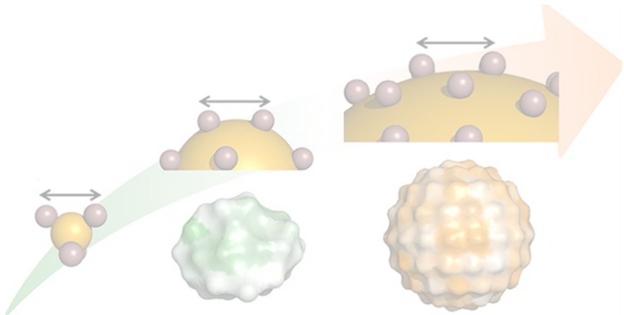
Atomistic Suite for CESMIX in julia

juliaLab MIT CSAIL MIT

Center for the Exascale Simulation of Materials in Extreme Environments

CESMIX

- ❖ General goal
  - ❖ *Advance the state-of-the-art in **predictive simulation.***
  - ❖ *Connect **quantum / molecular simulations** of materials with state-of-the-art **programming languages, compilers**, and **hpc.***
  - ❖ ***Uncertainty quantification***
- ❖ Motivating problem
  - ❖ *Predict the degradation of complex materials under extreme loading, inaccessible to direct experimental observation*
  - ❖ *In particular: **ultrahigh temperature ceramics in hypersonicflows**.*
- ❖ Supported by*: Department of Energy's Predictive Science Academic Alliance Program (PSAAP-III)*
- ❖ Interdisciplinary project: *CCSE, CSAIL, SDSC, external collaborations*



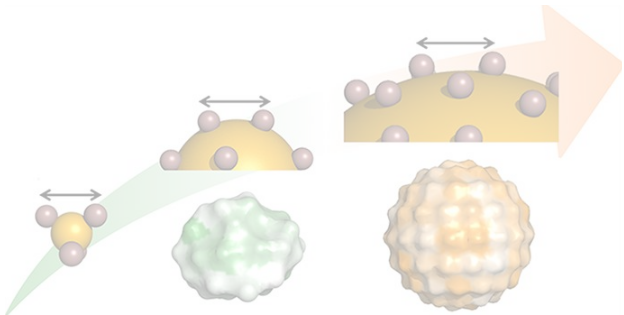
- General goals

  - ***Facilitate** the **development** of chemical and materials simulations in **CESMIX***

  - *Demonstrate that it is possible to address these simulations using*
    a single high-level programming language without compromising performance

  - ***Expand** the capabilities of the **atomistic ecosystem in Julia**, as well as its user **community***

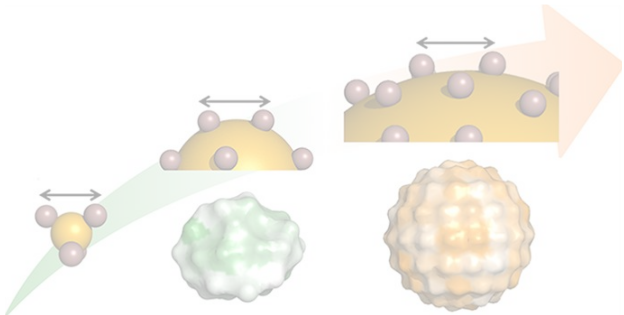
  - *Contribute to a **better integration** of the atomistic software ecosystem*



Atomistic Suite for CESMIX in julia

❖ Specific goals

- ❖ ***Composable workflows*** and *new software* ***abstractions***
- ❖ ***UQ-driven integration***
- ❖ ***AD*** *enabling* ***UQ, training, and force*** *calculations*
- ❖ *Simplify the* ***definition***, ***fitting,*** *and* ***integration*** *of* ***new*** *interatomic* ***potentials***
- ❖ *Integrate with state-of-the-art tools. E.g.* ***LAMMPS***.
- ❖ *Leverage* ***Julia***!



Software roadmap

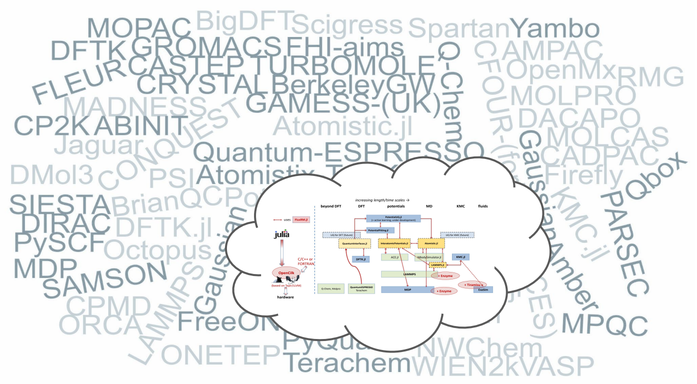
		Year 1	Year 2	Year 3	Years 4–5
CS tools	OpenCilk	Initial stable releases	Target new runtimes and backends	New tools using CSI	New parallel constructs in Tapir
	Enzyme	Forward mode/mixed mode	Parallel-specific, Julia parallelism	BLAS and split forward mode	Checkpointing
	Julia + OpenCilk	Start integration effort	Compiler and runtime integration	Performance eng. of integration	
	Julia + FluxRM	Integration (with LLNL)	Initial design for UQ workflows		
	Julia + Spindle	Integration			
	Tiramisu	Initial Exasim application	Improve MPI support		Adapt to DFT use cases
Interfaces/ abstractions	Julia + HPC arch	Tapir IR extension, HPC readiness	Parallel optimization, parallel runtimes	Perf opt for specific HPC archs	
	LAMMPS.jl	Initial design of LAMMPS.jl	Enzyme for AD in LAMMPS	FluxRM.jl support for key workflows	
	Atomistic.jl/etc	Initial design	Complete coverage; interface with ACE.jl, ASE, KMC.jl		
DFT	DFTK.jl	Forward-mode AD	Reverse AD; sensitivity analysis. DFTK.jl on GPUs	Expose multi-fidelity methods; GPU performance optimization	Explore DSL (Tiramisu) opportunities
MD + potentials	MDP	Initial design & development	Julia interfaces; profiling + optimization	Automated UQ + active learning workflows for MD; integration with FluxRM.jl	Integration of MD UQ workflow with KMC and fluids simulations
	Julia atomistic suite	Initial design & development	Integration of all potentials, DFT codes, MD simulators, KMC		
	KMC	Initial design & development			
fluids	Exasim	Exasim + Tiramisu; Julia interface	Exasim + Enzyme; Integration with Mutation++	Performance optimization: chemistry on GPUs	Surface chemistry/erosion models
UQ	CESMIX-UQ	Bayesian potential fitting (PotentialUQ.jl)	UQ for MD simulations Initial active learning workflows	UQ + multi-fidelity for DFT Parametric UQ for Exasim	Coupled UQ workflows (Y4: only molecular; Y5: including fluids)

Integrating atomistic software…

❖ **Many computational tools** under development dedicated to **chemistry** and **materials science**

❖ **Integration is a must!**

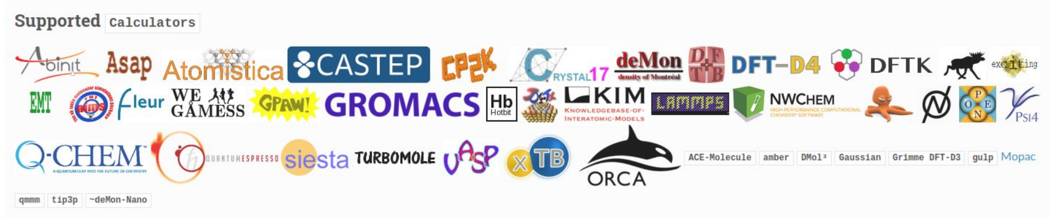
- ❖ ***Broadens*** *the spectrum of possible workflows*
- ❖ ***Manage*** *problems arising from the* ***interactions of components***
  - *E.g.* ***competition for hardware*** *resources*
- ❖ ***Catalyze advances*** *in chemistry and materials science*



❖ But it is also a **challenge**…

- ❖ *Software development is becoming more and more accessible*
  - *Increase in the number of tools*
  - *Increase in complexity associated with* ***overlapping functionality***
- ❖ *Horizontal hierarchies and large communities make it* ***difficult to reach consensus***

❖ Current integration efforts… ASE (Atomic Simulation Environment)



Supported `calculators`

`ACE-Molecule` `amber` `DMol³` `Gaussian` `Grimme DFT-D3` `gulp` Mopac

`qmmm` `tip3p` `~deMon-Nano`

julia *can help!*

- ***Fast** & **easy** :-D*
    - *Addresses the **two language problem**: Prototype algorithms in a user-friendly language, and then have to rewrite them in a faster language.*
    - *More **control** over your work*
    - ***Abstracts performance** management as much as possible*
    - *Avoid duplication of effort*
    - *More time spent on your research rather than on performance*
    - *Optimized code is easy to read*
    - *Collaboration and integration are easier*

- Software **composability** through multiple dispatch
    - *Accelerate the development of **atomistic workflows**.*
    - *Accelerate the development of **machine learning interatomic potentials** defined by the composition of neural networks with state-of-the-art interatomic potential descriptors.*
    - *Allows it to express **object-oriented** and **functional** programming patterns*

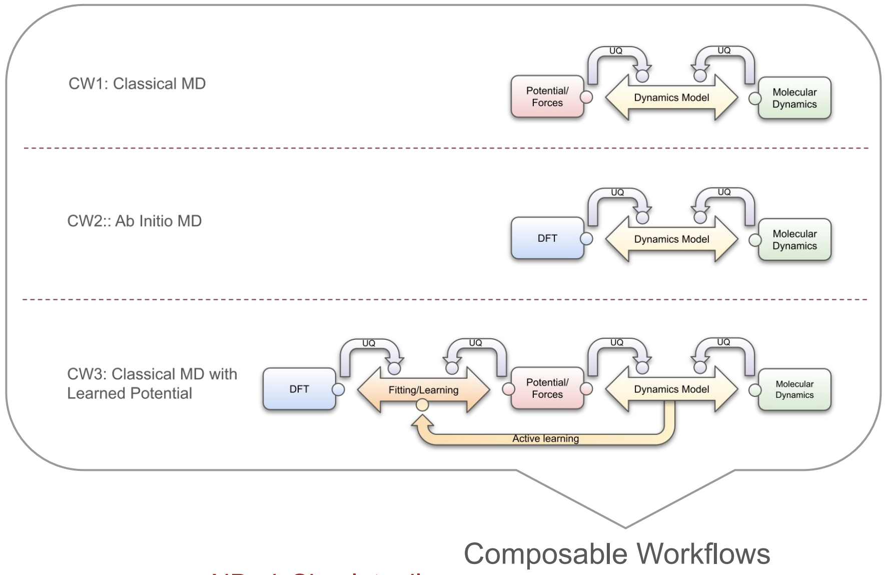
- SciML
    - *Open Source Software for Scientific Machine Learning.*
    - *E.g. **Optimization.jl**: brings together several **optimization packages into one unified Julia interface**.*

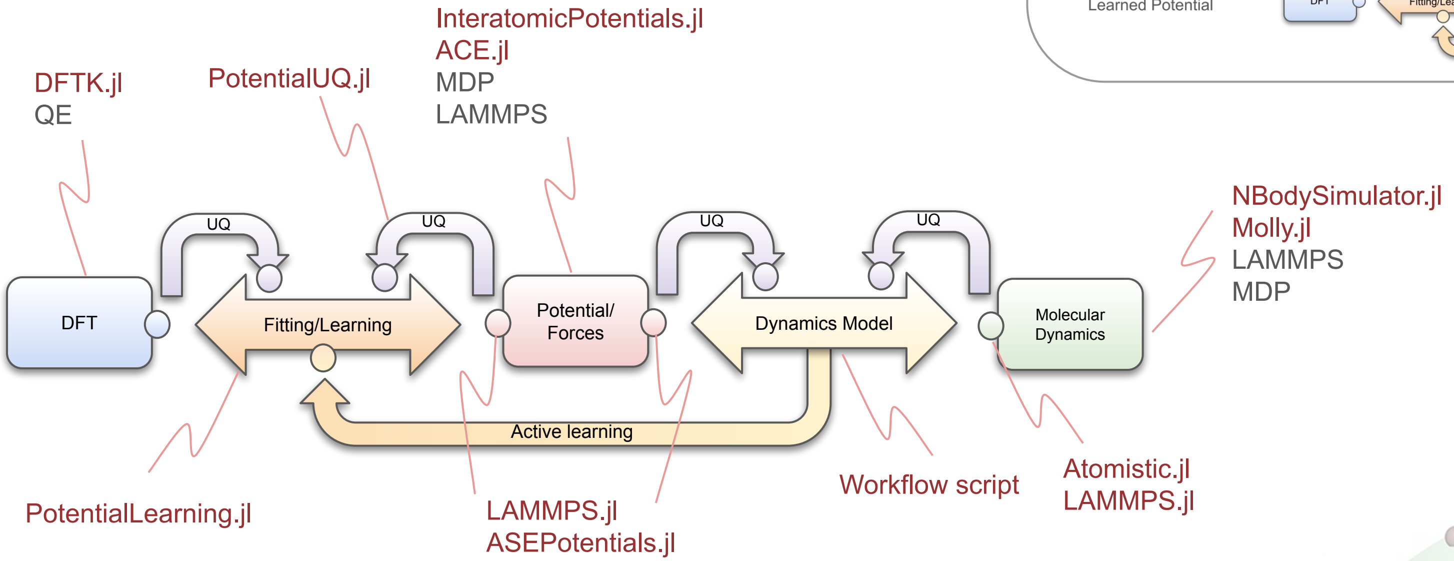
- **Automatic differentiation**
    - *Accelerate development by automatically calculating **loss function gradients, forces, sensitivity of QoI,** etc.*
    - *Allows to calculate **gradients across multiple languages** and **simulation codes**.*
    - *Powerful tools, such as **Enzyme.jl** or **Zygote.jl.***

- **HPC** and **Machine learning** abstractions
    - *Accelerate the development and execution of **machine learning interatomic potentials. Flux.jl** makes parallel learning simple using the NVIDIA GPU abstractions of **CUDA.jl**.*

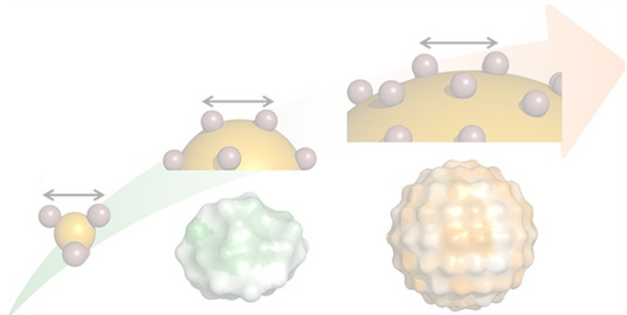
- More!
    - *Dynamically typed, but with support for **optional type declarations**.*
    - ***Feels like a scripting language**, but **can be compiled to efficient native code** for multiple platforms via LLVM.*
    - *…*

Integrating atomistic software in CESMIX: Composable Workflows





Only if needed: ○ ○ ○ ○ ○

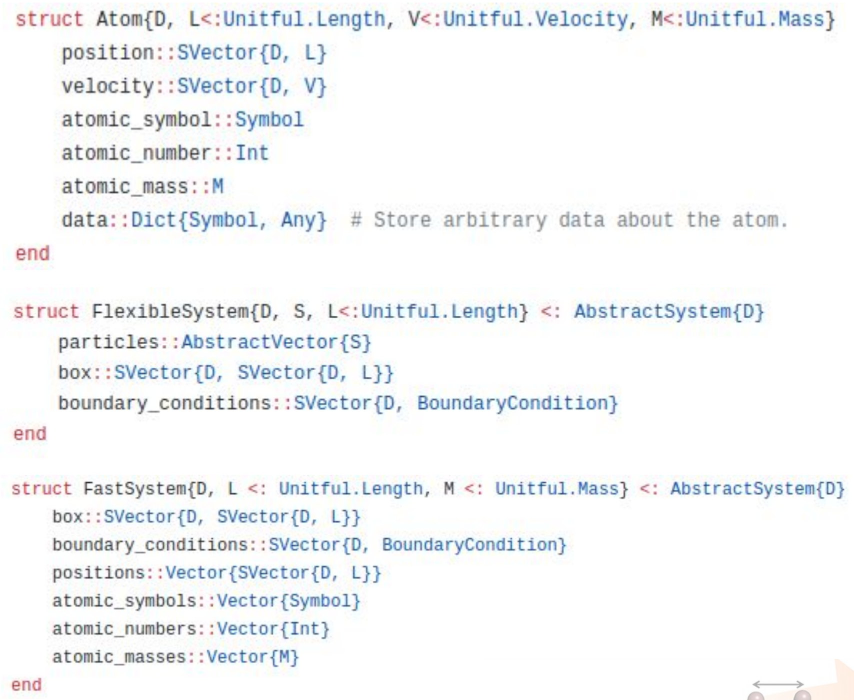


AtomsBase.jl

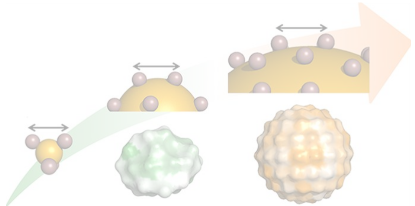
❖ AtomsBase.jl: *abstract interface for representation of* ***atomic geometries in Julia****. It aims to be a lightweight means of facilitating interoperability between various tools including…*

- ❖ *Chemical simulation engines. E.g. DFT, MD, etc.*
- ❖ *File I/O with standard formats (.cif, .xyz, ...)*
- ❖ *Numerical tools: sampling, integration schemes, etc.*
- ❖ *Automatic differentiation and ML systems*
- ❖ *Visualization (e.g. plot recipes)*

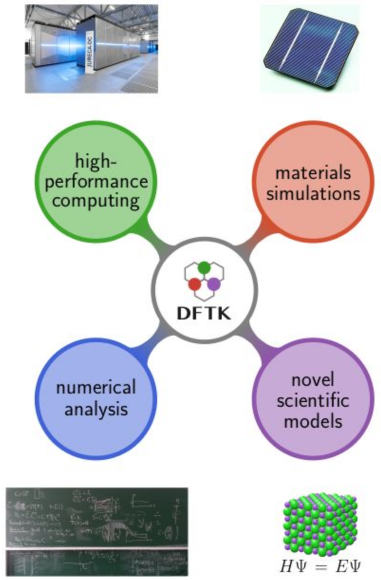
❖ AtomIO.jl*: standard* ***IO*** *package for atomic structures integrating with FileIO, AtomsBase, and others.*



```julia
struct Atom{D, L<:Unitful.Length, V<:Unitful.Velocity, M<:Unitful.Mass}
    position::SVector{D, L}
    velocity::SVector{D, V}
    atomic_symbol::Symbol
    atomic_number::Int
    atomic_mass::M
    data::Dict{Symbol, Any}  # Store arbitrary data about the atom.
end

struct FlexibleSystem{D, S, L<:Unitful.Length} <: AbstractSystem{D}
    particles::AbstractVector{S}
    box::SVector{D, SVector{D, L}}
    boundary_conditions::SVector{D, BoundaryCondition}
end

struct FastSystem{D, L <: Unitful.Length, M <: Unitful.Mass} <: AbstractSystem{D}
    box::SVector{D, SVector{D, L}}
    boundary_conditions::SVector{D, BoundaryCondition}
    positions::Vector{SVector{D, L}}
    atomic_symbols::Vector{Symbol}
    atomic_numbers::Vector{Int}
    atomic_masses::Vector{M}
end
```



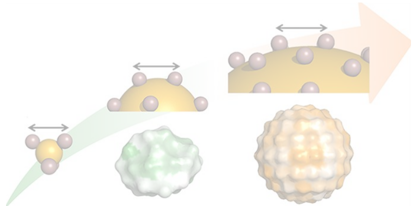
DFTK.jl: Density-functional toolkit



- julia code for plane-wave DFT, started in 2019
- Fully composable with julia ecosystem:
  - Algorithmic differentiation (AD)
  - Numerical error control
  - GPU acceleration (GSoC confirmed) planned
  - Uncertainty quantification (UQ) planned
- Key tool across domains and their use cases:
  - Mathematical analysis (reduced models)
  - Scale-up to applications ($\simeq 1000$ electrons)
  - Features incl. meta-GGA, response, MPI parallelisation
  - Speed within factor 2–4 to established codes

$\Rightarrow$ Build to enable multidisciplinary synergies

- Low entrance barrier across backgrounds:
  - Only 7000 lines of code, open-source components
  - Avoids two-language problem: Just julia
- High-productivity research framework:
  - GSoC student (10 weeks) for initial AD support



```julia
using AtomsBase, Unitful, UnitfulAtomic
# Define an atomic system
element = :Ar
atom1     = Atom(element, ( @SVector [1.0, 0.0, 0.0] ) * 1u"Å")
atom2    = Atom(element, ( @SVector [1.0, 0.25, 0.0] ) * 1u"Å")
box = [[1., 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]] * 1u"Å"
bcs = [DirichletZero(), Periodic(), Periodic()]
system   = FlexibleSystem(atoms, box , bcs)
```

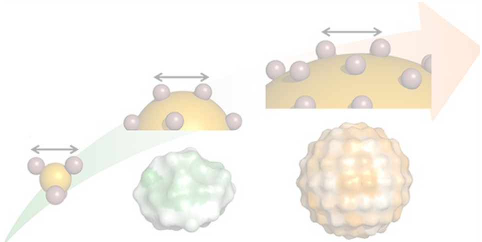
```julia
ϵ = 1.0 * 1u"eV"
σ = 0.25 * 1u"Å"
rcutoff  = 2.25 * 1u"Å"
lj       = LennardJones(ϵ, σ, rcutoff, [element])
```

```julia
pe       = potential_energy(system, lj)
f        = force(system, lj)
v        = virial(system, lj)
v_tensor = virial_stress(system, lj)
```

```julia
EmpiricalPotential <: AbstractPotential
BornMayer <: EmpiricalPotential
LennardJones <: EmpiricalPotential
Coulomb      <: EmpiricalPotential
ZBL          <: EmpiricalPotential

LinearCombinationPotential <: MixedPotential

# See InteratomicBasisPotentials.jl
BasisPotential <: AbstractPotential
SNAP           <: BasisPotential
ACE            <: BasisPotential
```

InteratomicPotentials.jl

- General goals
  - *Manipulate interatomic potentials in Julia. Allows to compute **energies, forces,** and **virial tensors** for a variety of interatomic potentials.*
- InteratomicPotentials.jl/InteratomicBasisPotentials.jl
  - *Lennard Jones, Born Mayer, Coulomb, ZBL, Morse.*
  - ***SNAP: new Julia implementation!***
  - ***ACE**: interface to ACE1.jl*
- Currently working on…
  - *Adding additional potentials*
    - ***Potentials for molecules**, EAM, 3-, 4-body potentials*
    - *Composition of **neural networks with IAP descriptors***
  - *Efficient **neighbor lists***
  - ***GPU** parallelization*
    - *Generic GPU kernels for potentials*
    - *SNAP, ACE*
- Main dependencies within this project
  - *AtomsBase.jl*



Create your own potential...

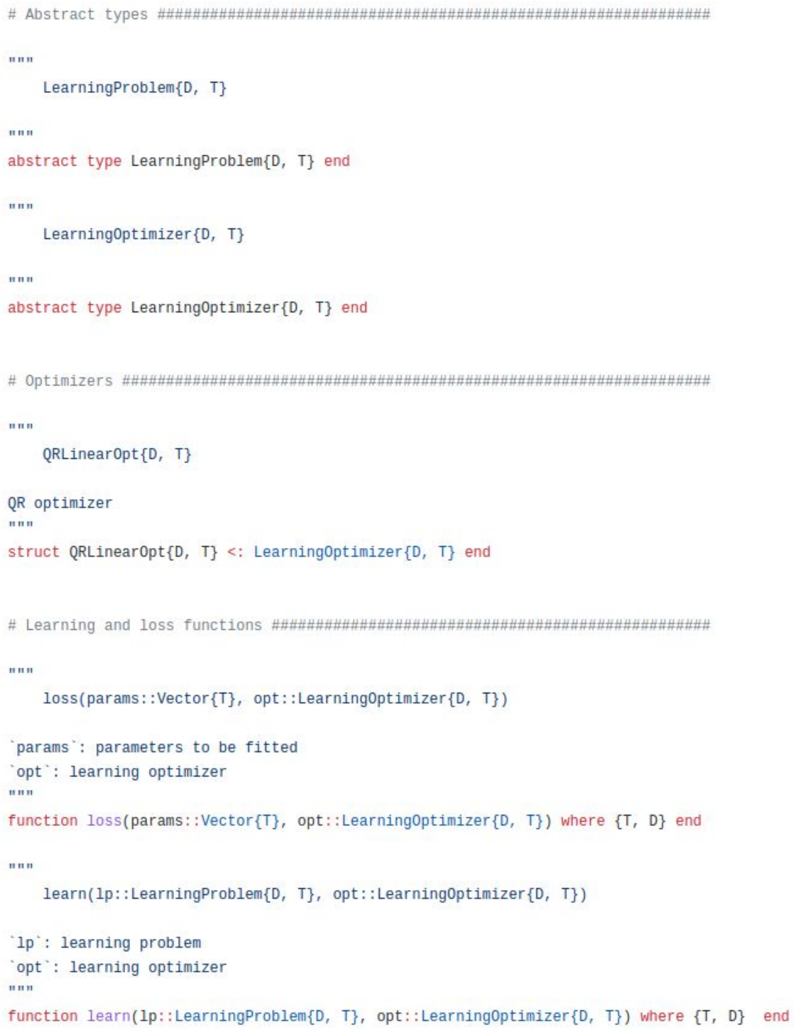
- Composability: *high level software that is readable, flexible, and* ***extensible!***

- Leverage multiple dispatch: add functionality to external software without modifying it
  - ***Define new types on which existing functions can be applied***
  - ***Define new functions which can be applied on existing types***

```julia
struct SimpleNNPotential <: AbstractPotential
   nn
   nn_params
end

function potential_energy(R::AbstractFloat,
                          p::SimpleNNPotential)
   return p.nn(R)
end

function force(R::AbstractFloat,
               r::SVector,
               p::SimpleNNPotential)
   dnndr = first(gradient((x)->p.nn(norm(x)), r))
   return -dnndr
end
```

```julia
model = Chain(Dense(1,32,Flux.relu),
              Dense(32,32,Flux.relu), Dense(32,1))
nn(R) = first(model([R]))
nn_params = Flux.params(model)

p = SimpleNNPotential(nn, nn_params)

potential_energy(1.0, p)

r_diff = SVector(1.0, 1.0, 1.0)
force(norm(r_diff), r_diff, p)
```

PotentialLearning.jl



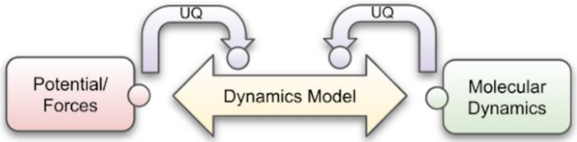
```
# Abstract types ################################################################

"""
    LearningProblem{D, T}

"""
abstract type LearningProblem{D, T} end

"""
    LearningOptimizer{D, T}

"""
abstract type LearningOptimizer{D, T} end


# Optimizers ####################################################################

"""
    QRLinearOpt{D, T}

QR optimizer
"""
struct QRLinearOpt{D, T} <: LearningOptimizer{D, T} end


# Learning and loss functions ##################################################

"""
    loss(params::Vector{T}, opt::LearningOptimizer{D, T})

`params`: parameters to be fitted
`opt`: learning optimizer
"""
function loss(params::Vector{T}, opt::LearningOptimizer{D, T}) where {T, D} end

"""
    learn(lp::LearningProblem{D, T}, opt::LearningOptimizer{D, T})

`lp`: learning problem
`opt`: learning optimizer
"""
function learn(lp::LearningProblem{D, T}, opt::LearningOptimizer{D, T}) where {T, D}  end
```

- General goals
  - *Implement an open source Julia library for* ***active learning*** *of interatomic potentials in atomistic simulations of materials.*
  - *Incorporates elements of* ***bayesian inference****,* ***machine learning****,* ***differentiable programming****,* ***software composability****, and* ***high-performance computing****.*

- Specific goals
  - *Intelligent* ***data subsampling****. Via DPP, clustering.*
  - *Inference of the* ***optimal values*** *and* ***uncertainties*** *of the model parameters, to propagate them through the atomistic simulation.*
    - *Interatomic potential* ***hyper-parameter*** *optimization.*
    - *Interatomic* ***potential fitting****.*

      The potentials addressed in this package are defined in InteratomicPotentials.jl and InteratomicBasisPotentials.jl.
  - *Measurement of QoI* ***sensitivity*** *to individual parameters.*
  - ***Input data*** *management and* ***post-processing****.*

- Main dependencies within this project
  - *AtomsBase.jl, InteratomicPotentials.jl, InteratomicBasisPotentials.jl, Carom.jl, Turing.jl, Enzyme.jl.*



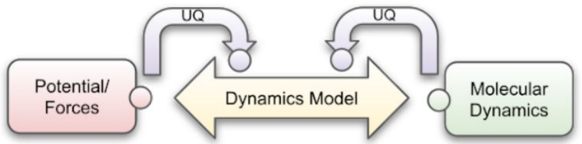
Classical MD

```julia
using Atomistic
using InteratomicPotentials
using NBodySimulator

N = 864
element = :Ar
box_size = 3.47786u"nm"
reference_temp = 94.4u"K"
thermostat_prob = 0.1 # this number was chosen arbitrarily
Δt = 1e-2u"ps"

initial_system = generate_atoms_in_cubic_cell(N, element, box_size, reference_temp)

eq_steps = 2000
eq_thermostat = NBodySimulator.AndersenThermostat(austrip(reference_temp),
                                               thermostat_prob / austrip(Δt))
eq_simulator = NBSimulator(Δt, eq_steps, thermostat = eq_thermostat)
potential = InteratomicPotentials.LennardJones(austrip(1.657e-21u"J"),
                                            austrip(0.34u"nm"),
                                            austrip(0.765u"nm"), [:Ar])

eq_result = simulate(initial_system, eq_simulator, potential)

prod_steps = 5000
prod_simulator = NBSimulator(Δt, prod_steps, t₀ = get_time(eq_result))

prod_result = simulate(get_system(eq_result), prod_simulator, potential)
```

Atomistic.jl

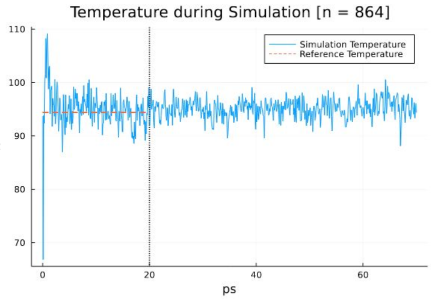
- ❖ General goal: *provide an integrated workflow for MD simulations.*
- ❖ Interface to
  - ❖ *Molly.jl*
  - ❖ *NBodySimulator.jl*
- ❖ Main dependencies within this project
  - ❖ *AtomsBase.jl, InteratomicPotentials.jl, and InteratomicBasisPotentials.jl*

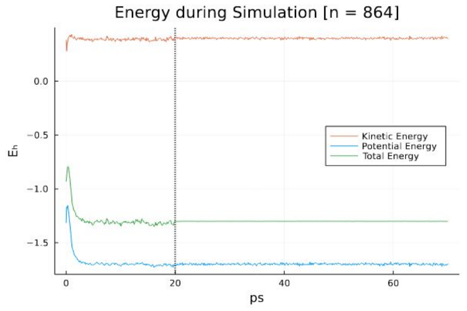
Atomistic.jl

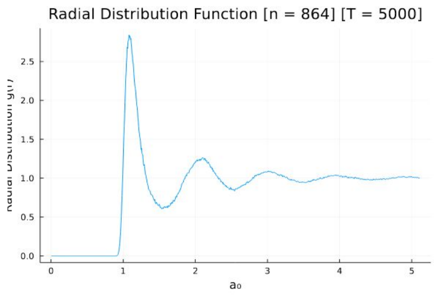


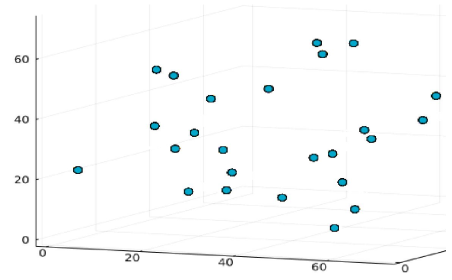
Classical MD

```
using Atomistic
using InteratomicPotentials
using Molly

N = 864
element = :Ar
box_size = 3.47786u"nm"
reference_temp = 94.4u"K"
coupling_factor = 10 # this number was chosen arbitrarily
Δt = 1e-2u"ps"

initial_system = generate_atoms_in_cubic_cell(N, element, box_size, reference_temp)

eq_steps = 2000
eq_thermostat = Molly.AndersenThermostat(reference_temp, Δt * coupling_factor)
eq_simulator = MollySimulator(Δt, eq_steps, coupling = eq_thermostat)
potential = InteratomicPotentials.LennardJones(austrip(1.657e-21u"J"),
                                               austrip(0.34u"nm"),
                                               austrip(0.765u"nm"), [:Ar])

eq_result = simulate(initial_system, eq_simulator, potential)

prod_steps = 5000
prod_simulator = MollySimulator(Δt, prod_steps, t₀ = get_time(eq_result))

prod_result = simulate(get_system(eq_result), prod_simulator, potential)
```

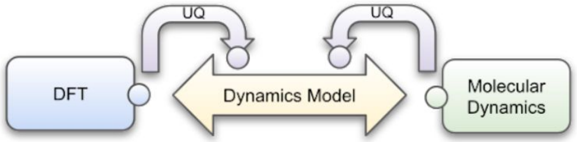








Atomistic.jl



Ab Initio MD

```julia
using Atomistic
using DFTK
using InteratomicPotentials
using NBodySimulator

setup_threading(n_blas = 4)

N = 8
element = :Ar
box_size = 1.5u"nm" # this number was chosen arbitrarily
reference_temp = 94.4u"K"
thermostat_prob = 0.1 # this number was chosen arbitrarily
Δt = 1e-2u"ps"

initial_system = generate_atoms_in_cubic_cell(N, element, box_size, reference_temp)
pspkey = list_psp(:Ar, functional = "lda")[1].identifier
for atom ∈ initial_system
    atom.data[:pseudopotential] = pspkey
end

eq_steps = 20000
eq_thermostat = NBodySimulator.AndersenThermostat(austrip(reference_temp),
                                                  thermostat_prob / austrip(Δt))
eq_simulator = NBSimulator(Δt, eq_steps, thermostat = eq_thermostat)
potential = InteratomicPotentials.LennardJones(austrip(1.657e-21u"J"),
                                               austrip(0.34u"nm"),
                                               austrip(0.765u"nm"), [:Ar])

eq_result = simulate(initial_system, eq_simulator, potential)

ab_initio_steps = 200
ab_initio_simulator = NBSimulator(Δt, ab_initio_steps, t₀ = get_time(eq_result))
dftk_potential = DFTKPotential(5u"hartree", [1, 1, 1]; damping = 0.7)

ab_initio_result = simulate(get_system(eq_result), ab_initio_simulator, dftk_potential)
```

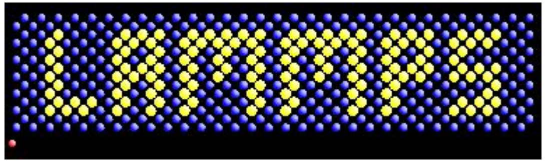
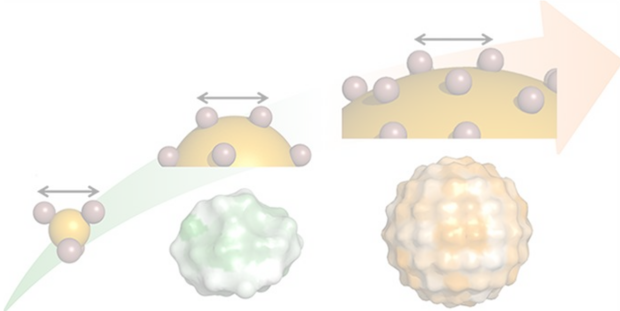
```julia
using Atomistic
using DFTK
using InteratomicPotentials
using Molly

setup_threading(n_blas = 4)

N = 8
element = :Ar
box_size = 1.5u"nm" # this number was chosen arbitrarily
reference_temp = 94.4u"K"
coupling_factor = 10 # this number was chosen arbitrarily
Δt = 1e-2u"ps"

initial_system = generate_atoms_in_cubic_cell(N, element, box_size, reference_temp)
pspkey = list_psp(:Ar, functional = "lda")[1].identifier
for atom ∈ initial_system
    atom.data[:pseudopotential] = pspkey
end

eq_steps = 20000
eq_thermostat = Molly.AndersenThermostat(reference_temp, Δt * coupling_factor)
eq_simulator = MollySimulator(Δt, eq_steps, coupling = eq_thermostat)
potential = InteratomicPotentials.LennardJones(austrip(1.657e-21u"J"),
                                               austrip(0.34u"nm"),
                                               austrip(0.765u"nm"), [:Ar])

eq_result = simulate(initial_system, eq_simulator, potential)

ab_initio_steps = 200
ab_initio_simulator = MollySimulator(Δt, ab_initio_steps, t₀ = get_time(eq_result))
dftk_potential = DFTKPotential(5u"hartree", [1, 1, 1]; damping = 0.7)

ab_initio_result = simulate(get_system(eq_result), ab_initio_simulator, dftk_potential)
```

LAMMPS.jl

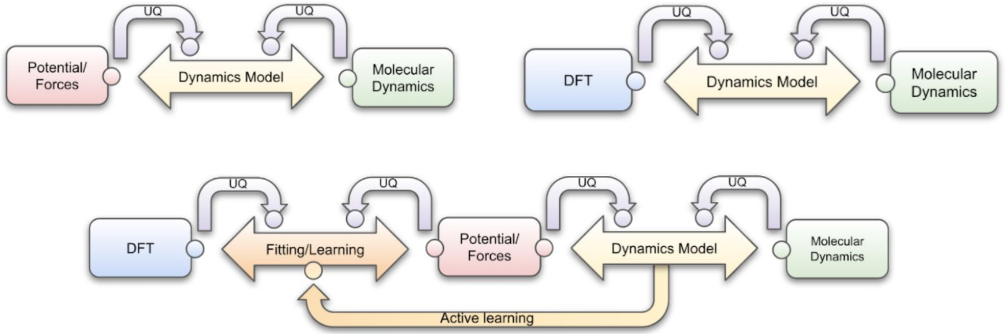
- *Goal*
  - Allow **direct interaction with LAMMPS** without round-tripping through Python

- *Achievements*
  - Precompiled binary available through LAMMPS_jll
    - A binary released as a Julia package that the package manager can install
    - Transparently loading dependencies and makes libraries available
  - Automatic conversion to Julia arrays (zero-cost access)
  - API coverage sufficient for simple SNAP example

- Next steps
  - **Integration with MPI.jl**
  - **Calling Julia potential from LAMMPS**
  - Increasing API coverage
  - Spack integration for binary deployment on HPC systems
  - Documentation and more examples
  - Deeper integration than possible through the C-API

Atomistic Composable Workflows



- General goal
  - *Gather easy-to-use CESMIX-aligned case studies*
  - *Integrate the latest developments of CESMIX and the Julia atomistic ecosystem with state-of-the-art tools.*

CW	Type	DFT	UQ	Fitting/ Learning	Potential / Forces	Molecular Dynamics
1	Ar		✓	✓	InteratomicPotentials.jl → Lennard Jones / ACE	LAMMPS.jl → LAMMPS
1	Ar		✓	✓	InteratomicPotentials.jl → LennardJones / ACE	Atomistic.jl → Molly.jl
1	Na				LAMMPS.jl → EAM	LAMMPS.jl → LAMMPS
3	Na	DFTK.jl	✓	✓	InteratomicPotentials.jl → SNAP / ACE	LAMMPS.jl → LAMMPS
3	HfO_2 , a-HfO_2	QE		✓	InteratomicBasisPotentials.jl → ACE → ACE1.jl	Atomistic.jl → Molly.jl

Other CESMIX tools and future integration efforts

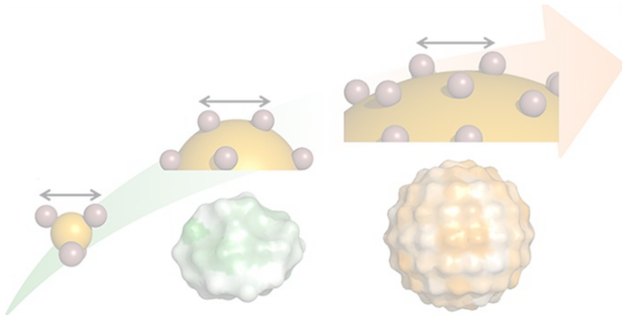
❖ *Enzyme.jl*
- *AD of statically analyzable LLVM*
- *Meet or exceed the performance of state-of-the-art AD tools.*

❖ *MLP.jl: ML potentials*
- *Fit and evaluate POD/SNAP/ACE potentials*
- *Generate and select configurations to perform DFT calculations to obtain DFT data for training potentials*
- *Develop NN and GNN potentials by leveraging Enzyme.jl for the differentiation of the loss function*

❖ *MDP.jl*
- *Run MD simulations for interatomic potentials that are fitted with MLP.jl*
- *Potentials will interact with InteratomicPotentials.jl*

❖ *KMC.jl*
- *Run Kinetic Monte Carlo in Julia.*
- *Will be built to interact with InteratomicPotentials.jl*

❖ *Carom.jl*
- *Interacting Particle Systems for Bayesian inference.*

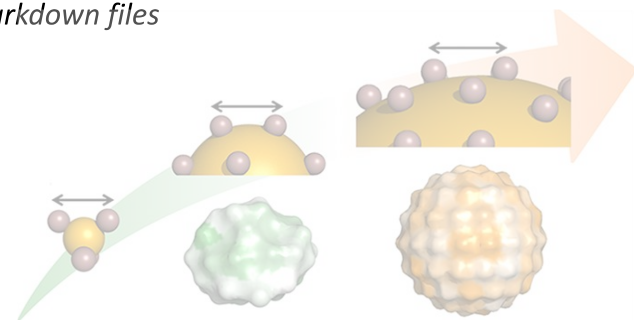


Software engineering practices

- **Composable Software**
  - *High level software that is readable, flexible, and extensible.*
  - *Deep integration: generic programming, abstractions, multiple dispatch.*

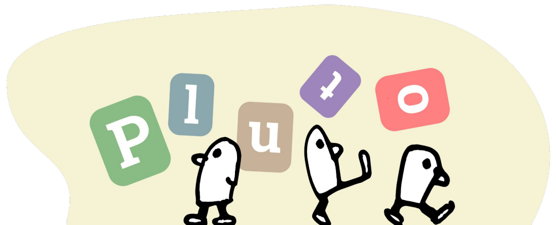
- Development platform: **Github**
  - *Internet **hosting***
  - *Version control: **Git***
    - *Used for coordinating work among programmers*
    - *Tracks and provides control over changes to source code*
    - *Speed, data integrity, and support for distributed, non-linear workflows (i.e., thousands of parallel branches running on different systems)*
  - ***Continuous integration** (CI)*
    - *Continuously build and test the code to make sure that the commit doesn't introduce errors*
  - ***Documentation***
    - *Documenter.jl: A package for building documentation from docstrings and markdown files*
  - ***Open Source***

- Other tools
  - *Visual Studio Code: Live Share*
  - *Pkg: Julia's package manager*
  - *Pluto notebooks: Julia notebooks*



Integrating using notebooks…

- *Reactive, lightweight, and easy Julia notebooks*
- *Easy entry point*
- *Programming directly from the **browser***
- *Condenses **documentation/tutorials**, **code** and **live results***
    - *Can be exported as html and used as a **blogpost***
- *It can be used as a **mini-laboratory** to experiment with modifying small parts of the code*
- *This feature is experimental: you can **run** the notebook **in the cloud**, so you do not need to install the software on your computer*

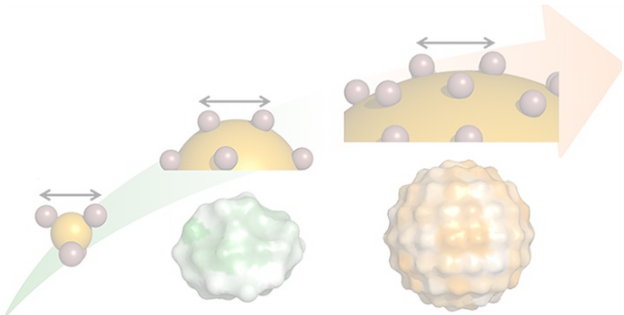


Animation: Pluto website

External software dependencies

- Current software dependencies
  - LAMMPS: large-scale atomic/molecular massively parallel simulator
  - QE: electronic-structure calculations and materials modeling
  - ACE.jl: atomic cluster expansion
  - Molly.jl: movement and interaction of molecules
  - NBodySimulator.jl: simulating systems of N interacting bodies
  - Unitful.jl: physical units
  - GalacticOptim.jl: global optimization package
  - Flux.jl, Lux.jl: ML frameworks
  - DataLoaders.jl: threaded data iterator for ML on out-of-memory datasets
  - BSON: binary JSON serialization format
  - CUDA.jl: NVIDIA GPUs support
  - Zygote.jl: AD
  - More!

- Potential future software dependencies
  - GraphNeuralNetwork.jl: matches DGL and PyTorch Geometric
  - FluxGeometric.jl: GNNs
  - Chemellia: ML ecosystem for systems made of atoms
  - Clustering.jl: data clustering
  - More!



Thanks!... questions?