An Efficient Implementation of Pressure Field Models

Vincent Huang
vvhuang@mit.edu

Franklyn Wang
franklyn_wang@college.harvard.edu

Eric Zhang
ekzhang@college.harvard.edu

***Abstract*—Traditional rigid-body physics models often produce unstable or inaccurate results for contact force simulation, while pressure field models, as introduced by [EDSR19], are more realistic but less commonly used because of the implementation difficulty. We implement a minimal physics simulator for contact forces on tetrahedral meshes from scratch in pure Julia [BEKS17], using the pressure field model, while incorporating automatic differentiation libraries like [IEF+19] in our setup. We re-derive and efficiently implement methods for force and torque computation within the pressure field framework, while also accelerating the algorithms using geometric space-partitioning structures, such as [BKSS90], for increased practical performance. Our work is open source and can be found in the `Hydroelastics.jl` package.**

I. Introduction

Reliable physics simulators are essential in training robots to perform manipulation tasks. However, we saw in class that simple rigid-body physics simulators with point forces and constraints are ineffective for modeling soft contact, as they run into a number of issues. Firstly, a lack of differentiability between the inputs and outputs of the simulation makes it difficult to use end-to-end gradient descent methods to train robots in simulators. Secondly, rigid body dynamics cannot easily replicate soft contact scenarios. For instance, simply having a ball fall onto the ground and remain at rest requires carefully choosing damping coefficients to stabilize the position of the ball. Similarly, it is difficult to have a gripper grasp a soft object with friction without having the object fly out of the effector if its width becomes slightly smaller than the object's diameter.

Another issue in previous hydroelastic force simulators is that their complexity precludes an ease of understanding. The original pressure field contact implementation is dense and difficult to parse for non-domain experts, since it relies on very complex libraries like `RigidBodyDynamics.jl`. Furthermore, neither of these libraries is actively maintained, with the last code activity being over 2 years old, and the pressure field contact library produces errors when running on modern versions of Julia. Therefore, it is of interest to have a simpler, minimal reference implementation that still captures the essence of pressure field models.

Our motivation in this project is to write a minimal physics simulator that addresses the limitations of rigid body dynamics, both to learn about soft-contact physics simulation algorithms and accelerate them with our own algorithmic experience. As a secondary goal, we aimed to extend hydroelastic simulators with modern automatic differentiation systems like Zygote.jl to implement differentiable simulators, which could theoretically be used later in tasks like learned locomotion. The pressure field framework is particularly robust and amenable to automatic differentiation due to continuous contact forces. We discuss the background for both of these topics in more detail in the next section.

II. Related Work

In this section, we discuss background related to our project topic, specifically past work done by others.

A. Automatic Differentiation

Automatic differentiation (AD) describes a collection of techniques at the intersection of machine learning and programming languages research. Generally, the goal of automatic differentiation is to algorithmically evaluate derivatives of composite functions, expressed via code, in a way that allows for synthesizing complex functions from simple primitives. This allows programmers to efficiently compute derivatives for expressions that would be intractable to symbolically differentiate, due to a combinatorial explosion in size from repeatedly applying the product and chain rules.

There are two primary modes of automatic differentiation, both with their own benefits and drawbacks:

- **Forward-mode:** Given scalar x and a vector function $\mathbf{y} = f(x)$, compute $f'(x)$. This is conceptually simpler and has a nice mathematical interpretation.
- **Reverse-mode:** Given a scalar function $y = f(\mathbf{x})$, compute the gradient $\partial y / \partial \mathbf{x}$. This is more complex, but it is also more useful in machine learning for training neural networks via gradient descent (also known as *backpropagation*).

Many mainstream programming languages, like Python, support automatic differentiation by allowing the programmer to construct dataflow graphs in an embedded computation language, like TensorFlow [ABC+16], or

explicitly supports a set of eagerly-evaluated differentiable library functions that generate dynamic computation graphs, like PyTorch [PGC+17]. These have the property that they can only differentiate code that is aware of the backend, requiring context-aware functions like `tf.matmul` and `tf.transpose` instead of more familiar primitives.

In contrast, Julia has a very powerful homoiconic metaprogramming system that allows library authors to directly implement source-code transformations on the language itself. This allows programmers to differentiate general functions written in pure Julia, regardless of whether they are specifically designed with differentiable libraries in mind. We used Zygote.jl in this project, which is a source-to-source automatic differentiation system that supports both forward-mode and reverse-mode AD, by writing transformations that operate on the static single assignment (SSA) intermediate representation form in the Julia compiler [IEF+19].

B. Pressure Field Models

In this section we give a brief overview of the pressure field models for soft contact forces as introduced in [EDSR19]. Technical and algorithmic details about our own implementation are given in Section III.

The primary goal of pressure field models is to replace point contact forces between two objects with forces applied over the entire surface at which two objects intersect. This change produces stabler results and also is a more accurate representation of real-world physics. At the same time, because pressure field models do not explicitly deform the objects themselves, they can remain efficient without having to resort to expensive finite-element methods (FEM).

At a high level, the pressure field model works by assigning each object a *pressure function* $p(\cdot)$. The pressure function assigns to each point in the interior of the object a nonnegative real number representing the pressure at that point, which is an intuitive notion of how much resistance a foreign body protruding into the object would experience at that point. When two objects with pressure functions $p_1(\cdot)$, $p_2(\cdot)$ intersect, there is a surface $\mathcal{S}$ inside the space of intersection at which the values of p_1 and p_2 are equal. After identifying this surface, we then define the total force exerted by one object on another as the integral

$$\iint_{\mathcal{S}} p_1(s)\mathbf{n}_s \, \mathrm{d}A,$$

where $\mathbf{n}_s$ is the normal vector to surface $\mathcal{S}$ at point s. Of course, in general it is difficult to accurately evaluate integrals over surfaces; we explain some simplifying assumptions as well as our implementation procedure in the next section.

III. Approach

In this section, we discuss the high-level overview of our mathematical and algorithmic approach to contact force simulation.

A. Assumptions

First we discuss the setup in our implementation of the pressure field model. We begin by making the following assumptions about our objects:

- Each object O is represented as a tetrahedral mesh, i.e. a collection of vertices $V \subseteq \mathbb{R}^3$ and a collection of non-intersecting tetrahedra T whose vertices are in V.
- In addition to the vertices of the object, whose coordinates are fixed at initialization, each object also has a pose, which we store using the 4×4 matrix representation of the rigid transform ${}^{O}X^{W}$.
- Each object has uniform mass density.
- Each object's rotational inertia is a scalar I, rather than the usual inertia tensor.
- Each vertex $v \in V$ of the object has a pressure $p(v)$ associated with it. We further assume that for any vertex v on the surface of the object, $p(v)$ is zero, while for any vertex v in the interior of the object, $p(v)$ is positive. Finally, for any point q in the object which is not one of the vertices, we can determine the pressure $p(q)$ at point q via linear interpolation using the vertices of the tetrahedron containing q.

The assumptions above allow us to construct objects O with a well-defined position, shape, mass distribution, and pressure field function $p(\cdot)$.

B. Computing the Pressure Field Explicitly

After defining the pressure $p(v)$ of each vertex $v \in V$ of an object O, we would like to compute the entire pressure function $p(\cdot)$ of the object. Recall in the previous section we wrote that we would assume pressure interpolated linearly within each tetrahedron. That is to say, given a tetrahedron t with vertices v_i, v_j, v_k, v_l, for any point $q \in t$ we should find constants a_i, a_j, a_k, a_l with

$$a_i v_i + a_j v_j + a_k v_k + a_l v_l = q, a_i + a_j + a_k + a_l = 1$$

and then define

$$p(q) = a_i p(v_i) + a_j p(v_j) + a_k p(v_k) + a_l p(v_l).$$

This observation allows us to explicitly compute the pressure field as a function of q. We begin by writing q as a 3×1 vector and augmenting it with a 1 to create the 4×1 vector $Q = \left[\begin{smallmatrix} q \\ 1 \end{smallmatrix}\right]$. Similarly, we'll write each v

as a 3×1 vector and augment them with ones to create the 4×4 matrix $V = \left[\begin{smallmatrix} v_i & v_j & v_k & v_l \\ 1 & 1 & 1 & 1 \end{smallmatrix}\right]$. Then it follows that

$$A = \begin{bmatrix} a_i \\ a_j \\ a_k \\ a_l \end{bmatrix}$$

satisfies $VA = Q$, and so we can solve $A = V^{-1}Q$. Now if $P = \left[\begin{smallmatrix} p(v_i) & p(v_j) & p(v_l) & p(v_k) \end{smallmatrix}\right]$ denotes the matrix of pressures, we know $p(q) = PA = PV^{-1}Q$, and this gives us the explicit expression for the pressure field within any tetrahedron.

C. Evaluating the Pressure Integral

Now, recall from our overview of the pressure field model that we compute the force between two objects with pressure functions $p_1(\cdot), p_2(\cdot)$ by identifying the surface $\mathcal{S}$ within the object intersection where $p_1(\cdot) = p_2(\cdot)$, and then evaluating the integral of $p_1(s)\mathbf{n}_s$ over the surface.

In general, this is a hard problem, but it is made significantly easier by the fact that the pressure functions p_1, p_2 are linear within each tetrahedron, so we can solve this problem for every pair of tetrahedra in the two objects and aggregate the results at the end. For a single pair of tetrahedra, the surface $\mathcal{S}$ is contained in the null space of $p_1 - p_2$, which due to linearity must be a plane, and therefore it follows that $\mathbf{n}_s$ is some constant normal $\mathbf{n}$ so that

$$\iint_{\mathcal{S}} p_1(s)\mathbf{n}_s \, \mathrm{d}A = \left(\iint_{\mathcal{S}} p_1(s) \, \mathrm{d}A \right) \mathbf{n}.$$

Now, since p_1 is linear and $\mathcal{S}$ is planar, it follows that the average value of $p_1(s)$ over $\mathcal{S}$ is the same as the value of $p_1(\mathrm{com}_{\mathcal{S}})$, where $\mathrm{com}_{\mathcal{S}}$ is the center of mass of $\mathcal{S}$. Therefore we can simplify

$$\left(\iint_{\mathcal{S}} p_1(s) \, \mathrm{d}A \right) \mathbf{n} = |\mathcal{S}| p_1(\mathrm{com}_{\mathcal{S}})\mathbf{n},$$

where $|\mathcal{S}|$ denotes the area of $\mathcal{S}$.

Now, since $\mathcal{S}$ is the intersection of the plane given by $p_1 = p_2$ with the boundaries of the two tetrahedra we are intersecting, it follows $\mathcal{S}$ is the intersection of a collection of linear constraints, making $\mathcal{S}$ a convex polygon. (While conceptually simple, actually computing the vertices of this convex polygon is quite challenging, and the algorithm we use is explained in Section III-E.) Therefore we can triangulate $\mathcal{S}$ into nonoverlapping triangles $\triangle_1, \triangle_2, \ldots, \triangle_k$ that cover $\mathcal{S}$. Now it's easy to compute the area $|\triangle_i|$ of each triangle, allowing us to obtain the total area $|\mathcal{S}| = \sum |\triangle_i|$. Similarly, the center of mass of each triangle com_i is the average of the three vertices, which can also be computed easily, and now the center of mass of the entire surface $\mathcal{S}$ is

$$\mathrm{com}_{\mathcal{S}} = \frac{1}{|\mathcal{S}|} \sum_i |\triangle_i| \mathrm{com}_i,$$

which can also be computed easily. After these observations, computation of the overall pressure is straightforward. Figure 1 depicts a colored visualization of the intersection surface and final result.

D. Approximating the Torque

The previous section details how we compute the contacting force between a pair intersecting tetrahedra. However, this is not enough for physics simulation – we also need a method of computing the torque exerted by each tetrahedron on the other one. Formally, if two tetrahedra t_1, t_2 are contained in objects with centers of mass $\mathrm{com}_1, \mathrm{com}_2$ and the equipressure surface between the tetrahedra is $\mathcal{S}$, then the torque τ_{12} exerted by t_2 on t_1, taken with respect to com_1, is

$$\tau_{12} = \iint_{\mathcal{S}} \mathbf{r} \times \mathbf{F} \, \mathrm{d}A = \iint_{\mathcal{S}} \mathbf{r} \times p_1(s)\mathbf{n} \, \mathrm{d}A,$$

where $\mathbf{r}$ is the moment arm from com_1 to s.

This integral is much harder to compute than the previous ones because the presence of $\mathbf{r}$ makes the integrand nonlinear (in fact, it's not a polynomial), so we choose to approximate $\mathbf{r}$ with the constant vector $\mathbf{r}_0 = \mathrm{com}_{\mathcal{S}} - \mathrm{com}_1$. Thus we have

$$\begin{aligned} \iint_{\mathcal{S}} \mathbf{r} \times p_1(s)\mathbf{n} \, \mathrm{d}A &\approx \mathbf{r}_0 \times \iint_{\mathcal{S}} p_1(s)\mathbf{n} \, \mathrm{d}A \\ &= (\mathrm{com}_{\mathcal{S}} - \mathrm{com}_1) \times \mathbf{F}_{21}, \end{aligned}$$

where $\mathbf{F}_{21}$ is the force exerted by object 2 on object 1. We already computed $\mathbf{F}_{21}$ and $\mathrm{com}_{\mathcal{S}}$ in the previous section, and the center of mass com_1 of object 1 is similarly straightforward to compute, so this gives a reasonable approximation of the torques involved.

This concludes our discussion of the model and algorithm basics. In the next sections we discuss the polygon intersection problem in greater detail and further performance optimizations.

E. Polygon Intersections

Recall in Section III-C we mentioned that the most complicated step in the force computation is determining the equipressure surface of two intersecting tetrahedra. More precisely, given two tetrahedra t_1, t_2 with pressure functions $p_1(\cdot), p_2(\cdot)$, we must compute the convex polygon formed by intersecting the plane $p_1 - p_2 = 0$ with the two regions t_1, t_2. We tried two approaches to this problem.

The first approach was to treat each tetrahedron as the intersection of four half-spaces defined by the faces of the tetrahedron. The intersection of the two tetrahedra

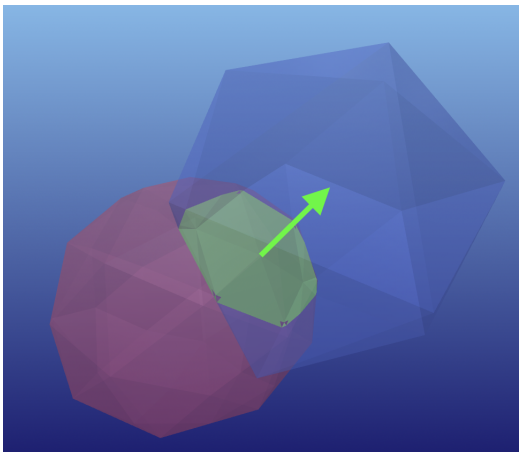

Figure 1. A diagram showing two objects (red and blue) intersecting. We compute the intersection surface (shown in green), and the force between the two objects is a vector (the green arrow) normal to the intersection surface with magnitude equal to the pressure integral over the surface.

thus becomes the intersection of eight half-spaces, and the plane $p_1 - p_2 = 0$ provides an additional linear equality constraint. We then took these nine constraints and passed them into the Polyhedra.jl library, which excels at computations related to polyhedral constraints.

The second approach was to compute the intersection polygon ourselves. We did this by first projecting each half-space constraint down one dimension to create a half-plane constraint, thereby reducing the problem to one about intersecting eight half-planes. We then implemented an algorithm from [Bur21] which computes the intersection of N half-planes in $O(N \log N)$ time. At a high level, the algorithm sorts the N half-planes by the angle each one forms with the positive x-axis. After the sort, we then process the half-planes one at a time to determine if each half-plane presents a constraint that is redundant given the constraints of the previous half-planes. Doing this allows us to find a minimal set of non-redundant half-planes, thereby giving us the boundary of our convex polygon, and then inverting the initial projection allows us to recover the intersection surface.

The first approach was very reliable because the `Polyhedra.jl` library performs computations in a robust manner, but it was also slower. By contrast, our implementation of the half-plane intersection algorithm was around two times faster, but it struggled with numerical instability resulting from coinciding and parallel lines.

IV. Performance Optimizations

In order to produce high-fidelity simulations, efficient algorithms are a must. This consideration also motivated our choice of Julia as our programming language, as Julia with optimizations can be as fast as a modern, high-performance C compiler.

The main performance bottleneck in the pressure field physics model is that, if two objects O_1 and O_2 are comprised of M and N tetrahedra respectively, then we must intersect every pair of tetrahedra to compute the intersection surface and force between the two shapes, which thereby results in performing an expensive operation MN times. As a result, we can achieve significant performance improvements by using geometric reasoning to short-circuit the computation, i.e. skipping unnecessary computations for pairs of tetrahedra that clearly do not intersect.

A. Bounding Box Checking

The simplest check we can perform is to associate each tetrahedron with an axis-aligned bounding box. For any tetrahedron, we can let m_x and M_x be the minimum and maximum x-coordinates of its vertices, and similarly define m_y, M_y, m_z, M_z to create a bounding box $[m_x, M_x] \times [m_y, M_y] \times [m_z, M_z]$ that contains the tetrahedron.

Then, before trying to compute the contact surface between a pair of tetrahedra, we can first check if the bounding boxes of the tetrahedra intersect, which is easy because they are axis-aligned. If the tetrahedra do not intersect, we can return that the forces and torques they exert on each other are all equal to zero, and abort the remainder of the computation. This simple heuristic alone is able to achieve a 3-4$\times$ speedup, with no further optimizations, because halfplane intersections is very costly and being able to short-circuit the evaluation there is quite useful.

B. Basic Intersection Checking

We can further reduce the number of force computations necessary with the following observation: given two tetrahedra, if either of them does not intersect their equipressure plane, then we do not need to intersect the tetrahedra. This check can easily be performed by finding the (signed) distance from each vertex of the tetrahedra to the plane, and short-circuiting if all values are positive or all values are negative.

C. Spatial Indexing Data Structures

In addition to the previous ideas, we also implemented a more advanced version of the bounding box intersection idea using R*-trees [BKSS90]. At a high level, an R*-tree organizes data into an indexed tree such that each node of the tree corresponds to some axis-aligned bounding box in n-dimensional space. Each vertex's bounding box contains the bounding boxes of its child vertices, creating a hierarchical ordering of spatial information and objects.

The key to the efficiency of R*-trees is that they dynamically rebalance data between tree branches in a way that minimizes overlap between the bounding boxes of sibling vertices. This allows for efficient spatial querying: if we have two objects A and B with tetrahedra collections T_1, T_2, we can store all the relevant spatial information for the tetrahedra in T_1 within an R*-tree. Then, for each tetrahedron $t \in T_2$, we query it against the R*-tree, starting at the root node and checking whether each branch of the tree contains tetrahedra that might intersect with the bounding box of t. The overlap-minimizing property of R*-trees ensures that relatively few branches will need to be inspected, thereby saving us from having to check for potential intersections between a large number of tetrahedron pairs.

As an additional insight, we realized that although the poses of A and B may change throughout the course of a simulation, the R*-trees of A and B do not need to. Instead, we can multiply the vertices of B by the rigid transformation ${}^{A}X^{B}$ to determine their positions in the frame of A, thereby allowing us to reuse the same R*-tree for A even as A changes over time. This saves us the computational burden of having to reconstruct R*-trees at every step in a simulation; we instead construct an R*-tree for each object once, at initialization.

R*-trees are a fairly complex data structure and can be difficult to implement from scratch, especially in a new programming language. Instead, to make the best use of our time, we incorporated an existing open-source Julia implementation from the SpatialIndexing.jl package.

D. Random Projections

Inspired by the success of spatial indexing data structures and bounding boxes, we take these ideas one step further. Namely, the following result is true: Two tetrahedra A and B do not intersect if and only if there exists $\mathbf{v}$ so that

$$\min_{x \in A}(\mathbf{x} \cdot \mathbf{v}) > \max_{x \in B}(\mathbf{x} \cdot \mathbf{v}).$$

This is known as the *separating hyperplane theorem*. Since tetrahedra are convex, one can evaluate the maximum of $\mathbf{x} \cdot \mathbf{v}$ over $x \in A$ simply by finding the maximum over A's vertices.

This suggests a quick algorithm: choose many random $\mathbf{v}$, and if $\min_{x \in A}(\mathbf{x} \cdot \mathbf{v}) > \max_{x \in B}(\mathbf{x} \cdot \mathbf{v})$ (or $\min_{x \in B}(\mathbf{x} \cdot \mathbf{v}) > \max_{x \in A}(\mathbf{x} \cdot \mathbf{v})$) we can short-circuit the tetrahedron intersection.

V. Simulation and Visualization

The previous sections allow for the computation of forces and torques between every pair of objects in a scene. This information allows us to then create physics simulations. Our simulation setup involves the following elements:

- We start with a *world*. This consists of a collection of objects, represented with triangular meshes as discussed earlier, with their initial poses as well as initial translational and angular velocities.
- We also pass functions $f_i(a, \alpha)$ into our simulation, where a is the linear acceleration and α is the angular acceleration, which take the accelerations computed from contact forces and calculate the true accelerations on the object, often by accounting for external forces. For instance, if we want to simulate the earth's gravity, we would pass in a function $f_i(a, \alpha)$ which adds $[0, 0, -9.8]$ to the translational acceleration a and keeps the angular acceleration α fixed, for each object O_i in the world. Since the pressure field model is intrinsically limited to computing forces resulting from contact, mechanisms like these are necessary for simulating a non-contact force such as gravity.

Now, given a world with a collection of objects $O = \{O_1, O_2, \ldots, O_k\}$ and functions f_i describing the actions of external forces on each object i (such as gravity), we can then move forward through a single timestep of the simulation via the following procedure for each object O_i:

1) We compute the aggregate force and torque F_i, τ_i on object O_i resulting from contacts with each of the other objects O_j. We then compute
$$a_i = \frac{F_i}{m_i}, \ \alpha_i = I_i^{-1}\tau_i$$
using the usual laws of motion.
2) We update $(a_i, \alpha_i) = f_i(a_i, \alpha_i)$, a user-provided function, to take into account the change in each object's acceleration due to external forces.
3) We update the translational velocity v_i and angular velocity ω_i via the usual updates
$$v_i = v_i + a_i \cdot dt, \ \omega_i = \omega_i + \alpha_i \cdot dt.$$
Note that this update intentionally occurs before the pose update for stability reasons. This is a variant of the symplectic Euler method [DR05].
4) The object's translational pose is incremented by $v_i \cdot dt$.
5) The object's rotational pose is more complicated to update. Following the work in [Wil], we define the skew matrix $S = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix}$ and angle $\theta = \|\omega\|$, and then we update the rotational matrix of O_i via left multiplication by the exponential map $\exp(S) = I + \frac{\sin\theta}{\theta}S + \frac{1-\cos\theta}{\theta^2}S^2$.

Interactive visualization in Pluto.jl notebooks was implemented using the MeshCat.jl library, which produces fast interactive 3-D renders through rasterization. All of our objects in our simulations were cubes or

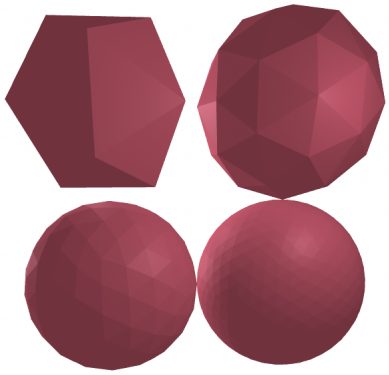

Figure 2. A comparison of our approximate sphere objects, at varying levels of subdivision. The third-order subdivision on the bottom-right is essentially indistinguishable from a real sphere.

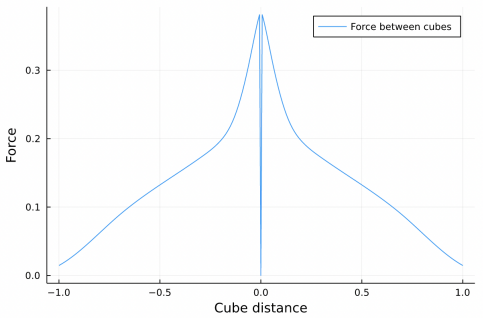

Figure 3. Magnitude of translational force between two cubes centered at $(0, 0, 0)$ and $(0, x, 0)$, as x varies.

subdivided icospheres, although our simulator supports general tetrahedral meshes. To represent cubes, we use the eight corners along with an additional ninth vertex at the center of the cube, then divide the cube into twelve tetrahedra that each contain its center, such that each of the six faces of the cube is divided among two tetrahedra. Similarly, to create spherical approximations using triangle meshes, we begin with an icosahedron and apply a subdivision surface algorithm that produces a uniform mesh in every direction. Figure 2 shows renderings of the spheres we use in simulation.

VI. Evaluation and Discussion

A. Test Cases

We tested our work through unit tests on each major code component, including object generation, tetrahedron intersections, and force computation. Most of these unit tests consisted of taking simple objects comprised of only one or two tetrahedra, working out the expected behavior of our programs by hand, and checking if the results matched. In addition to unit tests, we used larger-scale experiments to evaluate the quality of our force predictions.

Figure 3 shows the results of an experiment where we take two unit cubes, centered at $(0, 0, 0)$ and $(0, x, 0)$ for $-1 \leq x \leq 1$, and consider the magnitude of the net force exerted by each cube on the other as x varies. The graph exhibits many expected behaviors, such as symmetry between x and $-x$, as well as the force having maximal magnitude at $x \approx 0$ but zero magnitude at $x = 0$ (due to the cubes overlapping perfectly).

Finally, we ran our physics simulator on a variety of simple scenarios to check if the results looked reasonable and realistic. We could not come up with a rigorous testing framework for simulation because it is difficult to determine what the "correct" answer should be, so we primarily resorted to eye tests. For instance, this video is our simulation of a ball bouncing on a flat surface, while this video is our simulation of a ball bouncing off an incline.

B. Performance Benchmarks

Next, we evaluated the performance of our code, with and without the optimizations we discussed in Section IV. We took two k-th order icosahedral subdivisions of spheres for $1 \leq k \leq 3$ and measured the runtime of the force computation algorithm while enabling or disabling various optimizations that we implemented. The results are shown in Table I. It is evident from the results that the optimizations we discussed result in substantial performance gains, and that these gains are more measured as the number of tetrahedra in the objects increases.

C. Anomalies

Despite our best efforts, there was some anomalous behavior that appeared at various different points in our implementation of force computation and physics simulation. For example, as previously mentioned in Section III-E, our implementation of polygon intersection failed in scenarios where two lines were parallel and very close to each other. Figure 4 shows one of these examples.

In general, most of the pathological behavior we encountered was due to problems in handling numerical instability and floating-point errors; these issues usually arose when we tried to simulate contact between objects whose surfaces contained planes parallel to each other. As mentioned in Section III-E, these problems mostly vanished when we fed the tetrahedron intersection problem into `Polyhedra.jl`.

D. Differentiability

Although we initially wanted to utilize Julia's powerful AD libraries to create a differentiable physics simulator, we ultimately were not able to do so. As

Case	Optimizations: Bounding Boxes	Intersection Checks	Spatial Indices	Random Projections	Icosphere Subdivisions: 1	2	3
(a)	-	-	-	-	0.263	3.77	58.16
(b)	✓	-	-	-	0.077 (**3.41x**)	0.957 (**3.94x**)	14.52 (**4.01x**)
(c)	✓	✓	-	-	0.071 (**3.70x**)	0.943 (**4.00x**)	13.71 (**4.24x**)
(d)	✓	✓	✓	-	0.041 (**6.41x**)	0.445 (**8.47x**)	5.30 (**10.97x**)
(e)	✓	✓	✓	✓	0.036 (**7.31x**)	0.368 (**10.24x**)	3.82 (**15.23x**)

Table I

THIS TABLE SHOWS THE EFFECT OF OUR VARIOUS SPEEDUPS ON FORCE COMPUTATION RUN-TIME, MEASURED IN SECONDS. NOTE THAT THE MULTIPLICATIVE IMPROVEMENT INCREASES AS THE NUMBER OF TETRAHEDRA INCREASES.

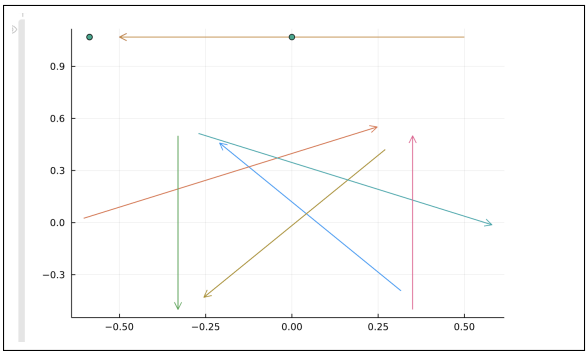

Figure 4. In this scenario, the colored lines define half-planes which don't share a common intersection. However, the brown horizontal line at the top is actually two nearly parallel lines in close proximity to each other; the algorithm then fails and detects a triangular region of intersection defined by the green dots (where the left green dot is actually two green dots overlayed on each other due to the proximity of the parallel lines).

mentioned in Section II-A, we implemented our force computation and physics simulation with the goal of incorporating `Zygote.jl` into our code. However, after finishing our implementations, we discovered that we were unable to differentiate through our force computation. The coverage for AD was not as wide as we had hoped, and at some point we applied operations for which AD was not supported, so we were unable to make the simulations differentiable.

VII. Conclusion

In this project, we successfully implemented a minimal physics simulator using the pressure field model. We developed and implemented efficient algorithms for contact force and torque computation using a combination of mathematical derivations and geometric optimizations. We then applied these algorithms to create realistic simulations of physical systems. Although we were ultimately unable to incorporate automatic differentiation into our physics simulator, our simulations can still be used to efficiently train robots using black-box optimization methods. Future work could involve replacing the non-differentiable elements of our code with differentiable ones, thereby creating fully differentiable physics simulators that support end-to-end gradient learning for robotics tasks.

VIII. Acknowledgements

We would like to thank Professor Russ Tedrake for the lectures on physics simulation and kinematics modeling that inspired this project, as well as for introducing us to hydroelastic contact forces and the pressure field model.

The project contributions were as follows: all authors worked on all components of the code and designed them together, reviewing each other's work and collaborating through version control. However, in terms of focus, most of the physics simulation, Julia data structures, and interactive rendering components were written by Eric, while the force computation and geometric algorithms were primarily developed by Franklyn and Vincent. All authors contributed equally to background research, testing, benchmarks, and the final report.

References

[ABC+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[BEKS17] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 322–331, 1990.

[Bur21] Oscar Burga. Half-plane intersection. *CP-Algorithms*, 2021.

[DR05] Denis Donnelly and Edwin Rogers. Symplectic integrators: An introduction. *American Journal of Physics*, 73(10):938–945, 2005.

[EDSR19] Ryan Elandt, Evan Drumwright, Michael Sherman, and Andy Ruina. A pressure field model for fast, robust approximation of net contact force and moment between nominally rigid objects. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 8238–8245. IEEE, 2019.

[IEF+19] Mike Innes, Alan Edelman, Keno Fischer, Chris Rackauckas, Elliot Saba, Viral B Shah, and Will Tebbutt. A differentiable programming system to bridge machine learning and scientific computing. *arXiv preprint arXiv:1907.07587*, 2019.

[PGC+17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NeurIPS 2017 AutoDiff Workshop*, 2017.

[Wil] Adam Williams. Computing the exponential map on SO(3). https://arwilliams.github.io/so3-exp.pdf.