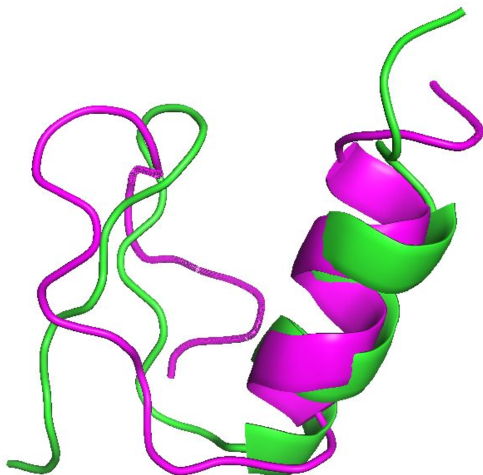


Differentiable molecular simulation with Molly.jl

Joe Greener
MRC Laboratory of Molecular Biology
http://jgreener64.github.io

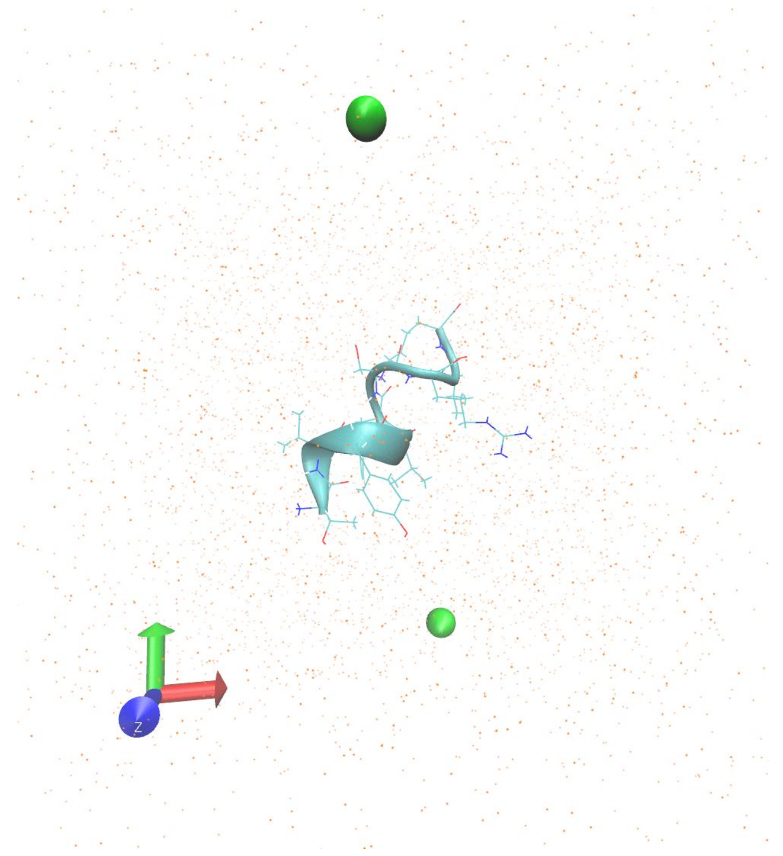
Molecular dynamics (MD)

- Set up a physical system, define the rules that determine the forces and press play.

- Numerical integration of F = ma. Often very computationally expensive, there are few shortcuts.

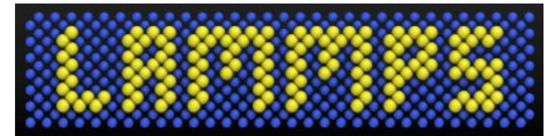
- MD has helped us understand many processes in chemistry and biology.

- Ever-improving compute resources, innovative machine learning approaches and better protein structure prediction mean these methods will continue to develop.



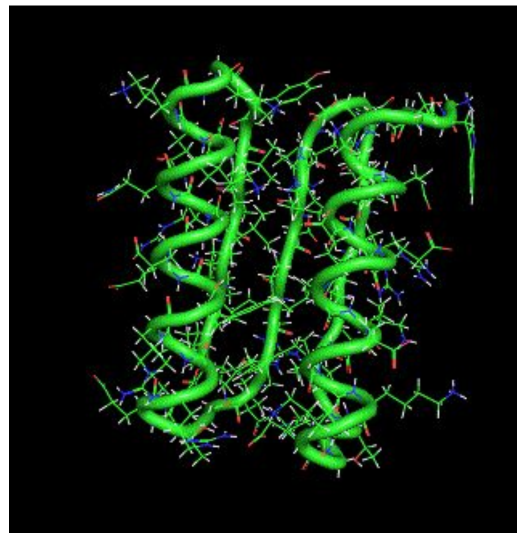
Software for molecular simulation

- There are a number of excellent, very mature software packages for MD.
- Most packages have one or more fast kernels to run simulations (C, C++, CUDA, Fortran), and a layer on top to interact with (binaries, Python, config files).
- Most packages are hard to interact with all the way down and have a mixed ability to customise.
- No mature MD package has support for differentiable simulations.



Molly.jl

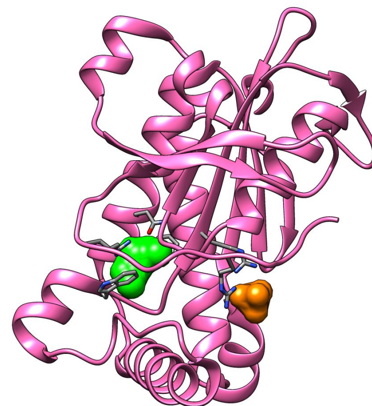
- A pure Julia implementation of MD. One language all the way down. Simulation scripts are Julia scripts.
- Closest in design to OpenMM, which has a Python API (but multiple kernels under the hood).
- User-defined potentials, simulators etc. are easy to define and as fast as built-in features. Everything is defined imperatively in Julia.
- Under active development, not stable or fully covered by tests yet. Can simulate standard proteins with the trajectories matching OpenMM.



Foldit1 (PDB ID 6MRR) simulated with Molly.jl in the a99SB-ILDN force field with explicit solvent (not shown).

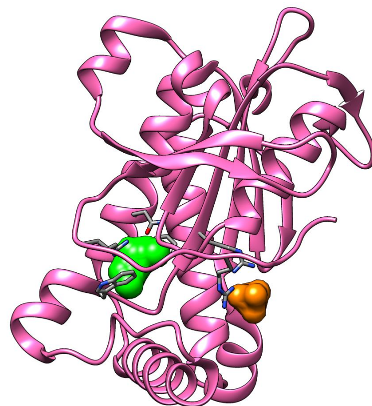
Features

- Non-bonded interactions: Lennard-Jones, Coulomb (plus reaction field), gravity, soft sphere, Mie
- Bonded interactions: harmonic bonds and angles, Morse/FENE bonds, cosine angles, periodic torsion angles
- Read in OpenMM and Gromacs force field files and coordinate files using Chemfiles.jl
- Implements AtomsBase.jl interface
- Verlet, velocity Verlet, Störmer-Verlet and flexible Langevin integrators
- Steepest descent energy minimisation
- Andersen, Berendsen and rescaling thermostats
- Periodic and infinite boundary conditions in a cubic/triclinic box



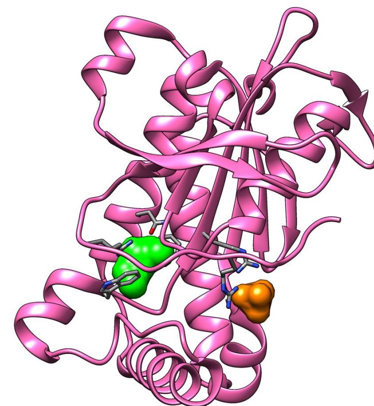
Features continued

- Flexible loggers to track arbitrary properties throughout simulations
- Cutoff algorithms for non-bonded interactions
- Various neighbour list implementations to speed up calculation of non-bonded forces, including use of CellListMap.jl
- Implicit solvent GBSA methods
- Unitful.jl compatible
- Some analysis functions, e.g. RDF
- Basic visualisation with GLMakie.jl
- Runs on CPU (threaded) or GPU



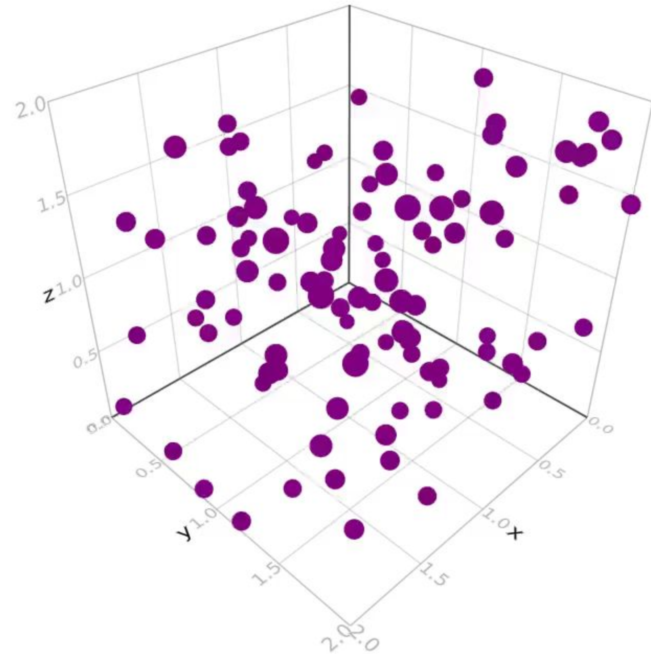
Missing features

- Constrained bonds and angles
- Particle mesh Ewald summation
- Pressure coupling and other temperature coupling methods
- System preparation - solvent box, add hydrogens etc.
- Domain decomposition algorithms
- Alchemical free energy calculations
- API stability
- High test coverage
- High performance



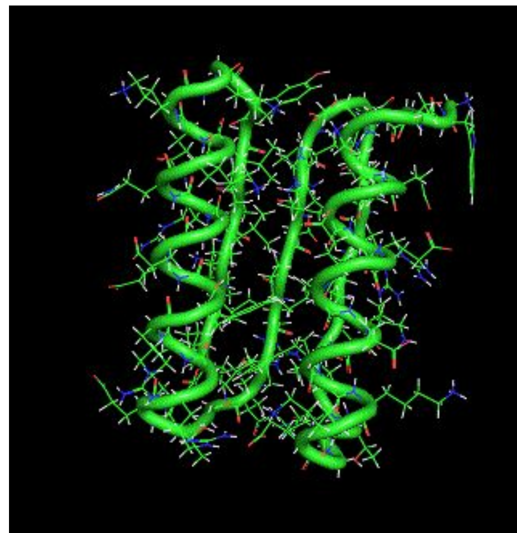
Can define components individually

```
using Molly
using GLMakie

n_atoms = 100
boundary = CubicBoundary(2.0u"nm", 2.0u"nm", 2.0u"nm")
temp = 298.0u"K"
atom_mass = 10.0u"u"

atoms = [Atom(mass=atom_mass, σ=0.3u"nm", ϵ=0.2u"kJ * mol^-1") for i in 1:n_atoms]
coords = place_atoms(n_atoms, boundary, 0.3u"nm")
velocities = [velocity(atom_mass, temp) for i in 1:n_atoms]
pairwise_inters = (LennardJones(),)
simulator = VelocityVerlet(
    dt=0.002u"ps",
    coupling=AndersenThermostat(temp, 1.0u"ps"),
)

sys = System(
    atoms=atoms,
    pairwise_inters=pairwise_inters,
    coords=coords,
    velocities=velocities,
    boundary=boundary,
    loggers=(coords=CoordinateLogger(10),),
)

simulate!(sys, simulator, 10_000)

visualize(sys.loggers.coords, boundary, "sim.mp4")
```



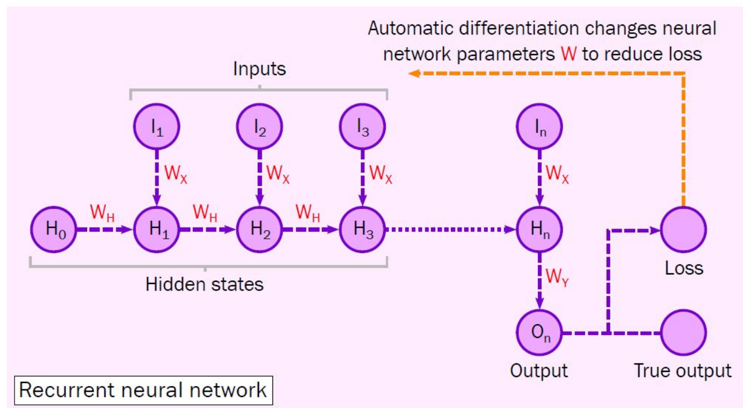
Can setup like OpenMM, but runs in one language

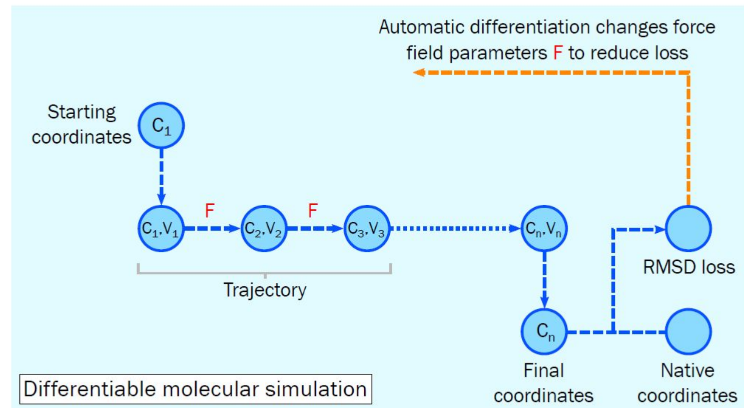
```julia
using Molly

ff = OpenMMForceField("ff99SBildn.xml", "tip3p_standard.xml")

sys = System(
    "6mrr_equil.pdb",
    ff;
    loggers=(
        energy=TotalEnergyLogger(10),
        writer=StructureWriter(10, "traj_6mrr_1ps.pdb", ["HOH"]),
    ),
)

minimizer = SteepestDescentMinimizer()
simulate!(sys, minimizer)

random_velocities!(sys, 298.0u"K")
simulator = Langevin(
    dt=0.001u"ps",
    temperature=300.0u"K",
    friction=1.0u"ps^-1",
)

simulate!(sys, simulator, 5_000; n_threads=Threads.nthreads())
```



Foldit1 (PDB ID 6MRR) simulated with Molly.jl in the a99SB-ILDN force field with explicit solvent (not shown).

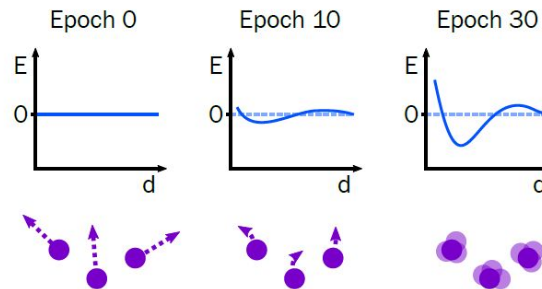
Differentiable molecular simulation



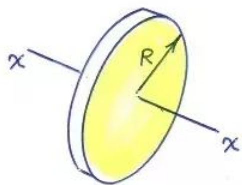


Learns to stabilise native structure

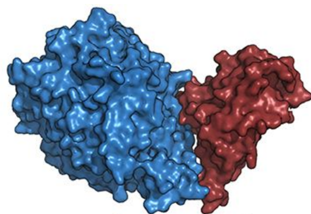
See Greener and Jones, PLoS ONE 16(9): e0256990 (2021)



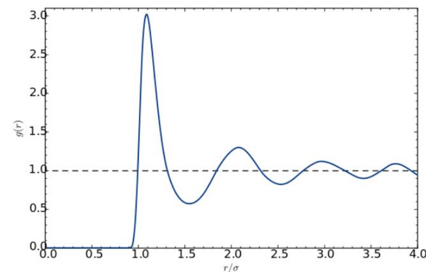
The variety of possible loss functions



Radius of gyration

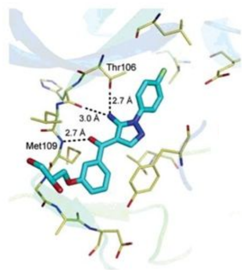


Protein-protein interaction

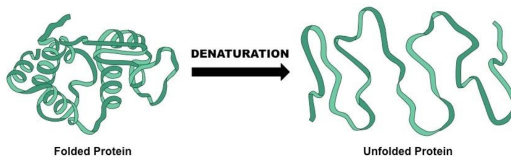


Radial distribution function

Flexibility

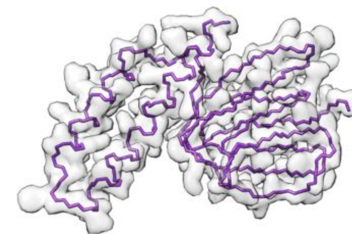


Protein-ligand binding



Phase change

Supramolecular geometry



Fit to experimental data

Software for differentiable simulations

- Some packages do exist for differentiable simulations:
    - Jax MD (in Jax)
    - TorchMD (in PyTorch)
    - Taichi (Python-based)

- These are all promising but are limited in some way by scope or performance.

- Molly can do differentiable simulations, including on proteins, and is being actively used for research in this area.

Automatic differentiation over thousands of steps

Method	*Description*	*Advantages*	*Disadvantages*
Reverse mode	Record computation graph, compute chain rule backwards from final state	Compute time independent of parameter number (hence most deep learning uses this)	Memory scales linearly with model depth, limiting MD steps (though can use checkpointing)
Forward mode	Pass value and gradient together, compute chain rule forwards from initial state	Memory independent of model depth	Compute time scales linearly with parameter number, finite differencing may be faster
Adjoint sensitivity	Solve an augmented ODE of the adjoint back in time	Memory independent of model depth, fast for reversible models (MD can be reversible)	Limited implementations, limited guidance

More in “Automatic differentiation in machine learning: a survey”, Baydin et al. arXiv 2015
and “Neural Ordinary Differential Equations”, Chen et al. NeurIPS 2018

GPU memory requirements

- During training, the need to store values for reverse-mode automatic differentiation means that the memory required scales with the number of steps.
- This is not a problem when using the learned potential for simulations.
- Potential solutions:
  - Forward-mode automatic differentiation.
  - Gradient checkpointing.
  - Invertible simulations.

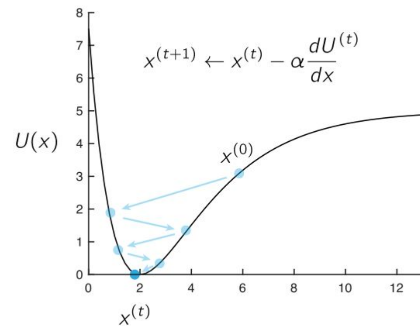
From https://en.wikipedia.org/wiki/Automatic_differentiation

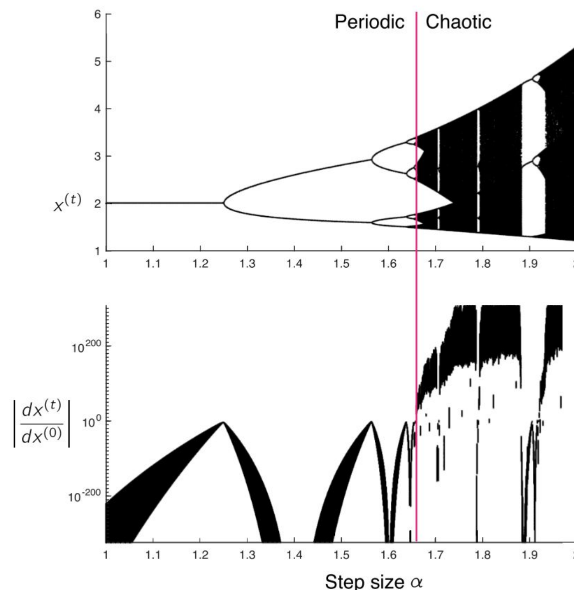
checkpoint

From https://github.com/cybertronai/gradient-checkpointing

Exploding gradients

- Automatic differentiation gives exact gradients but with respect to the numerical integration.
- Some functional forms of force fields, e.g. hard sphere interactions, will give exploding gradients when used with standard integrators.
- Which integrators are suitable for taking gradients through? Is a more conservative time step required?
- Fortunately there is lots of prior work on stabilising gradients through deep RNNs.

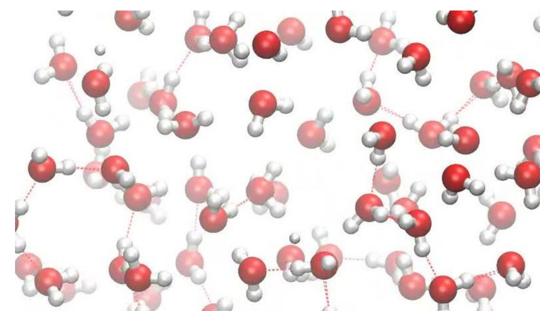




From Ingraham et al. 2019

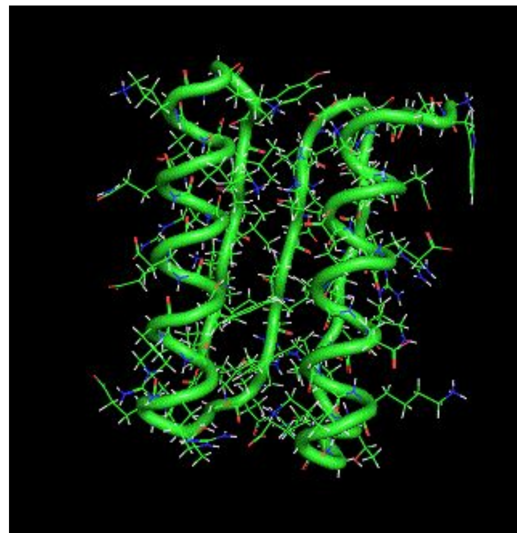
Algorithmic challenges

- Long-range electrostatics with particle-mesh Ewald: difficult to implement, let alone differentiably. Currently using reaction field.
- Bond and angle constraints. A smaller time step should be used if not constraining bonds.
- Stochastic simulations, e.g. using certain thermostats during training or Langevin dynamics.
- Neighbour lists: not required to be differentiable since output is binary.



Differentiability in Molly.jl

- Up to now, Zygote.jl has been used for AD (with ForwardDiff.jl to speed up broadcasting). Gradients match finite differencing for reverse and forward mode AD.
- The requirements of Zygote - no mutation or GPU kernels - mean broadcasting over the whole neighbour list. This leads to poor memory usage and GPU performance.
- Recently the force/energy summation algorithms have been re-written to use mutation on CPU and as CUDA.jl kernels on GPU.
- AD for these is carried out with Enzyme.jl. Seems to work!
- Performance and memory usage are vastly improved. Currently on `kernels` branch.



Foldit1 (PDB ID 6MRR) simulated with Molly.jl in the a99SB-ILDN force field with explicit solvent (not shown).

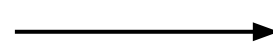
CUDA kernels

- Current force summation kernel is simple: each thread calculates the force for one neighbouring pair and atomically adds it to the force array.
- Performance is actually okay, ~5x slower than the equivalent in OpenMM without serious optimisation.
- Enzyme is the only way to differentiate through Julia CUDA kernels like this.
- The next challenge: mature MD packages use clever neighbour ordering and reductions in much more complex kernels.

```
function pairwise_force_kernel!(forces::CuDeviceMatrix{T}, coords_var, atoms_var, boundary,
                                inters, neighbors_var, ::Val{D}, ::Val{F}) where {T, D, F}
    coords = CUDA.Const(coords_var)
    atoms = CUDA.Const(atoms_var)
    neighbors = CUDA.Const(neighbors_var)

    tidx = threadIdx().x
    inter_i = (blockIdx().x - 1) * blockDim().x + tidx

    if inter_i <= length(neighbors)
        i, j, weight_14 = neighbors[inter_i]
        coord_i, coord_j = coords[i], coords[j]
        dr = vector(coord_i, coord_j, boundary)
        f = force_gpu(inters[1], dr, coord_i, coord_j, atoms[i], atoms[j], boundary, weight_14)
        for inter in inters[2:end]
            f += force_gpu(inter, dr, coord_i, coord_j, atoms[i], atoms[j], boundary, weight_14)
        end
        if unit(f[1]) != F
            # This triggers an error but it isn't printed
            # See https://discourse.julialang.org/t/error-handling-in-cuda-kernels/79692
            #   for how to throw a more meaningful error
            error("Wrong force unit returned, was expecting $F but got $(unit(f[1]))")
        end
        for dim in 1:D
            fval = ustrip(f[dim])
            if !iszero(fval)
                Atomix.@atomic :monotonic forces[dim, i] += -fval
                Atomix.@atomic :monotonic forces[dim, j] +=  fval
            end
        end
    end
    return nothing
end
```

Differentiable simulation in Molly.jl

- Run DMS on small proteins in Molly:
  - *Alanine dipeptide in water (2,917 atoms)*:
    ~25x ms per step with gradient on GPU,
    ~14 hours for 1 ns.
  - *Trp-cage with GBSA implicit solvent (284 atoms)*:
    ~12 ms per step with gradient on GPU,
    ~17 hours for 5 ns.



- Achievable to improve all-atom implicit solvent force fields with this iteration of the software.

- With further optimisation explicit solvent force fields will be in reach for improvement.

atom_N_σ
0.01064

atom_H_mass
0.0007533

inter_CO_coulomb_const
7.475e-7

inter_LJ_weight_14
0.001836

inter_PT_C/N/CT/C_k_1
-2.017e-5

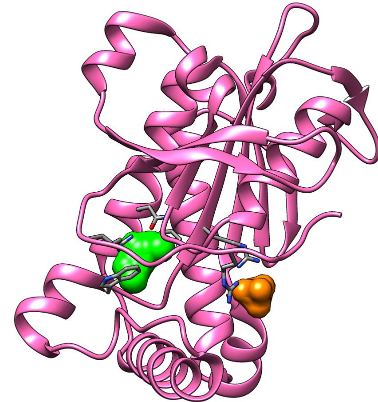
Sample gradients after 1 ns (10^6 steps), loss is RMSD to starting structure

Ongoing work - contributions welcome!

- Development currently very active, many features added in the last few months.

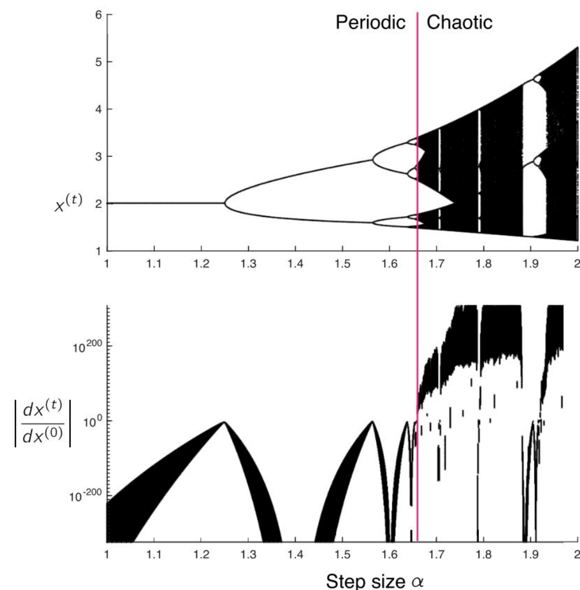
- High performance, differentiable GPU kernels will be a focus for development:
  - Force/energy summation.
  - Particle-mesh Ewald summation.
  - Neighbour lists - doesn’t need to be differentiable but does need to be fast.

- Will be advertising for postdocs and PhD students towards the end of the year.



Lessons from taking gradients through long simulations

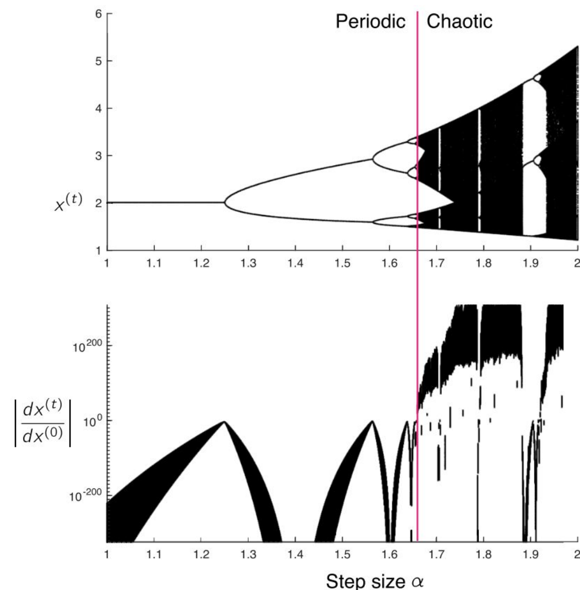
- Controlling temperature is important to prevent gradient explosion. Berendsen/rescaling thermostats seem to work.
- Stochastic simulators like Langevin dynamics are not to be feared; the stochasticity seems to have a regularising effect.
- You should be sampling over different starting conformations and velocities.
- Gradient norm clipping helps prevents gradient explosion like in deep RNNs. Make sure to clip all gradients equally!



From Ingraham et al. 2019

Lessons from taking gradients through long simulations

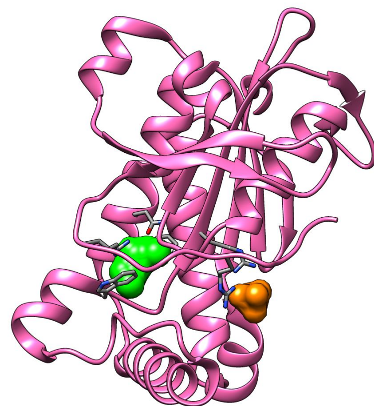
- Reverse mode AD with checkpointing and clipping every ~100 steps seems to work.
- Forward mode AD is generally slower than finite differencing with non-differentiable software.
- Float32 and 1 fs time step seems okay for implicit solvent molecular systems (no bond constraints).
- Getting accurate gradients is fiddly - only worth it over finite differencing for lots of parameters.
- Always test against finite differencing!



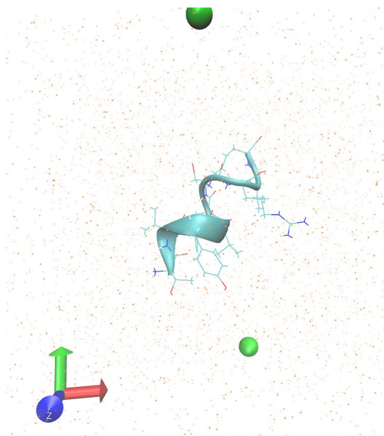
From Ingraham et al. 2019

Challenges for Enzyme

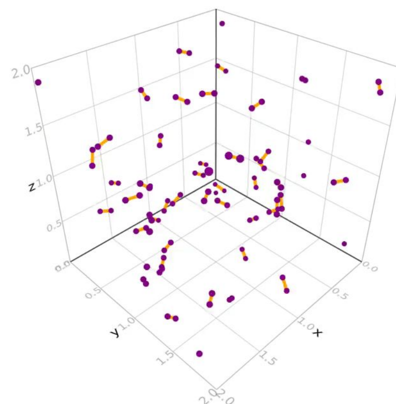
- Mixed CPU-GPU programming, i.e. allowing generic broadcasting of GPU arrays.
- Mature rules system, integrated with ChainRules.jl and the existing ecosystem.
- These two things would allow Molly to use Enzyme as its main AD, increasing speed and simplicity.
- Easier way to call Enzyme inside rrules, including for GPU kernels.
- Continued support for GPU programming, e.g. atomics and shared memory.



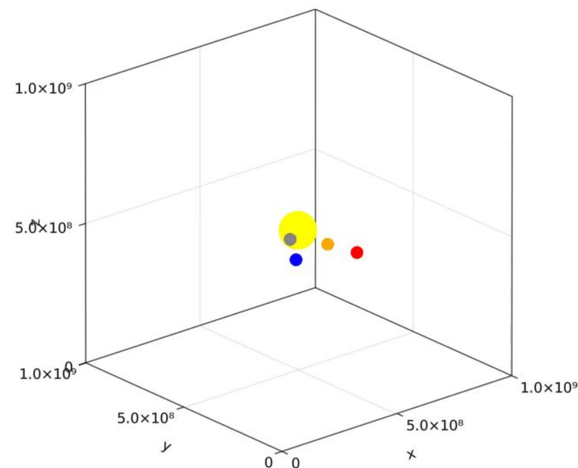
Peptide (shown with VMD)



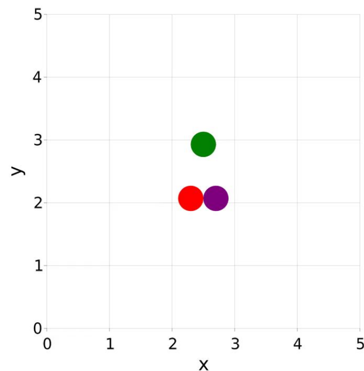
Diatomic gas



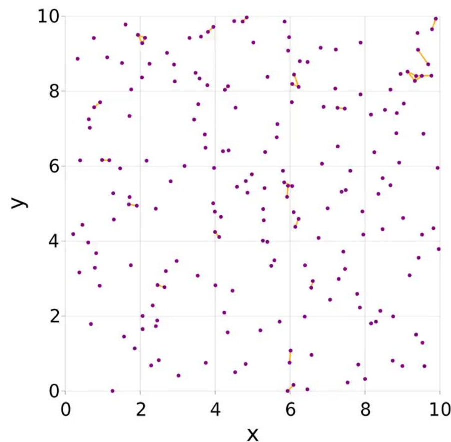
Solar system



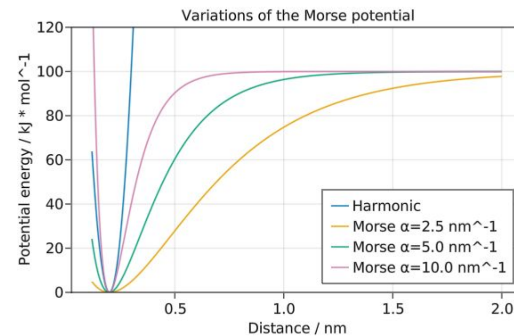
Training to reproduce a familiar logo



Making and breaking bonds



Plotting potential energies



Acknowledgements

- Molly.jl contributors (incomplete list):
  - Noé Blassel
  - Sebastian Micluța-Câmpeanu
  - Leandro Martínez
  - Jaydev Singh Rao
  - Ethan Meitz
  - Pranay Venkatesh
  - James Schloss
  - Ehsan Irani
  - Andrés Riedemann
  - Maximilian Scheurer
- All Julia contributors and package authors, particularly Zygote, CUDA, StaticArrays, Chemfiles, ChainRules, AtomsBase and Unitful
- William Moses, Valentin Churavy and the Enzyme team for all the bug fixes
- OpenMM and Gromacs
- Sjors Scheres group, MRC-LMB
- David Jones group, UCL