COLLISION RESOLUTION FOR CLOTH SIMULATIONS USING MACHINE LEARNING

An Undergraduate Research Scholars Thesis

by

AHSAN YAHYA

Submitted to the LAUNCH: Undergraduate Research office at
Texas A&M University
in partial fulfillment of requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor: Dr. Shinjiro Sueda

May 2023

Major: Computer Science

RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

I, Ahsan Yahya, certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with my Faculty Research Advisor prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

TABLE OF CONTENTS

Page

ABSTRACT

Resolving Softbody Collisions with Machine Learning

Ahsan Yahya
Department of Computer Science of Engineering
Texas A&M University

Faculty Research Advisor: Dr. Shinjiro Sueda
Department of Computer Science and Engineering
Texas A&M University

Texas A&M University

A key aspect of 3D animation is physics based simulation. To create realistic animations for objects such as cloth and fabric, a category of mathematical models known as softbody simulations are used. There are a wide variety of techniques for simulating the behavior of cloth, with varying degrees of performance and visual fidelity. One such technique that this project builds off of involves calculating the frictional forces between a cloth and intersecting geometries. This ensures that cloth objects display realistic behavior when sliding over rigid surfaces or being dragged by high friction materials. It is especially useful for depicting the movement of garments, as this technique can capture how real world clothing clings to the human form. With this technique, as with many, there is a tradeoff between performance and visual realism. While the latter is desirable for many applications in filmmaking and video game development, it can come at the cost of rendering time or real time performance. This friction based simulation produces visually realistic results, but it can be computationally expensive to

simulate, especially for complex geometries. This project aims to create a deep learning model that can approximate the results of friction based collision resolution with better performance. By training a model on data collected from the real results of this technique, it could give similar results at potentially greater performance.

DEDICATION

To our friends, families, instructors, and peers who supported us throughout the research process.

ACKNOWLEDGEMENTS

Contributors

I would like to thank my faculty advisor, Dr. Shinjiro Sueda, for his guidance and support throughout the course of this research.

Thanks also go to my friends and colleagues and the department faculty and staff for making my time at Texas A&M University a great experience.

I would also like to thank my good friend Heather Drouse for allowing me to use her graphics card to process data for this project.

The base code for the C++ simulation and renderer was provided by Dr. Shinjiro Sueda as part of his Computer Animation course.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

This project received no funding.

NOMENCLATURE

M	Diagonal mass matrix
v	Velocity matrix
h	Simulation time step
K	Stiffness matrix
x	Position in 3D space

1. INTRODUCTION

1.1 Cloth Simulation Techniques

Tools that can simulate the movement of cloth and fabric are integral in the field of 3D computer animation. Many video games and animated films utilize algorithms that can simulate the effect of stretching and compression on a piece of cloth, as well as resolve collisions between the cloth and other objects. There are a wide variety of techniques for achieving this. Typically, a piece of cloth can be represented as a grid of points in 3D space with some mass connected by some constraint. The simplest approach is to treat the particles as individual rigid bodies, and connect them with rigid constraints. This may be efficient but will not give realistic stretching that real world fabrics experience. Another approach is to treat each particle as a member of a mass-spring system, and simulate spring forces between adjacent particles according to Hooke's law [3]. In this technique, a "stiffness matrix" is created, which can be considered analogous to the spring constant of a single mass spring system. With this matrix, the previous positions of each of the cloth's particles, and their velocities, implicit integration can be used to solve for the particles' updated positions. There are many cloth simulation techniques with different approaches, but most of them fundamentally use a similar concept of a system of masses connected by some elastic constraint. The finite element method (FEM) is another technique for simulating softbody dynamics that can be applied to cloth and fabric. Instead of connecting particles by linear springs, nodes are connected with tetrahedral shapes. This technique is computationally complex, and yields results that are much closer to real world behavior. Another technique incorporates the typical mass-spring system but computes additional forces to simulate frictional contact a cloth and surfaces it comes in contact with [4]. This allows for many

situations where an object may be sliding or hanging over a solid object to be simulated more realistically. This is especially useful for applications such as garment animations, as this can accurately depict the way cloths cling to a human body as it moves.

1.2 Challenges in Cloth Simulation

While these techniques are effective, there is often a tradeoff between performance and accuracy. For complex geometries, a single frame of animation can take a long time to render due to the complexity of the simulations being conducted [1]. This impacts the efficiency of animators and can make techniques unsuitable for real time applications. There is therefore an interest in not only increasing the fidelity of cloth simulations, but also to make them more performant, as to save time and processing power. One approach for this challenge is to utilize deep learning in order to approximate simulation models. Artistic applications such as animation, filmmaking, and video game development do not necessarily need the most exact or scientifically accurate results in their product. Rather, some artistic license can be taken to create a visually appealing result. For this reason, deep learning could potentially used to create a faster approximation of any given simulation technique [2]. By creating a dataset consisting of collision incidents and their resultant forces according to the real method, which in the case of this project, will be the friction based collision resolution, a machine learning model can be trained to approximate this result. This model would take some location and particle data corresponding to a collision as an input and output the force needed to separate the geometries. Once a model can be trained to evaluate the force to resolve a collision at speeds and accuracy close to the explicit formula, optimizations can be made to the model, such as simplifying the architecture, to reduce the time taken to compute the resultant force of the collisions each frame. While these optimizations may cause the model's results to not be exactly accurate, the goal is to

create visually appealing results, so small discrepancies that would go unnoticed by an audience are acceptable for performance gains.

2. METHODS

This project utilizes course material from Dr. Shinjiro Sueda's CSCE 450 Computer Animation course. Specifically, I used the assignment on cloth simulation as a basic framework. The initial assignment was to write a simulator in C++ with implicit integration and a simple repulsive force to resolve collision. From there, I worked on incorporating a machine learning model using Python.

2.1 Simulation framework

The core of this project uses a simulation renderer written in C++. The renderer was provided by Dr. Sueda as part of the assignment's base code. Scene geometries are rendered using OpenGL. Objects can be visualized with both Phong lighting and a wireframe model. Physics simulation occurs within the render loop. The main scene used for training and evaluation consists of two geometries, a cloth and a fixed rigid body moving in sinusoidal motion along the world z axis. The cloth is represented by a two dimensional array of n particles in 3D space, represented as small spheres with a mass and velocity. The particles in the top right and top left corners are fixed, and experience no forces. At the start of the simulation, only gravitational forces act on the cloth. When the rigid body intersects with the cloth as it moves along its path, forces are applied to the cloth to push away the colliding vertices. These forces are calculated according to different models which will be discussed in the following sections.

2.2 Integration techniques

To simulate intra-cloth movement, spring forces are modelled between adjacent particles according to Hook's law. To understand the intuition between using spring mechanics to simulate cloth movement, consider the case of a single spring attached at two points. Given a

constant associated with the spring itself and the distance between the two points, it is trivial to compute the force being exerted by the spring against the two points. With this force, an integration step can be conducted with respect to time in order to calculate the velocity and position of the two attachment points resulting from the force of the spring. Extending this to a large grid with springe connections between points is useful to capture the elasticity of fabric materials while maintaining realistic constraints. For this project, spring connections are established between adjacent points on a horizontally, diagonally and vertically, as well as "bending" connections, which extend horizontally and vertically from each particle to the second one away, providing increased stability as the cloth bends and folds.

In order to predict the frame by frame changes for each particle in the system, the backward Euler method is used. While explicit integration can work for this type of application, it is severely limited by instability at large time intervals. With the implicit integration method, each frame, the following system of equations is solved to predict each particle's position on the next frame:

$$Ax = b \tag{1}$$

$$A = M - h^2K \tag{2}$$

$$b = Mv + hf \tag{3}$$

For a system with *n* particles, each with a consistent index *i,* Matrix *M* is a diagonal matrix of size *nxn* representing the masses of each particle. *v* is a vector of size *n* containing the velocity of each particle. *f* is a vector of the same size containing the net forces on each particle. *h* represents the time interval between frames. The stiffness matrix *K* is an *nxn* matrix that can be thought of as analogous to the spring constant of a single mass-spring system. It consists of the derivatives of each particle's force with respect to its position, organized throughout the matrix

according to the pairs of particles applying forces to each other. For a particular spring, its stiffness matrix is computed using the following equation:

$$K_s = \frac{E}{l^2}\left(\left(1 - \frac{l-L}{l}\right)(\Delta x^T \Delta x) + \frac{l-L}{l}(\Delta x^T \Delta x)I\right) \tag{4}$$

where *E* is a stiffness constant for the entire cloth, analogous to the spring constant of a single mass-spring system, *l* is the distance between two points on a spring, *L* is the rest length of the spring at equilibrium, and Δx is the difference between the two particles attached to either end of the spring. This result is a square matrix of size three, and is used for the following two by two block matrix:

$$K = \begin{pmatrix} -K_s & K_s \\ K_s & -K_s \end{pmatrix} \tag{5}$$

This elements of this block matrix are added to the overall stiffness matrix at the respective indices of the particles attached to the spring. For particles of a spring with indices i and j respectively, the elements of the block matrix will be added, from left to right and starting at the top row, indices (i,j), (i,i), (j,i) and (j,j) respectively.

For performance, this implementation uses sparse matrices. For a sparse solver, the Eigen library's conjugate gradient solver is used to solve for *x*, using 25 iterations and a tolerance of 1e-6. This gives visually realistic results without any noticeable artifacts, on par with a dense matrix solver.

Collision between the cloth and the rigidbody is resolved by checking for intersections between the rigid geometry's triangles and the particles on the cloth, as well as intersections between the rigid body's vertices and the cloth's triangles. The naïve model for resolving the

collision entails applying a repulsive force to the cloth's particle(s) by multiplying the collision surface normal by the penetration depth and scaling by some constant *c*. This constant is not exact and is determined through a process of trial and error according to whatever gives the best looking results.

Collision resolution has its own set of procedures for stiffness calculations. For the simplest case of a rigid body triangle intersecting with a cloth point, the stiffness block matrix consists only of the repulsive force magnitude multiplied by a 3x3 identity matrix. This matrix is added to the stiffness matrix at the index of the cloth particle. A more complicated case is for the resolution of a rigid body vertex intersecting with a cloth triangle. Since the collision point is not exactly at one of the three triangle vertices, the force must be interpolated across the triangle by calculating its barycentric coordinates. Since each vertex has its net force changed, their respective entries in the stiffness matrix must be updated as well. For each pair of vertices on the intersecting triangle, the stiffness block matrix is added at the corresponding indices scaled by the product of both vertices barycentric coordinates.

2.3 Naive model

The following equation represents the collision resolution force implemented in the naïve model.

$$f = cnd \tag{6}$$

where *n* is the normal of the collision surface, *d* is the penetration distance, and *c* is a constant scalar. The penetration distance is computed by taking the magnitude of the projection of the vector from one of the triangle vertices to the intersecting vertex onto the normal. This can be represented by the following equation where *a* is an arbitrary point on the collision plane, and *x* is the intersecting vertex position:

$$\mathrm{d} = (x - a) \cdot n \tag{7}$$

This force is applied when an intersection is detected between either of the scene geometries. Collisions are detected in the update loop before the integration step. To check if a vertex *x* is intersecting with a triangle of the other geometry, the barycentric coordinates of the triangle are computed with the *x*. If all the barycentric coordinates are within the range of zero to one, then we know that the *x'*s projected position onto the triangle plane is within the bounds of the triangle. If this check passes, the triangle normal is compared to the vector from the triangle center to *x* using the dot product. If the result is negative, the distance is computed from the triangle to *x* using the method for calculating the distance between a point and a plane specified above. A collision is registered if the distance is smaller than an arbitrarily small value, which is used to ensure that false positives do not occur. Without this epsilon, a collision could be triggered by far away triangles just because they are facing away from *x*.

A hack is used for this model during the collision detection phase. Clipping artifacts occasionally arise when this model when *c* is too low, where colliding objects appear to be inside one another. The seemingly logical fix for this issue is to just increase the value of *c* to increase the magnitude of the repulsive force, but this can cause some side effects. When *c* is too high, the large force can give the appearance that the cloth is being "punched" away from the intersection point, causing jumpy and abrupt velocity changes instead of being smoothly pushed away. To prevent clipping with lower *c* values, the local position of *x* is scaled away from the center of its geometry by a small epsilon, around 0.025 units. By increasing the threshold of the distance required between geometries to register a collision, the simulation can begin to handle intersections one or two frames before they actually occur. While upon close inspection this can

seem to create a small “force field” around the colliding geometry, it is largely unnoticeable for small enough epsilons, and still successfully mitigates the clipping problem.

2.4 Approximating Naïve model

To create a proof of concept as to whether or not machine learning was feasible for the goal of approximating cloth simulation algorithms, a deep learning model was trained to replicate the naïve collision resolution function. The first step was to model triangle to vertex collisions. A trained neural network would need to take in positional information about 3 vertices comprising a triangle, and a fourth vertex intersecting with it. For this task, a training loop was written in a Python script using the Pytorch deep learning library. The goal of the neural network was to take in some data about an intersection and output a three dimensional vector representing the resultant force. The model’s architecture was that of a fully connected feed forward network with five hidden layers, each having 16 nodes and an activation function of Leaky ReLU. Since this is an essentially a regression problem, Leaky ReLu seemed to be a good choice as it is unbounded. The model has seven inputs, which are the locations of each vertex involved in the collision. Ordinarily, to represent four vertices in 3D space, a minimum of twelve total values is needed, three for each vertex. However, we can make the assumption that it doesn’t matter where in the world the vertices are, as long as we know their positions relative to each other. This is due to the fact that the value of the output is dependent on only the penetration depth and the collision surface normal. The former is only dependent on the relative positions of the vertices, and while the direction depends on the orientation of the input vertices, it can be transformed to a reduced local space and back to global space. Given this, the input data can be transformed in any way as long as the relative positions of each vertex remain the same. To reduce dimensionality, an arbitrary vertex is chosen to be the origin. Each vertex is translated such that

the new origin vertex is zero. One translated vertex is chosen to be aligned to the horizontal axis, and a rotation matrix is constructed to rotate the vertex around the origin to be aligned to the x axis. The entire system is rotated as well, ensuring that all the relative positions of each vertex remain the same. By conducting this transformation, the origin vertex becomes a constant that can be discarded in the network's input, and the second vertex only needs its transformed horizontal component to be specified. The number of inputs to the model is thereby reduced from twelve dimensions to only seven.

The dataset used for training was generated by the C++ simulation code. Each time step, each collision and its resultant force is appended as a row to a .csv file. Each row recorded in the data set consists of the reduced vertex coordinates as the model input and the result force for the model output. In order to help the model to generalize for different scenarios, data is recorded for many different configurations of the rigid body's initial position as to capture information on a wide range of angles and penetration depths. To simplify training for this simple collision resolution mode, the constant c is kept as a constant of around 0.25 throughout the force calculations. This dataset is read by a separate Python script and used to create training and validation sets with which the model is trained and evaluated. Training was conducted with the Adam gradient descent optimizer and a step learning rate scheduler, used to ensure the model converges to the global minima by gradually decreasing the learning rate over time. After 3000 epochs, the model converges to a mean squared error loss value of around 1e-5. This naïve model converged notably quickly, not needing many thousands of epochs to converge to a low loss. This can most likely be attributed to the fact that the training data is very linear, as it essentially is just a constant multiplied by a scalar and a vector.

In order to qualitatively evaluate the efficacy of the trained model, code is implemented in the C++ simulation to replace the existing collision resolution formula with the results of the newly trained model. Helper functions written in Python to allow access to the trained model can be called using the Python runtime's C++ bindings. At the start of the simulation, the model is loaded from the disk into the Python runtime's memory. For each collision that is detected, the vertex coordinates are transformed and passed to a Python function that evaluates the output of the model with this input. This output is then applied to the colliding particles as the value of *f* as described above.

2.5 Friction based collision resolution

While the naïve collision resolution model has some benefits in its simplicity and relatively low computational complexity, there are instances where one would want more accuracy and control over the behavior of simulated materials. For this, we use a more complex model that accounts for friction that occurs between colliding surfaces, previously proposed by Xu et. al [5]. The new computation for the resultant force of a collision consists of two major components, the repulsive contact force and the friction force tangent to the collision normal. The first force is calculated as such:

$$f_c = \left(-k_n * d + k_d d\dot{d}\right)sn \tag{8}$$

where k_n is the contact stiffness coefficient, k_d is the damping coefficient and *s* is the force scale, analogous to *c* from the naïve model. *d* remains the penetration depth, but a new variable $\dot{d}$ represents the collision speed. This value is computed by subtracting the two colliding body's velocities and taking the magnitude. For situations where the velocity of a body is not uniform throughout the mesh, as is the case with the cloth, the velocity of nearby particles are

interpolated using their barycentric coordinates. The velocity of the intersection is further utilized in the second component of this model, which computes forces caused by friction between the two surfaces as follows:

$$f_t = -min(k_t\|\dot{t}\|, \mu\|f_c\|)s\frac{\dot{t}}{\|\dot{t}\|} \tag{9}$$

where k_t is the stiffness coefficient of the friction force and μ is the coulomb frictional coefficient, and $\dot{t}$ is the tangential velocity of the collision. The two coefficients can be thought of analogous to coefficients of static friction and dynamic friction respectively. The first term of the minimization function grows proportionally with the magnitude of the tangential velocity, and the second term simulates dynamic friction by applying a force proportional to the collision force, which is normal to the collision surface. The tangential velocity of the collision, $\dot{t}$, is computed using the collision velocity calculated for the contact force in the previous step by taking the component of the velocity along the direction of the normal and subtracting it from the net velocity. This results in a vector that represents the component of the velocity tangent to the surface of the collision. Since friction resists the direction of an object's movement along a surface, this tangential velocity can be used to give a direction to the frictional force.

2.6 Data wrangling

To create and train a neural network to approximate this more complicated model for the collision force, a similar approach was taken for the naïve model and extended to account for the new variables. As before, collision data was collected from the simulation, which was run with the explicitly defined naïve model. Data recorded consisted of the locally transformed vertex coordinates of the intersection points, but also another input that wasn't present when approximating the naïve model, that being the collision velocity. A separate Python helper script

was used to read in the rows of recorded position and velocity data and compute the forces resulting from each collision. Because the force required to resolve an intersection is heavily dependent on several configurable coefficients, they must be included in the model inputs so that combinations of different physical materials can be simulated. To ensure the trained neural network can account for these variables, different values of coefficients are used to compute force for each sampled collisions, sampling from a predefined set of values and testing each possible combination. This presented a challenge, however, due to the volume of data resulting from this. With four different values tested for each of the five coefficients, there are 1,024 new rows generated per initial sample. If all the recorded data was processed, then the final dataset would come to around 21 gigabytes. While more data would undoubtedly help the model to more efficiently generalize, this sheer amount of data is prohibitively large for prototyping, due to RAM constraints and the time it would take to propagate the data through the network. For this reason, only some of the recorded rows are processed, contributing to a smaller dataset which can be used for prototyping and testing.

While this approach manages to make some usable data for training, it lacks uniqueness due to how the same vertices are repeated in the dataset for every combination of friction coefficients and that only a few friction coefficients are included in the data. This results in the model overfitting to a few points that make it into the final training set, but not generalizing well. Training with this approach did not bring a validation loss less than 1e-1. To ensure the data is more uniformly distributed and that there is representation for a wider range of situations, a different approach was taken for data generation. Let N be the total number of samples generated by the simulation, and let n be the number of samples intended to be used for training purposes. We can take n random samples from the original dataset, and for each of them generate a

random set of friction coefficients between a predefined range. This way, more positional values are able to be included in the dataset, as well as a full range of possible friction coefficients.

A similar neural network architecture as was used to approximate the naïve model was used for this more complicated one, consisting of a feed forward network with several hidden layers, but now instead with an activation function of Leaky ReLu. Converging the loss for this model demonstrated to be more complex and challenging than the naïve model, likely due to the complexity of the force components and the added nonlinearity in the friction term. Another factor that made training challenging was how drastically the scale varied between the input features, especially the friction coefficients. For example, the friction stiffness coefficient could be as large as 1,000, but the coulomb frictional coefficient is restricted to the range between zero and one. This disparity between the scales of each feature can cause inputs with larger numerical values to become emphasized over smaller features. To better understand how this effect occurs, consider the following bivariate model:

$$y = \alpha x_1 + \beta x_2 \tag{10}$$

Suppose that both parameters α and β are equal to one. If the input $x_1$is much larger than x_2, such as might be the case with the two friction coefficients previously mentioned, then this will cause the output y to be extremely large compared to the smaller input. One would think that if the parameters for both inputs are equal, then they must be equally important for the model, but this demonstrates how the scale of a feature can exaggerate its significance in a model. To mitigate this problem, input features are scaled using the sklearn library's implementation of min-max scaling. This entails taking the minimum and maximum of each input feature, subtracting the minimum value, and finally dividing the values by the maximum

minus the minimum, effectively normalizing the range for all the inputs between zero and one. The sklearn standard scaler was also applied to the target force data, as the large values of the forces caused high loss values and prevented the model from converging. Unlike the min-max scaler, the standard scaler computes the standard deviation and mean of each feature and scales it by subtracting the mean, then dividing the result by the standard deviation. This produces a feature with a mean of zero and a standard deviation of one. While this output does not represent the correct force value, it can be easily transformed back by multiplying by the standard deviation and adding the mean.

2.7 Stiffness Approximation

One major obstacle when implementing friction based collision is the need to recalculate the stiffness matrix. Due to how simple the naïve model was, computing its derivative was trivial and simple to implement. The friction based force uses a much more complicated formula, however, and deriving the relevant gradients of the formula isn't so straightforward. Without deriving or computing the new stiffness matrix in some way, the simulation would only be able to conduct explicit integration, which is far more unstable than the implicit integration that has been used up to this point. One of the benefits, therefore, of using a neural network to approximate the force equation is that we can use the gradients of the network to approximate the stiffness. Pytorch has several useful built-in functions for approximating derivatives. The one most relevant for this task was torch.autograd.functional.Jacobian. This function utilizes the accumulated gradients created during the training process to approximate the Jacobian matrix of trained neural network's output with respect to some arbitrary input. If this can be used to take the partial derivatives of the collision force's components with respect to the positional of the

particles the force is acting on, that would give us a close approximation of its 3x3 stiffness matrix.

While this sounds good in theory, this approach comes with a few caveats. Firstly, dimensionality reduction can't be applied to the data before training as described in section 2.4. Although this form of dimensionality reduction is extremely helpful when it comes to training, the input vertices must be represented in global space so that the output of the model and its partial derivatives also remain in global space. Even without dimensionality reduction techniques, it is still possible to limit the amount of data that needs to be fed into the model by doing some precomputation. For the purpose of approximating the stiffness matrix, the relevant input is the vector between the collision point and the colliding vertex, or alternatively, the normal of the collision plane multiplied by the collision distance. This can be derived from the four colliding points and passed as a vector of rank 3. Thus, the inputs required for this task are the x, y and z values for the collision vector, the collision velocity, and the friction coefficients. The force scale can be omitted, since it can easily be used to scale the model output after evaluating, leaving just 10 inputs for the model to learn.

In order to retrain the model with this set of parameters, batch training was implemented. This was due to the fact that the size of the dataset had grown greatly, as well as that experimentally, batching data instead of back propagating the loss on the entire dataset seemed reduced the chance of the model plateauing at high loss values. Through process of trial and error, assisted by a Bayesian search conducted using the Python library Ax, developed by Facebook for hyperparameter tuning, the best working model found consists of an architecture of 7 hidden layers. The first two layers have 256 nodes, and each layer after that has 128 nodes. The model was trained at a learning rate of 1e-4 with a batch size of 512. After 45,500 epochs, the

learning rate was raised to 1e-3 and the batch size was increased to 32,768 in order to refine the model further. This was motivated by findings by Smith et al [6], which suggest that increasing the batch size throughout the process of training can produce the same effect as lowering the learning rate and prevent the model from plateauing sooner.

3. RESULTS

3.1 Training results

Table 1: Loss over epochs.

Epochs trained	Training Loss	Validation Loss
3000	0.00191	0.00291
5000	0.00177	0.00297
7000	0.0016	0.00303
10,000	0.00151	0.00309
45,500	0.141	0.0705
45,600	0.00297	0.002875
45,700	0.00258	0.00293

Table 1 shows the mean squared error loss of the model over the course of the training time. It should be noted that at epoch 45,500, the training script restarted with new parameters and reshuffled data, causing a temporary spike in loss. At the end of training, the validation loss came down to a 2.58e-3 and seemed to hover around that value. Some overfitting can be observed before the restart, as the validation loss shows a pattern of increasing while the training loss decreases. It would seem that modulating the batch size and learning rate as described in section 2.7 did not produce significant benefits, as the best validation loss after restarting is no better than the best validation loss before the restart. An beneficiel strategy to optimize the change in batch size and learning rate might have been to run a wider Bayesian search on a larger range of hyperparameters, but this approach was constrained by the time it would take. On the hardware available, this training loop took about eight hours to train. For future work on this project, it would certainly be beneficial to spend time exploring different configurations for training.

3.2 Animation test

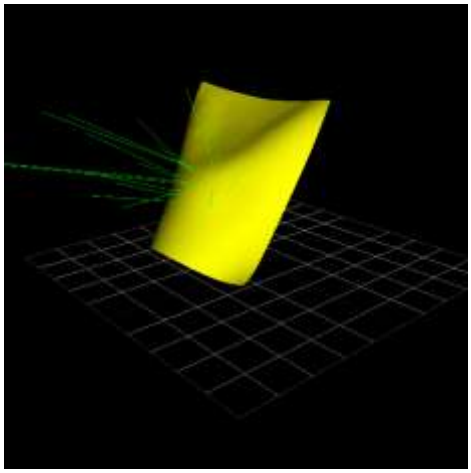

Figure 1: Sphere colliding with cloth pinned at top corners

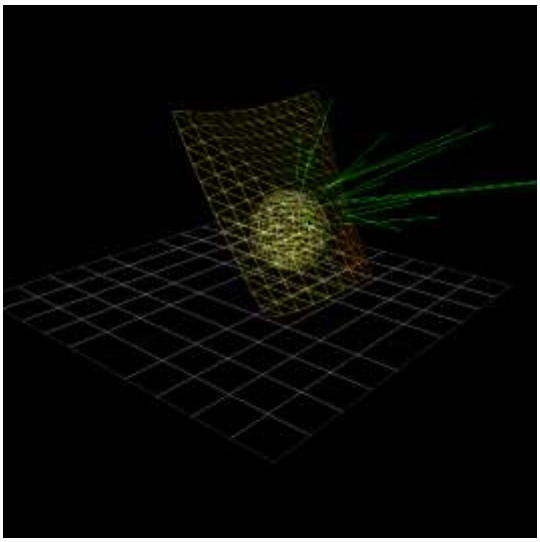

Figure 2: Colliding objects in wireframe view

Figures 1 and 2 display the result of implementing the trained model in the simulation's collision resolution step. The green line segments represent the forces acting on each cloth particle due to a collision, as computed by the neural network. The stiffness matrix calculation was also replaced by the approximation of the model Jacobian, as described in 2.7. Overall, the results of the new simulation seemed to give mostly correct results. There is some noise in the output, as can be observed by the somewhat skewed directions of the force vectors in figures 1 and 2, but the force does push the cloth away from the sphere at the point of intersections. Importantly, the cloth remains stable, and the velocities of the cloth's particles do not "explode" in magnitude. Without the stiffness matrix, the simulation becomes no better than explicit Euler integration, notorious for becoming quickly unstable. The fact that the cloth can keep its shape without the distances between particles in the cloth exploding to extremely large values indicates that the stiffness matrix is being computed correctly.

Although these results are encouraging, there are some caveats. Firstly, the noise in the model output, while not excessive, is still large enough to be noticeable by anyone watching the animation. For this model to be a full replacement for the explicit equation, more work would have to be done to reduce the loss much further. Furthermore, the goal of improving simulation speed was not fully met. With the force equation explicitly implemented, one period of the sphere's movement takes approximately 21.95 seconds to complete. With the trained model, it takes 31.14 seconds. For future work on this topic, different optimizations could be implemented during training, such as using dropout layers to reduce the number of weights that need to be processed, or autoencoding to reduce the feature space.

4. CONCLUSION

The goal of this project was to develop a machine learning technique to approximate and optimize a complex friction based force to resolve collisions with cloth objects. While more work remains to be done in order to improve the performance of the model evaluation, this project demonstrates that it is indeed possible to model such simulated forces using machine learning techniques. Moreover, we show that machine learning can also be used to approximate the stiffness matrix, a key part of the commonly used implicit Euler integration method. Although there is still work that can be done to improve our method and make it fully usable for animation applications, this project demonstrates that this technique is possible, and may hopefully serve as the groundwork for a practical implementation.

REFERENCES

[1] Adam W. Bargteil University of Maryland, A. W. Bargteil, U. of Maryland, T. S. U. of California, T. Shinar, U. of California, U. of C. V. Profile, P. G. K. M. G. University, P. G. Kry, M. G. University, and O. M. V. A. Metrics, "An introduction to physics-based animation: SIGGRAPH Asia 2020 Courses," ACM Conferences, 01-Nov-2020. [Online]. Available: https://dl.acm.org/doi/10.1145/3415263.3419147. [Accessed: 11-Sep-2022].

[2] I. Santesteban, M. A. Otaduy, and D. Casas, "Snug: Self-supervised neural dynamic garments," arXiv.org, 05-Apr-2022. [Online]. Available: https://arxiv.org/abs/2204.02219. [Accessed: 11-Sep-2022].

[3] L. Fulton, V. Modi, D. Duvenaud, D. I. W. Levin, and A. Jacobson, "Latent-space dynamics for reduced deformable simulation," Eurographics DL Home, 01-Jan-1970. [Online]. Available: https://diglib.eg.org/handle/10.1111/cgf13645. [Accessed: 11-Sep-2022].

[4] M. G. E. T. H. Zürich, M. Geilinger, E. T. H. Zürich, E. T. H. Z. V. Profile, D. H. E. T. H. Zürich, D. Hahn, J. Z. U. de Montréal, J. Zehnder, U. de Montréal, U. de M. V. Profile, M. B. D. Research, M. Bächer, D. Research, D. R. V. Profile, Bernhard Thomaszewski ETH Zürich and Université de Montréal, B. Thomaszewski, ETH Zürich and Université de Montréal, ETH Zürich and Université de MontréalView Profile, S. C. E. T. H. Zürich, S. Coros, M. P. I. Informatik, and O. M. V. A. Metrics, "Add: Analytically differentiable dynamics for multi-body systems with frictional contact: ACM Transactions on Graphics: Vol 39, no 6," ACM Transactions on Graphics, 01-Dec-2020. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3414685.3417766. [Accessed: 11-Sep-2022].

[5] J. Xu, "An end-to-end differentiable framework for contact-aware robot design," *An End-to-End Differentiable Framework for Contact-Aware Robot Design*. [Online]. Available: http://diffhand.csail.mit.edu/. [Accessed: 20-Feb-2023].

[6] S. L. Smith, P.-J. Kindermans, C. Ying, and Q. V. Le, "Don't decay the learning rate, increase the batch size," *arXiv.org*, 24-Feb-2018. [Online]. Available: https://arxiv.org/abs/1711.00489. [Accessed: 03-Apr-2023].