A machine learning approach to modelling and simulating the behaviour of systems with deformable particles

Zvonimir Lavrić* and Krešimir Osman**
* Zagreb University of Applied Sciences – Department of Mechanical Engineering, Zagreb, Croatia
** Zagreb University of Applied Sciences – Department of Electrical Engineering, Zagreb, Croatia
zvonimir.lavric@tvz.hr, kresimir.osman@tvz.hr

Abstract **- The paper presents an approach to model and simulate a discrete system with deformable particles. The observed system is represented and considered as a system with concentrated parameters (particles) connected by elastic springs and damping. The system model thus created is used for learning the graphical neural network, which we intend to use to obtain the results of the simulation with less CPU power. The mathematical model of this system was implemented by creating a computer solution using the Java programming language and the OpenGL 3D graphics standard. Using this model, the problem analysis was performed and the simulation results are presented graphically.**

Keywords - modelling, simulation, system with deformable particles, machine learning, graphic neural network

I. Introduction

By the end of the 20th century, computers became much more accessible and can be said to be an integral part of today's society. The great demand for personal computers has led to the rapid discovery of new technologies to enable more efficient and faster operation. Due to the great consumer demand for the most sophisticated computer graphics, a graphics card is being developed, a special computer hardware that speeds up the process of displaying images. About 10 years ago, they began to be used not only for displaying images, but also for solving complex mathematical problems. The major graphics card manufacturers have recognized the need for more computing power and are investing more and more in research into special graphics card cores, that speed up the performance of mathematical operations.

Recently, computer simulations have been used in almost all areas of science. In some branches of science, the word simulation is associated with the concept of a computer model [1]. Models can mimic discrete mathematical laws that describe the behaviour of natural systems and processes. Examples include the mimicking of human cells, the interaction of proteins, the design and production of cars, planes, houses and nuclear weapons.

Due to the growing need of science for complex models and the limitations of computer hardware, software solutions are proposed to speed up the simulation process [2]. Since the description of physical models in discrete form has some deviation from real models due to initial assumptions and limitations, attempts are made to create an approximate model. This model also approximates the real model while requiring significantly less power CPU. The approximate model can be based on machine learning technologies of discrete models, that can later be learned from measurements on a real model. Depending on how accurate we want the approximate model to be, we can also dose the amount of information that the approximate model processes. In this way, we increase or decrease the speed with which the observed model behaviour reacts to the prediction.

II. Presentation of Mathematical Models used for Implementation of the Simulation

The simulation of system with deformable particles is based on a realistic simulation of mechanical systems and the properties of deformable bodies. The particles in the observed system are represented as concentrated parameters [3, 4] that are linked together. For deformable bodies the distance between the particles (concentrated parameters) can be changed dynamically. The centre of mass of such a body does not lie in its centre of gravity, but depends on the relative positions of the concentrated parameters within the body.

To make the simulation of the mechanical system as realistic as possible, the concentrated parameters must represent as faithfully as possible the particles we find around us in the real world. Therefore, each concentrated parameter must have a value for specific mass, position, velocity and force [3, 4]. We assume that the mass for each concentrated parameter always has a constant value. The initial position depends on where the concentrated parameter is located within the observed system with deformable particles, while in further iterations it also depends on the velocity it possesses. The velocity in each iteration is calculated by the acceleration integral and added to the velocity from the previous iteration. The acceleration is determined with the fundamental law of motion, with the force effect being recalculated at each iteration step. For a system with deformable particles, the force can be calculated in several ways. The approach chosen is to calculate the force on a single concentrated parameter, so that a system with spring and damping is applied for each particle connection. Concentrated parameters with this type of connection are related to each neighbouring concentrated parameter.

A. Calculation of the forces in a system by a discrete method

In order to calculate the new positions of the concentrated parameters, the forces acting on each of them must first be calculated. The discrete calculation method aims to describe the system as accurately as possible and is based on a set of mathematical expressions and laws used in the relevant literature.

According to the law of energy conservation [4], in a closed mechanical system the sum of all forms of energy is constant. With this approach, we can decompose all forces acting on each of the observed concentrated parameters. For the mentioned system of spring and damping, we can write [5] that the sum of all forces is equal to the sum of the forces generated by the spring F_s, the damping F_d and the gravity F_g (1):

$$\sum F = F_s + F_d + F_g \qquad (1)$$

The force with which a spring acts on concentrated parameters can be described by Hooke's law [6]. The linear dependence of spring force F_s (2) is related to the spring constant k_s, which the user can specify in the window "Parametrizacija". It also depends on distance vector *x*.

$$F_s = k_s * x \qquad (2)$$

The connection between the two concentrated parameters is described by a spring that is free on both sides, which means that the force on the two connected particles is generated by the so-called internal forces F_{sA} and F_{sB}. The formula for such a spring [6] differs from Hooke's law, which describes a spring that is fixed on one side and free on the other ($\vec{n}$ is normal vector). The distance between two vectors is represented as the norm of the vector $||x_A - x_B||$ in order to know the direction of the force in two-dimensional space. The system with deformable particles tends to maintain the same length between its particles, which is achieved in the formula with spring force F_{sA} in point A by a constant value of the initial length of the spring L_0 (3):

$$F_{sA} = k_s * \vec{n} * (||x_A - x_B|| - L_0) \qquad (3)$$

Every real system has some form of energy loss. It can be represented in the form of resistance in the wire, air or friction. To make the system as realistic as possible, damping [4] is added in parallel with the spring (4) (k_d – damping coefficient, $\vec{v}_A, \vec{v}_B$ – velocity vectors in points A and B). Damping force F_d is calculated as

$$F_d = k_d * (\vec{v}_A - \overrightarrow{v_B}) \qquad (4)$$

Most simulations of mechanical systems apply gravity to particles [4]. To experience the deformable particle system more realistically, gravity (5) is added to the sum of all forces. Gravity force F_g is calculated as (m – particle mass, $\vec{g}$ – gravity coefficient):

$$F_g = m * \vec{g} \qquad (5)$$

Then the total sum $\sum F$ of all the above forces (equations 2 – 5) can be calculated.

Within the program, the calculation of forces is implemented with the JOML library (© Oracle), which facilitates the performance of mathematical operations with vectors. The force acting on the concentrated parameter is initially set to value 0. The spring and damping forces are then summed for each connection of the concentrated parameters. After summing these forces for each individual connection, a gravitational force acting on the concentrated parameter is added.

If we know the magnitude of the force acting on a concentrated parameter, we can determine the acceleration of the concentrated parameter by applying the fundamental law of motion [4], shown on expression (6):

$$\vec{a} = \frac{\vec{F}}{m} \qquad (6)$$

To obtain the position of the concentrated parameter, it is necessary to calculate the double integral of the acceleration vector. Euler's integration [7] is a numerical form of solving integrals with trigonometric functions.

It is the so-called asymmetric method, which means that it finds solutions for the time interval ${}_\Delta t$, but only uses the derivative information from the beginning of the interval. The first acceleration integral yields the velocity vector $\overrightarrow{v_k}$, to which the previous velocity vector of the concentrated parameter $\overrightarrow{v_{k-1}}$ is added (7). The position x_k results from the integral of the velocity vector $\overrightarrow{v_k}$, to which we add the previous value of the position of the concentrated parameter x_{k-1} (8).

$$\overrightarrow{v_k} = \overrightarrow{v_{k-1}} + \frac{\vec{F} * {}_\Delta t}{m} \qquad (7)$$

$$x_k = x_{k-1} + v_k * {}_\Delta t \qquad (8)$$

The visual impression of the simulation results from the number of images per second, which is due to the FPSAnimator [8] class. Each time a new image is generated, the positions of the concentrated parameters are calculated. From this we can conclude that the more images per second are generated, the faster the simulation of the deformable particle system is performed. The visual impression of the simulation speed can also be adjusted via the time interval ${}_\Delta t$. If the interval is longer, the simulation is carried out faster.

B. Calculation of forces in a system by an approximation method - application of machine learning approach

The system with deformable particles described in this paper can be seen in the structure of the graph [9]. The concentrated parameters represent the nodes *V*, and the relations between the concentrated parameters represent the edges $\mathcal{E}$ of the undirected graph. This approach is not only an elegant way to represent a system with deformable particles, but also provides a mathematical basis to analyse, understand and learn from real-world problems. Graphical neural networks (GNN) [10] are a special type of neural network [11] that act on a graph-like data structure. The idea is to establish a connection between nodes and all neighbouring nodes to obtain the desired

information.
Since the structure written in the form of a graph does not have the specified data set, a special method must be used to transfer the data into Euclidean space [12]. In other words, we want to represent the nodes in latent space [13], where a geometric connection represents the relationship between the nodes. The approach to this problem is solved with encoding and decoding part [9]. Framework is learning on a graph, which is divided into two steps. First, encoding positions the node $u \in V$ in the vector $Z_u \epsilon\ R^n$ of the lower dimension (*embedding*).

In most cases, a *Shallow embedding* is used (9), which takes the value Z_u from the matrix of coding pairs and nodes *V*:

$$ENC\ (v) : Z_u \tag{9}$$

The decoder then takes the thus written vector Z_u of node u ∈ V and reconstructs the neighbourhood information of node *u.* The standard procedure is to have a paired decoder $Z \epsilon R^n$ that predicts a connection between pairs of nodes (10):

$$DEC(ENC\ (v), ENC\ (u)\ ) = DEC(Z_u, Z_v) \tag{10}$$

The goal is to create a combination of encoding and decoding that best describes the desired connection between nodes. *Message passing* [12] is a complex approach with encoding and decoding. Just like the previous process, it consists of two basic blocks. *Aggregate* [12] is a first process in which the neighbours of node $N_{(v)}$ are aggregated by the selected function to form a message $m_{N_{(v)}}$. This is a permutation process, i.e. the order in which the nodes sum up in the message $m_{N_{(v)}}$ is irrelevant. An update process is then performed, combining the message ${m_{N_{(v)}}}^{(k)}$ and the previous state of the node ${h_v}^{(k)}$ to obtain a new state of the node ${h_v}^{(k+1)}$:

$${h_v}^{(k+1)} = UPDATE({h_v}^{(k)}, AGGREGATE^{(k)}(N_{(v)})) \tag{11}$$

$${h_v}^{(k+1)} = UPDATE({h_v}^{(k)}, {m_{N_{(v)}}}^{(k)}) \tag{12}$$

In this work, GNN is used as a tool to calculate the force acting on concentrated parameters. GNN can replace the entire force calculation process, but in this work, it is used exclusively to replace the calculation of the spring force acting on the concentrated parameter. The spring force calculation must apply formula (3) for each connection of the concentrated parameter. The more complex the formula for calculating the connections between concentrated parameters is and the more connections a concentrated parameter has, the more demanding the process of force calculation becomes. If we replace this process with the GNN model, the calculation of the force no longer depends on the complexity and the number of connections of the concentrated parameter, but only on the time needed to pass through the neural network. With this approach, the time taken to compute the spring force will be constant. We can conclude that as the complexity of the connections between the concentrated parameters increases, the GNN model becomes more favourable for computing the spring force.

The GNN structure [11] consists of two blocks (Figure 1). The first block is used to combine messages from neighbouring nodes into a common message $m_{N_{(v)}}$ (16). In this work, the contraction is implemented as the arithmetic mean of the state of node *Z* from each node in the neighbourhood $N_{(v)}$. The state of node *Z* and the message $m_{N_{(v)}}$ are vectors with positions in the form of two numbers representing the *x* and *y* axes of the coordinate system.

$$m_{N_{(v)}} = \frac{\sum_{n=0}^{N(v)} Z_n}{N_{(v)}} \tag{15}$$

Updating is the second block within the GNN model, which is created with a neural network. As described in formula (15), the neural network expects at the input the collected message $m_{N_{(v)}}$ and the current state of node h_v. The state of node h_v is written as the position of the node in *x* and *y* coordinates and as the message $m_{N_{(v)}}$. At the output of the neural network you get the force of the spring F_s acting on the node.

$$F_s = Neuronska\ mre\check{z}a(h_v\ , m_{N_{(v)}}\ ) \tag{13}$$

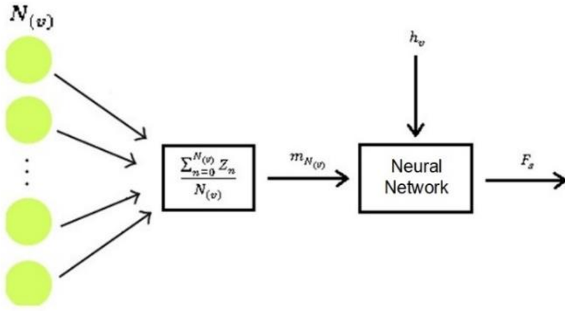


Figure 1. Structure of Graphical Neural Network (GNN) [11]

Force of the spring can be written via GNN:

$$F_s = Neuronska\ mre\check{z}a(h_v\ , \frac{\sum_{n=0}^{N(v)} Z_n}{N_{(v)}}) \tag{14}$$

Since the system with deformable particles is based on absolute truth, the neural network is learned by higher-order learning. The learning data is written in pairs of values. The input values are represented by four numbers, two of which are the arithmetic mean of the positions of neighbouring nodes and the other two numbers are the position of the node. The output values are written as two numbers representing the spring force, also written in *x* and *y* coordinates.

TensorFlow [14] is a free open source library developed for machine learning in the Python programming language (© Python Software Foundation). It is used in numerical mathematical problems that use data flow graphs so that the computational process can be performed in parallel. The Keras library [16] is an extension of the TensorFlow library that implements functions to facilitate the definition and learning of neural networks. The Java programming language (© Oracle) is not able to use the resources of the TensorFlow library, but it can implement Keras models learned from other programs. The neural network used for the system with deformable particles was learned in the Python programming language, then saved in a special format and implemented in a simulation in the Java programming language. The learned data is generated at a periodic interval within the simulation loop and written to the file "treningPodatci.txt". Within the program you can specify the default interval for data recording and the amount of data that will be written to the specified file. In an interval, a pair of input/output values is calculated and recorded for each concentrated parameter in the observed system.

The arithmetic means of the positions of adjacent concentrated parameters $m_{N_{(v)}}$ is calculated by summing all positions of adjacent concentrated parameters within the for loop and dividing by the number of adjacent concentrated parameters at the end of the loop. The spring force F_s is obtained by summing all the individual spring forces acting on the concentrated parameter for each connection that the concentrated parameter has. The state of the concentrated parameter h_v is started by the parameter positions. The arithmetic means of the positions of adjacent concentrated parameters, the total spring force, and their state are written in file in the two-dimensional coordinate system. Six numbers are written for each input-output data pair. The data are written in the specified file. Ten thousand input and output data pairs were recorded for neural network learning.

The neural network for the system with deformable particles used in this work was learned in the Python programming language with the Keras library. To learn the neural network, the data from the text file "treningPodatci.txt" must be loaded into the Python programming language. A script was written that loads the data in the form of input-output value pairs. The input value pair consists of four numbers representing the arithmetic value of the average of the neighbouring concentrated parameters and the current position of the concentrated parameter. The output pair values are two numbers representing the spring force on the concentrated parameter. Input and output data are each written to a separate tensor [14]. Tensor is a special form of two-dimensional fields required for learning neural networks with the Keras library. With this library, an input layer of four neurons was created, corresponding to the number of input parameters. A layer of two neurons was also created for the number of output parameters [11]. The determination of the depth of the network and the number of neurons per layer is arbitrary and depends solely on the experience of the programmer. For this work, a network depth of two layers with ten neurons per layer was chosen. This network structure was chosen because it provides higher accuracy than a single layer network with multiple neurons. The conclusion was drawn based on the accuracy of the individual neural networks. *The Rectified Linear Unit (ReLU) function* was chosen for the transfer function of the neurons in the hidden layer. The advantage of this function over other functions is that the calculation of the gradient is simpler. Therefore, the process of learning a network with *ReLU* transfer functions is much faster than others [17].

Before learning a neural network, you must select an optimization algorithm for the learning process. Keras supports several different algorithms, the latest of which is Adam [17]. Adam is an optimization algorithm that uses the Adaptive Gradient Algorithm. Since there is no noise or error in the learning data, the mean square error is used [18]. The algorithm collects the mean error value of each weighting factor and threshold. Batch Size is the number of iterations in which the error is collected before the optimization algorithm is applied. The learning speed of the neural network also depends on the batch size. For learning the neural network, the set is ten. The neural network was trained for one hundred epochs, i.e., for the entirety of the learned data [19]. After the learning process of the neural network, the calculated accuracy in this work is 97%. We can say that the percentage of accuracy is satisfactory and the network is stored in a file with extension H5, which symbolizes the records in *Hierarchical Data Format* (*HDF*). Two libraries are needed to implement a neural network in the Java programming language. The library *Deeplearning4j* [20] is a library written in the *Java Virtual Machine* (*JVM*) (© Oracle) that allows Deep Learning algorithms to be programmed. This library was used to load the HDF format in which the neural network written in the Python programming language was created. The neural network is written in the *MultiLayerNetwork* class, which we can then use to call the Output function to get the network output. This function requires a special form of data for which the *Nd4j* library is used [21]. It is designed to be used in production environments, which means that it runs faster and with less RAM. It is used to apply multifunctional functions of single or multi-dimensional fields within the Java programming language. Functions for loading and retrieving the output of the neural network have been implemented in the *ModelNN* [20] class. Selecting an approximate simulation in the window “Parametrizacija” loads the neural network from the repository.

III. Computer Program Interface Implementation

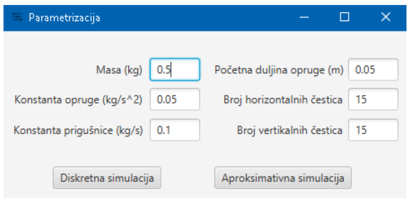
The computer program to display the simulation was written in the Intellij IDE software environment (© JetBrains), as it was easier to implement the additional libraries required for the project. Java 15 programming language (© Oracle) was used. The specified computer program consists of two windows.

A. *Window Parametrizacija*

In the given window, we have six named parameters with their units of measurement. In addition to each of the above parameters, there is a text box (Text Box) where you can enter the values of the variables. When the program is started, the default parameter values are entered into the text boxes. At the bottom of the window there are two buttons that start the simulation in a new window with the entered parameters. The "Diskretna simulacija" button starts a simulation based on a model created in a discrete domain. The "Aproksimativna simulacija" button starts a simulation whose model was created with machine learning over a discrete model (Figure 2). Due to its simplicity, the window was created with the JavaFX library (© Oracle) and the Scene Builder program (© Gluon).

B. *Window Simulacija*

The window “Simulacija” visually displays a system with deformable particles consisting of concentrated parameters and their bonds. The parameters are represented in the scheme by small red squares, which in their entirety form a lattice structure in the shape of a square.



Masa (kg)	0.5	Početna duljina opruge (m)	0.05
Konstanta opruge (kg/s^2)	0.05	Broj horizontalnih čestica	15
Konstanta prigušnice (kg/s)	0.1	Broj vertikalnih čestica	15

Figure 2. Window *Parametrizacija* display

The size and width of the grid structure is defined in the window “Parametrizacija” as the number of horizontal and vertical particles. Each parameter is connected to a neighboring parameter by a white line.

The representation of the observed system in the window is interactive with the possibility of setting an external perturbation by controlling the position with the mouse. If we select one of the concentrated parameters with the left mouse button while the program is running, we have the possibility to control the position for this parameter. When you release the left mouse button, the control over the specified concentrated parameter is removed. This approach allows the user to test the system for specific perturbations. The OpenGL programming interface (© Silicon Graphics) is used for this window implementation.

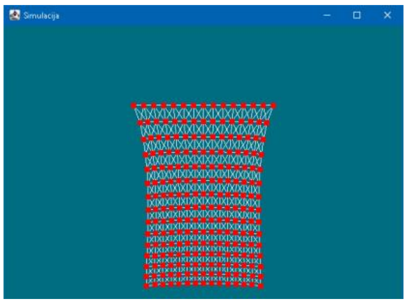


Figure 3. Window Simulacija display

C. *Implementation of a simulation in a computer program*

The simulation consists of three basic objects that must be initialized when the program is started. Concentrated parameters are objects on which mass, position, velocity and force act. The relationships between concentrated parameters are described by two positions between concentrated parameters and their mathematical relationship. A system with deformable particles is an object consisting of concentrated parameters and the connections between them. Depending on the parameters entered by the user, concentrated parameters are positioned within the object of the system in the form of a grid structure. The Buffer object is also created, which is a special form of data storage to send information to the graphics card.

For performing a simulation, Newton's second law of motion, Euler integration, and the values of the previous velocities of the observed parameter from the previous iteration step are used. The new position of the parameter is then calculated. The speed of the iteration loop depends on the FPSAnimator class, which maintains a constant number of iteration steps per second.

In order for the simulation to have a visual effect after the new positions of the concentrated parameters have been calculated, a new image is generated with the OpenGL interface. The new positions of the concentrated parameters are entered into a specially defined dynamic buffer (buffer) so that the graphics card has access to the data about the positions of the concentrated parameters. In addition to the specified container, there is another static container whose data does not change during the program execution. Such a container contains information about how the concentrated parameters are related and is needed to graphically display the relationships between the concentrated parameters. Before we can set the graphics card to draw primitive types (points and lines), we need to activate the program that was written and checked before starting the simulation loop. Next, set the pixel size of the lines and points to be drawn within the window. Increasing the size of the point will create the shape of a square, which is a concentrated parameter. To draw the concentrated parameters, you just need to change the color of the square to red.

After we set the color, we use the function to draw all the points inside the buffer to draw all the concentrated parameters. Before we plot the relationships between the concentrated parameters, we set the plot color to white. Then, we call the *Line Stripes* function, which connects the red squares with white lines to form the described system with deformable particles. This procedure for calculating the force and drawing graphic objects is performed until the program is terminated by pressing the button to close the window.

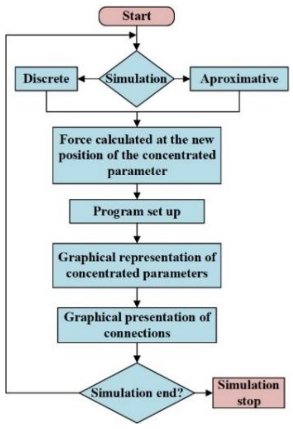

Start
Discrete
Simulation
Aproximative
Force calculated at the new position of the concentrated parameter
Program set up
Graphical representation of concentrated parameters
Graphical presentation of connections
Simulation end?
Simulation stop

Figure 4. Flowchart of the simulation loop in the program

IV. Discussion On Obtained Results

After the previously described types of simulation have been performed, the obtained time results of the process execution and their graphical representations can be compared. Since the neural network has an accuracy of 97% compared to the discrete method, a graphical representation of the difference in the simulations cannot be observed. We can say that all concentrated parameters within the observed system with deformable particles behave almost identically in both types of simulations. What can be noticed in the visual method is that the approximate simulation produces fewer frames per second. The reason for this is the length of the time interval in which the output of the neural network is determined. The average time required to obtain the neural network results is 500 μs. For example, if we have 15 rows and 15 columns in a system with deformable particles in the simulation, we can observe that the simulation works slowly due to the too long calculation of the positions of the concentrated parameters. However, if we reduce the number of concentrated parameters to, say, 5 rows and 5 columns, there is almost no difference visually between these two types of simulations. It can be observed that there is an approximately linear dependence between the number of connections of the concentrated parameter and the average length of the discrete simulation calculation. In the discrete type of simulation, the execution time t depends on the number of connections of the concentrated parameter in the system n. It should be noted that there are three basic types of connections that the observed concentrated parameter can have in this system. The average time to calculate the discrete formula for the three compounds is 20 μs. The next possible number of compounds is five, requiring an average time of 45 μs. The maximum possible number of connections with concentrated parameters is eight. The average execution time for this number of connections is 80 μs (Figure 10).

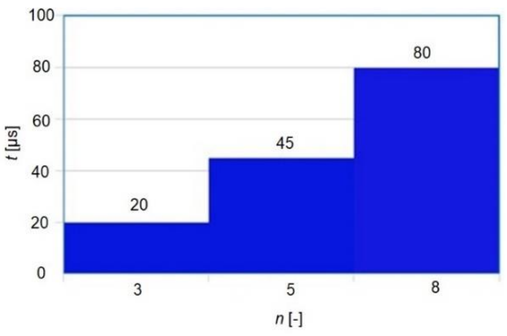


Figure 5. Display of the dependence of the execution time on the number of connections for each concentrated parameter in the system

V. Conclusion and some possible ways of future research

The paper presents the implementation of a software solution intended for the simulation of a system with deformable particles. It includes two implemented simulation approaches: the discrete and the approximate approach. The discrete approach proved to be better in constructing and testing the interface for these systems. The reason is that it is more suitable for systems with fewer connections for a single concentrated parameter. On the other hand, the approximate simulation approach was performed with a neural network over graphs and it was found to compute faster than the discrete approach that have more than twenty-two connections per concentrated parameter.

One of the directions of future research is to extend this type of simulation to the calculation of the total force on concentrated parameters. A great contribution to increase the speed of the simulation would be the use of our own libraries, which would perform the calculation of the neural network on the graphics card with the OpenGL interface.

References

[1] 1. A. Borrelli and J. Wellmann: “Computer Simulations Then and Now: an Introduction and Historical Reassessment”, NTM Zeitschrift für Geschichte der Wissenschaften, Technik und Medizin volume 27, 2019, SpringerLink, pp. 407–417, doi: https://doi.org/10.1007/s00048-019-00227-6

[2] A. Sanchez-Gonzalez, J. Godwin, T. Pfaff, R. Ying, J. Leskovec and P. W. Battaglia: “Learning to Simulate Complex Physics with Graph Networks”, Proceedings of the 37th International Conference on Machine Learning, Online, PMLR 119, 2020., URL: https://arxiv.org/pdf/2002.09405v2.pdf

[3] F. Matejiček: “Kinematika sa zbirkom zadataka”, Grafika d.o.o. Osijek, 2014.

[4] F. Matejiček: “Kinetika sa zbirkom zadataka”, Grafika d.o.o., Osijek, 2014.

[5] P. Edwards: “Mass-Spring-Damper Systems”, Bournemouth University, 2001.

[6] T. M. Atanackovic, A. Guran: “Theory of Elasticity for Scientists and Engineers”, Springer Science+Business Media, LLC, New York, 2000.

[7] D. F. Griffiths, D. J. Higham: “Numerical Methods for Ordinary Differential Equations”, Springer, 2010.

[8] J. X. Chen, C. Chen: “Foundations of 3D Graphics Programming: Using JOGL and Java3D”, 2nd edition, Springer, 2008.

[9] W. L. Hamilton: “Graph Representation Learning”, Morgan & Claypool publishers, McGill University, USA, 2020.

[10] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C, Li, M. Sun: “Graph neural networks: A review of methods and applications”, AI Open, vol. 1, 2020, pp. 57-81., doi: https://doi.org/10.1016/j.aiopen.2021.01.001

[11] Branko Novaković, Dubravko Majetić, Mladen Široki: “Umjetne neuronske mreže”, Fakultet strojarstva i brodogradnje, Zagreb, 2011.

[12] A. Howard: “Elementary Linear Algebra”, 5th edition, Wiley, New York, USA, 1987.

[13] E. Thiu: “Understanding Latent Space in Machine Learning”, URL: https://towardsdatascience.com/understanding-latent-space-in-machine-learning-de5a7c687d8d

[14] P. Galeone: “Hands-On Neural Networks with TensorFlow 2.0”, Packt Publishing, 2019.

[15] S. Pal, A. Gulli: “Deep Learning with Keras: Implementing deep learning models and neural networks with the power of Python”, Packt Publishing, 2017.

[16] J. Brownlee : “A Gentle Introduction to the Rectified Linear Unit (ReLU)”, URL: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks

[17] J. Brownlee : “Gentle Introduction to the Adam Optimization Algorithm for Deep Learning”, URL: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning

[18] E.L. Lehmann, G. Casella: “Theory of Point Estimation”, 2nd edition, New York: Springer, New York, USA, 1988.

[19] S. Sharma: “Epoch vs Batch Size vs Iterations“, URL: https://towardsdatascience.com/epoch-vs-iterations-vs-batch-size-4dfb9c7ce9c9

[20] R. Ray: “Java Deep Learning Cookbook”, 1st edition, Packt Publiching, 2019.