_list)-1-k):
            if j == range(len(x_list)-1-k)[-1]:
                tri1 = [(x_list[j], y_list[i]), (x_list[j+1],
                                                 y_list[i]), (
                                                 x_list[j], y_list
                                                 [i+1])]
                my_lst.append(tri1)
            else:
                tri1 = [(x_list[j], y_list[i]), (x_list[j+1],
                                                 y_list[i]), (
                                                 x_list[j], y_list
                                                 [i+1])]
                my_lst.append(tri1)
                tri2 = [(x_list[j+1], y_list[i]), (x_list[j+1],
                                                 y_list[i+1]), (
                                                 x_list[j], y_list
                                                 [i+1])]
                my_lst.append(tri2)
        k=k+1
    return my_lst
```

solutionK.py

```
import numpy as np

def global_dict(mesh):
    my_dict={}
    my_lst=list(set([item for lst in mesh for item in lst]))
    k=1
    for obj in my_lst:
        my_dict[obj]=k
        k=k+1
    return my_dict



def area_of_triangle(lst): # input:[(),(),()] output:area
     n_array=np.array([[lst[0][0],lst[0][1],1],[lst[1][0],lst[1][1]
                                           ,1],[lst[2][0],lst[2][1],1]])
     det = np.linalg.det(n_array)
     area=0.5*det
     return area

def B_elemental(lst): # input:[(),(),()] output: B elemental
                                  matrix of order (2*3)
     n_array=np.array([[lst[1][1]-lst[2][1],lst[2][1]-lst[0][1],lst
                                           [0][1]-lst[1][1]],[lst[2][0]-
                                           lst[1][0],lst[0][0]-lst[2][0]
                                           ,lst[1][0]-lst[0][0]]])
     div=2*area_of_triangle(lst)
     n_array=n_array/div
     return n_array



def k_elemental(lst): # input:[(),(),()] output: local k elemental
                                   matrix of order (3*3)
     array_1=B_elemental(lst)
     array_2=np.transpose(array_1)
     array_3=np.matmul(array_2,array_1)

     area=area_of_triangle(lst)
     array_3=area*array_3
```

```python
    return array_3



def Le_elemental(lst,glob_dict):# input:[(),(),()] output: local
                                    Le elemental matrix of order (3*
                                    338)
    a=np.zeros((3,len(glob_dict)))
    for i in range(0,3):
      u=glob_dict[lst[i]]
      a[i,u-1]=1
    return a



def k_global(mesh,glob_dict):
   k=np.zeros((len(glob_dict),len(glob_dict)))

   for obj in mesh:
      le=Le_elemental(obj,glob_dict)
      le_trans=np.transpose(le)
      ke=k_elemental(obj)
      k=k+np.matmul(le_trans,np.matmul(ke,le))

   return k
```

solutionF.py

```
import numpy as np
import math

def val_dict(col_vector, input_dict):
    keys = list(input_dict.keys())
    values = col_vector.flatten().tolist()
    return {keys[i]: values[i] for i in range(len(keys))}

def Me_elemental(lst):
    new_array=[[1,lst[0][0],lst[0][1]],[1,lst[1][0],lst[1][1]],[1,
                                        lst[2][0],lst[2][1]]]
    return new_array

def Le_elemental(lst,glob_dict):# input:[(),(),()] output: local
                                   Le elemental matrix of order (3*
                                   len(glob_dict))
    a=np.zeros((3,len(glob_dict)))
    for i in range(0,3):
      u=glob_dict[lst[i]]
      a[i,u-1]=1
    return a

def f_global(mesh,glob_dict,v_left,v_right,l):
    f=np.zeros((len(glob_dict),1))

    new_lst=[]
    for obj in mesh:
        a=[ele[0] for ele in obj]
        if a.count(0) == 2:
            new_lst.append(obj)

    for obj in new_lst:
        a_1=obj[2][1]-obj[0][1]
        a_2=0.5*(obj[2][1]-obj[0][1])*(obj[2][1]+obj[0][1])
        arr=np.array([[a_1],[0],[a_2]])
        minus_q_bar=(1)*np.dot(np.array(v_left),np.array([-1,0]))
        me=Me_elemental(obj)
        me_inverse=np.linalg.inv(me)
```

```
        me_inverse_trans=np.transpose(me_inverse)
        le=Le_elemental(obj,glob_dict)
        le_trans=np.transpose(le)
        arr=minus_q_bar*arr
        f=f+np.matmul(le_trans,np.matmul(me_inverse_trans,arr))

    new_lst=[]
    for obj in mesh:
        a=[ele[0] for ele in obj]
        if a.count(l) == 2:
            new_lst.append(obj)

    for obj in new_lst:
        a_1=obj[1][1]-obj[0][1]
        a_2=0.5*(obj[1][1]-obj[0][1])*(obj[1][1]+obj[0][1])
        arr=np.array([[a_1],[l*a_1],[a_2]])
        minus_q_bar=(1)*np.dot(np.array(v_right),np.array([1,0]))
        me=Me_elemental(obj)
        me_inverse=np.linalg.inv(me)
        me_inverse_trans=np.transpose(me_inverse)
        le=Le_elemental(obj,glob_dict)
        le_trans=np.transpose(le)
        arr=minus_q_bar*arr
        f=f+np.matmul(le_trans,np.matmul(me_inverse_trans,arr))

    return f
```

solutionFDM.py

```
import numpy as np
import math

def f_x(obj,l,b,del_x,del_y,value_dict,bndry_circle_pts_y_cord):
    bndry_circle_pts_y_cord_copy=bndry_circle_pts_y_cord.copy()
    my_dict={}

    y_1=[round(i/10,1) for i in range(0,int(round(10*obj[1])),int(
                                    round(10*del_y)))]
    y_3=[round(i/10,1) for i in range(int(round(10*(obj[1]+del_y))
                                    ),int(10*(b+del_y)),int(round
                                    (10*del_y)))]
    y=y_1+y_3

    for ele in y:
        new_dict={key[0]:value for key,value in value_dict.items()
                                            if key[1] == ele}
        sorted_dict = {k: v for k, v in sorted(new_dict.items())}
        x_list=np.array(list(sorted_dict.keys()))
        z_list=np.array(list(sorted_dict.values()))
        dz_dx=np.gradient(z_list,x_list)
        my_dict.update({(x,ele):z for x,z in zip(x_list,dz_dx)})

    bndry_circle_pts_y_cord_copy.pop(-1)
    bndry_circle_pts_y_cord_copy.pop(0)

    for ele in bndry_circle_pts_y_cord_copy:
        new_dict={key[0]:value for key,value in value_dict.items()
                                            if key[1] == ele and key
                                           [0] < obj[0] }
        sorted_dict = {k: v for k, v in sorted(new_dict.items())}
        x_list=np.array(list(sorted_dict.keys()))
        z_list=np.array(list(sorted_dict.values()))
        dz_dx=np.gradient(z_list,x_list)
        my_dict.update({(x,ele):z for x,z in zip(x_list,dz_dx)})

    for ele in bndry_circle_pts_y_cord_copy:
        new_dict={key[0]:value for key,value in value_dict.items()
```

```
                                                  if key[1] == ele and key
                                                  [0] > obj[0] }
        sorted_dict = {k: v for k, v in sorted(new_dict.items())}
        x_list=np.array(list(sorted_dict.keys()))
        z_list=np.array(list(sorted_dict.values()))
        dz_dx=np.gradient(z_list,x_list)
        my_dict.update({(x,ele):z for x,z in zip(x_list,dz_dx)})

    return my_dict

def f_y(obj,l,b,del_x,del_y,value_dict,bndry_circle_pts_x_cord):
    my_dict={}
    bndry_circle_pts_x_cord_copy=bndry_circle_pts_x_cord.copy()

    x_1=[round(i/10,1) for i in range(0,int(round(10*obj[0])),int(
                                  round(10*del_x)))]
    x_3=[round(i/10,1) for i in range(int(round(10*(obj[0]+del_x))
                                  ),int(10*(l+del_x)),int(round
                                  (10*del_x)))]
    x=x_1+x_3

    for ele in x:
        new_dict={key[1]:value for key,value in value_dict.items()
                                                  if key[0] == ele}
        sorted_dict = {k: v for k, v in sorted(new_dict.items())}
        y_list=np.array(list(sorted_dict.keys()))
        z_list=np.array(list(sorted_dict.values()))
        dz_dy=np.gradient(z_list,y_list)
        my_dict.update({(ele,y):z for y,z in zip(y_list,dz_dy)})

    bndry_circle_pts_x_cord_copy.pop(-1)
    bndry_circle_pts_x_cord_copy.pop(0)

    for ele in bndry_circle_pts_x_cord_copy:
        new_dict={key[1]:value for key,value in value_dict.items()
                                                  if key[0] == ele and key
                                                  [1] < obj[1] }
        sorted_dict = {k: v for k, v in sorted(new_dict.items())}
        y_list=np.array(list(sorted_dict.keys()))
```

```
        z_list=np.array(list(sorted_dict.values()))
        dz_dy=np.gradient(z_list,y_list)
        my_dict.update({(ele,y):z for y,z in zip(y_list,dz_dy)})

    for ele in bndry_circle_pts_x_cord_copy:
        new_dict={key[1]:value for key,value in value_dict.items()
                                                  if key[0] == ele and key
                                                 [1] > obj[1] }
        sorted_dict = {k: v for k, v in sorted(new_dict.items())}
        y_list=np.array(list(sorted_dict.keys()))
        z_list=np.array(list(sorted_dict.values()))
        dz_dy=np.gradient(z_list,y_list)
        my_dict.update({(ele,y):z for y,z in zip(y_list,dz_dy)})

    return my_dict

def dp_dx(v_x,v_y,dv_x_dx,dv_x_dy):
    rho=100
    my_dict={}

    for key in v_x.keys():
        val=(-1)*rho*((v_x[key]*dv_x_dx[key])+(v_y[key]*dv_x_dy[
                                                 key]))
        my_dict[key]=val

    return my_dict

def dp_dy(v_x,v_y,dv_y_dx,dv_y_dy):
    rho=100
    my_dict={}

    for key in v_x.keys():
        val=(-1)*rho*((v_x[key]*dv_y_dx[key])+(v_y[key]*dv_y_dy[
                                                 key]))
        my_dict[key]=val

    return my_dict

def pressure(obj,l,b,del_x,del_y,dp_dx_dict,dp_dy_dict,
```

```
                                            bndry_circle_pts ,P):
    my_dict={}

    new_dict={key[0]:value for key,value in dp_dx_dict.items() if
                                            key[1] == obj[1] and key[0] >
                                            obj[0]}
    sorted_dict = {k: v for k, v in sorted(new_dict.items())}
    sorted_dict.popitem()
    # print(sorted_dict)

    for key in reversed(sorted_dict.keys()):
        P=P-(sorted_dict[key]*del_x)
    # print(P)

    for i in range(len(bndry_circle_pts)):
        if i==0:
            my_dict.update({bndry_circle_pts[i]:P})
        else:
            P=P-((dp_dx_dict[bndry_circle_pts[i]]*del_x)+(
                                                dp_dy_dict[
                                                bndry_circle_pts[i]]*
                                                del_y))
            my_dict.update({bndry_circle_pts[i]:P})

    return my_dict

def force(press_dict,r):
    length=(r*math.pi)/6
    force_arr=np.array([0,0])
    theta=0
    for key,value in press_dict.items():
        F = value*length
        force_arr=force_arr+np.array([1*F*math.cos(theta),-1*F*
                                        math.sin(theta)])
        theta=theta+((math.pi)/6)
    return force_arr.tolist()
```

csvCodes.py

```
import csv

def dictionary_to_csv(d, filename):
    with open(filename, 'w', newline='') as csvfile:
        writer = csv.writer(csvfile)
        writer.writerow(['x', 'y', 'value'])
        for key, value in d.items():
            x, y = key
            writer.writerow([x, y, value])

def force_to_csv(xy_tuple, ab_list,file_name):
    with open(file_name, 'a', newline='') as file:
        writer = csv.writer(file)
        writer.writerow(list(xy_tuple) + ab_list)
```

plots.py

```
import matplotlib.pyplot as plt
import numpy as np
import random

def combine(dict_1,dict_2):
    my_dict={}
    for key in dict_1.keys():
        my_dict[key]=(dict_1[key],dict_2[key])
    return my_dict

def plot_triangles(triangles, filename):
    plt.clf()
    for triangle in triangles:
        x = [point[0] for point in triangle]
        y = [point[1] for point in triangle]
        plt.plot(x + [x[0]], y + [y[0]], 'k-')
    plt.axis('equal')  # ensures that x and y scales are equal
    plt.savefig(filename, dpi=150, bbox_inches='tight')
    # print(f"Plot saved as {filename}")

# example dictionary
def velo_plot(velocity_dict,obj,l,b):
    plt.clf()

    # calculate 20% of the total number of keys in the dictionary
    n = len(velocity_dict)
    n_20_percent = int(n * 1)

    # randomly select that many keys from the dictionary
    selected_keys = random.sample(list(velocity_dict.keys()),
                                  n_20_percent)

    # create the plot
    plt.xlabel('x (meter)')
    plt.ylabel('y (meter)')
    plt.xlim(0, l)
    plt.ylim(0, b)
```

```
# create a list of colors for the arrows
colors = plt.cm.rainbow(np.linspace(0, 1, len(selected_keys)))

# iterate over the selected keys and plot the velocity vectors
for i, key in enumerate(selected_keys):
    vx, vy = velocity_dict[key]
    scaling_factor = 0.00005
    color = colors[i]
    plt.arrow(key[0], key[1], vx*scaling_factor, vy*
                                        scaling_factor,
                                        head_width=0.01,
                                        head_length=0.05, fc=
                                        color, ec=color)

plt.savefig('velo_plot_png/'+'vel_'+str(obj)+'.png')
```

path.py

```
def eliminate(dictionary):
    new_dictionary = {}
    for key, value in dictionary.items():
        if len(value) >= 2: # 2 if no of output points is 2
            new_dictionary[key] = value
    return new_dictionary



def eliminate_key_value_pairs(d, l, b):
    new_dict = {}
    for key, value in d.items():
        # Check if any tuple in the value list violates the
                                        conditions
        if any(x > l or y > b for x, y in value):
            continue
        else:
            new_dict[key] = value
    return new_dict

def eliminate_duplicates(dictionary):
    new_dict = {}
    for key, value in dictionary.items():
        new_list = [key] + value
        i = 0
        while i < len(new_list) - 1:
            if new_list[i] == new_list[i+1]:
                break
            i += 1
        else:
            new_dict[key] = value
    return new_dict

def path(force_dict,m,l,b):
    # m=mass
    acc_dict={(x, y): [f_x/m, f_y/m] for (x, y), [f_x, f_y] in
                                  force_dict.items()}

    my_dict={}
```

```
for key,value in acc_dict.items():
    del_t=0.9 # 1 sec
    v_x,v_y=0,0
    x_initial,y_initial=key[0],key[1]
    accn_x=value[0]
    accn_y=value[1]

    my_lst=[]
    for i in range(2): # output no of points = 2
        x_new=x_initial+(v_x*del_t)+(0.5*(accn_x*del_t**2))
        y_new=y_initial+(v_y*del_t)+(0.5*(accn_y*del_t**2))
        v_x_new=v_x+(accn_x*del_t)
        v_y_new=v_x+(accn_y*del_t)
        tple=(round(x_new,1),round(y_new,1))

        tple_new=tple
        my_lst.append(tple_new)

        v_x=v_x_new
        v_y=v_y_new
        x_initial=tple_new[0]
        y_initial=tple_new[1]

        try:
            accn_x=acc_dict[(x_initial,y_initial)][0]
            accn_y=acc_dict[(x_initial,y_initial)][1]
        except:
            break
    my_dict[key]=my_lst

my_dict=eliminate(my_dict) # if less than 2 elements in value
                                 list , eliminate that key,
                                 value pair
print(len(my_dict))
my_dict=eliminate_key_value_pairs(my_dict,l,b) # if any value
                                 in list is greater than l or
                                 b eliminate that key,value
                                 pair
print(len(my_dict))
```

```
    my_dict=eliminate_duplicates(my_dict)
    print(len(my_dict))

    return my_dict
```

loadData.py

```
import numpy as np
import random



def load_data(my_dict):

    list_keys=list(my_dict.keys())
    rndm_list_keys=random.sample(list_keys, int(0.8*len(list_keys)
                                    )) # 80 % represents amount
                                    of data objects you want in
                                    training phase
    other_list_keys=list(set(list_keys)-set(rndm_list_keys))

    training_dict={k:my_dict[k] for k in rndm_list_keys}
    test_dict={k:my_dict[k] for k in other_list_keys}

    training_inputs=[np.array(x).reshape(2,1) for x in
                                    training_dict.keys()]
    training_results=[np.array(y).reshape(2*len(y),1) for y in
                                    training_dict.values()]
    training_data=list(zip(training_inputs,training_results))

    test_inputs=[np.array(x).reshape(2,1) for x in test_dict.keys
                                    ()]
    test_results=[np.array(y).reshape(2*len(y),1) for y in
                                    test_dict.values()]
    test_data=list(zip(test_inputs,test_results))

    return (training_data,test_data,test_dict)

if __name__=='__main__':
    load_data(my_dict,l,b)
```

network.py

```
import numpy as np
import random
import os.path

class Network(object):

    def __init__(self,sizes):
        self.num_layers=len(sizes)
        self.sizes = sizes

        if os.path.exists('neural_net/biases.npy'):
            self.biases=[]
            with open('neural_net/biases.npy', 'rb') as f:
                self.biases.append(np.load(f))
                self.biases.append(np.load(f))
        else:
            self.biases = [np.random.randn(y, 1) for y in sizes[1:
                                                ]]

        if os.path.exists('neural_net/weights.npy'):
            self.weights=[]
            with open('neural_net/weights.npy', 'rb') as f:
                self.weights.append(np.load(f))
                self.weights.append(np.load(f))
        else:
            self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]

    def SGD(self, training_data, epochs, mini_batch_size, eta,
                                        test_data=None):
        if test_data: n_test = len(test_data)
        n = len(training_data)
        for j in range(epochs):
            random.shuffle(training_data)
            mini_batches = [
                training_data[k:k+mini_batch_size]
                for k in range(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
```

```
                self.update_mini_batch(mini_batch, eta)
            if test_data:
                print("Epoch {0}: {1} / {2}".format(
                    j, self.evaluate(test_data), n_test))
            else:
                print("Epoch {0} complete".format(j))

            # saving last biases and weights in file
            if j==epochs-1:
                with open('neural_net/biases.npy', 'wb') as f:
                    for obj in self.biases:
                        np.save(f,obj)

                with open('neural_net/weights.npy', 'wb') as f:
                    for obj in self.weights:
                        np.save(f,obj)

    def update_mini_batch(self, mini_batch, eta):
        """Update the network's weights and biases by applying
        gradient descent using backpropagation to a single mini
                                            batch.
        The ``mini_batch`` is a list of tuples ``(x, y)``, and ``
                                            eta``
        is the learning rate."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b,
                                                delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w,
                                                delta_nabla_w)]
        self.weights = [w-(eta/len(mini_batch))*nw
                        for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b-(eta/len(mini_batch))*nb
                       for b, nb in zip(self.biases, nabla_b)]

    def backprop(self, x, y):
        """Return a tuple ``(nabla_b, nabla_w)`` representing the
```

```
        gradient for the cost function C_x.  ``nabla_b`` and
        ``nabla_w`` are layer-by-layer lists of numpy arrays,
                                         similar
        to ``self.biases`` and ``self.weights``."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # feedforward
        activation = x
        activations = [x] # list to store all the activations,
                                         layer by layer
        zs = [] # list to store all the z vectors, layer by layer
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = sigmoid(z)
            activations.append(activation)
        # backward pass
        delta = self.cost_derivative(activations[-1], y) * \
            sigmoid_prime(zs[-1])
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
        # Note that the variable l in the loop below is used a
                                         little
        # differently to the notation in Chapter 2 of the book.
                                         Here,
        # l = 1 means the last layer of neurons, l = 2 is the
        # second-last layer, and so on.  It's a renumbering of the
        # scheme in the book, used here to take advantage of the
                                         fact
        # that Python can use negative indices in lists.
        for l in range(2, self.num_layers):
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta)
                                             * sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].
                                             transpose())
        return (nabla_b, nabla_w)
```

```
    def cost_derivative(self, output_activations, y):
        """Return the vector of partial derivatives \partial C_x /
        \partial a for the output activations."""
        return (output_activations-y)

    def evaluate(self, test_data):
        test_results = [(self.feedforward(x), y)
                        for (x, y) in test_data]

        j=0
        for x,y in test_results:
            if np.array_equal(x,y):
                j=j+1
        return j
        # changes to be made wrt to allowd thresold for each
                                            noumber in output column,
                                             right now it should be
                                            equal

    def feedforward(self, a):
        """Return the output of the network if ``a`` is input."""
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a


#miscalllenous function

def sigmoid(z):
    """The sigmoid function."""
    return 1.0/(1.0+np.exp(-z))

def sigmoid_prime(z):
    """Derivative of the sigmoid function."""
    return sigmoid(z)*(1-sigmoid(z))
```

constrained.py

```
def constrain_dict(dictionary):
    # Find the minimum and maximum values for the keys and values
    all_coords = []
    for key, tuple_list in dictionary.items():
        all_coords.append(key[0])
        all_coords.append(key[1])
        for original_tuple in tuple_list:
            all_coords.append(original_tuple[0])
            all_coords.append(original_tuple[1])
    minimum = min(all_coords)
    maximum = max(all_coords)

    # Constrain the keys and values using the minimum and maximum
                                                    values
    constrained_dict = {}
    for key, tuple_list in dictionary.items():
        constrained_key = ((key[0] - minimum) / (maximum - minimum
                                    ), (key[1] - minimum) / (
                                    maximum - minimum))
        constrained_tuple_list = []
        for original_tuple in tuple_list:
            constrained_x = (original_tuple[0] - minimum) / (
                                        maximum - minimum)
            constrained_y = (original_tuple[1] - minimum) / (
                                        maximum - minimum)
            constrained_tuple_list.append((constrained_x,
                                        constrained_y))
        constrained_dict[constrained_key] = constrained_tuple_list
    return (constrained_dict,minimum,maximum)



def unconstrain_dict(constrained_dict, minimum, maximum):
    original_dict = {}
    for constrained_key, constrained_tuple_list in
                                    constrained_dict.items():
        original_key_x = constrained_key[0] * (maximum - minimum)
                                    + minimum
        original_key_y = constrained_key[1] * (maximum - minimum)
```

```
                                                    + minimum
        original_key = (original_key_x, original_key_y)
        original_tuple_list = []
        for constrained_tuple in constrained_tuple_list:
            original_x = constrained_tuple[0] * (maximum - minimum
                                                ) + minimum
            original_y = constrained_tuple[1] * (maximum - minimum
                                                ) + minimum
            original_tuple_list.append((original_x, original_y))
        original_dict[original_key] = original_tuple_list
    return original_dict
```

runNetwork.py

```
# Always delete weights.npy and biases.npy, before running every program
import pickle

with open('dict_for_NN_3.pickle', 'rb') as f:
    dict_for_NN = pickle.load(f)
    print('No of dataset:',len(dict_for_NN))



from constraint_and_unconstrain_dict import constrain_dict, unconstrain_dict
dict_for_NN_constrained,minimum,maximum=constrain_dict(dict_for_NN)


from neural_net  import data_loader
from neural_net import network
from neural_net import in_out

training_data,test_data,test_dict = data_loader.load_data(dict_for_NN_constrained)
print('No of training data',len(training_data))



net=network.Network([2,30,4]) # input: a tuple =2; , middle layer=30; output: 5 tuples=2*5 points = 10
net.SGD(training_data,30,10,1.1)# epochs, mini_batch_size, learning rate

dict_by_NN_On_test_data_constrained=in_out.in_out(test_dict)

dict_by_NN_On_test_data_unconstrained=unconstrain_dict(dict_by_NN_On_test_data_constrained,minimum,maximum)
test_dict_unconstrained=unconstrain_dict(test_dict,minimum,maximum)
print(test_dict_unconstrained)
print(dict_by_NN_On_test_data_unconstrained)
```

```
from plot import create_plots,create_animations
create_plots(test_dict_unconstrained,3,1,1) # 3 for 3 by 3 picture
                                    , l=1,b=1 for x lim an y lim
```

Development of Neural Network to Predict path of an object in a 2 D potential flow

ORIGINALITY REPORT

12%

SIMILARITY INDEX

PRIMARY SOURCES

#	Source	Type	Words	Percentage
1	vdoc.pub	Internet	119 words	1%
2	Rong, Jian. "On weighted regression of time series for surveillance of influenza mortality", Proquest, 20111003	ProQuest	96 words	1%
3	Rahul Yedida, Snehanshu Saha. "Beginning with machine learning: a comprehensive primer", The European Physical Journal Special Topics, 2021	Crossref	80 words	1%
4	ia600202.us.archive.org	Internet	64 words	1%
5	exceptionshub.com	Internet	53 words	< 1%
6	Chun-Sheng Wang, Ying-Ho Liu, Kuo-Chung Chu. "Closed inter-sequence pattern mining", Journal of Systems and Software, 2013	Crossref	52 words	< 1%
7	programtalk.com	Internet	52 words	< 1%

IOE Graduate Conference
[Placeholder for Publication Information]

Development of a Neural Network to Predict Path of an Object in a Two Dimensional Potential Flow

Rijan Niraula [a]

[a] *Department of Mechanical and Aerospace Engineering, Pulchowk Campus, IOE, Tribhuvan University*
niraularijan080@gmail.com

Abstract
Neural networks have been widely used in various fields, including fluid dynamics, to predict complex phenomena that are difficult to model analytically. In this research, a neural network is developed to predict the path taken by a circular body in a two-dimensional fluidic domain. The study involves simulating the potential flow over a rectangular domain inside which a circular body is placed. Fluctuations in different parameters such as pressure, forces, and velocity field during the motion of the body are studied. The Laplace equation is solved at each time step by applying the techniques of finite element method (FEM) to obtain accurate data, which is fed into the neural network. The neural network comprises of three layers input , middle and output layer. The study is carrried out using computational methods that relies on open-source software Python and its modules like NumPy. The results of the neural network's predictions are compared with accurate data to analyze the error. Fluctutaion of error with respect to different hyperparameters of the network is calculated and accordingly suitable hyperparameters of the network are determined.

Keywords
Neural Network, FEM, Potential Flow, Trajectory

1. Introduction

Fluid dynamics field involves solving set of Partial Differential Equations (PDE) for understanding the behavior of fluids. Most of the problems in fluid dynamics are complicated and difficult to solve. In most cases, these equations cannot be solved analytically. While experimental methods can be used for accurate results, they are expensive and require specialized facilities. As a result, computational methods have become popular, such as Finite Element Method (FEM) and Finite Difference Method (FDM). These methods transform complex differential equations into algebraic equations, which can be solved with the help of computers [1]. A newer method that combines modern numerical techniques and high-speed digital computers is the use of neural networks to solve PDEs. [2]

In this research, the focus is on development and use of neural networks to predict the path of a circular body in a fluidic domain by solving the Laplace equation at each time step and feeding the obtained data into a neural network . The two dimensional fluid domain is assumed to be inviscid, incompressible and irrotational. The neural network is based on gradient descent algorithm and is able to predict the path of the body given its initial location in two dimensional domain [3] . Data is generated by numerically solving laplace equation [4] using the techniques of finite element method (FEM), thus simulating a potential flow over the rectangular domain and analyzing fluctuations in different parameters during motion of the body. Evaluation of the error between the actual and predicted path by the network is also carried out and finally, hyperparameters of the network are tweaked accordingly to minimise error in the network.

2. Research Methodology

The study was divided into two parts: Calculation of field information using finite element method (FEM) and development of the neural network. Both analysis are carried out in Python [5] using python module like Numpy [6]. In FEM, strong form and weak form of the governing equations are developed and are applied to each cell of the mesh tranforming the PDE into the set of algebraic linear equations. For incompressible, inviscid, and steady state irrotational flow, governing

equations [7] for fluid flow are given by equation 1 and equation 2.

$$\nabla^2\phi = 0 \text{ with } \vec{V} = \nabla\phi \tag{1}$$

$$\rho(\vec{V}.\nabla)\vec{V} = -\nabla \mathrm{p} + \rho\mathbf{g} \tag{2}$$

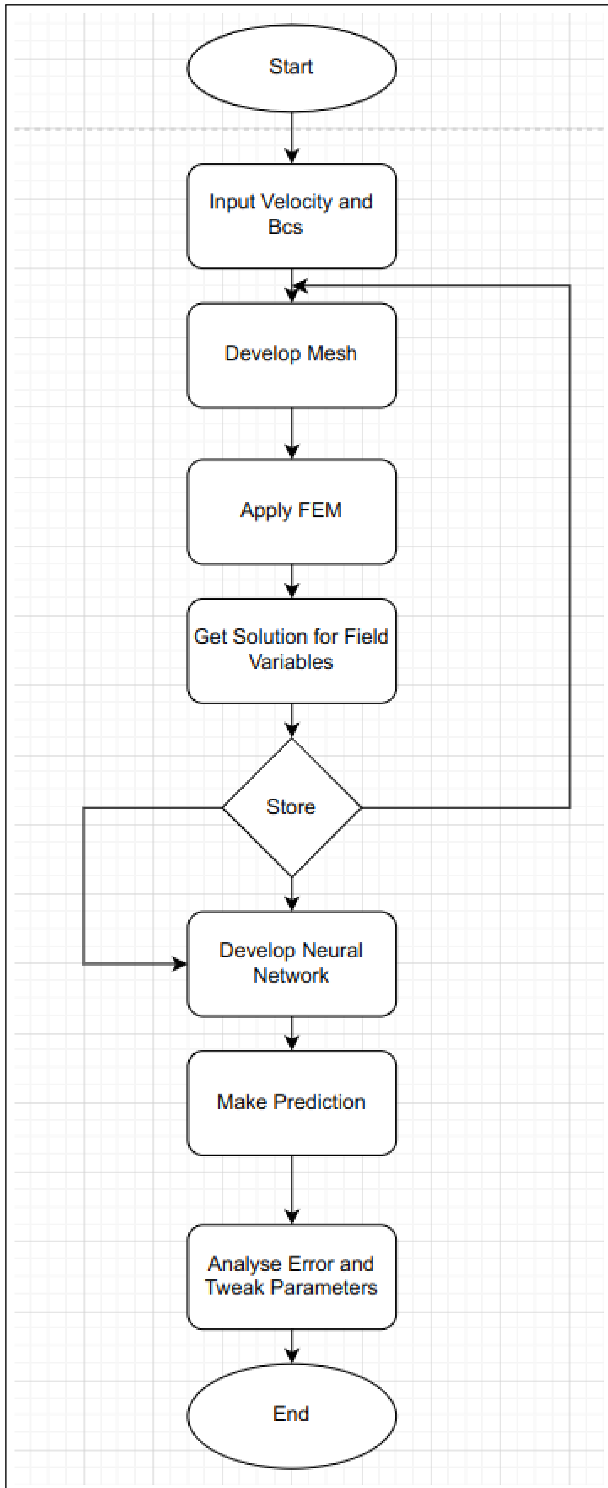

Figure 1: Methodology chart

Once the input parameter of the problem are specified in the program, the program creates a rectangular flow domain, and a mesh is generated over it. After the users input all the required information, like boundary and wall condition, the program prepares the mesh structure, applies the wall and boundary condition to solve the problem and obtain solution of the potential function $\phi(x,y)$. The technique of finite element method (FEM) is used to obtain this solution [8]. Such solution of the potential fucntion ϕ(x, y) is calculated by varying the location of the object throughout the domain. After obtaining the solution of the Laplace equation, the gradient ∇ is applied to potential fucntion $\phi(x,y)$ to obtain the velocity field information. The techniques of the Finite Difference Method (FDM) (Second Order) method are used to calculate the involved derivatives [9]. From the velocity field, the distribution of pressure is calculated throughout the domain using equation 2. The obtained data are stored, which will be further used to calculate actual trajectory of the body and later by a neural network for correcting its weights and biases for prediction purpose.

The next section involves the development of the three layered neural network i.e input, output and middle layer [10]. The stored data is divided into feed and test data, and the neural network is constructed by using the gradient descent algorithm. The predicted result for a given set of test data by the trained neural network is computed. Finally, the predicted path is compared against the actual path, and the observed errors are analyzed by varying various parameters of the neural network like learning rate, number of neurons in middle layer and number of data in the dataset.

3. Results

3.1 Development of the mesh

The region between the circle and the rectangular domain are meshed using triangular meshes, where each mesh cell can be represented in Python by a list of three tuples [11]. In addition, the object within the domain is placed at different locations, and meshes are created for each of these as well. This is necessary in order to accurately calculate the trajectory of each object, taking into account its position and movement within the meshed domain. The length and breadth of the rectangular domain is taken to be 1m each and radius of the body is taken as 0.1m. Such meshes are

shown in figure 2.

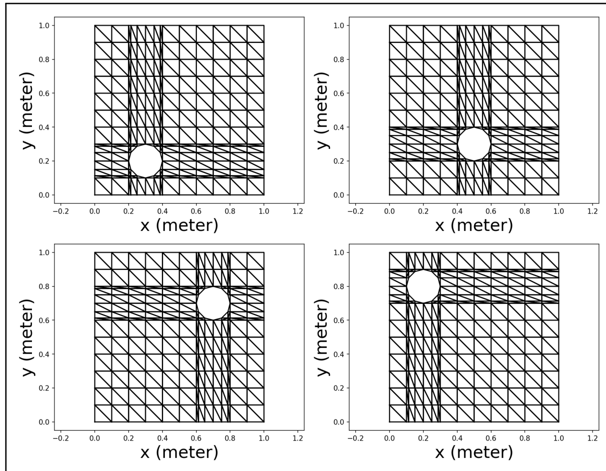

Figure 2: Mesh for different locations

3.2 Velocity Field

After applying the weak form of the governing equation to each cell in the mesh, one obtains the algebraic linear equation for the potential $\phi(x,y)$ function. Solving the linear equation gives the solution of potential function $\phi(x,y)$ over the whole domain. Taking its gradient, the velocity field information is obtained. One such velocity field graph is shown for a specific location.

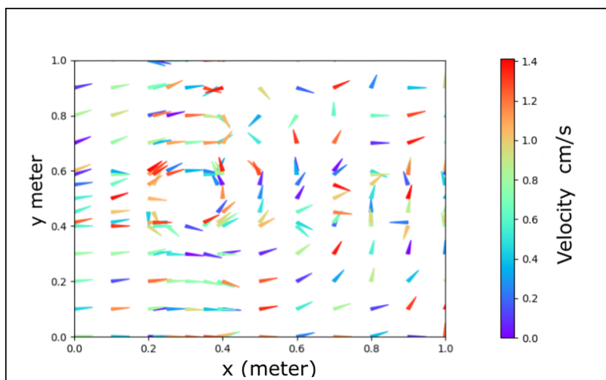

Figure 3: Velocity Field

3.3 Force and Path Trajectory

The pressure around the perimeter of the circular body is calculated via equation 2. The initial condition of the pressure is assumed to be at atmospheric pressure at the outlet. Integrating this pressure values along the perimeter of the body gives the force and accelaration experienced by the body at a specific location. Basic kinematic equations are used recursively to obtain the path co-ordinates of the body.

S.N	Start Position	Path Co-ordinates
1	(0.2,0.2)	(0.25,0.28),(0.31,0.5),(0.41,0.68)
2	(0.5,0.3)	(0.56,0.72),(0.83,0.53),(0.94,0.64)
3	(0.6,0.4)	(0.61,0.43),(0.64,0.50),(0.67,0.58)

Table 1: Path Co-ordinate table

3.4 Mean Square Error (MSE)

In neural networks, Mean Squared Error (MSE) [12] is a commonly used metric to evaluate the performance of the model. It measures the average squared difference between the predicted output and the actual output over all the samples in the dataset.

3.4.1 MSE vs Learning rate

The key step in optimizing the performance of the network is by seeing how MSE varies with different learning rate for the network. Learning rate is a hyperparameter of the network that determines the step size taken in the direction of the negative gradient during neural network training. By monitoring the MSE during training with different learning rates, one can determine an optimal learning rate that leads to the best performance on the given dataset.

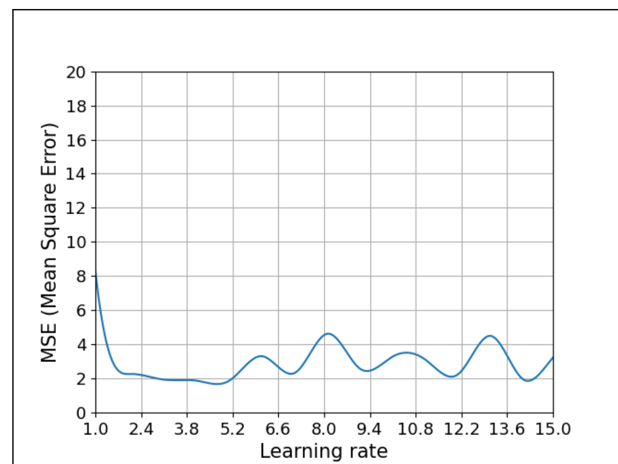

Figure 4: MSE vs Learning Rate

Figure 4 shows that MSE is minimum and stable when learning rate is in the range of 1.6-3.5. Thus learning rate of 2 is employed in the network. Learning rate fluctuates beyond the value of 3.5 which implies selection of learning rate from this region has chances of incurring high MSE for a random sample dataset.

3.4.2 MSE vs Number of Neurons

To understand the impact of the number of neurons present in the middle layer on the network performance, an examination of how the MSE varies with different numbers of neurons is carried out. A random shuffle of the dataset is done by selecting 80 percent of the data and calculating MSE vs number of neurons in middle layer for various dataset.

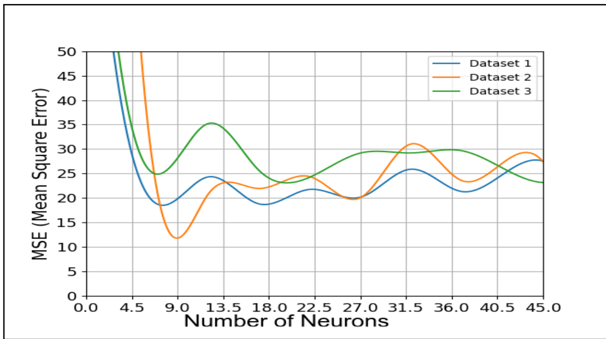

Figure 5: MSE vs Number of Neurons

Figure 5 shows that MSE is minimum, when number of neurons in middle layer is in the region of 10. Minimum of MSE also occurs at other values but higher number of neuron will imply higher computational time by the network. Thus, number of neurons in the middle layer is selected to be in the range of 8-12.

3.4.3 MSE vs Number of Dataset

An analysis on the impact of dataset size on model performance is calculated by noting how MSE varies for different numbers of data samples through which one can determine the minimum amount of data required for the model to achieve acceptable performance.

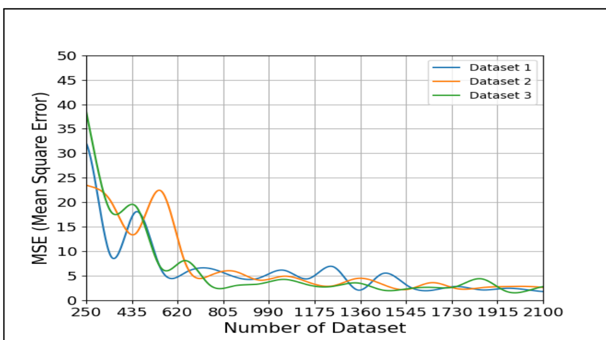

Figure 6: MSE vs Number of Data samples

Figure 6 shows that this number is in range of 2000 for the network.

4. Conclusion

Based on the numerical calculation of the research, a neural network capable of predicting path of an object in potential flow has been developed. Additionally, the optimal values for hyperparameters of the network are calculated. This hyperparameters range can guide guide the design of future neural network models for similar problems. The development of the neural network required numerical computation of the velocity and pressure fields, and the resulting model provides faster results compared to numerically solving the governing equations. These findings suggest that neural networks can be a valuable tool for accelerating the design and optimization of fluid dynamic systems. Overall, this research has demonstrated the potential of neural networks in predicting the path of an object in a potential flow, and provides a foundation for future research in this area.

References

[1] Yue Liu and Xiangdong Gao. Numerical simulation of fluid mechanics problems using fdm, fvm, and fem. *Procedia Engineering*, 99:2092–2100, 2015.

[2] Steven L Brunton and J Nathan Kutz. Neural networks for solving differential equations. *Annual Review of Fluid Mechanics*, 51:539–574, 2019.

[3] Michael A Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, USA, 2015.

[4] Xianming Dai and Yunjie Yang. The laplace equation and its solutions. *Procedia Computer Science*, 130:336–343, 2018.

[5] Naveen Kumar and Sheetal Taneja. Python programming: A modular approach, 2018.

[6] Ivan Idris. *NumPy*. Packt Publishing Ltd., Birmingham, UK, 2018.

[7] Yunus Cengel and John Cimbala. *Fluid Mechanics: Fundamentals and Applications*. McGraw-Hill Education, 2006.

[8] Ioannis Koutromanos. *Fundamentals of Finite Element Analysis: Linear Finite Element Analysis*. John Wiley & Sons, 2018.

[9] L Cheng. Finite difference methods for poisson equation, 2022.

[10] Chris M Bishop. Neural networks and their applications. *Review of scientific instruments*, 65(6):1803–1832, 1994.

[11] Cody J. Friesen. *Finite Element Mesh Generation with Python*, pages 89–102. Elsevier, 2018.

[12] Wei Shi and Charles Qi. Mean squared error: A review of applications in deep learning. *arXiv preprint*, 2021.