JAX, M.D.

End-to-End Differentiable, Hardware Accelerated, Molecular Dynamics in Pure Python

Samuel S. Schoenholz
Google Brain
schsam@google.com

Ekin D. Cubuk
Google Brain
cubuk@google.com

Abstract

A large fraction of computational science involves simulating the dynamics of particles that interact via pairwise or many-body interactions. These simulations, called Molecular Dynamics (MD), span a vast range of subjects from physics and materials science to biochemistry and drug discovery. Most MD software involves significant use of handwritten derivatives and code reuse across C++, FORTRAN, and CUDA. This is reminiscent of the state of machine learning before automatic differentiation became popular. In this work we bring the substantial advances in software that have taken place in machine learning to MD with JAX, M.D. (JAX MD). JAX MD is an end-to-end differentiable MD package written entirely in Python that can be just-in-time compiled to CPU, GPU, or TPU. JAX MD allows researchers to iterate extremely quickly and lets researchers easily incorporate machine learning models into their workflows. Finally, since all of the simulation code is written in Python, researchers can have unprecedented flexibility in setting up experiments without having to edit any low-level C++ or CUDA code. In addition to making existing workloads easier, JAX MD allows researchers to take derivatives through whole-simulations as well as seamlessly incorporate neural networks into simulations. This paper explores the architecture of JAX MD and its capabilities through several vignettes. Code is available at

www.github.com/google/jax-md

along with an interactive Colab notebook that goes through all of the experiments discussed in the paper.

1 Introduction

Understanding complex many-body systems is a challenge that underlies many of the hard problems in the physical sciences. A ubiquitous tool at our disposal in trying to understand such systems is to posit interactions between the constituents and then simulate the resulting dynamics. If interactions can be identified such that the simulation captures macroscopic behaviors observed in experiments, then the simulation can be studied to gain insight into the physical system. Since one has access to the full microscopic state at each step, it is possible to test hypotheses and make measurements that would otherwise be impossible. Such techniques, generally called molecular dynamics (MD), have been used to understand a wide range of systems including molecules, crystals, glasses, proteins, polymers, and whole biological cells.

Significant effort has gone into a number of high quality MD packages such as LAMMPS (Plimpton, 1995), HOOMD-Blue (Anderson et al., 2008; Glaser et al., 2015), and OpenMM (Eastman et al., 2017). Traditional simulation environments are large and specialized codebases written in C++

Second Workshop on Machine Learning and the Physical Sciences (NeurIPS 2019), Vancouver, Canada.

or FORTRAN, along with custom CUDA kernels for GPU acceleration. These packages include significant amounts of code duplication and hand written gradients. The state of affairs is reminiscent of Machine Learning (ML) before the popularization of Automatic Differentiation (AD). Researchers trying a new idea often have to spend significant effort computing derivatives and integrating them into these large and specialized codebases. Simultaneously, the amount of data produced from MD simulations has been rapidly increasing, in part due to ever increasing computational resources along with more efficient MD software. Furthermore, deep learning is becoming a popular tool both for making MD simulations more accurate and for analyzing data produced in the simulations. Unfortunately, the issues facing MD libraries are even more pronounced when combining MD with deep learning, which typically involves complicated derivatives that can take weeks to derive and implement.

Here we introduce JAX, M.D. (JAX MD) which is a new MD package that leverages the substantial progress made in ML software to improve this state of affairs. JAX MD is end-to-end differentiable, written in pure python, and is fast since simulations are just-in-time compiled to CPU, GPU, or TPU using XLA. Moreover, JAX MD is based on JAX (Bradbury et al., 2018; Frostig et al., 2018) which has a strong neural network ecosystem that can be used seamlessly with simulations. In addition to a strong neural network ecosystem and just-in-time compilation, JAX can automatically vectorize calculations over one- or multiple-devices. This makes it easy to simulate ensembles of systems in JAX MD. We explore the features of JAX MD through several experiments:

- Efficient generation of ensembles of systems.
- Using neural networks to do machine learning of a potential.
- Meta-optimization through a simulation to optimize physical parameters.

While these examples are designed to be illustrative, they are similar to problems faced in actual research. Moreover, all of these would be significantly more difficult using existing tools. Additionally, we discuss implementation details and related work in the Appendix.

JAX MD has so far implemented simple pairwise potentials (Lennard-Jones, soft-sphere, Morse) and the embedded atom method (EAM) (Daw & Baskes, 1984). It can also work with the Atomic Simulation Environment (Larsen et al., 2017) and other first-principles calculations that can be accessed from Python (e.g. Quantum Espresso (Giannozzi et al., 2009)). Due to its efficient spatial partitioning strategy, it can simulate millions of particles on a single GPU. On the ML side, JAX MD has access to all of the ML developments in JAX, including state-of-the-art convolutional networks and graph networks.

2 Three Vignettes

2.1 Vectorized Generation of Ensembles

Increases in computing power are increasingly due to device parallelism rather compute speed. Indeed GPUs are designed to process significant amounts of data in parallel and TPUs move futher in this direction by offering high speed interconnects between chips. This parallelism is often used to simulate ever larger systems. However, there are other interesting uses of parallelism that have received less attention. Many of these methods (e.g. replica exchange MCMC sampling (Swendsen & Wang, 1986) or nudged elastic band (Henkelman et al., 2000)) involve simulating an ensemble of states simultaneously.

Thanks to JAX, ensembling can be done automatically in JAX MD. For small systems, the amount of necessary compute can be sub-linear in the number of replicas since it can otherwise be difficult to saturate the parallelism of accelerators. Here we go through an example where we use automatic ensembling to quickly compute statistics of a simulation. Suppose we have a function `simulate(key)` that simulates a single system given a random key and returns its final positions using code similar to Section B. JAX includes the function `vmap` that automatically vectorizes computations. Here to run an ensemble of simulations we simply define,

```
vectorized_simulation = vmap(simulation)
```

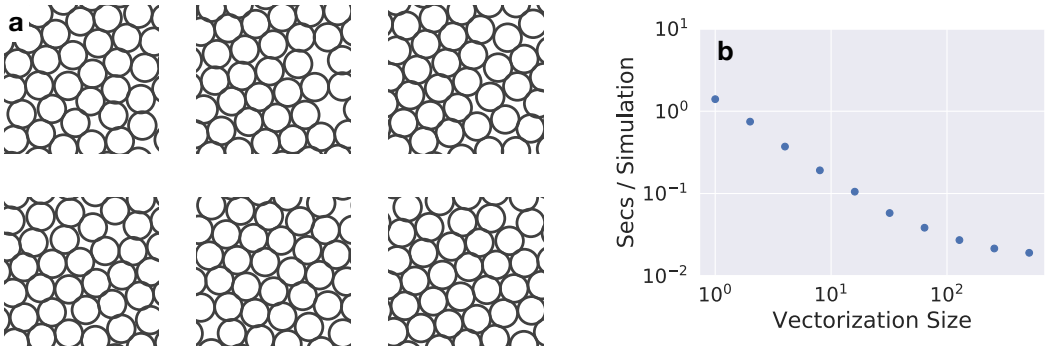

Figure 1: On the left are 6 of the configurations produced by the vectorized simulation function. On the right is the time-per-simulation using the `vmap` functionality of JAX.

Fig. 1 (a) shows some example simulations of small, 32-particle systems that were performed in parallel on a single GPU. These simulations are too small to saturate the compute on a single GPU and Fig 1 (b) shows that the time-per-simulation decreases with the number of simulations being performed in parallel. This scaling continues until a batch size of about 100 when the GPU compute becomes saturated.

2.2 Easy Machine Learned Potentials

Historically energy functions were often derived by hand based on coarse heuristics and scarce experimental results were used to fit parameters. More recently, energy functions with a larger number of fitting parameters (e.g. ReaxFF (Van Duin et al., 2001)) have become popular due to their ability to accurately describe certain systems. However, these methods traditionally involve significant expert knowledge and fail for systems that deviate too much from those that they were designed for. A natural progression of this trend is to use neural networks and large datasets to learn energy functions. There were a number of early efforts that received mixed success; however, it was not until 2007 when Behler and Parrinello (Behler & Parrinello, 2007) published their general purpose neural network architecture that learned energy functions emerged as a viable alternative to traditional approaches.

Since then large amounts of work has been done on this topic, however the substantial progress in machine learned potentials has not seen as much use as might be expected. At the root of this discrepancy are two points of friction at the intersection of ML and MD that prevent rapid prototyping and deployment of learned energies. First, simulation code and machine learning code are written in different languages. Second, due to the lack of automatic differentiation in molecular dynamics packages, including neural network potentials in physics simulations can require substantial work which often prohibits easy experimentation (see Eq. 1 below).

To address these issues, several projects developed adapters (Artrith & Urban, 2016; Artrith et al., 2017; Lot et al., 2019; Onat et al., 2018) between common ML languages, like Torch and Tensorflow, and common MD languages like LAMMPS. However, these solutions require researchers to be working in exactly the regime serviced by the adapter. One of the consequences of this is that the atomistic features which get fed into the neural network need to be differentiated by hand within the MD package to compute forces. Trying out a new set of features can easily take weeks or months of work to compute and implement these derivatives.

As an example, we will fit a neural network to the bubble potential defined in Eq. (2) and see how JAX MD gets around these issues easily. The Behler-Parrinello architecture describes the total energy of the system as a sum over individual contributions from per-atom neural networks, $U(\{\vec{r}_i\}) = \sum_j E(\theta; \boldsymbol{G}_j(\{\vec{r}_i\}))$ where $f(\theta; \boldsymbol{x})$ is a fully-connected neural network with parameters θ and $\boldsymbol{G}_j(\{\vec{r}_i\})$ are hand-designed, many-body, features for a particle j. While many choices of features exist, one simple set are given by the local pair correlation function, $g_i(\rho) = \sum_j \delta(r_{ij} - \rho)$,

which measures the the density of particles a distance ρ from a central particle. The Behler-Parrinello architecture can be described and initialized in two lines of python.

```
init_fun, E = stax.serial(
    stax.Dense(no_hidden_units), stax.Relu, # hidden layer 1
    stax.Dense(no_hidden_units), stax.Relu, # hidden layer 2
    stax.Dense(1)) # readout
_, params = init_fun(key, (-1, number_of_features))
```

`stax` is JAX's native neural network library. It is also easy to define the Behler-Parrinello loss using `vmap` and the JAX MD function `pair_corr_fun = quantity.pair_correlation(displacement)` as shown below.

```
g = quantity.pair_correlation(displacement)
U = lambda params, positions: np.sum(E(params, g(positions))) # Eq. 4.

def per_example_loss(params, positions):
  return (U(params, positions) - energy_fun(positions))**2

def loss(params, batch_positions):
  return np.mean(vmap(per_example_loss, in_axis=(None, 0))(params, batch_positions))
```

`per_example_loss` defines the MSE loss on a single state (atomic configuration) and `loss` is the total loss over a minibatch of states. We see a comparison between the learned energies and ground truth energies after training the above architecture for 20 seconds on 800 example states in Fig 2.2 (a).

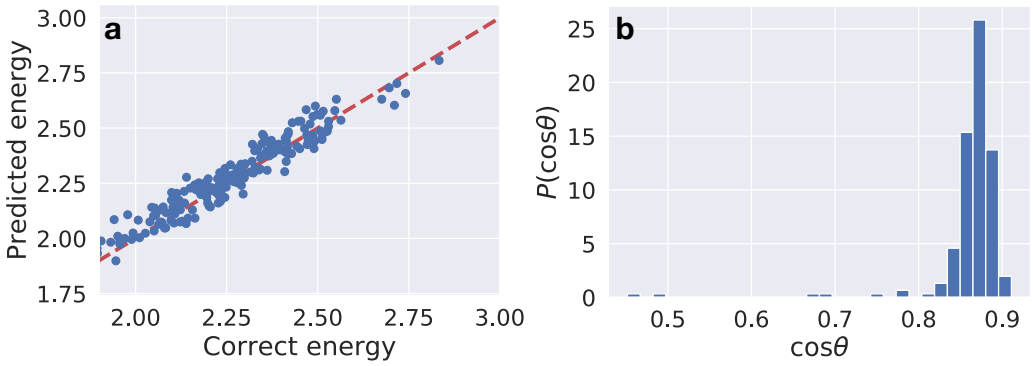

Figure 2: Left panel shows the agreement between predicted energies and the correct energies of the bubble rafts in the test set. Right panel shows the distribution of the inner product between the correct force and the predicted force.

We now compute forces with JAX MD and consider how this would be implemented in a standard MD package. Recall that $F_i = -\partial_{\vec{r}_i} U(\{\vec{r}_i\})$, where U is the potential energy of the system defined in eq (2.2). Thus,

$$F_i = -\sum_j \frac{dE(\theta; \boldsymbol{G}_j(\{\vec{r}_i\}))}{dG_j(\{\vec{r}_i\})} \frac{dG_j(\{\vec{r}_i\})}{d\vec{r}_i}. \tag{1}$$

using the chain rule. Since dE/dG_j is the gradient of a neural network it is easy get in most neural network packages and feed into MD. However, traditionally dG_j/dr_i is a pain point and has to be coded up by hand. In JAX, MD, we get dG_j/dr_i for free without any extra work using JAX's `grad` function as `grad(lambda params, r: -E(params, g(r)), argnums=1)`

This energy function and force can now be used in any JAX MD simulation. In Fig. 2 (b) we see a comparison between the predictions of this network after 20 seconds of training on states generated in JAX MD. Despite the small amounts of compute involved, we see reasonable agreement for energies and forces between the machine-learned potential and the real potential.

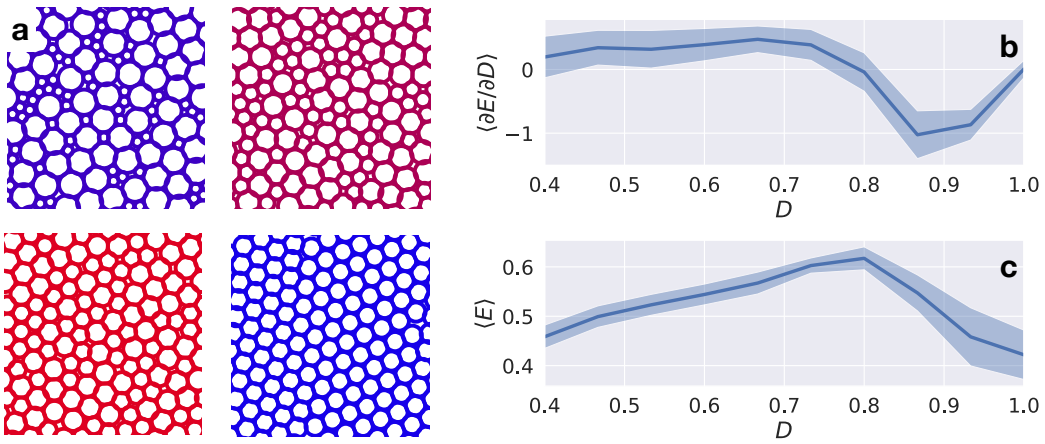

Figure 3: Panel a shows the average energy and the standard deviation of the energy at D. Panel b shows the derivative we calculate by differentiating through energy optimization by gradient descent as a function of D.

2.3 Optimization Through Dynamics

So far we have demonstrated how JAX MD can make common workloads easier. However, combining molecular dynamics with automatic differentiation opens the door for qualitatively new research. One such avenue involves differentiating through the simulation trajectory to optimize physical parameters. There have been several excellent applications so far in e.g. protein folding (AlQuraishi, 2019; Ingraham et al., 2018), but until now this has involved significant amounts of specialized code. This vein of research is also similar to recent work in machine learning on meta-optimization (Andrychowicz et al., 2016; Metz et al., 2018).

We revisit the bubble raft example above. In this case, we will control the structure of the bubble raft by differentiating through the simulation. As we saw in Section B, bubble rafts form a hexagonal structure when all of the bubbles have the same size. However, when the bubbles have different sizes the situation can change considerably. To experiment with these changes, we're going to set up a simulation of a bubble raft with bubbles of two distinct sizes. To keep things simple, we'll let half of the bubbles have diameter 1 and half have diameter D. To control the conditions of the experiment, we will keep the total volume of the bubbles constant (see appendix E. Unlike the previous simulations, we will minimize the energy of the system using a function `simulate(diameter, key)` that returns the energy of a system given a diameter and a random key. Using `vmap` we can vectorize the simulation to compute the statistics for an ensemble of states at different diameters in parallel on the GPU.

Some example states at different diameters along with the energy as a function of diameter can be found in Fig 3. We see that the hexagonal structure breaks down in the two-species case. Moreover, we see that there is a "most disordered" point when $D = 0.8$, which can be seen as the highest energy point in Fig. 3(a). The study of such disordered systems is often referred to as the study of "Jammed (O'hern et al., 2003)" solids. However, this was a somewhat brute-force way to investigate the role of size-disparity in the structure of bubble rafts. Could we have seen the same result more directly? Since each energy calculation is a result of a differentiable simulation, we can differentiate through the minimization with respect to D. This would allow us to find extrema of the minimized-energy as a function of diameter using first-order optimization methods. This could be implemented in JAX MD as, `dE_dD_fun = grad(simulate)`. Of course the `dE_dD_fun` function can be vectorized to aggregate statistics from an ensemble.

The gradient is plotted in Fig. 3 (b). We see that the gradient is positive and constant for $D < 0.8$ corresponding to the linear increase in the average energy. Moreover, we see that the derivative crosses zero exactly at the maximum average energy. Finally, we observe that the gradient goes back to zero at $D = 1$. This suggests that $D = 0.8$ is the point of maximum disorder, as we found by brute force above. It also shows that $D = 1$ is the minimum energy configuration of the diameter. Although we hadn't hypothesized it, we realize this must be true since $D < 1$ states are symmetric with $D > 1$ as we keep the total packing fraction constant.

References

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)*, pp. 265–283, 2016.

Mohammed AlQuraishi. End-to-end differentiable learning of protein structure. *Cell systems*, 8(4): 292–301, 2019.

Joshua A Anderson, Chris D Lorenz, and Alex Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of computational physics*, 227(10):5342–5359, 2008.

Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in neural information processing systems*, pp. 3981–3989, 2016.

Nongnuch Artrith and Jörg Behler. High-dimensional neural network potentials for metal surfaces: A prototype study for copper. *Physical Review B*, 85(4):045439, 2012.

Nongnuch Artrith and Alexander Urban. An implementation of artificial neural-network potentials for atomistic materials simulations: Performance for tio2. *Computational Materials Science*, 114: 135–150, 2016.

Nongnuch Artrith, Tobias Morawietz, and Jörg Behler. High-dimensional neural-network potentials for multicomponent systems: Applications to zinc oxide. *Physical Review B*, 83(15):153101, 2011.

Nongnuch Artrith, Alexander Urban, and Gerbrand Ceder. Efficient and accurate machine-learning interpolation of atomic energies in compositions with many species. *Physical Review B*, 96(1): 014112, 2017.

Nongnuch Artrith, Alexander Urban, and Gerbrand Ceder. Constructing first-principles phase diagrams of amorphous li x si using machine-learning-assisted sampling with an evolutionary algorithm. *The Journal of chemical physics*, 148(24):241711, 2018.

Albert P Bartók, James Kermode, Noam Bernstein, and Gábor Csányi. Machine learning a general-purpose interatomic potential for silicon. *Physical Review X*, 8(4):041048, 2018.

Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian Goodfellow, Arnaud Bergeron, Nicolas Bouchard, David Warde-Farley, and Yoshua Bengio. Theano: new features and speed improvements. *arXiv preprint arXiv:1211.5590*, 2012.

Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of machine learning research*, 18 (153), 2018.

Jörg Behler. Atom-centered symmetry functions for constructing high-dimensional neural network potentials. *The Journal of chemical physics*, 134(7):074106, 2011.

Jörg Behler and Michele Parrinello. Generalized neural-network representation of high-dimensional potential-energy surfaces. *Physical review letters*, 98(14):146401, 2007.

Christian H Bischof, H Martin Bücker, Arno Rasch, Emil Slusanschi, and Bruno Lang. Automatic differentiation of the general-purpose computational fluid dynamics package fluent. *Journal of fluids engineering*, 129(5):652–658, 2007.

Erik Bitzek, Pekka Koskinen, Franz Gähler, Michael Moseler, and Peter Gumbsch. Structural relaxation made simple. *Physical review letters*, 97(17):170201, 2006.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Luca Capriotti. Fast greeks by algorithmic differentiation. *Available at SSRN 1619626*, 2010.

Richard Car and Mark Parrinello. Unified approach for molecular dynamics and density-functional theory. *Physical review letters*, 55(22):2471, 1985.

Gregory R Carmichael, Adrian Sandu, et al. Sensitivity analysis for atmospheric chemistry models via automatic differentiation. *Atmospheric Environment*, 31(3):475–489, 1997.

Isabelle Charpentier and Mohammed Ghemires. Efficient adjoint derivatives: application to the meteorological model meso-nh. *Optimization Methods and Software*, 13(1):35–63, 2000.

Stewart J Clark, Matthew D Segall, Chris J Pickard, Phil J Hasnip, Matt IJ Probert, Keith Refson, and Mike C Payne. First principles methods using castep. *Zeitschrift für Kristallographie-Crystalline Materials*, 220(5/6):567–570, 2005.

Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. Torch: a modular machine learning software library. Technical report, Idiap, 2002.

Ekin D Cubuk, Samuel Stern Schoenholz, Jennifer M Rieser, Brad Dean Malone, Joerg Rottler, Douglas J Durian, Efthimios Kaxiras, and Andrea J Liu. Identifying structural flow defects in disordered solids using machine-learning methods. *Physical review letters*, 114(10):108001, 2015.

Ekin D Cubuk, Samuel S Schoenholz, Efthimios Kaxiras, and Andrea J Liu. Structural properties of defects in glassy liquids. *The Journal of Physical Chemistry B*, 120(26):6139–6146, 2016.

Ekin D Cubuk, Brad D Malone, Berk Onat, Amos Waterland, and Efthimios Kaxiras. Representations in neural network based empirical potentials. *The Journal of chemical physics*, 147(2):024104, 2017a.

Ekin Dogus Cubuk, RJS Ivancic, Samuel S Schoenholz, DJ Strickland, Anindita Basu, ZS Davidson, Julien Fontaine, Jyo Lyn Hor, Y-R Huang, Y Jiang, et al. Structure-property relationships from universal signatures of plasticity in disordered solids. *Science*, 358(6366):1033–1037, 2017b.

Murray S Daw and Michael I Baskes. Embedded-atom method: Derivation and application to impurities, surfaces, and other defects in metals. *Physical Review B*, 29(12):6443, 1984.

Filipe de Avila Belbute-Peres, Kevin Smith, Kelsey Allen, Josh Tenenbaum, and J Zico Kolter. End-to-end differentiable physics for learning and control. In *Advances in Neural Information Processing Systems*, pp. 7178–7189, 2018.

Volker L Deringer, Noam Bernstein, Albert P Bartók, Matthew J Cliffe, Rachel N Kerber, Lauren E Marbella, Clare P Grey, Stephen R Elliott, and Gábor Csányi. Realistic atomistic structure of amorphous silicon from machine-learning-driven molecular dynamics. *The journal of physical chemistry letters*, 9(11):2879–2885, 2018a.

Volker L Deringer, Miguel A Caro, Richard Jana, Anja Aarva, Stephen R Elliott, Tomi Laurila, Gábor Csányi, and Lars Pastewka. Computational surface chemistry of tetrahedral amorphous carbon by combining machine learning and density functional theory. *Chemistry of Materials*, 30(21): 7438–7445, 2018b.

Peter Eastman, Jason Swails, John D Chodera, Robert T McGibbon, Yutong Zhao, Kyle A Beauchamp, Lee-Ping Wang, Andrew C Simmonett, Matthew P Harrigan, Chaya D Stern, et al. Openmm 7: Rapid development of high performance algorithms for molecular dynamics. *PLoS computational biology*, 13(7):e1005659, 2017.

Albert Einstein. On the motion of small particles suspended in liquids at rest required by the molecular-kinetic theory of heat. *Annalen der physik*, 17:549–560, 1905.

J e Enkovaara, Carsten Rostgaard, J Jørgen Mortensen, Jingzhe Chen, M Dułak, Lara Ferrighi, Jeppe Gavnholt, Christian Glinsvad, V Haikola, HA Hansen, et al. Electronic structure calculations with gpaw: a real-space implementation of the projector augmented-wave method. *Journal of Physics: Condensed Matter*, 22(25):253202, 2010.

Felix A Faber, Luke Hutchison, Bing Huang, Justin Gilmer, Samuel S Schoenholz, George E Dahl, Oriol Vinyals, Steven Kearnes, Patrick F Riley, and O Anatole Von Lilienfeld. Prediction errors of molecular machine learning models lower than hybrid dft error. *Journal of chemical theory and computation*, 13(11):5255–5264, 2017.

Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing, 2018.

Paolo Giannozzi, Stefano Baroni, Nicola Bonini, Matteo Calandra, Roberto Car, Carlo Cavazzoni, Davide Ceresoli, Guido L Chiarotti, Matteo Cococcioni, Ismaila Dabo, et al. Quantum espresso: a modular and open-source software project for quantum simulations of materials. *Journal of physics: Condensed matter*, 21(39):395502, 2009.

Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 1263–1272. JMLR. org, 2017.

Jens Glaser, Trung Dac Nguyen, Joshua A Anderson, Pak Lui, Filippo Spiga, Jaime A Millan, David C Morse, and Sharon C Glotzer. Strong scaling of general-purpose molecular dynamics simulations on gpus. *Computer Physics Communications*, 192:97–107, 2015.

Jürgen Hafner. Ab-initio simulations of materials using vasp: Density-functional theory and beyond. *Journal of computational chemistry*, 29(13):2044–2078, 2008.

Paul Z Hanakata, Ekin D Cubuk, David K Campbell, and Harold S Park. Accelerated search and design of stretchable graphene kirigami using machine learning. *Physical review letters*, 121(25): 255304, 2018.

Graeme Henkelman, Blas P Uberuaga, and Hannes Jónsson. A climbing image nudged elastic band method for finding saddle points and minimum energy paths. *The Journal of chemical physics*, 113(22):9901–9904, 2000.

Stephan Hoyer, Jascha Sohl-Dickstein, and Sam Greydanus. Neural reparameterization improves structural optimization. *arXiv preprint arXiv:1909.04240*, 2019.

John Ingraham, Adam Riesselman, Chris Sander, and Debora Marks. Learning protein structure with a differentiable simulator. 2018.

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

Ask Hjorth Larsen, Jens Jørgen Mortensen, Jakob Blomqvist, Ivano E Castelli, Rune Christensen, Marcin Dułak, Jesper Friis, Michael N Groves, Bjørk Hammer, Cory Hargus, et al. The atomic simulation environment—a python library for working with atoms. *Journal of Physics: Condensed Matter*, 29(27):273002, 2017.

Ruggero Lot, Franco Pellegrini, Yusuf Shaidu, and Emine Kucukbenli. Panna: Properties from artificial neural network architectures. *arXiv preprint arXiv:1907.03055*, 2019.

Xiaoguang Ma, Zoey S Davidson, Tim Still, Robert JS Ivancic, SS Schoenholz, AJ Liu, and AG Yodh. Heterogeneous activation, local structure, and softness in supercooled colloidal liquids. *Physical review letters*, 122(2):028001, 2019.

Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, volume 238, 2015.

Glenn J Martyna, Michael L Klein, and Mark Tuckerman. Nosé–hoover chains: The canonical ensemble via continuous dynamics. *The Journal of chemical physics*, 97(4):2635–2643, 1992.

Luke Metz, Niru Maheswaranathan, Brian Cheung, and Jascha Sohl-Dickstein. Meta-learning update rules for unsupervised representation learning. *arXiv preprint arXiv:1804.00222*, 2018.

J-D Müller and P Cusdin. On the performance of discrete adjoint cfd codes using automatic differentiation. *International journal for numerical methods in fluids*, 47(8-9):939–945, 2005.

Berk Onat, Ekin D Cubuk, Brad D Malone, and Efthimios Kaxiras. Implanted neural network potentials: Application to li-si alloys. *Physical Review B*, 97(9):094106, 2018.

Corey S O'hern, Leonardo E Silbert, Andrea J Liu, and Sidney R Nagel. Jamming at zero temperature and zero applied stress: The epitome of disorder. *Physical Review E*, 68(1):011306, 2003.

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. 2017.

Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of computational physics*, 117(1):1–19, 1995.

Pankaj Rajak, Rajiv K Kalia, Aiichiro Nakano, and Priya Vashishta. Neural network analysis of dynamic fracture in a layered material. *MRS Advances*, 4(19):1109–1117, 2019a.

Pankaj Rajak, Aravind Krishnamoorthy, Aiichiro Nakano, Priya Vashishta, and Rajiv Kalia. Structural phase transitions in a mowse 2 monolayer: Molecular dynamics simulations and variational autoencoder analysis. *Physical Review B*, 100(1):014108, 2019b.

Samuel S Schoenholz. Combining machine learning and physics to understand glassy systems. In *Journal of Physics: Conference Series*, volume 1036, pp. 012021. IOP Publishing, 2018.

Samuel S Schoenholz, Ekin D Cubuk, Daniel M Sussman, Efthimios Kaxiras, and Andrea J Liu. A structural approach to relaxation in glassy liquids. *Nature Physics*, 12(5):469, 2016.

Samuel S Schoenholz, Ekin D Cubuk, Efthimios Kaxiras, and Andrea J Liu. Relationship between local structure and relaxation in out-of-equilibrium glassy systems. *Proceedings of the National Academy of Sciences*, 114(2):263–267, 2017.

Maria Schuld, Ville Bergholm, Christian Gogolin, Josh Izaac, and Nathan Killoran. Evaluating analytic gradients on quantum hardware. *Physical Review A*, 99(3):032331, 2019.

Atsuto Seko, Akira Takahashi, and Isao Tanaka. First-principles interatomic potentials for ten elemental metals via compressed sensing. *Physical Review B*, 92(5):054113, 2015.

Austin D Sendek, Ekin D Cubuk, Evan R Antoniuk, Gowoon Cheon, Yi Cui, and Evan J Reed. Machine learning-assisted discovery of solid li-ion conducting materials. *Chemistry of Materials*, 31(2):342–352, 2018.

Tristan A Sharp, Spencer L Thomas, Ekin D Cubuk, Samuel S Schoenholz, David J Srolovitz, and Andrea J Liu. Machine learning determination of atomic dynamics at grain boundaries. *Proceedings of the National Academy of Sciences*, 115(43):10943–10947, 2018.

José M Soler, Emilio Artacho, Julian D Gale, Alberto García, Javier Junquera, Pablo Ordejón, and Daniel Sánchez-Portal. The siesta method for ab initio order-n materials simulation. *Journal of Physics: Condensed Matter*, 14(11):2745, 2002.

Qiming Sun, Timothy C Berkelbach, Nick S Blunt, George H Booth, Sheng Guo, Zhendong Li, Junzi Liu, James D McClain, Elvira R Sayfutyarova, Sandeep Sharma, et al. Pyscf: the python-based simulations of chemistry framework. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 8(1):e1340, 2018.

Daniel M Sussman, Samuel S Schoenholz, Ekin D Cubuk, and Andrea J Liu. Disconnecting structure and dynamics in glassy thin films. *Proceedings of the National Academy of Sciences*, 114(40): 10601–10605, 2017.

Robert H Swendsen and Jian-Sheng Wang. Replica monte carlo simulation of spin-glasses. *Physical review letters*, 57(21):2607, 1986.

Teresa Tamayo-Mendoza, Christoph Kreisbeck, Roland Lindh, and Alán Aspuru-Guzik. Automatic differentiation in quantum chemistry with applications to fully variational hartree–fock. *ACS central science*, 4(5):559–566, 2018.

Jeffrey P Thomas, Earl H Dowell, and Kenneth C Hall. Using automatic differentiation to create a nonlinear reduced-order-model aerodynamic solver. *AIAA journal*, 48(1):19–24, 2010.

Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, volume 5, pp. 1–6, 2015.

Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22, 2011.

Adri CT Van Duin, Siddharth Dasgupta, Francois Lorant, and William A Goddard. Reaxff: a reactive force field for hydrocarbons. *The Journal of Physical Chemistry A*, 105(41):9396–9409, 2001.

Andrea Walther. Automatic differentiation of explicit runge-kutta methods for optimal control. *Computational Optimization and Applications*, 36(1):83–108, 2007.

Qian Yang, Carlos A Sing-Long, and Evan J Reed. Learning reduced kinetic monte carlo models of complex chemistry from molecular dynamics. *Chemical science*, 8(8):5781–5796, 2017.

Kun Yao, John E Herr, David W Toth, Ryker Mckintyre, and John Parkhill. The tensormol-0.1 model chemistry: A neural network augmented with long-range physics. *Chemical science*, 9(8): 2261–2269, 2018.

A Architecture

We begin by briefly describing JAX before discussing the architectural choices we made in designing JAX MD. JAX is the successor to Autograd and shares key design features. As with Autograd, the main user-facing API of JAX is in one-to-one correspondence with Numpy (Van Der Walt et al., 2011), the ubiquitous numerical computing library for Python. On top of this, JAX implements sophisticated "tracing" machinery that takes arbitrary python functions and builds an Abstract Syntax Tree (AST) for the function called a "Jaxpr". JAX includes a number of transformations that can be applied to Jaxprs to produce new Jaxprs. Examples of such transformations are: automatic differentiation (`grad`), vectorization on a single device (`vmap`), parallelization across multiple devices (`pmap`), and just-in-time compilation (`jit`). To see an example of this see Appendix F, which shows that the `grad` function ingests a function and returns a new, transformed function that computes its gradient. This is emblematic of JAX's functional design; all transformations take functions and return transformed functions. These function transformations are arbitrarily composable so one can write e.g. `jit(vmap(grad(f)))` to just-in-time compile a function that computes per example gradients of a function f. As discussed above, JAX makes heavy use of the accelerated linear algebra library, XLA, which allows compiled functions to be run in a single call on CPU, GPU, or TPU. This effectively removes execution speed issues that generally face Python programs.

JAX MD adopts a similarly functional style with immutable data and first-class functions. In a further departure from most other MD software, JAX MD features no classes or object hierarchies and instead uses a data driven approach that involves transforming arrays of data. JAX MD often uses named tuples to organize collections of arrays. This functional and data-oriented approach complements JAX's style and makes it easy to apply the range of function transformations that JAX provides. JAX MD makes extensive use of automatic differentiation and automatic vectorization to concisely express ideas (e.g. force as the negative gradient of energy) that are challenging in more conventional libraries. Since JAX MD leverages XLA to compile whole simulations to single GPU calls, it can be entirely written in Python while still being extremely fast. Together this means that implementing simulations in JAX MD looks almost verbatim like textbook descriptions of the subject. We now go through the major systems underlying JAX MD.

A.1 Spaces

In MD we simulate a collection of N particles in either two- or three-dimensions. In the simplest case, these particles are defined by a collection of position vectors, $\{\vec{r}_i\}_{1 \leq i \leq N}$. Some simulations are performed with $\vec{r}_i \in \mathbb{R}^D$ where $D = 2, 3$ is the spatial dimension of the system; this is implemented in JAX MD using the `space.free()` function. However, as discussed in Section B, it is more common to use periodic boundary conditions in which case $\vec{r}_i \in \mathbb{R}^D$ with the association that $\vec{r}_i = \vec{r}_i + L\hat{e}_k$

for basis vectors e_k and some "box size" L. In this case the simulation space is homeomorphic to a D-torus; this is implemented in JAX MD using the `space.periodic(box_size)` function.

These boundary conditions can be implemented by defining two functions. First, a function $\vec{d} : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}^D$ that computes the displacement between two particles. This function can in turn be used to define a metric on the space $d : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$ by $d(\vec{a}, \vec{b}) = |\vec{d}(\vec{a}, \vec{b})|_2$. Note, that in systems with periodic boundary conditions $\vec{d}$ computes the displacement between a particle and the nearest "image" of the second particle. Second, a shift function $\mu : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}^D$ must be defined that moves a particle by a shift. Motivated by this, in JAX MD we implement the spaces outlined above by functions that return a "displacement" and "shift" functions. We show an example below.

```
import jax.numpy as np
from jax_md import space
r_1 = np.array([0.5, 0, 0])
r_2 = np.array([0, 0.5, 0])

displacement, shift = space.periodic(1.)
dR = displacement(r_1, r_2, t=0.1)
```

A.2 Energy and Forces

As discussed above, MD simulations often proceed by defining a potential energy function, $U(\{\vec{r}_i\})$, between particles. The degree to which $U(\{\vec{r}_i\})$ approximates reality has a significant influence of the fidelity of the simulation. For this reason, approximating potential energy functions has received significant attention from the ML community. JAX MD allows potential energy functions to be arbitrary "JAX traceable" functions, $E : \mathbb{R}^{N \times D} \to \mathbb{R}$, including arbitrary neural networks.

JAX MD provides a number of predefined, common, choices of energy functions including several pair potentials - Lennard-Jones, soft-sphere, and Morse - as well as the Embedded Atom (Daw & Baskes, 1984) many-body potential and soft-spring bonds. Functions to compute the energy of a system are constructed by providing a displacement function for example, `energy_fun = energy.soft_sphere_pair(displacement)`. Forces can easily be computed using a helper function `quantity.force(energy_fun)` which is a thin wrapper around `grad`. In addition to the pre-defined energy functions, it is easy to add new potential energy functions to JAX MD. In Section 2.2 we show how to add a neural network many-body potential called the Behler-Perrinello (Behler, 2011). In sec. D we describe some additional tools provided by JAX MD to easily define custom energy functions.

In many applications, the scalar function $u(r)$ has compact support such that $u(r) = 0$ if $r > r_c$ for some cutoff, $r_c \in \mathbb{R}$. We say that particles are not interacting if $r_{ij} > r_c$. The pairwise function defined in Eq. (4) is wasteful in this case since the number of computations scales as $O(N^2)$ even though the total number of interactions scales as $O(N)$. To improve calculations in this case we provide the function `cell_list_fun = smap.cell_list(fun, box_size, r_c, example_positions)` that takes a function and returns a new function that uses spatial partitioning to provide a speed up.

A.3 Dynamics and Simulations

Once an energy function has been defined, there are a number of simulations that can be run. JAX MD supports simple constant energy (NVE) simulation as well as Nose-Hoover (Martyna et al., 1992), Langevin, and Brownian simulations at constant temperature (NVT). JAX MD also supports Fast Inertial Relaxation Engine (Fire) Descent (Bitzek et al., 2006) and Gradient Descent to minimize the energy of systems. All simulations in JAX MD follow a pattern that is inspired by JAX's optimizers; for simplicity, we will use Brownian motion as an example in this section. Simulations are pairs of two functions: an initialization function, `state = init_fun(key, positions)`, that takes particle positions and returns an initial simulation state and an update function, `state = update_fun(state)`, that takes a state and applies a single update step to the state. To see an example of this in the case of Brownian motion, see code from the warm-up in sec. C. Simulation functions can also feature time-dependent temperatures or spaces in which case a time parameter can be passed to the update function, `state = update_fun(state, t=t)`.

We now continue from example **??** to show how to simulate the particles for a few steps of Brownian motion.

```
from jax_md import simulate

dt = 1e-3
temperature = 0.1

init_fun, update_fun = simulate.brownian(
       energy_fun, shift, dt, temperature)
state = init_fun(key, positions)

for _ in range(10):
    state = update_fun(state)
```

Here the `simulate.brownian` takes an energy function and a shift function along with a timestep and a temperature. It returns the initialization function and update function defined above. Here the energy function can be any potential energy function and the shift function is as described in Section A.1.

B Simulating a Bubble Raft

We begin with a lightning introduction to MD simulations. As an example, we're going to imagine some bubbles floating on water so that they live on a two-dimensional interface between water and air. We describe N bubbles by positions, $\{\vec{r}_i\}_{1\leq i\leq N}$. Since the bubbles are confined to the water's surface, the positions will be 2-dimensional vectors, $\vec{r}_i \in \mathbb{R}^2$. For simplicity, we can assume that the bubbles all are the same size and let their diameter be 1 without a loss of generality. We now have to posit interactions between the simulated bubbles that qualitatively model how real bubbles behave. More accurate interactions will produce more accurate simulations, which will in turn capture more realistic phenomena. For the purposes of this example, we assume that bubbles interact with each other in pairs. We model pairs of bubbles by defining an energy function for the pair that depends only on the distance between them. We will choose an energy that is zero if the bubbles aren't touching and then increases gradually as they get pushed together. Specifically, if r_{ij} is the distance between bubble i and j, we use a pairwise energy function,

$$U(r_{ij}) = \begin{cases} (1 - r_{ij})^2 & \text{if } r_{ij} < 1 \\ 0 & \text{if } r_{ij} > 1 \end{cases} \tag{2}$$

Once an energy has been defined we can compute the forces on a bubble, F_i, as the negative gradient of the energy. From their definition, we see that forces move bubbles to minimize their energy. From Eq. (2) low energy configurations will be those where bubbles are touching as little as possible. However, bubbles are situated on water which is full of water molecules that are moving around. These water molecules bump into the bubbles and push them slightly. To model the interaction between the bubbles and the water we will assume that there are very small random forces from the water that push the bubbles. This is a model called Brownian motion and it is described by a first-order differential equation relating the velocity of bubbles to the forces on them along with random kicks coming from the water,

$$\frac{d\vec{r}_i(t)}{dt} = \vec{F}_i(t) + \sqrt{2k_BT}\vec{\xi}_i(t) \tag{3}$$

Here $\vec{F}_i(t)$ are forces, $\vec{\xi}_i \sim \mathcal{N}(0, 1)$ is i.i.d. Gaussian distributed noise, and k_BT specifies the temperature of the water. Incidentally, this model of objects in water dates back to Einstein (1905).

In Appendix C we show an example where we define a function, `final_positions = simulation(rng_key)`, that takes a random number generator state and returns the final positions of the particles after simulating for some time. In this example, although we only simulated a small number of bubbles we were able to emulate a much large bubble raft by using what are known as "periodic boundary conditions" (which are used in the popular game, "Asteroids"). With periodic boundary conditions bubbles can wrap around the side of the simulation to the other side, this is a ubiquitous technique for simulating the "bulk" properties of a system. In Appendix C we also show

figures from a real experiment compared with the results of the simulation which shows striking similarities despite the significant simplifying assumptions we made in defining our simulation.

C Bubble Raft Example Code

Listing 1: A simulation function that takes a random key and returns final particle positions.

```
N = 32
dt = 1e-1
temperature = 0.1

simulation_steps = np.arange(1000)
key = random.PRNGKey(0)

box_size = box_size_at_number_density(particle_count=N, number_density=1)
displacement, shift = space.periodic(box_size)

energy_fun = energy.soft_sphere_pair(displacement)

def simulation(key):
  pos_key, sim_key = random.split(key)

  R = random.uniform(pos_key, (N, 2), maxval=box_size)

  init_fn, apply_fn = simulate.brownian(
        energy_fun, shift, dt, temperature)
  state = init_fn(sim_key, R)

  do_step = lambda state, t: (apply_fn(state, t=t), t)
  state, _ = lax.scan(do_step, state, simulation_steps)

  return state.position

positions = simulation(key)
```

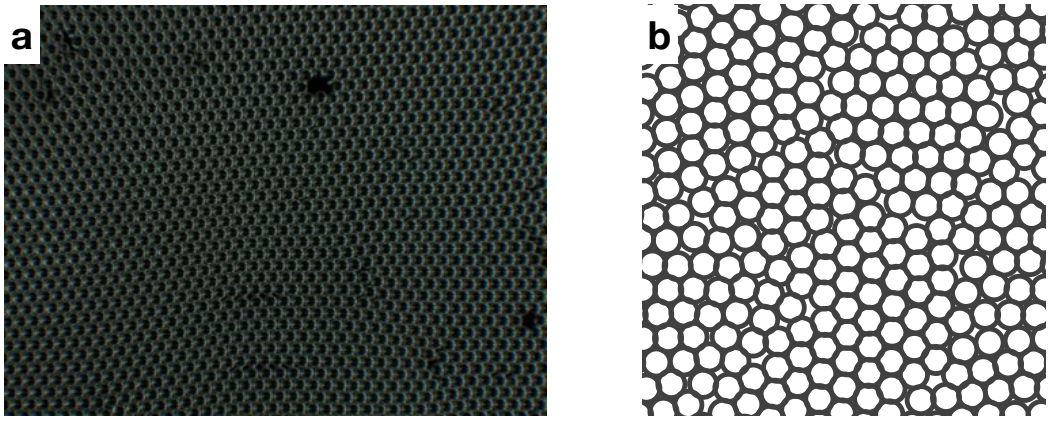

Figure 4: **An experiment of a bubble raft compared with the results of a simulation.**

D Defining Custom Potentials

Many popular potential energy functions are either pairwise or bonded in the sense that

$$U_{\text{pair}}(\{\vec{r}_i\}) = \sum_{1 \le i < j \le N} u(r_{ij}) \qquad \text{or} \qquad U_{\text{bond}}(\{\vec{r}_i\}) = \sum_{1 \le i \le B} u(r_{b_{i1} b_{i2}}) \tag{4}$$

where r_{ij} is the distance between particles i and j and $\{b_{ia}\}_{1\leq i\leq B}$ indexes a bond between particles b_{i1} and b_{i2} for a total of B bonds. In this case, we offer the functions `energy_fun = smap.pair(scalar_energy_fun, displacement_fun)` and `energy_fun = smap.bond(scalar_energy_fun, displacement_fun)` that will convert a scalar function, $u(r)$, to the either pair-potential defined in Eq. (4). An example of this is shown below.

```
from jax_md import smap

scalar_energy_fun = lambda r, **kwargs: r ** 2
metric_fun = space.metric(displacement_fun)
pair_energy_fun = smap.pair(scalar_energy_fun, metric_fun)
E_pair = pair_energy_fun(positions)

bonds = np.array([[0, 1]])
bond_energy_fun = smap.bond(scalar_energy_fun, metric_fun, bonds)
E_bond = bond_energy_fun(positions)
```

The difference between these two functions amounts to the choice of whether to use d_{pair} or d_{bond} defined above. The two-line examples above and in Section 2.2 should be contrasted with the significant undertaking that would be required to implement these features in traditional MD packages.

E Optimization Through Dynamics

If there are N bubbles then the total volume of water filled by bubbles is,

$$V_{\text{bubbles}} = \frac{N}{8}\pi(D^2 + 1)$$

where the factor of 8 comes from the fact that our system is split into two halves and we are using diameters not radii. Since the volume of our simulation is $V = L^2$ if we want to keep the "packing fraction", $\phi = V_{\text{bubbles}}/V$ constant then we will have to scale the size of the box to be, $L = \sqrt{\frac{V_{\text{bubbles}}}{\phi}}$.

F Example Jaxpr and its gradient

$f(x) = x^3$

Listing 2: Python

```
def f(x):
    return x ** 3
print(f(2.0)) # 8.0
```

Listing 3: Jaxpr

```
{ lambda ;  ; a.
  let b = pow a 3.0
  in [b] }
```

$\frac{df}{dx} = 3x^2$

Listing 4: Python

```
from jax.api import grad
df_dx = grad(f)
print(df_dx(2.0)) # 12.0
```

Listing 5: Jaxpr

```
{ lambda ;  ; a.
  let b = pow a 2.0
      c = mul 3.0 b
      d = safe_mul 1.0 c
  in [d] }
```

G Related Work

Automatic differentiation has enjoyed a rich history in machine learning as well as the physical sciences (Baydin et al., 2018). Backprop has been the core algorithm in the recent explosion of ML research, enabled by packages like TensorFlow (Abadi et al., 2016), Torch (Collobert et al., 2002), and Theano (Bastien et al., 2012). While still generally focused on ML, more recent packages have made automatic differentiation more generally available as a computational tool (e.g. Autograd

(Maclaurin et al., 2015), JAX (Bradbury et al., 2018; Frostig et al., 2018), PyTorch (Paszke et al., 2017), Chainer (Tokui et al., 2015), and Zygote (Innes et al., 2019)).

In the physical sciences, automatic differentiation has been applied to a large variety of problems in structural optimization (Hoyer et al., 2019), quantum chemistry (Tamayo-Mendoza et al., 2018), fluid dynamics (Müller & Cusdin, 2005; Thomas et al., 2010; Bischof et al., 2007), computational finance (Capriotti, 2010), atmospheric modelling (Charpentier & Ghemires, 2000; Carmichael et al., 1997), optimal control (Walther, 2007), physics engines (de Avila Belbute-Peres et al., 2018), protein modelling (Ingraham et al., 2018; AlQuraishi, 2019), and quantum circuits (Schuld et al., 2019). For further related work at the intersection of ML and MD, please see Appendix G. Despite significant work on the topic, the number of general purpose simulation environments that are integrated with AD is scarce.

General MD packages have been widely used to simulate molecules and solids, either using first-principles potentials (using software packages that derive the potential from quantum mechanics (Car & Parrinello, 1985) e.g. Quantum Espresso (Giannozzi et al., 2009), VASP (Hafner, 2008), SIESTA (Soler et al., 2002), GPAW (Enkovaara et al., 2010), CASTEP (Clark et al., 2005), PySCF (Sun et al., 2018)) or with empirical potentials (using approximate potentials that descrive specific atomic interactions e.g. LAMMPS (Plimpton, 1995), HOOMD-Blue (Anderson et al., 2008; Glaser et al., 2015), and OpenMM (Eastman et al., 2017)). HOOMD-Blue in particular has been built with GPU acceleration in mind from the beginning, with the ability to script MD experiments using Python.

Coupled with the growing interest in deep learning, machine learning (ML) has become a popular tool for analyzing data that is produced by MD (Cubuk et al., 2015; Schoenholz et al., 2016; Cubuk et al., 2016; Schoenholz et al., 2017; Cubuk et al., 2017b; Schoenholz, 2018; Rajak et al., 2019b,a; Sharp et al., 2018; Sussman et al., 2017; Hanakata et al., 2018; Sendek et al., 2018; Yang et al., 2017; Ma et al., 2019), as well as making MD simulations faster and more accurate (Behler & Parrinello, 2007; Behler, 2011; Artrith et al., 2011; Artrith & Behler, 2012; Artrith et al., 2018; Deringer et al., 2018a; Bartók et al., 2018; Yao et al., 2018; Seko et al., 2015; Deringer et al., 2018b; Cubuk et al., 2017a; Faber et al., 2017; Gilmer et al., 2017).