A Mass Spring Particle Model for Bouncing Mechanics of Deformable Solids

Final Project for MECHENG 599 Fall 2021

Mitry Anderson
mitryand@umich.edu

***Abstract*—This project, completed for the course MECHENG 599: Introduction to Robotic Manipulation at the University of Michigan taught by Nima Fazeli, explores the use of a mass spring particle model to simulate the mechanics of various deformable objects bouncing on a flat surface, geared toward future applications in a manipulation pipeline. A review of a few relevant studies is conducted, then the equations developed for the model are described in detail. We then evaluate the performance of the simulator by testing it with different objects as well as measuring the time it takes the algorithm to execute for various numbers of particles. The simulator performed well for a variety of object shapes, though as the input parameters were varied problems arose with high stiffness, high damping, and low mass values, which required increasingly smaller time steps to resolve. The execution time of the simulator appears to increase linearly with the number of particles, so for large numbers of particles decreasing the time step becomes less feasible. Despite these limitations, this exploration of a simple mass spring particle system demonstrates the predictive strength of the method within the bounds of its acceptable parameters.**

I. Introduction

Accurate and fast methods for evaluating the dynamics of deformable bodies are highly sought after in fields such as robotics and computer graphics, where a robust system model is relied upon for other tasks (for example control systems or animation). One common approach to modeling a deformable solid is to conceptualize it as consisting of numerous point masses connected by linear elastic springs. The goal of this project is to develop a general mass spring model (MSM) for a 2 dimensional bouncing deformable object of arbitrary shape in Python 3, using the numpy library [1] for matrix calculations and the matplotlib library [2] for rendering. The object's geometry will be represented as a png image, with black cells representing areas of mass and white cells representing free space. Our program will then take each black cell in the image and generate a point there. The points will then be connected with springs, and a standard Newtonian model of physics for a point mass will be used to calculate the force on each mass. Euler integration will be used to calculate the effect this force has on each mass over a discrete time step. Parameters such as the length of the time step, the mass of each point, the stiffness of the springs, damping coefficient, and so on will all be configurable. The effects of adjusting these parameters as well as the object geometry will be explored in this report, and the performance of the algorithm with respect to the number of points will be evaluated.

II. Literature Review

Deformable body mechanics is an ongoing and broad field of research. As this project focuses on the bouncing of deformable bodies, a review of a few studies relevant to this area is presented here. Basic bouncing mechanics could be described by a single mass spring damper system, as presented by Nagurka et. all [3]. Their model enables the calculation of an equivalent stiffness and damping coefficient for a ball from measurements of the contact time during the bounce, as well as an estimate for the coefficient of restitution. Despite the fact that model was limited by not accounting for the drag force on the ball, insight into the mechanics of bouncing with deformation can still be gleaned. For example, they mention that a larger coefficient of

restitution correlates with less damping but doesn't correlate with stiffness.

For robotic manipulation, a more detailed model of the object is often desired, as the goal of manipulation is often to change either the pose or the geometry of the object. Common methods for modeling the deformation of deformable bodies include particle based systems where collision mechanics dominate, mass spring systems as used in this project, finite element methods, finite volume methods, and more, as summarized by Arriola-Rios et. all [4]. The selected model can then be fit into the manipulation pipeline as they discuss, to be used for model predictive control or manipulation planning. As discussed by Arriola-Rios et. all, mass spring models are useful for their ease of implementation and computational efficiency, however they are limited by the difficulty both of maintaining volume and of tuning the constants. Some work has been done on the selection of appropriate constants for discrete mass spring systems to approximate the effects of the Young's modulus for a particular material, as explored by Lloyd et all [5]. The paper develops a method to assign spring stifness values to triangular, rectangular, and tetrahedral MSM elements by relating the stiffness matrix for each element in its equilibrium position to that found using a finite element model. A major limitation of this approach was that the stiffnesses they found were valid only for a specific Poisson ratio of $\frac{1}{3}$. Even still, MSM methods for defining deformable objects are a promising area of research.

III. Methodology

Each particle in our simulation is represented by a python class that contains the particle's instantaneous position, initial position, instantaneous velocity, and a list of other particles it is connected to. The class also contains the methods needed to update the force and position of the particle.

Before any dynamics can be simulated, we first must have particles to simulate. To this end, we implement an algorithm to parse a png image to create particles as well as assign connections between particles. Given a png image and a desired particle spacing, a particle is created for every point in the image with a black cell, with an inital position given by Eq. 1, where $\vec{x}_i$ is the position of the point

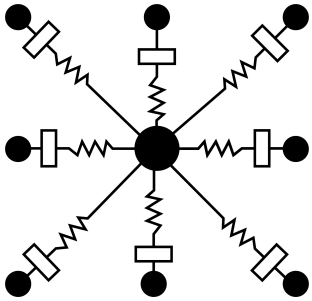

Fig. 1. The 8 possible connections between the central point (larger for emphasis) and its neighbors are shown. Obviously, when there is no particle there, the connection is neglected.

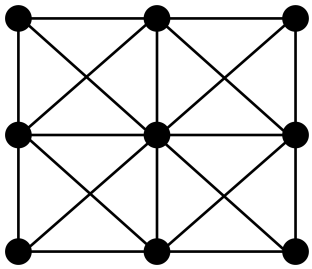

Fig. 2. The mesh structure is shown for a 3x3 square. This structure is continued throughout the object.

corresponding to location (u,v) in the image, and s is the scale factor [m].

$$\vec{x}_i = \begin{bmatrix} u * s \\ v * s \end{bmatrix} \tag{1}$$

As the particles are created, they are stored in a list, with zeros as placeholders for empty cells in the image. Once every particle is created, this list is iterated through again, and a list of connected particles is generated for each particle, such that each particle is connected as seen in Fig. 1. Details at the level of figuring out the array indexing are not included here for brevity. Each of these connections represents a spring and damper in series in between the two points. The mesh created by these connections is visualized in Fig. 2. The diagonal elements help the object to keep its form during simulation.

With the particles created in the object's initial configuration, we now need a way to change the initial orientation. To achieve an initial angle of θ_0 and an initial drop height of h_0, the original particles are iterated through once more. Each particle is rotated by θ_0 radians using a standard 2D rotation

matrix, and it is then moved upwards a distance of h_0 by simply adding this value to the y-component of the particle's position. The particle's value for the original position is also updated to reflect the new starting position.

Once every particle is connected, the simulation is advanced by first calculating the force on each particle, then updating the position and velocity of each particle. For the following equations, particle i refers to the current particle in the list of all particles, and particle j refers to the current particle in particle i's list of connected particles. The force acting on particle i at time t ($\vec{f}_{i,t}$) is given by Eq. 2, where $\vec{f}_{k,i,j}$ is the spring force on particle i corresponding to particle j, $\vec{f}_{c,i,j}$ is the damping force on particle i corresponding to particle j, $\vec{f}_g$ is the force due to gravity, $\vec{f}_{air}$ is the force due to air resistance, and $\vec{f}_c$ is the contact force.

$$\vec{f}_i = \sum \vec{f}_{k,i,j} + \sum \vec{f}_{d,i,j} + \vec{f}_g + \vec{f}_{air} + \vec{f}_c \quad (2)$$

The spring force and damping force both act in the direction between the particles ($\hat{n}_{i,j}$), as defined in Eq. 3, where $\vec{x}_{i,t}$ is the current position of particle i and $\vec{x}_{j,t}$ is the current position of particle j.

$$\hat{n}_{i,j} = \frac{\vec{x}_{i,t} - \vec{x}_{j,t}}{||\vec{x}_{i,t} - \vec{x}_{j,t}||} \quad (3)$$

The spring force on particle i is given by Eq. 4, where k_{ij} is the stiffness of the link between particles i and j, $\vec{x}_{i,0}$ is the initial position of particle i, and $\vec{x}_{j,0}$ is the initial position of particle j. For our implementation, k_{ij} is a constant for all links, but it could be specified individually per link.

$$\vec{f}_{k,i,j} = -k_{ij}(||\vec{x}_{i,t} - \vec{x}_{j,t}|| - ||\vec{x}_{i,0} - \vec{x}_{j,0}||)\hat{n}_{i,j} \quad (4)$$

The damping force on particle i is given by Eq. 5, where c_{ij} is the damping coefficient of the link between particles i and j, $\vec{v}_{i,t}$ is the current velocity of particle i, and $\vec{v}_{j,t}$ is the current velocity of particle j. Again, for our implementation, c_{ij} is a constant for all links, but it could be specified individually per link.

$$\vec{f}_{d,i,j} = -c_{ij}(\hat{n}_{i,j} \cdot \vec{v}_{i,t} - \hat{n}_{i,j} \cdot \vec{v}_{j,t})\hat{n}_{i,j} \quad (5)$$

The force due to gravity was assumed to be a constant of $-9.81m_i$ in the y direction for all time steps, where m_i is the mass of particle i. Viscous damping due to air resistance is added to more accurately simulate the real world, and is given in Eq. 6, where $c_{air,i}$ is the drag coefficient for particle i. Again, this is set to a constant value for simplicity in our implementation, but it could be adjusted on a per particle basis to reflect differences in surface roughness or other drag related properties.

$$\vec{f}_{air,i} = c_{air,i}\vec{v}_{i,t} \quad (6)$$

With each of these forces accounted for, the conditions for contact are then checked to determine if it is necessary to add additional contact forces to particle i at this time step. In our simple simulation of an object falling to the ground, contact is assumed when the vertical position of particle i is less than or equal to 0. However, this formulation of contact could be handled for an arbitrary number of contact surfaces of arbitrary orientation. One would simply have to check for contact and apply the equations below for each surface. The formulae below are expressed in general form, with the direction of the contact normal being $\hat{n}_c$ and the direction perpendicular to that being $\hat{n}_t$. The contact force acting on particle i is given by Eq. 7, where $\vec{f}_N$ is the normal force, $\vec{f}_f$ is the force of friction, and $\vec{f}_i$ is an impulsive force.

$$\vec{f}_{c,i} = \vec{f}_N + \vec{f}_f + \vec{f}_i \quad (7)$$

An expression for $\vec{f}_N$ is given by Eqn. 8, and an expression for $\vec{f}_f$ is given by Eqn. 9, where $\vec{f}$ is the net force calculated on the particle before adding the contact forces and μ_i is the coefficient of friction between the contact surface and particle i.

$$\vec{f}_N = (\hat{n}_c \cdot (-\vec{f}))\hat{n}_c \quad (8)$$

$$\vec{f}_f = -sign((\hat{n}_t \cdot \vec{v}_{i,t})\hat{n}_t)\mu||\vec{f}_N||\hat{n}_t \quad (9)$$

The impulsive force is calculated such that the speed at the next time step will be rebounding from the point of contact, according the impulse momentum theorem. A formula for the velocity of particle i at the next time step ($\vec{v}_{i,t+1}$) is given in Eq. 10, where ϵ is the coefficient of restitution and, Δt is the time step length for the simulation. A formula for the corresponding impulsive force is

given in Eq. 11.

$$\vec{v}_{i,t+1} = -\epsilon(\hat{n}_c \cdot \vec{v}_{i,t})\hat{n}_c + (\hat{n}_t \cdot \vec{v}_{i,t})\hat{n}_t \tag{10}$$

$$\vec{f}_i = \frac{m(\vec{v}_{i,t+1} - \vec{v}_{i,t})}{\Delta t} \tag{11}$$

With the above equations, the force on particle i can be calculated for each time step. Once the force is calculated, the position and velocity of the particle can be updated according to Eq. 12 and Eq. 13, where t - 1 refers to the value of a vector at the previous time step, which would be the value of that vector used to determine the forces above. The $\frac{\vec{f}_t}{m}$ term comes from Newton's second law applied to particle i with the force calculated above. At least within the bounds of this project, everything really does come back to $\vec{f}_{net} = m\vec{a}$.

$$\vec{v}_{i,t} = \vec{v}_{i,t-1} + \frac{\vec{f}_t}{m}\Delta t \tag{12}$$

$$\vec{x}_i = \vec{x}_{i,t-1} + \vec{v}_{i,t}\Delta t \tag{13}$$

With these equations in hand, MSM systems of particles can be simulated from t_0 to an arbitrary end time by finding the force acting on each particle then updating the state of each particle for each time step. This method of updating the state space is fairly standard, and could be integrated with a controller without much change to the model itself.

IV. Experiments

Using the MSM model described above, the simulation was run with a variety of parameters. The parameter set tested was far from exhaustive, but some general trends could be gleaned from the study.

A. Bouncing Tests

The goal of the project was to simulate bouncing objects, so the first tests involved loading in different objects drawn in Adobe Photoshop™. The results of a test run with the parameters shown in Table I are shown in Fig. 3 for a square, for a beam shape, and for a very coarse Michigan University "M" logo. For these full animations, as well as others, see the Appendix.

TABLE I
Parameters used for each trial.

Parameter	Value	Unit
k	5000	$\frac{N}{m}$
m	0.2	kg
c	10	$\frac{Ns}{m}$
c_{air}	0.9	$\frac{Ns}{m}$
ϵ	0.7	–
μ	0.6	–
θ_0	0.5	rad
h_0	1	m
s	0.15	m
Δt	0.001	s

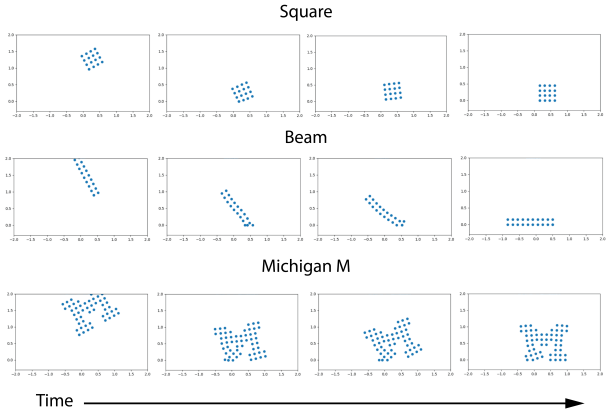

Fig. 3. Some frames illustrating key moments in each simulation are shown above, with time increasing to the right.

As seen in Fig. 3, the objects generated by the simulator crumple and bend as one might expect. It may be difficult to see, but the square actually bounced up a bit after contact, while the beam and M dissipated a lot of energy in bending and did not bounce nearly as much.

In addition to these tests with constant parameters, trials were also run while varying the parameters. A complete quantitative analysis of the effects of varying each parameter is beyond the scope of this report just due to the shear number of them. However, some general trends were observed as follows. For high values of k (greater than 100,000), high values of c (greater than 100), and low values of m (lower than 0.1), the simulation would break.

We suspect that the minimum viable time step depends on the ratio of k and m to some degree, and that by decreasing the time step higher stiffness to mass ratios could be tested. However, as stiffness increases, the system resembles a rigid body more and more, so luckily the simulation still works fine for the lower stiffness values that will cause the system to exhibit the deformation this project sought to study. As one might expect, objects bounced higher for higher values of ϵ and higher drop heights, the objects accelerated less for higher values of c_{air}, and the vibrations ceased sooner for higher values of c. Lower values of μ caused objects to slide more readily. Altering the starting angle θ_0 greatly affected the observed dynamics, as it would affect which particle experienced the first contact as well as what sort of torque that contact would exhibit on the system. It was satisfying to observe this "torque" effect, as there is no actual implementation for torque in the equations, but it emerges anyway, likely due to the lattice structure of the mesh.

B. Timing Tests

The algorithm involves many iterations over many particles, and as the particle count got higher, the time it took to run was non trivial, so it made sense to characterize the run time of the algorithm for various numbers of particles. To this end, square objects of dimensions increasing from 1x1 to 10x10 were dropped with the same parameters, and the time it took to execute the calculations for each time step was measured and added up over the course of each run.

The results showing the execution time versus number of particles are shown in Fig. 4. The relationship looked to be roughly linear for n between 1 and 100, and as such a linear regression was performed using Microsoft Excel™. This yeilded a best fit equation for the data, show in in Eq. 14, where n is the number of particles and $t_{execution}$ is the execution time. This regression had an R^2 value of 0.9892, suggesting that the data is very linear, at least in the studied region. This linear behavior makes sense, as there is essentially a constant number of operations to do to calculate update each particle in each iteration.

$$t_{execution} = 0.8454n - 1.653 \tag{14}$$

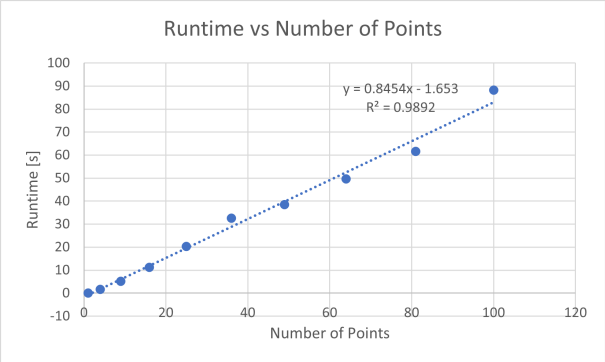

Fig. 4. The execution time of the function is graphed versus the number of particles in the simulation.

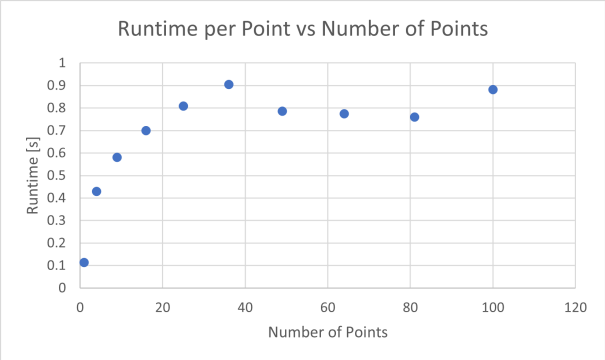

Fig. 5. The execution time of the function normalized w.r.t the number of particles is graphed versus the number of particles in the simulation.

Additionally, the execution time per number of particles was graphed against the number of particles, as seen in Fig. 5. The relationship was notably non-linear. We expect that this is due to the hybrid nature of the system, as with larger numbers of particles, more particles were likely to be inside the object, and therefore never would contribute the added run time needed to resolve contact. 5.

V. Discussion

Overall, we would classify this project as a success. The goal of simulating the bouncing behavior of deformable objects was achieved, and it was satisfying to watch as the objects bounced and vibrated during the simulations. That said, there are many things that could be improved upon. The method for calculating the discrete integrations is known as explicit Euler integration [4], and though it was simple to implement, it was likely part of

the reason for the degradation of the simulation for larger time steps. Contact between particles was not explored, but could be implemented in much the same way as contact between a particle and the ground plane. Additionally, the ground plane itself could be simulated as another object, allowing for simulations of trampoline-like interactions. The parameters were set to values that seemed to yield interesting animations, but in reality tuning these to accurately reflect real world conditions would be a project in and of itself. Perhaps the spring constants could be approximated from the material properties [5], and the friction and restitution could be tested as well to yield more accurate results for real objects. Additionally, nonlinear spring behavior near the material's yield point could be explored, as could shear forces splitting an object into two separate objects.

Generating a png image was convenient for our testing, but in a real manipulation pipeline a method to directly calculate a numpy array of the particle positions from visual feedback would be more useful. For large numbers of particles the simulation runs slower than real time, so optimizations would need to happen to enable it to operate for part of a controller.

Additionally, small aesthetic changes to the rendering would have been helpful, such as drawing a line to show the ground and drawing lines to show the meshes or object boundary. Also, because the focus was not on configurability of the starting configuration, a more complicated method to rotate the object about its centroid rather than the origin was not employed, although such an approach could be useful to implement in the future.

VI. Conclusion

References

[1] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[2] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[3] M. Nagurka and S. Huang, "A mass-spring-damper model of a bouncing ball," vol. 22, 01 2004, pp. 499 – 504 vol.1.

[4] F. F. K. D. S. B. Arriola-Rios VE, Guler P and W. JL, "Modeling of deformable objects for robotic manipulation: A tutorial and review," *Frontiers in Robotics and A.I.*, vol. 7, no. 82, 2020.

[5] B. Lloyd, G. Székely, and M. Harders, "Identification of spring parameters for deformable object simulation," *IEEE transactions on visualization and computer graphics*, vol. 13, pp. 1081–94, 10 2007.

Appendix

The full source code, gif files of the animations discussed in this report, and more can be found at the following google drive link: https://drive.google.com/drive/folders/1RTjQuMoAETEDZAcUif9HSFP25WmYXKkn?usp=sharing