KIM-based Learning Intergrated-Fitting Framework: *KLIFF*

Training, Deployment and Validation of MLIPs

Amit Gupta

April 6, 2023

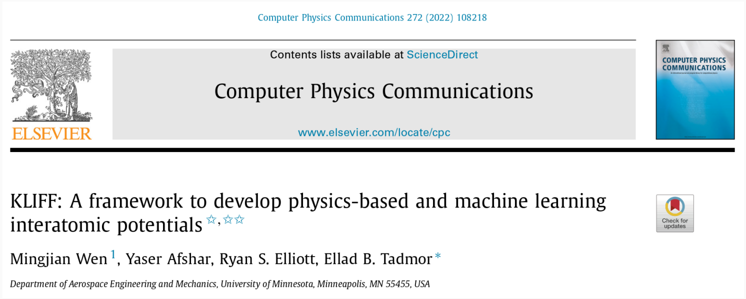
Dept. of Aerospace Engineering and Mechanics, UMN

So, we heard you have a model...

- Is it just parameter search?
- Is it some fancy ML model?
- Do you want to distribute it?
- Do you want it to be validated?
- Do you want it to be platform agnostic?
- Are there any uncertainties you are uncertain about?
- Is performance important?

If you answered yes to any of these questions, then let us help you!
`https://github.com/ipcamit/mach2023`

What it is?


Computer Physics Communications 272 (2022) 108218

Contents lists available at ScienceDirect

Computer Physics Communications

www.elsevier.com/locate/cpc

ELSEVIER

KLIFF: A framework to develop physics-based and machine learning interatomic potentials ☆,☆☆

Mingjian Wen [1], Yaser Afshar, Ryan S. Elliott, Ellad B. Tadmor *

Department of Aerospace Engineering and Mechanics, University of Minnesota, Minneapolis, MN 55455, USA

- Modern potential fitting framework
- From dataset to production models
- Flexible, easy to use, feature rich
- High performance, parallel

`https://github.com/openkim/kliff`[1,2]

[1] KLIFF: A framework to develop physics-based and machine learning interatomic potentials, Mingjian Wen et. al., Comp. Phys. Comm., 272, 2022

[2] Extending OpenKIM with an Uncertainty Quantification Toolkit for Molecular Modeling, Yonatan Kurniawan et. al., arXiv:2206.00578, 2022

What can it do?

```python
ds  = Dataset(colabfit_database="my_colabfit_database", colabfit_dataset="my_config_dataset")
kgg = KIMTorchGraphGenerator(cutoff=3.0, n_layers=3, as_torch_geometric_data=True)
dl = DataLoader(ds, batch_size=5, collate_fn=kgg.collate_fn); next(iter(dl))
```

```python
sf = Descriptor("SymmteryFunctions", cutoff=3.77, hyperparam="set51")
model = Sequential(Linear(51,10), ReLU(), Linear(10,10), Tanh(), Linear(10, 1))
tw = TrainingWheels(sf, model)
# Train your model here
#...
tw.export_kim_model("TorchMLModel3_Graph")
```

```
# Initialize KIM Model
kim init TorchMLModel3_Graph metal

# Load data and define atom type
read_data test_si.data
kim interactions Si
```

What is a model?

- Function mapping a set of coordinates to properties (energy, forces, stress*)
- Module: `kliff.models`, `torch.nn.Module`
- Conventional (physics based) and ML models
- Trainable and const parameters

Supported Models

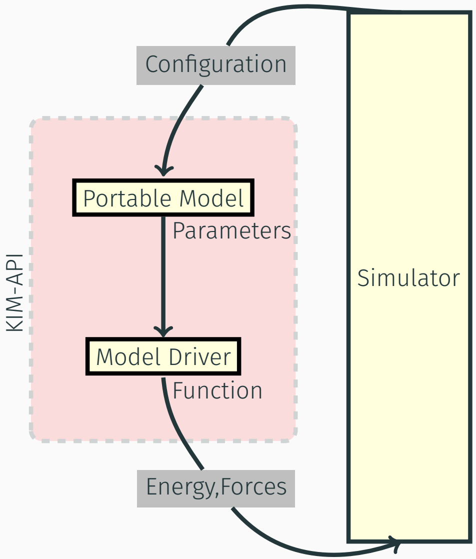
Physics Based

+ Analytical energy and gradients
+ Fixed functional form
+ High performance and transferable
- "Mix of art and science"
- More involved development (implement KIM model first)

Machine Learning Based

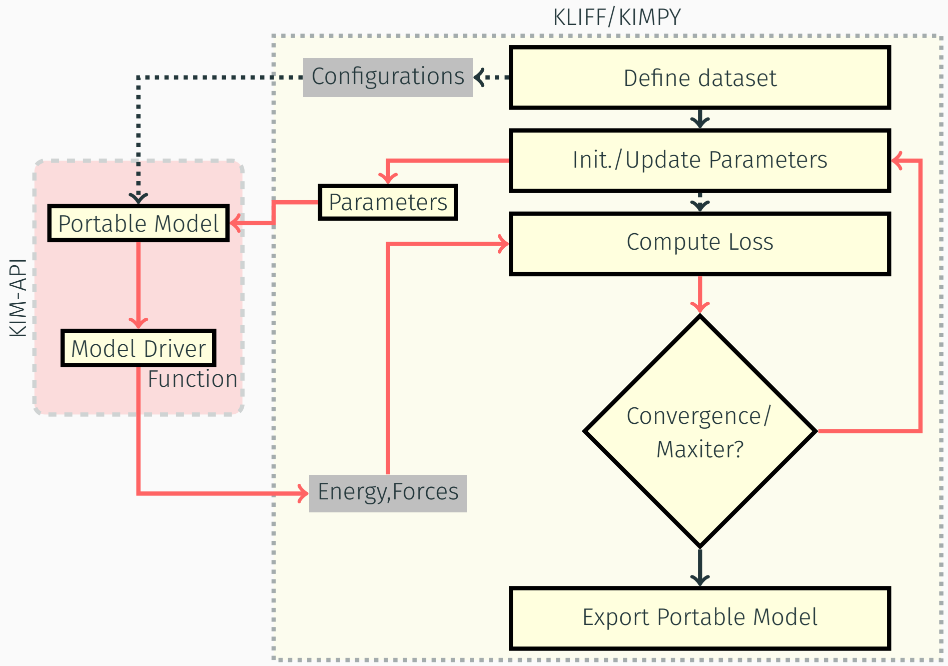
+ More flexible functional form
+ Easier to implement
* Higher accuracy, higher computation cost
- Difficult to deploy (convert NN ops to C++ ops)
- Non-transferable

Introduction: KIM-API



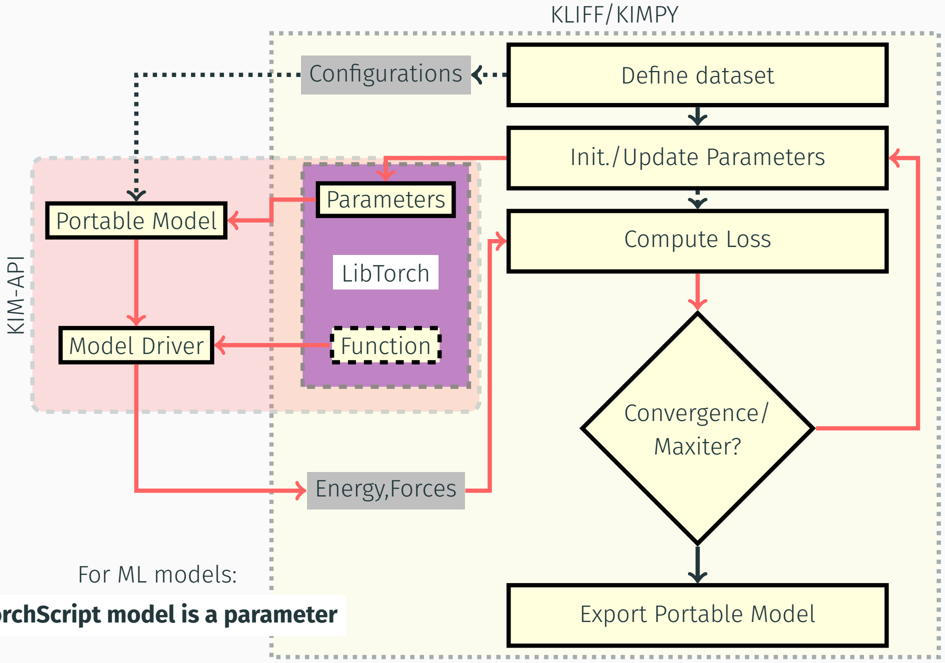
Simulator:	MD software, e.g. LAMMPS
Model Driver:	Executing code/ function
Portable Model:	Parameters for Model Driver

Introduction: KLIFF/KIMPY



Physics Based Model example

Introduction: KLIFF with ML Model Driver



For ML models:

TorchScript model is a parameter

Supported Models

Model is the self-contained function : KIM-API executes it.

Physics based models: All KIM-API portable models

Physics based models: Three kind of ML inputs supported

Model kind	Model signature
Generic	`model(Z, coords, n_neigh, nlist, contributing)`
Descriptors based models	`model(descriptor)`
Graph Neural Networks	`model(Z, coords, graph1, graph2, ... , contributing)`

Output:

Model kind	Model output
Self contained	`tuple(c10::tensor energy, c10::tensor force)`
Energy model	`c10::tensor energy`

Energy model: Model driver uses `torch::autograd::grad` to compute forces

Exporting

`TrainingWheels.export_kim_model("ModelName")` saves a portable model

Model depends on `TorchMLModelDriver`

Note:

Models should be exportable to TorchScript

Note:

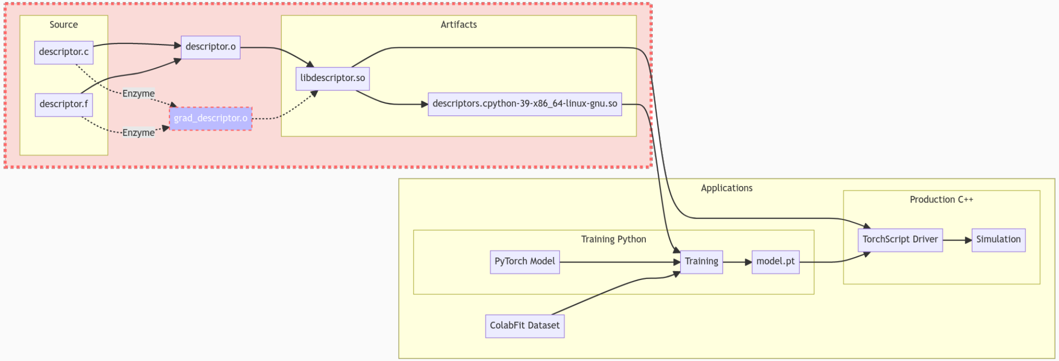
For GPU evaluation set `KIM_MODEL_EXECUTION_DEVICE` environment variable to `"cuda"`

ML Models

Preprocessing: Descriptors (libdescriptor)

Aim:

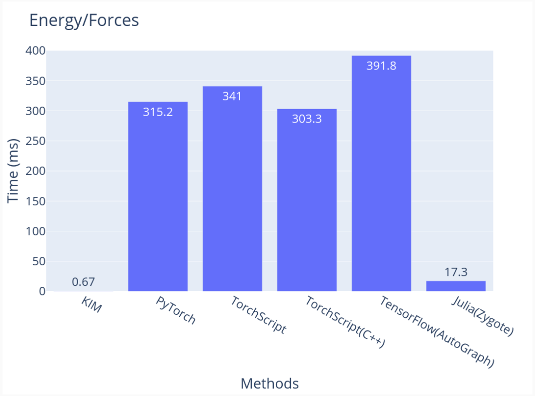
- High performance
- Easy to extend
- ML compatible (gradients)
- Easy to use (In training and production)

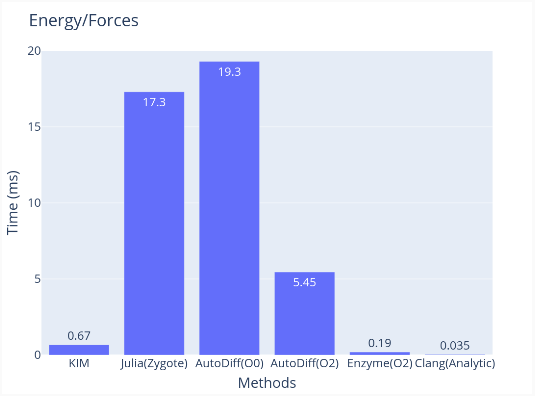


`libdescriptor`: Auto-differentiated C++ High-performance descriptor library [`github.com/ipcamit/libdescriptor/`]

Preprocessing: Descriptors

Stillinger-Weber Forces from AD





Enzyme generated gradients same performance order as analytical, Autodiff has high ease of use.

3

[3]* KIM called from ASE, so it incurs additional overhead, ** Scaled KIM:Enzyme time: 3.5 s

Preprocessing: Descriptors

C++

```cpp
#include <Descriptors.hpp>

string file_name = "descriptor.dat";
auto dbs = Descriptor::DescriptorKind::initDescriptor(file_name,Descriptor::KindSymmetryFunctions);
Descriptor::compute_single_atom(i, n_atoms, species, neighbors, n_neigh, coords, desc, dbs);
auto desc_tensor = to_tensor(desc);
auto energy = torchMLModel(desc_tensor);
energy.backward();
Descriptor::gradient_single_atom(i, n_atoms, species, neighbors, n_neigh, coords, forces,
                     desc, desc_tensor.grad().data_ptr<double>(), dbs);
```

Python

```python
import libdescriptor as lds
ds = lds.DescriptorKind.init_descriptor("descriptor.dat", lds.AvailableDescriptors.SymmetryFunctions)
desc = tensor(lds.compute_single_atom(ds, index, species, neig, coords))
energy = ml_model(desc)
energy.backward()
forces = lds.gradient_single_atom(ds, index, species, neig, coords, desc, desc.grad)
```

Extending:

1. Inherit `Descriptor::DescriptorKind` class
2. Implement `DescriptorKind::compute` (single atom compute, “forward” function)
3. Add to `switch` in `Descriptor::compute(_one_atom)` and `Descriptor::gradient(_one_atom)`

Preprocessing: Descriptors

… with goodness of AD

- + High performance
- + Gradients included
- + Easy to extended and maintain
- + Unified Python/C++ interface
- + Derivatives against parameters
- + Numerical derivatives checks
- - Enzyme is on v 0.0.42, might get flaky

Accepting requests and contributions!

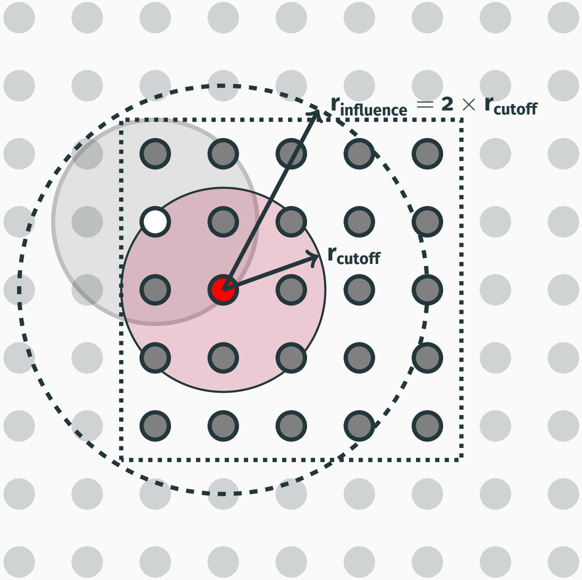
1. Behler Symmetry Function (Included)
2. Bispectrum (Included)
3. SOAP (WIP)
4. ACE (WIP)

Not limited to KLIFF or OpenKIM

Deploying ML Models

Preprocessing: Graphs

- Conventional graph structure: recursive with periodic distances (MIC)
- KIM design purely functional: no “global” crystal information
- Cutoff vs Influence distance
- Inherently compatible with arbitrary domain decomposition.
- Self-contained graphs
- Highly parallelizable and granular graph convolution



Graphs

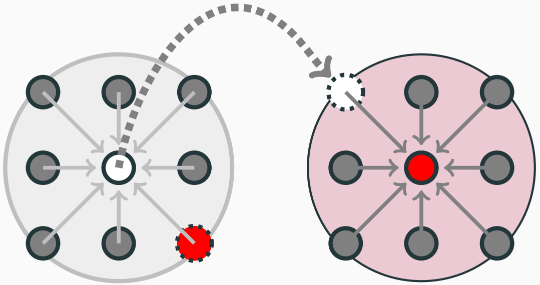
2 Conv Layers:

MIC

```
for i in range(n_conv):
    h = graph_conv(h, edge_graph)
```

Infl. distance

```
h = graph_conv_0(h, edge_graph0)
h = graph_conv_1(h, edge_graph1)
```



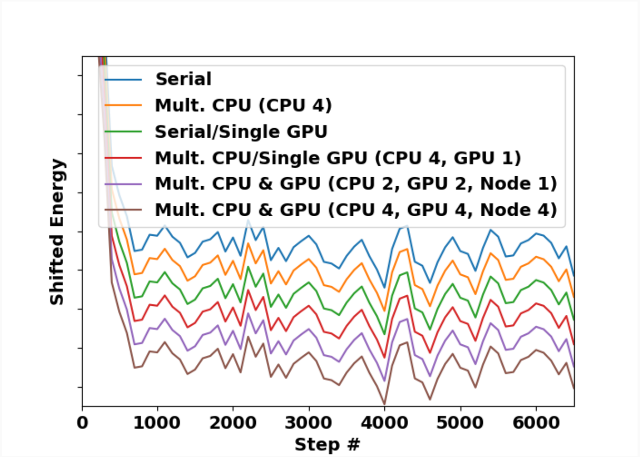
Graph Conv 0 Graph Conv 1

$(\sum |h_{MIC} - h_{Infl}|)_{\mathbb{R}^{10}}$

- Conv 0 : **3.8858** $\times$ **10^{-16}**
- Conv 1 : **6.3838** $\times$ **10^{-16}**
- Conv 2 : **1.2767** $\times$ **10^{-15}**

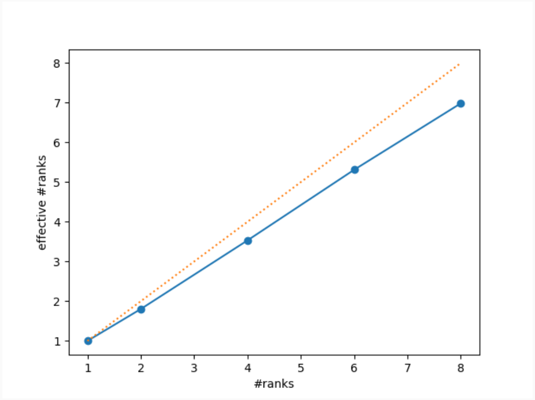
Why though?: Parallelization

```
# Initialize KIM Model, same name as the installed KLIFF model
kim init TorchMLModel3_Graph metal

# Load data and define atom type
read_data test_si.data
kim interactions Si

```

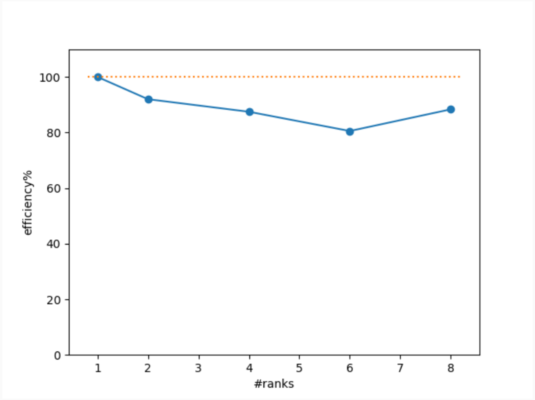


Parallelization

Strong scaling

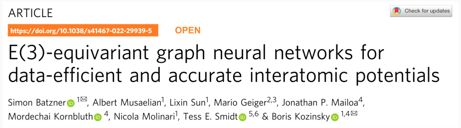


Weak scaling



NEQUIP Port example

Example: NEQUIP port


ARTICLE

https://doi.org/10.1038/s41467-022-29939-5 OPEN

Check for updates

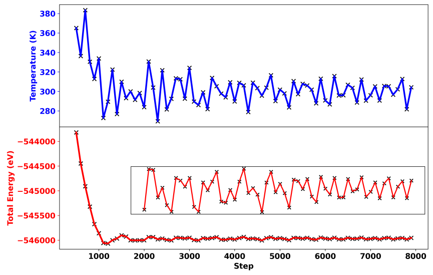
E(3)-equivariant graph neural networks for data-efficient and accurate interatomic potentials

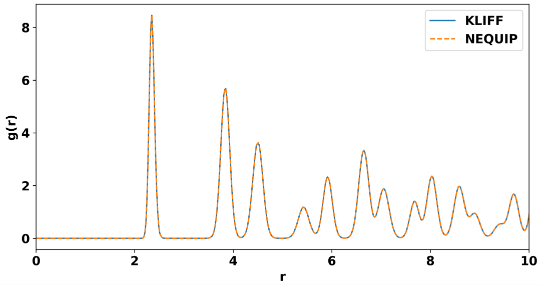
Simon Batzner[1✉], Albert Musaelian[1], Lixin Sun[1], Mario Geiger[2,3], Jonathan P. Mailoa[4], Mordechai Kornbluth[4], Nicola Molinari[1], Tess E. Smidt[5,6] & Boris Kozinsky[1,4✉]

```python
def forward(self, input: AtomicDataDict.Type):
    for module in self:
        input = module(input)
    return input
```

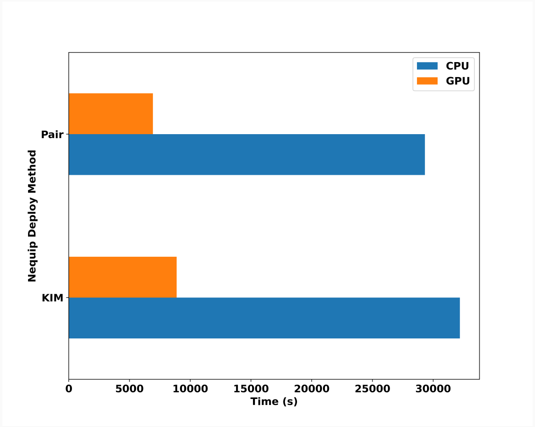
```python
def forward(self, x, pos, edge_index0, edge_index1, edge_index2,
  contributing):
    # Embedding
    x_embed = self[0](x)
    h = x_embed
    # Edge embeddings
    edge_vec0, edge_sh0 = self[1](pos, edge_index0)
    edge_vec1, edge_sh1 = self[1](pos, edge_index1)
    edge_vec2, edge_sh2 = self[1](pos, edge_index2)
    # Radial basis functions
    edge_lengths0, edge_length_embeddings0 = self[2](edge_vec0)
    edge_lengths1, edge_length_embeddings1 = self[2](edge_vec1)
    edge_lengths2, edge_length_embeddings2 = self[2](edge_vec2)
    # Atomwise linear node feature
    h = self[3](h)
    # Conv
    h = self[4](x_embed, h, edge_length_embeddings2, edge_sh2,
      edge_index2)
    h = self[5](x_embed, h, edge_length_embeddings1, edge_sh1,
      edge_index1)
    h = self[6](x_embed, h, edge_length_embeddings0, edge_sh0,
      edge_index0)
    # Atomwise linear node feature
    h = self[9](x, self[8](self[7](h)))[contributing==1]
    return h
```

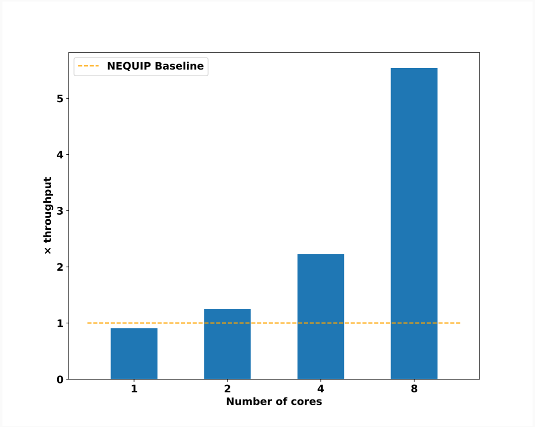
Example: NEQUIP port





Example: NEQUIP port





Fisher Analysis

- $J^T J$, using numerical differentiation
- Upper bound of uncertainity

Uncertainity Quantificaion

- Bayesian MCMC (parallel tempered affine-invariant, `ptemcee`)

```python
from kliff.uq import MCMC, get_T0
from multiprocessing import Pool

# Get the dimensionality of the problem, Number of parameters
ndim = calc.get_num_opt_params()
nwalkers = 2 * ndim # Number of parallel walkers

# Generate a temperature ladder
T0 = get_T0(loss)
Tladder = np.sort(np.append(np.logspace(0, 7, 15), T0)); ntemps = len(Tladder) # Number of temperatures

# Instantiate a sampler
sampler = MCMC(loss, nwalkers=nwalkers, logprior_args=(np.tile([-8, 8], (ndim, 1)),),
        Tladder=Tladder, random=np.random.RandomState(2022))
sampler.pool = Pool(processes=nwalkers)

# Initial starting points for each walker
p0 = np.random.uniform(low=-6.0, high=6.0, size=(ntemps, nwalkers, ndim))
sampler.run_mcmc(p0, 150000)
sampler.pool.close()
```

UQ example

Verification and Testing

Torch Model Driver makes ML models first-class citizens in OpenKIM

+ Easy to use
+ Easy to archive
+ Easy to test
+ Easy to verify

KIM Tests and Verification Checks: Cover the corner cases

VC example

Where to now?

Regular development tasks.

- Add more descriptors
- Performance tuning
- KIM Verification Checks and Tests

Moonshots (in order of ease and priority):

- More comprehensive UQ (Bootstrap UQ method[4], Experimental support for HMC)
- OMP parallelism in model driver
- Model driver for TensorFlow/JAX (both use HLO XLA backend)
- Repository of ML models

[4]See Yonatan Kurniawan's talk

Take home message

If your ML models ...

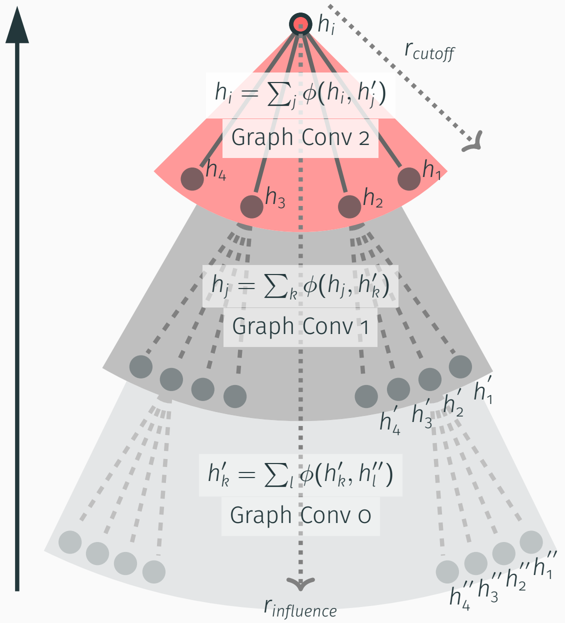
- Can be compiled to TorchScript
- Follows one of the three call signature:
    - `model(species, coords, n_neigh, nlist, contributing)`
    - `model(descriptor)`
    - `model(species, coords, graph1, graph2, ... , contributing)`
- Are a local model

then

- `TorchMLModelDriver` can run it out of the box with LAMMPS, ASE, and others
- KLIFF can be used to train it from scratch

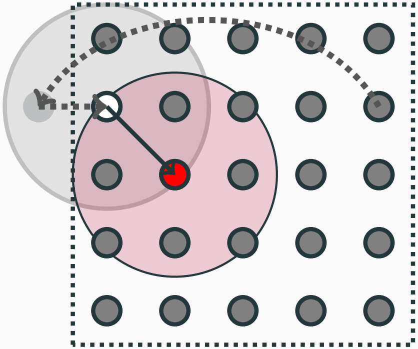
Thank you

Staged Convolution

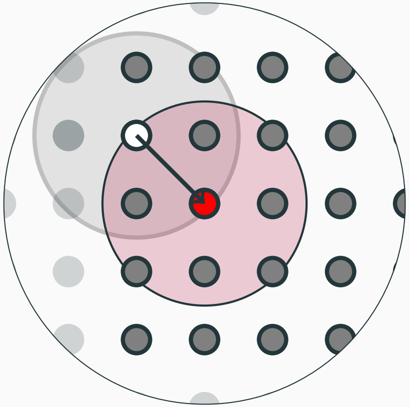


Graphs

Minimum Image

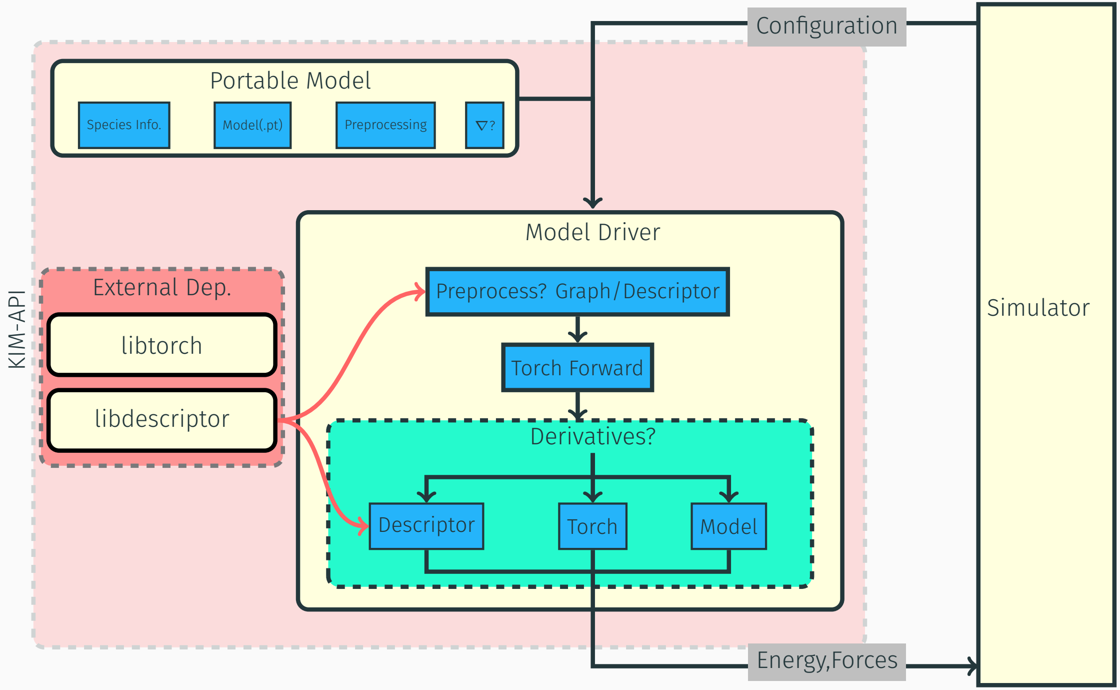


Unrolled influence distance



```
temp_r_ij = r_i - r_j
if all(temp_r_ij @ lattice_vectors < box_dims):
    r_ij = temp_r_ij
else:
    rij = temp_r_ij - mask( temp_r_ij @ lattice_vectors
       ) * lattice_vectors
    if distance(r_ij) < cutoff:
        h_i = phi_h(h_i, h_j)
```

```
r_ij = r_i - r_j
if (distance(r_ij) < cutoff:
   h_i = phi_h(h_i, h_j)

```



Configuration
Portable Model
Species Info.
Model(.pt)
Preprocessing
∇?
KIM-API
External Dep.
libtorch
libdescriptor
Model Driver
Preprocess? Graph/Descriptor
Torch Forward
Derivatives?
Descriptor
Torch
Model
Simulator
Energy,Forces

```cpp
#include <iostream>

// Enzyme arg kinds
int enzyme_dup, enzyme_const, enzyme_out;

// function to diff
double pow(int x, double y){
    double z = 1.0;
    for (int i = 0; i < x; i++){
        z *= y;
    }
    return z;
}

// declaration of diff
double __enzyme_autodiff_d_pow(double (*)(int , double) /* pointer to function to diff */ ,
                                int /* kind of arg */, int /* x */,
                                int /* kind of arg */, double /* y */);

int main(){
    int x = 3; double y = 4.0;
    // call to gradient
    double d_pow_y =  __enzyme_autodiff_d_pow(pow, enzyme_const, x, enzyme_out, y);
    std::cout << "Derivative: " << d_pow_y << "\n";
    return 0;
}
```

```
# Descriptor parameters SymFun
# n_species
1

# Full cutoff matrix n_species x n_species
3.77

# Cutoff function
cos

# descriptor width
51

# 3 body
True

# number of symmetry funcitons, their name, number of elements, values
2
g2
g4
16
129

0.001
0.0
0.01
0.0
0.02
0.0
0.035
0.0
0.06
0.0
0.1
0.0
```

```python
from kliff.models import KIMModel
from scipy.optimize import minimize
 ...
 # Torch DataLoader
dataloader = DataLoader(tset, batch_size=1, shuffle=True)

# Define KIM model
model = KIMModel(model_name="SW_StillingerWeber_1985_Si__MO_405512056662_005")
# Energy = model(configuration)
model.set_opt_params(A=[[10.0]], B=[[0.5]])
x0 = np.array([10.0, 0.5])

# Loss function over entire dataset
def loss(x0):
    loss = 0.0
    model.update_model_params(x0)
    for configuration in dataloader:
        E = model(configuration[0],compute_forces=False)["energy"]
        loss += (E - configuration[0].energy)**2
    return loss
# Update
result = minimize(loss, x0, method="Nelder-Mead", tol=1e-12, options={"maxiter":1000})

# Implicit method
opt_params = model.parameters()

opt = OptimizerScipy(model, opt_params, tset, optimizer="Nelder-Mead",
    target_property=["energy", "forces"],)
opt.minimize()
```

```python
# Descriptors
sf = Descriptors("SymmetryFunctions", cutoff={"Si-C": 7.0, "C-C": 5.0,"Si-Si": 6.0}, hyperparameters="set51")

# Define model and calculate energy
model = Sequential(Linear(51, 10), ReLU(), Linear(10, 10), Tanh(), Linear(10, 1))
tw = TrainingWheels(sf, model)

# Explicit loss and optimization
def loss_fn(ef_dict, target_energy, target_forces):
    loss = torch.sum((ef_dict["energy"] - target_energy) ** 2)
    loss = loss + torch.sum((ef_dict["forces"] - target_forces) ** 2)
    return loss

opt = torch.optim.Adam(tw.parameters, lr=lr)
for i in range(epochs):
    for conf in tset:
        ef_dict = tw(conf)
        loss = loss_fn(ef_dict, conf.energy, torch.from_numpy(conf.forces))
        opt.zero_grad()
        loss.backward()
        opt.step()

# Implicit method
optimizer = OptimizerTorch(tw, tw.parameters(), dataset,
                          optimizer=torch.optim.Adam(tw.parameters(),lr=0.01),
                          epochs=300)
```