ADMPM

Kyle Perez

April 23, 2021

The material point method MPM is a powerful method that allows us to model continuous systems by splitting an object up into various small sections of continua, which we explore the dynamics of, then interpolate its density, velocity, etc to static nodes.

What follows serves to document the $ADMPM$ code, developed by me, Kyle Perez. It is not a particularly advanced or well-developed code meant to process complicated geometries, powerful shocks, or similarly complicated topics. Rather, it focuses on modeling simple 1D elastic solids.

The program is modular; with initial conditions, boundary conditions, and simulation properties able to be modified, alongside the core functionality of the program: its capabilities to model elastic solids, ready to be changed to model ideal gasses.

However, the system takes a close-to-the-metal approach to running problems; source code must be modified and recompiled in order to run new problems, or the same problem with new parameters. My hope is that it inspires you to make modifications to the program itself, so you can further optimize it or extend its functionality.

The code itself can be found at `https://github.com/kyleperez-rice/ADMPM`, which includes the source code written in C and Java, in addition to the Python files used to create the plots.

Finally, I would like to thank Dr. Duan Zhang, my advisor at Los Alamos National Laboratory, for helping to guide me through the material point method and providing feedback when needed; without his help, the code would be in a much rougher state, with plenty of runtime ready to be shaved off. Of course, I must also thank the various anonymous programmers of the Internet, who have encountered many of the same issues and errors I have encountered throughout the assembly of this project, and have provided and archived solutions on the Internet, for the access and use of all.

Contents

1 Introduction

The Material Point Method (MPM) is a method of numerically solving the differential equations that describe continuous systems.

Examples of continuous systems include fluids and solids, the dynamics of which are guided by pervasive nonlinearities that prevent description of analytic solutions to the equations of motion except in a special few situations (and often those special situations are themselves subject to approximations).

Indeed, being able to model the nonlinear equations of motion with a computer not only allows comparison the analytic solution to the numerically obtained solution (in addition to comparison with experiments) to validate the code, but it also allows people to forward into the unknown and explore systems that analytic solutions and experiments have not yet touched.

The procedures used to implement this method have been adopted from Dr. Duan Zhang's paper on the topic[2], with the particular restriction that the following program is not built to handle multiphase systems, but rather restricts itself to a single material.

As a final matter of notation, in general, when referencing nodes, this documentation will use the index l, and with material points, the index p. The distinction is very important, and there will be some amount of work to prevent confusion in the matter of indices.

With this in mind, it is time to examine the material point method.

1.1 Prelude

Many continuous systems can be fully described with three equations: a continuity equation which describes how mass is conserved in the system, an Euler-type equation that describes how the velocity of the system changes in time, and a constitutive relation or equation of state that describes the internal force of the system (the stress).

For example, a linearly elastic solid is described by the following equations of motion[1]:

$$\frac{\partial \rho}{\partial t} + \partial_i(\rho v_i) = 0 \tag{1.1a}$$

$$\rho(\frac{\partial v_i}{\partial t} + v_j \partial_j v_i) = \partial_j \sigma_{ij} + \rho f_i \tag{1.1b}$$

$$\sigma_{ij} = E_{ijkl}\epsilon_{kl} = \frac{1}{2}E_{ijkl}(\partial_j u_i + \partial_i u_j) \tag{1.1c}$$

Where $v_i = \frac{\partial u_i}{\partial t}$. ρ is the density is of the system, u_i is the displacement in some direction of the system, v_i is the velocity in some direction, f_i is the external force of the system, σ_{ij} is the stress tensor of the system, and E_{ijkl} is the Elasticity Tensor.

It is easy to see that these systems correspond to 10 nonlinear coupled differential equations, and it is likely that this system is far outside of the world's current mathematical ability to solve.

This is where MPM comes in; it provides a method to solve these equations.

The crux of MPM will be to enforce that the solutions to the equations of motion will be in a certain form, in particular that:

$$q(x,t) = \sum_{l:\text{ nodes}} q_l(t) S_l(x) \tag{1.2}$$

Where q is some arbitrary quantity, such as density or velocity. $S_l(x)$ is called the shape function of a node l, and q_l is the node's given quantity.

So, a mesh must be created for the system; a mesh is just a set of nodes. Each node has a position, and holds information about the various quantities it models. The nodes act as a sampling of the function over an area/volume, and it is clear that the more nodes added, the more accurately the system is modeled.

The volume between the nodes (in 1D, it is just the space between two neighboring nodes, and in 2D, it is the area enclosed by the quadrilateral formed by 4 nearby nodes) will be populated by material points, which are small bits of material that model the continua.

The Material Point Method will evolve the material points in time, and use the material point quantities to update the node quantities. In this way, the simulation revolves primarily around the material points, and not the nodes.

Indeed, the q values at a material point, and those at a node are related via

$$q_p = \sum_{l:\text{ nodes}} q_l S_l(x_p) \tag{1.3a}$$

[1]This document uses Einstein Summation Notation; summation over repeated indices is implied

$$q_l = \frac{1}{m_l} \sum_{p:\ \text{particles}} q_p m_p S_l(x_p) \tag{1.3b}$$

Where m_l and m_p are the respective masses of the nodes and particles.

In essence, the goal of MPM is to evolve the particle/material point quantities, and then use them to show what the node quantities are (which can be used to visualize the system).

So, without further ado, it is time to see how to evolve the material points.

1.2 The Continuity Equation

Since the whole system is being modelled with a whole host of material points, which are individual particles, the following equation needs to be solved for each particle p:

$$\frac{\partial \rho_p}{\partial t} + \nabla \cdot (\rho_p \vec{v}_p) = 0 \tag{1.4}$$

The density of the particles do not change in time; the continua has been split into a bunch of particles, and while they are free to move around, ultimately cannot change their mass. These particles are also not able to change their volume (although they can become deformed). These are fundamental constraints that are imposed on the particles.

With this, the continuity equation is immediately satisfied. This is much like in standard Newtonian Physics how there is no need to worry about solving the continuity equation when solving the 2-body problem, a mass on a spring, etc; it is satisfied vacuously. Similarly, splitting up the continua into a bunch of particles has made it so that there is no need to worry about the continuity equation!

1.3 Euler's Equation

Euler's Equation is of the form (using $\frac{d}{dt} = \frac{\partial}{\partial t} + \vec{v} \cdot \nabla$)

$$\rho \frac{dv_i}{dt} = \nabla \cdot \sigma_i + \rho f_i \tag{1.5}$$

It is this equation that needs focus on solving, as it displays the most interesting behaviour given its convective $(\vec{v} \cdot \nabla)\vec{v}$ term.

So, the goal is to develop a method to find an approximate solution to this equation using the material point method.

First, multiplying each side by some function h_i and integrating it across the entire volume:

$$\int_V \rho \frac{dv_i}{dt} h_i dV = \int_V (\nabla \cdot \sigma_i + f_i) h_i dV \tag{1.6}$$

And then, using the approximation set forward from the beginning:

$$q_i(x,t) = \sum_{j:\text{ nodes}} q_{ij}(t) S_j(x) \tag{1.7}$$

Applying this to the velocity,

$$\int_V (\rho \sum_j \frac{\partial v_{ij}}{\partial t} S_j(x) + v_{ij}(t) \vec{v} \cdot \nabla S_j(x)) h_i dV = \int_V (\nabla \cdot \sigma_i + \rho f_i) h_i dV \tag{1.8}$$

Now, so far this has been done for an arbitrary h_i. So, picking

$$h_i = \sum_l \delta v_{il} S_l(x) \tag{1.9}$$

That is, h_i is something like a small change in the velocity.

Under this, everything becomes

$$\sum_j \sum_l \int_V \rho \frac{\partial v_{ij}}{\partial t} S_j(x) S_l(x) + \rho v_{ij} \vec{v} \cdot \nabla S_j(x) S_l(x) dV \delta v_{il} = \sum_l \int_V (\nabla \cdot \sigma_i + \rho f_i) S_l(x) dV \delta v_{il} \tag{1.10}$$

Integration by parts, index manipulation, and the fact that the shape functions all eventually go to 0 means that the second term on the left goes to 0. Moreover, since v_{ij} is only a function of time, then $\frac{\partial v_{ij}}{\partial t} = \frac{dv_{ij}}{dt}$ and the result simplifies to

$$\sum_j \sum_l (\int_V \rho S_j(x) S_l(x) dV) \frac{dv_{ij}}{dt} \delta v_{il} = \sum_l \int_V (\nabla \cdot \sigma_i + \rho f_i) S_l(x) dV \delta v_{il} \tag{1.11}$$

On the right hand side, using integration by parts/the Divergence theorem:

$$\sum_j \sum_l (\int_V \rho S_j(x) S_l(x) dV) \frac{dv_{ij}}{dt} \delta v_{il} = \sum_l (\int_V \rho f_i S_l(x) dV + \int_{\partial V} \sigma_i S_l(x) d\vec{A} - \int_V \sigma_i \cdot \nabla S_l(x) dV) \delta v_{il} \tag{1.12}$$

Making the definition:

$$m_{jl} = \int_V \rho S_j(x) S_l(x) dV \tag{1.13}$$

And thus

$$\sum_j \sum_l m_{jl} \frac{dv_{ij}}{dt} \delta v_{il} = \sum_l (\int_V \rho f_i S_l(x) dV + \int_{\partial V} \sigma_i S_l(x) d\vec{A} - \int_V \sigma_i \cdot \nabla S_l(x) dV) \delta v_{il} \tag{1.14}$$

Next, using the fact that δv_{il} was picked to be arbitrary. Due to this, the sum over l can be removed and then remove the δv_{il} altogether, obtaining

$$\sum_j m_{jl} \frac{dv_{ij}}{dt} = \int_V \rho f_i S_l(x) dV + \int_{\partial V} \sigma_i S_l(x) d\vec{A} - \int_V \sigma_i \cdot \nabla S_l(x) dV \tag{1.15}$$

Considering that there are n l values (IE: n nodes), this corresponds to a total of n linearly coupled equations that allow solves for $\frac{dv_{ij}}{dt}$ for any j, and it's just a matter of calculating all of these integrals.

However, attempting to use this equation in a large-scale simulation, a program would have to deal with absolutely enormous matrices (where most of the entries are 0, especially true for linear shape functions).

So, to save on computational time, without sacrificing too much accuracy, an approximation that these shape functions are tightly bound can be made and thus

$$m_l = \int_V \rho S_l(x) dV \approx m_{jl} \tag{1.16}$$

This decouples the derivatives of the velocity at the nodes:

$$m_l \frac{dv_{il}}{dt} = \int_V \rho f_i S_l(x) dV + \int_{\partial V} \sigma_i S_l(x) d\vec{A} - \int_V \sigma_i \cdot \nabla S_l(x) dV \tag{1.17}$$

There is more to be done. Applying $f_i = \sum_j f_{ij} S_l(x)$ to the system:

$$\int_V \rho f_i S_l(x) dV = \sum_j \int_V \rho S_j S_l dV f_{ij} \tag{1.18}$$

And applying similar approximations to what has been done above, the external force term simplifies, and comes to

$$\frac{dv_{il}}{dt} = f_{il} - \frac{1}{m_l} \int_V \sigma_i \cdot \nabla S_l(x) dV + \frac{1}{m_l} \int_{\partial V} \sigma_i S_l(x) d\vec{A} \tag{1.19}$$

So far, the fact that the system has been split up into material points (and not just nodes) has not been used. Using the material points to simplify the remaining volume integral. In particular, writing

$$\int_V \sigma_i \cdot \nabla S_l dV \approx \sum_{p:\ \text{mps}} \sigma_{ip} \cdot \nabla S_l(x_p) V_p \tag{1.20}$$

That is, the volume integral can be approximated as a finite sum over the material points, where σ_{ip} are the relevant components of the stress tensor for the material points, x_p is the position of a particle, and V_p is the volume of a particle.

Applying a very similar approximation to the value of m_l, so that

$$m_l = \sum_p m_p S_l(x_p) \tag{1.21}$$

Finally, for the purposes of this program, the boundary term will be ignored (boundary conditions will also be fixed separately), and thus for the ith component of velocity for a node l:

$$\frac{dv_{il}}{dt} = f_{il} - \frac{1}{m_l} \sum_{p:\text{ particles}} \sigma_{ip} \cdot \nabla S_l(x_p) V_p \tag{1.22a}$$

$$m_l = \sum_{p:\text{ particles}} m_p S_l(x_p) \tag{1.22b}$$

Of course, the goal of the simulation is to evolve the material points, and then use them to build the node velocity. So, why was $\frac{dv_{il}}{dt}$ computed instead of the equivalent for the particles?

Well, $q_p = \sum_l q_l S_l(x_p)$, so picking to evolve our material point velocities as

$$v_{ip}(t + dt) = v_{ip}(t) + \frac{dv_{ip}}{dt} dt \tag{1.23}$$

Then

$$v_{ip}(t + dt) = v_{ip}(t) + \sum_l \frac{dv_{il}}{dt} S_l(x_p) dt \tag{1.24}$$

So really, calculating $\frac{dv_{il}}{dt}$ for all nodes (which only depends on the particles and the external force), then the new particle velocities can be found

Using the standard node velocity v_{il} and introducing the Lagrangian node velocity v_{il}^L defined as

$$v_{il}^L \equiv v_{il} + \frac{dv_{il}}{dt} dt \tag{1.25}$$

Then

$$v_{ip}(t + dt) = v_{ip}(t) + \sum_l (v_{il}^L - v_{il}) S_l(x_p) \tag{1.26}$$

However, more than this, the particle need to move. Given that the material points are Lagrangian in nature[2], their position evolves as

$$x_p(t + dt) = x_p(t) + v_{ip}^L dt = x_p(t) + \sum_l v_{il}^L S_l(x_p) dt \tag{1.27}$$

So, there is now a method to evolve the velocity of the particles in time, and to move the particles in time.

[2]This is to say that they do not remain fixed relative to the boundaries; fixed Eulerian nodes and Lagrangian particles exhibit different behaviour.

With this, using the following relations to compute the updated node velocity and node mass; both being needed to advance the simulation a further timestep:

$$v_l(t+dt) = \frac{1}{m_l(t+dt)} \sum_p m_p v_p(t+dt) S_l(x_p(t+dt)) \tag{1.28a}$$

$$m_l(t+dt) = \sum_p m_p S_l(x_p(t+dt)) \tag{1.28b}$$

Finally, computing a node's density by just dividing its mass by its length:

$$\rho_l(t+dt) = \frac{m_l(t+dt)}{V_l} \tag{1.29}$$

So, with the Euler equation, and basic identities, the velocity and density of the nodes can be evolves (and it is these quantities that need to be recorded).

However, advancing to the next timestep required the stress at the current time. This implies that the stress needs to also be evolved. This step needs to be visited.

1.4 Constitutive Relations

In general constitutive relations relate the stress of the system to other variables in the system.

Examples of such relations are

$$\sigma_{ij} = E_{ijkl}\epsilon_{kl} \text{ Linear Elasticity} \tag{1.30a}$$

$$\sigma_{ij} = -p\delta_{ij} = -(\gamma - 1)\epsilon\rho\delta_{ij} \text{ Ideal Gas} \tag{1.30b}$$

Much like with the other variables of the system, the relation between particle stress and node stress is:

$$\sigma_{ijp}(t) = \sum_l \sigma_{ijl}(t) S_l(x_p) \tag{1.31}$$

Interestingly, the if the node stress is not cared about for analysis, the nodes only need their stress calculated once, since the node velocity is advanced in time using the particle stress. This is useful for saving on memory.

From here though, the methods of evolving the stress are varied, and depend on the equation of state at hand.

1.4.1 Linear Elasticity

The fundamental equation of state for linear elasticity (for a particle) reads:

$$\sigma_{ijp} = E_{ijkl}\epsilon_{klp} \tag{1.32}$$

This is to say, the strain of the particle ϵ_{klp} is used to build the stress. The strain and stress have a linear relationship. As a matter of simplicity, the Young's Modulus Tensor is assumed to be a constant.

However, using the fact that the strain tensor is defined as

$$\epsilon_{klp} = \frac{1}{2}(\frac{\partial u_{kp}}{\partial x_l} + \frac{\partial u_{lp}}{\partial x_k}) \tag{1.33}$$

Then, noting that

$$\frac{\partial \epsilon_{klp}}{\partial t} = \frac{1}{2}(\frac{\partial v_{kp}}{\partial x_l} + \frac{\partial v_{lp}}{\partial x_k}) \tag{1.34}$$

That is, the rate of change of the strain tensor is related to how the particle velocity changes in space.

However, writing:

$$v_{ip}(x,t) = \sum_l v_{il} S_l(x) \tag{1.35}$$

And thus plugging in this definition into the time derivative of the strain tensor,

$$\frac{\partial \epsilon_{klp}}{\partial t} = \frac{1}{2} \sum_{n:\text{ nodes}} v_{kn} \frac{\partial S_n}{\partial x_l}(x_p) + v_{ln} \frac{\partial S_n}{\partial x_k}(x_p) \tag{1.36}$$

This means that the derivatives of the shape functions only need to be computed, instead of derivatives of the node velocity. Given that the shape functions are picked means that they should be well known. Note that in 1D, the notation is much simpler (just $\frac{\partial \epsilon}{\partial t} = \sum_n v_n \frac{\partial S_n}{\partial x}(x_p)$).

With this in mind, evolving the strain in time using Euler's Method:

$$\epsilon_{klp}(t+dt) = \epsilon_{klp}(t) + \frac{dt}{2} \sum_{n:\text{ nodes}} v_{kn} \frac{\partial S_n}{\partial x_l}(x_p) + v_{ln} \frac{\partial S_n}{\partial x_k}(x_p) \tag{1.37}$$

And thus by the equation of state:

$$\sigma_{ijp}(t+dt) = E_{ijkl}\epsilon_{klp}(t+dt) \tag{1.38}$$

So, the particle stress has been evolved in time (and the strain, which is a more observable quantity).

The updated stress/strain can be sent to the nodes via

$$q_l = \frac{1}{m_l} \sum_p m_p q_p S_l(x_p) \tag{1.39}$$

1.4.2 Ideal Gas

So, the equation of the state for an ideal gas can be compressed better as

$$p = (\gamma - 1)\epsilon\rho \tag{1.40a}$$

$$\sigma_{ij} = -p\delta_{ij} \tag{1.40b}$$

Now, consider that for an ideal gas, at $t = 0$ the pressure and density are known initially. This means that the equation of state can be used to get ϵ: the internal energy per unit mass (not to be confused with the strain; which does not exist in a fluid).

Initializing the system as normal, passing down the pressure, internal energy, and so on to the nodes and then to the particles.

And now, the goal is to calculate how p for a particle changes in time.

For one, it is known that

$$\frac{\partial p_p}{\partial t} = (\gamma - 1)\frac{\partial \epsilon_p \rho_p}{\partial t} \tag{1.41}$$

However, an apparent problem comes up: the internal energy needs to be evolved in time.

It turns out that there is the following equation, as per the FLASH user manual[1]:

$$\frac{\partial \rho\epsilon}{\partial t} + \nabla \cdot ((\rho\epsilon + p)\vec{v}) - \vec{v} \cdot \nabla p = 0 \tag{1.42}$$

This same equation should apply for the particles, and thus

$$\frac{\partial \rho_p \epsilon_p}{\partial t} = -\nabla \cdot ((\rho_p \epsilon_p + p_p)\vec{v}_p) + \vec{v}_p \cdot \nabla p_p \tag{1.43}$$

Note that the density of our particles is constant in time and space, and is just $\frac{m_p}{V_p}$, so it can better be put as

$$\frac{\partial \epsilon_p}{\partial t} = -\nabla \cdot (\epsilon_p \vec{v}_p) - \frac{V_p}{m_p}\nabla \cdot (p_p \vec{v}_p) + \frac{V_p}{m_p}\vec{v}_p \cdot \nabla p_p \tag{1.44}$$

Which simplifies to

$$\frac{\partial \epsilon_p}{\partial t} = -\nabla \cdot (\epsilon_p \vec{v}_p) - \frac{V_p}{m_p} p_p \nabla \cdot \vec{v}_p \tag{1.45}$$

From here, abusing the fact that $q_p = \sum_l q_l S_l(x_p)$,

$$\frac{\partial \epsilon_p}{\partial t} = -\sum_l \epsilon_l \vec{v}_l \cdot \nabla S_l(x_p) + \frac{V_p}{m_p} p_p \vec{v}_l \cdot \nabla S_l(x_p) \tag{1.46}$$

Which finally simplifies to

$$\frac{\partial \epsilon_p}{\partial t} = -\sum_l (\epsilon_l + \frac{V_p}{m_p} p_p)\vec{v}_l \cdot \nabla S_l(x_p) \tag{1.47}$$

From here, using Euler's Method:

$$\epsilon_p(t + dt) = \epsilon_p(t) - dt \sum_l (\epsilon_l + \frac{V_p}{m_p} p_p)\vec{v}_l \cdot \nabla S_l(x_p) \tag{1.48}$$

And from here just using the equation of state again

$$p_p(t+dt) = (\gamma - 1)\frac{m_p}{V_p}\epsilon_p(t+dt) \tag{1.49}$$

It follows that

$$\sigma_{ijp}(t+dt) = -p_p(t+dt)\delta_{ij} \tag{1.50}$$

Thus the stress has been evolved in time.

Note that in order to evolve the stress, the pressure had to be evolved, but in order to evolve the pressure, the internal energy needed to be evolved.

Also, though the use of continued substitution of the equation of state into the internal energy evolution equation, it follows that

$$\frac{\partial p_p}{\partial t} = (\gamma - 1)\vec{v}_p \cdot \sum_l p_l \nabla S_l(x_p) - \gamma \sum_l p_l \vec{v}_l \cdot \nabla S_l(x_p) \tag{1.51}$$

1.5 The Shape Functions

So far, shape functions have been mentioned in passing, but have not delved into. So, it is time to describe them in further detail.

The shape functions of our system form a Partition of Unity, which implies that

$$\sum_l S_l(x) = 1 \tag{1.52}$$

That is, summing up all of the shape functions (using x as a generic variable; they may very well be functions of x, y, z) nets 1.

With this in mind, there are an infinite amount of these shape functions that can be picked and used in the program (at least in theory). However, given that these functions have to be programmed in, they should constructed these in a simple method that allows them to be easily constructed for each given node, given an arbitrary number of nodes.

To construct simple, uniform shape functions, it seems that picking them to be polynomials is a great start. However, the shape functions should be localized for performance reasons. This means that the shape functions need to be 0 past a certain point. It is obvious that this can be accomplished using polynomials.

However, given the fairly crude and simple nature of this program, the use of linear shape functions has been adopted, that is shape functions that are described by lines. In particular, functions have been picked that go to 0 at the nodes right next to our current node.

So, the following conditions at a node with a position x_l apply:

$$S_l(x_l) = 1 \tag{1.53a}$$

$$S_l(x_{l-1}) = S_l(x_{l+1}) = 0 \tag{1.53b}$$

Now, using these conditions to construct lines between the given x points,

$$S_l(x) = \begin{cases} \frac{x - x_{l-1}}{x_l - x_{l-1}} & x_{l-1} < x \leq x_l \\ \frac{x_{l+1} - x}{x_{l+1} - x_l} & x_l < x < x_{l+1} \\ 0 & \text{Otherwise} \end{cases} \tag{1.54}$$

This works well for a mesh that is not spaced equally, however, enforcing an evenly spaced mesh, then $x_l - x_{l-1} = dx$, where dx is the spacing between nodes. Similarly, $x_{l\pm1} = x_l \pm dx$ and thus the shape functions to be used are

$$S_l(x) = \begin{cases} 1 + \frac{x - x_l}{dx} & x_l - dx < x \leq x_l \\ 1 - \frac{x - x_l}{dx} & x_l < x < x_l + dx \\ 0 & \text{Otherwise} \end{cases} \tag{1.55}$$

Note how now these shape functions depend only on x, x_l, and dx, which correspond to a simple variable, a property of the node, and a global variable. In this regard, there is no need to worry about passing other nodes as arguments to the shape functions when it comes to programming them in.

Given the shape functions, it has been shown that there is a need for their derivatives. Following a trivial calculation:

$$\frac{\partial S_l}{\partial x}(x) = \begin{cases} \frac{1}{dx} & x_l - dx < x \leq x_l \\ -\frac{1}{dx} & x_l < x < x_l + dx \\ 0 & \text{Otherwise} \end{cases} \tag{1.56}$$

Note how this derivative can be programmed directly into the code, instead of numerically calculating it, which will provide extra accuracy.

However, the shape functions chosen has not actually been shown to actually be shape functions. This should be proven. Observe that

$$S_{l-1}(x) + S_l(x) = \begin{cases} 1 + \frac{x - x_{l-1}}{dx} & x_{l-1} - dx < x \leq x_{l-1} \\ 2 - \frac{x - x_l}{dx} + \frac{x - x_{l-1}}{dx} & x_{l-1} < x < x_l \\ 1 - \frac{x - x_l}{dx} & x_l < x < x_{l+1} \\ 0 & \text{Otherwise} \end{cases} \tag{1.57}$$

Simplifying to

$$S_{l-1}(x) + S_l(x) = \begin{cases} 1 + \frac{x - x_{l-1}}{dx} & x_{l-1} - dx < x \leq x_{l-1} \\ 2 - \frac{x_l - x_{l-1}}{dx} & x_{l-1} < x < x_l \\ 1 - \frac{x - x_l}{dx} & x_l < x < x_{l+1} \\ 0 & \text{Otherwise} \end{cases} \tag{1.58}$$

And by $x_l - x_{l-1} = dx$,

$$S_{l-1}(x) + S_l(x) = \begin{cases} 1 + \frac{x - x_{l-1}}{dx} & x_{l-1} - dx < x \leq x_{l-1} \\ 1 & x_{l-1} < x < x_l \\ 1 - \frac{x - x_l}{dx} & x_l < x < x_{l+1} \\ 0 & \text{Otherwise} \end{cases} \tag{1.59}$$

From here, using $S_{l-2} + S_{l-1} + S_l + S_{l+1}$, things predictably simplify to

$$S_{l-2} + S_{l-1} + S_l + S_{l+1} = \begin{cases} 1 + \frac{x - x_{l-1}}{dx} & x_{l-2} - dx < x \leq x_{l-2} \\ 1 & x_{l-2} < x < x_{l+1} \\ 1 - \frac{x - x_l}{dx} & x_{l+1} < x < x_{l+2} \\ 0 & \text{Otherwise} \end{cases} \tag{1.60}$$

So, the more shape functions that get added together, the larger the interval where the sum is equal to 1 gets. From here, picking x_0 and x_n being the first and last nodes of the interval $[a, b]$, in particular with $x_0 = a$ and $x_n = b$,

$$\sum_l S_l(x) = \begin{cases} 1 + \frac{x - x_0}{dx} & x_0 - dx < x < x_0 \\ 1 & x_0 \leq x \leq x_n \\ 1 - \frac{x - x_n}{dx} & x_n < x < x_n + dx \\ 0 & \text{Otherwise} \end{cases} \tag{1.61}$$

So, being restricted to the interval between x_0 and x_n, the shape functions sum to 1 on that interval, and thus the assertion made that $q(x,t) = \sum_l q_l(t) S_l(x)$ is a fair approximation on the interval. There is just a need to make sure that the material points do not leave the area between x_0 and x_n. Thankfully, boundary conditions will be able to help enforce this!

Indeed, it will turn out that the most notable barrier to maintaining simulation accuracy will not be particles leaving the simulation area, but rather particles leaving the cells they were initially placed in (which is one of the inaccuracies of MPM; particles must be in their starting cells to keep simulation accuracy).

As a final word on extending these linear shape functions to multiple dimensions (outside the scope of this project), if there is a valid shape function in one of the dimensions $S_l(x)$, it follows that

$$\Sigma_l(x,y,z) = S_l(x) S_l(y) S_l(z) \text{ 3D Shape Function} \tag{1.62}$$

That is, for doing this simulations in 3D,the shape function of the nodes can be written as the product of 3 shape functions, each one describing a different variable.

2 Algorithms

Throughout the theory of modeling the equations of motion, there were frequent encounters of sums that are indexed over all of the nodes or material points.

Given that the program can be run with any number of nodes or material points[3], it is clear that when either the number of nodes or particles per cell is increased, the computation time will increase drastically.

This is not so much of an issue for a small test code like the one assembled here, where fairly limited problems are run to just to test the waters, often only having nodes/particles that number at most a few thousand. Thus, a computation time of 15 minutes is not horrific, especially considering that if the program was modified to use an MPI (Message Passing Interface) so multiple cores/processors can be used in execution, in addition to sending the code directly to a powerful server (which, thanks to Java, is as easy as dropping in all of the '.class' files without recompiling), the computation time can be substantially cut down.

However, this can only be stretched so far; a program can only be run on so many processors before receiving diminishing, or even worse, negative returns. A version of this code that can simulate very realistic situations may involve millions of material points, meaning that the efficiency of the code is paramount, and so good algorithms that allow us to save as much time as possible are needed.

For the purposes of these investigations, Java code/pseudocode will not be used to explore algorithms, but rather C, due to the fact that C very close to Assembly while still being at a higher level.

[3]Up to what the machine allows

2.1 Obtaining the Nearest Nodes

The first investigation will foray into finding an efficient algorithm to find the nearest nodes to a material point.

The first thought of an algorithm to get the nearest nodes of a material point go as

```
int nearest_node = 0;

double min_distance;
double test_distance;

for ( int i = 0; i < num_mps; ++i ) {

    min_distance = abs(mps[i].xpos - nodes[nearest_node].xpos);

    for ( int j = 1; j < num_nodes; ++j ) {

        test_distance = abs(mps[i].xpos - nodes[j].xpos);

        if ( test_distance < min_distance ) {
            min_distance = test_distance;
            nearest_node = j;
        }

    }


    if ( mps[i].xpos - nodes[nearest_node].xpos < 0. ) {
        mps[i].left_nn = nearest_node;
        mps[i].right_nn = nearest_node + 1;
    }
    else if ( mps[i].xpos - nodes[nearest_node].xpos > 0. ) {
        mps[i].left_nn = nearest_node - 1;
        mps[i].right_nn = nearest_node;
    }

}
```

This should get the two closest nodes to a given material point, for every material point. This takes advantage of the fact that the nearest nodes to a given material point should be separated by only a ± 1 to the index, as long as a material point is not right on top of a node (which is not allowed as it implies cell crossing).

This can be improved further by noting the fact that upon reaching the nearest node, the distances between subsequent nodes increases. Utilizing this and implementing a break statement makes the above algorithm more efficient.

However, note that with this algorithm, it is $O(n^2)$. It increases as the square of the

number of nodes[4]. There are improvements to be made.

Now, reinspecting the 1D interval, the system falls between the interval [xmin, xmax], in addition to there being a total of n nodes.

Noting that node 0 has the position of xmin, and node $n-1$ has the position of xmax.

With this, a function $f : [\text{xmin}, \text{xmax}] \to [0, n-1]$ can be created that is just

$$f(x) = \frac{x - \text{xmin}}{dx} \tag{2.1}$$

Where $dx = \frac{\text{xmax}-\text{xmin}}{n-1}$; the step size of the system.

With this, taking a material point's position (which lies between xmin and xmax), and then mapping it to the new space, it will lie between two integers which then correspond to the two nodes nearest to the material point.

This can be implemented as

```
double nearnode_approx;

for ( int i = 0; i < num_mps; ++i ) {

    nearnode_approx = (mps[i].xpos - xmin)/dx;

    mps[i].left_nn = floor(nearnode_approx);
    mps[i].right_nn = ceil(nearnode_approx);

}
```

Indeed, with this, the code is not only shorter, but it now runs at $O(n)$, a great improvement!

[4]The number of steps is something like $num\_nodes * num\_mps$, but $num\_mps = num\_nodes * c$ for some constant c.

2.2 Summing with Shape Functions

Now, a common procedure throughout this program is doing sums of the form

$$q_p = \sum_{l:\text{ nodes}} q_l S_l(x_p) \tag{2.2a}$$

$$q_l = \sum_{p:\text{ mps}} q_p S_l(x_p) \tag{2.2b}$$

That is, summing across nodes or material points is done quite often.

The most straightforward way of doing this looks like

```
// Sum over nodes
for ( int i = 0; i < num_mps; ++i ) {

    for ( int j = 0; j < num_nodes; ++j ) {

        mps[i].q += nodes[j].q * shapef( nodes[j], mps[i].xpos );

    }

}

// Sum over particles
for ( int i = 0; i < num_nodes; ++i ) {

    for ( int j = 0; j < num_mps; ++j ) {

        nodes[i].q += mps[j].q * shapef( nodes[i], mps[j].xpos );

    }

}
```

Observe that both of these are of $O(n^2)$, and thus they can blow up in computational time very quickly.

However, making certain restrictions/assumptions about the shape functions allows restructuring to reduce computational time.

For one, throughout this program, linear shape functions are assumed. That is,

$$S_l(x) = \begin{cases} 1 + \frac{x - x_l}{dx} & x_l - dx < x \leq x_l \\ 1 - \frac{x - x_l}{dx} & x_l < x < x_l + dx \\ 0 & \text{Otherwise} \end{cases} \tag{2.3}$$

The shape functions are only nonzero on an interval that stretches between the neighboring nodes.

This means that given a material point's position, it is only in the range of 2 nodes' shape functions. This means that the number of terms in the double-indexed sums can be cut down.

Really, this means that the double for-loop sums that are $O(n^2)$ do not need to be done. Instead, getting the nearest nodes to the material points (remember: only an $O(n)$ action[5]), consider the following code that does the exact same sums:

```
int left_nn, right_nn;

get_near_nodes(mps);

// Sum over nodes
for ( int i = 0; i < num_mps; ++i ) {

    left_nn = mps[i].left_nn;
    right_nn = mps[i].right_nn;

    mps[i].q += nodes[left_nn].q * shapef( nodes[left_nn], mps[i].xpos );
    mps[i].q += nodes[right_nn].q * shapef( nodes[right_nn], mps[i].xpos );

}

// Sum over particles
for ( int i = 0; i < num_mps; ++i ) {

    left_nn = mps[i].left_nn;
    right_nn = mps[i].right_nn;

    nodes[left_nn].q += mps[i].q * shapef( nodes[left_nn], mps[i].q );
    nodes[right_nn].q += mps[i].q * shapef( nodes[right_nn], mps[i].q );

}
```

Note how each of these only requires a single loop, and is thus $O(n)$, improving performance.

Thus, when doing sums of the above reported forms of sums over nodes or material points, the $O(n)$ version of these sums will be implicitly done.

[5]Also this only has to be done once due to particles crossing cells invalidating the results.

3 Simulation Options

In the file *Constants.java*, there is a class that holds many of the options that are free to be set before running the simulation.

These options are generated at compile time, so upon changing a value in here, the system needs to be recompiled (most easily done using the included makefile).

For the purposes of the rest of this document, the class name 'Constants' will be neglected[6], and its members will be implied to have its namespace.

[6] In C++: 'using namespace Constants;', assuming you've set the namespace to begin with.

3.1 System Geometry

First, there is a declaration of how many nodes there are in the system. More nodes usually implies better results from running the program, but at the expense of computational time. Note that the number nodes minus 1 is equal to the number of cells in the system; each cell holds the material points.

In the case of a fairly simple 1D simulation, only the lower and upper bound of the simulation area needs to be set.

These bounds are respectively named 'xlowbnd' and 'xhighbnd'. The lower bound is free to be nonzero, but doing so is an unnecessary complication; it's best to keep it simply 0.

The number of nodes and the geometry of the system to calculate dx, which is the size of a cell, which is also the same as the distance between nodes. With emphasis:

$$dx \equiv \frac{\text{xhighbnd} - \text{xlowbnd}}{num\_nodes - 1} \tag{3.1}$$

3.2 Particle Properties

From here, certain particle properties can be specified. There is a declaration of how many material points there are per cell; more particles per cell gives a better simulation, but also increases the time it takes for the program to run.

There is a choice as to whether or not the positions of the particles are updated at each timestep. That is, there is a choice of having static or moving particles. Not moving the particles results in a faster execution time, and ensures that particles will not cross cells, but in return the density of the system will not evolve in time and the results will not be as accurate[7].

That being said, not moving the cells can still result in fairly accurate results when our system is subject to a small perturbation (that is evolved over time). Thus, forcing the particles to be static has its uses in testing problems quickly and also to help in debugging.

[7]Excluding inaccuracies due to cell crossing.

3.3 Simulation Time

Next, there are options related to how long the simulation runs for. One sets the max time the simulation will run until, which does not necessarily affect the time it takes to run the program.

Next the number of steps the program should take to get to the above time is set. Increasing the number of steps predictably increases the time to run.

Note that some problems require particularly small timesteps. For example, if to evolve a state that ought to behave like a wave, then the simulation should have the timestep $dt << \frac{dx}{c}$; the timestep much less than the node spacing divided by the speed of sound of the material.

3.4 Data Output

The first two settings in this section are in regards to what the output files should be named. They should definitely be given *.csv* filetypes, but beyond this, the names are up to user preference. Again, note that changing the filename will require recompiling the program.

Next, a number that tells the program how often to write the data to a file is set. 'record_frequency' directly corresponds to how many steps taken between saving data. The smaller this number, the more data output, and thus the longer it will take the program to run.

Now, there is a declaration as to whether or not the program should produce a debug file that spits out a huge amount of information about the problem at hand. It is explained in more detail later, but it provides practically every useful bit of information about the system at every recorded timestep, organized in a useful way. However the file produced is exceptionally long, and thus adds to the computation time of the program[8].

Finally, two lists of strings that correspond to the headers of the node and material point data files are declared. Adding/removing entries from these lists does not affect the data output from the program; this document will revisit how to modify data output later, and will describe how to modify the headers.

[8] A simulation can easily produce a debug file of over 23 million lines, so be careful with opening it; use a lightweight text editor such as Vim, Vi, nano, etc or the computer being used could freeze up.

4 Main

The Main program is set in a file called MPMSolve.java, which is where all of the subprograms and variables are put together to simulate the system at hand.

After starting by setting a few options for the simulation, previously described, the system gets initialized (usually at $t = 0$).

The program then evolves the system in time, periodically exporting data, and then terminates.

Of course, this process requires more detail, which will be explored now.

4.1 Preparation

The first thing done is set the time t to be 0. In theory, this could be set to be nonzero, but there's no point in complicating the system unnecessarily.

Then, the output files are created, giving them headers. If a debug file is being created, it will be prepared too, otherwise a simple message will be written, and it will be closed.

Note that all of the simulation options come from a file: *Constants.java* that contains the relevant information that describes the simulation.

4.2 Initializing the System

After setting the various simulation options, we then begin to initialize the system.

First, some Lists that hold the data produced are declared; these Lists act as data storage containers and feed the functions that write the data.

One list, of length *num_nodes* holds the nodes, and another list of length $num\_particles = (num\_nodes-1)*num\_mps$ is created that holds the material points. The number of particles is chosen since there are $num\_nodes - 1$ cells, and each cell has *num_mps* per cell.

After generating the Lists (populated with uninitialized nodes and particles), the program begins to set their properties. Functions from *Initializations.java* are called to do this.

It starts by initializing the nodes, using the initial states, along with the geometry of the system to initialize the node positions, and then use these positions to set the density, velocity, and so on of the nodes.

After the nodes have been initialized, the material points are initialized, using the geometry to set the initial particle positions, then using the nodes (including the node shape functions) to set the mass, velocity, and so on of the material points.

Then, as a final step before beginning to evolve the system in time, the program finds the two nearest nodes to each material point, and then calculates all of the node masses. These ensure that we can evolve the system in a proper and timely fashion.

4.3 Time Evolution

Now, it is finally time to evolve the system in time.

As a preliminary step, the current data (contained in Lists) is written to the output files. This is only done after certain number of steps, as determined by a frequency variable previously discussed; but it is always written on the first passthrough of the loop. The functions used to save the data can be found in *DataWrite.java*.

Finally, the time evolution starts by evolving the particle strain of the system in time; this is done first since the algorithms need the velocity at the current time to find the strain at the next time; thus the strain must be before the velocity.

Then, the velocity is evolved in time, since it requires the stress at the current time. In the velocity evolution, the particles are optionally moved and node masses recomputed, in addition to the node velocity being updated. Both the particle and node velocity are updated in this one function.

After this, the particle stress is recalculated, which only requires the strain that was just calculated two function calls ago. This is done after the velocity calculation since the velocity requires the particle stress at the current time to advance the system.

Next, using the updated node masses and particle quantities, the node density, strain, and stress are recalculated. These quantities depend solely on the particle quantities.

Then, the time is advanced by adding dt to the current time.

This process is repeated until the exit condition is true.

4.4 Finishing Up

After the simulation loop has been broken out of, the data is written for a final time before the program terminates.

After this, the output files get flushed and closed.

With this, the simulation is over and the output files are produced, and the data is ready to be analyzed.

5 Subprograms

The main program essentially calls upon several subprograms which carry the bulk of the workload.

The purpose of the main program is essentially to call all the subprograms in the proper order, and in doing so, the equations of motion can be accurately modeled.

Indeed, an understanding on how these various subprograms works is a necessity to get new problems to run. Importantly, by understanding how the inner workings of our program works, further extensions to the code can be made to improve it, add more features, and develop future works.

A brief summary of the subprograms and how they work- will be presented in alphabetical order.

5.1 Accelerations.java

This Java file contains information on the external accelerations acting on the system.

As a concrete example, the Euler Equation reads

$$\frac{d\vec{v}}{dt} = -\frac{1}{\rho}\nabla p + \vec{f} \tag{5.1}$$

Where $\vec{f}$ is the external acceleration on the system, such as gravity, or any other external forces come to mind.

Indeed, this file describes the components of $\vec{f}$, provided they are sufficiently simple. They are declared an 'Accelerations' class, as members.

Note that the acceleration members are declared as functions of the 'node' class and a time t.

It goes without saying that some forces depend explicitly on time. However, the reason why the accelerations are declared to depend on a node itself is to keep the amount of function arguments to a minimum.

In essence, the 'node' class (to be explored in the future) contains practically all of the necessary physical information that needs to be known. As such, it can be expected that the external acceleration to depend on these node quantities.

By passing a node as an argument, velocity and position dependent external forces are easily defined without having to add extra arguments the acceleration functions. This generalizes quite nicely for any additional node members added.

Again, be warned that attempting to implement extra forces that are not strictly functions of velocity/position/time (such as viscosity) in this method will fail.

5.2 BoundaryConditions.java

This Java file contains information on the boundary conditions of the system.

For the most part, constraining the velocity on the boundary (alongside the initial states) is enough to fully solve the equations of motion for a continua. As such, this is the regime that has been implemented, but the application of extra boundary conditions is trivial.

In particular, in this simple 1D case, two functions of time that describe the left and right boundary of the system are declared.

The functions in this class are called twice every time the entire system updates in time: once upon fixing the Lagrangian Velocity, and another upon determining the proper node velocity.

Note that this program is built to handle one-dimensional problems, which are markedly more simple than higher dimensional problems, as such, the boundary condition functions are called on the first and last nodes. Scaling this up to higher dimensions, and including multiple boundaries or geometries is far beyond the scope of this project, and is much better fit to a team of developers over several years to build a robust system.

5.3 Constants.java

This file contains all of the various simulation options, such as the number of nodes, amount of time to simulate, and so on. This file is described in far more detail in the *Simulation Options* section of the documentation.

5.4 DataWrite.java

This Java file contains the functions that write the data produced by the true simulation part of the program to a file. In particular, these functions are built around writing *csv* files, which are useful for post-simulation analysis.

This file contains three methods that will be described.

5.4.1 .MakeHeaders

This method is quite simple, taking the freshly made, empty output files, and giving them headers that allow 'easy' differentiation between the columns of data.

This function works both for othe material point and node output files, just make sure to use the right list of strings for the respective files. These lists can be found in *Constants.java*.

5.4.2 .Node

This method writes the data of the nodes at a current time. In particular, it takes a List of Nodes.

In order to save on memory (and due to the fact that the system is Newtonian[9]), the current time of the system is passed as an argument, which gets written on the leftmost column for every node, which makes it quite easy to see when the time of the system changes in the output file.

How this program works is that it writes the data contained in the nodes in different columns, with each node being sequentially written. The leftmost column of the system corresponds to the time at which the data corresponds to. It does this for all times the simulation is run for.

5.4.3 .MaterialPoint

This program works essentially the same as its sister function, replacing 'Node' with 'MaterialPoint'. Instead of writing the material point density, this function writes the material point mass for debugging purposes.

[9]This means that Special Relativity is not considered

5.5 Debug.java

An entire section is dedicated to this file, detailed later. It contains various tools useful for debugging problems with data output.

5.6 InitialState.java

This file contains information about the initial state of the system.

The Euler Equation, and its closely connected differential equations, describe how the system evolves in time. This implies a known state at $t = 0$, from both a physical and mathematical perspective.

Indeed, this file describes the $t = 0$ velocity, strain, and density of the system. These are functions of space. Moreover, the Young's Modulus can be written as a function of space for completeness (but some work is required to get the program to behave correctly for a non-constant Young's Modulus). The stress is not included as an initial state since the constitutive relation can just be used to obtain it.

Note that these initial conditions should be consistent with the system's boundary conditions. Moreover, the various quantities should be related to one another in a predictable manner (see the equations of motion of the system).

5.7 Initializations.java

This file contains two functions which take the initial conditions and then initializes either the nodes or the material points.

Importantly, the nodes must be initialized first, followed by the material points.

As a matter of simplicity, the two methods contained in this class take Lists that have already been populated with the proper number of elements. The reasoning for this is that it allows the functions to be of *void* type instead of a List return type.

5.7.1 .InitializeMesh

To initialize the mesh (the same thing as our set of nodes),the lower bound of our system and the x step size need to be taken as arguments; these two variables plus the size of the list of nodes is enough to determine the system given the initial state.

From here, the node lengths are all set, with each one just being step size dx. From here, using the lower bound on x and dx, the x positions of the nodes can be built. It can confirmed that when there are n nodes, with standard lists that are indexed at 0 and index 0 denoting the leftmost element, that

$$\text{Node i's x position} = \text{xlowbnd} + \text{i*dx} \tag{5.2}$$

From here, using node positions alongside the $t = 0$ states of the system (found in InitialState.java) the node quantities can be set via

$$\text{Node i's t=0 quantity} = \text{InitialState.Quantity(Node i's x position)} \tag{5.3}$$

That is, for every relevant node quantity (density, strain, etc), it is set just by evaluating the quantity's function at the node's position. In this way, the nodes act as samplings of a truly continuous space.

Also, the constitutive relation can be used to get the initial stress, as opposed to calling it directly from a function.

5.7.2 .InitializeMaterialPoints

Now, with the mesh initialized, the material points are ready to be initialized.

To do this, the list of material points is passed as an argument (like the nodes, already populated with all the elements needed), in addition to the list of fully initialized nodes. As a matter of simplicity, the number of material points per cell, the dx step size, and the lower bound of x are taken as further arguments (implicitly or explicitly).

First the lengths of the material points are set. If there are m particles per cell, and each cell (IE: distance between nodes) is of length dx, then clearly:

$$\text{Material Point Length} = \frac{dx}{m} \tag{5.4}$$

The next step is to set the x position of a given particle, which is not as simple to think up.

First, consider the fact that each cell is fully filled by non-overlapping material points at $t = 0$. Next, the particles cannot be initialized on the boundary of the cells; these boundaries are reserved for nodes.

For one, each node has a shape function that spans a length of $2 * dx$, with half of that length to the left of the node's position, and the other half to the right. Imposing a similar restriction on material points of the system, where a particle's x position is right in the center of the particle.

Thus, the first material point in the list should have an x position of $0.5 * dx/num\_mps$ where num_mps is the number of material points per cell. Similarly, the last material point should have an x position of $(num\_mps - 0.5) * dx/num\_mps$. In both of these conditions, only the very boundary of each material point touches the node's x position, and thus these two points fit perfectly for what is desired.

This generalizes nicely to

$$\text{Particle i's x position} = (\text{i} + 0.5) * \text{Particle Length} \tag{5.5}$$

From here, the material point quantities, such as the mass, velocity, and so on, need to be initialized. This is done with

$$q_p = \sum_{l:\ \text{nodes}} q_l * S_l(x_p) \tag{5.6}$$

Where q_p is a material point's relevant quantity, and q_l is the same for the nodes, S_l is the node's shape function, and x_p is a particle's x position. The sum is done over all nodes.

However, given that this program is restricted to linear shape functions, which are only nonzero on the interval between the nodes immediately to the left and right of a given node, the sums simplify to

$$q_p = q_{l+} * S_{l+}(x_p) + q_{l-} * S_{l-}(x_p) \tag{5.7}$$

Where the '+' and '-' correspond to the nodes to the right or left of a given material point.

So, upon finding the nearest nodes to a given material point, this sum can then be done for every relevant quantity that material points can hold, excepting stress and mass.

To get the stress, $\sigma = E\epsilon$ is used.

Finally, to get the mass, the density of the material point is computed like any other quantity, and then the density gets multiplied by the particle's length to get its mass.

5.8 MPMMath.java

This file is a container for various functions that do various important slices of math.

At the moment, it only has a single function, but it is an excellent place to put any code used for further extensions of the program.

5.8.1 .GetNearNodes

This function takes in the list of material points (which must have their positions initialized), alongside the node step size dx and the minimum x value of the geometry.

These arguments are used to find the two nearest nodes to a given material point.

It is reasonably straightforward how this is done.

The geometry of the 1D interval can be represented by the set $[\text{xmin}, \text{xmax}]$, which can be linearly mapped to $[0, n-1]$, where n is picked to be the number of nodes. Clearly then

$$f(x) = \frac{x - \text{xmin}}{dx} \tag{5.8}$$

Where $dx = \frac{\text{xmax}-\text{xmin}}{n-1}$, as is familiar..

This function can then be used to map the x position of a material point to somewhere on the $[0, n-1]$ interval. Then, taking the floor and ceiling of the mapped position corresponds to the nearest nodes to a given material point. These two integers are then saved as class members of the material point.

Interestingly, this function only has to be called once. This is due to the fact that the described MPM method is only valid as long as particles do not cross cells; that is all of the particles stay in their starting cells.

5.9 MPMSolve.java

This file contains the main program. It has been described earlier in this document.

5.10 MaterialPoint.java

This file contains a single class: the MaterialPoint class. A given element acts as a container for data that describes a material point.

It holds immutable information: the length and mass of the particle, which will not change during the simulation[10].

It also holds the position and the physical velocity (the velocity at which the particle moves in space at).

Finally, it holds the measurable velocity, stress and strain, plus the young's modulus of the particle. These can all be used to build the node quantities.

Finally, a given member contains integers which correspond to the node located to the left and right of the particle.

[10]These variables are not private though. They can be publicly accessed.

5.11 Node.java

This file contains the Node class, which acts as a container for data that describes a given node.

It holds the length of the node, which does not change over time.

Moreover, it contains both the density and mass of the node (which both change during the simulation), alongside the velocity, stress, strain, and young's modulus of the node.

Finally, each node contains two functions: the node's shape function, and its derivative.

Again, a linear shape function is used. It evaluates to 1 at a given node's position, and linearly goes to 0 at nodes immediately to the left and right of a given node.

The implementation is as follows, using x_l as the position of node l and dx being the length of the node.

$$S_l(x) = \begin{cases} 1 + \frac{x - x_l}{dx} & x_l - dx < x \leq x_l \\ 1 - \frac{x - x_l}{dx} & x_l < x < x_l + dx \\ 0 & \text{Otherwise} \end{cases} \tag{5.9}$$

Analytically take the derivative (which guarantees better results; numeric derivatives tend to have high error):

$$\frac{\partial S_l(x)}{\partial x} = \begin{cases} \frac{1}{dx} & x_l - dx < x \leq x_l \\ -\frac{1}{dx} & x_l < x < x_l + dx \\ 0 & \text{Otherwise} \end{cases} \tag{5.10}$$

Unlike the shape function, which was continuous, the derivative is not continuous. This implementation averages the two function values on either side of the discontinuity.

5.12 SimUpdate.java

This file contains several classes and functions that tell the system how to update in time, be it updating node masses, particle positions, or anything else.

It has been constructed in a fairly modular way that easily allows the addition of additional functionality for added quantities that require evolution.

5.12.1 .ComputeNodeMasses

This function updates the masses of the nodes, given the system's particles.

It starts by zeroing out the node masses

From here, the node masses are computed via

$$m_l = \sum_{p:\text{mps}} m_p * S_l(x_p) \tag{5.11}$$

Again, given linear shape functions, this sum has many terms that are zero.

However, instead of computing this sum for a particular node in one loop, All material points are looped through, looking at their nearest nodes, and then using a given material point to find its contribution to the mass of its nearest nodes.

This, in essence, sends this mass update function from a $O(n^2)$ calculation to an $O(n)$ one. Thus when updates of node quantities are performed, it is implied that the above update method is used. t=The details of this loop method have been covered in more detail earlier in the document.

5.12.2 .MoveParticles

This function updates the positions of the particles in our system.

Its role is quite simple, doing the following update for every particle:

$$\text{MP xpos } = \text{ MP xpos + (MP physical velocity } * dt) \tag{5.12}$$

From this, it can be seen that it does a simple case of Euler's method to advance the position. This is used due to the fact that it is simple, fast, and the second derivative of velocity is not necessarily known.

5.12.3 .UpdateVelocity

This function is a serious workhorse in the program, handling several computations and reusing variables, allowing memory to be saved.

As a reminder, particle velocity is updated with

$$v_p(t + dt) = v_p + \sum_{l:\text{ nodes}} (v_l^L - v_l) * S_l(x_p s) \tag{5.13}$$

Where

$$v_l^L = v_l + \frac{dv_l}{dt} * dt \tag{5.14}$$

With

$$\frac{dv_l}{dt} = f(x,v,t) - \frac{1}{m_l}\sum_{p:\text{ mps}} l_p * \sigma_p * \frac{\partial S_l}{\partial x}(x_p) \tag{5.15}$$

And finally, the node velocity is just

$$v_l(t+dt) = \frac{1}{m_l}\sum_{p:\text{ mps}} m_p v_p(t+dt) S_l(x_p(t+dt)) \tag{5.16}$$

And the physical velocity of the particles is

$$v_{p,phys} = \sum_{l:\text{ nodes}} v_l^L S_l(x_p) \tag{5.17}$$

This is a lot to digest, so one step needs to be taken at a time. So, in order to use as few of variables as possible, The computation starts with computing only part of $v_p(t+dt)$, in particular:

$$v_p = v_p - \sum_{l:\text{ nodes}} v_l * S_l(p.xpos) \tag{5.18}$$

Then, v_l^L should be computed in the following steps:

$$v_l = v_l + f(x,v,t) * dt \tag{5.19a}$$

$$v_l = v_l - \frac{dt}{m_l}\sum_{p:\text{ mps}} l_p * \sigma_p * \frac{\partial S_l}{\partial x}(x_p) \tag{5.19b}$$

Then the boundary conditions should be applied

$$v_0 = \text{BoundaryConditions.XVelocity.Left}(t) \tag{5.20a}$$

$$v_{last} = \text{BoundaryConditions.XVelocity.Right}(t) \tag{5.20b}$$

With this, v_l^L has been computed for every node, and has just been stored in the v_l variable. Next, the physical velocity of the particles should be computed with:

$$v_{p,phys} = \sum_{l:\text{ nodes}} v_l S_l(x_p) \tag{5.21}$$

And then the physical velocity can be used to finish finding the new velocity of our particles with

$$v_p = v_p + v_{p,phys} \tag{5.22}$$

So, the velocity of the particles have been updated in time!

Now, the particles should be moved and the node masses recomputed (only if wanted) (see the other functions in this SimUpdate class). A boolean check in the UpdateVelocity function is present so that the particle moving and mass recomputations can be enabled or disabled, to save on performance. Note that the node masses do not need to be updated if the particles are static, since the node mass depends only on the particle positions. Thus if the particles are static, the node mass is constant in time

Finally, the new node velocity is:

$$v_l = \frac{1}{m_l} \sum_{p:\ \text{mps}} m_p v_p * S_l(x_p) \tag{5.23}$$

Lastly, the boundary conditions get applied as second time

$$v_0 = \text{BoundaryConditions.XVelocity.Left}(t) \tag{5.24a}$$

$$v_{last} = \text{BoundaryConditions.XVelocity.Right}(t) \tag{5.24b}$$

And then updates of the positions of the particles, the node masses, and the velocities of both the nodes and the particles are finished. It is important to remember that this effectively advances the timestep from t to $t + dt$, the updated particle strain needs to be calculated before updating the velocity!

Also, remember that most of the loops used to compute these quantities can run significantly faster if the nearest node method of computing material point and node quantities is used.

5.12.4 .UpdateParticle

The functions of this class serve to update certain particle quantities. At a base, it is just the strain and stress of the system, but it can easily be extended to further variables.

.Strain To compute the strain at a particle, consider the following:

$$\epsilon_p = \frac{\partial u_p}{\partial x} \tag{5.25}$$

Where u_p is the displacement of the particle. So far, this is trivial.

However, consider that $u_p = \sum_l u_l S_l(x_p)$, and thus

$$\epsilon_p = \sum_l u_l \frac{\partial S_l}{\partial x}(x_p) \tag{5.26}$$

Recall that in this program the displacement is not a recorded quantity, so the above cannot be used to update the particle strain.

However, note that upon taking the time derivative on each side:

$$\frac{\partial \epsilon_p}{\partial t} = \sum_l v_l \frac{\partial S_l}{\partial x}(x_p) \tag{5.27}$$

And then though the use of Euler's Method:

$$\epsilon_p = \epsilon_p + \sum_l v_l \frac{\partial S_l}{\partial x}(x_p) \tag{5.28}$$

.Stress Once the strain of the system has been recomputed, the stress quickly follows as

$$\sigma_p = E_p * \epsilon_p \tag{5.29}$$

5.12.5 .UpdateNode

This class serves to provide methods to update certain node quantities, in all, they are all computed essentially the same methods.

.Density When updating a node's density, all that is really needed is the node's mass and its length, and thus

$$\rho_l = \frac{m_l}{l_l} \tag{5.30}$$

.Strain Now, finding a node's strain is the same as finding the node velocity, given the material points. It is simply

$$\epsilon_l = \frac{1}{m_l} \sum_p m_p * \epsilon_p * S_l(x_p) \tag{5.31}$$

.Stress The stress is easily computed with

$$\sigma_l = E_l * \epsilon_l \tag{5.32}$$

6 Debugging Tools

Few programs are written without speedbumps, and while compilers are capable of catching syntax errors, they cannot catch calculation mistakes on the programmer's part.

As such, this program contains a few different functions that allow us the numerical output of the system to debugged via output to the console, or an option to produce a file output of the entire system.

These functions are contained in the file Debug.java.

6.1 Node and Material Point Output

Noted as *NodeOutput* and *MPOutput*, these are functions that allow the node and material point properties to be printed en masse.

Each of the functions of these two classes will go through the list of nodes or particles presented, and then spit out all of the density, xvelocity, etc values associated with the list.

These are just prewritten for loops that enclose print statements, but allow for a compact execution of these code blocks, helping to prevent bloat in Main.

For nodes, there is support for output of node density, x velocity, stress, strain, young's modulus, and mass.

For particles, there is support for output of particle position, physical velocity, standard velocity, stress, strain, young's modulus, and mass.

Note that it is quite easy to extend these functions to produce output for any user-added node or particle class members.

These functions are not very useful when simulating systems for more than a few timesteps, as the sheer length of output will be enormous for a real simulation. However, these functions work well for ensuring the expected output gets produced over a few timesteps, and for quickly checking to see if any NaNs have been output.

6.2 The Data Dump

The Data Dump functions provide the most extensive output possible, nearly organized into a file.

By default the two associated functions are inserted in the main program, but whether or not they run is subject to a boolean variable in the Simulation Options.

Note that the debug file produced is extensive and long, and will thus take quite some time to write, so it should not be written once a simulation working without error.

The first associated function is *DataDumpHeader*, which simply writes a header for the debug file. It is fairly uninteresting, and just gives minimal preliminary information about the file that is to come.

Next, the *DataDump* function is systematically called throughout the simulation. It is what records all of the information of the system.

First, the current time the dump function is called at is written. Thus, the file is organized by the various simulation times.

Next, it tabs and prints a node, denoting it with its index in the list of nodes.

From here, it goes through all of particles, and finds all of the particles that are within the reach of the given node's shape function.

From here, it writes all of the material points contained in the node, and gives the index of each of these particles.

Then it prints the given node's position, then tabs and lists all of the contained material point positions.

It does quite similar operations with the node density/mass, velocity, stress, strain, and young's modulus.

Finally, if there are any material points who do not fall into a node's reach, they will be written to an 'unused' section of the current time division, with all of their properties written.

From here, the function terminates, waiting to be called again later down the line.

The entries of the file are appropriately tabbed, to keep it as neat as possible.

Be warned that each call of this function produces about $8 * num\_mps * num\_nodes$ lines of text, so a single function call can easily produce a few thousand lines of text. Thus it contributes heavily to computation time for a long simulation, in addition to the output file itself being enormous.

Attempting to open a long file without the use of a lightweight text editor can cause a computer to freeze for an extended period of time.

Despite the performance drawbacks of creating the debug file, it allows manual calculation of the expected node/particle quantities at any given instance of time, and thus enabling a method to check the output of the program with what is expected, allowing for debugging of the algorithms used to calculate the various node quantities.

Moreover, the file can easily be used to ensure that the material points have been positioned correctly correctly, and to see when cell crossing occurs.

Indeed, the debug file is exceptionally useful when attempting to extend the program to simulate systems that do not obey linear elasticity.

7 Running New Problems

So far, all the work that has been presented so far has been on the topic of linear elasticity, in particular, it hass been assumed that

$$\sigma = E\epsilon \tag{7.1}$$

There are, of course, other equations of state, and so there are countless problems could be run, but just adding the functionality for a single problem is enough to demonstrate how anyone could extend the program for whatever their mind might so think.

In particular, the goal is to go from modeling solids, to modeling compressible ideal gasses, with the equation of state of

$$-\sigma = p = (\gamma - 1)\rho\epsilon$$

So, the changes that need to be made to the program need an overview.

7.1 New Variables

The stress associated with an elastic solid is related to the strain ϵ. However, for an ideal fluid, there is a contribution only from the pressure p, with a related adiabatic γ factor.

As such, in *Node.java* and *MaterialPoint.java* files, there is no need for the strain and young's modulus members. These should be replaced with pressure and gamma members. Observe that

$$\text{Strain} \rightarrow \text{ Pressure} \tag{7.2a}$$

$$\text{Young's Modulus} \rightarrow \ \gamma \tag{7.2b}$$

Ultimately, this is just a matter of renaming variables; no new ones really need to be added (which has memory implications). In the future, it can be seen that no extra variable is needed for the internal energy ϵ.

Then, going into the *InitialState.java* file, where the $t = 0$ states are specified, instead of specifying the initial strain and young's modulus, these get replaced with two functions (or renamed) so that they represent the initial pressure and adiabatic gamma (which is more commonly a constant everywhere; the $t = 0$ function can just be made constant).

Note that for a valid fluid problem, $p > 0$ for all time; a negative pressure implies that there is a negative energy density of the system, and it will definitely cause issues down the line; much like how the density must be greater than zero, the same is true with pressure.

There is a commented out function down at the bottom that specifies the initial stress of the system, using the stress-strain relation. For completeness, use the following relation:

$$\sigma = -p \tag{7.3}$$

This way, this function is enabled, bugs will not pop up.

A good choice of initial states would be to leave everything at zero, or constant. A constant pressure and density, and zero velocity works good for testing. This will be built on later.

$$\rho(x, 0) = \rho_0 \tag{7.4a}$$

$$v(x, 0) = 0 \tag{7.4b}$$

$$p(x, 0) = p_0 \tag{7.4c}$$

$$\gamma \text{ is nonzero, constant and defined} \tag{7.4d}$$

7.2 External Forces and Boundary Conditions

Then going into *Accelerations.java*, any desired external accelerations can be added. In this test case, they are being left as 0, but for a general new problem, this may be something that should be noted for modification.

Similarly, going into *BoundaryCondition.java*, the desired boundary conditions can be set. An excellent set of testing boundary conditions would be to produce a wave on the right side of the system that propagates to the left, since a small disturbance in a fluid should produce waves that we can be modelled analytically. As an example:

$$v_0(t) = 0 \tag{7.5a}$$

$$v_n(t) = v' \sin(\omega t) \tag{7.5b}$$

Note that boundary conditions on v are all that need to be specified.

7.3 Initializing the System

Now with the initial states and boundary conditions (which should be compatible) set, the initial state of the nodes and particles need to be generated.

Opening *InitialState.java* and then going to the function that initializes the nodes, the lines that initialize the young's modulus and strain from the InitialState functions need to be replaced. These class members need to be replaced with the new pressure and gamma fields, and the now defunct initial strain and modulus functions replaced with the new pressure and gamma ones.

Next, on the line where the node stress initialized, using the fact that $\sigma = -p$ this line gets updated.

Then, *Initializations.java* should be open and the above changed made.

Now going to the function that initializes the material points, and making the same changes of replacing the strain with pressure, Young's Modulus with γ, and updating the constitutive relation, the initial state of the system should be ready to go.

7.4 Updating the Simulation

Now, the variables of the system need to update in time. Going into *SimUpdate.java* the largest undertaking yet begins.

It turns out that the *UpdateParticle* subclass needs to be updated first, and is where the most work needs to be done.

For one, from the equation of state, and how the internal energy evolves in time, it has already been shown that the pressure evolves in time as

$$\frac{\partial p_p}{\partial t} = (\gamma - 1)v_p \cdot \sum_l p_l \nabla S_l(x_p) - \gamma \sum_l p_l v_l \nabla S_l(x_p) \tag{7.6}$$

Updating the system using Euler's Method:

$$p_p(t + dt) = p_p(t) + \frac{\partial p_p}{\partial t} dt \tag{7.7}$$

These can be used to change the UpdateParticle.Strain to an UpdateParticle.Pressure function, and including the above equations.

The new implementation should borrow heavily from the optimized algorithm already implemented into the strain updater to save on computation time. Given the long equation that updates the pressure, it is recommended to implement it over at least 4 lines. In general, borrow heavily from the included source code as a basis for updates.

Jumping down to the UpdateParticle.Stress function, $\sigma = -p$ is used to update the stress.

Going to the *UpdateNode* subclass, the 'Strain' function needs to be renamed to a 'Pressure' function, including the arguments.

Similarly, the stress updating function is updated to $\sigma = -p$.

7.5 Labels and Writing Data

Now, going into *Constants.java*, and hopping on down to the two lists of strings, 'strain' gets replaced with 'pressure', or 'pres'. This makes sure that the columns in the output files have updated names, coinciding with the new variables of the system. It is also a good idea to rename the output files themselves, so that the program does not overwrite any old data present in the current directory.

Now, hopping into *DataWrite.java*, in both of the important writing functions present (not the one that writes the headers), the section that actually writes the data to a file needs some modification. Instead of writing the now nonexistent strain, the function should write the pressure instead.

7.6 Updating Debugging Tools

Opening *Debug.java*, all of the debugging functions need to have all of the strain and young's modulus variables replaced with the pressure and γ variables. Note that there is some text that is hardcoded (for organization purposes) that reads 'strain' and 'young's modulus' that will not produce a compilation error if left unchanged, and will result in a cosmetic mislabeling of the pressure as strain, etc.

Ultimately, if these changes seem to be too much to handle, the makefile can be modified to not compile *Debug.java*, and then all references of debugging functions can be removed from the main function.

7.7 Finishing Up

Now, the main function in *MPMSolve.java* needs to be opened and the function names updated; any function names that have been changed must now be updated.

In this example case, this only involves updating the UpdateNode/Particle functions, making sure that instead of updating strain, the pressure gets updated.

From here, the program should be ready to run and should be able to produce valid results.

For the given initial states and boundary conditions, there is an analytic solution of:

$$c \equiv \sqrt{\gamma \frac{p_0}{\rho_0}} \tag{7.8a}$$

$$v(x,t) = \begin{cases} 0 & x < L - ct \text{ and } t \in [0, \frac{L}{c}] \\ v_0 \sin(\omega(\frac{x-L}{c} + t)) & x > L - ct \text{ and } t \in [0, \frac{L}{c}] \\ 2v_0 \sin\frac{\omega x}{c} \cos\omega(t - \frac{L}{c}) & x < ct - L \text{ and } t \in [\frac{L}{c}, \frac{2L}{c}] \\ v_0 \sin(\omega(\frac{x-L}{c} + t)) & x > ct - L \text{ and } t \in [\frac{L}{c}, \frac{2L}{c}] \end{cases} \tag{7.8b}$$

With a similar profile for p and ρ. Note that results generally become cleaner when the particles are made to be static.

Indeed, observe the simulated velocity field upon implementing the above ideas with the following arguments:

Simulated on $[0, 1]$
Nodes: 76
Particles/cell : 2
Moving Particles
$\gamma = \rho_0 = p_0 = 1$
$v(x = 1, t) = 0.01 * \sin(10 * t)$
$v(x, t = 0) = 0$
$\rho(x, t = 0) = p(x, t = 0) = 1$ Timesteps/second[11]: 100,000

[11] This means that if the max simulation time is set to be 3, then there are 300,000 steps.

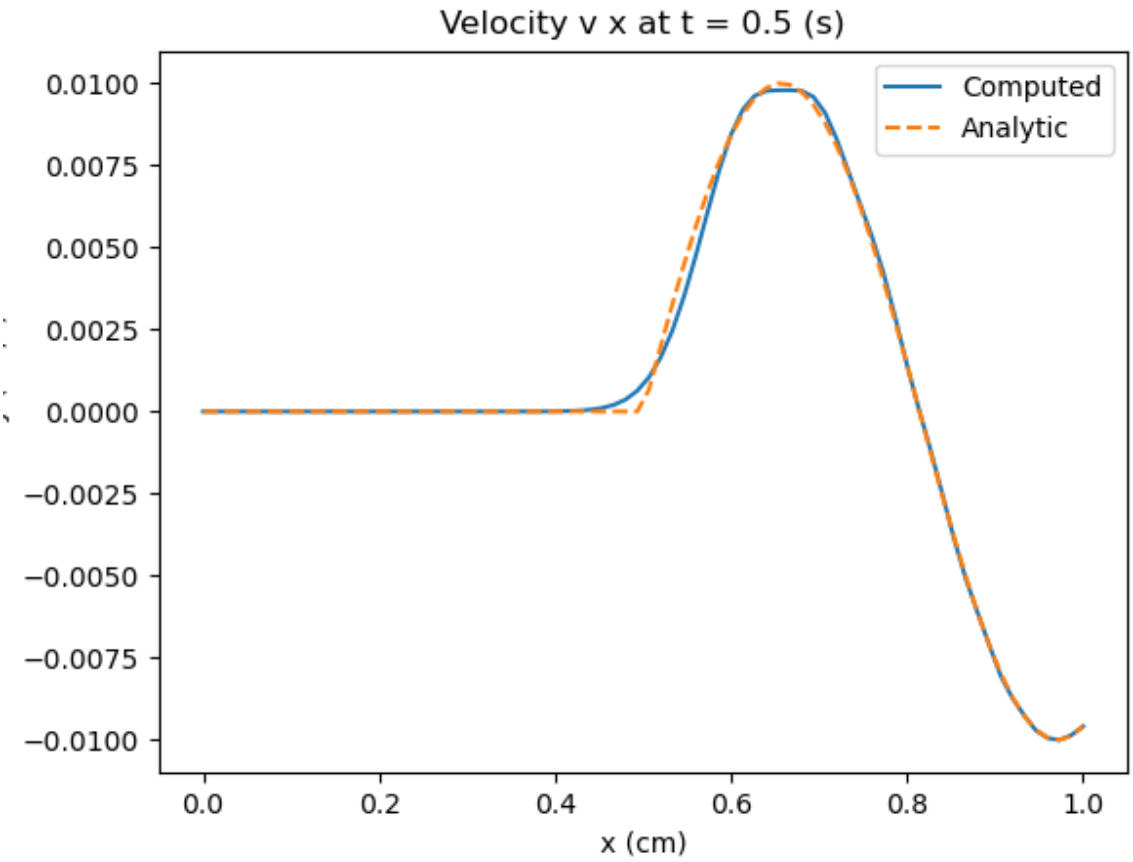

Figure 1: The velocity of the system at $t = 0.5s$. This snapshot occurs before the incoming wave reflects off of the left boundary.

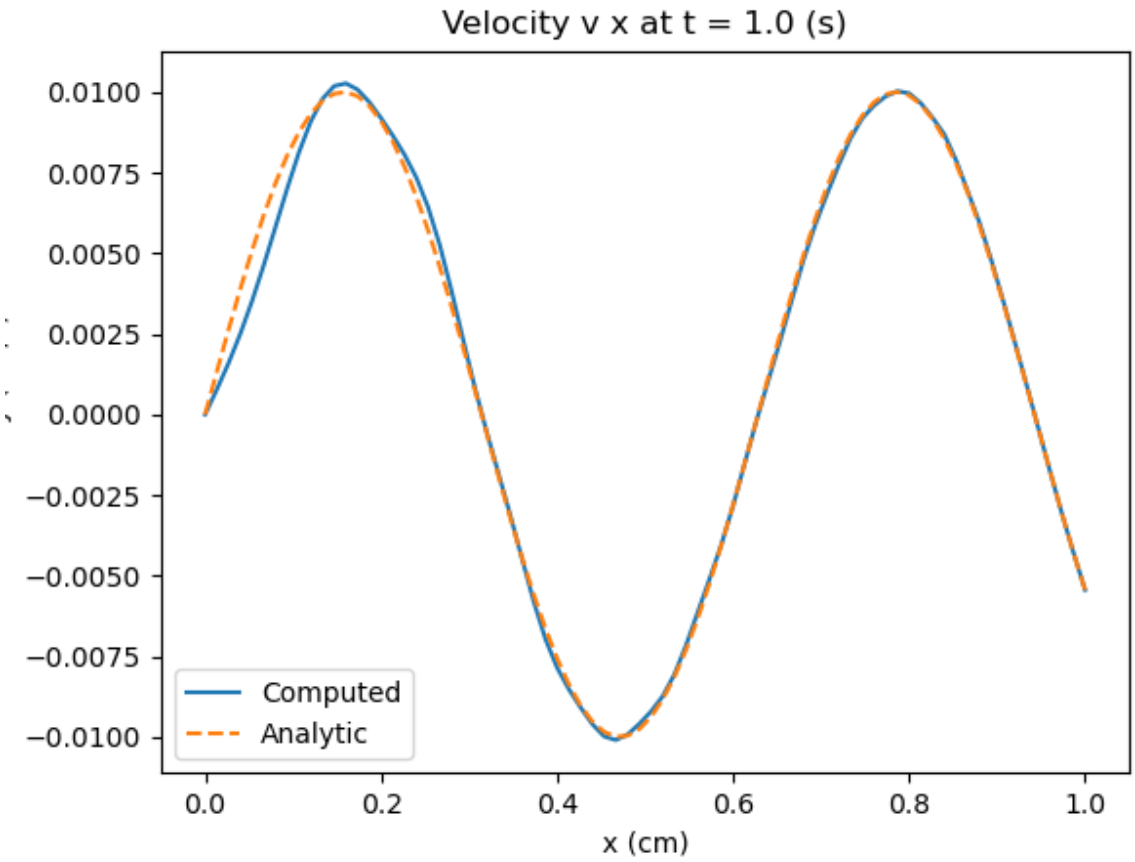

Figure 2: The velocity of the fluid at $t = 1s$. This is right at the moment the incoming wave strikes the left boundary.

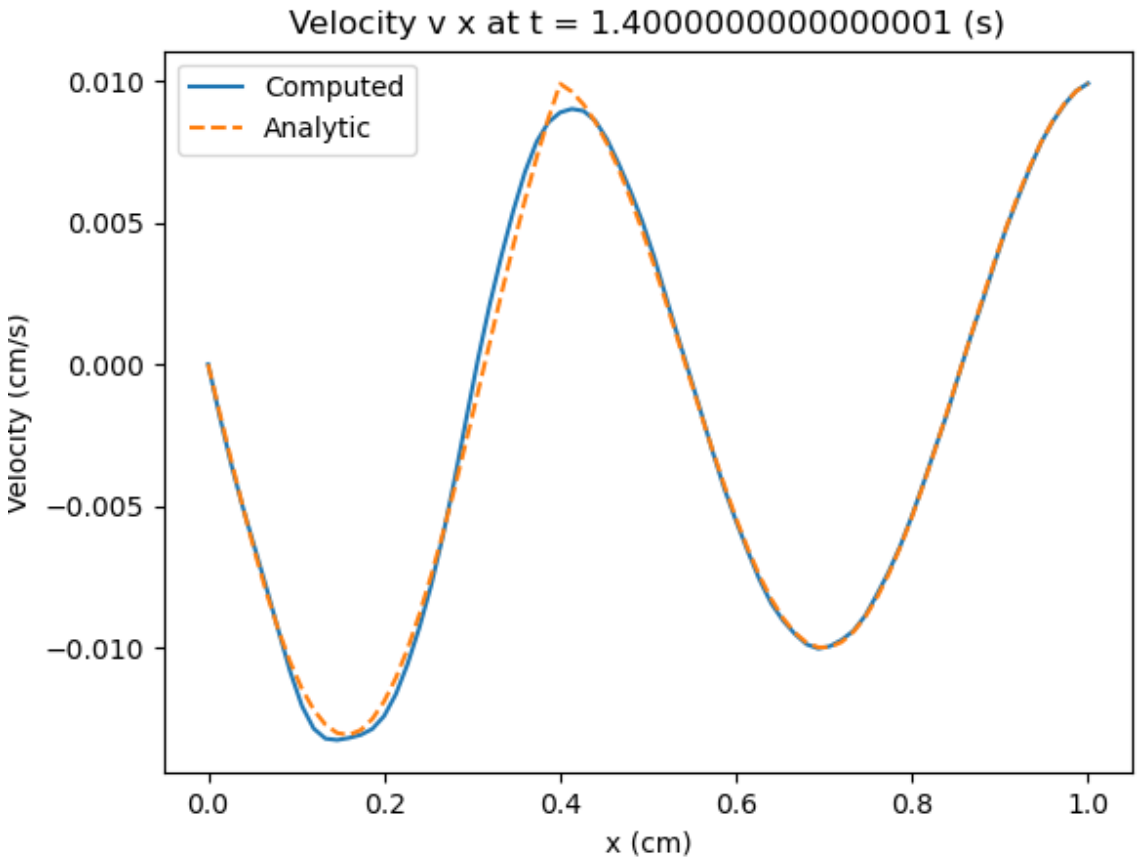

Figure 3: The velocity of the system at $t = 1.4s$. The reflected wave adds to the incoming wave to increase the amplitude of the velocity in the left part of the system.

Indeed, it most certainly seems as if the analytic and numerical solutions line up nicely, though there are clearly a few discrepancies.

8 Test Problems

A few test problems have been run to show that the code is sufficiently stable, capable of handling weak shocks, reflection, nonzero boundary conditions, and so on.

These all make excellent problems to test out of the bag to ensure that the provided code not only works, but to also rerun after modifications have been made.

Analytic solutions have been provided to as much detail as is reasonable so that they can be compared to the results obtained numerically, in addition to example outputs obtained via running the code.

8.1 Standing Waves

8.1.1 The Analytic Solution

For a solid not subject to external forces, the equations of motion in one dimension are:

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x}(\rho v) = 0 \tag{8.1a}$$

$$\rho \frac{dv}{dt} = \frac{\partial \sigma}{\partial x} \tag{8.1b}$$

$$\sigma = E\epsilon = E\frac{\partial u}{\partial x} \tag{8.1c}$$

Now, suppose that the solid is at rest. The behaviour of the system should be explored when it is subject it to a small perturbation in the displacement. That is

$$u(x,t) = u'(x,t) \tag{8.2}$$

Where u' is small.

Taking the derivative with respect to time gets the velocity, or with respect to space gets the strain. It is also assumed that these quantities are appropriately small. Similarly, the change in density is also small. Finally, the Young's Modulus E is taken to be constant in space and time.

So, with this, the quantities of the system are written as

$$u(x,t) = u'(x,t) \tag{8.3a}$$

$$v(x,t) = v'(x,t) \tag{8.3b}$$

$$\rho(x,t) = \rho_0 + \rho'(x,t) \tag{8.3c}$$

$$\epsilon(x,t) = \epsilon'(x,t) \tag{8.3d}$$

Where all primed quantities are small.

Plugging these into the equations of motion:

$$\frac{\partial \rho'}{\partial t} + \frac{\partial}{\partial x}[(\rho_0 + \rho')v'] = 0 \tag{8.4a}$$

$$(\rho_0 + \rho')(\frac{\partial v'}{\partial t} + v'\frac{\partial v'}{\partial x}) = E\frac{\partial \epsilon'}{\partial x} \tag{8.4b}$$

$$\epsilon' = \frac{\partial u'}{\partial x} \tag{8.4c}$$

Now, the product of two small quantities is essentially 0. This means that any two primed quantities multiplied together goes to 0.

Under this, the equations of motion simplify to

$$\frac{\partial \rho'}{\partial t} + \rho_0 \frac{\partial v'}{\partial x} = 0 \tag{8.5a}$$

$$\rho_0 \frac{\partial v'}{\partial t} = E \frac{\partial \epsilon'}{\partial x} \tag{8.5b}$$

$$\epsilon' = \frac{\partial u'}{\partial x} \tag{8.5c}$$

Note that with these approximations, the nonlinear differential equations have all become linearly coupled!

Now, using the fact that $v' = \frac{\partial u'}{\partial t}$, and chaining in $\epsilon' = \frac{\partial u'}{\partial x}$,

$$\frac{\partial^2 u'}{\partial t^2} = \frac{E}{\rho_0} \frac{\partial^2 u'}{\partial x^2} \tag{8.6}$$

That is to say that the displacement in this slightly disturbed material is modeled by the wave equation!

Now since the displacement's behaviour is known, the rest of the system's behaviour can be obtained through continually chaining in the equations of motion correctly. See that for all quantities:

$$\frac{\partial^2 A}{\partial t^2} = \frac{E}{\rho_0} \frac{\partial^2 A}{\partial x^2} \tag{8.7}$$

Where A can be the displacement, velocity, strain/stress, or density. They are all modeled by the wave equation. In particular, the system has a speed of sound $c = \sqrt{\frac{E}{\rho_0}}$

With this in mind, these equations should be solved in a simple enough of a situation.

So, suppose the system is subject to the following boundary conditions, which is equivalent to having ends that are held fixed:

$$v(x = 0, t) = 0 \tag{8.8a}$$

$$v(x = L, t) = 0 \tag{8.8b}$$

So, the velocity needs to be found such that it is modeled by the wave equation and subject to these boundary conditions.

There are no doubt countless such solutions, but observe that the following guess gets the job done:

$$v(x, t) = v_0 \sin \frac{n\pi x}{L} \cos \frac{n\pi ct}{L} \; n \in \mathbb{Z} \tag{8.9}$$

This implies the following related quantities by the constitutive relation and the continuity equation:

$$\epsilon(x, t) = \frac{v_0}{c} \cos \frac{n\pi x}{L} \sin \frac{n\pi ct}{L} \tag{8.10a}$$

$$\rho(x, t) = \rho_0 (1 - \frac{v_0}{c} \cos \frac{n\pi x}{L} \sin \frac{n\pi ct}{L}) = \rho_0 (1 - \epsilon) \tag{8.10b}$$

These equations are valid for any n that is an integer, so these could be summed up for various n values and a valid solution is still obtained, but in that case, the analytic solution would be much harder to understand.

So, instead this example will focus on a single value for n: $n = 1$.

Setting $t = 0$ for the analytic solution obtains the initial states that should be able to produce the desired results in a numerical setting:

$$v(x, t = 0) = v_0 \sin \frac{\pi x}{L} \tag{8.11a}$$

$$\epsilon(x, t = 0) = 0 \tag{8.11b}$$

$$\rho(x, t = 0) = \rho_0 \tag{8.11c}$$

Now, it is very important to remember that this analytic solution is only valid for small disturbances, so v_0 should be appropriately small, in particular $v_0 << c$.

With this in mind, this problem will be simulated with the following parameters, with special cases for moving and static particles:

Simulated on $[0, 1]$
Nodes: 51
Particles/cell: 2
$E = \rho_0 = 1$
$v(x, t = 0) = 0.01 * \sin(\pi x)$
$\epsilon(x, t = 0) = 0$
$\rho(x, t = 0) = \rho_0$
Timesteps/second = 100,000

8.1.2 Results for Static Particles

For the static particle case, observe:

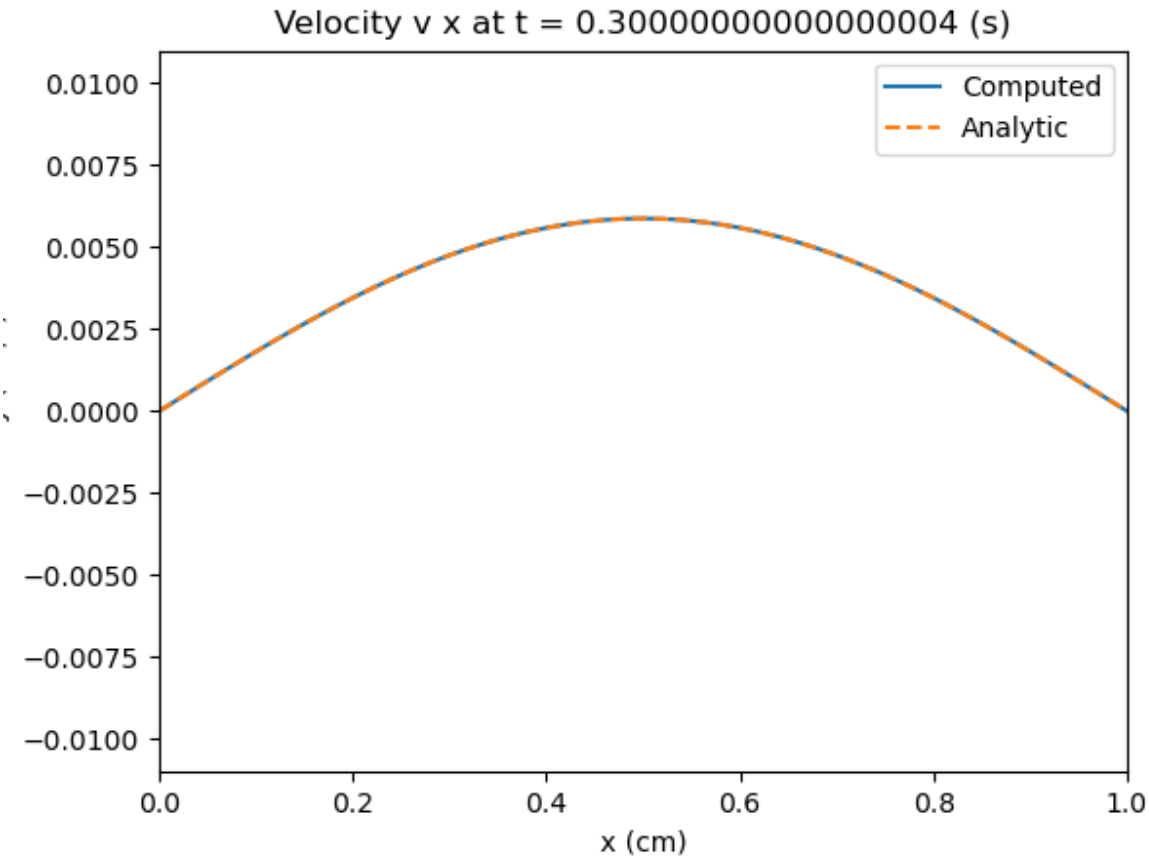

Figure 4: The velocity of the system at $t = 0.3s$.

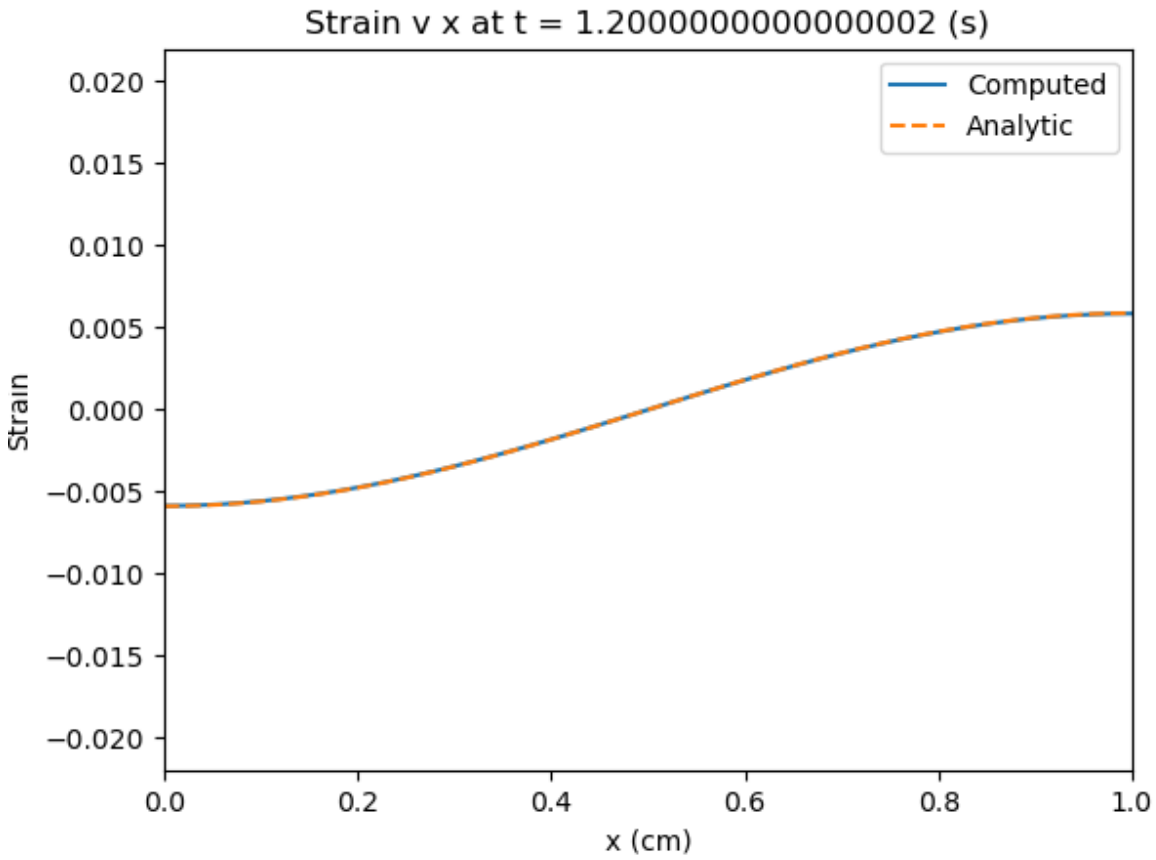

Figure 5: The strain of the system at $t = 1.2s$. Even though the strain does not explicitly have boundary conditions, the strain at the boundaries has opposite signs.

Even in the case of fixing the particles, the velocity and strain distributions match up very well to the analytic solution. This applies to all times in the simulation[12].

[12]Of course, the space that is reserved for pictures in this document is limited. Run the simulation to confirm this!

8.1.3 Results for Moving Particles

Observe the following results for moving particles, each plot displaying the velocity, density, or strain at a certain time.

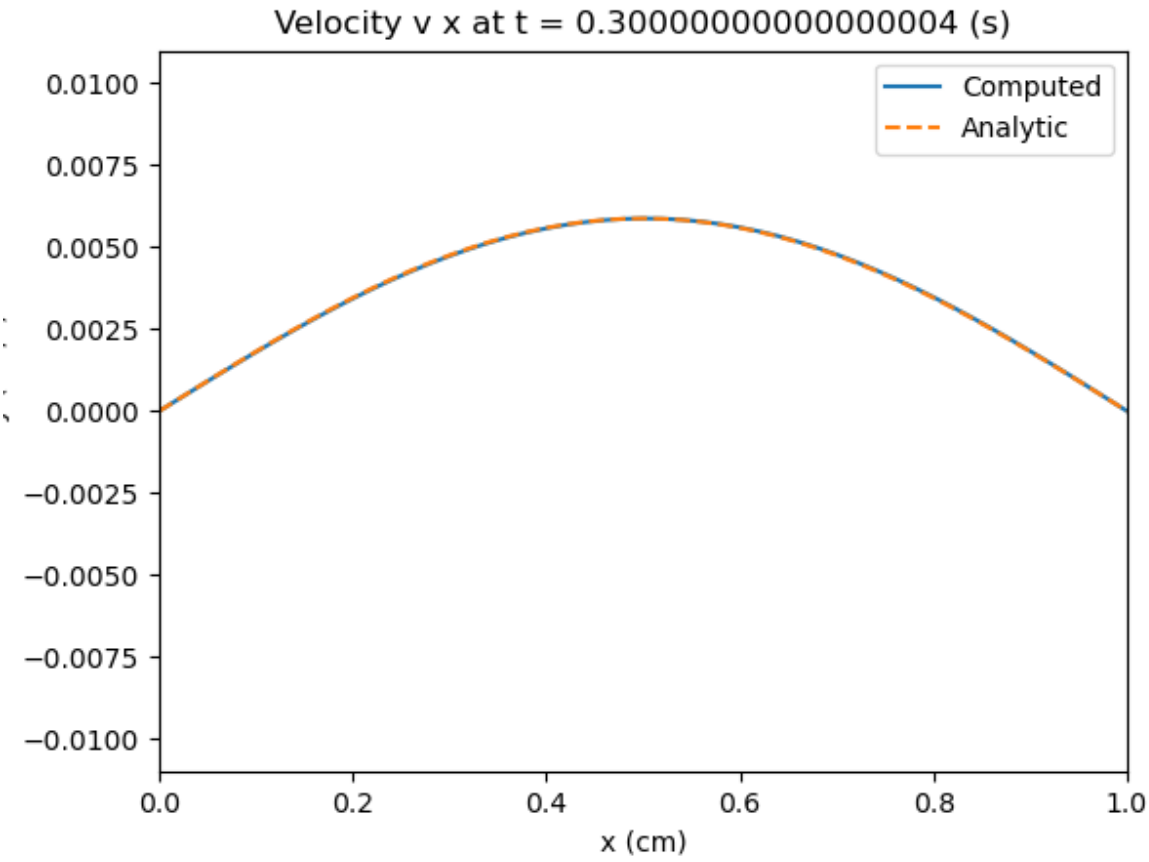

Figure 6: The velocity of the system at $t = 0.3s$.

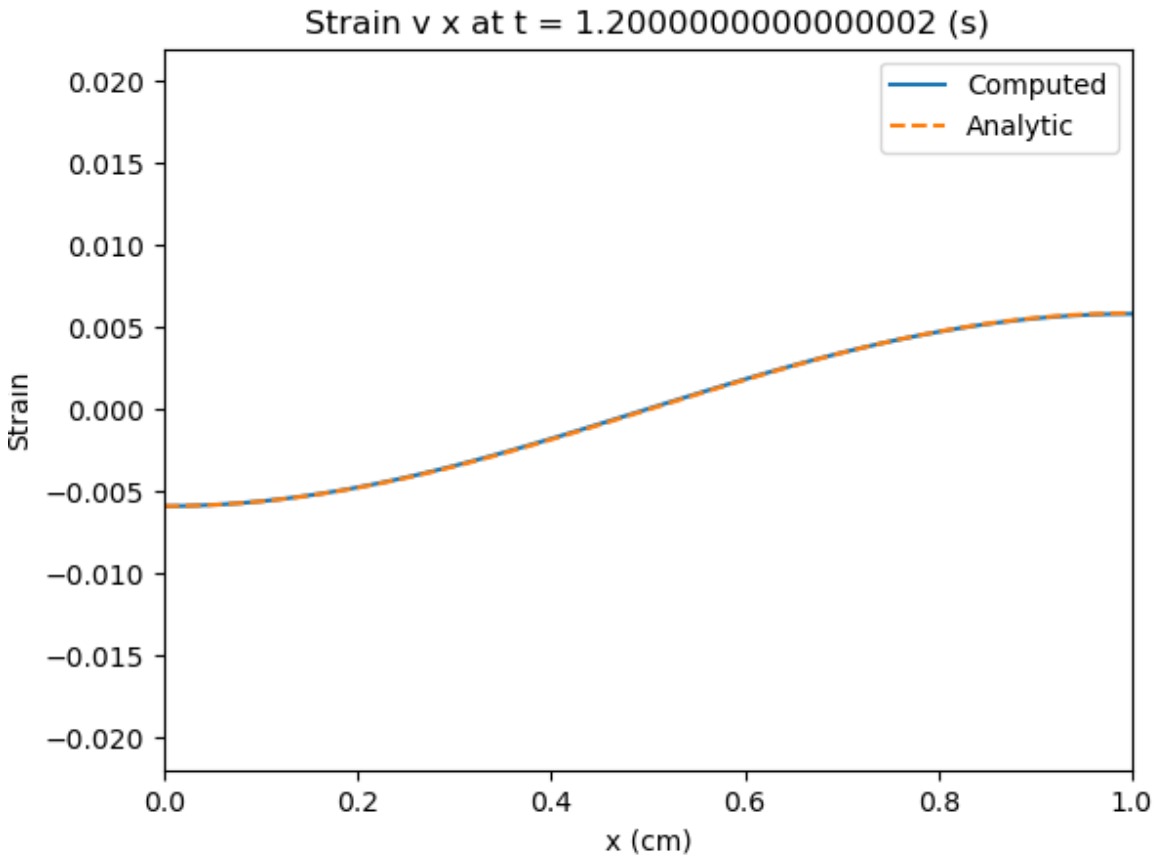

Figure 7: The strain of the system at $t = 1.2s$. Even though the strain does not explicitly have boundary conditions, the strain at the boundaries has opposite signs.

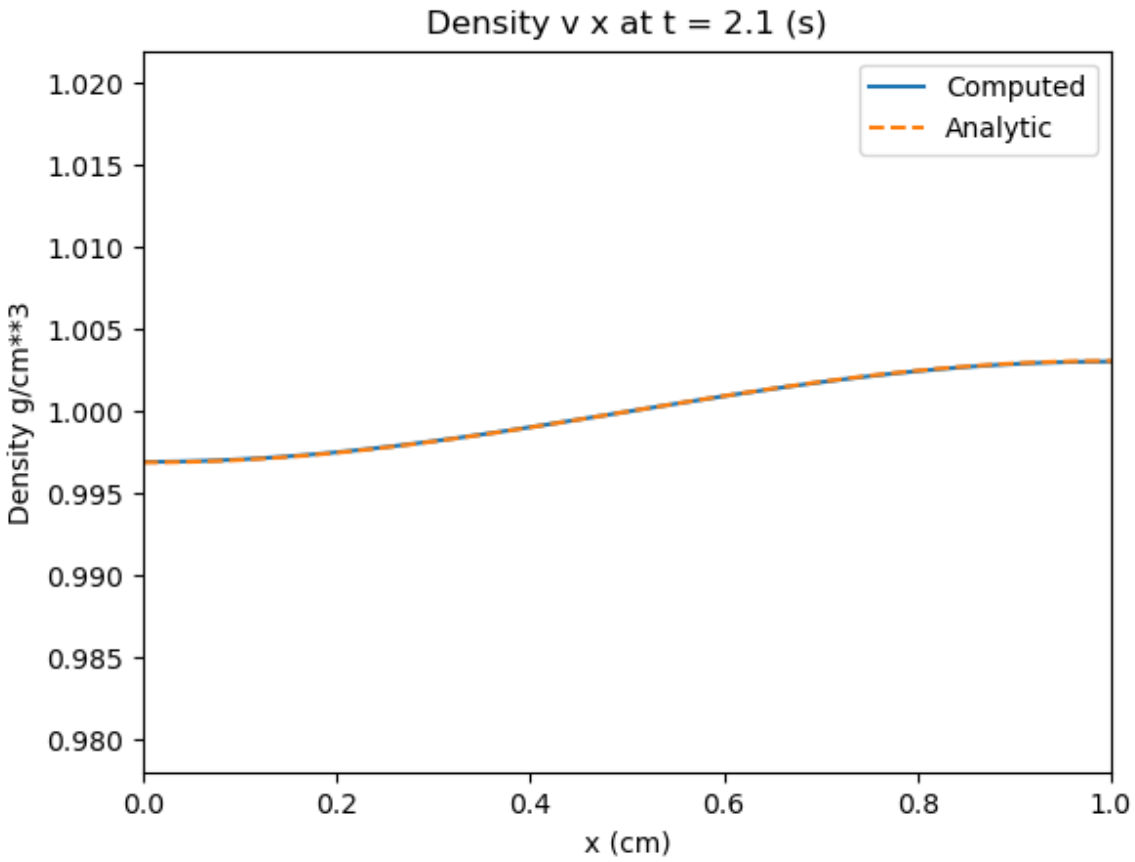

Figure 8: The density of the system at $t = 2.1s$. This is nearly a half-period after the strain plot, so this is why the density plot resembles the strain plot.

Indeed, the results obtained fit very well to the analytic results. It is important to note that taking a too large of a timestep, or similarly creating too many nodes in the system will result in inaccurate results. For this problem in particular, balancing the number of nodes is important; there needs to be enough to get fine detail for the system, but not too many as to go below the constraint of $dt << \frac{dx}{c}$.

8.2 Wave Reflection

8.2.1 The Analytic Solution

Now, revisiting the previous situation: a small disturbance in the velocity; this results in the system being modeled by the wave equation.

However, instead of subjecting the system to fixed boundaries, the left boundary will be left fixed, and the right boundary will be driven with a frequency ω. In particular:

$$v(x=0,t)=0 \tag{8.12a}$$

$$v(x=L,t)=v_0\sin\omega t \tag{8.12b}$$

Where v_0 is appropriately small.

Again, the goal is to compute the analytic solution for this scenario.

So, starting with

$$v(x,t=0)=0 \tag{8.13}$$

Physically, this corresponds to having our solid at total equilibrium, and then a wave being driven into it. The wave will then propagate (with speed c) and then reflect off of the left boundary, and go traveling to the right.

For simplicity's sake, the analytic solution will only be calculated up until the point that the wavefront re-impacts the right boundary; this will capture the analytic solution up until the moment before a second reflection.

So, when looking at the initial conditions, and considering the fact that our wave propagates with speed c, it is noted that the following velocity works as long as the wave has not reflected off of the left wall:

$$v(x,t)=\begin{cases}0 & x<L-ct \text{ and } t<\frac{L}{c}\\ v_0\sin(\omega(\frac{x-L}{c}+t)) & x>L-ct \text{ and } t<\frac{L}{c}\end{cases} \tag{8.14}$$

Note how this solution captures the continuity of the velocity at $x=L-ct$.

Now, the system reflecting off of the left boundary needs to be dealt with, which occurs at $t=\frac{L}{c}$.

Now, note that for $t>\frac{L}{c}$, much like for the previous times, the velocity is described in a piecewise manner. With this, the right portion of the velocity will be the exact same as the wave produced by the driven boundary. The left hand side, on the other hand, can be written as a sum of this wave plus the contribution by the reflected wave, so

$$v(x,t)=\begin{cases}v_0\sin(\omega(\frac{x-L}{c}+t))+v'(x,t) & x<ct-L \text{ and } t\in[\frac{L}{c},\frac{2L}{c}]\\ v_0\sin(\omega(\frac{x-L}{c}+t)) & x>ct-L \text{ and } t\in[\frac{L}{c},\frac{2L}{c}]\end{cases} \tag{8.15}$$

Of course, this system needs to be continuous at $x=ct-L$, which means

$$v'(0,t)=-v_0\sin(\omega(t-\frac{L}{c})) \tag{8.16a}$$

$$v'(ct-L,t)=0 \tag{8.16b}$$

Conveniently, this implies that

$$v'(x,t) = -v_0 \sin(\omega(\frac{-x-L}{c} + t)) \tag{8.17}$$

Noting that on the left side,

$$v_L(x,t) = v_0 \sin(\omega(\frac{x-L}{c} + t)) - v_0 \sin(\omega(\frac{-x-L}{c} + t)) \tag{8.18}$$

Using the fact that

$$\sin x - \sin y = 2 \sin \frac{x-y}{2} \cos \frac{x+y}{2} \tag{8.19}$$

The left side becomes

$$v_L(x,t) = 2v_0 \sin \frac{\omega x}{c} \cos \omega(t - \frac{L}{c}) \tag{8.20}$$

So, that the velocity can be written as:

$$v(x,t) = \begin{cases} 0 & x < L - ct \text{ and } t \in [0, \frac{L}{c}] \\ v_0 \sin(\omega(\frac{x-L}{c} + t)) & x > L - ct \text{ and } t \in [0, \frac{L}{c}] \\ 2v_0 \sin \frac{\omega x}{c} \cos \omega(t - \frac{L}{c}) & x < ct - L \text{ and } t \in [\frac{L}{c}, \frac{2L}{c}] \\ v_0 \sin(\omega(\frac{x-L}{c} + t)) & x > ct - L \text{ and } t \in [\frac{L}{c}, \frac{2L}{c}] \end{cases} \tag{8.21}$$

Which implies that

$$\epsilon(x,t) = \begin{cases} 0 & x < L - ct \text{ and } t \in [0, \frac{L}{c}] \\ \frac{v_0}{c} \sin(\omega(\frac{x-L}{c} + t)) & x > L - ct \text{ and } t \in [0, \frac{L}{c}] \\ 2\frac{v_0}{c} \cos \frac{\omega x}{c} \sin \omega(t - \frac{L}{c}) & x < ct - L \text{ and } t \in [\frac{L}{c}, \frac{2L}{c}] \\ \frac{v_0}{c} \sin(\omega(\frac{x-L}{c} + t)) & x > ct - L \text{ and } t \in [\frac{L}{c}, \frac{2L}{c}] \end{cases} \tag{8.22}$$

Note that $\rho(x,t) = \rho_0(1 - \epsilon(x,t))$ still applies.
Now, from here, the energy density of the system can be computed with

$$e(x,t) = \frac{1}{2}\rho v^2 + \frac{1}{2} Y \epsilon^2 \tag{8.23}$$

Thus, the total energy equals

$$E(t) = \frac{1}{2} \int_0^L \rho v^2 + Y \epsilon^2 dx \tag{8.24}$$

Using the definition of the density:

$$E(t) = \frac{1}{2} \int_0^L \rho_0 v^2 + Y \epsilon^2 - \rho_0 \epsilon v^2 dx \tag{8.25}$$

Now, the third term here can be neglected $v_0 << c$ is already true, and it can be plainly seen that the highest order terms are second order, while the third term is third order, so its contribution will be small[13].

So,

$$E(t) \approx \frac{1}{2}\int_0^L \rho_0 v^2 + Y\epsilon^2 dx \tag{8.26}$$

The details of the lengthy and tedious calculation will not be bothered with; it is quite involved. Needless to say that continued and very careful calculation results in:

$$E(t) = \begin{cases} \rho_0 v_0^2 c(\frac{t}{2} - \frac{1}{4\omega}\sin(2\omega t)) & t \in [0, \frac{L}{c}] \\ \rho_0 v_0^2 c[\frac{t}{2} - \frac{1}{2\omega}\sin(\omega(t - \frac{2L}{c}))\cos(\omega(3t - \frac{2L}{c})) - \frac{1}{8\omega}\sin(4\omega(t - \frac{L}{c}))] & t \in [\frac{L}{c}, \frac{2L}{c}] \end{cases} \tag{8.27}$$

So, if the state was initially $v = \epsilon = 0$ and $\rho = \rho_0$ at $t = 0$, subject to our right sinusoidal boundary condition, then the above results apply.

For this simulation, the following settings apply:

Simulated on $[0, 1]$
Nodes: 101
Particles/cell: 2
$Y = \rho_0 = 1$
$v(x = 1, t) = 0.01 * \sin(10 * t)$
$v(x, t = 0) = \epsilon(x, t = 0) = 0$
$\rho(x, t = 0) = \rho_0$
Timesteps/second = 100,000

[13]Importantly, this also shows that the energy is a second order contribution

8.2.2 Results for Static Particles

Running this 'shaking wave' problem for static particles nets among the following:

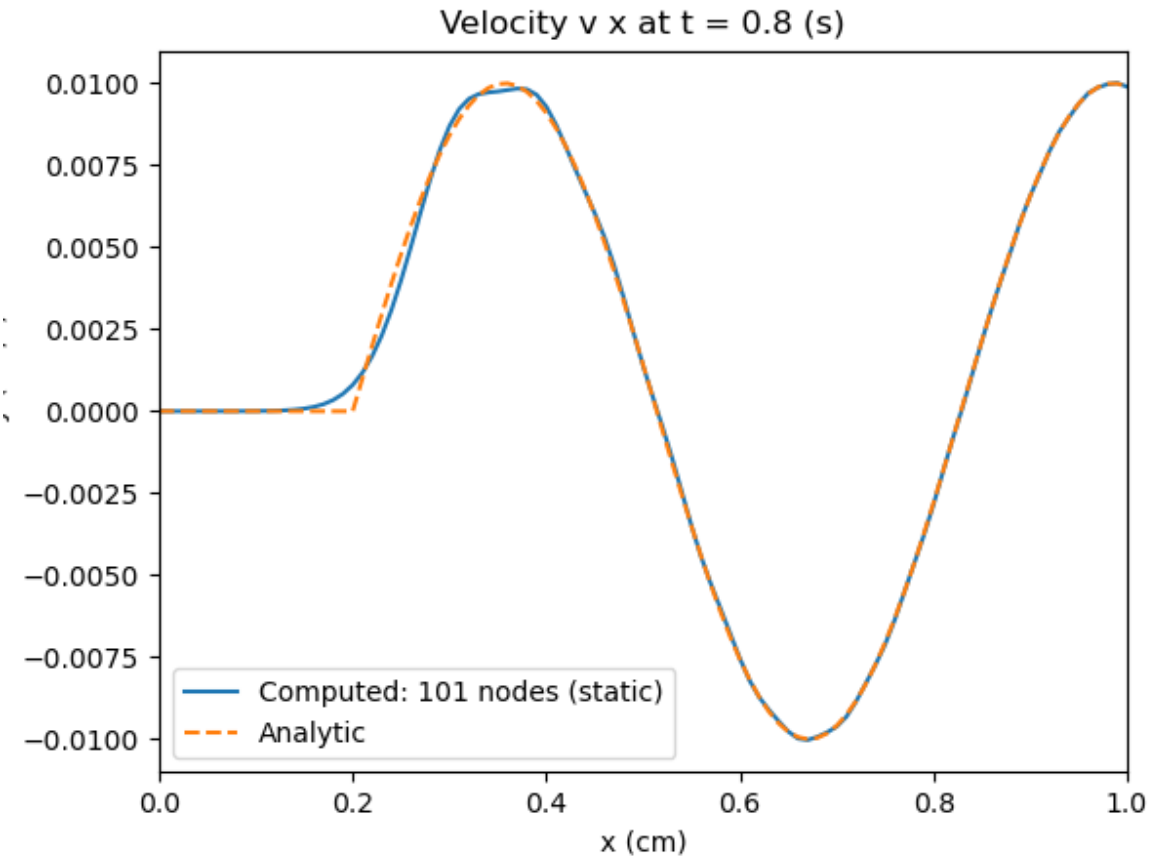

Figure 9: The velocity of the system at $t = 0.8s$. This occurs before the wavefront impacts the left boundary.

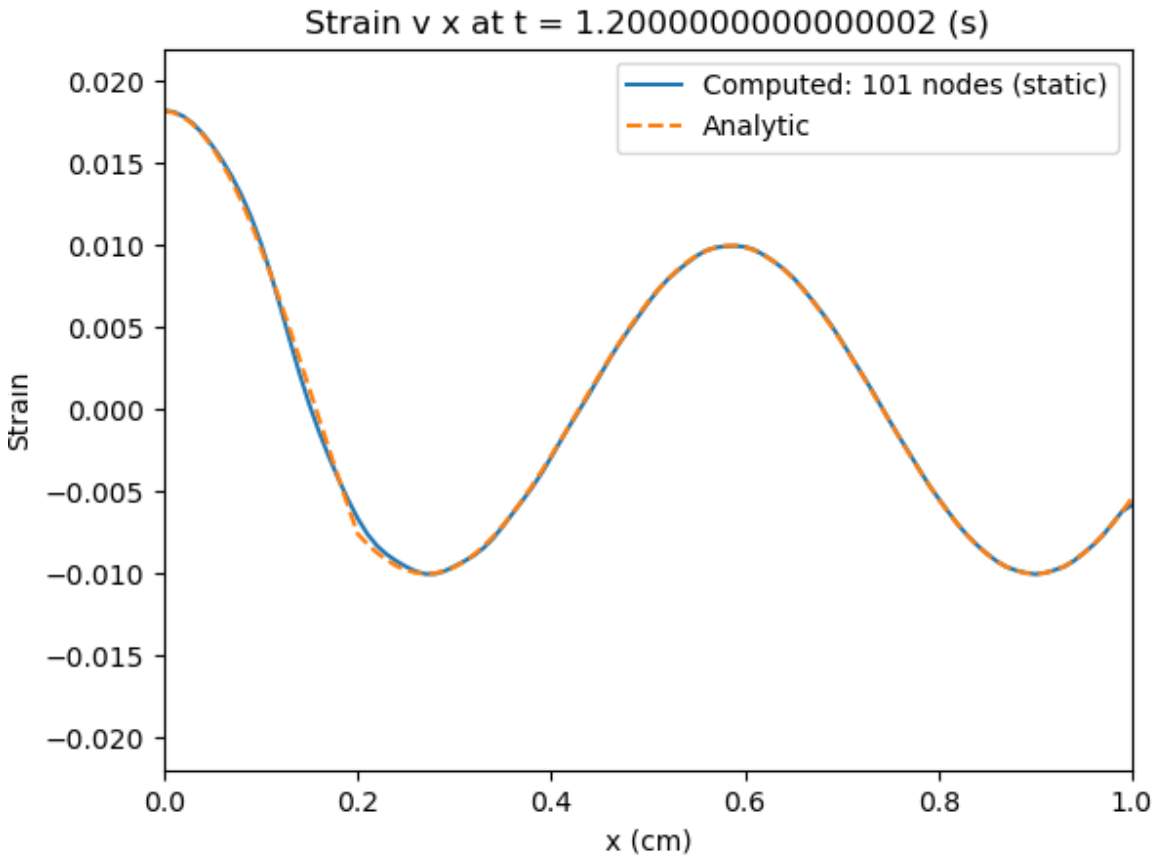

Figure 10: The strain of the system at $t = 1.2s$. This is shortly after the wave reflects off of the boundary. The constructive interference between the reflected and incoming wave is visible.

Indeed, in this static case, the strain and velocity line up with the analytic solutions very well.

Of course, beyond these quantities, the total energy of the system is something that the analytic solution presented covers, so looking at the energy:

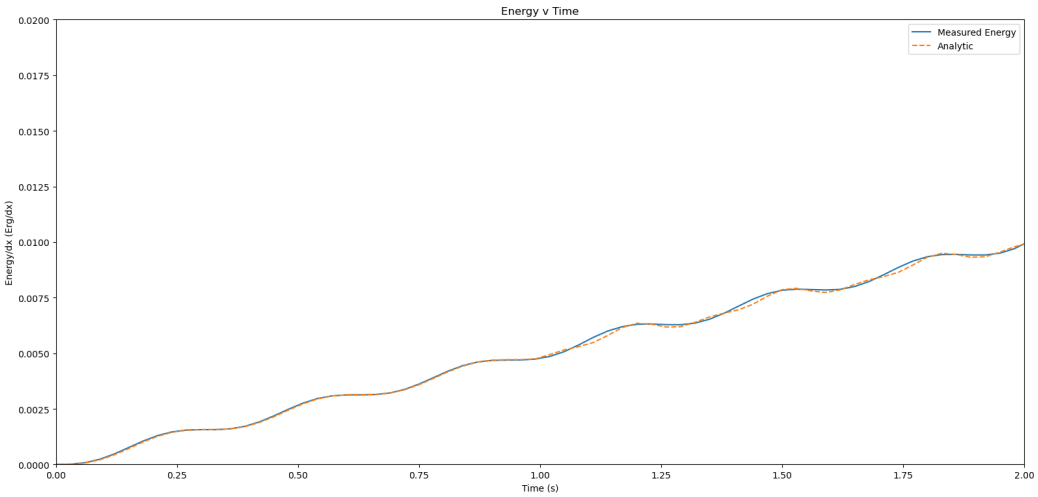

Figure 11: The energy of the system as a function of time. Note a somewhat linear behaviour.

It is clear that the total energy lines up closely for the first 1 second (before the reflection), and then it begins to deviate, but not by too much. This is not entirely unexpected, as the velocity and strain in the simulation are updated using Euler's Method, which means that the these quantities are subject to a small overshoot, causing an increase in energy over time.

8.2.3 Results for Moving Particles

For moving particles, the results also look familiar.

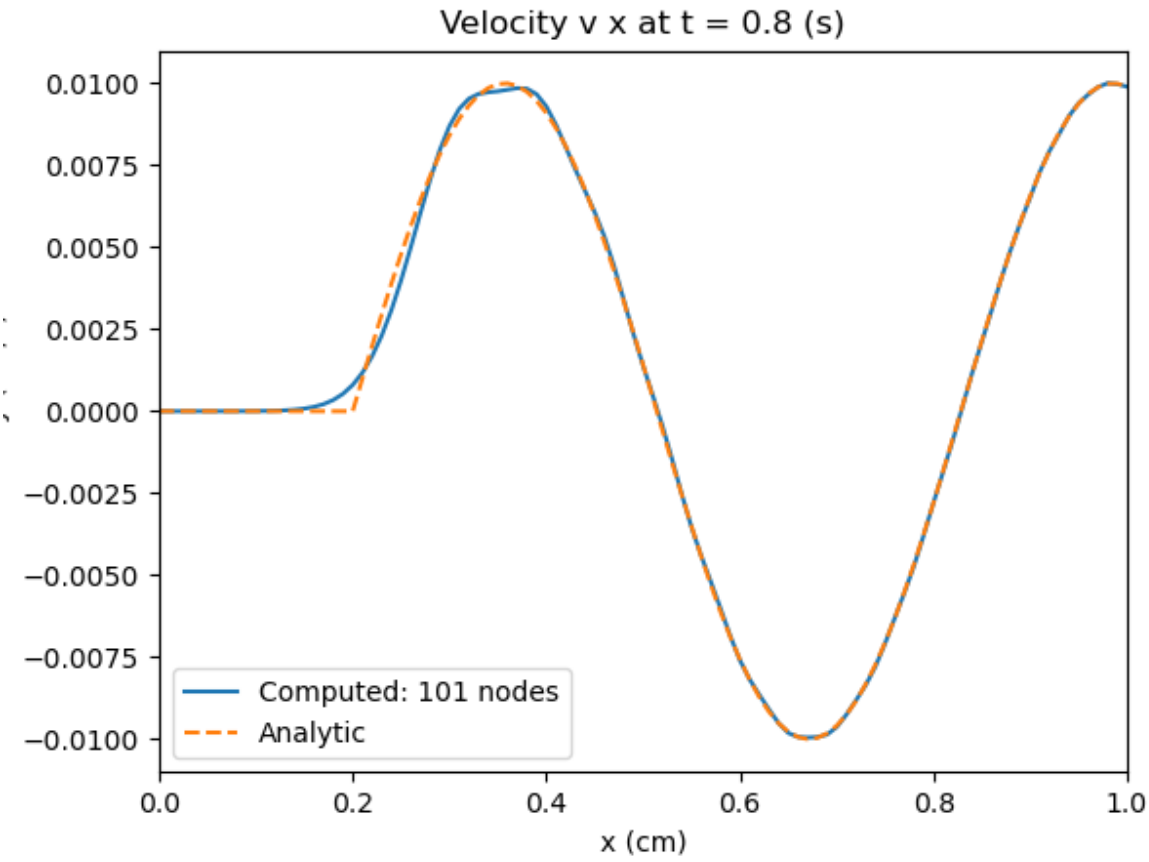

Figure 12: The velocity of the system at $t = 0.8s$. This occurs before the wavefront impacts the left boundary.

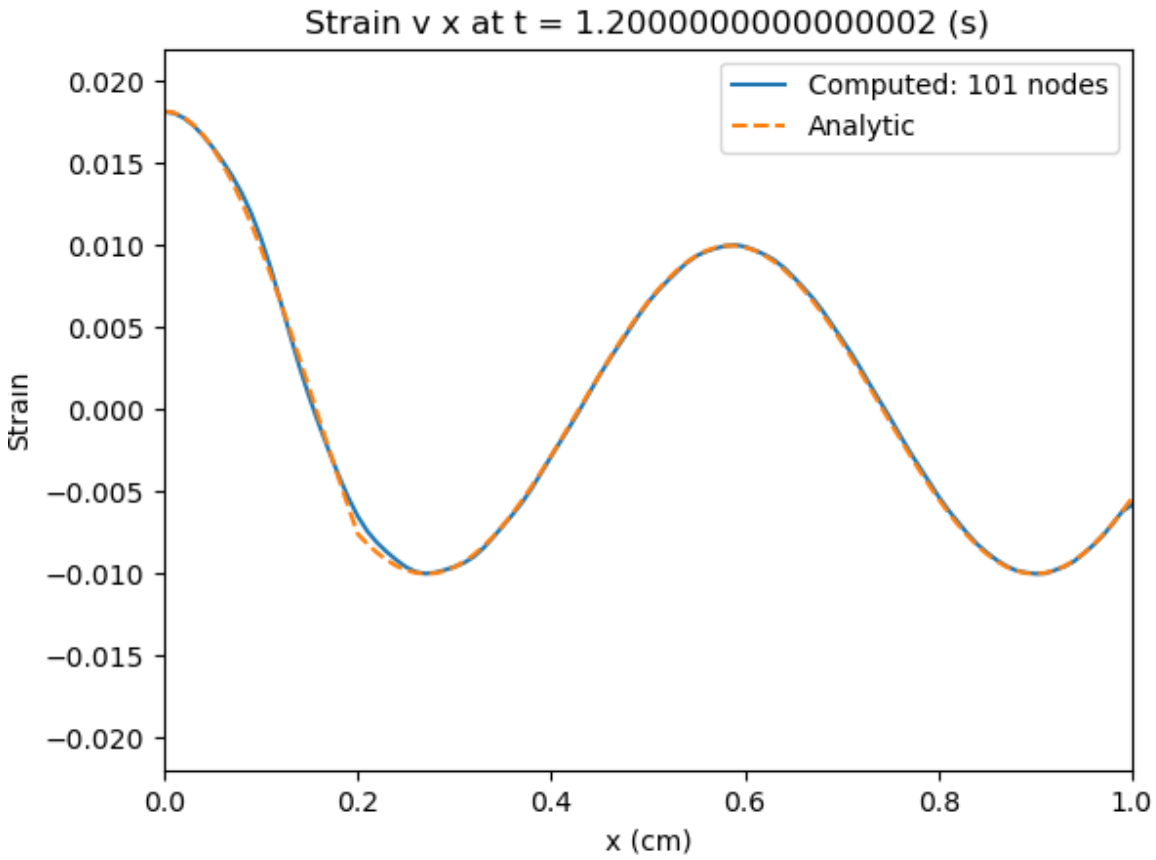

Figure 13: The strain of the system at $t = 1.2s$. This is shortly after the wave reflects off of the boundary. The constructive interference between the reflected and incoming wave is visible.

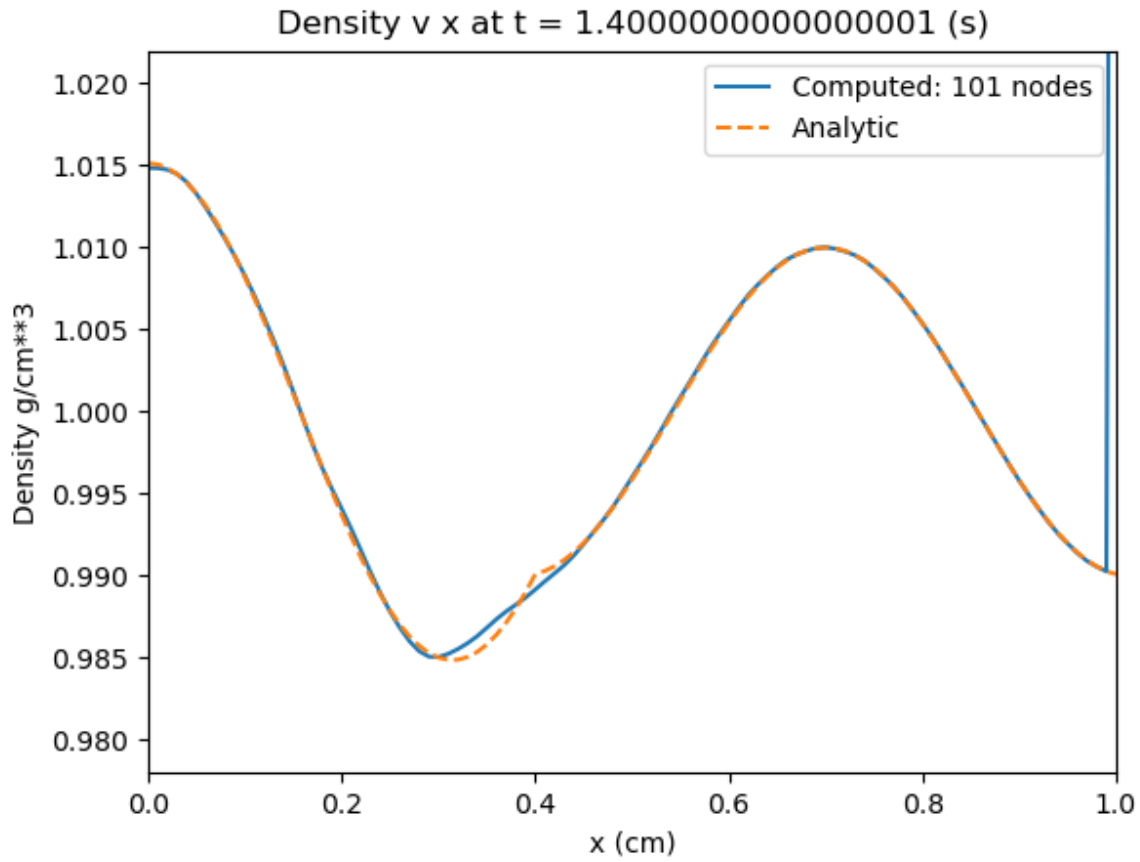

Figure 14: The density of the system at $t = 1.4s$. Note the slight kink in the analytic solution. Also note the jump on the far right.

There is nothing to discuss about the velocity or strain for this system; the results are as expected. The results for the density at the given time very nearly match those of the analytic solution, but there are some notable discrepancies. For one, there is some mismatching right near the wavefront of the reflected wave, but this could be due to an insufficient number of nodes used to generate the solution.

However, the jump in the density at the right of the system implies cell crossing is either immenent, or has already begun. This jump is caused by particles bunching up right next to the position of the node, causing the density to shoot up. As per the MPM formulation, the node locations serve as cell boundaries, so if these particles (already close to the cell boundary) hop to another cell, some notable disturbances will be generated in the code.

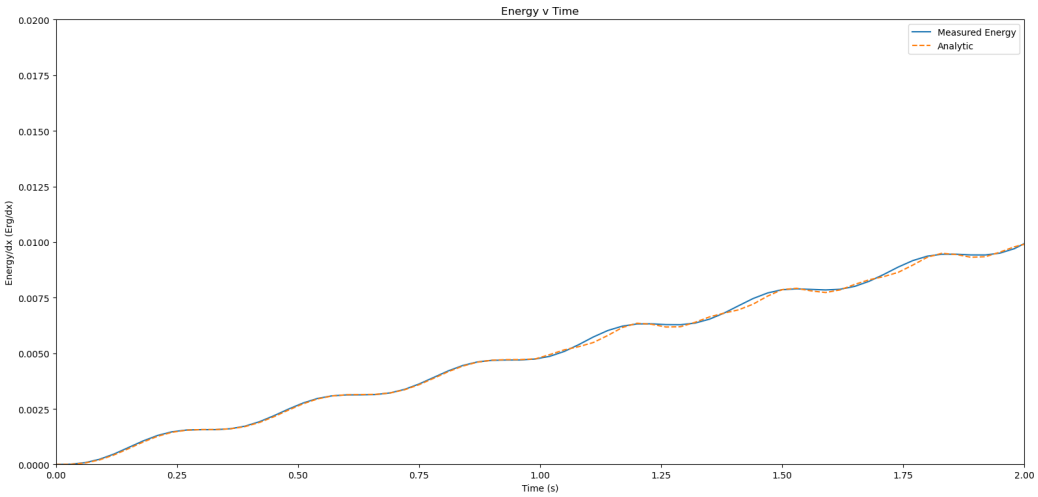

Figure 15: The total energy of the system as a function of time.

Much like with the static case, the total energy in the case of moving particles closely matches that of the analytic solution pre-reflection, and there are some small divergences after the reflection, possibly the result of the change in density term being neglected in the calculation of the analytic solution, or through the use of Euler's Method to evolve the system.

8.3 A Small Shock

The real power of computational physics is to be able to handle extremely complicated systems that cannot be solved analytically (except maybe in one or two special situations).

Shock waves are just the very tip of the iceberg, where, alongside standard boundary conditions, the discontinuities in the system itself imply that there are boundary conditions between the various sections of the system.

Indeed, the dynamics of such situations are wildly complicated and far beyond the scope of this project.

Regardless, for a weak shock[14], the front of the wave (the shock front), should move with a velocity c. That is:

$$x(t) = x_0 + ct \tag{8.28}$$

Where x_0 is the $t = 0$ position of the shock.

In this simulation, the following properties apply: For this simulation, the following settings apply:

$$\begin{gathered}
\text{Simulated on } [0, 1] \\
\text{Nodes: } 401 \\
\text{Particles/cell: } 2 \\
Y = \rho_0 = 1 \\
v(x, t = 0) = 0 \\
\rho(x, t = 0) = \rho_0 \\
\epsilon(x < 0.5, t = 0) = -0.001 \\
\epsilon(x > 0.5, t = 0) = 0 \\
v(x = 0, t) = v(x = L, t) = 0 \\
\text{Timesteps/second} = 800{,}000
\end{gathered}$$

Note the extremely small perturbation in the strain; larger strain differences can produce shocks that travel beyond a speed of c in addition to MPM failing due to cell crossing.

Other than this, note the large number of cells and the high amount of timesteps taken to simulate the system for a given amount of time. A large number of nodes is needed to produce clear and sensical results, so very small timesteps are needed to satisfy the constraint that $dt << \frac{dx}{c}$.

[14] That is, the shock needs to be very small; below 1 in units, and as close to 0 as we can get.

Here is a plot of the initial strain:

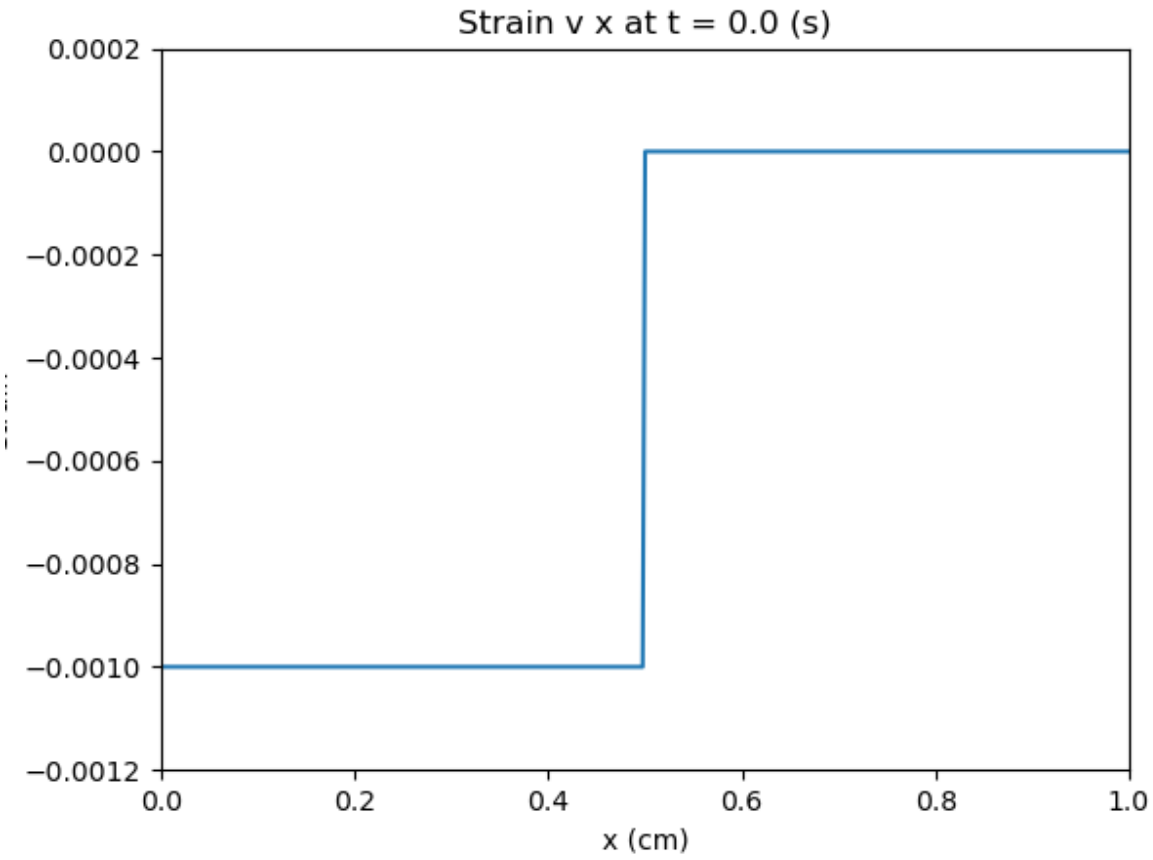

Figure 16: The $t = 0$ strain of the system.

The reason for picking the left strain to be negative is fairly easy to understand. $\epsilon = \frac{\partial u}{\partial x}$, so this implies that u x, meaning that the displacement increases linearly in the disturbed zone. This is somewhat analogous to stretching a string or rubber band. The negative sign on ϵ ensures that the displacement is to the left of $x = 0.5$, so it is stretching, and not compression.

8.3.1 Results for Static Particles

Here are results for evolving this shock in the strain in time.

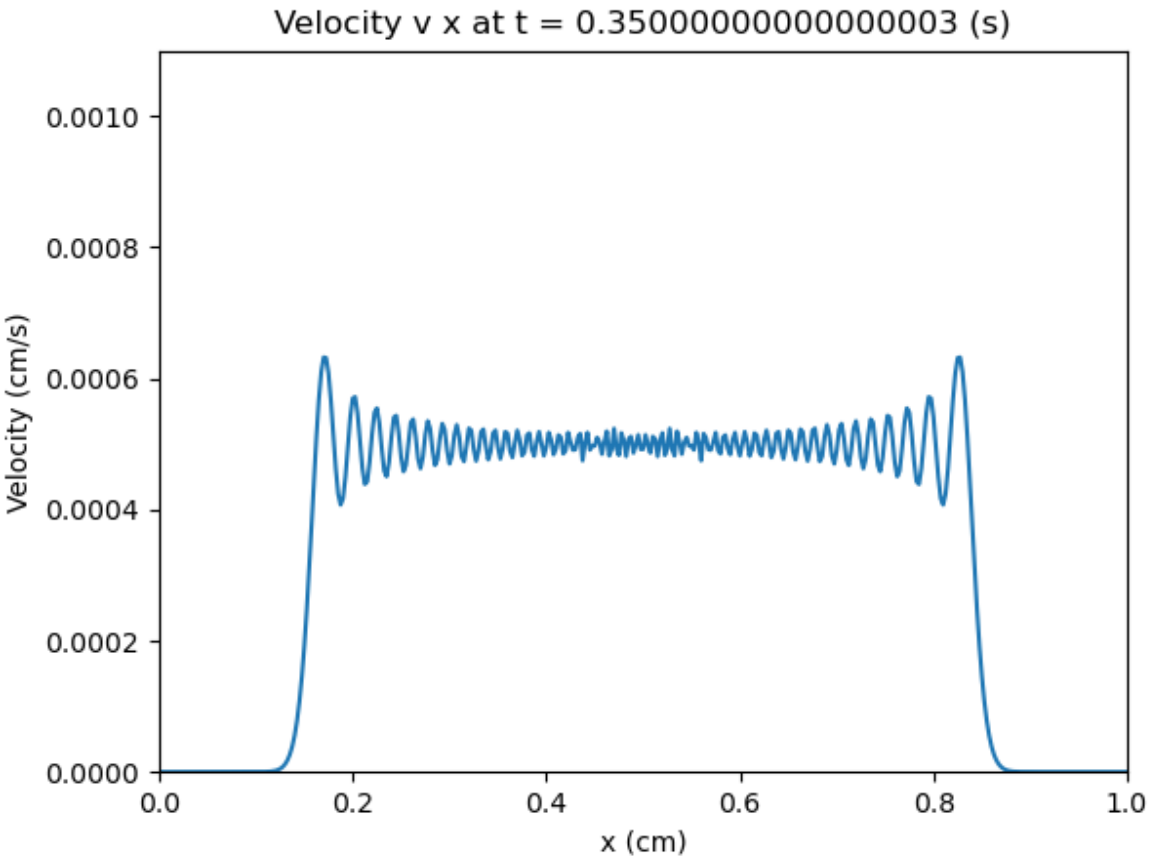

Figure 17: The velocity at $t = 0.35s$. Note how a there are disturbances in the velocity, somewhat analogous to sound waves, behind the shock front.

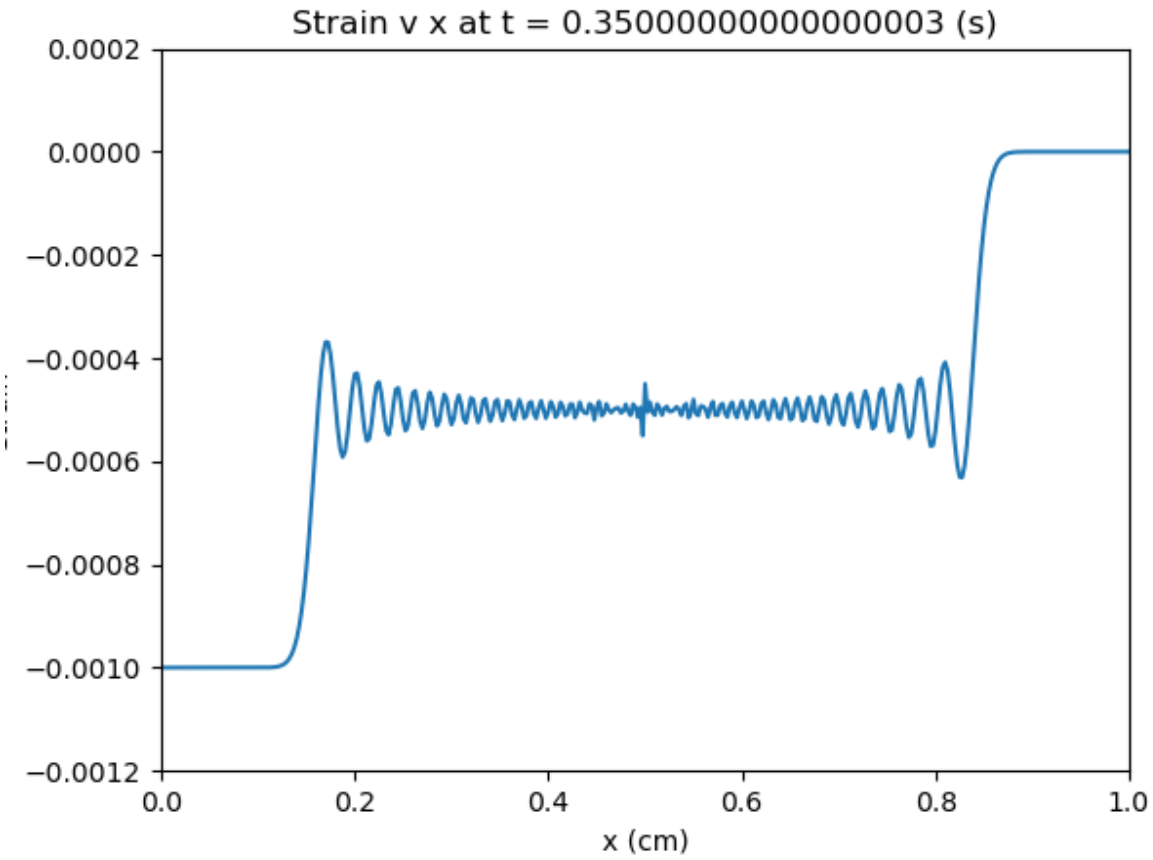

Figure 18: The strain at $t = 0.35s$.

Note the very interesting waves that are formed between the shock fronts. They originate at the initial shock location ($x = 0.5$) and propagate from there.

8.3.2 Results for Moving Particles

For moving particles, the density is available to see.

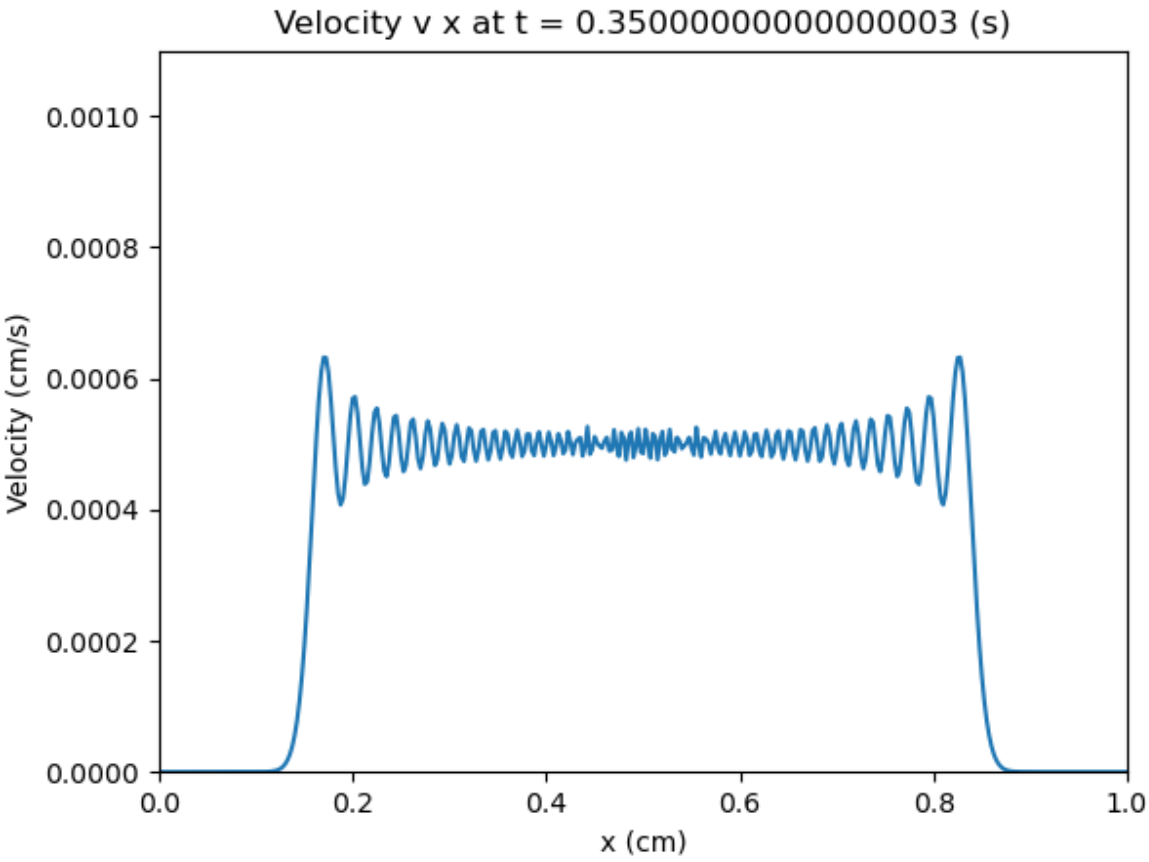

Figure 19: The velocity at $t = 0.35s$.

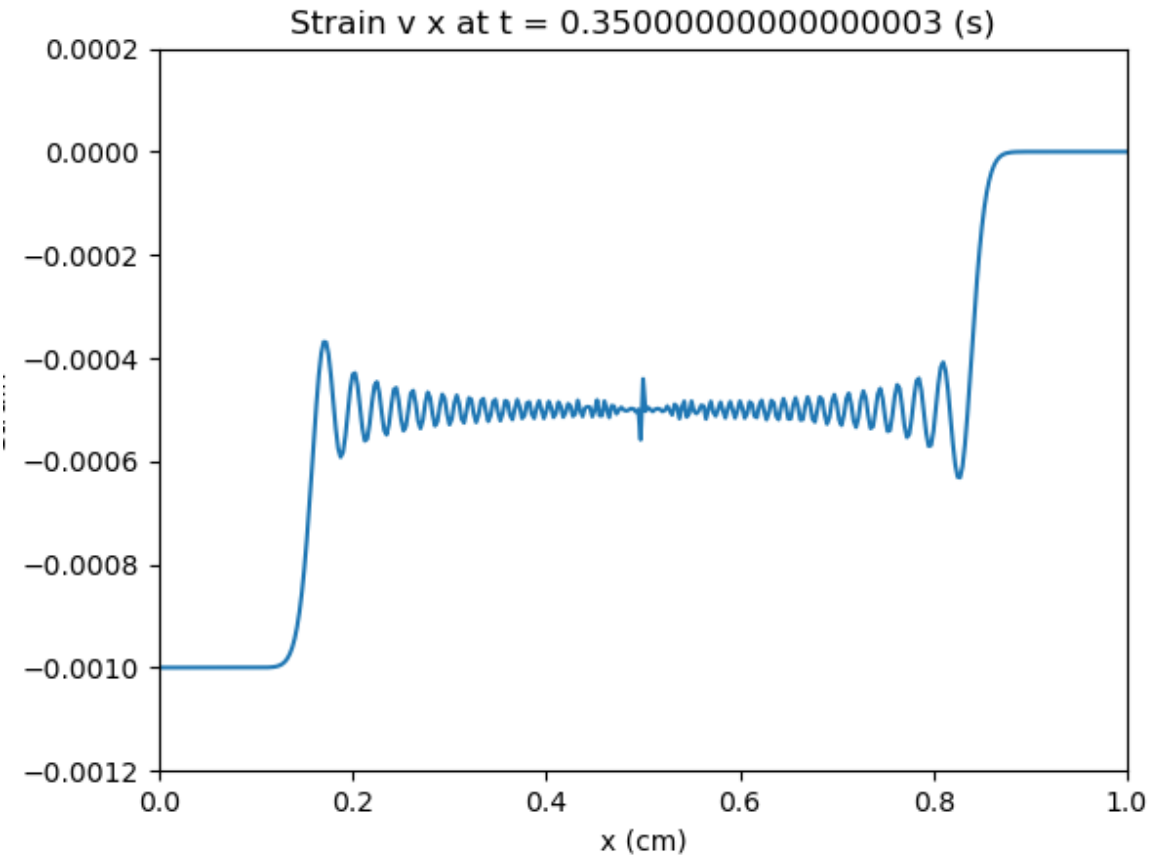

Figure 20: The strain at $t = 0.35s$.

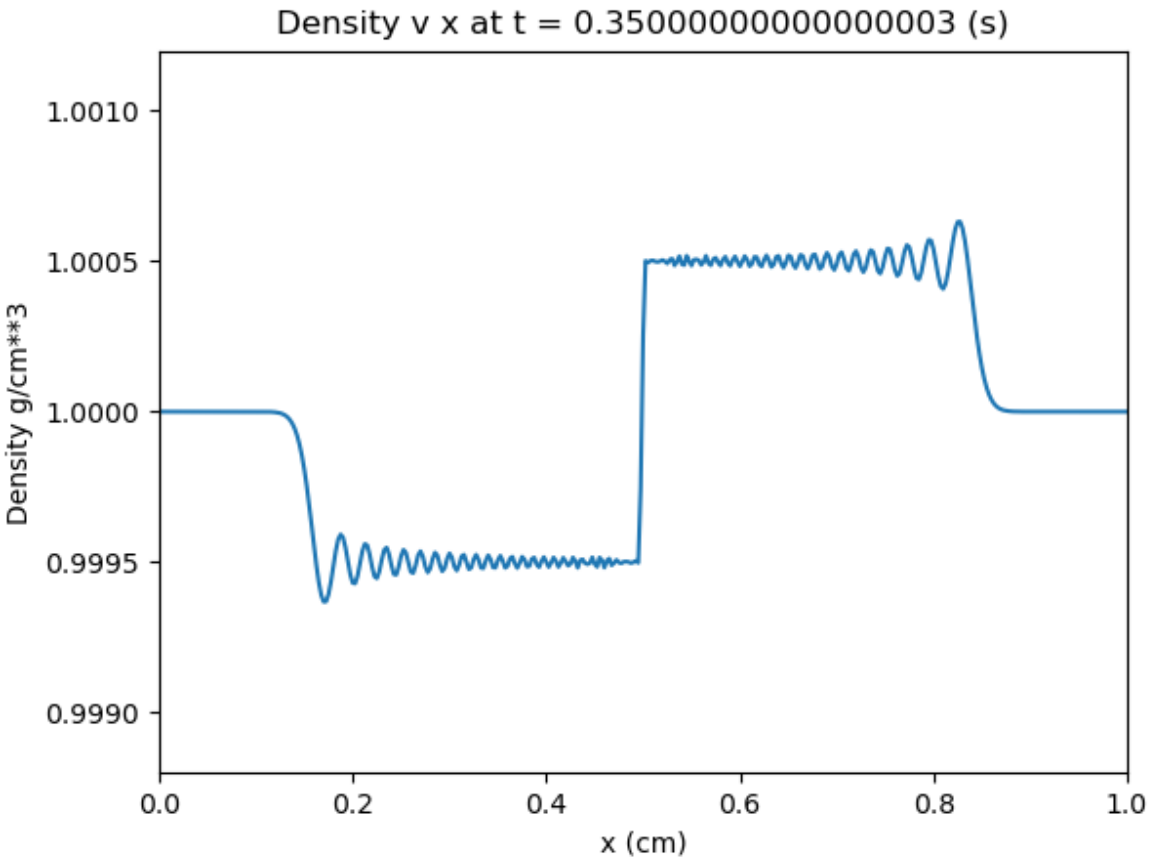

Figure 21: The density at $t = 0.35$. Note the still-present discontinuity in density.

Being able to see the density reveals that these is a discontinuity in the density that is formed after the system begins to evolve. Note how the total mass of the system is conserved; the velocity at the boundaries is set to be 0[15], so there is no outflow of mass, and so $\int \rho dx$ can be done by inspection and immediately seen to conserve the total mass.

The interesting part about this shock problem is that there was no coded behaviour about how shocks ought to behave; the program handled it via its normal time evolution routines and was able to produce spectacular and unique results.

[15]Boundary conditions are irrelevant to this problem as long as the shock does not reach the boundary.

Now, from these plots, it is possible to look at the shock front of the system, eyeballing its x position and using the current time to compute the average speed of its propagation.

However, using the *csv* data produced by the simulation, the wavefront of the system can be found to a good approximation by considering the above results. Finding the max/min value of the density, or the max value of the velocity in either the left or right half of the system corresponds to the shock front, so a simple program can be written to find this value for a given element of time.

The shock front position can then be plotted with its current time, and the procedure repeated for many times (before the shock impacts the boundary). Indeed, doing this results in a plot of the form:

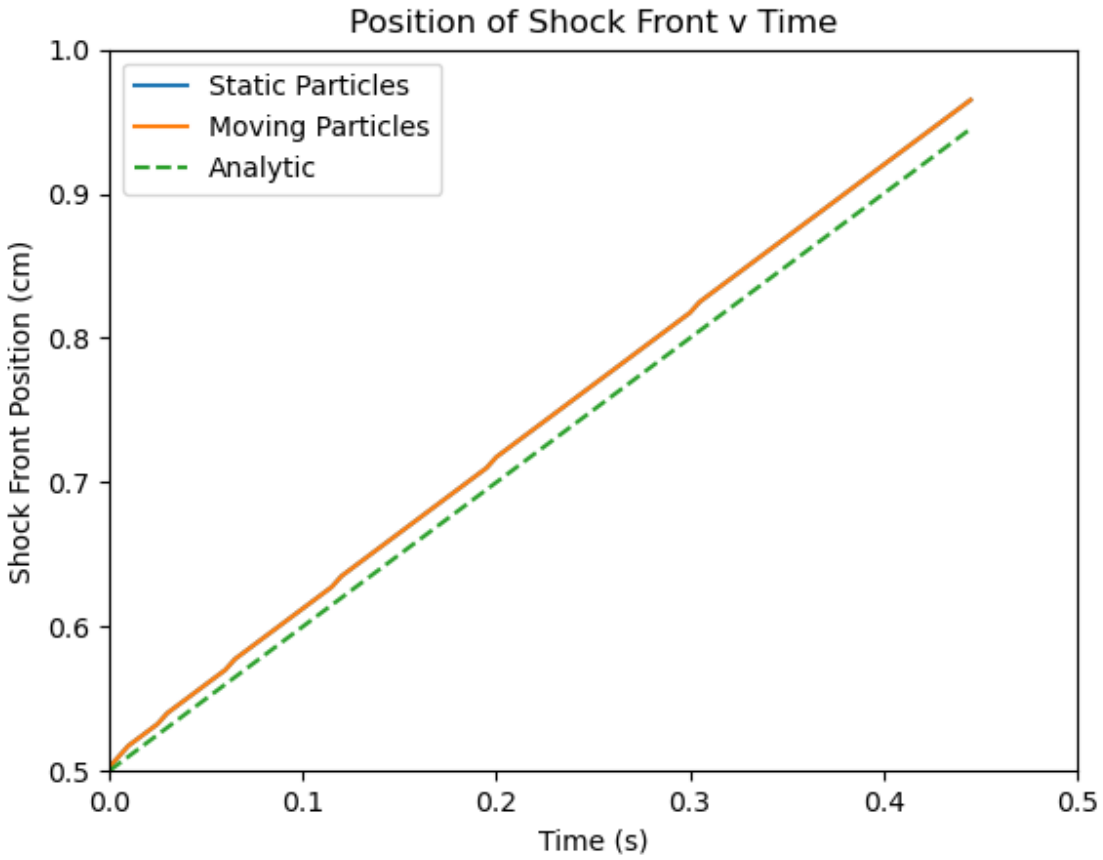

Figure 22: The shock front's position as a function of time for a left disturbance in the strain of -0.001.

Note how the case of static and moving particles fold into one; their data overlaps in their entirety. Also note that the shock front position most certainty overshoots the analytic solution.

Indeed, in order to find the velocity of the shock, the following steps were taken for the set of positions produced:

$$v_i = \frac{x_{i+1} - x_i}{dt} \tag{8.29a}$$

$$v_{shock} = \frac{1}{N}\sum_{i=0}^{N-1} v_i \tag{8.29b}$$

While this method is not as accurate as a proper linear fit tool, it is nonetheless a quick and easy way to get a value of the shock velocity.

Using the data set produced, the numerical result is

$$v_{shock} = 1.039\frac{\text{cm}}{\text{s}} \tag{8.30}$$

For this system, this means

$$v_{shock} = 1.039c \tag{8.31}$$

While the shock's velocity is not the predicted value of exactly c, it is about 4% away from this value.

Indeed, this problem can be run again with a greater strain shock, indeed using a left strain of -0.01 nets the following shock front plot:

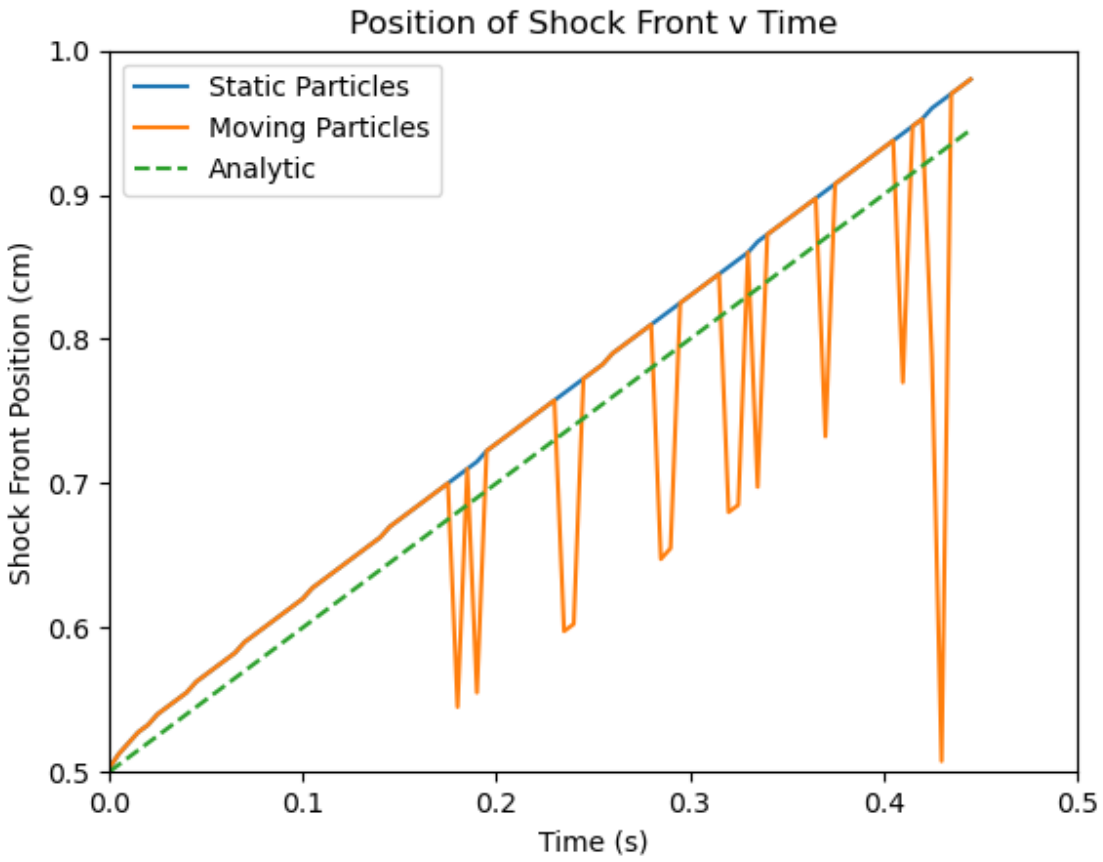

Figure 23: The shock front's position as a function of time for a left disturbance in the strain of -0.01.

The strange spikes in the moving particle data is due to cell crossing; cell crossing as produced densities and/or velocities that cause the shock-front finding program to pick unphysical values for the shock front.

Regardless, doing a careful analysis on the static particles case reveals that

$$v_{shock} = 1.073 \frac{\text{cm}}{\text{s}} \tag{8.32}$$

Indeed, a stronger shock has produced a faster moving shock. At the very least this seems to support the idea that a weaker and weaker shock will propagate at a speed closer and closer to c.

9 Bonus: ADMPM in C

What follows is a description of the same MPM code, modified to run in C.

Given that C and Java are generally not of the same language type (procedural versus object-oriented), there are a few differences. For example, C does not have a linked list by default[16], and so data structures generally have their sizes set at compile time and are quite inflexible to changes.

Importantly, the *C* code written is bare, lacking the debugging features that the Java code contains, but in terms of numerical capabilities it is on par with the Java code.

In this section, only a cursory glance at the *C* code will be presented; its ideas are equivalent to that of the Java code, with syntax being the only real difference between the two.

[16]That means it is possible to code a linked list manually.

9.1 mpm_solve.c

This is the main program where everything comes together and is solved.

Things start by setting the time initially to 0, then declaring two arrays of structs[17]. One array holds the nodes, and the other the material points.

Then, the output files are created and the headers written.

Next, the mesh gets initialized, and then the mesh is used to initialize the material points.

Then, the initial node masses get computed before evolving the system in time.

Over time, the data is written periodically, while updating the particle strain, velocity, and stress.

Afterwards, the node density, strain, and stress is written before advancing the system in time.

After finishing the simulation time, the program cleans up, doing a final write of the data and then closing the output files.

Note how the 'update' functions no longer take strings as arguments; this is due to the fact that strings are not particularly well defined in C. Instead, these functions use a *char* to shore up this weakness. e stands for ϵ, or the strain, s for σ, for stress, r for ρ: the density, and so on.

[17]Structs are a precursor to classes; they can essentially only hold variables, and cannot hold functions or overload operators.

9.2 accelerations.h

This file simply holds the external accelerations of the system, a function of both time and a node struct. Much like in the Java program, passing a node as an argument gives easy velocity-dependence to a force.

9.3 boundary_conditions.h

The boundary conditions file contains a single function that takes in the whole list of nodes, the time, and an identifying character that sets the boundary conditions of a certain type.

The velocity boundary conditions are the only ones implemented implemented (which should be all that is required for basic fluid/solid simulations), but the functionality can be easily extended.

Note how the set_boundary function is called and it handles the two boundary conditions in one function call, instead of fixing the two boundaries in two separate lines of code like was done in the Java program.

9.4 data_write.h

This file contains the functions we need to write the output files for analysis.

It contains a header-writing function; there is some warning about write_header; it assumes that the the array of strings used to write the headers are of the same size, for both the nodes and the material points. Fair warning!

The other two functions are familiar node and material point writing functions that each write a line in the CSV for each call of the function.

9.5 initial_state.h

Here the initial states of the system are kept. As base, it is the initial density, velocity, young's modulus, strain, and stress.

By default, the program does not support use of a non-constant young's modulus, so it will require some rewriting to get it to work in the program, so that the young's modulus evolves in time and space.

9.6 initializations.h

In this file, the functions that initialize the nodes and material points are stored. In order to initialize the particles, the nodes to be initialized first.

Again, by default the system assumes a constant Young's Modulus, in addition to having the constitutive relation hard-coded into the system.

9.7 material_point.h

The material point struct contains several fields that describe a material point in its entirety. Containing fairly immutable elements, such as length and mass, to the most important elements: strain, velocity, and so on, and with the two nearest nodes to a given material point.

9.8 mpm_math.h

This file contains a single function, which finds the nearest nodes to a given material point. Even though this file only has one function, it is a great container to store extra functions, via adding more and declaring them in the header file.

9.9 node.h

The node.h file contains the declaration of the node struct, completely describing the nodes of the system. Moreover, it contains two function declarations, one for a node's shape function and its derivative.

The limitations of C means that a struct cannot contain a function[18], so a given node is passed as an argument to our shape functions. This is an equivalent procedure to putting a function inside of a class.

[18]Note that in C++ a struct can contain functions.

9.10 sim_update.h

Here lives all the functions that are used to update the system in time. There are functions to compute the node masses, and to move the particles (and these functions are only called if the particles are allowed to be moved).

There is also the update_velocity function that updates both the particle and node velocity in a single procedure.

Finally, there are the various code blocks that update the nodes or particles to the next timestep. Given limitations of strings in C, char values are used to identify what quantity to update.

9.11 solver_options.h/c

The solver_options files set various attributes of the program at compile time.

The actual *C* file stores the headers of the output *csv* files, but their existence is declared in the header file.

Indeed, the header file contains all of the other definitions needed to fully customize the system's resolution and bounds. Note some important differences over the Java version though.

For one, in order to have boolean values, the *stdbool* library needs to be imported; instead, an integer is used as a stand-in (which is how *C* is designed to work; booleans were a later addition). 0 is false, and anything else is true.

Next, there is extensive use of the $\#define$ preprocessor command. The reason for this is that the compilers require these defines in order to use global variables to set the sizes of arrays. Use of *const*, *extern*, and so on simply is not considered strong enough[19], unlike in Java where a variable was simply declared as $final$.

[19] In C++, *const* alone is strong enough for this purpose!

10 Bonus: Basic Benchmarking

So far, two codes for ADMPM have been presented, the only notable difference being the language they have been written in.

Importantly, it is not necessarily the language itself that contributes to how quickly the code runs (especially in the case where the codes are nearly identical), but rather it is how efficiently preprogrammed packages and functions are coded, and how well the compiler/interpreter eventually converts the source code into CPU instructions.

Note that the RAM of the system is going to affect the performance of a program; if the RAM gets full, there will be a noticeable performance hit as the computer either completely fails to run the program correctly, or as if shifts around data that needs to be accessed. I will not be tracking the RAM usage of the programs.

However, note that the computer used to run the MPM solver has the following specifications:

OS: Ubuntu 20.04.2 LTS (run on Hyper-V 10.019041.1)
Processor: Virtualized 1-Core x86_64 CPU @ 3.30 GHz
RAM: 15.6 GB
Java Version: 11.0.10
C Compiler: gcc 9.3.0

The program in C is compiled with gcc without optimizations in one run, and with -03 optimizations in another. Java's compiler automatically optimizes the program, and compiles it down to bytecode which is interpretted at runtime.

As a benchmark, the following initial and boundary conditions have been used:

$$v(x, t = 0) = 0 \tag{10.1a}$$

$$\epsilon(x, t = 0) = 0 \tag{10.1b}$$

$$\rho(x, t = 0) = 1. \tag{10.1c}$$

$$v(0, t) = 0 \tag{10.1d}$$

$$v(1, t) = 0.01 \sin 10t \tag{10.1e}$$

$$\text{Young's Modulus } = 1 \text{ everywhere} \tag{10.1f}$$

$$\text{Computational Domain: } [0, 1] \tag{10.1g}$$

$$\text{No Data Dump File} \tag{10.1h}$$

10.1 Run 1

Nodes: 51
Particles/cell: 2
Max time: 3
Timesteps: 300,000
Recording Frequency: 300
Moving Particles

Runtimes
Java: 6.849 (s)
gcc : 6.904 (s)
gcc -O3 : 3.350 (s)

10.2 Run 2

Nodes: 101
Particles/cell: 2
Max time: 6
Timesteps: 600,000
Recording Frequency: 3000
Moving Particles

Runtimes
Java: 13.300 (s)
gcc : 26.762 (s)
gcc -O3 : 12.523 (s)

10.3 Run 3

Nodes: 201
Particles/cell: 4
Max time: 10
Timesteps: 1,000,000
Recording Frequency: 5000
Moving Particles

Runtimes
Java: 76.529 (s)
gcc : 176.020 (s)
gcc -O3 : 82.689 (s)

10.4 Run 4

Nodes: 51
Particles/cell: 2
Max time: 3
Timesteps: 300,000
Recording Frequency: 300
Static Particles

Runtimes
Java: 6.102 (s)
gcc : 5.984 (s)
gcc -O3 : 2.980 (s)

10.5 Run 5

Nodes: 101
Particles/cell: 2
Max time: 6
Timesteps: 600,000
Recording Frequency: 3000
Static Particles

Runtimes
Java: 10.928 (s)
gcc : 22.723 (s)
gcc -O3 : 10.976 (s)

10.6 Run 6

Nodes: 201
Particles/cell: 4
Max time: 10
Timesteps: 1,000,000
Recording Frequency: 5000
Static Particles

Runtimes
Java: 62.948 (s)
gcc : 148.401 (s)
gcc -O3 : 69.065 (s)

10.7 Results

One thing that should be noted is that these results were obtained for only a single run of the program; the runtime will vary slightly every time it is executed. An ideal benchmark would run the same program many times to find the average execution time.

However, it should nonetheless be obvious that there does not seem to be a notable advantage from running this program in C (at least at our low level of computation). This is quite shocking, as C is commonly known to be faster than Java. It seems like enabling the max optimizations in gcc is able to get the C and Java code execution times up to parity, but other than this, it seems that Java's built-in compiler optimizations can do far better than just porting the code over to C.

In all fairness though, the weaknesses of Java's memory management may come to fruition when attempting to run this MPM solver with huge numbers of nodes, and on a large-scale, C may come to dominate in performance. However, for these simple, quickly executing programs, it seems as if Java ought to be the prefered language and runtime environment.

Language choice aside, it can be definitively concluded that running the MPM solver with static particles takes notably less time than it does to run it with moving particles (exactly as was expected).

References

[1] University of Chicago Flash Center for Computational Science. FLASH User's Guide. page 206, 2020.

[2] Duan Zhang, Qisu Zou, W. Brian VanderHayden, and Xia Ma. Material point method applied to multiphase flows. *Journal of Computational Physics*, 2008(227), 2006.