Coupling of Smoothed Particle Hydrodynamics solver with Machine Learning algorithm for prediction of fluid behavior

by

Won Tae Lee

A thesis submitted in partial fulfillment of the requirements for the degree of

Master's degree

Friedrich-Alexander-Universität Erlangen-Nürnberg

2021

Approved by ____________________

____

Program Authorized
to Offer Degree____________________

Date ____________________

Friedrich-Alexander-Universität Erlangen-Nürnberg

Abstract

MASTER'S DEGREE

By Won Tae Lee

LSTM: Prof.Delgado

A number of computational resources (CPU, GPU, and RAM) are required to simulate fluid flow in Computational Fluid Dynamics (CFD). In order to save the computational resources, Machine Learning (ML) has received much attention in recent years. ML can predict highly accurate solutions for partial differential equations with less computational resources. (Usman et al., 2021) The purpose of this study is to develop machine learning models for a reverse roll coating problem, which required intensive computation time in solving with Fluid Differential Method (FDM). The FDM will be replaced with Smoothed Particle Hydrodynamics (SPH), due to the reason that grid-based methods have limitations in expressing the interaction between objects and fluids. Additionally, with the help of ML, simulation results from SPH solver were further trained to create a prediction model. The input parameters of ML models were chosen as follows: rotating speed of two rolls, mass, kinematic viscosity, respectively. For the ML models training, simulation data was applied to predict fluid particles' positions, velocities, accelerations, density, and pressure at every time-step. Multivariable Linear Regression was used as an algorithm of the ML models. As a result, prediction accuracy of the ML models was strongly dependent on the results of simulation, notably number of particles' collisions with the rolls.

TABLE OF CONTENTS

TABLE OF FIGURES

NOMEMCLATURE

Variable	Description	Unit
d	Distance	m
x	The x-coordinates of the particles	m
y	The y-coordinates of the particles	m
h	Smoothing length	m
T	Temperature	K
ρ	Density	kg/m^3
p	Pressure	Pa

Table 1 Nomenclature

1. INTRODUCTION

Numerical simulation plays a significant role to solve complex problems in fields of engineering and science. (Moubin & Gui-rong, 2003) Numerical simulation enables engineers to simulate physical problems without experiments. (Moubin & Gui-rong, 2003). Computational Fluid Dynamics (CFD) is one of numerical simulation methods to support theories and experiments.

In Computational Fluid Dynamics, there are mainly two approaches to simulate fluid flow. One is Lagrangian description, and the other one is Eulerian description. For the last decades, both were mostly used for grid-based methods which have advantages to solve differential or partial differential equations.(Liu & Liu, 2010). However, although their usefulness, grid-based methods have limitations on computing problems with large deformation, free surface, or cases which have many domains. (Liu & Liu, 2010)

On the other hand, Smoothed Particle Hydrodynamics (SPH) methods that express fluid flows by Lagrangian description are particle-based methods. These methods have advantages on simulating problems with large deformation, free surface, moving material interfaces, or large displacement cases. (Moubin & Gui-rong, 2003).

Even though the SPH solvers have computational advantages, generally numerical simulations require a lot of computational resources for fast and accurate simulations. In recent years, Machine Learning (ML) has been applied as a complementary method to overcome the limitation in Computational Fluid Dynamics. For example, (Debiagi et al., 2020) showed a ML model to predict coal combustion CFD simulations, (Zhou & Ooka, 2020) attempted to predict indoor airflow with machine learning, and (Ringstad et al., 2021) trained a ML model to predict performances of CO_2 ejectors.

Unfortunately, the accuracy of ML models is greatly influenced by the data used for training. If the ML models were trained based on inaccurate data, the ML models predict inaccurate results. Likewise, if the ML models were trained with inaccurate data generated by grid-based methods, the ML models predict inaccurate behavior of fluid flow.

This paper focus on creating Machine Learning models about reverse roll coating. Generally, in reverse roll coating, the rpm of rolls is the one of key factors. As mentioned earlier, SPH solvers have advantages to simulate moving material interfaces. Therefore, high quality of data can be obtained from the SPH solvers.

Moreover, grid-based methods have difficulties to generate meshes between the two rolls due to the fact that large deformation of fluid is occurred. Otherwise, particle-based methods are free from generating meshes.

Based on the high-quality data, which is generated by the SPH solvers, machine learning techniques are able to indicate the optimized values of numerical simulations. For example, in reverse roll coating, the rpm of rolls, the gap between the rolls, and the fluid properties. Therefore, the approach, in this study, aims to develop ML models which are trained based on data generated by Smoothed Particle Hydrodynamics solver (SPH solver) in order to overcome the limitation of FDM method which is one of time-consuming method and grid-based methods.

1.1 EXPERIMENT OVERVIEW

As can be seen in Figure 1, the experiment of this paper is divided into 2 sections. The simulation process is presented to produce CFD simulation results from the SPH solver. All codes were written in Python 3 and Philip Mocz's code was referenced. (Mocz, 2020) In the second part, the method and theory how Machine Learning train with CFD data and predict the results are shown. Park Hae-sun's book was referenced for the mathematical theory of linear regression and machine learning code. (Sun, 2020)

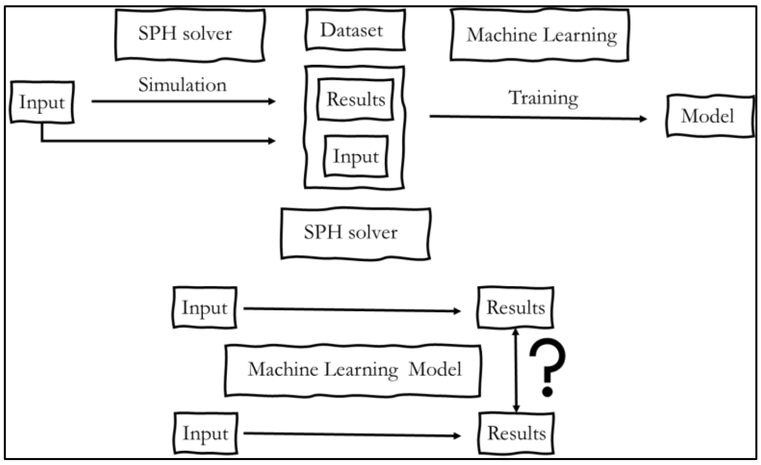


Figure 1 Experiment overview. Machine Learning model is created based on the CFD data made by SPH solver.

1.2. COMPUTER SPECIFICATION

The computer specification was as follow: PyCharm was used to code the CFD model on Windows 10. And Jupyter Notebook was used to create machine learning model.

Processor: AMD Ryzen 7 4800H with Radeon Graphics 2.90 GHz

RAM: 16.0 GB

Display Adapter: NVIDIA GTX 1660ti

Operating System: Windows 10 64bit

2. SIMULATION

2.1 GEOMETRY

As can be seen in Figure 2, The simulation consists of two rolls co-rotating in anticlockwise direction at arbitrary speeds in a two-dimensional cartesian coordination. The shape of rolls remained unchanged. The center points of the two rolls were fixed. And there was a narrow gap between the two rolls. Fluid particles are generated in the "Fluid Generating Area" as shown in Figure 2

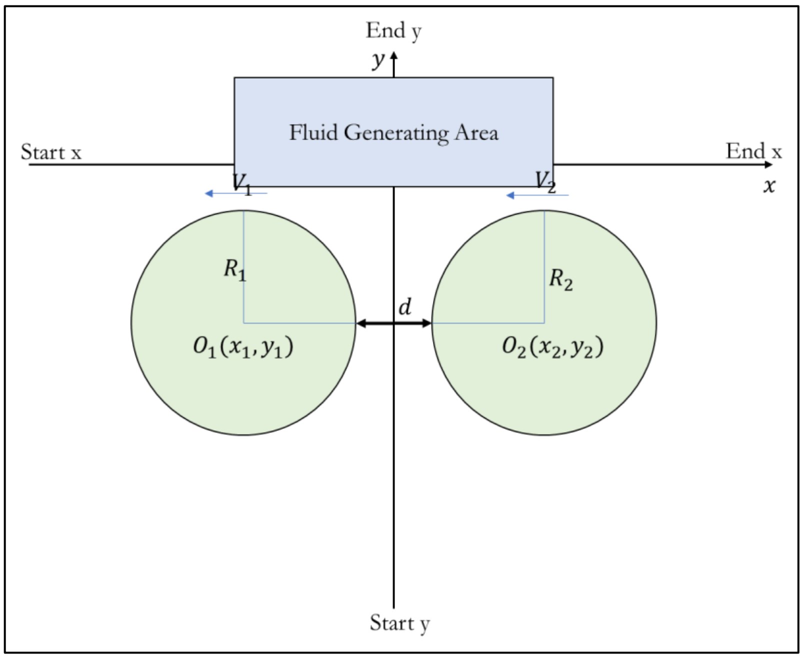


Figure 2 Geometric sketch of the experimental System. (*R*: radius, *O*: the center point of the roller)

Both rolls have same radii.

$$R_1 = R_2 = 0.048\ m$$

The center points of the two rolls:

$$O_1 = (-0.05\ m, -0.06\ m)$$

$$O_2 = (0.05\ m, -0.06\ m)$$

The absolute rotating speed was arbitrary fixed within a predetermined range.

$$0.48\ m/s \leq v_1 \leq 1.92\ m/s$$

$$0.48\,m/s \le v_2 \le 1.92\,m/s$$

The gap between two rolls:

$$d = 0.004\,m$$

The area where particles were generated:

$$(-0.02\,m \le x < 0.02\,m) \text{ and } (-0.02\,m \le y < 0.02\,m)$$

2.2 FLUID PROPERTIES

The fluid is incompressible, and its each particle has same mass and kinematic viscosity in one simulation. Conversely, each simulation has a different value of mass and kinematic viscosity of particles, and the values of both properties are fixed arbitrarily within a predetermined range.

Mass of one particle:

$$0.0001kg \le m \le 0.0002kg.$$

Kinematic viscosity (ν) of one particle:

$$1 \times 10^{-7} m^2/s \le \nu \le 3 \times 10^{-7} m^2/s.$$

The rest density of liquid in 2-dimensional coordinate:

$$\rho_0 = 1000\,kg/m^3$$

2.3 INITIAL CONDITION

One hundred particles were generated at the "Fluid Generating Area" in the Figure 2 before starting the simulation. Initial velocities of the particles are zero.

Total number of input particles:

$$N_{input} = 100.$$

Velocities of each particle:

$$\frac{dx}{dt} = 0\ m/s, \qquad \frac{dy}{dx} = 0\ m/s.$$

Total simulation time and time step:

$$t_{total} = 0.2\ sec$$

$$t_{one\ time\ step} = 0.001$$

2.4 SMOOTHED PARTICLES HYDRODYNAMICS SOLVER

In general, Smoothed Particle Hydrodynamics is a meshfree and Lagrangian method to simulate fluid flows and solid mechanics. It was developed by Lucy, Gingold and Monaghan for astrophysical problems which deal with cope nonaxisymmetric phenomena (Lucy, 1977). Nowadays it become popular in order to overcome difficulties that conventional grid-based methods have (Moubin & Gui-rong, 2003). For example, grid-based methods have problem to deal with free surface, moving interface, and deformable boundary (Liu & Liu, 2010).

2.4.1 Class Structure

This SPH solver was developed with reference to the SPH solver written by (Mocz, 2020). In addition, it is written in Python 3 in order to use NumPy arrays.

This SPH solver was designed with Object-Oriented Programming. As shown in Figure 3, there are four kinds of input objects in the solver. “Mesh” and “Roller” objects contain geometric information, “Time” object does information about time, “Liquid Property” object does information about fluid properties.

The simulation was computed in the object which was named “Particle” after copying the values from the four objects.

All results were saved in NumPy array format during the simulation at every time-step.

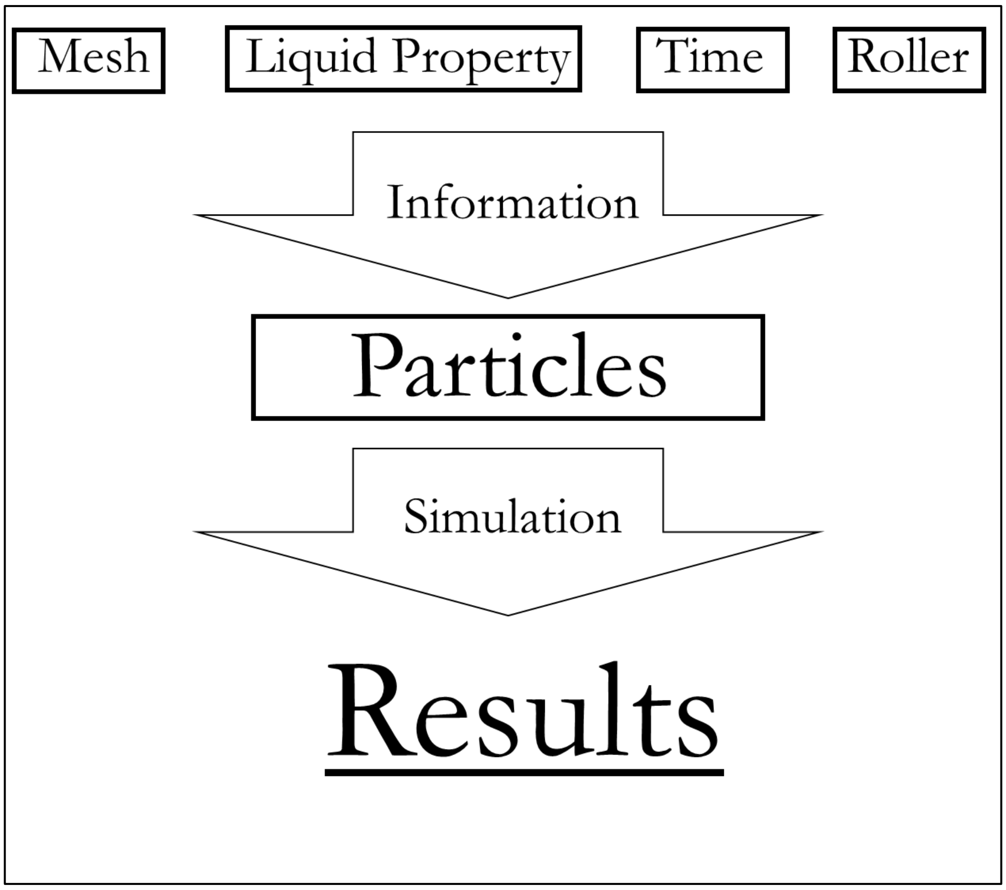


Figure 3 The class overview of the SPH solver. “Particles” class call by values from “Mesh”, “Liquid Property”, “Time”, and “Roller” objects.

2.4.2 Kernel function

All particles were affected by their neighboring particles based on kernel function, $W(r,h)$. The kernel function indicates the degree of the influence of a particle on the other particle according to the distance between particles. Details about kernel function can be found in (Liu & Liu, 2010).

The concept of SPH method starts from the following equation:

$$f(x) = \int_{\Omega} f(x')\delta(x - x')dx', \quad (1)$$

where f is a function of the three-dimensional position vector, x' , $\delta(x - x')$ is Dirac delta function, and Ω is the volume of integral that contains x.

There are various Kernel functions W which is used instead of Dirac delta function.

$$f(x) \approx \int f(x')W(x - x', h)\, dx', \quad (2)$$

where h is smoothing length in Figure 4.

In this simulation, the smoothing length h:

$$h = 0.01\, m.$$

The Kernel function,W, is usually desired to be an even function. It satisfy several conditions in this simulation as follows: (Moubin & Gui-rong, 2003) (Koschier et al., 2020)

The first condition is that the function, f , is normalized:

$$\int_{\Omega} f(x')\delta(x - x')dx' = 1. \quad (3)$$

The second condition is related to Dirac delta function:

$$\lim_{h \to 0} W(x - x', h) \geq \delta(\mathrm{x} - \mathrm{x}'), \quad (4)$$

where h is smoothing length.

The third condition is the compact support condition which necessary to obtain high speed simulations. As computers do not require to compute particles that rarely affect other particles:

$$W(x - x', h) = 0 \text{ for } \parallel x - x' \parallel \geq kh, \quad (5)$$

where k is constant.

The fourth condition is that Kernel function is always greater than or equal to zero as shown in Figure 5:

$$W(x - x', h) \geq 0 \qquad (6)$$

The fifth condition is symmetry condition:

$$W(x - x', h) = W(-(x - x'), h). \qquad (7)$$

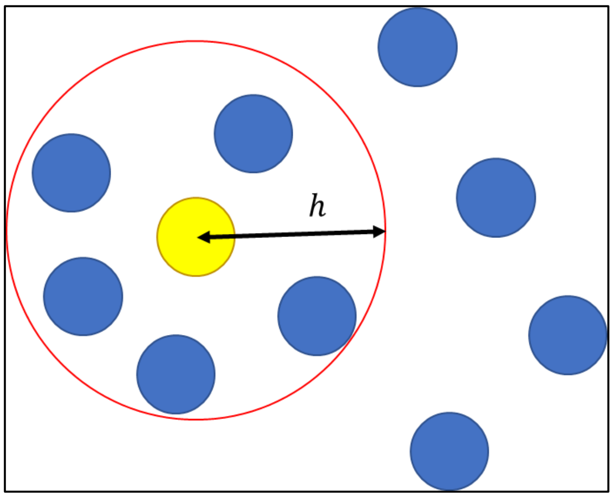


Figure 4 Smoothing length h. Smoothing length determine the distance the particle is affected by the neighboring particles.

There are various kinds of Kernel functions depending on the simulation. The kernel in this simulation is as follow:

The Kernel function in 2-dimensional space (Koschier et al., 2020):

$$W(r, h) = \alpha_d \begin{cases} 6(q^3 - q^2) + 1 & 0 \leq q < \frac{1}{2} \\ 2(1 - q)^3 & \frac{1}{2} < q \leq 1 \\ 0 & 1 < q \end{cases}, \qquad (8)$$

where $\alpha_d = \frac{40}{7\pi h^2}$ in 2-dimensional space, $q = \frac{1}{h}||r||$, and r is the distance between two particles.

And the derivative form of the kernel function (Koschier et al., 2020):

$$\nabla W(r,h) = \alpha_d \begin{cases} 18q^2 - 12q & 0 \le q < \frac{1}{2} \\ -6(1-q)^2 & \frac{1}{2} < q \le 1 \\ 0 & 1 < q \end{cases}, \qquad (9)$$

where $\alpha_d = \frac{40}{7\pi h^2}$ in 2-dimensional space, $q = \frac{1}{h}||r||$, and r is the distance between two particles.

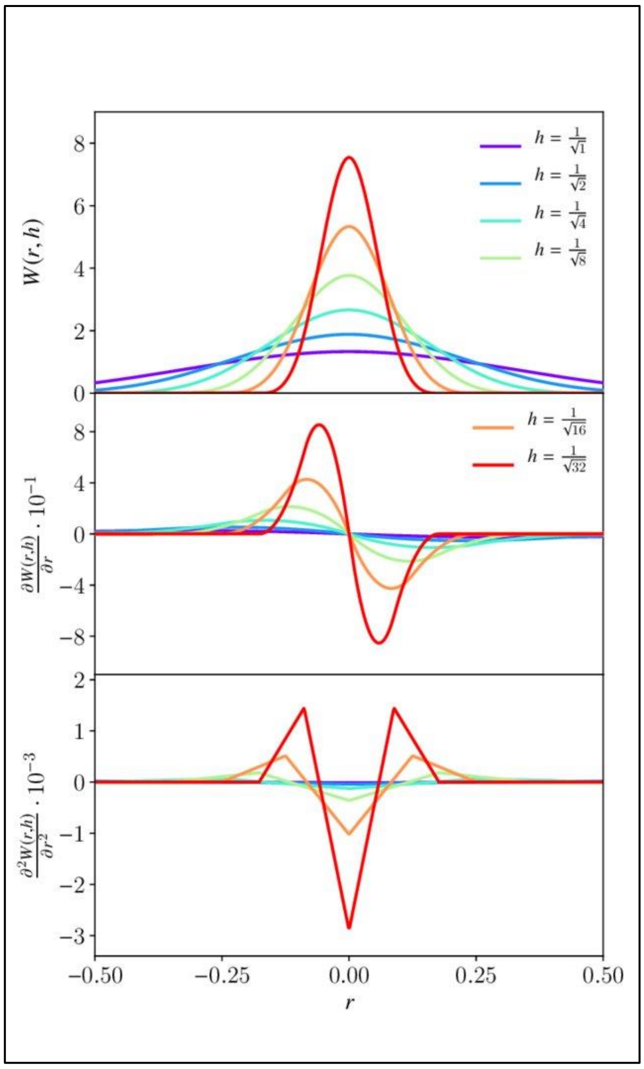


Figure 5 Kernel function. r is distance between two particles. h is smoothing length.(Koschier et al., 2020)

2.4.3 Acceleration

In this simulation, the total acceleration term was sum of three components: accelerations of pressure, viscosity, and gravity.

$$a_{i\,total} = a_{i\,pressure} + a_{i\,viscosity} + a_{i\,gravity} \tag{10}$$

The governing equation of the SPH solver is the Lagrangian form of the Navier-Stokes equation. Details about discretization of SPH solver are given in (Koschier et al., 2020), (Teschner).

$$\frac{dv_i}{dt} = -\frac{1}{\rho_i}\nabla p_i + \nu\nabla^2 v_i + \frac{F_{gravity}}{m_i} \tag{11}$$

2.4.3.1 Pressure

Pressure forces work on the surface of the fluid (Teschner). The acceleration of pressure can be described by the Lagrangian form of Navier-Stokes equation:

$$-\frac{1}{\rho_i}\nabla p_i \tag{12}$$

In the simulation, the fluid is incompressible. Therefore, the Equation 11 must be 0. However, it is assumed that incompressible fluid is actually compressible only in order to compute the momentum of pressure which is designated as artificial compressibility. (Moubin & Gui-rong, 2003)

Before computing pressure values, it is required to compute the density of all particles due to the fact that pressure occurs from density differences. In this simulation, the density was dependent on the position of neighboring particles and mass of each particle.

Density of particles:

$$\rho_i = \sum_j m_j W_{ij}, \tag{13}$$

where ρ_i is density, $m_j$is mass, $W_{ij}$is Kernel function.

Pressure term was computed based on density value (Teschner) (Koschier et al., 2020):

$$p = k\left(\frac{\rho_i}{\rho_0} - 1\right), \tag{14}$$

where $k = 1$ is constant of state equation, ρ_0 is rest density.

Pressure differences between particles cause acceleration. And the acceleration goes to the direction where the change of volume over density will be reduced.

Discretized form:

$$a_{i\,pressure} = -\sum_j m_j \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2}\right) \nabla W_{ij}. \tag{15}$$

2.4.3.2 Viscosity

Friction forces occurs on the surface of the fluid. It comes from the velocity differences between particles.

Friction term of acceleration:

$$\nu \nabla^2 v_i, \tag{16}$$

where ν is Kinematic viscosity.

Discretized form:

$$a_{i\,viscosity} = 2\nu \sum_j \frac{m_j}{\rho_j} v_{ij} \frac{x_{ij} \cdot \nabla W_{ij}}{x_{ij} \cdot x_{ij} + 0.01h^2}, \tag{17}$$

where, $x_{ij} = x_i - x_j$.

3.4.3.3 Gravity

Gravity force exists only in y-direction in the simulation.

$$g = 9.8\, m/s^2$$

Gravity term of acceleration:

$$\frac{F_{gravity}}{m_i}. \tag{18}$$

Discretized form:

$$a_{i\,gravity} = g. \tag{19}$$

2.4.4 Neighbors-searching algorithm

Each particle is affected by its neighboring particles including itself in the SPH solver. There are two checkpoints in order to find out how neighboring particles influence each particle: the one is the distance between particles, and the other is the number of neighboring particles.

Generally, in SPH solver, an algorithm which is called "Neighbors-searching" plays a significant role in order to limit the number of particles to be computed. The "Neighbors-searching" algorithm search neighboring particles of each particle based on coordinates of particles. The reason why the algorithm search neighboring particles is that only neighboring particles are able to affect the particle that the computer is computing. Therefore, the computer does not have to consider all particles owing to the "Neighbors-searching" algorithm. Consequently, the "Neighbors-searching" algorithm helps to reduce simulation time. (Onderik & Durikovic, 2008)

There was not a specific method to express "Neighbors-searching" algorithm in this SPH solver. However, the "Neighbors-searching" algorithm of this SPH solver exists conceptually in the "Particles" object. In the SPH solver, the "Neighbors-searching" algorithm searches every information of all particles at every time-step in "Particles" object, which is why only 100 particles were used in this simulation. So, it does not take long time to simulate particles as can be seen in Chapter 4.

2.5 ROLLER ALGORITHM

Particles are always influenced by the two rolls both directly and indirectly. When a particle collides with one of the two rolls, then the velocities of the particle become changed. As the changed velocities of the particle affects the kernel function of the neighboring particles, the velocities of the non-collision neighboring particles are also influenced by the kernel function. In this chapter, the algorithm is presented how particles change their velocities when they collide with the two rolls.

$$(x - O_x)^2 + \left(y - O_y\right)^2 = r^2, \qquad (20)$$

where, O_x is the x-coordinate at the center of the roll, O_y is the y-coordinate at the center of the roll, and r is the radius of the roll.

There are some possibilities for a single particle to penetrate rolls in following cases due to the fact that rolls do not exist "physically" in the computational simulation.

Case 1: In the SPH solver, Lagrangian approaches compute velocities (Teschner).

If the current position of a particle is:

$$x_i(t) = (x_i, y_i), \qquad (21)$$

and the velocities of the particle are:

$$v_i(t) = (u_i, v_i), \qquad (22)$$

then the position of the particle at the next time step is:

$$x_i(t+1) = (x_i + \Delta t \cdot u_i, \quad y_i + \Delta t \cdot v_i\,). \qquad (23)$$

When the particle has high velocity, the particle is able to penetrate the roll in Figure 6.

Case 2: Following the Eq. 23, the particle is able to penetrate the roll when the simulation has long time derivative.

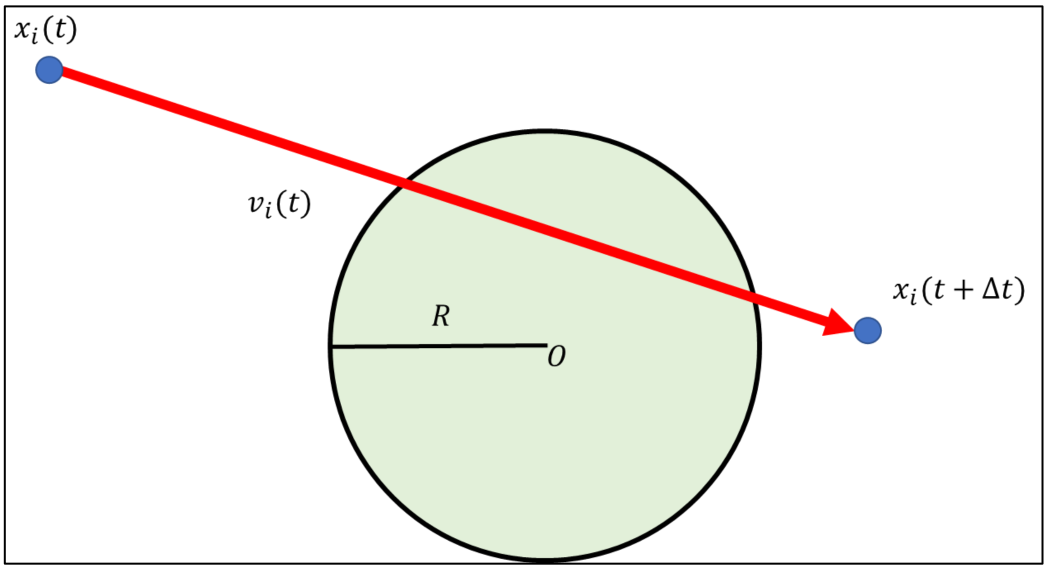


Figure 6 There are several possibilities that a particle can penetrate a roll. One is in case that velocities are very high. And the other one is when the simulation has long time derivative.

As a result, short time derivative is necessary to avoid for those particles penetrating the rolls. If time is short enough, the penetrating of particle by high velocities can be also prevented.

It is significantly important to prevent the penetrating of particles due to the fact that it is difficult to be occurred in reality. Moreover, the velocities of particles must be changed when they collide with rolls.

The SPH solver was developed in Object-Orient Programming. Every simulation is computed in the "Particles" object which was made of the "Particles" class. There is a method which was called "addRollerEffect" in order to add the effect of particle colliding with rolls in the "Particles" object. This method works based on the coordinates of particles. For example, when a particle is being located at $x_i(t)$ at the n time-step like Figure 7, the "addRollerEffect" method is not activated. However, if the particle is going to move to the inside of any roll at the $t+1$ time-step, the "addRollerEffect" becomes activated.

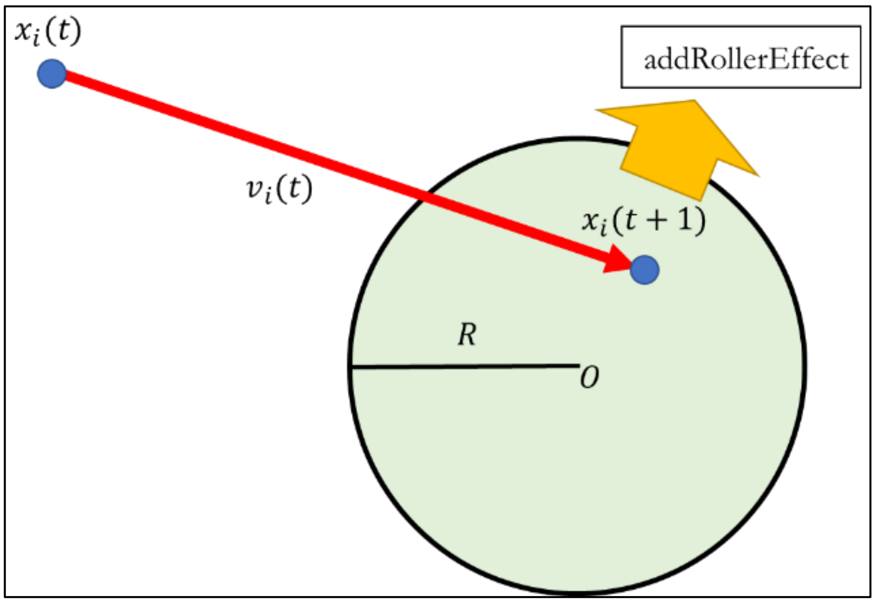


Figure 7 The "addRollerEffect" method. If a particle is going to be located at the inside of one of rolls at the next time-step, then the "addRollerEffect" method is executed.

As shown in Figure 8, when the "addRollerEffect" method is executed, particles change the position and velocity. It is assumed that it is non-slip condition on the surface of the roller. Therefore, the velocity of particle is going to follow the angular velocity of the roll at next time-step.

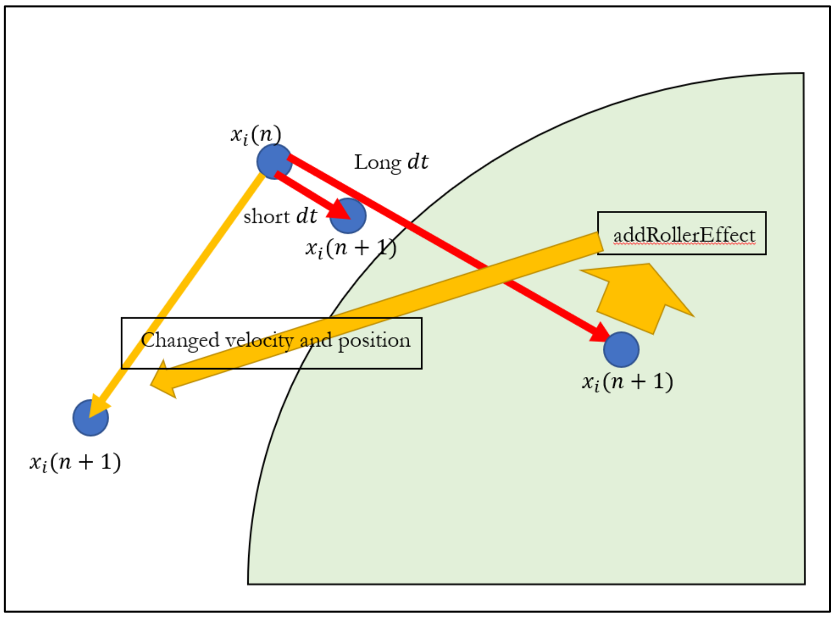


Figure 8 How particles move when the "addRollerEffect" method is executed.

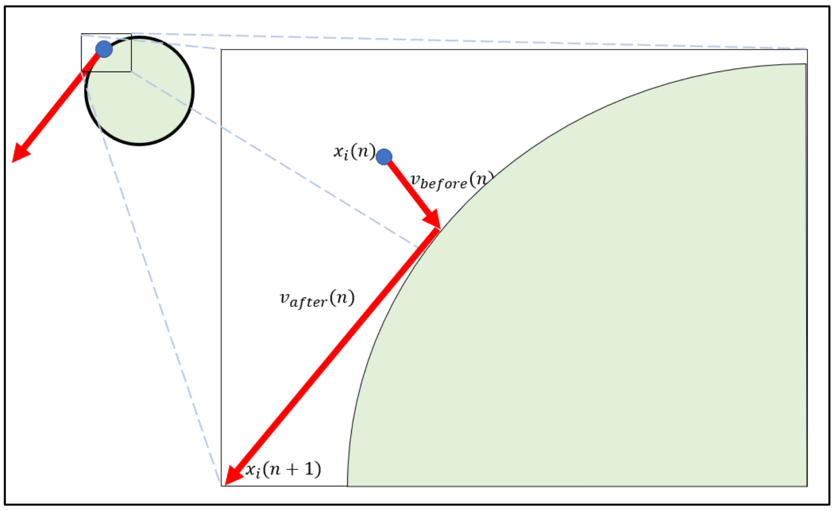


Figure 9 Particles must change their movement on the surface of the roller for more accurate results.

As can be seen in Figure 9, particles change their velocities on the surface of rolls in reality. However, in this simulation particles do not change their velocities and positions on the surface of rolls. Consequently, it is one of reason to occur errors in the simulation.

This simulation work is not aimed to build a perfect CFD model. Otherwise, there is one thing to reduce errors. It is to shorten the time derivatives so that the particles move very short distance at one time-step as can be shown in Figure 10. Particles change their position from their current position when they collide with the rolls at next time-step. If the time derivatives get longer and the length of the particles travelling within a time step becomes longer, the particle is more likely to change position at a farther distance. Consequently, short time derivate is necessary because of this reason.

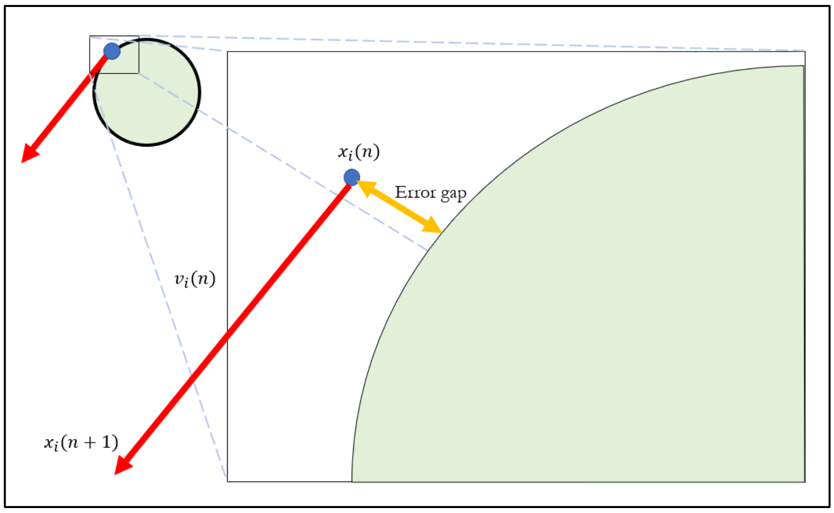


Figure 10 Error gap which is occurred after computing the "addRollerEffect" method. Error gap is distance between the position at the past time step and the surface of rolls

3. MACHINE LEARNING

Machine Learning approaches are broadly divided into three categories, which are supervised learning, unsupervised learning, and reinforcement learning, depending on whether under supervision or not. (Géron, 11 oct. 2019)

Supervised learning aims to find patterns or hypothesis based on labeled data. There are two typical supervised learning algorithms. One is Classification algorithm which is used in order to distinguish data. The other one is Regression algorithm. Regression predicts the value of target based on a feature which is called "predictor variable". In supervised learning, there are Support-vector machines, Logistic regression, Decision trees, K-nearest neighbor algorithm, and Neural networks (Géron, 11 oct. 2019).

Unsupervised learning is one of machine learning techniques which does not require any labeled data. (Hinton, 2012) Consequently, unsupervised learning must find out patterns by itself. For example, Clustering algorithm groups data which have similar features. In unsupervised learning, there are Visualization and Dimensionally reduction, and Association rule learning. (Géron, 11 oct. 2019)

Reinforcement learning is a very different type of algorithm from supervised learning and unsupervised learning. Both supervised and unsupervised learning are affected by the quality of the data, on the other hand in reinforced learning, the agent is trained with rewards through actions and environment as can be seen in Figure 11. (Micheal, 2020) DeepMind's AlphaGo is a good example of reinforcement learning. AlphaGo won against Lee Sedol who was one of the best Go players in the Google DeepMind Challenge Match.

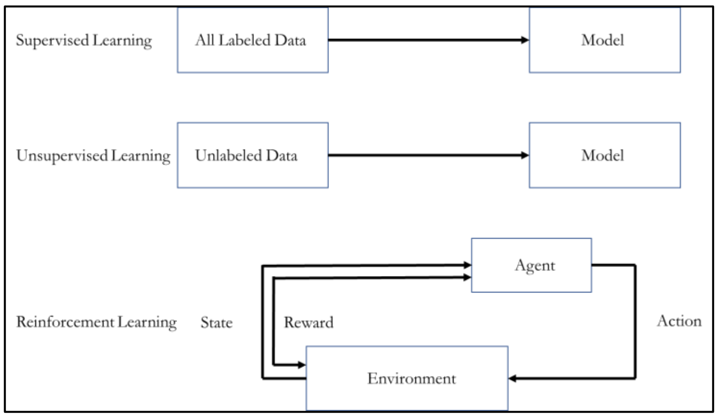


Figure 11 Categories of machine learning. Machine learning approaches are categorized according to existence of supervision.

3.1 CHOICE OF ALGORITHM

In supervised learning, it is aimed to find the most proper weights and bias based on data. This process consists of two parts. One is forward propagation and the other one is backward propagation. The forward propagation produces results with variables and machine learning model which is composed of weights and bias. Whereas the "Backward propagation" modifies the machine learning model in order to reduce discrepancy between the actual data and predicted data which results from forward propagation.

In this study, the machine learning model was trained by multivariable linear regression algorithm which is one of the supervised learning algorithms. The machine learning model was referred to the book(Sun, 2020)which presents details about mathematical algorithm of multivariable linear regression and how to program machine learning model.

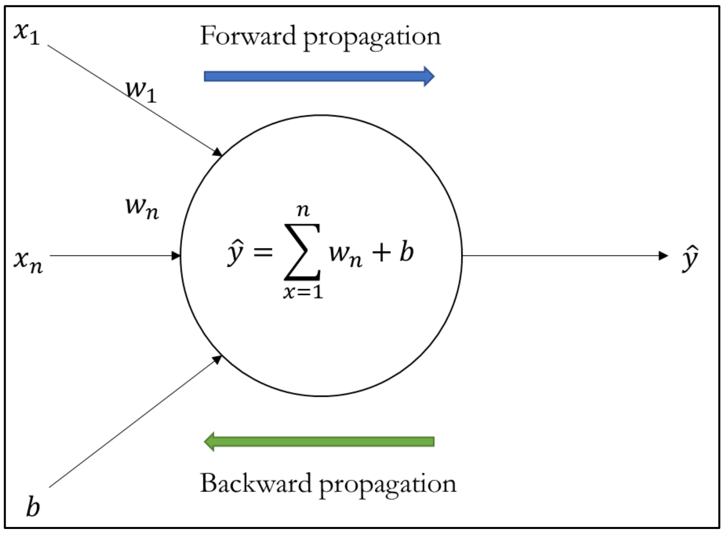


Figure 12 A neuron in multivariable linear regression. Predicted data $\hat{y}$ is computed through forward propagation. (Weights: ω, bias: b)

Multivariable linear regression consists of weights and bias within a single layer, as shown in Figure 12. The number of weights is same as the number of features. The number of biases is always one in multivariable linear regression.

For the forward propagation, it is only necessary to multiply the weights by the parameters and add the bias:

$$\hat{y} = \sum_{i=1}^{n} \omega_i x_i + b, \qquad (24)$$

where $\hat{y}$ is predicted value, x_n is feature, ω_n is weights, and b is bias.

In order to carry out the backward propagation in multivariable linear regression, definition of cost function should be preceded. Cost function presents errors between actual data y and predicted data,$\hat{y}$, as shown in Figure 13:

$$error = y - \hat{y}, \qquad (25)$$

where y is actual data, $\hat{y}$ is predicted data.

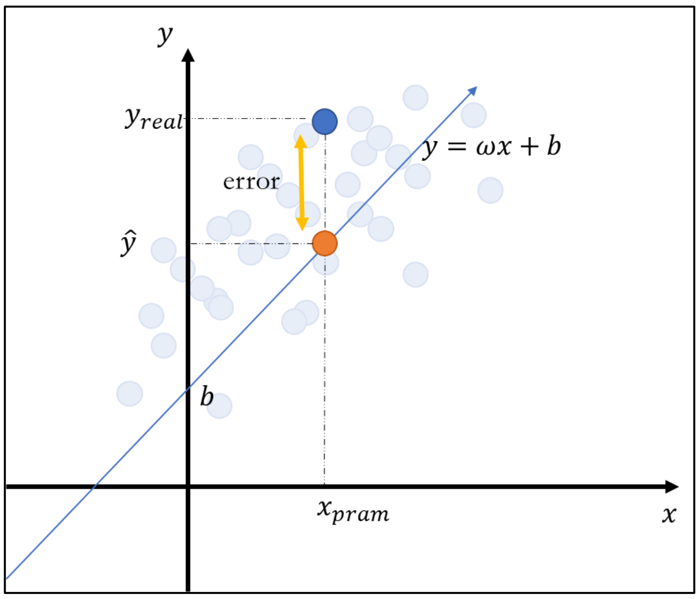


Figure 13 Error in linear regression algorithm. ($\boldsymbol{\omega}$: weights, $\boldsymbol{b}$: bias)

Afterwards, the backward propagation is able to compute the gradient of cost function. The weights and bias are modified by the gradient of cost function. In this study, the cost function is represented as squared error (SE):

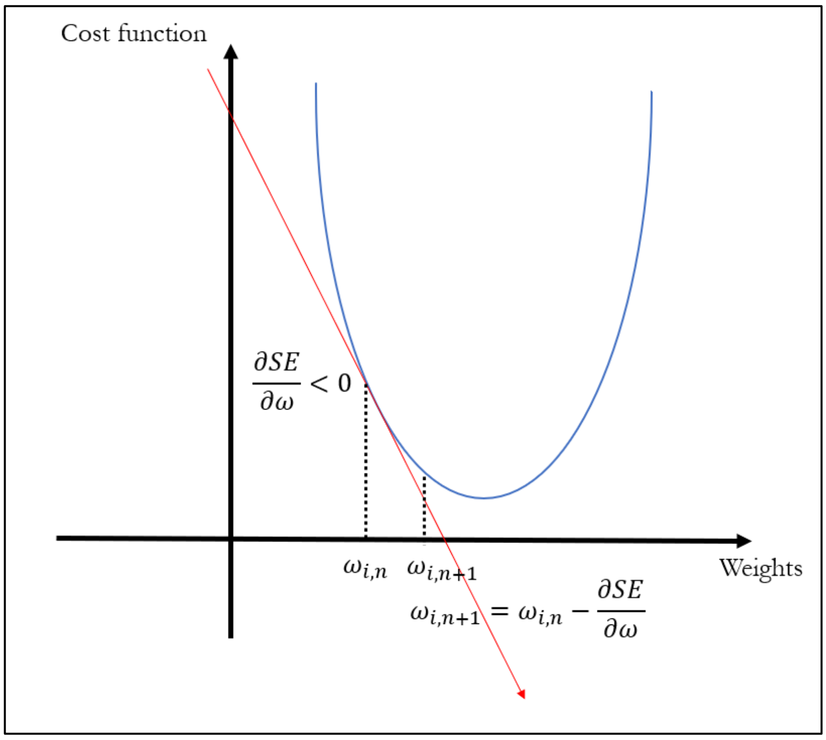
$$SE = (y - \hat{y})^2, \qquad (26)$$

In this study, the Stochastic Gradient Descent (SGD) was used to find out the optimal point at which the partial derivate of the cost function with respect to weights $\frac{\partial SE}{\partial \omega}$ and the partial derivate of cost function with respect to bias $\frac{\partial SE}{\partial b}$ are minimized.

The partial derivate of the cost function with respect to weights:

$$\frac{\partial SE}{\partial \omega_i} = \frac{\partial}{\partial \omega_i}(y - \hat{y})^2$$
$$= 2(y - \hat{y})(-x) = -2(y - \hat{y})x, \tag{27}$$

where ω_i: i-th weights.



Cost function
$\frac{\partial SE}{\partial \omega} < 0$
$\omega_{i,n}$
$\omega_{i,n+1}$
Weights
$\omega_{i,n+1} = \omega_{i,n} - \frac{\partial SE}{\partial \omega}$

Figure 14 Training weights. (ω: weights SE: cost function)

As shown in Figure 14, the weights in the next epoch $\omega_{i,n+1}$ are modified based on the current weights $\omega_{i,n}$ and the partial derivate of the cost function with respect to weights $\frac{\partial SE}{\partial \omega}$:

$$\omega_{i,n+1} = \omega_{i,n} - \frac{\partial SE}{\partial \omega_i} = \omega_{i,n} + 2(y - \hat{y})x. \qquad (28)$$

In general, "Learning rate" is necessary to control the speed at which weights and bias are trained. If the "Learning rate" is too high, machine learning models pass the optimal point as can be seen in Figure 15. Whereas if the "Learning rate" is too low, it takes long time that machine learning models reach the optimal point (Saikat Dutt).

As a result, the weights can be trained with following equation:

$$\omega_{i,n+1} = \omega_{i,n} - \frac{\partial SE}{\partial \omega_i} \times Learning\ rate. \qquad (29)$$

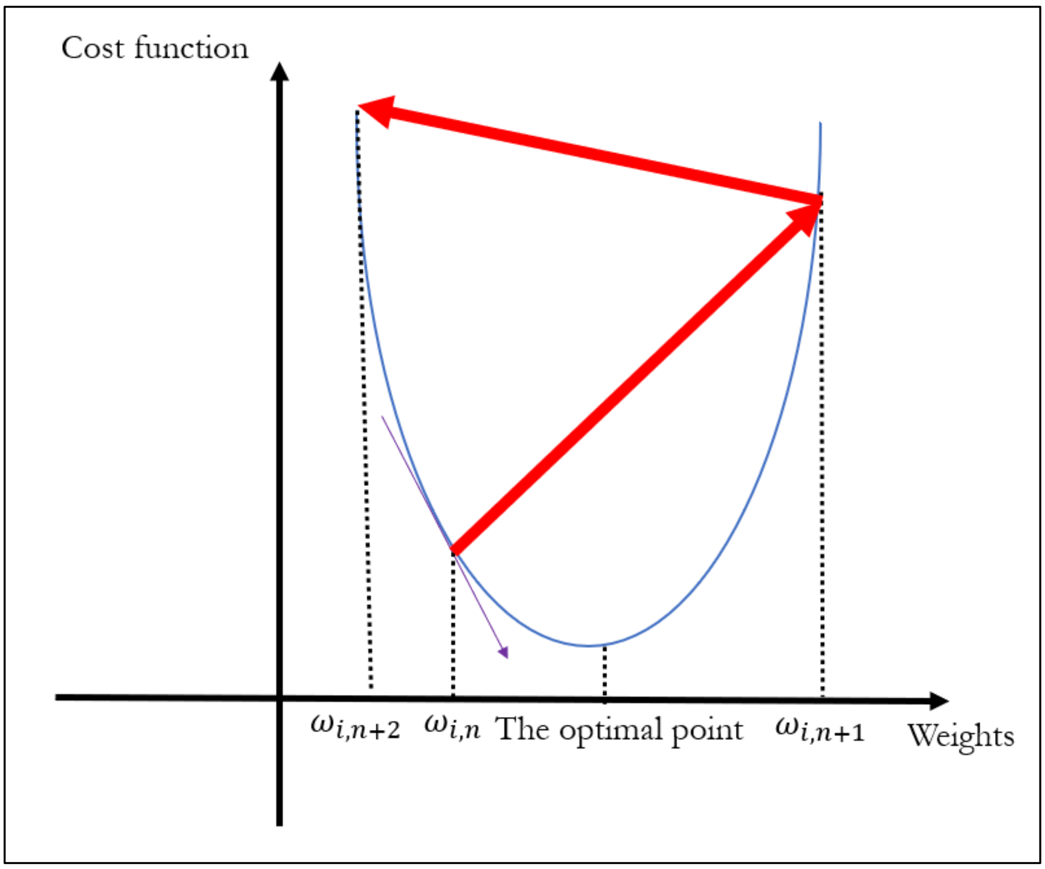


Figure 15 when the "Learning rate" is too high, machine learning models pass the optimal point

It is also necessary to derive the partial derivate of cost function with respect to bias $\frac{\partial SE}{\partial b}$ in order to find out the optimal point of bias with following equation:

$$\frac{\partial SE}{\partial b} = 2\frac{\partial}{\partial b}(y-\hat{y})\left(-\frac{\partial}{\partial b}\hat{y}\right) = -2(y-\hat{y}), \qquad (30)$$

The bias in the next epoch b_{n+1} are modified based on the current bias b_n and the partial derivate of the cost function with respect to bias $\frac{\partial SE}{\partial b}$:

$$b_{n+1} = b_n - \frac{\partial SE}{\partial b} = b + 2(y-\hat{y}), \qquad (31)$$

Learning rate is also needed to control the bias in the next epoch b_{n+1}. As a result, the training for the bias b becomes as follows:

$$b_{n+1} = b_n - \frac{\partial SE}{\partial b} \times Learning\ rate. \qquad (32)$$

3.2 DATA PREPROCESSING

This study is also aimed to build high fast machine learning as well as simulations. Therefore, it is required that every variable of each machine learning model occupies the space of memory card during training. Due to the fact that generally Artificial Neural Networks requires a lot of weights, it is also one of the reasons that multivariable linear regression was chosen as an algorithm in this study.

In this study, the goal of this machine learning model is to predict the position, velocity, acceleration, density, and pressure of the fluid at every time step based on the rotating speed of each roll, the kinematic viscosity of the fluid, the mass of a particle.

During the simulation, the raw data of the positions, velocities, accelerations, density, and pressures of particles were saved as NumPy array in order to achieve fast performance in Python code. (Svensson & Galfi, 2021)

3.2.1 Input dataset

The input dataset is a set of variables that is different for each simulation, which is predetermined before the simulation starts as follow:

The rotating speed of the rolls:

$$0.48\ m/s \leq v_1 \leq 1.92\ m/s$$

$$0.48\ m/s \leq v_2 \leq 1.92\ cm/s.$$

The mass of one particle:

$$0.0001kg \leq m \leq 0.0002kg.$$

The kinematic viscosity:

$$1 \times 10^{-7} m^2/s \leq \nu \leq 3 \times 10^{-7} m^2/s.$$

The input dataset consists of a four-dimensional NumPy array. As shown in Figure 16, each axis is as follows.

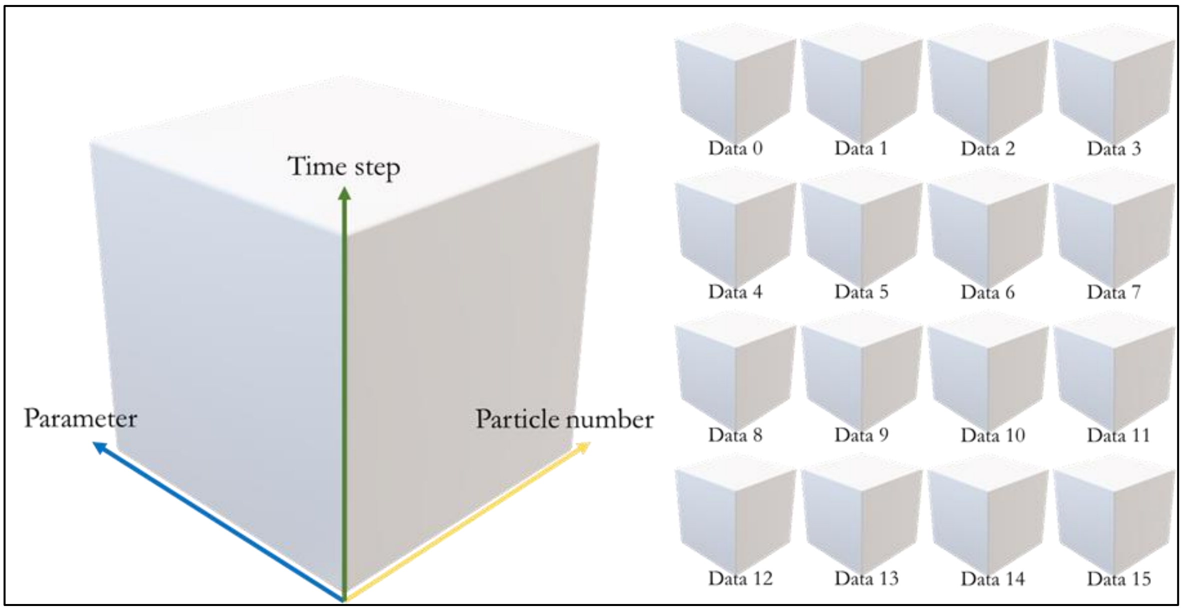


Figure 16 The structure of input data. The first axis is the number of time-steps. The second axis is the number of particles. The third axis is the number of parameters. The fourth axis is the number of simulated data, respectively.

$$Input\ Dataset: The\ number\ of\ Data \times time\ steps \times Particles \times features$$

$$The\ number\ of\ Data = 100$$

$$time\ steps = 200$$

$$Particles = 100$$

$$features = 4$$

Therefore, one input data in the input dataset contains 80000 values, which is why each particle of each time-step have 4 features.

3.2.2 Target datasets

As can be seen in Figure 17, there are 8 kinds of the target datasets. They are categorized into positions, velocities, accelerations, density, and pressures at each time step.

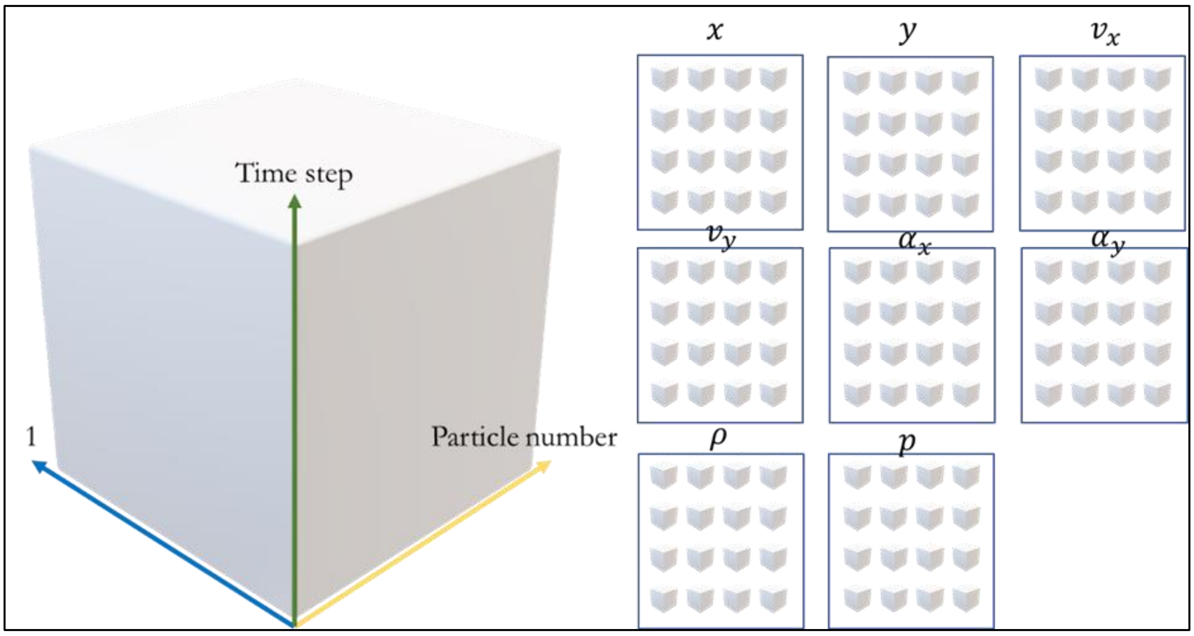


Figure 17 The structure of target datasets. Each target dataset has 4-dimensional dataset. (x: the x-coordinates of the particles, y: the y-coordinates of the particles, v_x: The x-direction velocity of the particles, v_y: The y-direction velocity of the particles, α_x: The x-direction acceleration of the particles, α_y: The y-direction acceleration of the particles, ρ: The density of the particles. p: The pressure of the particles)

$$Target\ Dataset: The\ number\ of\ Data \times time\ steps \times Particles \times 1$$

$$The\ number\ of\ Targets = 8$$

$$The\ number\ of\ Data = 100$$

$$time\ steps = 200$$

$$Particles = 100$$

One target data in one target dataset contains 20000 values, which is why each target dataset aims to one kind of simulation result. Each target dataset contains following simulation results, respectively. (x: the x-coordinates of the particles,y: the y-coordinates of particles, v_x: The x-direction velocity of particles, v_y: The y-direction velocity of particles, α_x: The x-direction acceleration of particles, α_y: The y-direction acceleration of particles, ρ: The density of particles. p: The pressure of the particles)

3.2.3 Features scaling

In data preprocessing, it is important to make all features have values within similar range, which is why machine learning models are prone to be influenced greater values. In this study, all data in the input dataset were scaled by min-max scaler which scales and translates the features into the range between zero and one. (learn, 2022).

The equation of min-max scaling is as follow:

$$X_{scaled} = \frac{X - X_{min}}{X_{max} - X_{min}}, \qquad (33)$$

where, X_{max} is the maximum value in the feature, X_{min} is the minimum value in the feature, X is the value, which is not scaled in the feature, X_{scaled} is the scaled value.

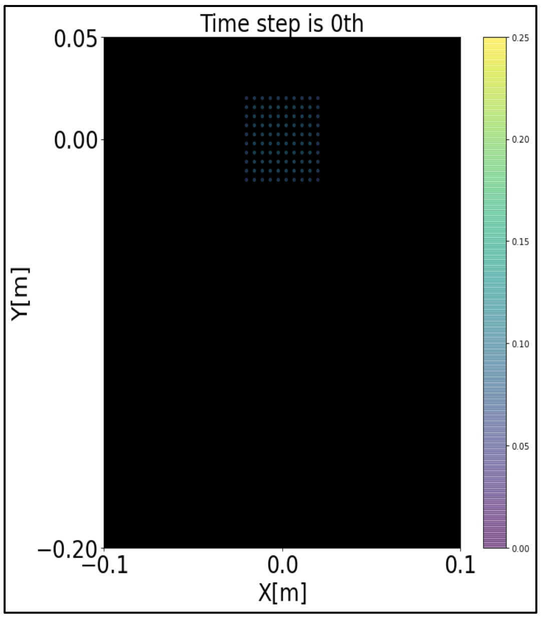
4. RESULT AND DISCUSSION

4.1 SIMULATION

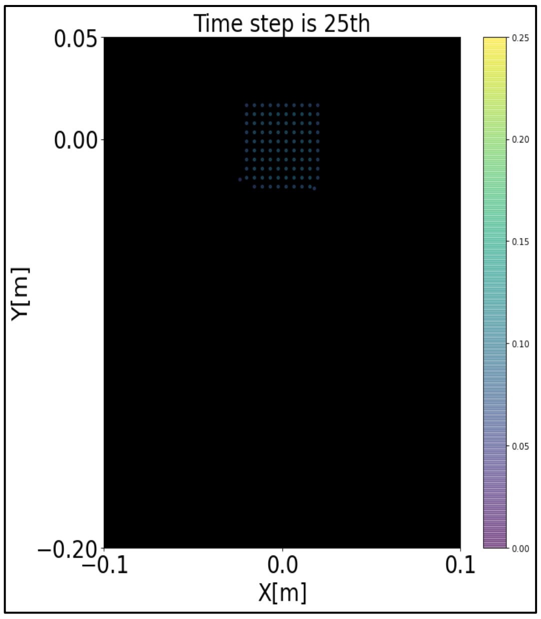
A Smoothed Particles Hydrodynamics solver (SPH solver) was used in order to compute the simulations. SPH solver is one of particle-based solver which follows Lagrangian description. Therefore, in the simulation, the fluid is expressed by particles and simulated by tracking the particles.

The simulation geometry was defined by two rolls of same size (radius of 0.048m) and a narrow gap between them which has width of 0.004m. The two rolls were positioned at the same horizon according to the cartesian coordinate and co-rotating anticlockwise at arbitrary speeds in a predetermined range, respectively.

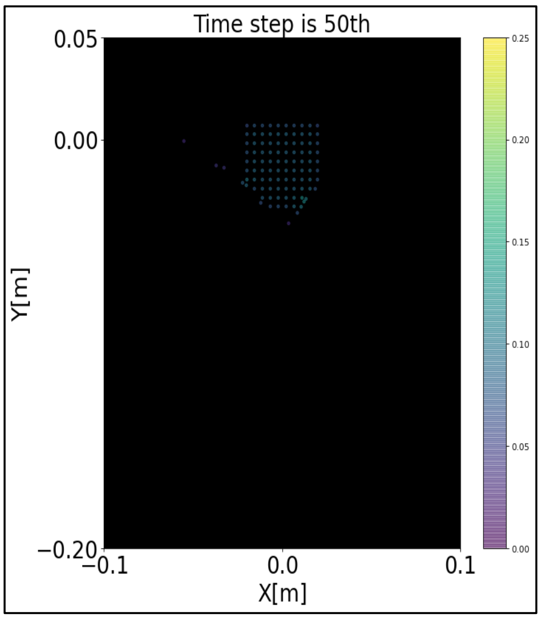
The working fluid was assumed to be Newtonian and incompressible. The mass and kinematic viscosity of the fluid were assigned in the initial condition. The fluid was generated as one hundred particles in the "Particle Generating Area" area and flew into the gap without any external force. In every simulation, each particle was located at the same position at the first time-step. As simulation proceeded, the position of particles was updated by Smoothed Particle Hydrodynamics solver at every time-step. The total time-step was two hundred.



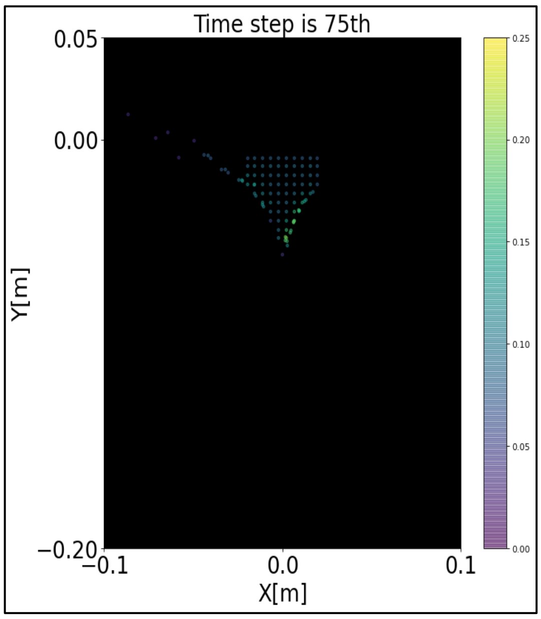
(a) A result of the simulations at 1st-time step



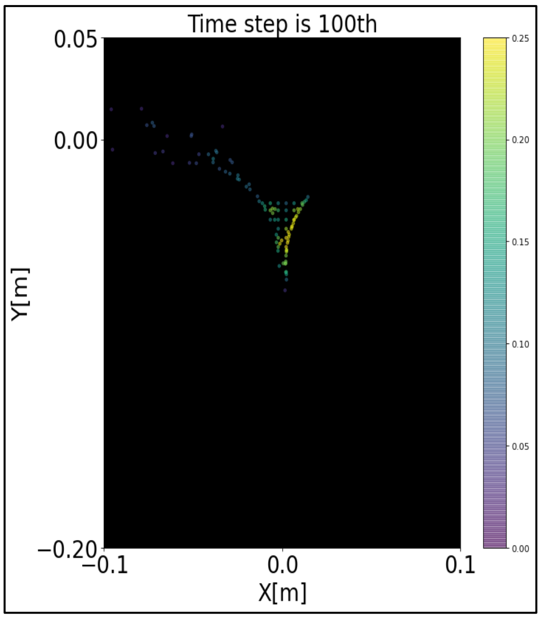
(b) A result of the simulations at 25th-time step



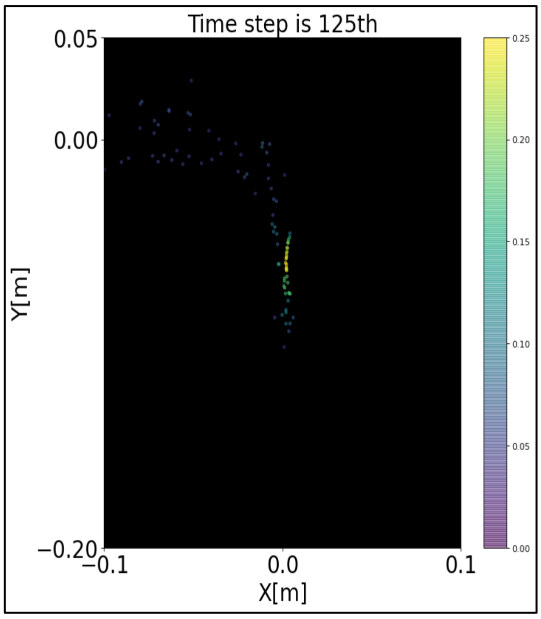
(c) A result of the simulations at 50th-time step



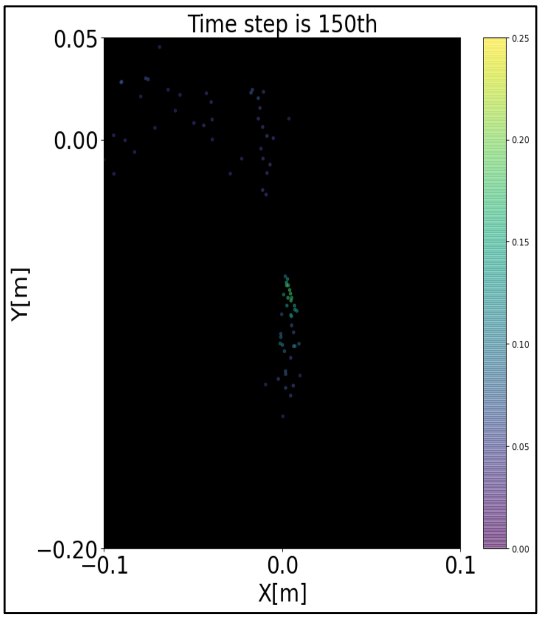
(d) A result of the simulations at 75th-time step



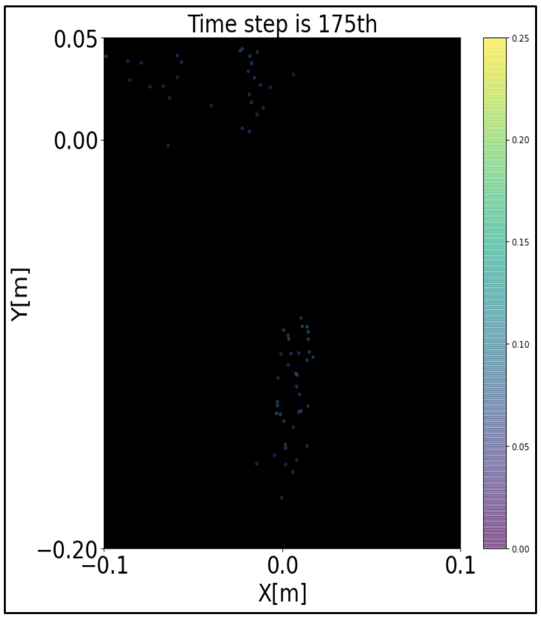
(e) A result of the simulations at 100th-time step



(f) A result of the simulations at 125th-time step



(g) A result of the simulations at 150th-time step



(h) A result of the simulations at 175th-time step

Figure 18 A Graphical result of the simulations

As mentioned before, the Smoothed Particles Hydrodynamics solver assumes that the fluid is a set of particles. which enables the results of particle position be graphically displayed with dots in the two-dimensional simulation window simultaneously. From Figure 18 shows exemplary results of one simulation at 0th, 25th ,50th ,75th ,100th, 125th, 150th, and 175th time-step respectively when the rotating speed of the left roll is 1.40 m/s and the rotating speed of the right roll is 0.55 m/s. Colors of dots in Figures imply the density kg/m^3 of each particle. It can be seen that the particles were attached to the left roll which has low speed.

Following Table 2 shows the time taken to run each of the 100 simulations 6 times. It depends on the computer specification. By the way, it took between 1 and 2 minutes to compute 100 simulations, and it is the reason why there was not a specific method to search neighboring particles in the SPH solver due to the fact that in general "Neighbors-searching" algorithm is used to save computation time in SPH solver.

Simulation	1	2	3	4	5	6
Time(sec)	112.32	113.48	113.48	111.56	111.85	112.76

Table 2 Simulation time

4.1.1 Acceleration

The biggest factor to decide how fluid flows is the two rotating rolls in the simulations. Therefore, in this section, it is presented how much the mean rotating speed affects the mean acceleration of particles.

The mean acceleration of particles per each simulation data was computed as follow:

$$a_m = \frac{\sum_{n_t=1}^{200} \sum_{n_p=1}^{100} a_{n_p,n_t}}{n_T}, \qquad (34)$$

where, n_t is time-step, n_p is particle number,

$$n_T = The\ number\ of\ timesteps \times The\ number\ of\ particles$$
$$= 20000.$$

The mean rotating speed was computed as follow:

$$v_m = \frac{v_L + v_R}{2}, \qquad (35)$$

where, v_m is the mean rotating speed, v_L is the rotating speed of left roll, and v_R is the rotating speed of right roll.

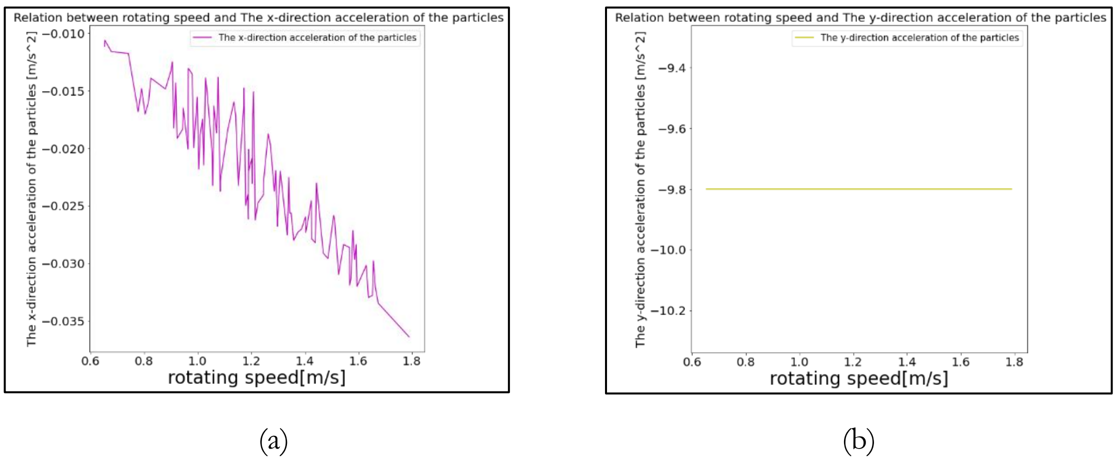


Figure 19 The relation between the acceleration and the rotating speed of the rolls.

Figure 19 shows that the relation between the acceleration and rotating speed of the rolls. As can be seen in Figure 19 (a), as the rotating speed increases, the acceleration in the -x direction generally increases, which is why the two rolls rotated anticlockwise. Otherwise, in Figure 19 (b), the acceleration in the y-direction did not change much due to the fact that the acceleration of gravity was dominant in the y direction, and the acceleration of particles were not changed by the acceleration which was occurred by rolls in this SPH solver. But their acceleration was changed by "addRollerEffect" algorithm in the chapter 2.

4.1.2 Velocity

As mentioned earlier, the rotating speed of the rolls largely affects fluid behavior. As can be seen in Figure 20 (a), the velocity in the $-x$ direction increases as the rotating speed become faster, and the tendency is very similar as the tendency of the mean accelerations of particles in Figure 19 (a). It is due to the fact that the SPH solver follows Lagrangian approaches as follow:

$$v_{i,t+1} = v_{i,t} + a_{i,t} \times dt,$$

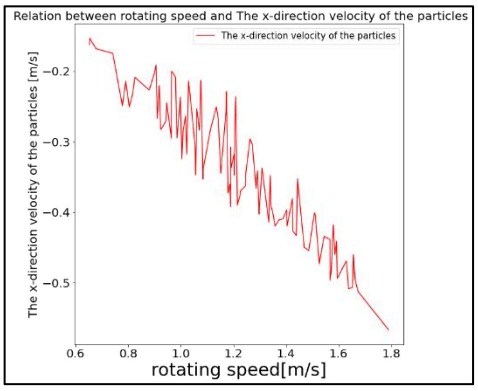
where, $v_{i,t}$ is the velocity of at t time-step, and $a_{i,t}$ is the acceleration.

The mean velocity of the particles per each simulation data is computed as follow:

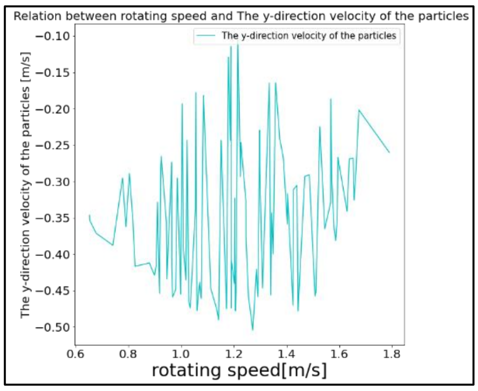
$$v_m = \frac{\sum_{n_t=1}^{200} \sum_{n_p=1}^{100} v_{n_p,n_t}}{n_T}, \qquad (36)$$

where, n_t is time-step, and n_p is particle number.

As can be seen in Figure 20 (b) there is a weak correlation between the speed in the y direction and the rotating speed of the rolls. because the velocity in y direction is directly influenced by the "addRollerEffect" algorithm.



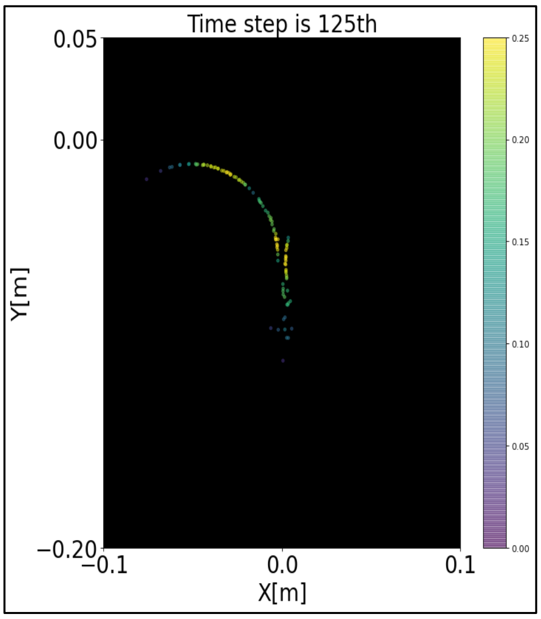
(a)



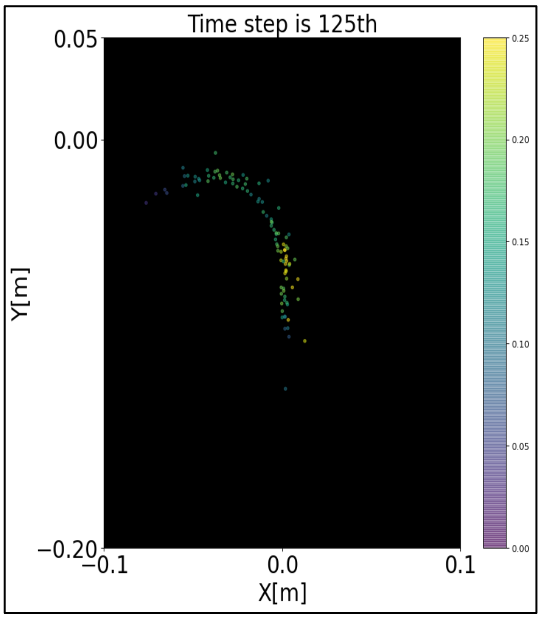
(b)

Figure 20 The relation between the velocity and the rotating speed of the rolls.

4.2 MACHINE LEARNING



(a)



(b)

Figure 21 The result of (a) SPH solver and (b) machine learning using the same input values at 125th time-step.

In this session, the differences between predicted values and the actual values were shown, and the reason why the differences occur significantly at which time-step and particle. Figure 21 shows the differences graphically.

The ML models, consisting of weights and bias, predict positions, velocities, accelerations, pressure, and density of the particles, respectively.

In a simulation, the data consists of one input dataset and eight target datasets. The input parameters are the mass and kinematic viscosity of particles, the rotating speed of each roll, The targets are categorized into positions, velocities, accelerations, density, and pressure of the particles.

The input dataset and the target datasets were configured to 4-dimensional arrays. Each axis of the input dataset and target datasets was assigned as presented in Table 2. The remarkable point is that 0th axis of target datasets is 1, which means each machine leaning model implements only one target value at once. As a results, eight different machine learning models are created.

Axis	3	2	1	0
Input dataset	The number of data	The number of time-steps	The number of particles	The number of parameters
Target dataset	The number of data	The number of time-steps	The number of particles	The number of targets (=1)

Table 3 The structure of input dataset and target dataset. The input dataset and target dataset are 4-dimensional arrays.

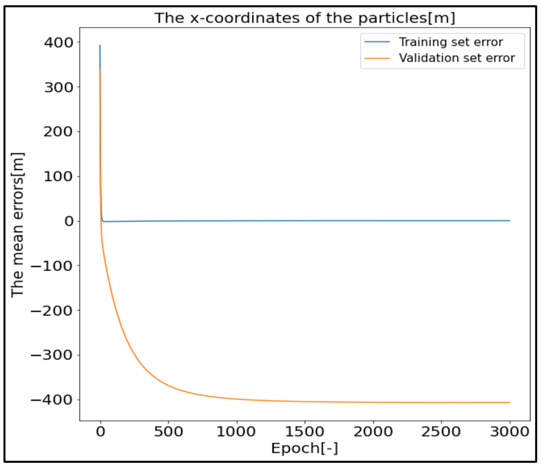
4.2.1 Hyperparameter

User-defined parameters which affect machine learning model are called "Hyperparameter" in machine learning technique. In this study, there are three kinds of hyperparameters in order to carry out machine learning efficiently and precisely. The first one is "Learning rate". Learning rate handle the rate of modifying machine learning model in one backpropagation which is a process to update weights and bias. The second one is "Epoch". Epoch defines the number of times that algorithms have been executed. (Brownlee, 2018) The last one is the cost function of the machine learning model. Cost functions are used to estimate differences between actual data and predicted data. (Mc., 2017) .The cost functions of all machine learning models are presented as squared error in this study. Table 4 shows learning rates and the number of optimal epochs of every machine learning model.

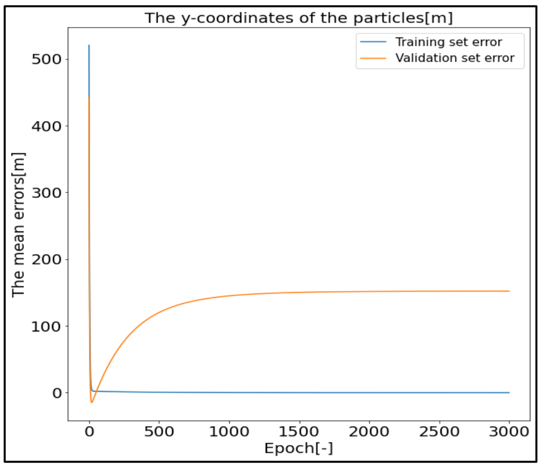
	X	Y	v_x	v_y	α_x	α_y	ρ	P
Learning rate	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
Epochs	3000	3000	3000	3000	3000	3000	3000	3000

Table 4 The optimal learning rate and the number of epochs (x: the x-coordinates of the particles, y: the y-coordinates of the particles, v_x: The x-direction velocity of the particles, v_y: The y-direction velocity of

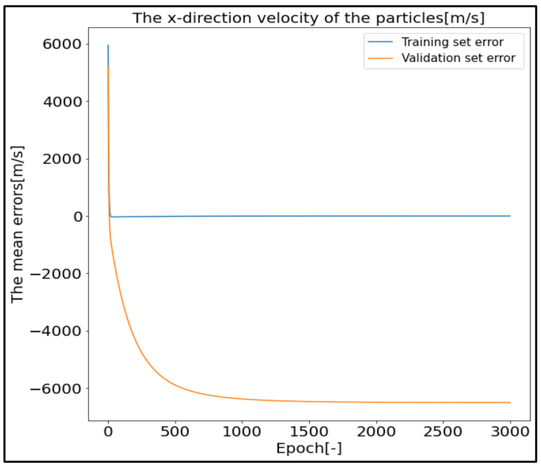
the particles, $\boldsymbol{\alpha_x}$: The x-direction acceleration of the particles, $\boldsymbol{\alpha_y}$: The y-direction acceleration of the particles, $\boldsymbol{\rho}$: The density of the particles. $\boldsymbol{p}$: The pressure of the particles)



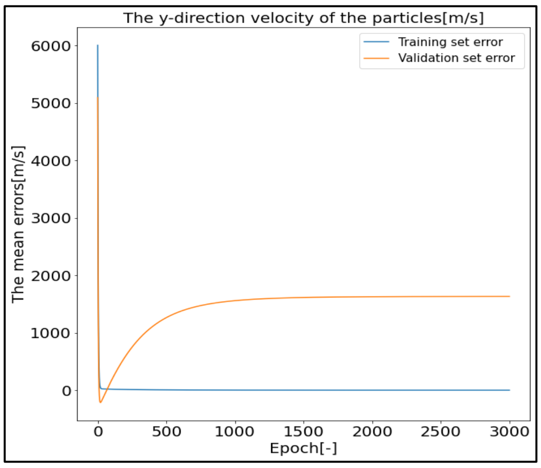
(a)



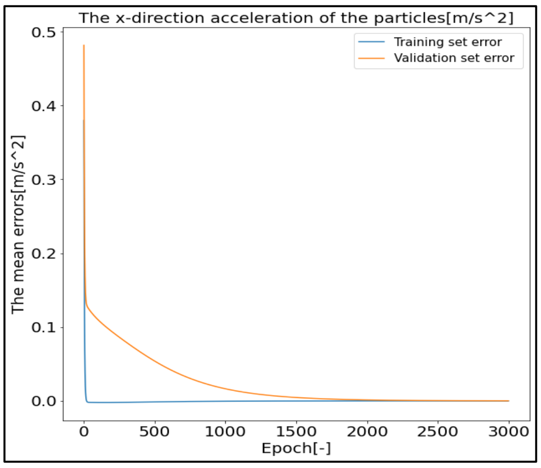
(b)



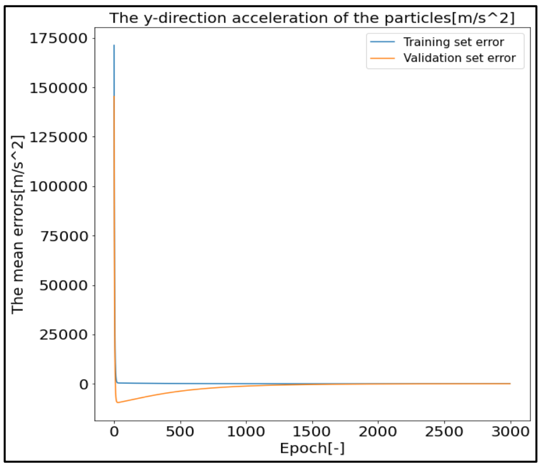
(c)



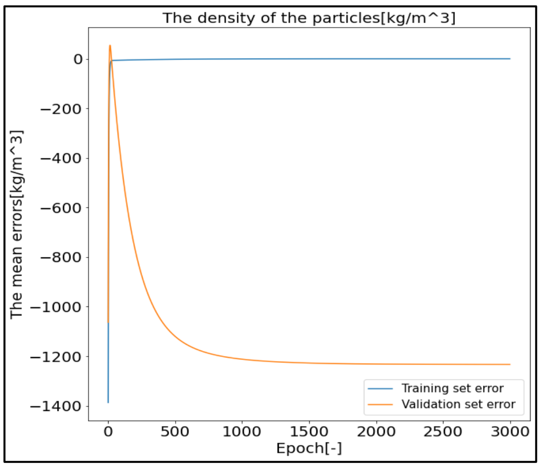
(d)



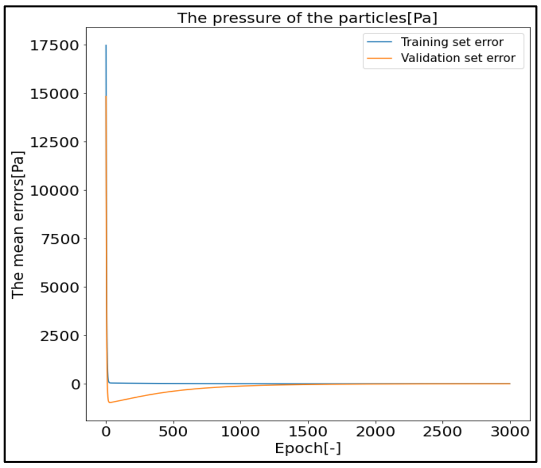
(e)



(f)



(g)



(h)

Figure 22 the results of each machine learning model (a: the x-coordinates of the particles, b: the y-coordinates of the particles, c: The x-direction velocity of the particles, d: The y-direction velocity of the particles, e: The x-direction acceleration of the particles, f: The y-direction acceleration of the particles, g: The pressure of the particles. h: The density of the particles)

Figure 22 shows the results of each machine learning model, and Table 5 represent the differences between actual data and predicted data that is computed with the validation dataset, which are called "Errors". The machine learning models were trained until 3000 epochs. As can be seen in Figure 22, each machine learning model has a point from which there is not much changed. In general, the epoch depends on users. Therefore, machine learning training can be stopped in order to save time and computational resources depending on the users.

Epochs	$X[m]$	$Y[m]$	$v_x\left[\frac{m}{s}\right]$	$v_y\left[\frac{m}{s}\right]$	$\alpha_x\left[\frac{m}{s^2}\right]$	$\alpha_y\left[\frac{m}{s^2}\right]$	$\rho\left[\frac{kg}{m^3}\right]$	$P[Pa]$
2970	7735.313	2889.913	123488.9	31017.65	0.002339	191.0681	23430.4	42.92505
2971	7735.316	2889.915	123488.9	31017.68	0.002333	190.6081	23430.41	42.87812
2972	7735.318	2889.918	123489	31017.7	0.002327	190.1492	23430.42	42.83131
2973	7735.321	2889.92	123489	31017.72	0.002322	189.6914	23430.43	42.78461
2974	7735.324	2889.922	123489.1	31017.74	0.002316	189.2347	23430.43	42.73802
2975	7735.326	2889.925	123489.1	31017.76	0.002311	188.7791	23430.44	42.69155
2976	7735.329	2889.927	123489.2	31017.78	0.002305	188.3247	23430.45	42.64518
2977	7735.332	2889.929	123489.2	31017.8	0.002299	187.8713	23430.46	42.59893
2978	7735.334	2889.932	123489.2	31017.83	0.002294	187.4189	23430.46	42.55279
2979	7735.337	2889.934	123489.3	31017.85	0.002288	186.9677	23430.47	42.50676
2980	7735.34	2889.936	123489.3	31017.87	0.002283	186.5176	23430.48	42.46084
2981	7735.342	2889.939	123489.4	31017.89	0.002277	186.0685	23430.48	42.41503
2982	7735.345	2889.941	123489.4	31017.91	0.002272	185.6206	23430.49	42.36933
2983	7735.347	2889.943	123489.5	31017.93	0.002266	185.1737	23430.5	42.32374
2984	7735.35	2889.946	123489.5	31017.95	0.002261	184.7279	23430.51	42.27826

2985	7735.352	2889.948	123489.5	31017.97	0.002255	184.2831	23430.51	42.23289
2986	7735.355	2889.95	123489.6	31018	0.00225	183.8395	23430.52	42.18763
2987	7735.358	2889.952	123489.6	31018.02	0.002244	183.3969	23430.53	42.14248
2988	7735.36	2889.955	123489.7	31018.04	0.002239	182.9553	23430.54	42.09744
2989	7735.363	2889.957	123489.7	31018.06	0.002234	182.5149	23430.54	42.0525
2990	7735.365	2889.959	123489.7	31018.08	0.002228	182.0754	23430.55	42.00768
2991	7735.368	2889.962	123489.8	31018.1	0.002223	181.6371	23430.56	41.96296
2992	7735.37	2889.964	123489.8	31018.12	0.002217	181.1998	23430.57	41.91835
2993	7735.373	2889.966	123489.9	31018.14	0.002212	180.7635	23430.57	41.87385
2994	7735.375	2889.968	123489.9	31018.16	0.002207	180.3283	23430.58	41.82945
2995	7735.378	2889.97	123489.9	31018.18	0.002201	179.8942	23430.59	41.78516
2996	7735.38	2889.973	123490	31018.2	0.002196	179.4611	23430.59	41.74098
2997	7735.383	2889.975	123490	31018.22	0.002191	179.029	23430.6	41.6969
2998	7735.385	2889.977	123490.1	31018.24	0.002185	178.598	23430.61	41.65293
2999	7735.388	2889.979	123490.1	31018.26	0.00218	178.168	23430.61	41.60907

Table 5 The sum of error (x: the x-coordinates of the particles, y: the y-coordinates of the particles, v_x: The x-direction velocity of the particles, v_y: The y-direction velocity of the particles, α_x: The x-direction acceleration of the particles, α_y: The y-direction acceleration of the particles, ρ: The density of the particles. p: The pressure of the particles)

Initial weights and bias play significant roles even though they are modified at every epoch. (Sun, 2020) In the previous article, (Yadav, 2018) presented the differences between zero initialization and random initialization in Neural Networks. In the (Yadav, 2018), the random initialization was more efficient even though it dose not mean that the random initialization is always correct initialization.

Otherwise, generally linear regression machine learning models have less weights and bias than Neural Networks. Therefore, in linear regression model, initialization weights and bias can be less important than initialization of Neural Networks in order to save time and carry out training. Moreover, Initialization weights and bias of linear regression model can be less important due to the fact that it has simple back propagation. Figure 23 and Figure 24 show how machine learning models are trained when they have different weights. As can be seen in Figure 23 and Figure 24, in this study, the two machine learning models which are trained multivariable linear regression algorithm and predict the x-coordinates of the particles obtain similar error (-400 m) in the end of training. However, there is not a theoretical method to find optimal points of hyperparameters. (Brownlee, 2017) consequently, the initial weights and bias of the machine learning models in this paper can be not the best initial weights and bias.

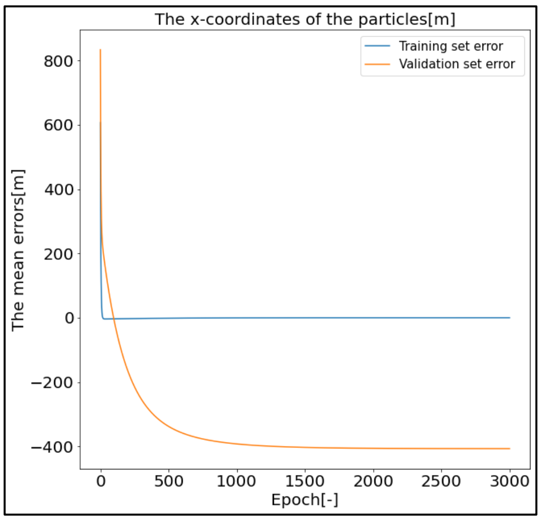


Figure 23 Training graph when the standard deviation of initial weights is $\boldsymbol{\sigma = 0.01}$ and the mean of weights is $\boldsymbol{m_{weights} = 0}$

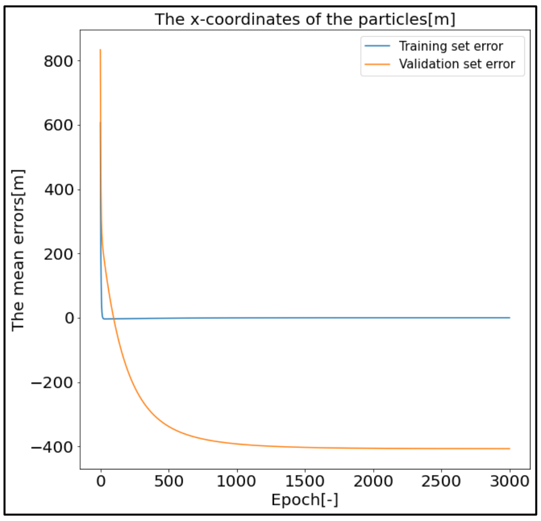


Figure 24 Training graph when the standard deviation of initial weights is $\boldsymbol{\sigma = 1}$ and the mean of weights is $\boldsymbol{m_{weights} = 0}$

4.2.2 Dataset

In order to prevent overfitting and train the machine learning model properly, it is required to divide the labeled data into three datasets, with training dataset, validation dataset, and test dataset, respectively. The machine learning models are trained based on their training dataset. After training, the machine learning model predict the target with the validation dataset. The hyperparameters of the machine learning were determined depending on the result of using the validation dataset. The accuracy of the machine learning model is estimated by using the test dataset. In this study, the total amount of the labeled data is one hundred. The amount of data in the training dataset, validation dataset, and test dataset are 56, 19, and 25 respectively.

4.2.3 Algorithm

The accuracies of the machine learning models are determined according to many factors. As mentioned earlier, machine learning models are required to be setup with well-defined hyperparameters. If machine learning models are overfitted to the training dataset, the machine learning models work precisely only in the training datasets. Moreover, the quality of data affects to the machine learning models. If there are a lot of outliers, which are out of tendencies, machine learning models cannot modify their weights and bias correctly.

Machine learning models, which are trained Linear Regression algorithm, are assumed that the relation between parameters and each target is linear. Even though, at first it is difficult to analyze the relation whether it is linear or not, it is an efficient choice to assume that the relation is linear.

Because generally Linear Regression algorithm requires less weights and bias compared to Artificial Neural Networks. In this study, each machine learning model had 80,000 weights and 80,000 biases totally in order to predict the target values of all particles in all time steps at once. It was also possible to perform machine learning sequentially for each time step to save memory capacity.

In this study, it is assumed that the relation between the target values and the input parameters linear. The input parameters are the mass and kinematic viscosity of particles, the rotating speed of each roll. The targets are the positions, velocities, accelerations, density, and pressure of particles at every time step.

This study focusses on the point that whether machine learning models, which are trained with Multivariable Linear Regression algorithm, are able to predict the target values of the fluid precisely.

Machine Learning is a useful technique to find regularity between input parameters and target values. In this study, the regularity of fluid flows is the Smoothed Particle Hydrodynamics solver due to the fact that all targets are the results of the simulation. Therefore, machine learning models must be able to perform like the SPH solver for accurate results.

4.2.4 Error analysis

In this study, the following process was set up to evaluate the model. The process is similar with forward propagation of neurons of machine learning models.

Predicted value by machine learning:

$$\widehat{y_p} = \sum_{i=1}^{n} \omega_i x_{i,test} + b, \qquad (37)$$

where $x_{i,test}$ is a parameter from test dataset, ω_i is weights, $b,$ is bias,

$\widehat{y_p}$ is predicted data.

The standard of accuracy of machine learning models is absolute errors on account of the fact that the predicted data $\widehat{y_p}$ can be positive or negative.

Absolute errors can be calculated by subtracting simulation data from the predicted data:

$$Absolute\ Errors = \left|\widehat{y_p} - \widehat{y_a}\right|, \qquad (38)$$

where $\widehat{y_p}$ is predicted data, $\widehat{y_a}$ is actual data.

$\widehat{y_p}$ and $\widehat{y_a}$ are 1-dimensional arrays due to the fact that the machine learning models predicted the behaviors of 100 particles for 200 time-steps on the number of simulation data in test dataset (=25). It means that they contained 500000 values in each NumPy array. According to their data-structure, the NumPy array which contain the absolute errors has same data-structure with them.

The mean absolute error e_m of each machine learning model is expressed through the following equation.

$$e_m = \frac{\sum_{n_D=1}^{n_D=25} \sum_{n_t=1}^{n_t=200} \sum_{n_p=1}^{n_p=100} e_{n_p,n_t}}{n_T}, \qquad (39)$$

where, n_t is time-step, n_p is particle number, n_D is the number of simulation data in test dataset.

$$n_T = The\ number\ of\ timesteps \times The\ number\ of\ particles \\ \times The\ number\ of\ simulation\ data\ of\ test\ Dataset \\ = 500000.$$

Table 6 shows the results which are computed by Eq.39, and the Table 6 present how different the predicted values e_m of machine learning models are on average from the actual value.

	$X[m]$	$Y[m]$	$v_x\left[\frac{m}{s}\right]$	$v_y\left[\frac{m}{s}\right]$	$\alpha_x\left[\frac{m}{s^2}\right]$	$\alpha_y\left[\frac{m}{s^2}\right]$	$\rho\left[\frac{kg}{m^3}\right]$	$P[Pa]$
Error	0.002111	0.004968	0.045671	0.116861	0.001684	0.001878	0.014905	0.00004

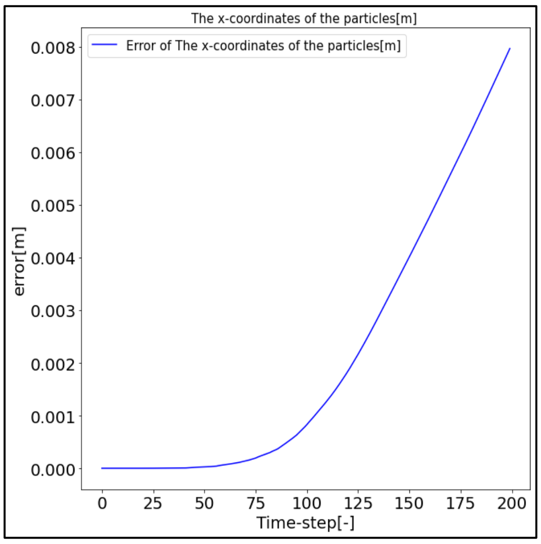
Table 6 The mean absolute error of each machine learning model (x: the x-coordinates of the particles, y: the y-coordinates of the particles, v_x: The x-direction velocity of the particles, v_y: The y-direction velocity of the particles, α_x: The x-direction acceleration of the particles, α_y: The y-direction acceleration of the particles, ρ: The density of the particles. p: The pressure of the particles)

4.2.4.1 Error per time-step.

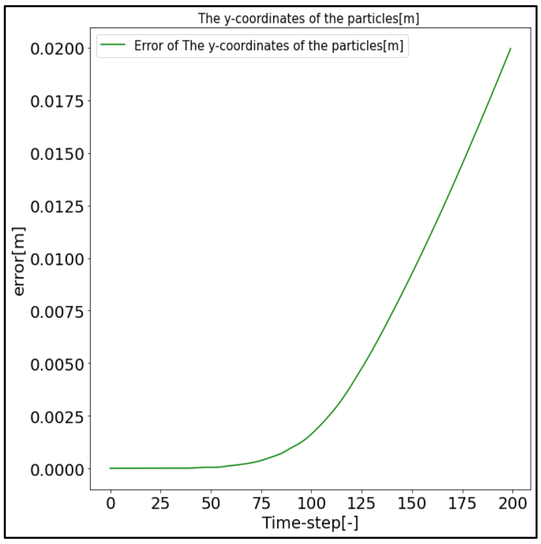
As mentioned earlier, the NumPy array of absolute error contains 500000 values. This session shows that the errors in the machine learning model increase significantly from which time step.

$$e_m = \frac{\sum_{n_D=1}^{n_D=25} \sum_{n_p=1}^{n_p=100} e_{n_p,n_t}}{n_{p,T} \times n_{D,T}}, \qquad (40)$$

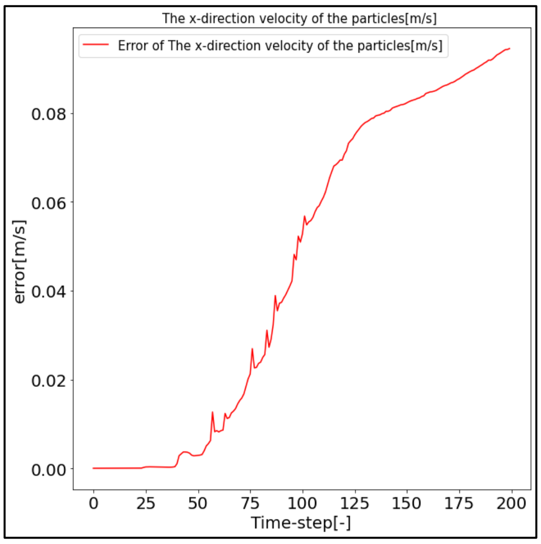
where, $n_{p,T}$ is the number of particles, and $n_{D,T}$.



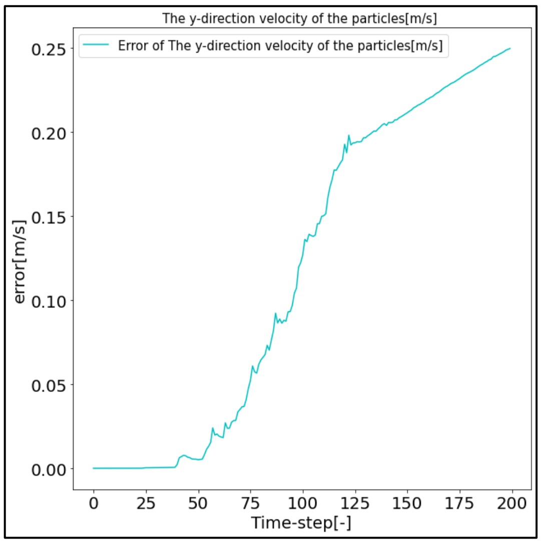
(a)



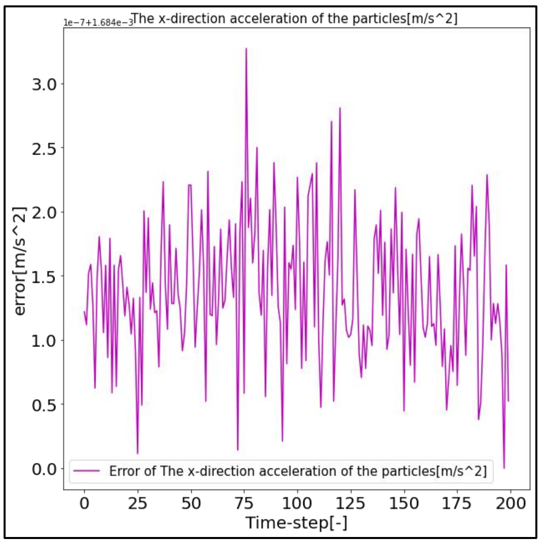
(b)



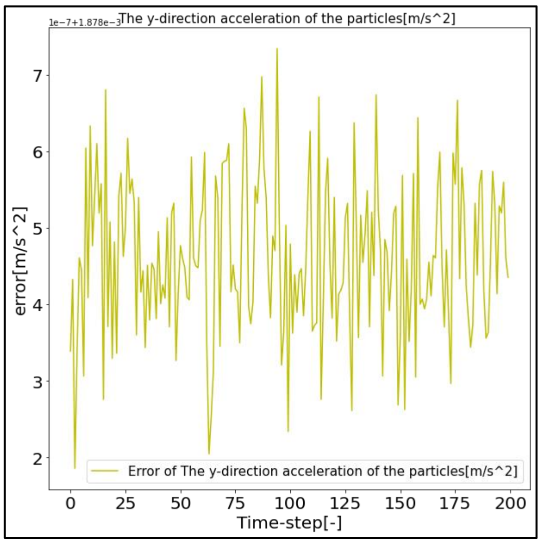
(c)



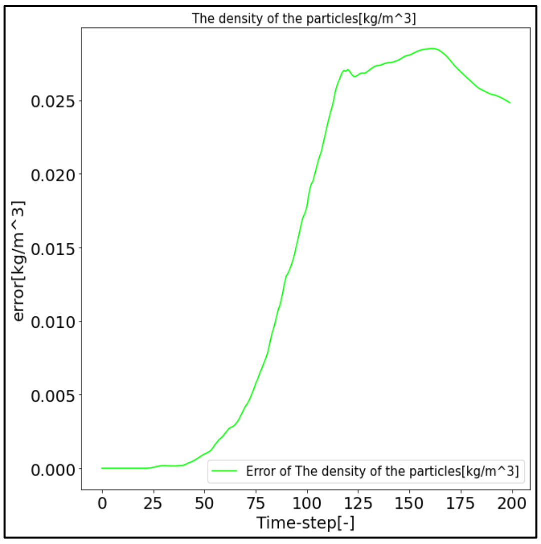
(d)



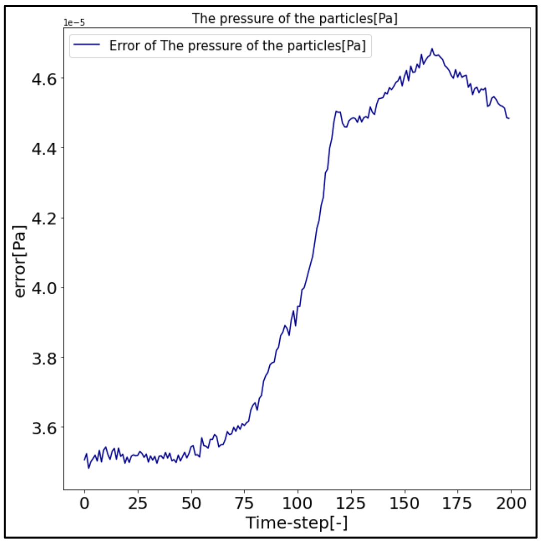
(e)



(f)



(g)



(h)

Figure 25 Errors per time-step. model (a: the x-coordinates of the particles, b: the y-coordinates of the particles, c: The x-direction velocity of the particles, d: The y-direction velocity of the particles, e: The x-direction acceleration of the particles, f: The y-direction acceleration of the particles, g: The pressure of the particles. h: The density of the particles)

Figure 25 shows the absolute error values for each time step for each target value. As can be seen in Figure 25 (a), (b), (c), and (d) the error values consistently increase over time. Whereas the absolute error values in Figure 25 (g) and(h) decrease after a certain time-step. In addition, the absolute errors in Figure 25 (e) and (f) do not correlated with time-step.

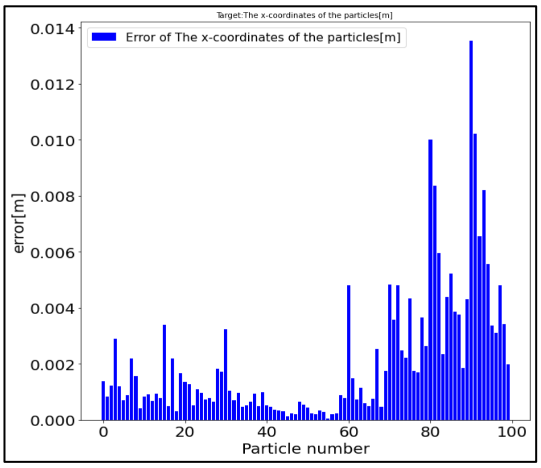
4.2.4.2 Error per particle.

As mentioned above, the two rotating rolls affect the particles significantly. Due to the fact that the frequency which particles collide with the rolls is different depending on the location of particles, it is required to find out how the absolute errors become changed based on the frequency.

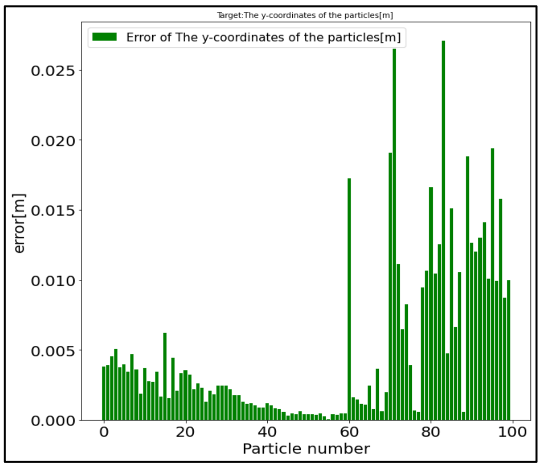
In order to compute the errors per particles $e_{p,m}$, it is necessary to define the function as follow:

$$e_{p,m} = \frac{\sum_{n_D=1}^{n_D=25} \sum_{n_t=1}^{n_t=200} e_{n_p,n_t}}{n_{t,T} \times n_{D,T}}, \tag{41}$$

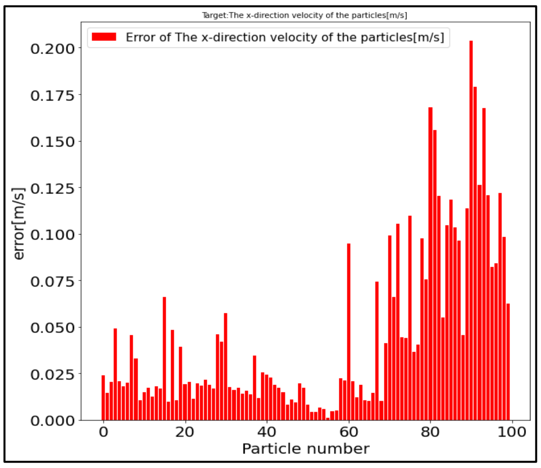
where, n_t is time-step, and $n_{p,T}$ is the number of time-steps, $n_{D,T}$ is the number of simulation data of test dataset.



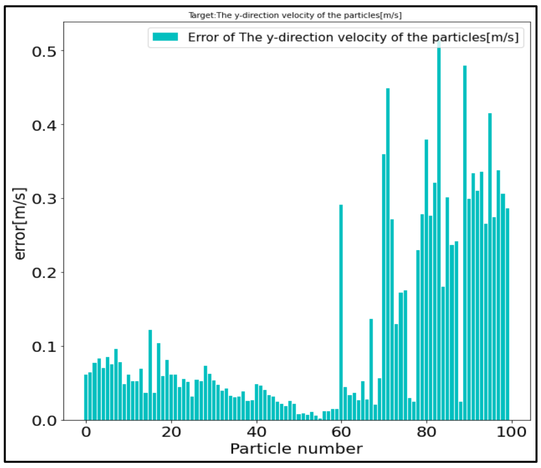
(a)



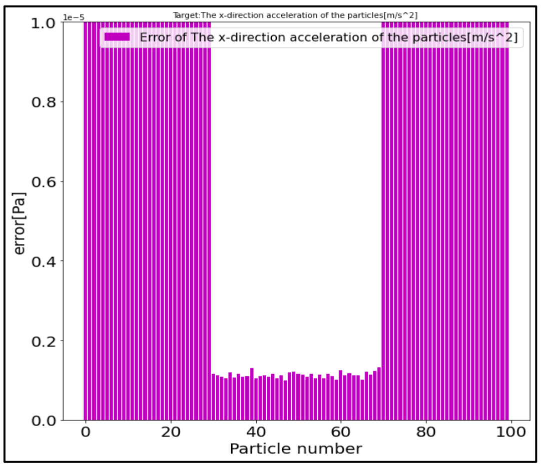
(b)



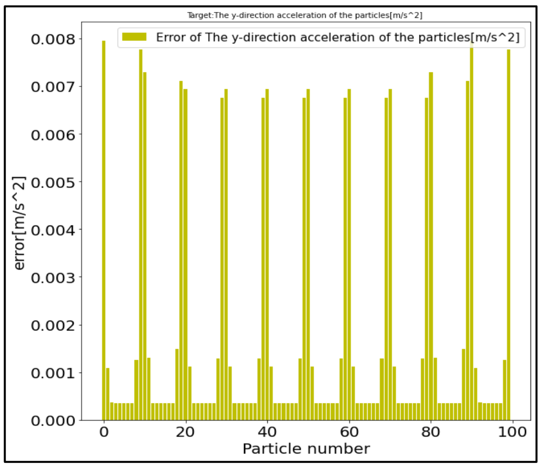
(c)



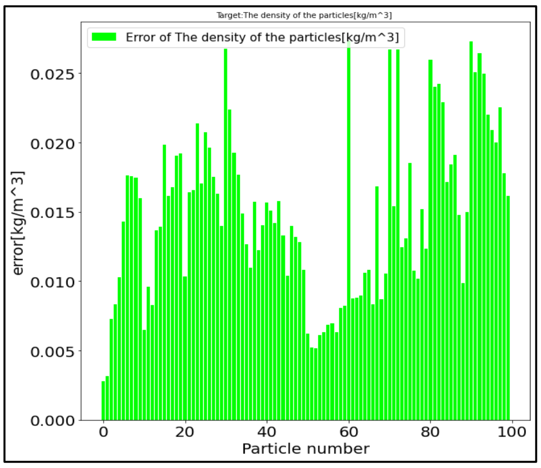
(d)



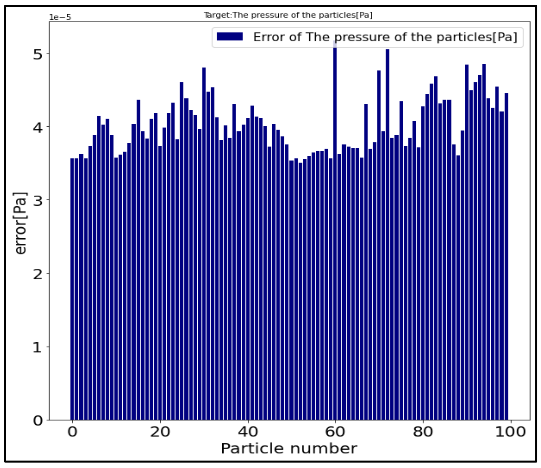
(e)



(f)



(g)



(h)

Figure 26 The absolute error per particle (a: the x-coordinates of the particles, b: the y-coordinates of the particles, c: The x-direction velocity of the particles, d: The y-direction velocity of the particles, e: The x-direction acceleration of the particles, f: The y-direction acceleration of the particles, g: The pressure of the particles. h: The density of the particles)

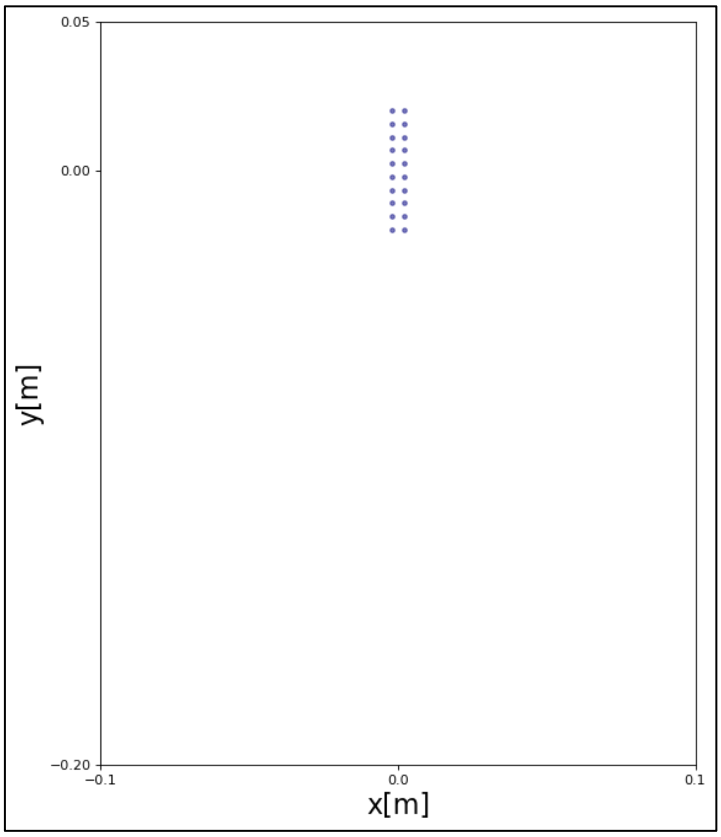


Figure 27 The positions of particles [From the particle number 40 to 59]

Figure 26 shows the absolute error values for each particle for each target value. Due to the fact that the two rolls rotated anticlockwise, it can be seen that particle number from 60 to 99 have greater error compared to particle number from 0 to 39.

Additionally, As can be seen in Figure 26, the particle number from 40 to 59 has a small number of absolute errors. As mentioned earlier in chapter2, particles are generated in the specific area as can be seen in Figure 2. All the positions of particles were fixed at the first-time step. Figure 27 presents the position of particle number from 40 to 59 at the first time-step. Figure 28 and Figure 29 show the number of average times that the Smoothed Particle Hydrodynamics Solver executed "AddRollerEffect" (it can be seen chapter 2.6 Roller Algorithm) for each particle and each time step, respectively. In Figure 28, the particle number from 40 to 59 were seldomly computed by "AddRollerEffect". In figure 28, it can be seen that the "AddRollerEffect" is more executed when the fluid particles pass through the gap. Therefore, based on Figure 25, Figure 26, Figure 27, and Figure 28, machine learning errors increase when particles collide with rolls.

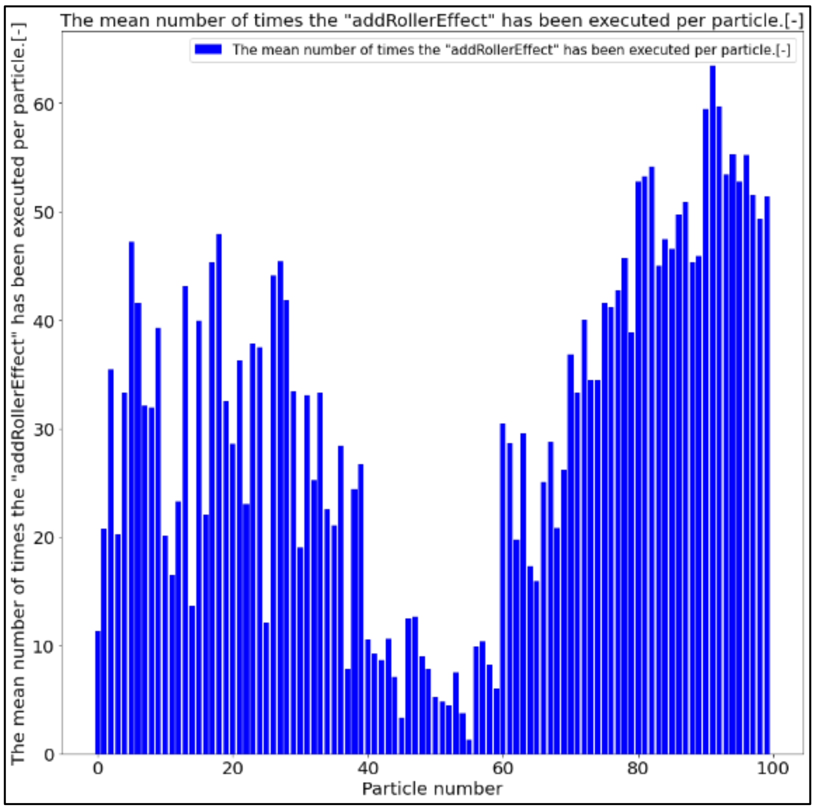


Figure 28 The mean number of times the "addRollerEffect" has been executed per particle

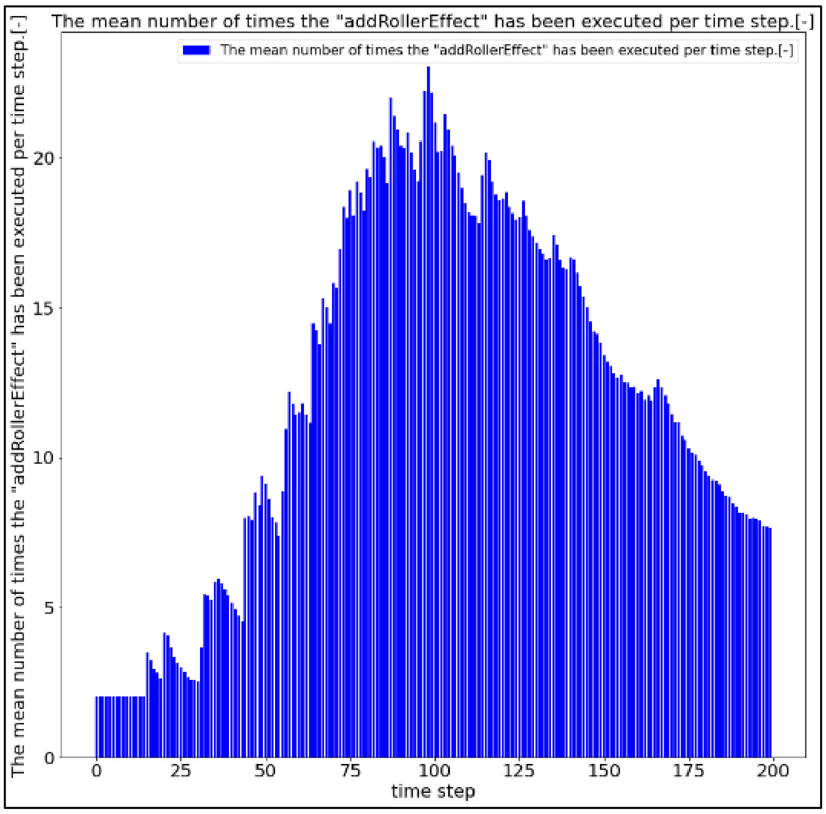
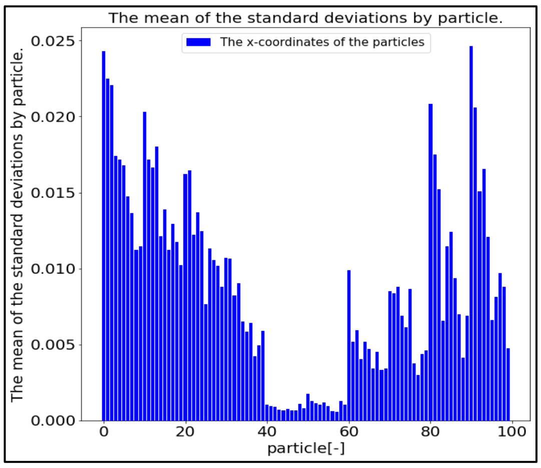
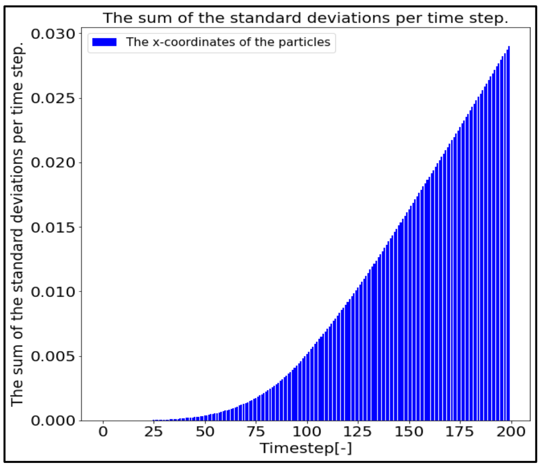


Figure 29 The mean number of times the "addRollerEffect" has been executed per time step

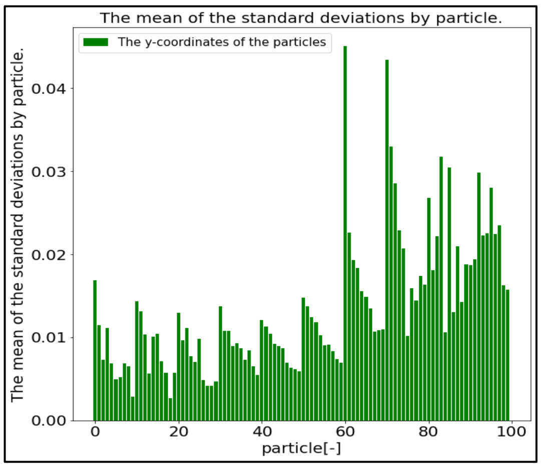
4.2.4.4 Relation between the standard deviation of data and the absolute errors.



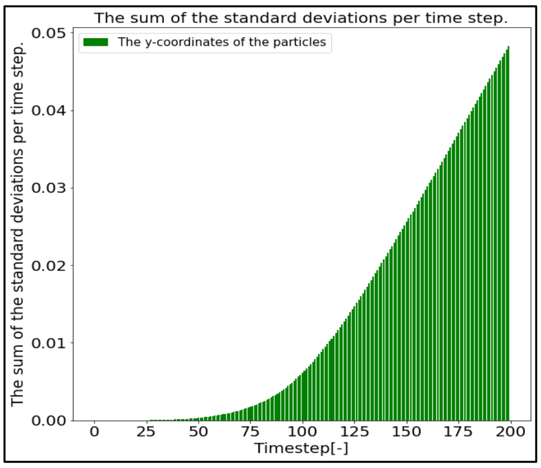
(a)



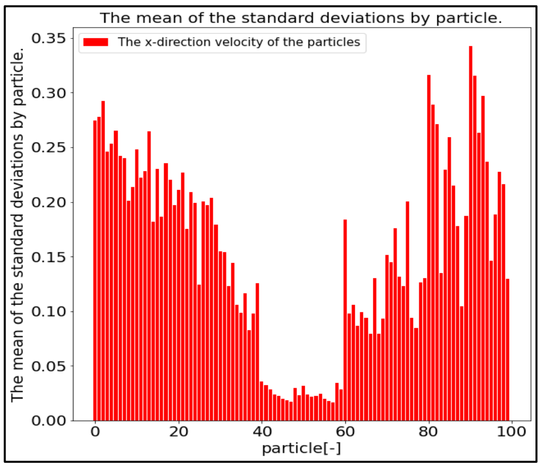
(b)



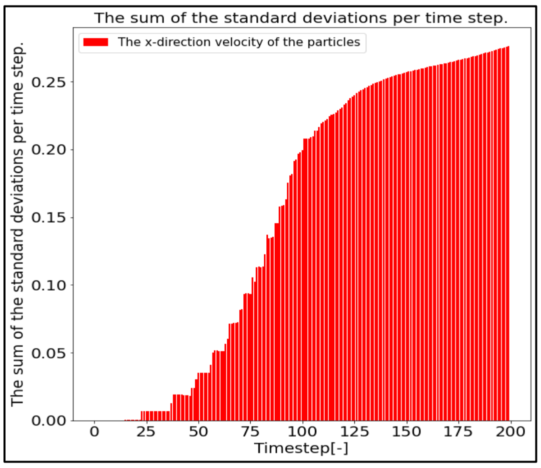
(c)



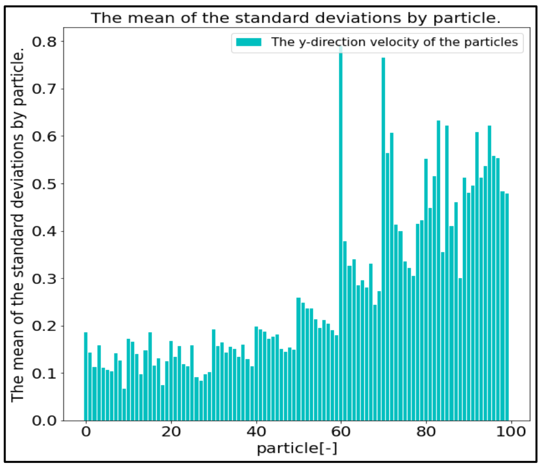
(d)



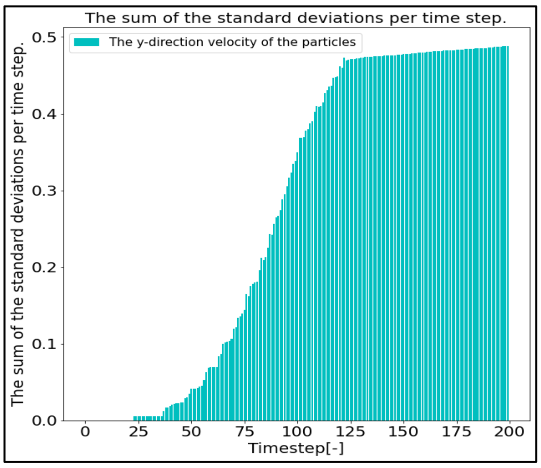
(e)



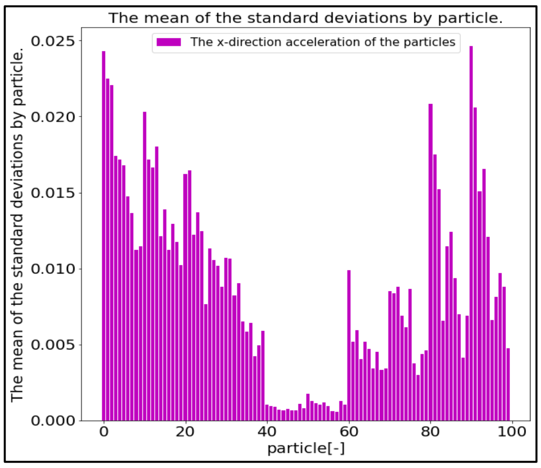
(f)


The mean of the standard deviations by particle.
The y-direction velocity of the particles
The mean of the standard deviations by particle.
particle[-]
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0
20
40
60
80
100

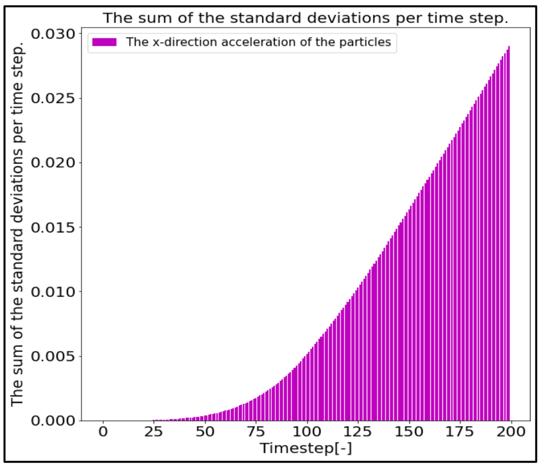
(g)


The sum of the standard deviations per time step.
The y-direction velocity of the particles
The sum of the standard deviations per time step.
Timestep[-]
0.0
0.1
0.2
0.3
0.4
0.5
0
25
50
75
100
125
150
175
200

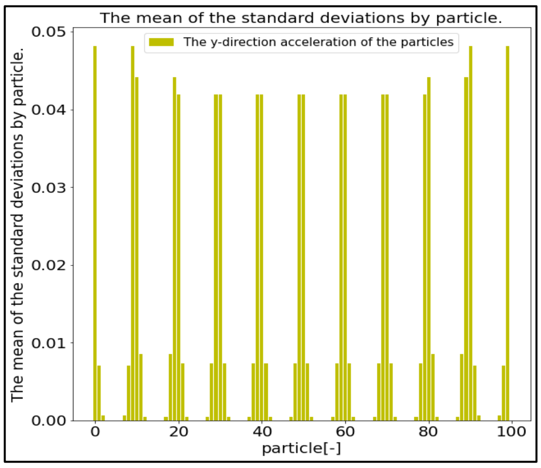
(h)


The mean of the standard deviations by particle.
The x-direction acceleration of the particles
The mean of the standard deviations by particle.
particle[-]
0.000
0.005
0.010
0.015
0.020
0.025
0
20
40
60
80
100

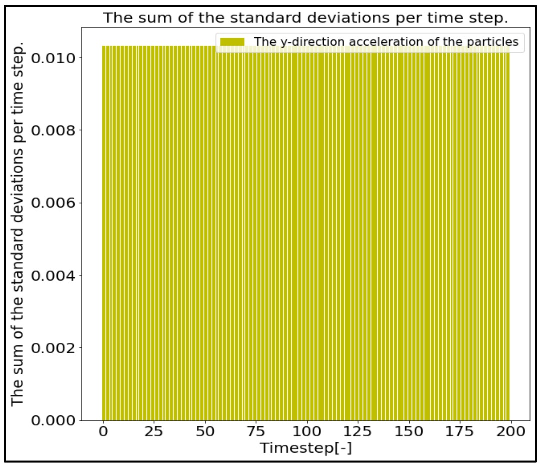
(l)


The sum of the standard deviations per time step.
The x-direction acceleration of the particles
The sum of the standard deviations per time step.
Timestep[-]
0.000
0.005
0.010
0.015
0.020
0.025
0.030
0
25
50
75
100
125
150
175
200

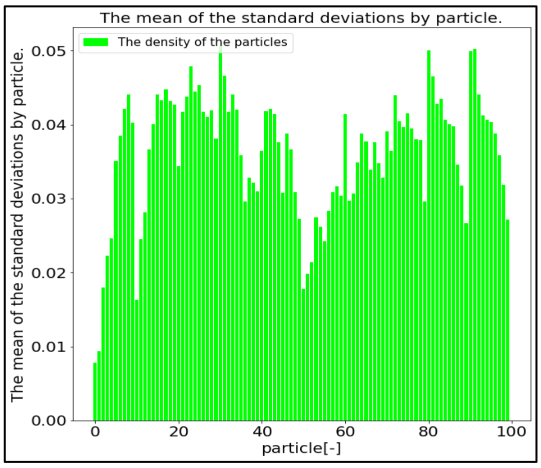
(j)


The mean of the standard deviations by particle.
The y-direction acceleration of the particles
The mean of the standard deviations by particle.
particle[-]
0.00
0.01
0.02
0.03
0.04
0.05
0
20
40
60
80
100

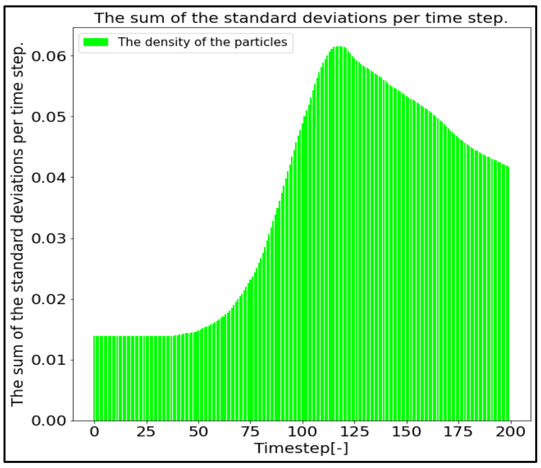
(k)


The sum of the standard deviations per time step.
The y-direction acceleration of the particles
The sum of the standard deviations per time step.
Timestep[-]
0.000
0.002
0.004
0.006
0.008
0.010
0
25
50
75
100
125
150
175
200

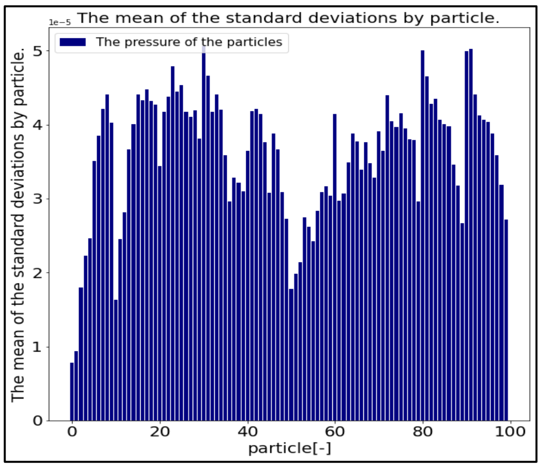
(l)



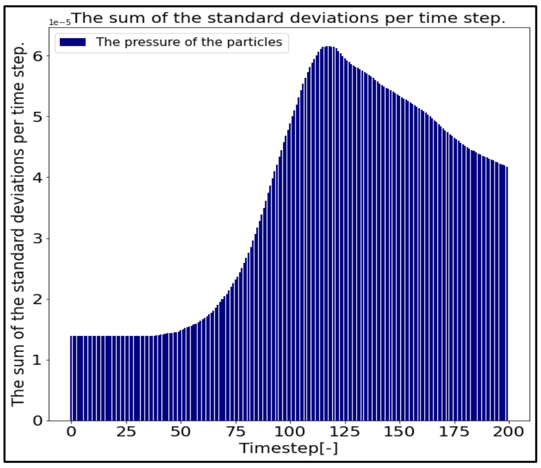
(m)



(n)



(o)



(p)

Figure 30 The sum of standard deviation per particle and time-step. (a, b: the x-coordinates of the particles, c, d: the y-coordinates of the particles, e, f: The x-direction velocity of the particles, g, h: The y-direction velocity of the particles, i, j : The x-direction acceleration of the particles, k, l: The y-direction acceleration of the particles, m, n: The pressure of the particles. o, p: The density of the particles)

As mentioned above, machine learning errors are proportional to the number of times the particles collide with the rolls. On the other hand, when particles collide with the rolls less, fewer errors occur in the machine learning model.

In addition, errors of machine learning models are related with the standard deviation of data. In general, if the standard deviation between the data is very small, the errors of machine learning models become small, and the weights are close to zero. In this session, the standard deviations of simulation data are shown, and it is compared with the errors of machine learning models, which are mentioned in the chapter 4.2.4.1 and 4.2.4.2.

As mentioned in the chapter 2, each target dataset consists of 100 simulation data, and each simulation data contains information about the behavior of a fluid composed of 100 particles for 200 time-steps. Therefore, the NumPy array of the standard deviation consists of 20000 values on account of fact that each particle has a standard deviation for each time-step.

The equation of standard deviation is as follow:

$$\sigma = \sqrt{\frac{(X_s - X_m)^2}{n}}, \tag{42}$$

where, σ is standard deviation, X_s is simulation data, and X_m is the mean value.

The equation for averaging the standard deviation values for each particle $\sigma_{p,m}$ is as follows.

$$\sigma_{p,m} = \frac{\sum_{n_t=1}^{n_t=200} \sigma_{n_p,n_t}}{n_{t,T}}, \tag{43}$$

where, $n_{p,T}$ is the total number of time-steps.

The equation for averaging the standard deviation values for each time-step $\sigma_{t,m}$ is as follows.

$$\sigma_{t,m} = \frac{\sum_{n_p=1}^{n_p=100} \sigma_{n_p,n_t}}{n_{p,T}}, \tag{44}$$

where, $n_{p,T}$ is the total number of particles.

Figure 30 shows the sum of standard deviation of targets per particle and time-step respectively. When the targets are pressure and density in Figure 30(m) and (o), the standard deviation of the particles (from number 40 to 60) is large, even if they don't frequently collide with rolls, on account of the fact that pressure and density are influenced by neighboring particles directly through smoothing length h. Except for these two cases, based on Figure 30 (a), (c), (e), (g), and (i), the particles (from number 40 to 60) that collide less with the rolls, the standard deviation of the particles (from number 40 to 60) is small.

As a result, most of errors occurred around the two rotating rolls. Therefore, in further study, it is considered that additional strategies might be required in order to improve machine learning models precisely. Due to the accuracy of machine learning was inversely proportional to the frequency of particles colliding with the rolls in almost all case, it is necessary to change the multivariable linear regression algorithm to Artificial Neural Network which is why is specialized in finding non-linear regularities. (May et al., 2008) Alternatively, machine learning can be used partially only for time-steps that do not collide with rolls.

5. CONCLUSION

This study therefore suggests the method to use supervised machine learning based on the data which is computed by SPH solvers in reverse roll coating. In the simulation, the fluid flew between the two rolls rotating counterclockwise. Based on the simulation, the machine learning models were trained with multivariable linear regression which is not require a lot of computational resources compared to other algorithms. Moreover, in general, numerical simulation is time-consuming process. Therefore, it is necessary that machine learning takes less time in order to become an effective method to overcome the limitation of numerical simulation in CFD. The Machine Learning models predict the position, velocities, accelerations, pressure, and density of particles at every time-step. Consequently, errors of machine learning model are proportion to the number of colliding with the rolls.

Based on this study, particle-based solver can be one option to compute reverse roll coating simulation. Due to the fact that particle-based solver does not require generating meshes, users can save time to create mesh in reverse roll coating. Moreover, by using machine learning technique, users can predict how fluid flows based on data that is simulated before.

In further study, in order to improve the accuracy of machine learning models, it might be necessary to switch to Deep Learning algorithms on account of the fact that Deep Learning algorithms are fitted to train models based on data which have non-linear relation between parameters, or it might be helpful to use the machine learning model partially only in the free space.

Numerical simulation in Computational Fluid Dynamics will be required computational resources (CPU, GPU, and memory) more and more in the future, and there will be a lot of times to compute simulations which have similar parameters. As a results, Machine learning will become one of the best solutions in order to save the computational resources.

REFERENCES

Brownlee, J. (2017, June 17 2019). *What is the Difference Between a Parameter and a Hyperparameter?* Retrieved November 20 from https://machinelearningmastery.com/difference-between-a-parameter-and-a-hyperparameter/

Brownlee, J. (2018, October 26, 2019). *Difference Between a Batch and an Epoch in a Neural Network*. Retrieved November 20 from https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/

Debiagi, P., Nicolai, H., Han, W., Janicka, J., & Hasse, C. (2020). Machine learning for predictive coal combustion CFD simulations—From detailed kinetics to HDMR Reduced-Order models. *Fuel*, *274*, 117720.

Géron, A. (11 oct. 2019). *Hands-on Machine Learning with Scikit-learn, Keras, and TensorFlow*. O`REILLY'.

Hinton, G. E. (2012). A practical guide to training restricted Boltzmann machines. In *Neural networks: Tricks of the trade* (pp. 599-619). Springer.

Koschier, D., Bender, J., Solenthaler, B., & Teschner, M. (2020). Smoothed particle hydrodynamics techniques for the physics based simulation of fluids and solids. *arXiv preprint arXiv:2009.06944*.

learn, s. (2022). *sklearn.preprocessing.MinMaxScaler*. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html

Liu, M. B., & Liu, G. R. (2010). Smoothed Particle Hydrodynamics (SPH): an Overview and Recent Developments. *Archives of Computational Methods in Engineering*, *17*(1), 25-76. https://doi.org/10.1007/s11831-010-9040-7

Lucy, L. (1977). Smoothed particle hydrodynamics-theory and application to non-spherical stars. *Astron. J*, *82*, 1013-1024.

May, R. J., Maier, H. R., Dandy, G. C., & Fernando, T. G. (2008). Non-linear variable selection for artificial neural networks using partial mutual information. *Environmental Modelling & Software*, *23*(10-11), 1312-1326.

Mc., C. (2017). *Machine learning fundamentals (I): Cost functions and gradient descent*. Retrieved November 20 from https://towardsdatascience.com/machine-learning-fundamentals-via-linear-regression-41a5d11f5220

Micheal, L. (2020). *Hands-On Reinforcement Learning for Games : Implementing Self-learning Agents in Games Using Artificial Intelligence Techniques* [Book]. Packt Publishing. https://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=2346941&lang=ko&site=ehost-live

Mocz, P. (2020). *Create Your Own Smoothed-Particle Hydrodynamics Simulation (With Python)*. https://philip-mocz.medium.com/create-your-own-smoothed-particle-hydrodynamics-simulation-with-python-76e1cec505f1

Moubin, L., & Gui-rong, L. (2003). *Smoothed Particle Hydrodynamics: A Meshfree Particle Method* [Book]. World Scientific. https://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=134095&lang=ko&site=ehost-live

Onderik, J., & Durikovic, R. (2008). Efficient neighbor search for particle-based fluids. *Journal of the Applied Mathematics, Statistics and Informatics*, *4*(1), 29-43.

Ringstad, K. E., Banasiak, K., Ervik, Å., & Hafner, A. (2021). Machine learning and CFD for mapping and optimization of CO2 ejectors. *Applied Thermal Engineering*, 117604.

Saikat Dutt, S. C., Amit Kumar Das. *Machine Learning*. Pearson Education India.

Sun, P. H. (2020). *Do it! 정직하게 코딩하며 배우는 딥러닝 입문*.

Svensson, P., & Galfi, F. (2021). Performance evaluation of NumPy, SciPy, PyMEL and OpenMaya compared to the C++ API in Autodesk Maya. In.

Teschner, M. *Simulation in Computer Graphics Particle-based Fluid Simulation*. https://cg.informatik.uni-freiburg.de/course_notes/sim_10_sph.pdf

Usman, A., Rafiq, M., Saeed, M., Nauman, A., Almqvist, A., & Liwicki, M. (2021). Machine Learning Computational Fluid Dynamics. 2021 Swedish Artificial Intelligence Society Workshop (SAIS),

Yadav, S. (2018). *Weight Initialization Techniques in Neural Networks*. Retrieved November 20 from https://towardsdatascience.com/weight-initialization-techniques-in-neural-networks-26c649eb3b78

Zhou, Q., & Ooka, R. (2020). Comparison of different deep neural network architectures for isothermal indoor airflow prediction. Building Simulation,

APPENDIX

1. Simulation

1.1 main.py

```
   """
Reference:

* coding style
Philip Mocz (2020) Princeton Univeristy, @PMocz
https://github.com/pmocz/sph-python

* kernel function
Smoothed Particle Hydrodynamics
Techniques for the Physics Based Simulation of Fluids and Solids
Dan Koschier1, Jan Bender2, Barbara Solenthaler3, and Matthias Teschner4
"""
import os


def main():

    import numpy as np
    from Particles import Particles
    from FluidProperties import FluidProperties
    from Time import Time
    from Mesh import Mesh
    from Roller import Roller
    import time as SimulationTime

    # simulation start
    start =SimulationTime.time()


    # smoothed length
    h=0.01
    # overview
    NData = 100
    NParticles = 100
    NTimeSteps = 200
    rollRadius = 0.048

    # set the random number generator seed
    np.random.seed(42)

    # random parameters
    rnMass = np.random.rand(NData) * 0.0001 + 0.0001
    rnLiquidViscosity = (np.random.rand(NData) * 3)* 10**-7
    rnAngularVelocity1 = np.random.rand(NData) * 30 + 10
    rnAngularVelocity2 = np.random.rand(NData) * 30 + 10

    # datasets
    npInputDataset = None
    npXTargetDataset = None
    npYTargetDataset = None
    npUTargetDataset = None
    npVTargetDataset = None
    npAccXTargetDataset = None
    npAccYTargetDataset = None
    npRhoTargetDataset = None
    npPTargetDataset = None
    npCollidingTotal = None

    os.makedirs(f"./DATA", exist_ok=True)

    for i in range(NData):

        print("NData",i)

        # set the random number generator seed
        np.random.seed(42)

        # simulation parameters
        time = Time(start_time=0, end_time=0.2, time_step=NTimeSteps)
        liquid_property = FluidProperties(mass=rnMass[i], SmoothingLength=h, LiquidViscosity=rnLiquidViscosity[i],
LiquidDropletRadius=1,
                                     restDensity=1000, stiffnessConstant=1)
        mesh = Mesh(start_x=-0.10, end_x= 0.10 , start_y= -0.20, end_y= 0.05)
        Roller1 = Roller(CenterX=0.05, CenterY=-0.06, Radius=rollRadius, AngularVelocity= rnAngularVelocity1[i])
        Roller2 = Roller(CenterX=-0.05, CenterY=-0.06, Radius=rollRadius, AngularVelocity=rnAngularVelocity2[i])
        particles = Particles(time=time, LiquidProperty=liquid_property,Mesh=mesh,)

        # Generate Initial Conditions
        particles.generateInitConditions(startGenerateX=-0.020, endGenerateX=0.020, startGenerateY=-0.020, endGenerateY=0.020,
column=10, row=10)

        # add Input initial x,y data
        X = np.linspace(-0.020, 0.020, 10).reshape(10,1)
        Y = np.linspace(-0.020, 0.020, 10).reshape(10,1)
        posX = (X * np.ones_like(Y).T).flatten().reshape(-1,1)
        posY = (Y * np.ones_like(X).T).T.flatten().reshape(1,-1)


        # calculate initial gravitational accelerations
        particles.acc = particles.getAcc()

        # simulation
        particles.simulateParticles(Roller1=Roller1, Roller2=Roller2)

        # Input data
        npInput = np.full((NTimeSteps,NParticles,1), rnMass[i])
        npInput = np.append(npInput, np.full((NTimeSteps,NParticles,1),rnLiquidViscosity[i]), axis=-1)
        npInput = np.append(npInput,np.full((NTimeSteps,NParticles,1), rnAngularVelocity1[i] * rollRadius), axis=-1)#
"*rollRadius" required to transfer to rotating speed
        npInput = np.append(npInput, np.full((NTimeSteps,NParticles,1), rnAngularVelocity2[i]* rollRadius), axis=-1)
        npInput = np.expand_dims(npInput,axis=0)

        # Target data

        npXTarget = np.expand_dims(particles.npXtarget,axis=0)
        npYTarget = np.expand_dims(particles.npYtarget,axis=0)
        npUTarget = np.expand_dims(particles.npUtarget,axis=0)
        npVTarget = np.expand_dims(particles.npVtarget,axis=0)
        npAccXTarget = np.expand_dims(particles.npAccXtarget,axis=0)
        npAccYTarget = np.expand_dims(particles.npAccYtarget,axis=0)
        npRhoTarget = np.expand_dims(particles.npRhotareget,axis=0)
        npPTarget = np.expand_dims(particles.npPtarget,axis=0)
        npCollinding = np.expand_dims(particles.npCollidingTotal, axis=0)

        # save as dataset
        if i == 0:
            npInputDataset = npInput
            npXTargetDataset = npXTarget
            npYTargetDataset = npYTarget
            npUTargetDataset = npUTarget
            npVTargetDataset = npVTarget
            npAccXTargetDataset = npAccXTarget
            npAccYTargetDataset = npAccYTarget
            npRhoTargetDataset = npRhoTarget
            npPTargetDataset = npPTarget
            npCollidingTotal = npCollinding
            continue

        npInputDataset = np.append(npInputDataset, npInput, axis=0)
        npXTargetDataset = np.append(npXTargetDataset, npXTarget,axis=0)
        npYTargetDataset = np.append( npYTargetDataset, npYTarget,axis=0)
        npUTargetDataset = np.append(npUTargetDataset ,npUTarget,axis=0)
        npVTargetDataset = np.append(npVTargetDataset, npVTarget,axis=0)
        npAccXTargetDataset = np.append(npAccXTargetDataset,npAccXTarget,axis=0)
        npAccYTargetDataset = np.append(npAccYTargetDataset,npAccYTarget,axis=0)
        npRhoTargetDataset = np.append(npRhoTargetDataset,npRhoTarget,axis=0)
        npPTargetDataset = np.append(npPTargetDataset ,npPTarget,axis=0)
        npCollidingTotal = np.append(npCollidingTotal,npCollinding,axis=0)

    # save arrays

    np.save(f"/Users/maxan/Documents/Master'sThesis/DATA/npInputDataset.npy",npInputDataset)
    np.save(f"/Users/maxan/Documents/Master'sThesis/DATA/npXTargetDataset.npy", npXTargetDataset)
    np.save(f"/Users/maxan/Documents/Master'sThesis/DATA/npYTargetDataset.npy",npYTargetDataset)
    np.save(f"/Users/maxan/Documents/Master'sThesis/DATA/npUTargetDataset.npy",npUTargetDataset)
    np.save(f"/Users/maxan/Documents/Master'sThesis/DATA/npVTargetDataset.npy", npVTargetDataset)
    np.save(f"/Users/maxan/Documents/Master'sThesis/DATA/npAccxTargetDataset.npy",npAccXTargetDataset)
    np.save(f"/Users/maxan/Documents/Master'sThesis/DATA/npAccYTargetDataset.npy", npAccYTargetDataset)
    np.save(f"/Users/maxan/Documents/Master'sThesis/DATA/npRhoTargetDataset.npy", npRhoTargetDataset)
    np.save(f"/Users/maxan/Documents/Master'sThesis/DATA/npPTargetDataset.npy",npPTargetDataset)
    np.save(f"/Users/maxan/Documents/Master'sThesis/DATA/npCollidingTotal.npy",npCollidingTotal)
    np.save(f"/Users/maxan/Documents/Master'sThesis/DATA/npRotatingVL.npy", rnAngularVelocity1*rollRadius)
    np.save(f"/Users/maxan/Documents/Master'sThesis/DATA/npRotatingVR.npy", rnAngularVelocity2*rollRadius)

    # simulation end
    end = SimulationTime.time()

    print(f"simulation time: {end -start:.2f} ")
    return 0


if __name__ == "__main__":
    main()
```

1.2 Particles.py

```
   import numpy as np
import os
import matplotlib.pyplot as plt
import pandas as pd
from scipy.special import gamma


class Particles():

    def __init__(self, time, LiquidProperty, Mesh):
        # Simulation parameters
        self.N = None  # Number of particles
        self.Tstart = time.Tstart  # current time of the simulation
        self.TEnd = time.Tend  # time at which simulation ends
        self.TimeStep = time.TimeStep
        self.dt = time.dt  # timestep
        self.m = LiquidProperty.m  # mass
        self.R = LiquidProperty.R  # liquid droplet radius
        self.h = LiquidProperty.h  # smoothing length
        self.k = LiquidProperty.k  # stiffness constant
        self.nu = LiquidProperty.nu  # damping
        self.rho0 = LiquidProperty.rho0
        self.Xstart = Mesh.start_x
        self.Xend = Mesh.end_x
        self.Ystart = Mesh.start_y
        self.Yend = Mesh.end_y
        self.pos = None  # randomly selected positions and velocities
        self.vel = None
        self.acc = None
        self.P = None
        self.rho = None
        self.w = None

        #targets
        self.npWtarget =None
        self.npXtarget = None
        self.npYtarget = None
        self.npUtarget = None
        self.npVtarget = None
        self.npAccXtarget = None
        self.npAccYtarget = None
        self.npRhotareget = None
        self.npPtarget = None

        # The number of colliding with rollers
        self.npColliding = None
        self.npCollidingTotal = None

    def printParticles(self):
        print("Number of particles", self.N)
        print("current time of the simulation", self.Tstart)
        print("time at which simulation ends", self.TEnd)
        print("timestep", self.dt)
        print("star radius", self.R)
        print("smoothing length", self.h)
        print("equation of state constant", self.k)
        print("damping", self.nu)

    def generateInitConditions(self, startGenerateX, endGenerateX, startGenerateY, endGenerateY, column, row):

        """
        pos    is a matrix of (x,y) positions
        vel    is a matrix of (x,y) velocity
        """
        self.N = column * row
        X = np.linspace(startGenerateX, endGenerateX, column).reshape(column,1)
        Y = np.linspace(startGenerateY, endGenerateY, row).reshape((row,1))
        posX = (X * np.ones_like(Y).T).flatten()
        posY = (Y * np.ones_like(X).T).T.flatten()

        self.pos = np.vstack((posX,posY)).T
        self.vel = np.zeros(self.pos.shape)
        self.npColliding = np.zeros_like(self.pos)

    def KernelFunction(self, x, y):

        """
        Gausssian Smoothing kernel (2D)
        x    is a vector/matrix of x positions
        y    is a vector/matrix of y positions
        z    is a vector/matrix of z positions
        h    is the smoothing length
        w    is the evaluated smoothing function
        alpha is constant
        coreZone is 0 <= q < 1/2
        sideZone is 1/2 <= q < 1
        """

        q = (np.sqrt(x ** 2 + y ** 2)) / self.h

        # save initial Numpy array shape
        indx = q.shape[0]
        indy = q.shape[1]

        # calculate alpha
        alpha = 40/(7 * np.pi * self.h)

        # flatten q and make w array which has same shape as q
        # if q is over than 1 the values of w array are 0
        q = q.flatten()
        w = np.zeros_like(q)

        # if q is smaller than 1
        sideZone = np.asarray(np.where(q <= 1)).flatten()
        w[sideZone] = alpha * 2 * (1-q[sideZone])**3

        # if q is smaller than 1/2
        coreZone = np.asarray(np.where(q < 1/2)).flatten()
        w[coreZone] = alpha * (6 * (q[coreZone]**3 - q[coreZone]**2)+1)

        # reshape to make shape as origin q array
        w = w.reshape(indx, indy)
        self.w = w

        return w

    def gradKernelFunction(self, x, y):
        """
        Gradient of the Gausssian Smoothing kernel (3D)
        x    is a vector/matrix of x positions
        y    is a vector/matrix of y positions
        z    is a vector/matrix of z positions
        h    is the smoothing length
        wx, wy, wz    is the evaluated gradient
        """

        q = (np.sqrt(x ** 2 + y ** 2)) / self.h

        # save initial Numpy array shape
        indx = q.shape[0]
        indy = q.shape[1]

        # calculate alpha
        alpha = 40 / (7 * np.pi * self.h)

        # flatten q and make w array which has same shape as q
        # if q is over than 1 the values of w array are 0
        q = q.flatten()
        w = np.zeros_like(q)

        # if q is smaller than 1
        sideZone = np.asarray(np.where(q <= 1)).flatten()
        w[sideZone] = alpha * -6 * (1 - q[sideZone]) ** 2

        # if q is smaller than 1/2
        coreZone = np.asarray(np.where(q < 1 / 2)).flatten()
        w[coreZone] = alpha * (18 * q[coreZone]**2 - 12 * q[coreZone])

        # reshape to make shape as origin q array
        w = w.reshape(indx, indy)
        wx = w * x
        wy = w * y

        return wx, wy

    def getPairwiseSeparations(self, ri, rj):
        """
        Get pairwise desprations between 2 sets of coordinates
        ri   is an M x 2 matrix of positions
        rj   is an N x 2 matrix of positions
        dx, dy, dz   are M x N matrices of separations
        """

        M = ri.shape[0]
        N = rj.shape[0]

        # positions ri = (x,y,z)
        rix = ri[:, 0].reshape((M, 1))
        riy = ri[:, 1].reshape((M, 1))

        # other set of points positions rj = (x,y,z)
        rjx = rj[:, 0].reshape((N, 1))
        rjy = rj[:, 1].reshape((N, 1))

        # matrices that store all pairwise particle separations: r_i - r_j
        dx = rix - rjx.T
        dy = riy - rjy.T

        return dx, dy

    def getDensity(self, r, pos):
        """
        Get Density at sampling loctions from SPH particle distribution
        r    is an M x 2 matrix of sampling locations
        pos  is an N x 2 matrix of SPH particle positions
        m    is the particle mass
        h    is the smoothing length
        rho  is M x 1 vector of accelerations
        """

        M = r.shape[0]

        dx, dy = self.getPairwiseSeparations(r, pos)

        rho = np.sum(self.m * self.KernelFunction(dx, dy), 1).reshape((M, 1))

        return rho

    def getPressure(self, rho):
        """
        k is user-defined stiffness constant that scales pressure, pressure gradient, and the resulting pressure force
        position based fluid PBF [Macklin 2013]
        """

        P = self.k*((rho / self.rho0)-1)

        return P
    def getViscosityAcc(self):

        """
        dx, dy       are matrix of distances differences
        nu           is  kinematic viscosity
        aVisX, aVisY  are matrix of acceleration of viscosity
        """

        dx, dy = self.getPairwiseSeparations(self.pos, self.pos)
        vx, vy = self.getPairwiseSeparations(self.vel, self.vel)
        nu = self.nu
        dWx, dWy = self.gradKernelFunction(dx, dy)

        aVisX = 2 * nu * (self.m / self.rho) * vx * (dWx / (dx * dx + 0.01 * self.h))
        aVisY = 2 * nu * (self.m / self.rho) * vx * (dWy / (dy * dy + 0.01 * self.h))

        aVisX = np.sum(aVisX, 1).reshape(self.N, 1)
        aVisY = np.sum(aVisY, 1).reshape(self.N, 1)

        return aVisX, aVisY


    def getAcc(self):
        """
        Calculate the acceleration on each SPH particle
        pos   is an N x 2 matrix of positions
        vel   is an N x 2 matrix of velocities
        m     is the particle mass
        h     is the smoothing length
        k     equation of state constant
        n     polytropic index
        nu    viscosity
        a     is N x 2 matrix of accelerations
        """

        # Calculate densities at the position of the particles
        self.rho = self.getDensity(self.pos, self.pos)

        # Get the pressures
        self.P = self.getPressure(self.rho)

        # Get pairwise distances and gradients
        dx, dy = self.getPairwiseSeparations(self.pos, self.pos)
        dWx, dWy = self.gradKernelFunction(dx, dy)

        # Add Pressure contribution to accelerations
        ax = - (np.sum(self.m * (self.P / self.rho ** 2 + self.P.T / self.rho.T ** 2) * dWx, 1).reshape((self.N, 1)))
        ay = - (np.sum(self.m * (self.P / self.rho ** 2 + self.P.T / self.rho.T ** 2) * dWy, 1).reshape((self.N, 1)))

        # Add Gravity
        ay -= 9.8

        # Add viscosity
        aVisX, aVisY = self.getViscosityAcc()

        ax += aVisX
        ay += aVisY

        # pack together the acceleration components
        acc = np.hstack((ax, ay,))

        return acc

    def outMesh(self):

        """
        x               is a vector of x position
        y               is a vector of y position
        outPos           is a vector of index out of position
        N               is the number of total particles
        pas              is a matrix positions of total particles
        vel              is a matrix velocities of total particles
        acc              is a matrix accelerations of total particles
        """
        x = self.pos[:, 0]
        y = self.pos[:, 1]

        # looking for index which is out of positions
        outPosX = np.unique(np.append(np.asarray(np.where(self.Xstart > x)), np.asarray(np.where(self.Xend < x))))
        outPosY = np.unique(np.append(np.asarray(np.where(self.Ystart > y)), np.asarray(np.where(self.Yend < y))))
        outPos = np.unique(np.append(outPosX, outPosY))

        # delete particles out of positions
        self.N -= outPos.shape[0]
        self.vel = np.vstack((np.delete(self.vel[:, 0], outPos), np.delete(self.vel[:, 1], outPos))).T
        self.pos = np.vstack((np.delete(self.pos[:, 0], outPos), np.delete(self.pos[:, 1], outPos))).T
        self.acc = np.vstack((np.delete(self.acc[:, 0], outPos), np.delete(self.acc[:, 1], outPos))).T

    def addParticles(self, newParticles):

        """
        newParticles    is the number of particles that we add to the simulation basket
        newPos           is a matrix positions of new particles
        newVel           is a matrix velocities of new particles
        N               is the number of total particles
        pas              is a matrix positions of total particles
        vel              is a matrix velocities of total particles
        acc              is a matrix accelerations of total particles
        """

        # generate new particles
        # np.random.seed(14)  # set the random number generator seed
        newPos = np.random.rand(newParticles, 2)  # randomly selected positions and velocities
        newPos = 1 * (newPos - 0.5)
        newVel = np.zeros(newPos.shape)

        # add new particles to matrix
        self.N += newParticles
        self.pos = np.append(self.pos, newPos, axis=0)
        self.vel = np.append(self.vel, newVel, axis=0)
        self.acc = self.getAcc()

    def addRollerEffect(self, Roller1, Roller2):

        """
        Roller1         is a object of the Roller class
        Roller2         is a object of the Roller class
        CenterX         is the x position of the Roller object
        CenterY         is the y position of the Roller object
        radius          is radius of the Roller object
        AngularVelocity is angular velocity of the Roller object
        """

        # set the vector of each position
        x = self.pos[:, 0]
        y = self.pos[:, 1]

        # get the distance between the position of each particles
        D1 = np.sqrt((x - Roller1.CenterX) ** 2 + (y - Roller1.CenterY) ** 2)
        D2 = np.sqrt((x - Roller2.CenterX) ** 2 + (y - Roller2.CenterY) ** 2)

        # find the indexes which is on the each rollers
        onRoller1 = np.asarray(np.where(D1 <= Roller1.Radius)).flatten()
        onRoller2 = np.asarray(np.where(D2 <= Roller2.Radius)).flatten()

        # save colliding particles
        self.npColliding = np.zeros_like(self.npColliding)
        self.npColliding[onRoller1] = 1
        self.npColliding[onRoller2] = 1


        # change the velocities vector of the particles on the rollers.
        velx = self.vel[:, 0]
        vely = self.vel[:, 1]

        velx[onRoller1] = (y - Roller1.CenterY)[onRoller1] * -Roller1.AngularVelocity
        vely[onRoller1] = (Roller1.CenterX - x)[onRoller1] * -Roller1.AngularVelocity
        velx[onRoller2] = (y - Roller2.CenterY)[onRoller2] * -Roller2.AngularVelocity
        vely[onRoller2] = (Roller2.CenterX - x)[onRoller2] * -Roller2.AngularVelocity


        velx = velx.reshape(-1, 1)
        vely = vely.reshape(-1, 1)

        self.vel = np.hstack((velx, vely))

    def saveData(self, timestep):

        npW = np.expand_dims(self.w,axis=0)
        npX = np.expand_dims(self.pos[:, 0], axis=0)
        npY = np.expand_dims(self.pos[:, 1], axis=0)
        npVelx = np.expand_dims(self.vel[:, 0], axis=0)
        npVely = np.expand_dims(self.vel[:, 1], axis=0)
        npAccx = np.expand_dims(self.acc[:, 0], axis=0)
        npAccy = np.expand_dims(self.acc[:, 1], axis=0)
        nprho = np.expand_dims(self.rho[:,0], axis=0)
        npP = np.expand_dims(self.P[:,0], axis=0)
        npColliding = np.expand_dims(self.npColliding[:,0],axis=0)

        if timestep == 0:
            self.npWtarget = npW
            self.npXtarget = npX
            self.npYtarget = npY
            self.npUtarget = npVelx
            self.npVtarget = npVely
            self.npAccXtarget = npAccx
            self.npAccYtarget = npAccy
            self.npRhotareget = nprho
            self.npPtarget = npP
            self.npCollidingTotal = npColliding
            return
        self.npWtarget = np.append(self.npWtarget, npW, axis=0)
        self.npXtarget = np.append(self.npXtarget,npX,axis=0)
        self.npYtarget = np.append(self.npYtarget,npY,axis=0)
        self.npUtarget = np.append(self.npUtarget,npVelx,axis=0)
        self.npVtarget = np.append(self.npVtarget,npVely,axis=0)
        self.npAccXtarget = np.append(self.npAccXtarget,npAccx,axis=0)
        self.npAccYtarget = np.append(self.npAccYtarget,npAccy,axis=0)
        self.npRhotareget = np.append(self.npRhotareget,nprho,axis=0)
        self.npPtarget = np.append(self.npPtarget,npP,axis=0)
        self.npCollidingTotal = np.append(self.npCollidingTotal,npColliding,axis=0)


    def simulateParticles(self, Roller1, Roller2):

        plotRealTime = True  # switch on for plotting as the simulation goes along

        # prep figure
        fig = plt.figure(figsize=(10, 10), dpi=80)
        ax = plt.subplot()

        # Simulation Main Loop
        for timestep in range(self.TimeStep):
```

```
        # kick
        self.vel += self.acc * self.dt
        self.addRollerEffect(Roller1, Roller2)  # particles are not influenced on the roller area.

        # drift
        self.pos += self.vel * self.dt

        # update accelerations
        self.getAcc()

        # get density for plottiny
        rho = self.getDensity(self.pos, self.pos)

        # save dataset
        self.saveData(timestep)

        # plot in real time (only use when  you want to plot the simulation)
#     if plotRealTime:
#         plt.sca(ax)
#         plt.cla()
#         cval = np.maximum(rho , 0).flatten()
#         plt.scatter(self.pos[:, 0], self.pos[:, 1], c=cval, s=10, alpha=0.5)
#         ax.set(xlim=(-0.10, 0.10), ylim=(-0.20, 0.05))
#         ax.set_aspect('equal', 'box')
#         ax.set_xticks([-0.10, 0, 0.10])
#         ax.set_yticks([-0.20, 0, 0.05])
#         ax.set_facecolor('black')
#         ax.set_facecolor((.1, .1, .1))
#
#         plt.pause(0.001)
#
# # Save figure
# # plt.savefig(f'{saveAddress}/sph.png', dpi=240)
    plt.close(fig)
# plt.show()


def main():
    return 0


if __name__ == "__main__":
    main()
```

1.3 Mesh.py

```
import numpy as np
import pandas as pd


class Mesh:
    # ny: The number of y-mesh
    # nx: The number of x-mesh
    # start_x: The start point of x-mesh
    # end_x: The end point of y-mesh
    # start_y: The start point of y-mesh
    # end_y: The end point of y-mesh

    def __init__(self, start_x, end_x, start_y, end_y ,nx =100, ny =100):

        self.start_x = start_x
        self.end_x = end_x
        self.start_y = start_y
        self.end_y = end_y
        self.nx = nx
        self.ny = ny

    def printMesh(self):

        print(f'The start point of x-mesh: {self.start_x}')
        print(f'The end point of x-mesh: {self.end_x}')
        print(f'The start point of y-mesh: {self.start_y}')
        print(f'The end point of y-mesh: {self.end_y}')

    def saveParams(self, address):

        df = pd.DataFrame(columns=["start_x","end_x","start_y","end_y"])
        df.loc[0, "start_x"] = self.start_x
        df.loc[0, "end_x"] = self.end_x
        df.loc[0, "start_y"] = self.start_y
        df.loc[0, "end_y"] = self.end_y

        df.to_csv(f"{address}/MeshParams.csv")
```

1.4 Roller.py

```
import pandas as pd
```

```python
class Roller:

    def __init__(self, CenterX, CenterY, Radius, AngularVelocity):
        self.CenterX = CenterX
        self.CenterY = CenterY
        self.Radius = Radius
        self.AngularVelocity = AngularVelocity

    def printRoller(self):
        print("CenterX", self.CenterX)
        print("CenterY", self.CenterY)
        print("Radius", self.Radius)
        print("AngularVelocity", self.AngularVelocity)

    def saveParams(self, address, RollerNumber):
        df = pd.DataFrame(columns=["CenterX", "CenterY", "Radius", "AngularVelocity"])
        df.loc[0, "CenterX"] = self.CenterX
        df.loc[0, "CenterY"] = self.CenterY
        df.loc[0, "Radius"] = self.Radius
        df.loc[0, "AngularVelocity"] = self.AngularVelocity

        df.to_csv(f"{address}/RollerParams{RollerNumber}.csv")
```

1.5 Time.py

```python
   import pandas as pd


class Time:
    def __init__(self, start_time, end_time, time_step):
        self.Tstart = start_time
        self.Tend = end_time
        self.TimeStep = time_step
        self.dt = (end_time - start_time) / time_step

    def printTime(self):
        print(f'start time = {self.Tstart}')
        print(f'end time = {self.Tend}')
        print(f'time step = {self.TimeStep}')
        print(f'dt = {self.dt}')

    def saveParams(self, address):
        df = pd.DataFrame(columns=["Tstart", "Tend", "TimeStep"])
        df.loc[0, "Tstart"] = self.Tstart
        df.loc[0, "Tend"] = self.Tend
        df.loc[0, "TimeStep"] = self.TimeStep

        df.to_csv(f"{address}/TimeParams.csv")
```

1.7 FluidProperties.py

```python
import numpy as np
import pandas as pd

class FluidProperties:

    def __init__(self, mass, SmoothingLength, LiquidViscosity, LiquidDropletRadius, restDensity, stiffnessConstant):

        self.m = mass
        self.h = SmoothingLength
        self.nu = LiquidViscosity
        self.rho0 = restDensity
        self.R = LiquidDropletRadius
        self.k = stiffnessConstant

    def print_liquid_property(self):

        print(f'Liquid viscosity = {self.nu}')

    def saveParams(self, address):
        df = pd.DataFrame(columns=["mass", "SmoothingLength",
"LiquidViscosity","PolytropicIndex","LiquidDropletRadius","StateConstant"])
        df.loc[0, "mass"] = self.m
        df.loc[0, "SmoothingLength"] = self.h
        df.loc[0, "LiquidViscosity"] = self.nu
        df.loc[0, "restDensity"] = self.rho0
        df.loc[0, "LiquidDropletRadius"] = self.R
        df.loc[0, "stiffnessConstant"] = self.k
        df.to_csv(f"{address}/FluidPropertiesParams.csv")
```

1.8 Simulation data visualization

```
   #!/usr/bin/env python
# coding: utf-8

# In[1]:


import pandas as pd
import numpy as np
import tensorflow as tf
import os
import matplotlib.pyplot as plt
from tensorflow import keras
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler


# In[2]:


npInputDataset = np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npInputDataset.npy")
npXTargetDataset= np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npXTargetDataset.npy")
npYTargetDataset= np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npYTargetDataset.npy")
npUTargetDataset= np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npUTargetDataset.npy")
npVTargetDataset= np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npVTargetDataset.npy")
npAccXTargetDataset = np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npXTargetDataset.npy")
npAccYTargetDataset= np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npAccYTargetDataset.npy")
npRhoTargetDataset = np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npRhoTargetDataset.npy")
npPTargetDataset = np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npPTargetDataset.npy")
npCollidingTotal = np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npCollidingTotal.npy")


# In[3]:


npXTargetDataset = np.expand_dims(npXTargetDataset, axis=-1)
npYTargetDataset= np.expand_dims(npYTargetDataset, axis=-1)
npUTargetDataset= np.expand_dims(npUTargetDataset, axis=-1)
npVTargetDataset= np.expand_dims(npVTargetDataset, axis=-1)
npAccXTargetDataset = np.expand_dims(npAccXTargetDataset, axis=-1)
npAccYTargetDataset= np.expand_dims(npAccYTargetDataset, axis=-1)
npRhoTargetDataset =np.expand_dims(npRhoTargetDataset, axis=-1)
npPTargetDataset = np.expand_dims(npPTargetDataset, axis=-1)
npCollidingTotal = np.expand_dims(npCollidingTotal,axis=-1)


# # Data overview

# In[4]:


# The number of data
NData = 100

# The number of particles
NParticles = 100

# The number of TimeSteps
NTimeSteps = 200

# The number of Targets
NTargets = 8


# In[5]:


class SPHData:

   def __init__(self,TargetName,Unit,NData,NParticles,NTimeSteps, TargetDataset):


        self.TargetName = TargetName
        self.Unit = Unit
        self.NData = NData
        self.NParticles = NParticles
        self.NTimeSteps = NTimeSteps
        self.TargetDataset = TargetDataset

   def meanDataset(self, NData):

        return  np.sum(self.TargetDataset, axis=-4)/NData

   def variDataset(self, NData):

        return np.sum((self.TargetDataset - self.meanDataset(NData))**2,axis=-4)/NData
```

```
    def stdDataset(self, Ndata):

        return np.sqrt(self.variDataset(Ndata))


    def rollerEffectperData(self, AngularVelocity):

        perData = np.sum(np.sum(self.TargetDataset, axis=-2),axis=-2)/(self.NParticles * self.NTimeSteps)
        pdData = pd.DataFrame(AngularVelocity)
        pdData["1"] = perData
        pdData.sort_values(by=0, inplace=True)

        return pdData.to_numpy()





# In[6]:


Model1 = SPHData(TargetName="The x-coordinates of the particles",
              Unit="[m]",
              NData= NData,
              NParticles= NParticles,
              NTimeSteps=NTimeSteps,
              TargetDataset=npXTargetDataset
              )
Model2 = SPHData(TargetName="The y-coordinates of the particles",
              Unit="[m]",
              NData= NData,
              NParticles= NParticles,
              NTimeSteps=NTimeSteps,
              TargetDataset=npYTargetDataset
              )
Model3 = SPHData(TargetName="The x-direction velocity of the particles",
              Unit="[m/s]",
              NData= NData,
              NParticles= NParticles,
              NTimeSteps=NTimeSteps,
              TargetDataset=npUTargetDataset
              )
Model4 = SPHData(TargetName="The y-direction velocity of the particles",
              Unit="[m/s]",
              NData= NData,
              NParticles= NParticles,
              NTimeSteps=NTimeSteps,
              TargetDataset=npVTargetDataset

              )
Model5 = SPHData(TargetName="The x-direction acceleration of the particles",
              Unit="[m/s^2]",
              NData= NData,
              NParticles= NParticles,
              NTimeSteps=NTimeSteps,
              TargetDataset=npAccXTargetDataset
              )
Model6 = SPHData(TargetName="The y-direction acceleration of the particles",
              Unit="[m/s^2]",
              NData= NData,
              NParticles= NParticles,
              NTimeSteps=NTimeSteps,
              TargetDataset=npAccYTargetDataset
              )
Model7 = SPHData(TargetName="The density of the particles",
              Unit="[kg/m^3]",
              NData= NData,
              NParticles= NParticles,
              NTimeSteps=NTimeSteps,
              TargetDataset=npRhoTargetDataset
              )
Model8 = SPHData(TargetName="The pressure of the particles",
              Unit="[Pa]",
              NData= NData,
              NParticles= NParticles,
              NTimeSteps=NTimeSteps,
              TargetDataset=npPTargetDataset
              )


# In[7]:


# Color List
ColorList = ["b","g","r","c","m","y","lime","navy"]

# ML model list at the optimal point
Model = [Model1,Model2,Model3,Model4,Model5, Model6,Model7,Model8]


# # Relation between rotating speed and Data

# In[8]:


rnAngularVelocity1= np.load(f"/Users/maxan/Documents/Master'sThesis//DATA/npRotatingVL.npy")
rnAngularVelocity2= np.load(f"/Users/maxan/Documents/Master'sThesis//DATA/npRotatingVR.npy")
```

```
# In[9]:


meanAngularVelocity = (rnAngularVelocity1 + rnAngularVelocity2)/2


# In[10]:


for i in range(NTargets):

    iModel = Model[i]
    rollerEffect = iModel.rollerEffectperData(meanAngularVelocity)


    plt.figure(i)
    plt.plot(rollerEffect[:,0], rollerEffect[:,1], label=iModel.TargetName, color=ColorList[i])
    plt.xlabel('rotating speed[m/s]', fontsize=30)
    plt.xticks(fontsize=20)
    plt.yticks(fontsize=20)
    plt.title(f" Relation between rotating speed and {iModel.TargetName}", fontsize=20)
    plt.ylabel(f"{iModel.TargetName} {iModel.Unit}", fontsize=20)
    plt.rcParams['figure.figsize'] = [10, 10]
    plt.legend(prop={'size': 15})
    plt.show()


# # Mean values

# ### per particle

# In[11]:


for i in range(NTargets):

    iModel = Model[i]
    meanData = iModel.meanDataset(NData)
    meanData= np.sum(meanData, axis=-3)/(NTimeSteps)

    plt.figure(i)
    plt.bar(np.arange(NParticles), meanData.flatten(), label=iModel.TargetName, color=ColorList[i])
    plt.xlabel('particle[-]', fontsize=20)
    plt.xticks(fontsize=20)
    plt.yticks(fontsize=20)
    plt.title("The mean value ", fontsize=20)
    plt.ylabel("The mean value.", fontsize=20)
    plt.rcParams['figure.figsize'] = [10, 10]
    plt.legend(prop={'size': 15})
    plt.show()


# ### per time-step

# In[12]:


for i in range(NTargets):

    iModel = Model[i]
    meanData = iModel.meanDataset(NData)
    meanData= np.sum(meanData, axis=-2)/(NTimeSteps)

    plt.figure(i)
    plt.bar(np.arange(NTimeSteps), meanData.flatten(), label=iModel.TargetName, color=ColorList[i])
    plt.xlabel('particle[-]', fontsize=20)
    plt.xticks(fontsize=20)
    plt.yticks(fontsize=20)
    plt.title("The mean value", fontsize=20)
    plt.ylabel("The mean value", fontsize=20)
    plt.rcParams['figure.figsize'] = [10, 10]
    plt.legend(prop={'size': 15})
    plt.show()


# # the sum of mean Standard deviation

# ### Standard deviation per perticle

# In[13]:


for i in range(NTargets):

    iModel = Model[i]
    stdData = iModel.stdDataset(NData)
    stdperParticle = np.sum(stdData, axis=-3)/(NTimeSteps)

    plt.figure(i)
    plt.bar(np.arange(NParticles), stdperParticle.flatten(), label=iModel.TargetName, color=ColorList[i])
    plt.xlabel('particle[-]', fontsize=20)
    plt.xticks(fontsize=20)
    plt.yticks(fontsize=20)
    plt.title("The mean of the standard deviations by particle.", fontsize=20)
    plt.ylabel("The mean of the standard deviations by particle.", fontsize=20)
    plt.rcParams['figure.figsize'] = [10, 10]
    plt.legend(prop={'size': 15})
    plt.show()
```

```
# ### Standard deviation per timestep

# In[14]:


for i in range(NTargets):

    iModel = Model[i]
    stdData = iModel.stdDataset(NData)
    stdperTimeSteps = np.sum(stdData, axis=-2)/NParticles

    plt.figure(i)
    plt.bar(np.arange(NTimeSteps),stdperTimeSteps.flatten() , label=iModel.TargetName, color=ColorList[i])
    plt.xlabel('Timestep[-]', fontsize=20)
    plt.xticks(fontsize=20)
    plt.yticks(fontsize=20)
    plt.title("The sum of the standard deviations per time step.",fontsize=20)
    plt.ylabel("The sum of the standard deviations per time step.",fontsize=20)
    plt.rcParams['figure.figsize'] = [10, 10]
    plt.legend(prop={'size': 15})
    plt.show()


# # The number of times that the "addRollerEffect" alogrithm was excuted

# ### per Particle

# In[15]:


CollidingPerParticle = npCollidingTotal
CollidingPerParticle = np.sum(CollidingPerParticle,axis=-4)/NData
CollidingPerParticle= np.sum(CollidingPerParticle,axis=-3)


# In[21]:


plt.figure(1)
plt.bar(np.arange(100),CollidingPerParticle.flatten(),label = "The mean number of times the \"addRollerEffect\" has been
executed per particle.[-]", color = 'b')
plt.xlabel('Particle number', fontsize = 20)
plt.xticks(fontsize= 20)
plt.yticks(fontsize= 20)
plt.title("The mean number of times the \"addRollerEffect\" has been executed per particle.[-]",fontsize=20)
plt.ylabel('The mean number of times the \"addRollerEffect\" has been executed per particle.[-]',fontsize=20)
plt.rcParams['figure.figsize'] = [15, 15]
plt.legend(prop={'size': 15})
plt.show()


# ### per timestep

# In[17]:


CollidingPerTimeStep = npCollidingTotal
CollidingPerTimeStep = np.sum(CollidingPerTimeStep,axis=-4)/NData
CollidingPerTimeStep = np.sum(CollidingPerTimeStep,axis=-2)


# In[18]:


plt.figure(1)
plt.bar(np.arange(200),CollidingPerTimeStep.flatten(),label = "The mean number of times the \"addRollerEffect\" has been
executed per time step.[-]", color = 'b')
plt.xlabel('time step', fontsize = 20)
plt.xticks(fontsize= 20)
plt.yticks(fontsize= 20)
plt.title("The mean number of times the \"addRollerEffect\" has been executed per time step.[-]",fontsize=20)
plt.ylabel('The mean number of times the \"addRollerEffect\" has been executed per time step.[-]',fontsize=20)
plt.rcParams['figure.figsize'] = [15, 15]
plt.legend(prop={'size': 15})
plt.show()


# # The initial position of particle no.40 to particle no.60

# In[19]:


X = np.linspace(-0.02, 0.02, 10).reshape(10,1)
Y = np.linspace(-0.02, 0.02, 10).reshape(10,1)
posX = (X * np.ones_like(Y).T).flatten()[40:60]
posY = (Y * np.ones_like(X).T).T.flatten()[40:60]
pos = np.vstack((posX,posY)).T


# In[20]:


fig = plt.figure(figsize=(10, 10), dpi=80)
ax = plt.subplot()
plt.sca(ax)
plt.cla()
cval = np.maximum(30 , 0).flatten()
plt.scatter(pos[:,0], pos[:,1], s=10,c="navy", alpha=0.5)
ax.set(xlim=(-0.10, 0.10), ylim=(-0.20, 0.05))
```

```
plt.xlabel("x[m]", fontsize = 20)
plt.ylabel("y[m]", fontsize = 20)
ax.set_aspect('equal', 'box')
ax.set_xticks([-0.10, 0, 0.10])
ax.set_yticks([-0.20, 0, 0.05])


plt.show()
```

2.1 Machine Learning

2.1 Machine Learning Models

```
   #!/usr/bin/env python
# coding: utf-8

# In[1]:


import pandas as pd
import numpy as np
import tensorflow as tf
import os
import matplotlib.pyplot as plt
from tensorflow import keras
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler


# In[2]:


npInputDataset = np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npInputDataset.npy")
npXTargetDataset= np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npXTargetDataset.npy")
npYTargetDataset= np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npYTargetDataset.npy")
npUTargetDataset= np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npUTargetDataset.npy")
npVTargetDataset= np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npVTargetDataset.npy")
npAccXTargetDataset = np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npAccXTargetDataset.npy")
npAccYTargetDataset= np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npAccYTargetDataset.npy")
npRhoTargetDataset = np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npRhoTargetDataset.npy")
npPTargetDataset = np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npPTargetDataset.npy")
npCollidingTotal = np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npCollidingTotal.npy")


# In[3]:


npXTargetDataset = np.expand_dims(npXTargetDataset, axis=-1)
npYTargetDataset= np.expand_dims(npYTargetDataset, axis=-1)
npUTargetDataset= np.expand_dims(npUTargetDataset, axis=-1)
npVTargetDataset= np.expand_dims(npVTargetDataset, axis=-1)
npAccXTargetDataset = np.expand_dims(npAccXTargetDataset, axis=-1)
npAccYTargetDataset= np.expand_dims(npAccYTargetDataset, axis=-1)
npRhoTargetDataset =np.expand_dims(npRhoTargetDataset, axis=-1)
npPTargetDataset = np.expand_dims(npPTargetDataset, axis=-1)
npCollidingTotal = np.expand_dims(npCollidingTotal,axis=-1)


# In[4]:


np.random.RandomState(42)


# In[5]:


# The number of data
NData = 100

# The number of particles
NParticles = 100

# The number of TimeSteps
NTimeSteps = 200

# The number of parameters
NParameters = 4

# The number of Targets
NTargets = 8


# # Machine Learning

# In[6]:


class Neuron:
```

```
def __init__(self, targetTitle):

    self.targetTitle = targetTitle
    self.w = None
    self.b = None
    self.losses = np.array([])
    self.val_losses = np.array([])
    self.predict_losses = np.array([])
    self.w_history = None
    self.learningRate = None
    self.timeStep = None
    self.NDatas = None
    self.NParticles = None
    self.NParameters = None
    self.NTargets = None
    self.X_train = None
    self.X_test = None
    self.X_Val = None
    self.y_train = None
    self.y_test = None
    self.y_val = None

def setup(self, inputDataset, targetDataset, randW, randb,test_size=0.25, random_state=42, learningRate=0.001):

    # Variables
    self.timeStep = inputDataset.shape[1]
    self.NDatas = inputDataset.shape[0]
    self.NParticles = inputDataset.shape[2]
    self.NParameters = inputDataset.shape[3]
    self.NTargets = targetDataset.shape[0]
    self.learningRate = learningRate

    self.w = randW
    self.b = randb


    # Dataset splitting
    self.X_train, self.X_test, self.y_train, self.y_test = train_test_split(inputDataset, targetDataset,
                                                            test_size=test_size,
                                                            random_state=random_state)



    self.X_train, self.X_Val, self.y_train, self.y_val = train_test_split(self.X_train, self.y_train,
                                                          test_size=test_size,
                                                          random_state=random_state)

    # Sklearn Input Dataset preprocessing
    # Triainset
    from sklearn.preprocessing import MinMaxScaler
    scaler =MinMaxScaler()
    NTrainData = self.X_train.shape[0]
    self.X_train = self.X_train.reshape(-1,self.NParameters)
    self.X_train = scaler.fit_transform(self.X_train)
    self.X_train = self.X_train.reshape(NTrainData,self.timeStep,self.NParticles,self.NParameters)
    # Testset
    NTestData = self.X_test.shape[0]
    self.X_test = self.X_test.reshape(-1, self.NParameters)
    self.X_test = scaler.transform(self.X_test)
    self.X_test = self.X_test.reshape(NTestData,self.timeStep,self.NParticles,self.NParameters)


def forpass(self, x):
    y_hat = x * self.w + self.b
    return y_hat

def backprop(self, x, err):
    w_grad = x * err
    b_grad = 1 * err
    return w_grad, b_grad

def update_val_loss(self):


    y_hat = self.forpass(self.X_Val)
    y_hat = np.sum(y_hat, axis=-1)
    y_hat = np.expand_dims(y_hat, axis=-1)
    err = -(self.y_val - y_hat)
    self.val_losses = np.append(self.val_losses, np.sum(err))


def prediction(self):

    NTest = self.X_test.shape[0]
    y_hat = self.forpass(self.X_test)
    y_hat = np.sum(y_hat, axis=-1)
    y_hat = np.expand_dims(y_hat, axis=-1)

    self.predict_losses = -(self.y_test - y_hat)

def saveResults(self,address,modelName):

    np.save(f"{address}/{modelName}weights.npy",self.w)
    np.save(f"{address}/{modelName}bias.npy",self.b)
    np.save(f"{address}/{modelName}losses.npy",self.losses)
    np.save(f"{address}/{modelName}val_losses.npy",self.val_losses)
    np.save(f"{address}/{modelName}predict_losses.npy",self.predict_losses)
    np.save(f"{address}/{modelName}X_train.npy",self.X_train)
    np.save(f"{address}/{modelName}X_test.npy",self.X_test)
    np.save(f"{address}/{modelName}X_Val.npy",self.X_Val)
```

```
        np.save(f"{address}/{modelName}y_train.npy",self.y_train)
        np.save(f"{address}/{modelName}y_test.npy",self.y_test)
        np.save(f"{address}/{modelName}y_val.npy",self.y_val)

    def fit(self, epochs=10):

        self.w_history = self.w
        np.random.seed(42)
        indexes = np.random.permutation(np.arange(self.X_train.shape[0]))
        for _ in range(epochs):
            loss = 0
            for i in indexes:
                y_hat = self.forpass(self.X_train[i])
                y_hat = np.sum(y_hat, axis=-1)
                y_hat = np.expand_dims(y_hat, axis=-1)
                err = -(self.y_train[i] - y_hat)
                w_grad, b_grad = self.backprop(self.X_train[i], err)
                self.w -= w_grad * self.learningRate
                self.b -= b_grad * self.learningRate
                loss += np.sum(err)
            self.losses = np.append(self.losses, loss)
            self.update_val_loss()



# In[7]:


# Epochs
Epochs = 3000

# stadard deviation of random initiai weights
stdScale = 0.01


# Model 1
Model1Epochs = 3000
Model1LearningRate = 0.001

# Model 2
Model2Epochs = 3000
Model2LearningRate = 0.001

# Model 3
Model3Epochs = 3000
Model3LearningRate = 0.001

# Model 4
Model4Epochs = 3000
Model4LearningRate = 0.001

# Model 5
Model5Epochs = 3000
Model5LearningRate = 0.001

# Model 6
Model6Epochs = 3000
Model6LearningRate = 0.001

# Model 7
Model7Epochs = 3000
Model7LearningRate = 0.001

# Model 8
Model8Epochs = 3000
Model8LearningRate = 0.001


# In[20]:


Model1 = Neuron(targetTitle="The x-coordinates of the particles[m]")
Model1.setup(npInputDataset,npXTargetDataset,
           randW = np.random.normal(scale= stdScale ,size=(NTimeSteps,NParticles,NParameters)),
           randb = np.zeros(shape=(NTimeSteps,NParticles,NParameters)),
           learningRate=Model1LearningRate)

Model1.fit(epochs=Model1Epochs)


# In[ ]:


Model2 = Neuron(targetTitle="The y-coordinates of the particles[m]")
Model2.setup(npInputDataset,npYTargetDataset,
           randW = np.random.normal(scale=  stdScale ,size=(NTimeSteps,NParticles,NParameters)),
           randb = np.zeros(shape=(NTimeSteps,NParticles,NParameters)),
           learningRate=Model2LearningRate)

Model2.fit(epochs=Model2Epochs)


# In[10]:


Model3 = Neuron(targetTitle="The x-direction velocity of the particles[m/s]")
Model3.setup(npInputDataset,npUTargetDataset,
          randW = np.random.normal(scale=  stdScale ,size=(NTimeSteps,NParticles,NParameters)),
           randb = np.zeros(shape=(NTimeSteps,NParticles,NParameters)),
           learningRate=Model3LearningRate)
```

```
Model3.fit(epochs=Model3Epochs)


# In[11]:


Model4 = Neuron(targetTitle="The y-direction velocity of the particles[m/s]")
Model4.setup(npInputDataset,npVTargetDataset,
            randW = np.random.normal(scale= stdScale ,size=(NTimeSteps,NParticles,NParameters)),
            randb = np.zeros(shape=(NTimeSteps,NParticles,NParameters)),
            learningRate=Model4LearningRate)
Model4.fit(epochs=Model4Epochs)


# In[12]:


Model5 = Neuron(targetTitle="The x-direction accelerlation of the particles[m/s^2]")
Model5.setup(npInputDataset,npAccXTargetDataset,
            randW = np.random.normal(scale= stdScale ,size=(NTimeSteps,NParticles,NParameters)),
            randb = np.zeros(shape=(NTimeSteps,NParticles,NParameters)),
            learningRate=Model5LearningRate)

Model5.fit(epochs=Model5Epochs)


# In[13]:


Model6 = Neuron(targetTitle="The y-direction accelerlation of the particles[m/s^2]")
Model6.setup(npInputDataset,npAccYTargetDataset,
             randW = np.random.normal(scale= stdScale ,size=(NTimeSteps,NParticles,NParameters)),
            randb = np.zeros(shape=(NTimeSteps,NParticles,NParameters)),
            learningRate=Model6LearningRate)
Model6.fit(epochs=Model6Epochs)


# In[14]:


Model7 = Neuron(targetTitle="The density of particles[kg/m^3]")
Model7.setup(npInputDataset,npRhoTargetDataset,
             randW = np.random.normal(scale= stdScale ,size=(NTimeSteps,NParticles,NParameters)),
            randb = np.zeros(shape=(NTimeSteps,NParticles,NParameters)),
            learningRate=Model7LearningRate)

Model7.fit(epochs=Model7Epochs)


# In[15]:


Model8 = Neuron(targetTitle="The pressure of particles[Pa]")
Model8.setup(npInputDataset,npPTargetDataset,
              randW = np.random.normal(scale= stdScale ,size=(NTimeSteps,NParticles,NParameters)),
            randb = np.zeros(shape=(NTimeSteps,NParticles,NParameters)),
            learningRate=Model8LearningRate)
Model8.fit(epochs=Model8Epochs)


# # Save

# #### Epochs: 3000

# In[16]:


#
SaveAddress = f"/Users/maxan/Documents/Master'sThesis/MachineLearning/ML_from_code_level/Results_3000epochs_std_set_0.01"
saveAddressValLosses = "Results_3000epochs_std_set_0.01_valLosses.xlsx"


# In[17]:


Model1.prediction()
Model2.prediction()
Model3.prediction()
Model4.prediction()
Model5.prediction()
Model6.prediction()
Model7.prediction()
Model8.prediction()


# In[18]:


Model1.saveResults(SaveAddress ,"Model1")
Model2.saveResults(SaveAddress ,"Model2")
Model3.saveResults(SaveAddress ,"Model3")
Model4.saveResults(SaveAddress ,"Model4")
Model5.saveResults(SaveAddress ,"Model5")
Model6.saveResults(SaveAddress ,"Model6")
Model7.saveResults(SaveAddress ,"Model7")
Model8.saveResults(SaveAddress ,"Model8")


# In[19]:
```

```
dfValLosses = np.abs(np.hstack([Model1.val_losses.reshape(Epochs,-1),Model2.val_losses.reshape(Epochs,-1),
                     Model3.val_losses.reshape(Epochs,-1),Model4.val_losses.reshape(Epochs,-1),
                     Model5.val_losses.reshape(Epochs,-1),Model6.val_losses.reshape(Epochs,-1),
                     Model7.val_losses.reshape(Epochs,-1),Model8.val_losses.reshape(Epochs,-1)]))
dfValLosses = pd.DataFrame(dfValLosses)
dfValLosses.to_excel(f"{SaveAddress}/{saveAddressValLosses}")
```

2.2 Machine Learning data visualization

```
   #!/usr/bin/env python
# coding: utf-8

# In[1]:


import pandas as pd
import numpy as np
import tensorflow as tf
import os
import matplotlib.pyplot as plt
from tensorflow import keras
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from mxnet import autograd, nd
import random


# In[2]:


npInputDataset = np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npInputDataset.npy")
npXTargetDataset= np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npXTargetDataset.npy")
npYTargetDataset= np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npYTargetDataset.npy")
npUTargetDataset= np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npUTargetDataset.npy")
npVTargetDataset= np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npVTargetDataset.npy")
npAccXTargetDataset = np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npXTargetDataset.npy")
npAccYTargetDataset= np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npAccYTargetDataset.npy")
npRhoTargetDataset = np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npRhoTargetDataset.npy")
npPTargetDataset = np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npPTargetDataset.npy")
npCollidingTotal = np.load(f"/Users/maxan/Documents/Master'sThesis/DATA/npCollidingTotal.npy")


# In[3]:


npXTargetDataset = np.expand_dims(npXTargetDataset, axis=-1)
npYTargetDataset= np.expand_dims(npYTargetDataset, axis=-1)
npUTargetDataset= np.expand_dims(npUTargetDataset, axis=-1)
npVTargetDataset= np.expand_dims(npVTargetDataset, axis=-1)
npAccXTargetDataset = np.expand_dims(npAccXTargetDataset, axis=-1)
npAccYTargetDataset= np.expand_dims(npAccYTargetDataset, axis=-1)
npRhoTargetDataset =np.expand_dims(npRhoTargetDataset, axis=-1)
npPTargetDataset = np.expand_dims(npPTargetDataset, axis=-1)
npCollidingTotal = np.expand_dims(npCollidingTotal,axis=-1)


# # Data Overview

# In[4]:


# The number of data
NData = 100

# The number of particles
NParticles = 100

# The number of TimeSteps
NTimeSteps = 200

# The number of Parameters
NParameters = 4

# The number of Targets
NTargets = 8

# The number of data in test dataset
Ntestdata =25


# In[5]:


class reNeuron:

   def __init__(self,TargetName,Unit,ModelNumber, address,NParticles = 100, NTimeStep = 200, NTrainData = 56, NValData = 19,
NTestData = 25):

        self.Unit = Unit
        self.TargetName = TargetName
        self.ModelNumber = ModelNumber
        self.NParticles = NParticles
        self.NTimeStep = NTimeStep
        self.weights = np.load(f"{address}/Model{ModelNumber}weights.npy")
        self.bias = np.load(f"{address}/Model{ModelNumber}bias.npy")
        self.valLosses = np.load(f"{address}/Model{ModelNumber}val_losses.npy")
        self.losses = np.load(f"{address}/Model{ModelNumber}losses.npy")
        self.predict_losses = np.load(f"{address}/Model{ModelNumber}predict_losses.npy")
        self.X_train = np.load(f"{address}/Model{ModelNumber}X_train.npy")
        self.X_test =np.load(f"{address}/Model{ModelNumber}X_test.npy")
        self.X_Val = np.load(f"{address}/Model{ModelNumber}X_Val.npy")
        self.y_train =np.load(f"{address}/Model{ModelNumber}y_train.npy")
        self.y_test = np.load(f"{address}/Model{ModelNumber}y_test.npy")
        self.y_val =np.load(f"{address}/Model{ModelNumber}y_val.npy")
        self.NTrainData = NTrainData
        self.NValData = NValData
        self.NTestData = NTestData

    def perParticle(self):

        errPerParticles = np.sum(np.abs(self.predict_losses),axis=-4)/self.NTestData
        errPerParticles = np.sum(errPerParticles,axis=-3)/self.NTimeStep

        return errPerParticles

    def perTimeStep(self):

        errPerTimeStep = np.sum(np.abs(self.predict_losses),axis=-4)/self.NTestData
        errPerTimeStep = np.sum(errPerTimeStep,axis=-2)/self.NParticles
        return errPerTimeStep

    def forpass(self, x):
        y_hat = x * self.weights + self.bias
        return y_hat


    def prediction(self, Data):

        predData = Data * self.weights + self.bias
        predData = np.sum(predData, axis=-1)
        predData = np.expand_dims(predData, axis=-1)

        return predData


# # 300 Epochs

# In[6]:


Model1 = reNeuron(TargetName= "The x-coordinates of the particles",
              Unit = "[m]",
              ModelNumber=1,

address=f"/Users/maxan/Documents/Master'sThesis/MachineLearning/ML_from_code_level/Results_3000epochs_std_set_0.01"
              )
Model2 = reNeuron(TargetName= "The y-coordinates of the particles",
              Unit = "[m]",
              ModelNumber=2,

address=f"/Users/maxan/Documents/Master'sThesis/MachineLearning/ML_from_code_level/Results_3000epochs_std_set_0.01"
              )
Model3 = reNeuron(TargetName= "The x-direction velocity of the particles",
              Unit = "[m/s]",
              ModelNumber=3,

address=f"/Users/maxan/Documents/Master'sThesis/MachineLearning/ML_from_code_level/Results_3000epochs_std_set_0.01"
              )
Model4 = reNeuron(TargetName= "The y-direction velocity of the particles",
              Unit = "[m/s]",
              ModelNumber=4,
          address=f"/Users/maxan/Documents/Master'sThesis/MachineLearning/ML_from_code_level/Results_3000epochs_std_set_0.01"
              )
Model5 = reNeuron(TargetName= "The x-direction acceleration of the particles",
              Unit = "[m/s^2]",
              ModelNumber=5,
        address=f"/Users/maxan/Documents/Master'sThesis/MachineLearning/ML_from_code_level/Results_3000epochs_std_set_0.01"
              )
Model6 = reNeuron(TargetName= "The y-direction acceleration of the particles",
              Unit = "[m/s^2]",
              ModelNumber=6,
         address=f"/Users/maxan/Documents/Master'sThesis/MachineLearning/ML_from_code_level/Results_3000epochs_std_set_0.01"
              )
Model7 = reNeuron(TargetName= "The density of the particles",
              Unit = "[kg/m^3]",
              ModelNumber=7,
        address=f"/Users/maxan/Documents/Master'sThesis/MachineLearning/ML_from_code_level/Results_3000epochs_std_set_0.01"
              )
Model8 = reNeuron(TargetName= "The pressure of the particles",
              Unit = "[Pa]",
              ModelNumber=8,
    address=f"/Users/maxan/Documents/Master'sThesis/MachineLearning/ML_from_code_level/Results_3000epochs_std_set_0.01"
              )


# In[7]:
```

```
# ML model list at 300 Epochs
Model = [Model1,Model2,Model3,Model4,Model5, Model6,Model7,Model8]


# In[19]:


for NModel in Model:
    plt.figure(NModel.ModelNumber)
    plt.plot((NModel.losses/NModel.NTrainData),label = "Training set error")
    plt.plot((NModel.valLosses/NModel.NValData), label = "Validation set error ")
    plt.xlabel('Epoch[-]', fontsize = 20)
    plt.xticks(fontsize= 20)
    plt.yticks(fontsize= 20)
    plt.title(f"{NModel.TargetName}{NModel.Unit}",fontsize=20)
    plt.ylabel(f'The mean errors{NModel.Unit}',fontsize=20)
    plt.rcParams['figure.figsize'] = [10, 10]
#     plt.ylim(0,np.abs(NModel.valLosses/NModel.NValData).max())
    plt.legend(prop={'size': 15})
    plt.show()


# # Mean Absolute Error

# In[9]:


npmeanAbsoluteError = np.array([np.sum(np.abs(Model1.predict_losses))/(Ntestdata*NTimeSteps*NParticles),
                           np.sum(np.abs(Model2.predict_losses))/(Ntestdata*NTimeSteps*NParticles),
                           np.sum(np.abs(Model3.predict_losses))/(Ntestdata*NTimeSteps*NParticles),
                           np.sum(np.abs(Model4.predict_losses))/(Ntestdata*NTimeSteps*NParticles),
                           np.sum(np.abs(Model5.predict_losses))/(Ntestdata*NTimeSteps*NParticles),
                           np.sum(np.abs(Model6.predict_losses))/(Ntestdata*NTimeSteps*NParticles),
                           np.sum(np.abs(Model7.predict_losses))/(Ntestdata*NTimeSteps*NParticles),
                           np.sum(np.abs(Model8.predict_losses))/(Ntestdata*NTimeSteps*NParticles)]

                         ).reshape(-1,NTargets)
pdAbsoluteError = pd.DataFrame(npmeanAbsoluteError)
pdAbsoluteError.columns = [Model1.TargetName,
                      Model2.TargetName,
                      Model3.TargetName,
                      Model4.TargetName,
                      Model5.TargetName,
                      Model6.TargetName,
                      Model7.TargetName,
                      Model8.TargetName]
pdAbsoluteError



# In[10]:


# ML model list at the optimal point
Model = [Model1,Model2,Model3,Model4,Model5, Model6,Model7,Model8]


# In[11]:


ColorList = ["b","g","r","c","m","y","lime","navy"]


# ### Error per particle

# In[28]:


for NModel in Model:

    errperParticles = NModel.perParticle()

    plt.figure(NModel.ModelNumber)
    plt.bar(np.arange(100),errperParticles.flatten(),label = f"Error of {NModel.TargetName}{NModel.Unit}", color
=ColorList[NModel.ModelNumber-1] )
    plt.xlabel('Particle number', fontsize = 20)
    plt.xticks(fontsize= 20)
    plt.yticks(fontsize= 20)
    plt.title(f"Target:{NModel.TargetName}{NModel.Unit} ",fontsize=10)
    plt.ylabel(f'error{NModel.Unit}',fontsize=20)
    plt.rcParams['figure.figsize'] = [10, 10]
    plt.legend(prop={'size': 15})
    plt.show()


# In[63]:


plt.bar(np.arange(100), Model5.perParticle().flatten(),label = f"Error of {Model5.TargetName}{Model5.Unit}", color
=ColorList[Model5.ModelNumber-1] )
plt.xlabel('Particle number', fontsize = 20)
plt.xticks(fontsize= 20)
plt.yticks(fontsize= 20)
plt.title(f"Target:{Model5.TargetName}{Model5.Unit} ",fontsize=10)
plt.ylabel(f'error{NModel.Unit}',fontsize=20)
plt.rcParams['figure.figsize'] = [10, 10]
plt.legend(prop={'size': 15})
plt.ylim(0.0, 1e-5)
plt.show()


# ### Error per time-step

# In[31]:


for NModel in Model:

    errperTimeStep = NModel.perTimeStep()
    plt.figure(NModel.ModelNumber)
    plt.plot(errperTimeStep,label = f"Error of {NModel.TargetName}{NModel.Unit}", color =ColorList[NModel.ModelNumber-1])
    plt.xlabel('Time-step[-]', fontsize = 20)
    plt.xticks(fontsize= 20)
    plt.yticks(fontsize= 20)
    plt.title(f"{NModel.TargetName}{NModel.Unit} ",fontsize=15)
    plt.ylabel(f'error{NModel.Unit}',fontsize=20)
    plt.rcParams['figure.figsize'] = [10, 10]
    plt.legend(prop={'size': 15})
    plt.show()


# # Comparison of machine learning models and real values

# ### Machine Learning

# In[14]:


Data = 10
TimeStep = 0


# In[15]:


# predicted Data
predX = Model1.prediction(Model1.X_test)
predY = Model2.prediction(Model2.X_test)
predRho = Model7.prediction(Model7.X_test)


# In[37]:


predRho.shape


# In[57]:


predRho.shape


# In[62]:


# visualization
for TimeStep in range(0,200,25):
    fig = plt.figure(figsize=(10, 10), dpi=80)
    ax = plt.subplot()

    Data = 24
    TimeStep = TimeStep
    plt.sca(ax)
    plt.cla()
    cval = predRho[Data][TimeStep].flatten()
    plt.scatter(predX[Data][TimeStep], predY[Data][TimeStep], c = cval,s=10, alpha=0.5)
    ax.set(xlim=(-0.10, 0.10), ylim=(-0.20, 0.05))
    ax.set_aspect('equal', 'box')
    ax.set_xticks([-0.10, 0, 0.10])
    ax.set_yticks([-0.20, 0, 0.05])
    ax.set_facecolor('black')
    plt.title(f"Time step is {TimeStep}th",fontsize= 25)
    ax.tick_params(labelsize='large')
    plt.colorbar()
    plt.xlabel('X[m]', fontsize = 25)
    plt.xticks(fontsize= 25)
    plt.yticks(fontsize= 25)
    plt.ylabel('Y[m]',fontsize=25)
    plt.clim(0.0,0.25)
    plt.show()


# ### SPH solver

# In[54]:


for TimeStep in range(0,200,25):
    fig = plt.figure(figsize=(10, 10), dpi=80)
    ax = plt.subplot()

    Data = 24
    plt.sca(ax)
    plt.cla()
    cval = np.maximum(Model7.y_test[Data][TimeStep] , 0).flatten()
    plt.scatter(Model1.y_test[Data][TimeStep], Model2.y_test[Data][TimeStep], c=cval, s=10, alpha=0.5)
    ax.set(xlim=(-0.10, 0.10), ylim=(-0.20, 0.05))
    ax.set_aspect('equal', 'box')
    ax.set_xticks([-0.10, 0, 0.10])
    ax.set_yticks([-0.20, 0, 0.05])
    ax.set_facecolor('black')
    plt.title(f"Time step is {TimeStep}th",fontsize= 25)
    ax.tick_params(labelsize='large')
    plt.colorbar()
    plt.xlabel('X[m]', fontsize = 25)
    plt.xticks(fontsize= 25)
    plt.yticks(fontsize= 25)
    plt.ylabel('Y[m]',fontsize=25)
    plt.clim(0.0,0.25)
    plt.show()
```