

AI-based force field and MD simulations

Tong Wang

School of Life Sciences, Tsinghua University

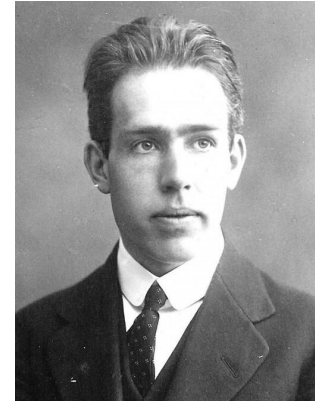
Personal information

Tong Wang (王童)

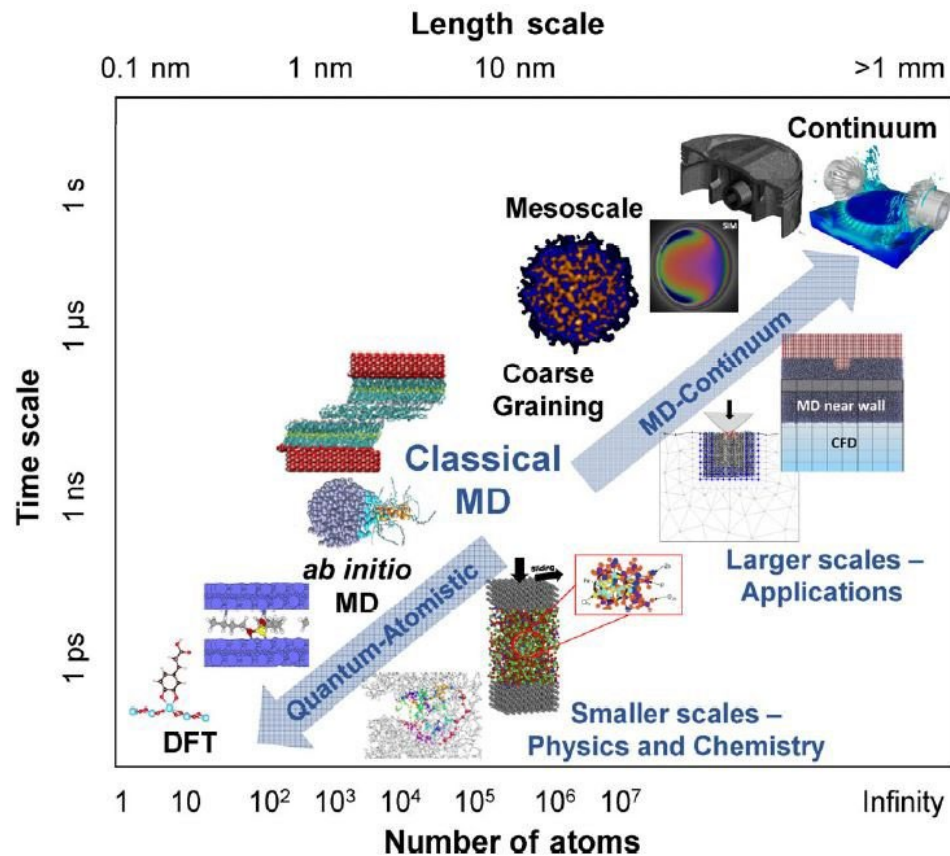
- Email: tongwang@mail.tsinghua.edu.cn
- Phone: 010-62794752
- Location: Biomedical Building A216A (office), A208 (lab)

Physicists understand the mathematics but not the world.

Niels Bohr

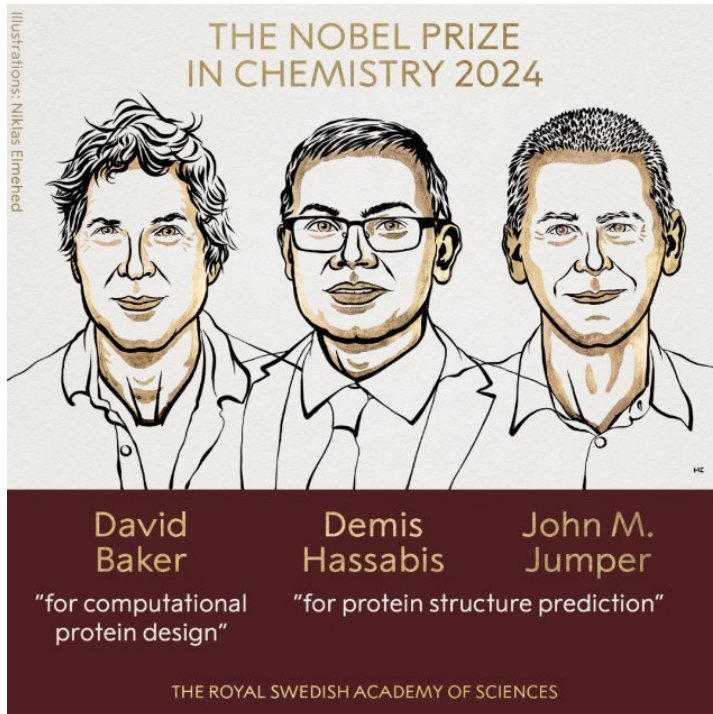


Accuracy vs. Efficiency in biomolecular simulation



- Molecular mechanics is fast but accuracy-limited
- Quantum mechanics is accurate but time-consuming

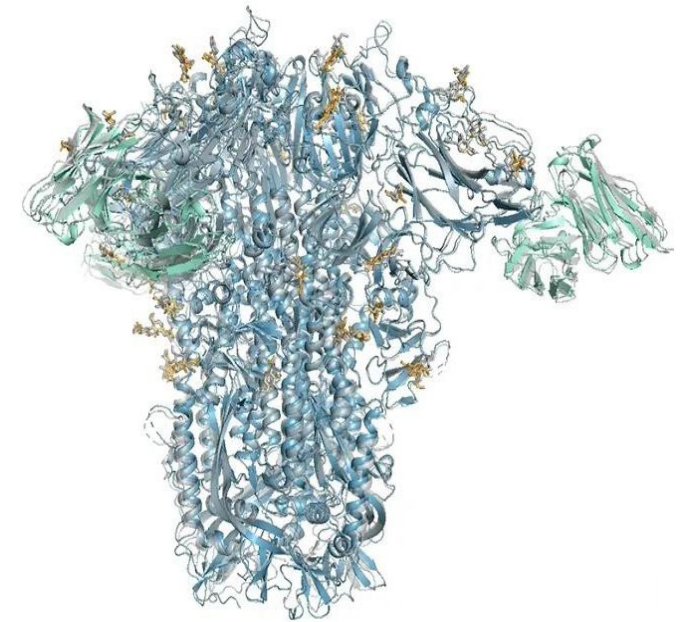
AI can make a difference



- AI driven protein structure prediction and design won Nobel Prize in Chemistry 2024.



- AlphaFold has predicted 214 million protein structures across more than 1 million species, covering nearly all known proteins.



- AI-driven protein structure prediction accelerates drug discovery and enzyme engineering design.

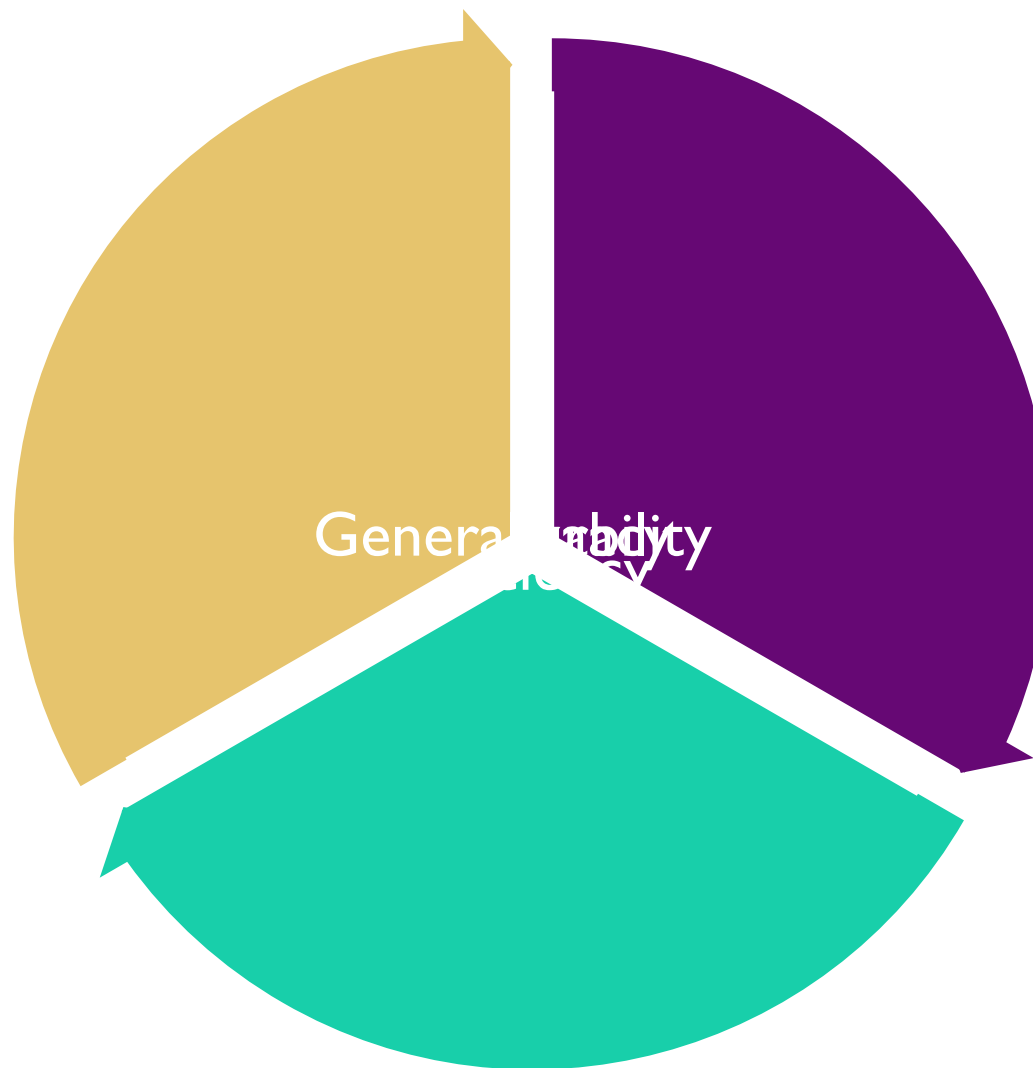
From static to dynamic: AI driven MD simulation



Key factors

- Generalized to different conformations of the same biomolecule
- Generalized to different biomolecules in the same kind
- Generalized to different kinds of biomolecules

- MM is fast and supports μs to ms simulations
- QM's time complexity is $O(N^3)$ to $O(N^7)$, cannot support biomolecular simulations



- MM has limited accuracy
- QM has ab initio accuracy
- AI driven method's accuracy approaches to ab initio (chemical accuracy)

- AI driven method's accuracy approaches to ab initio, supports ns to μs simulations

AI-driven MD simulations based on MLFF

- Parametrization of classical molecular force fields via machine learning
- End-to-end training machine learning force fields
- Universal MLFF for diverse biomolecular systems

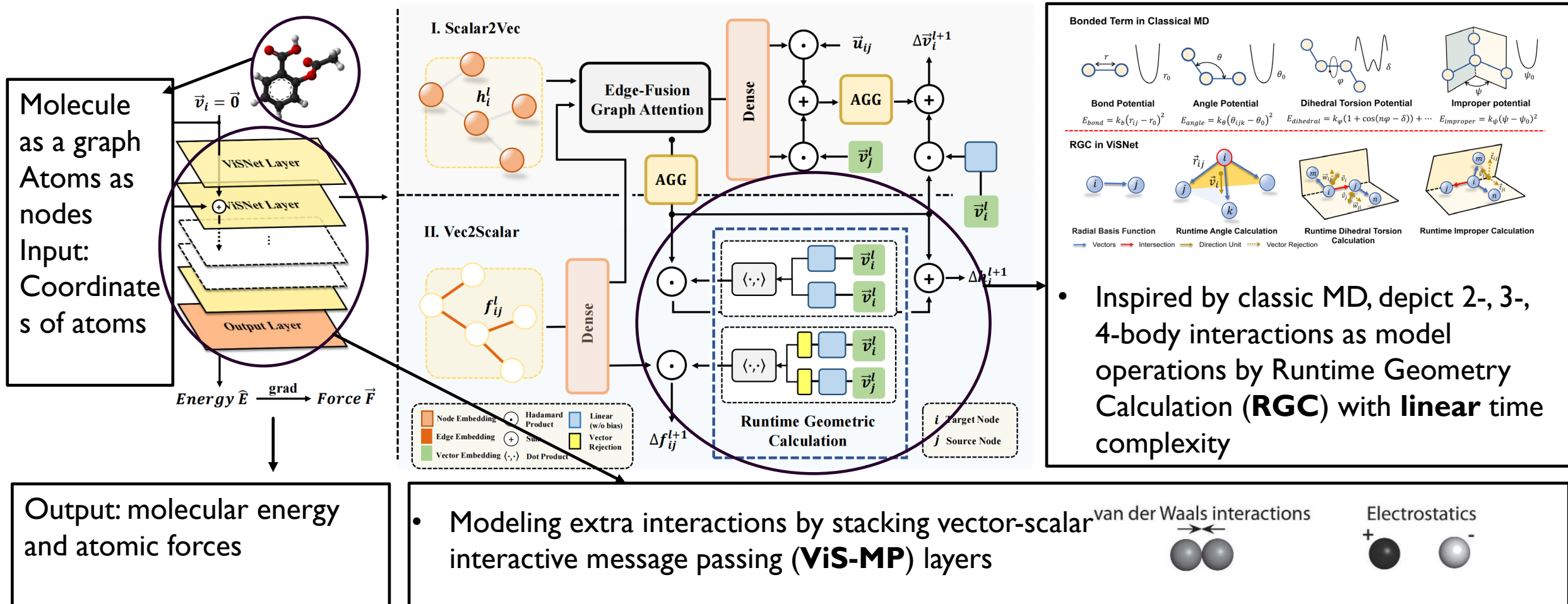
Parametrization of classical molecular force fields via ML

- Goal: predict the coefficients in classical potential energy functions by machine learning model
- Input: molecular graph (atom and bond)
- Output: coefficients of potential energy functions
- Pros:
 - Learn continuous, environment-sensitive parameters, avoid predefined atom types or bond stiffness
 - Enhance precision over classical MD while keeping transferability
 - Run as fast classical MD simulations
- Cons:
 - Fixed analytical forms in classical energy functions
 - Limited accuracy improvements
- Methods: GAFF-2.2, OpenFF-2.0, Espaloma-0.3, ByteFF, Grappa

End-to-end training machine learning force fields

- Goal: directly predict the molecular energies, atomic forces (and other properties)
- Input: atomic coordinates and atomic number
- Output: potential energy, forces, other properties (dipoles, partial charges and so on)
- Pros:
 - End-to-end training and direct predictions
 - No limitation on analytical forms
 - Near ab initio accuracy
 - Relatively fast simulation speed on GPU
- Cons:
 - Poor generalization to different kinds of biomolecules
 - Highly rely on the quality of training dataset (out-of-distribution risk)
- Methods: ViSNet, MACE, SO3Krate, Allegro, Equiformer v2, GemNet

Transfer classic MD energy terms into model operations to design MLFF



Runtime Geometry Calculation

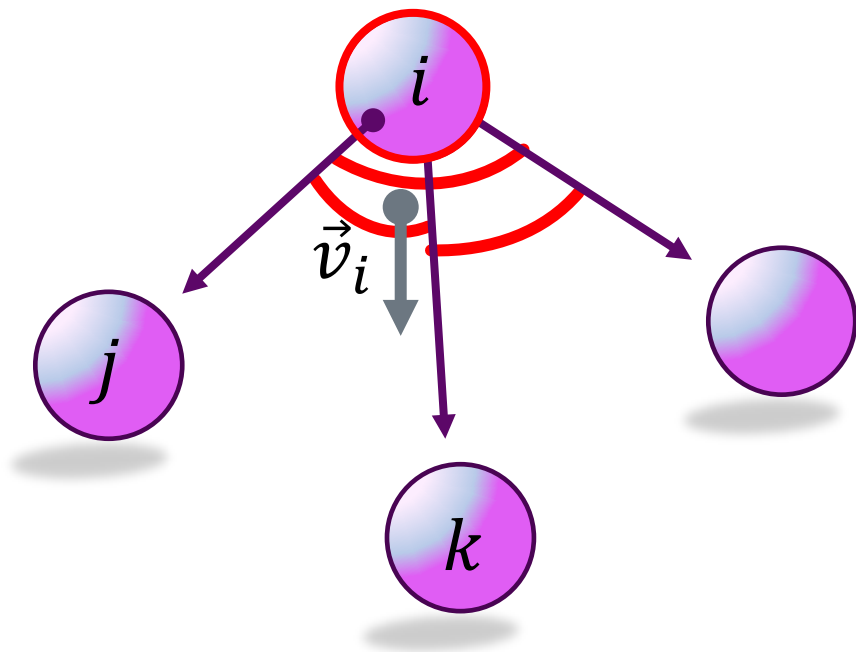
Direction unit: the summarization of all directional information of 1st order neighbors

Direction unit acts as the vectorized representation of node i

Three-body interaction calculation with linear time complexity

Inner product of direction unit represents the summarization of cosine values of all 3-body interactions between the central node and its 1st order neighbors

Node i is the intersection of angles



$$\vec{u}_{ij} = \frac{\vec{r}_{ij}}{\|\vec{r}_{ij}\|} = \frac{(x_j - x_i)}{\|(x_j - x_i)\|}$$

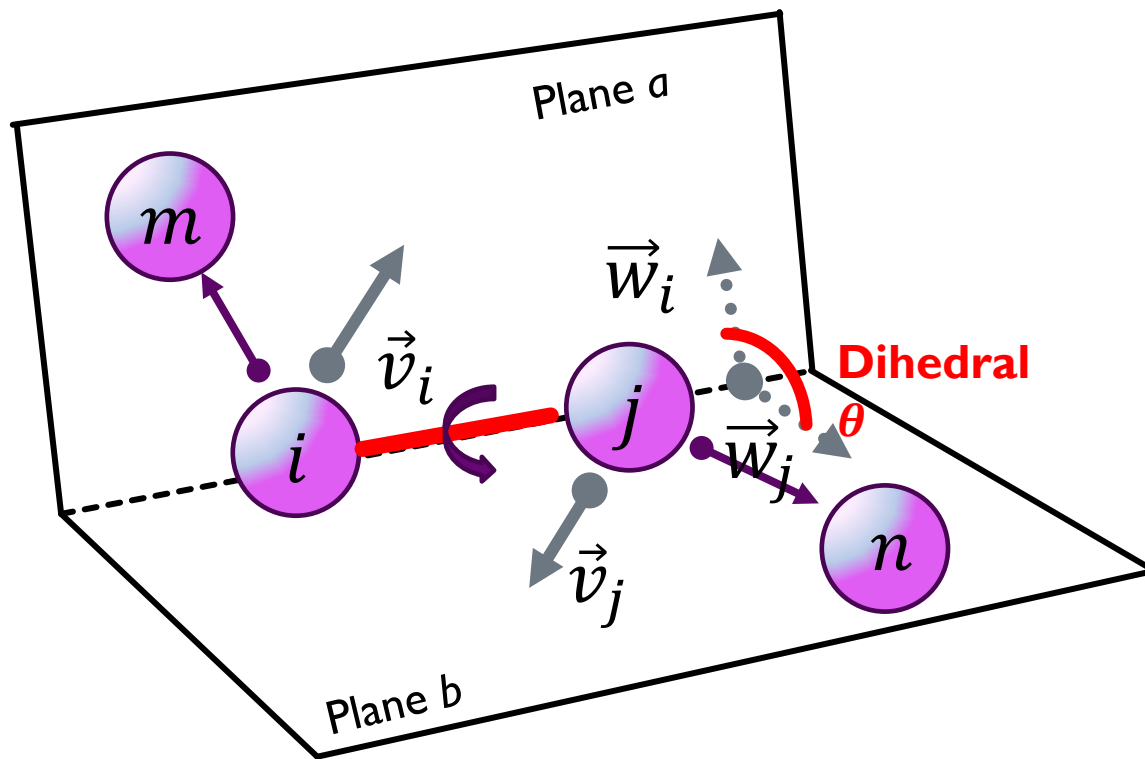
$$\vec{v}_i = \sum_j \vec{u}_{ij}$$

$$\|\vec{v}_i\|^2 = \sum_{j,k} \langle \vec{u}_{ij}, \vec{u}_{ik} \rangle = \sum_{j=1}^N \sum_{k=1}^N \cos \alpha_{jik}$$

The inner product of direction unit represents the summarization of cosine values of all angles between the central node and its 1st order neighbors

Runtime Geometry Calculation

- Four-body interaction calculation with linear time complexity
 4-body interaction is derived from inner product of projections of direction units $\vec{v}_i$ and $\vec{v}_j$
 Edge e_{ij} is the intersection of all dihedrals, f_{ij} is the hidden of e_{ij}



$$\vec{w}_i = \text{Rej}_{\vec{u}_{ij}}(\vec{v}_i) = \vec{v}_i - \langle \vec{v}_i, \vec{u}_{ij} \rangle \cdot \vec{u}_{ij}$$

$$\vec{w}_j = \text{Rej}_{\vec{u}_{ji}}(\vec{v}_j) = \vec{v}_j - \langle \vec{v}_j, \vec{u}_{ji} \rangle \cdot \vec{u}_{ji}$$

$$\begin{aligned} \langle \vec{w}_i, \vec{w}_j \rangle &= \sum_{m=1}^{N_i} \sum_{n=1}^{N_j} \langle \text{Rej}_{\vec{u}_{ij}}(\vec{v}_{im}), \text{Rej}_{\vec{u}_{ji}}(\vec{v}_{jn}) \rangle \\ &= \sum_{m=1}^{N_i} \sum_{n=1}^{N_j} \cos \theta_{mijn} \end{aligned}$$

Vector-Scalar Interactive Message Passing

- Scalar to Vector:

Use scalar message m_i and vector message $\vec{m}_i$ to update vector representation, direction unit $\vec{v}_i$

$$m_{ij}^l = \phi_{MS}(h_i^l, h_j^l, f_{ij}^l)$$

$$\vec{m}_{ij}^l = \phi_{MV}(m_{ij}^l, \vec{r}_{ij}, \vec{v}_j^l)$$

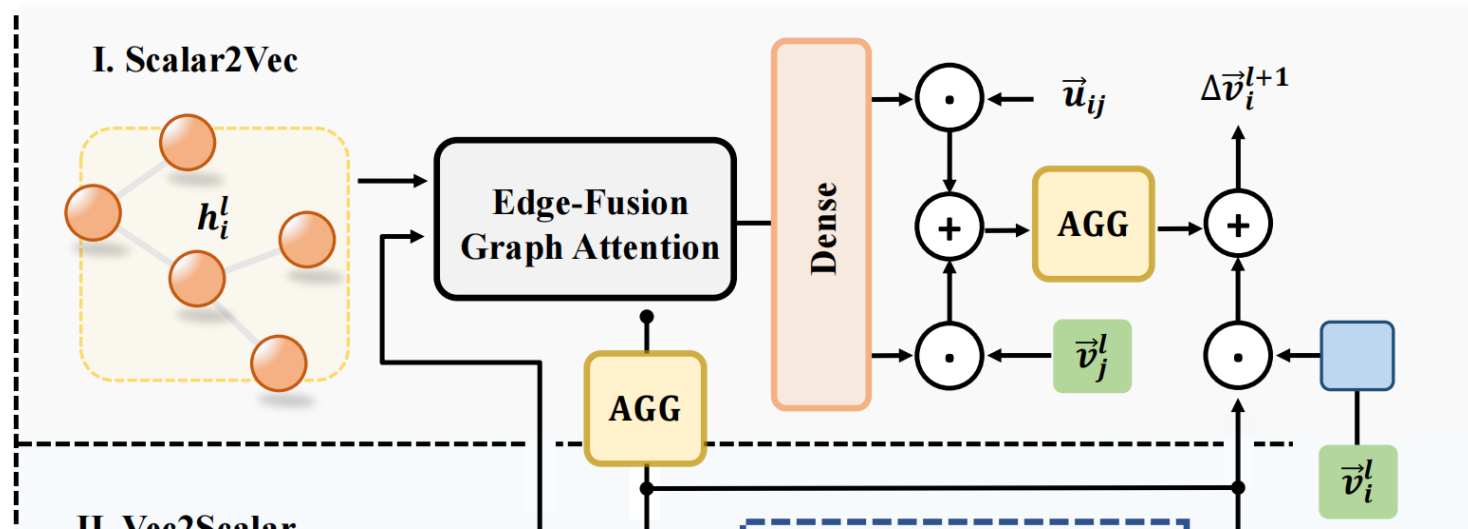
$$m_i^l = \sum_{j \in \mathcal{N}(i)} m_{ij}^l$$

$$\vec{m}_i^l = \sum_{j \in \mathcal{N}(i)} \vec{m}_{ij}^l$$

$$h_i^{l+1} = \phi_{US_N}(h_i^l, m_i^l, \langle \vec{v}_i^l, \vec{v}_i^l \rangle)$$

$$\vec{v}_i^{l+1} = \phi_{UV_N}(\vec{v}_i^l, m_i^l, \vec{m}_i^l)$$

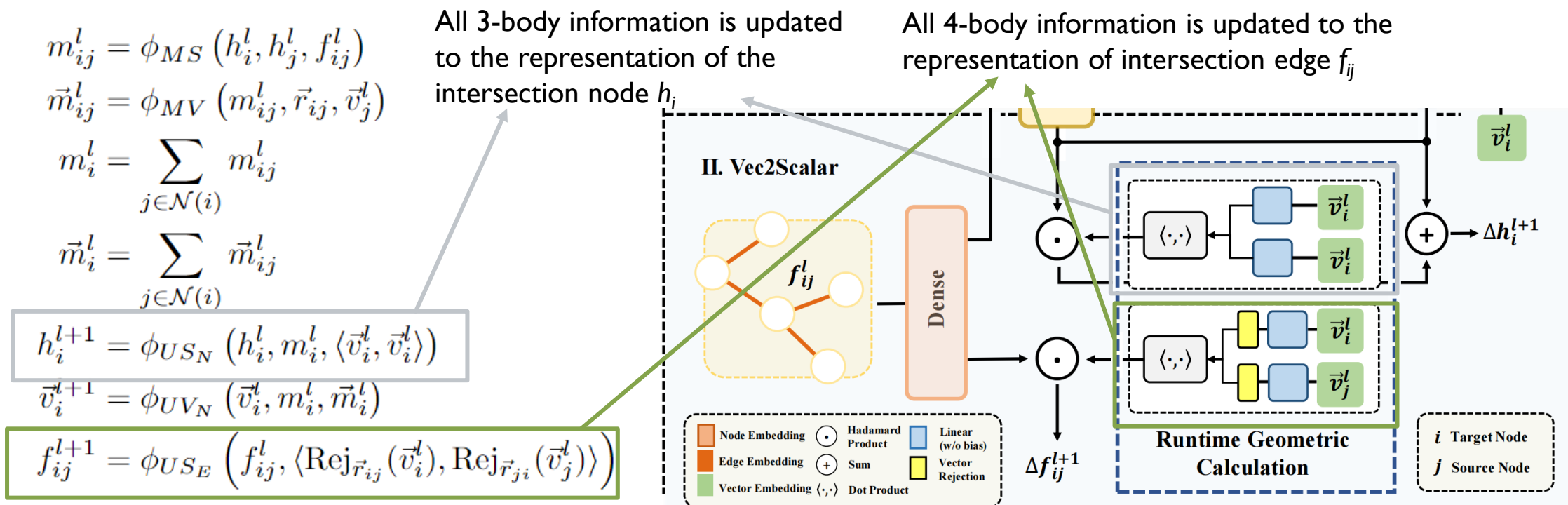
$$f_{ij}^{l+1} = \phi_{US_E}(f_{ij}^l, \langle \text{Rej}_{\vec{r}_{ij}}(\vec{v}_i^l), \text{Rej}_{\vec{r}_{ji}}(\vec{v}_j^l) \rangle)$$



Vector-Scalar Interactive Message Passing

- Vector to Scalar:

Use vector representation, direction unit $\vec{v}_i$ and runtime geometric information to update scalar node representation h_i and edge representation f_{ij}

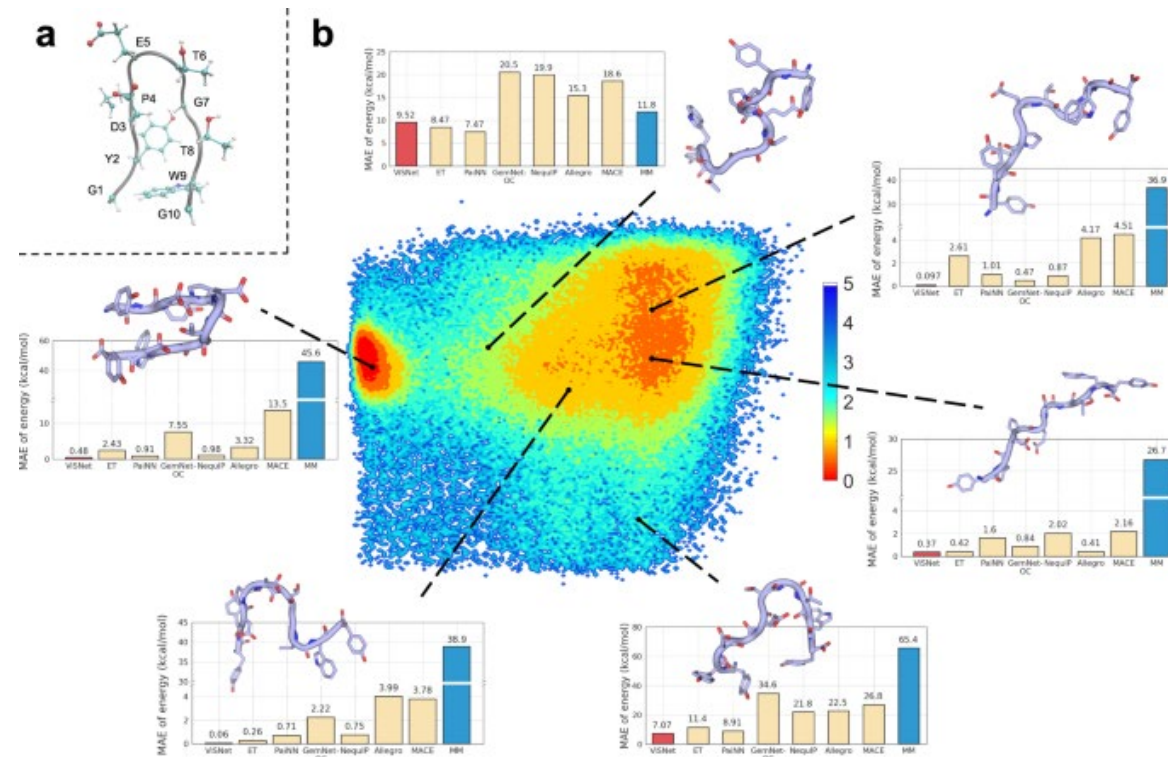


ViSNet model

Performance evaluation

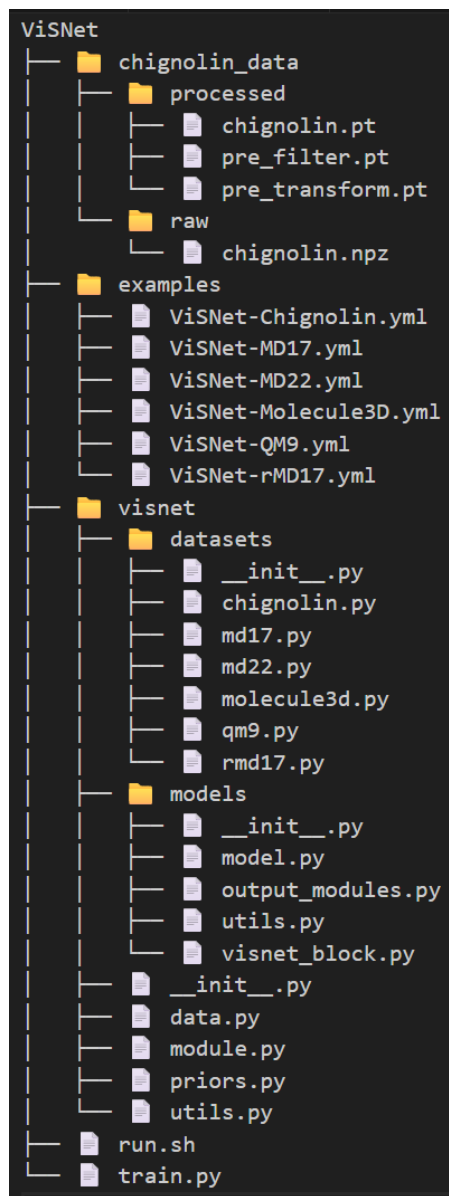
Molecule		SchNet	DimeNet	PaiNN	SpookyNet	ET	GemNet ^a	NequIP ^b	SO3KRATES	ViSNet
Aspirin	Energy	0.37	0.204	0.167	0.151	0.123	—	0.131	0.139	0.116
	Forces	1.35	0.499	0.338	0.258	0.253	0.217	0.184	0.236	0.155
Ethanol	Energy	0.08	0.064	0.064	0.052	0.052	—	0.051	0.061	0.051
	Forces	0.39	0.230	0.224	0.094	0.109	0.085	0.071	0.096	0.060
Malondialdehyde	Energy	0.13	0.104	0.091	0.079	0.077	—	0.076	0.077	0.075
	Forces	0.66	0.383	0.319	0.167	0.169	0.155	0.129	0.147	0.100
Naphthalene	Energy	0.16	0.122	0.116	0.116	0.085	—	0.113	0.115	0.085
	Forces	0.58	0.215	0.077	0.089	0.061	0.051	0.039	0.074	0.039
Salicylic acid	Energy	0.20	0.134	0.116	0.114	0.093	—	0.106	0.106	0.092
	Forces	0.85	0.374	0.195	0.180	0.129	0.125	0.090	0.145	0.084
Toluene	Energy	0.12	0.102	0.095	0.094	0.074	—	0.092	0.095	0.074
	Forces	0.57	0.216	0.094	0.087	0.067	0.060	0.046	0.073	0.039
Uracil	Energy	0.14	0.115	0.106	0.105	0.095	—	0.104	0.103	0.095
	Forces	0.56	0.301	0.139	0.119	0.095	0.097	0.076	0.111	0.062

SoTA performance for energy & force predictions on MD17, MD22, QM9, Molecule 3D datasets



Accurate energy prediction for different conformations of proteins than MM and other MLFFs

ViSNet code overview



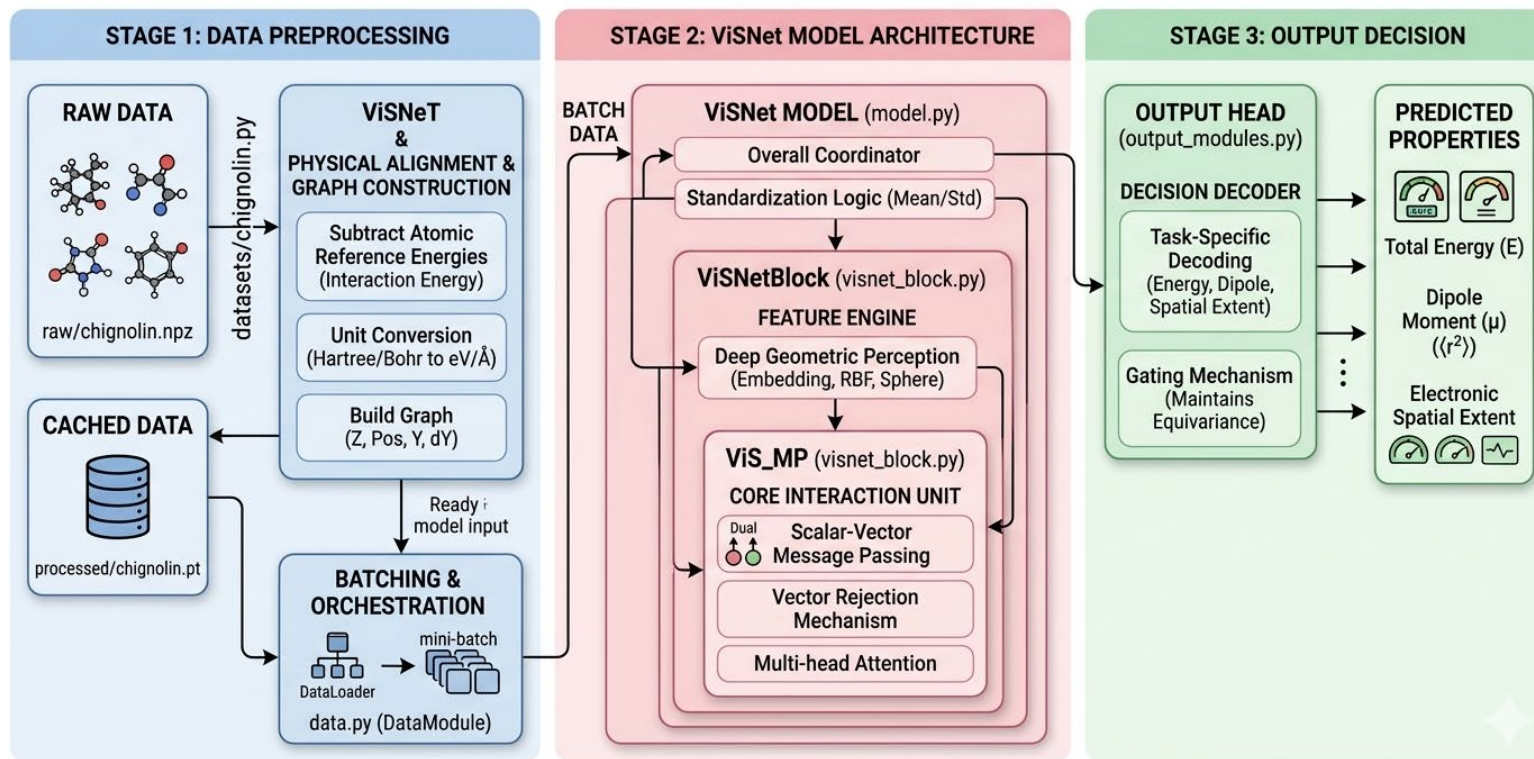
➤ Datasets

➤ Models

➤ Modules

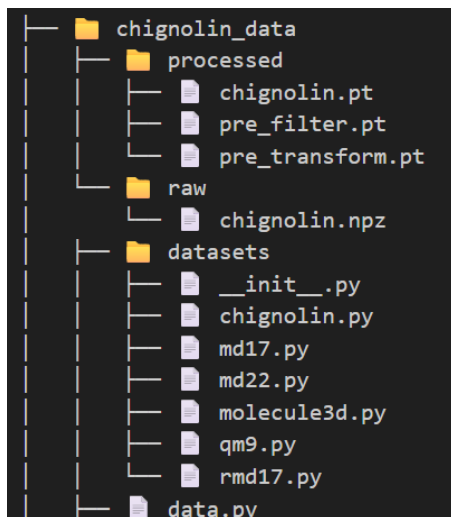
➤ Scripts

➤ Arguments



ViSNet Workflow

ViSNet Datasets -- Use chignolin as example



```
class Chignolin(InMemoryDataset):

    self_energies = {
        1: -0.496665677271,
        6: -37.8289474402,
        7: -54.5677547104,
        8: -75.0321126521,
        16: -398.063946327,
    }

    def __init__(self, root, transform=None, pre_transform=None):

        super(Chignolin, self).__init__(root, transform, pre_transform)

        self.data, self.slices = torch.load(self.processed_paths[0])

    @property
    def raw_file_names(self):
        return [f'chignolin.npz']

    @property
    def processed_file_names(self):
        return [f'chignolin.pt']

    def process(self):
        for path, processed_path in zip(self.raw_paths, self.processed_paths):
```

Chignolin.p

- Data collecting and Cleaning: Unify atom counts and tensor shapes.
- Standardize units and subtract self-energies to simplify learning.
- Convert NumPy arrays into PyTorch Geometric (GNN) tensors.

```
class DataModule(LightningDataModule):

    def __init__(self, hparams):
        super(DataModule, self).__init__()
        self.hparams.update(hparams.__dict__) if hasattr(hparams, "__dict__") else self.hparams.update(hparams)
        self.mean, self.std = None, None
        self._saved_dataloaders = dict()
        self.dataset = None

    def prepare_dataset(self):

        assert hasattr(self, f"_prepare_{self.hparams['dataset']}_dataset"), f"Dataset {self.hparams['dataset']} not defined"
        dataset_factory = lambda t: getattr(self, f"_prepare_{t}_dataset")()
        self.idx_train, self.idx_val, self.idx_test = dataset_factory(self.hparams["dataset"])

        print(f"train {len(self.idx_train)}, val {len(self.idx_val)}, test {len(self.idx_test)}")
        self.train_dataset = Subset(self.dataset, self.idx_train)
        self.val_dataset = Subset(self.dataset, self.idx_val)
        self.test_dataset = Subset(self.dataset, self.idx_test)

        if self.hparams["standardize"]:
            self._standardize()

    def train_dataloader(self):
        return self._get_dataloader(self.train_dataset, "train")

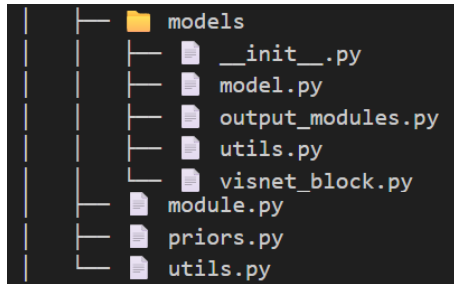
    def val_dataloader(self):
        loaders = [self._get_dataloader(self.val_dataset, "val")]
        delta = 1 if self.hparams['reload'] == 1 else 2
        if (
            len(self.test_dataset) > 0
            and (self.trainer.current_epoch + delta) % self.hparams["test_interval"] == 0
        ):
            loaders.append(self._get_dataloader(self.test_dataset, "test"))
        return loaders

    def test_dataloader(self):
        return self._get_dataloader(self.test_dataset, "test")
```

Data.py

- Multi-Dataset Integration
- Data Splitting
- Data Standardization
- Automated DataLoader

ViSNet Models & Modules



```
class ViSNet(nn.Module):
    def __init__(
        self,
        representation_model,
        output_model,
        prior_model=None,
        reduce_op="add",
        mean=None,
        std=None,
        derivative=False,
    ):
        super(ViSNet, self).__init__()
        self.representation_model = representation_model
        self.output_model = output_model

        self.prior_model = prior_model
        if not output_model.allow_prior_model and prior_model is not None:
            self.prior_model = None
            rank_zero_warn(
                "Prior model was given but the output model does "
                "not allow prior models. Dropping the prior model."
            )

        self.reduce_op = reduce_op
        self.derivative = derivative

        mean = torch.scalar_tensor(0) if mean is None else mean
        self.register_buffer("mean", mean)
        std = torch.scalar_tensor(1) if std is None else std
        self.register_buffer("std", std)

        self.reset_parameters()

    def reset_parameters(self):
        self.representation_model.reset_parameters()
        self.output_model.reset_parameters()
        if self.prior_model is not None:
            self.prior_model.reset_parameters()

    def forward(self, data: Data) -> Tuple[Tensor, Optional[Tensor]]:
        if self.derivative:
            data.pos.requires_grad_(True)

        x, v = self.representation_model(data)
        x = self.output_model.pre_reduce(x, v, data.z, data.pos, data.batch)
        x = x * self.std
```

model.py

- Create model and output network
- Equivariant Prediction: Handles forward passes
- Automatic Differentiation: Implements the energy-to-force derivative calculation via PyTorch autograd.

ViSNet Models & Modules

```
class ViSNetBlock(nn.Module):

    def __init__(...

    def reset_parameters(self): ...

    def forward(self, data: Data) -> Tuple[Tensor, Tensor]:

        z, pos, batch = data.z, data.pos, data.batch

        # Embedding Layers
        x = self.embedding(z)
        edge_index, edge_weight, edge_vec = self.distance(pos, batch)
        edge_attr = self.distance_expansion(edge_weight)
        mask = edge_index[0] != edge_index[1]
        edge_vec[mask] = edge_vec[mask] / torch.norm(edge_vec[mask], dim=1).unsqueeze(1)
        edge_vec = self.sphere(edge_vec)
        x = self.neighbor_embedding(z, x, edge_index, edge_weight, edge_attr)
        vec = torch.zeros(x.size(0), ((self.lmax + 1) ** 2) - 1, x.size(1), device=x.device)
        edge_attr = self.edge_embedding(edge_index, edge_attr, x)

        # ViS-MP Layers
        for attn in self.vis_mp_layers[:-1]:
            dx, dvec, dedge_attr = attn(x, vec, edge_index, edge_weight, edge_attr, edge_vec)
            x = x + dx
            vec = vec + dvec
            edge_attr = edge_attr + dedge_attr

        dx, dvec, _ = self.vis_mp_layers[-1](x, vec, edge_index, edge_weight, edge_attr, edge_vec)
        x = x + dx
        vec = vec + dvec

        x = self.out_norm(x)
        vec = self.vec_out_norm(vec)

        return x, vec
```

visnet_block.py -- ViSNetBlock

- **Input Encoding:** Convert atomic numbers (Z) into high-dimensional embeddings and expand interatomic distances via RBF (Radial Basis Functions) into rich geometric features.
- **Spatial Awareness:** Use the Sphere module to process atomic direction vectors, enabling the model to perceive molecular rotation and orientation in 3D space.
- **Iterative Update:** Stack multiple ViS_MP layers to continuously refine and update atomic state information.

ViSNet Models & Modules

```
class ViS_MP(MessagePassing):
    def __init__(...

    @staticmethod
    def vector_rejection(vec, d_ij):
        vec_proj = (vec * d_ij.unsqueeze(2)).sum(dim=1, keepdim=True)
        return vec - vec_proj * d_ij.unsqueeze(2)

    def reset_parameters(self): ...

    def forward(self, x, vec, edge_index, r_ij, f_ij, d_ij): ...

    def message(self, q_i, k_j, v_j, vec_j, dk, dv, r_ij, d_ij):
        attn = (q_i * k_j * dk).sum(dim=-1)
        attn = self.attn_activation(attn) * self.cutoff(r_ij).unsqueeze(1)

        v_j = v_j * dv
        v_j = (v_j * attn.unsqueeze(2)).view(-1, self.hidden_channels)

        s1, s2 = torch.split(self.act(self.s_proj(v_j)), self.hidden_channels, dim=1)
        vec_j = vec_j * s1.unsqueeze(1) + s2.unsqueeze(1) * d_ij.unsqueeze(2)

        return v_j, vec_j

    def edge_update(self, vec_i, vec_j, d_ij, f_ij):
        w1 = self.vector_rejection(self.w_trg_proj(vec_i), d_ij)
        w2 = self.vector_rejection(self.w_src_proj(vec_j), -d_ij)
        w_dot = (w1 * w2).sum(dim=1)
        df_ij = self.act(self.f_proj(f_ij)) * w_dot
        return df_ij

    def aggregate(
        self,
        features: Tuple[torch.Tensor, torch.Tensor],
        index: torch.Tensor,
        ptr: Optional[torch.Tensor],
        dim_size: Optional[int],
    ) -> Tuple[torch.Tensor, torch.Tensor]:
        x, vec = features
        x = scatter(x, index, dim=self.node_dim, dim_size=dim_size)
        vec = scatter(vec, index, dim=self.node_dim, dim_size=dim_size)
        return x, vec

    def update(self, inputs: Tuple[torch.Tensor, torch.Tensor]) -> Tuple[torch.Tensor, torch.Tensor]:
        return inputs
```

visnet_block.py -- Vis_MP

- **Dual-Channel (Scalar-Vector):** It propagates both atomic energy features (scalar x) and directional/moment features (vector vec) simultaneously.
- **Vector Rejection:** The `vector_rejection` function decomposes 3D vectors into parallel and perpendicular components relative to atomic bonds, ensuring high physical precision for rotational invariance and equivariance.
- **Attention Mechanism:** Multi-head Attention is incorporated to let the model automatically learn which atomic interactions contribute most to the total energy.

ViSNet Models & Modules

```
class GatedEquivariantBlock(nn.Module):
    """
    Gated Equivariant Block as defined in Schütt et al. (2021):
    Equivariant message passing for the prediction of tensorial properties and molecular spectra
    """
    def __init__(...
    def reset_parameters(self): ...
    def forward(self, x, v): ...

class OutputModel(nn.Module, metaclass=ABCMeta):
    def __init__(self, allow_prior_model):
        super(OutputModel, self).__init__()
        self.allow_prior_model = allow_prior_model

    def reset_parameters(self):
        pass

    @abstractmethod
    def pre_reduce(self, x, v, z, pos, batch):
        return

    def post_reduce(self, x):
        return x
```

output_modules.py

- **Task-Specific Heads:** Implements specialized modules to predict diverse physical properties like Total Energy, Dipole Moments, and Spatial Extent.
- **Equivariant Gating:** Utilizes GatedEquivariantBlock to maintain physical consistency and rotational equivariance for tensorial properties.
- **Physical Constraints:** Incorporates chemical knowledge, such as center-of-mass (COM) calculations and atomic masses, into the neural network's logic.
- **Hierarchical Reduction:** Decouples per-atom feature mapping (pre_reduce) from global molecular property derivation (post_reduce).

ViSNet Models & Modules

```
class LNNP(LightningModule):
    def __init__(self, hparams, prior_model=None, mean=None, std=None):
        super(LNNP, self).__init__()

        self.save_hyperparameters(hparams)

        if self.hparams.load_model:
            self.model = load_model(self.hparams.load_model, args=self.hparams)
        else:
            self.model = create_model(self.hparams, prior_model, mean, std)

        self._reset_losses_dict()
        self._reset_ema_dict()
        self._reset_inference_results()

    def configure_optimizers(self):
        optimizer = AdamW(
            self.model.parameters(),
            lr=self.hparams.lr,
            weight_decay=self.hparams.weight_decay,
        )
        scheduler = ReduceLROnPlateau(
            optimizer,
            "min",
            factor=self.hparams.lr_factor,
            patience=self.hparams.lr_patience,
            min_lr=self.hparams.lr_min,
        )
        lr_scheduler = {
            "scheduler": scheduler,
            "monitor": "val_loss",
            "interval": "epoch",
            "frequency": 1,
        }
        return [optimizer], [lr_scheduler]

    def forward(self, data):
        return self.model(data)

    def training_step(self, batch, batch_idx):
        loss_fn = mse_loss if self.hparams.loss_type == 'MSE' else l1_loss

        return self.step(batch, loss_fn, "train")

    def validation_step(self, batch, batch_idx, *args):
        if len(args) == 0 or (len(args) > 0 and args[0] == 0):
            # validation step
            return self.step(batch, mse_loss, "val")
        # test step
```

modules.py

- **PyTorch Lightning Wrapper:** Automates the training, validation, and testing loops for the ViSNet model.
- **Multi-Objective Optimization:** Jointly optimizes Energy and Force predictions with configurable weights and exponential moving averages (EMA) for loss smoothing.
- **Advanced Scheduling:** Implements learning rate warmup and plateau-based reduction to ensure stable convergence.
- **Metrics Logging & Inference:** Tracks training statistics (Loss, LR) and aggregates prediction results for post-training physical analysis.

ViSNet scripts -- train.py

```
def main():
    args = get_args()
    pl.seed_everything(args.seed, workers=True)

    # initialize data module
    data = DataModule(args)
    data.prepare_dataset()

    default = ",".join(str(i) for i in range(torch.cuda.device_count()))
    cuda_visible_devices = os.getenv("CUDA_VISIBLE_DEVICES", default=default).split(",")
    dir_name = f"output_ngpus_{len(cuda_visible_devices)}_bs_{args.batch_size}_lr_{args.lr}_seed_{args.seed} + \
    f"_reload_{args.reload}_lmax_{args.lmax}_vnorm_{args.vecnorm_type}" + \
    f"_vertex_{args.vertex_type}_L_{args.num_layers}_D_{args.embedding_dimension}_H_{args.num_heads}" + \
    f"_cutoff_{args.cutoff}_E_{args.energy_weight}_F_{args.force_weight}_loss_{args.loss_type}"

    if args.load_model is None:
        args.log_dir = os.path.join(args.log_dir, dir_name)
        if os.path.exists(args.log_dir):
            if os.path.exists(os.path.join(args.log_dir, "last.ckpt")):
                args.load_model = os.path.join(args.log_dir, "last.ckpt")
            csv_path = os.path.join(args.log_dir, "metrics.csv")
            while os.path.exists(csv_path):
                csv_path = csv_path + '.bak'
            if os.path.exists(os.path.join(args.log_dir, "metrics.csv")):
                os.rename(os.path.join(args.log_dir, "metrics.csv"), csv_path)

    prior = None
    if args.prior_model:
        assert hasattr(priors, args.prior_model), (
            f"Unknown prior model {args['prior_model']}. "
            f"Available models are {'', '.join(priors.__all__)}"
        )
        # initialize the prior model
        prior = getattr(priors, args.prior_model)(dataset=data.dataset)
        args.prior_args = prior.get_init_args()

    # initialize lightning module
    model = LNNP(args, prior_model=prior, mean=data.mean, std=data.std)
```

Get arguments

Data preparation

Load checkpoint*

ViSNet scripts -- train.py

Train

```
if args.task == "train":

    checkpoint_callback = ModelCheckpoint(
        dirpath=args.log_dir,
        monitor="val_loss",
        save_top_k=10,
        save_last=True,
        every_n_epochs=args.save_interval,
        filename="{epoch}-{val_loss:.4f}-{test_loss:.4f}",
    )

    early_stopping = EarlyStopping("val_loss", patience=args.early_stopping_patience)
    tb_logger = TensorBoardLogger(args.log_dir, name="tensorbord", version="", default_hp_metric=False)
    csv_logger = CSVLogger(args.log_dir, name="", version="")
    ddp_plugin = DDPStrategy(find_unused_parameters=False)

    trainer = pl.Trainer(
        max_epochs=args.num_epochs,
        gpus=args.ngpus,
        num_nodes=args.num_nodes,
        accelerator=args.accelerator,
        default_root_dir=args.log_dir,
        auto_lr_find=False,
        callbacks=[early_stopping, checkpoint_callback],
        logger=[tb_logger, csv_logger],
        reload_dataloaders_every_n_epochs=args.reload,
        precision=args.precision,
        strategy=ddp_plugin,
        enable_progress_bar=True,
    )

    trainer.fit(model, datamodule=data, ckpt_path=args.load_model)

test_trainer = pl.Trainer(
    logger=False,
    max_epochs=-1,
    num_nodes=1,
    gpus=1,
    default_root_dir=args.log_dir,
    enable_progress_bar=True,
    inference_mode=False,
)

if args.task == 'train':
    test_trainer.test(model=model, ckpt_path=trainer.checkpoint_callback.best_model_path, datamodule=data)
elif args.task == 'inference':
    test_trainer.test(model=model, datamodule=data)
    torch.save(model.inference_results, os.path.join(args.log_dir, "inference_results.pt"))
```

Inference

Train Task Command:

```
python train.py \
    --conf examples/ViSNet-Chignolin.yml \
    --log-dir path to log dir
```

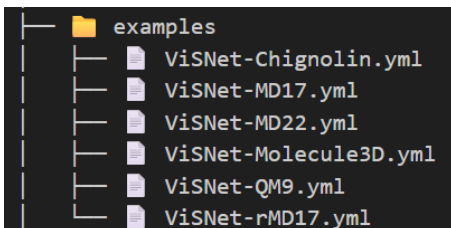
Output is saved in metrics.csv.bak:

```
epoch,lr,train_loss,val_loss,train_loss_y,train_loss_dy,val_loss_y,val_loss_dy,step,test_loss,test_loss_y,test_loss_dy
994.0,1.0141205564195843e-07,0.007922888733446598,0.010897649452090263,0.08564256131649017,0.0038323805201798677,0.128120765
995.0,1.0141205564195843e-07,0.008026283234357834,0.010311990045011044,0.08769819140434265,0.0038330245297402143,0.116394676
996.0,1.0141205564195843e-07,0.00806686282157898,0.010167432948946953,0.08836275339126587,0.0038407633546739817,0.1135077706
997.0,1.0141205564195843e-07,0.008002318441867828,0.0101396469399333,0.08723167330026627,0.0038323530461639166,0.11296665661
998.0,1.0141205564195843e-07,0.008100644685328007,0.010135232470929623,0.08919958770275116,0.0038322792388498783,0.112874686
999.0,1.0141205564195843e-07,0.00798735860735178,0.01014835387468338,0.08688429743051529,0.0038348883390426636,0.11313199251
```

Inference Task Command:

```
python train.py \
    --conf examples/ViSNet-Chignolin.yml \
    --log-dir path to log dir \
    --task inference \
    --load-model path to ckpt
```

ViSNet arguments



```
load_model: null

# training settings
num_epochs: 1000
lr_warmup_steps: 3000
lr: 0.0002
lr_patience: 10
lr_min: 1.e-07
lr_factor: 0.8
weight_decay: 0.0
early_stopping_patience: 100
loss_type: MSE
loss_scale_y: 0.05
loss_scale_dy: 1.0
energy_weight: 0.05
force_weight: 0.95
```

```
# dataset specific
dataset: Chignolin
dataset_arg: null
dataset_root: ./chignolin_data
derivative: true
split_mode: null

# dataloader specific
reload: 1
batch_size: 4
inference_batch_size: 4
standardize: true
splits: null
train_size: 0.8
val_size: 0.1
test_size: null
num_workers: 12
```

Training Settings:

- Total epochs for training
- Learning rate
- minimum allowed learning rates
- The training weight of energy & force in loss function

Dataset Settings:

- The number of graphs processed in a single training step
- Fraction of train, validation and test set
- Number of CPU cores used for parallel data loading.

```
# model architecture specific
model: ViSNetBlock
output_model: Scalar
prior_model: null

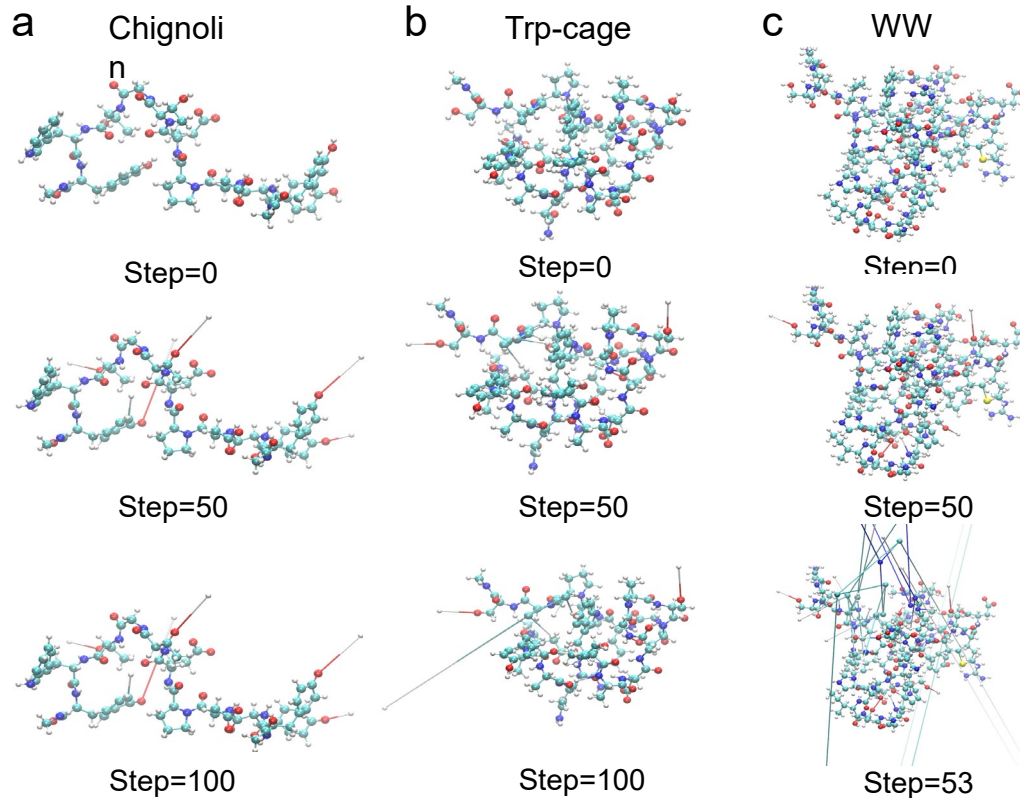
# architectural specific
embedding_dimension: 128
num_layers: 6
num_rbf: 32
activation: silu
rbf_type: expnorm
trainable_rbf: false
attn_activation: silu
num_heads: 8
cutoff: 5.0
max_z: 100
max_num_neighbors: 32
reduce_op: add
lmax: 1
vecnorm_type: none
trainable_vecnorm: false
vertex_type: None

# other specific
ngpus: -1
num_nodes: 1
precision: 32
log_dir: /path/to/logs
task: train
seed: 1
distributed_backend: ddp
redirect: false
accelerator: gpu
test_interval: 100
save_interval: 1
```

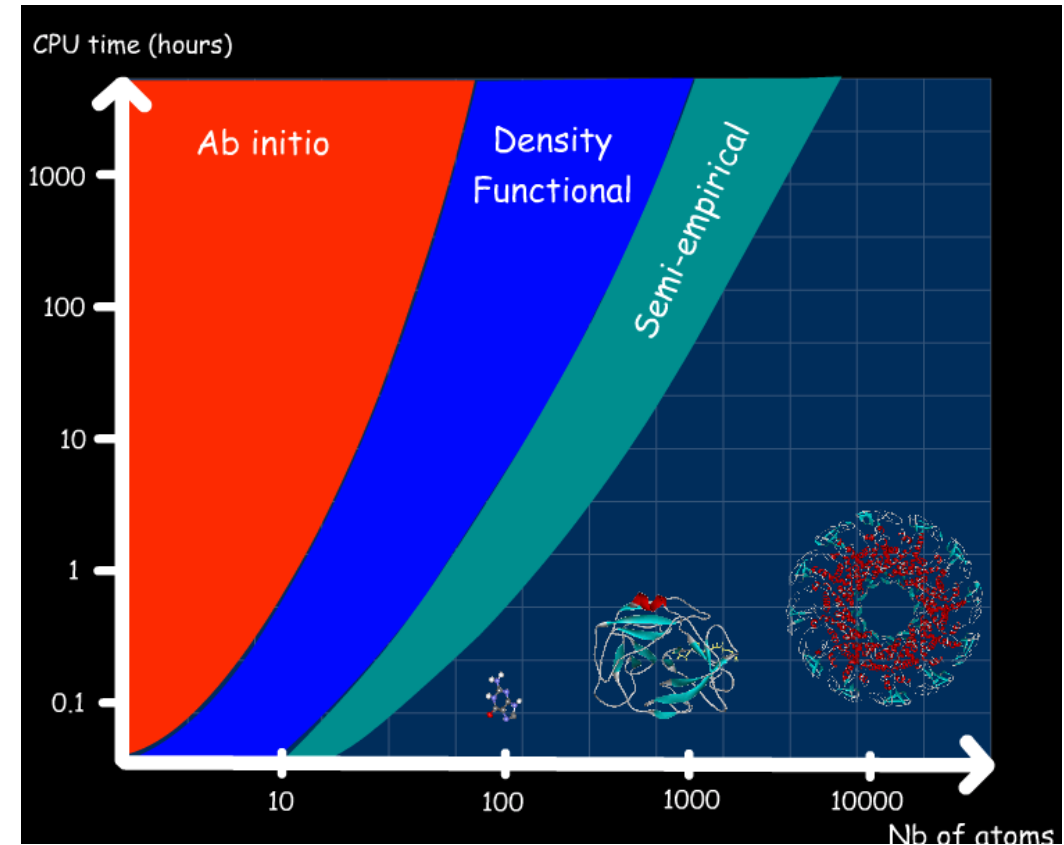
Model Settings:

- The size of the hidden vector representing each atom
- The number of ViS_MP (Message Passing) blocks stacked together
- Number of attention heads used in the message-passing layers
- The maximum distance (in Å) for atoms to be considered "neighbors"
- The maximum degree of spherical harmonics for geometric vector representation
- Perform train or inference task

Generalization challenge in AI powered MD



MLFF lacks generalization to other proteins and leads simulation collapse

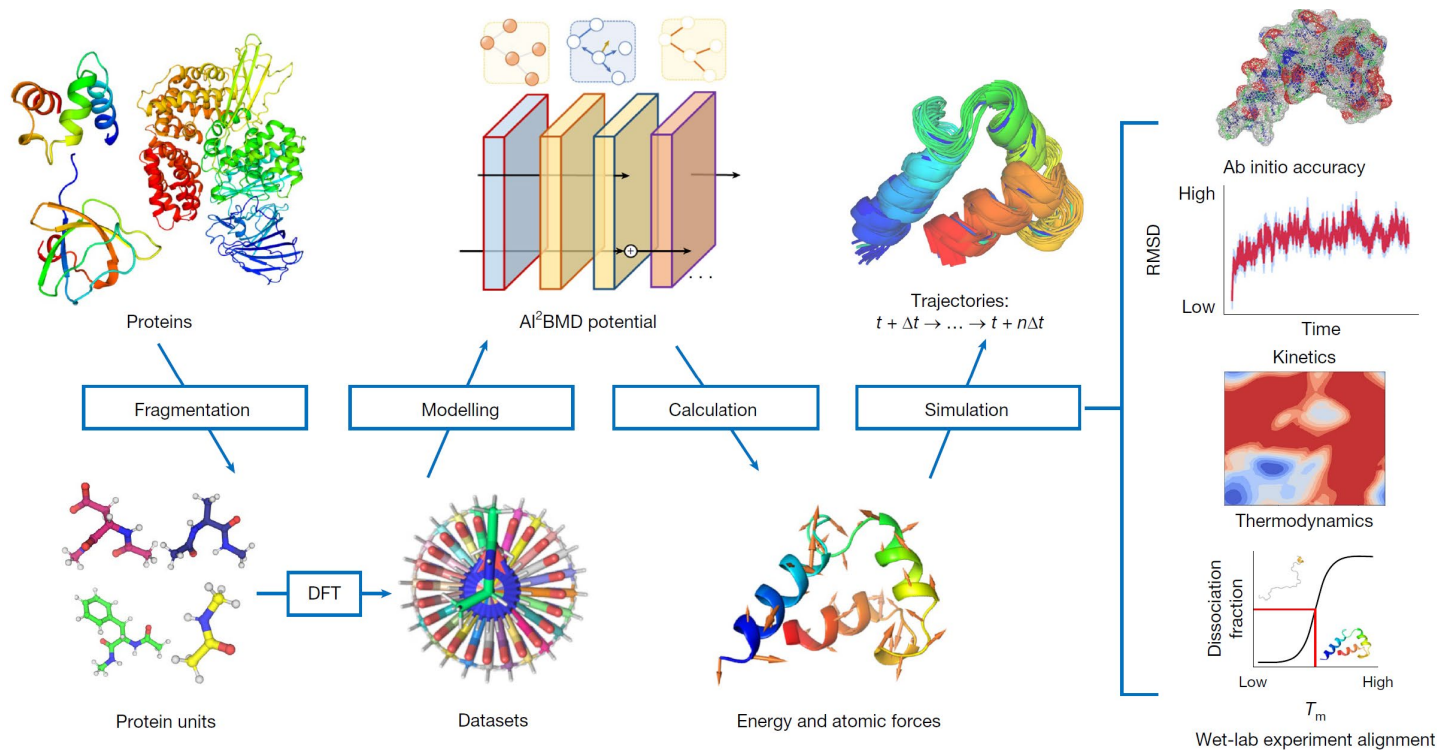


To train MLFF for various proteins, data generation at DFT level is prohibitive

Universal MLFF for diverse biomolecular systems

- Goal: generalizable to different biomolecular MD simulations
- Input: atomic coordinates and atomic number
- Output: potential energy and forces (fragment level and global level)
- Pros:
 - Generalizable fragmentation to different biomolecules
 - Near ab initio accuracy
 - Simulation for different molecules without retraining
- Cons:
 - Relatively slow simulation speed due to fragmentation
 - Complicated fragmentation approach
- Methods: AI²BMD, GEMS

The overall flowchart of AI²BMD



[nature](#) > [articles](#) > [article](#)

Article | [Open access](#) | Published: 06 November 2024

Ab initio characterization of protein molecular dynamics with AI²BMD

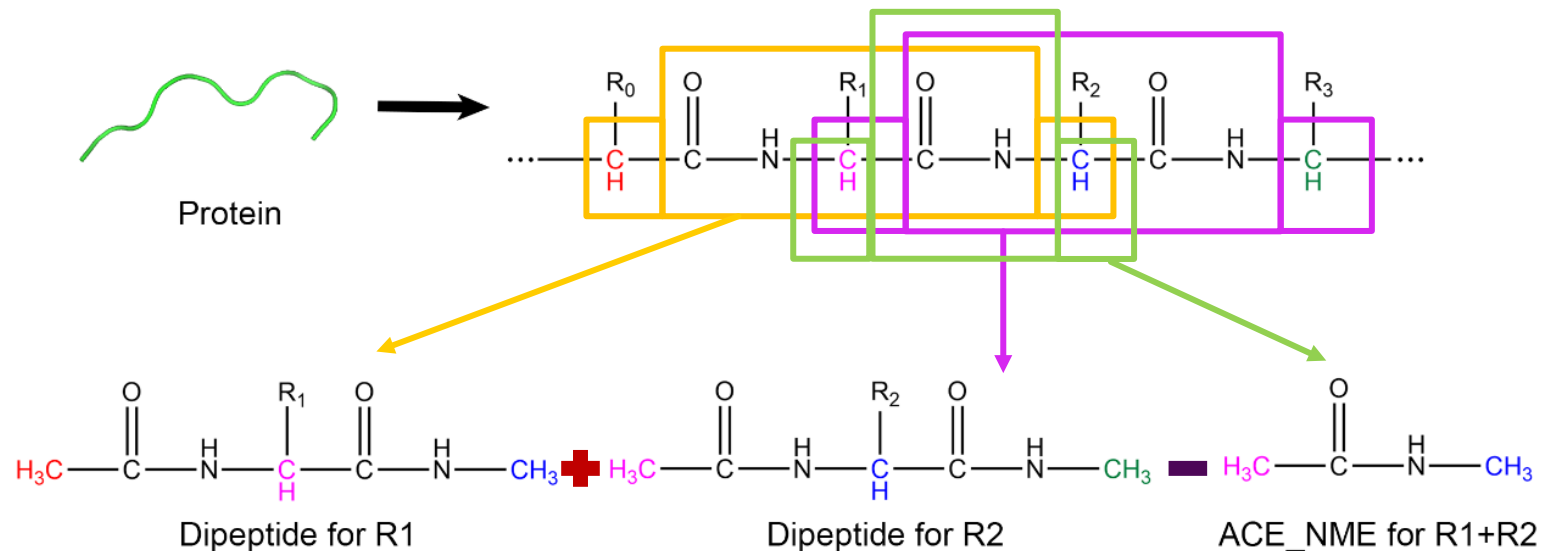
[Tong Wang](#) , [Xinheng He](#), [Mingyu Li](#), [Yatao Li](#), [Ran Bi](#), [Yusong Wang](#), [Chaoran Cheng](#), [Xiangzhen Shen](#), [Jiawei Meng](#), [He Zhang](#), [Haiguang Liu](#), [Zun Wang](#), [Shaoning Li](#), [Bin Shao](#)  & [Tie-Yan Liu](#)

Nature **635**, 1019–1027 (2024) | [Cite this article](#)

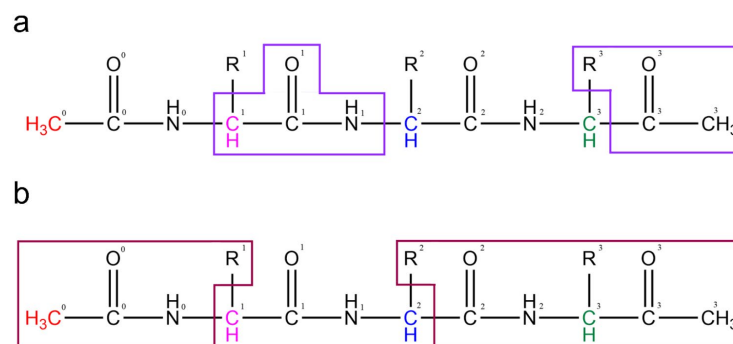
56k Accesses | **202** Altmetric | [Metrics](#)

Generalizable protein fragmentation

Fragmentation process

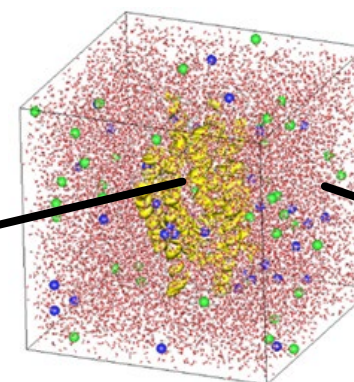


Combination process



extra interactions among
nonoverlapped protein
units

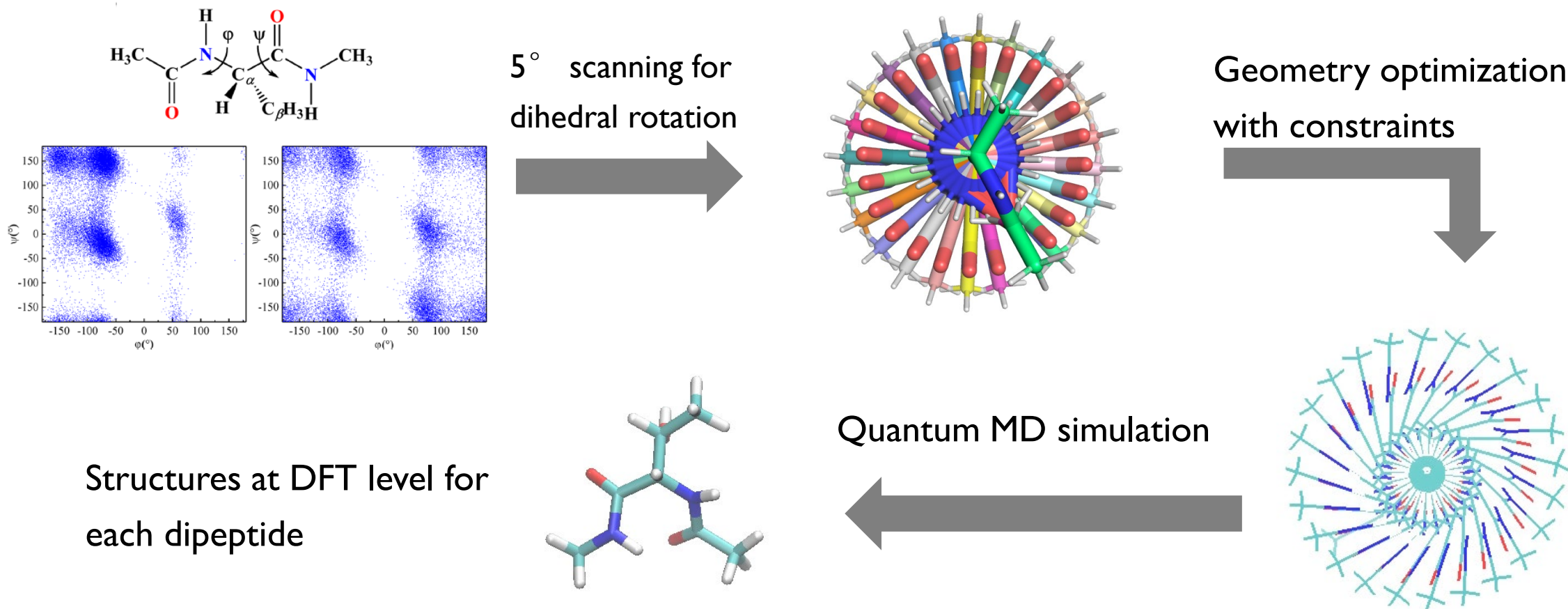
Calculation
by MLFF



extra interactions
with solvent and
ions

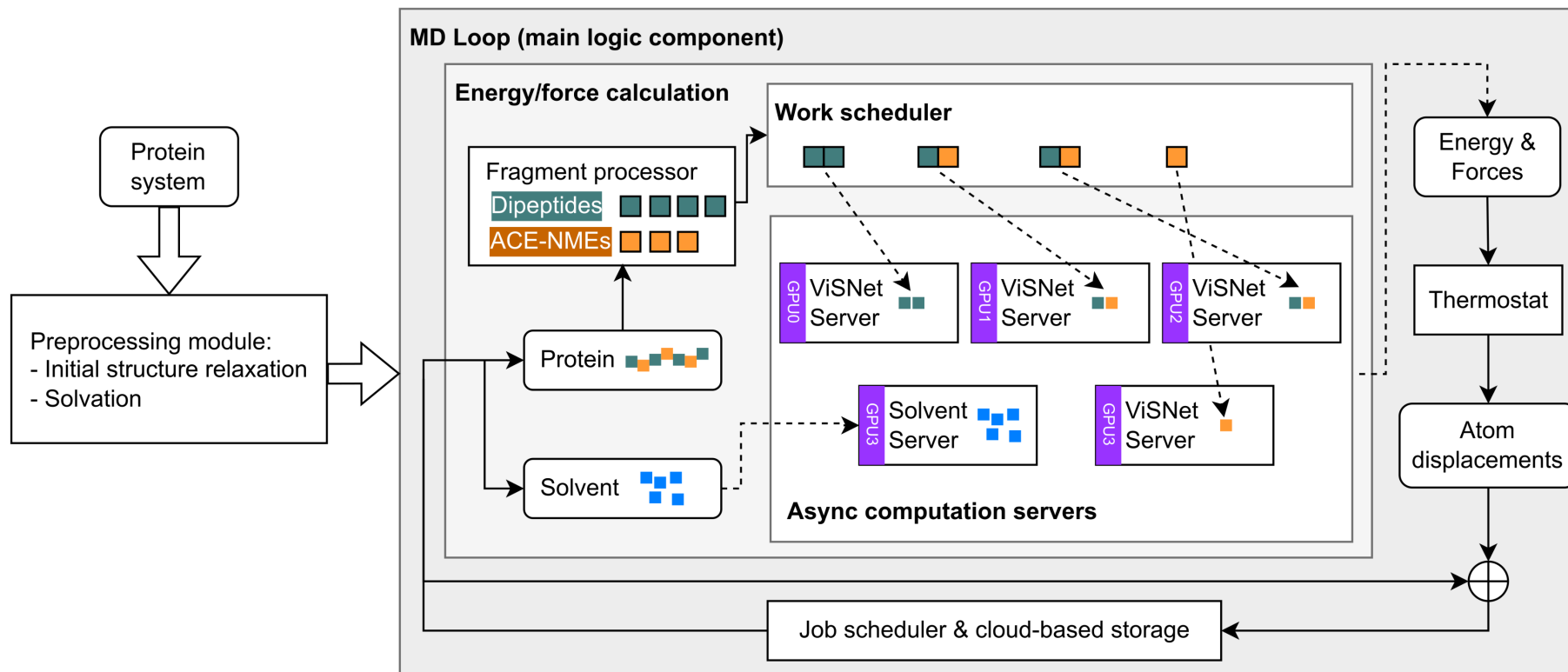
Calculation
by Amoeba

Ab initio dataset: Protein Unit Dataset (PUD)



21 M samples at DFT level at the cost of ~1,600 CPU core years

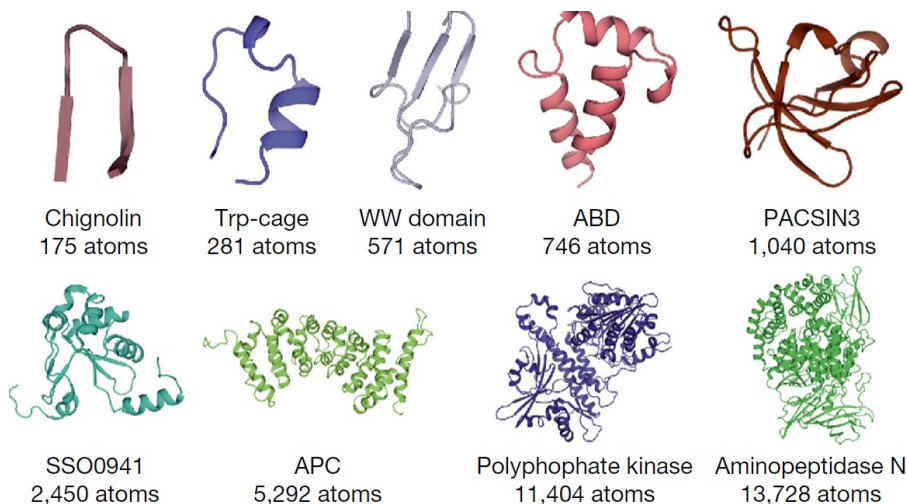
AI²BMD simulation program



- Tunable CPU / GPU device strategy (90X acceleration)
- Asynchronous client/server paradigm (90X acceleration)
- Cloud-oriented computing (20X acceleration)

> 200 X simulation speed acceleration compared to a basic implementation

AI²BMD: accuracy

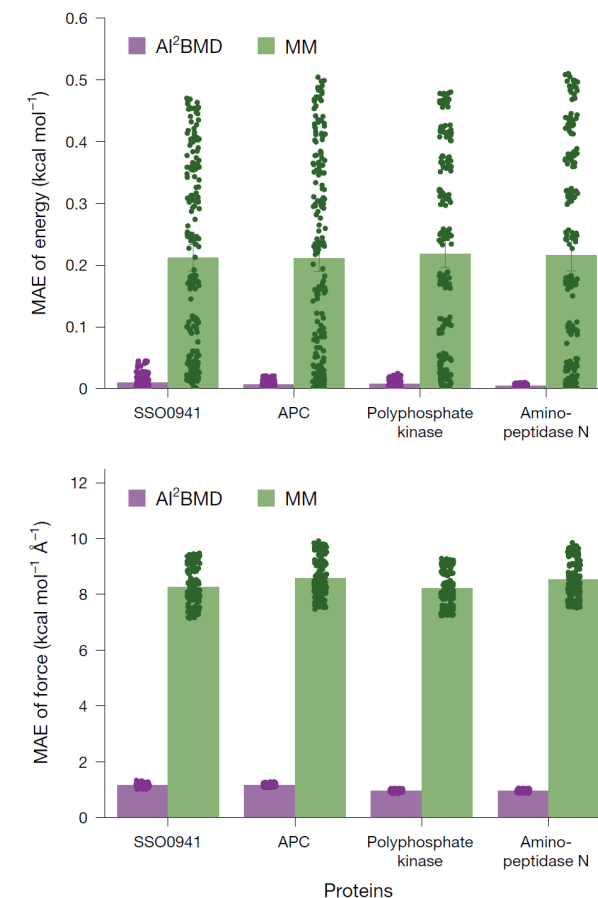
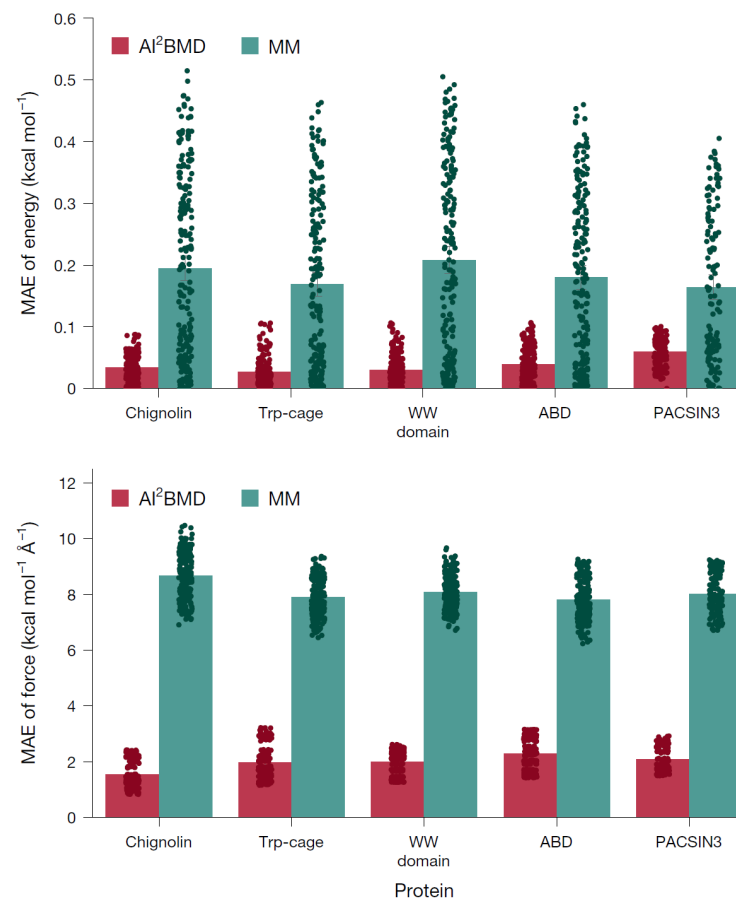


Each protein has 200 conformations: 50 folded structures, 50 unfolded structures and 100 intermediate structures.

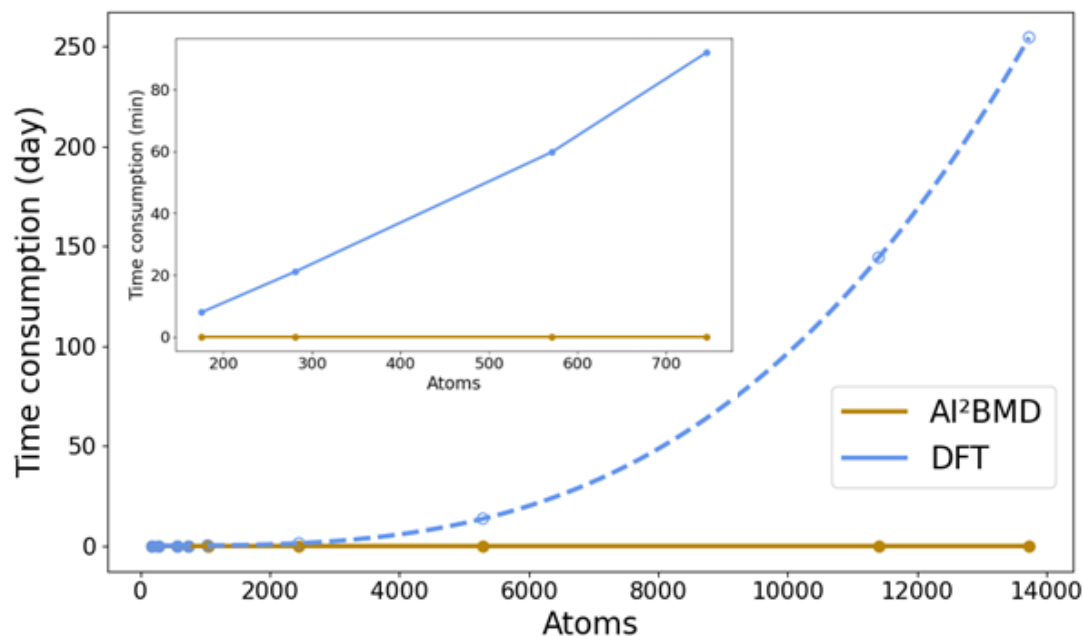
Metric: mean absolute errors (MAE) of potential energy (per atom) and atomic force

Reference: DFT calculation (M06-2X, 6-31g*)

Comparison: Amber FF19SB



AI²BMD: efficiency



- 6 orders less time consumption compared to DFT
- 2.6 seconds (AI²BMD) vs. 254 days (DFT estimation) for a large protein (> 13K atoms)

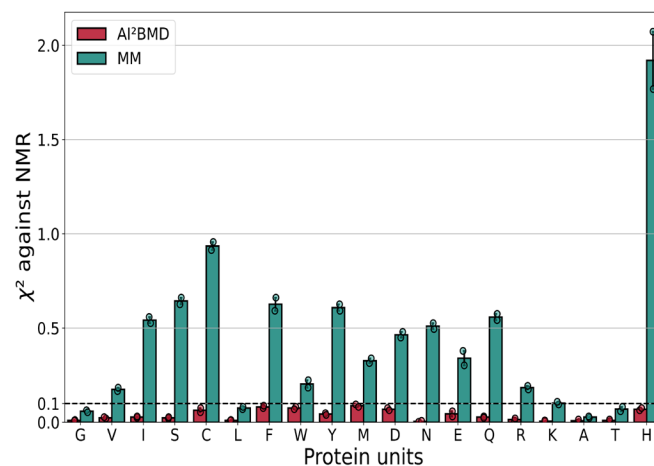
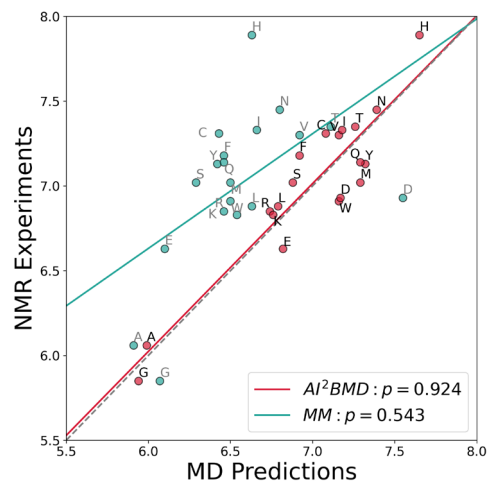
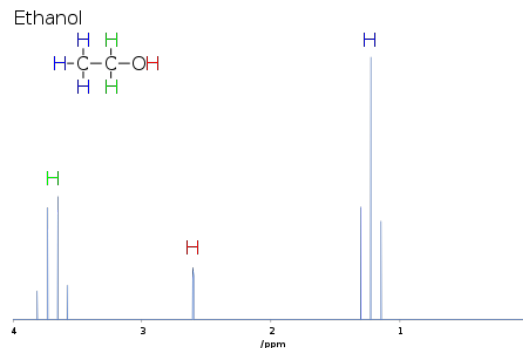
Protein	Atom number of protein	Atom number of system	AI ² BMD (s)	DPMD (s)	Allegro (s)
Chignolin	175	4,715	0.047	0.040	0.238
Trp-cage	281	6,067	0.052	0.055	0.322
WW domain	571	10,678	0.070	0.095	0.626
ABD	746	11,793	0.085	0.106	0.712
PACSIN 3	1,040	17,923	0.106	0.162	-
SSO0941	2,450	44,401	0.213	0.414	-
APC	5,292	54,999	0.449	0.580	-
Polyphosphate Kinase	11,404	97,657	0.966	-	-

Comparison with other MD simulation program

AI-based simulation applications

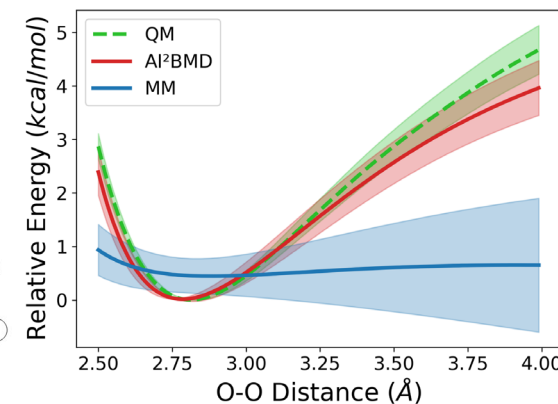
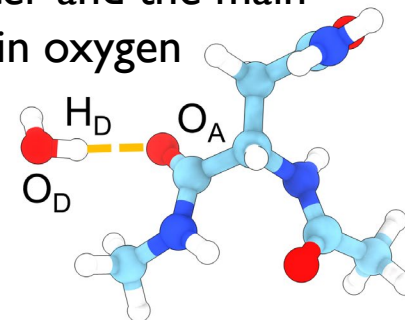
J coupling

An indirect interaction between two nuclear spins that arises from hyperfine interactions between the nuclei and local electrons

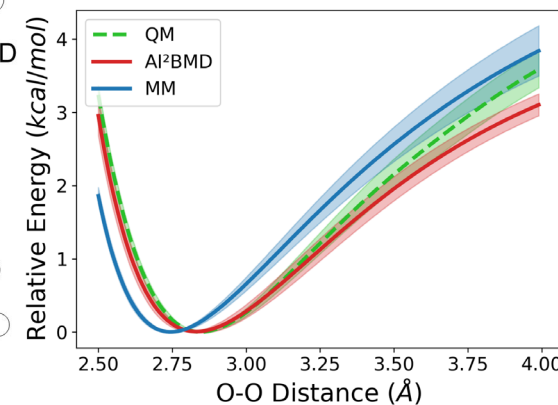
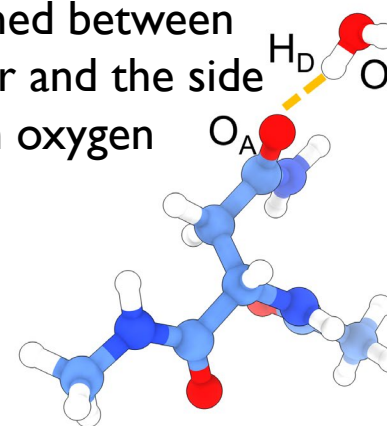


Hydrogen bond energy

Formed between water and the main chain oxygen

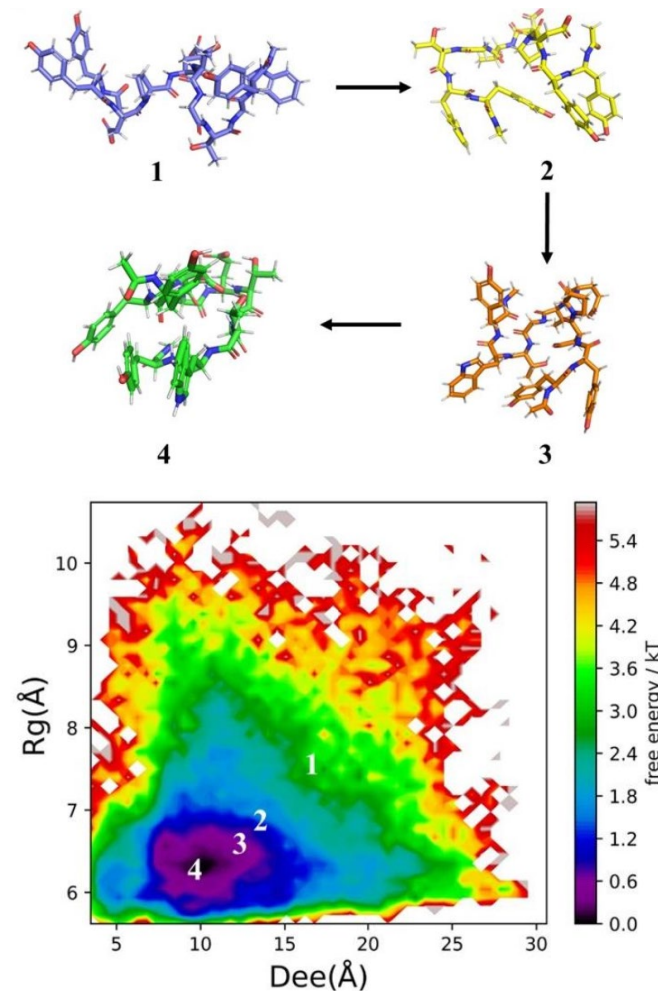
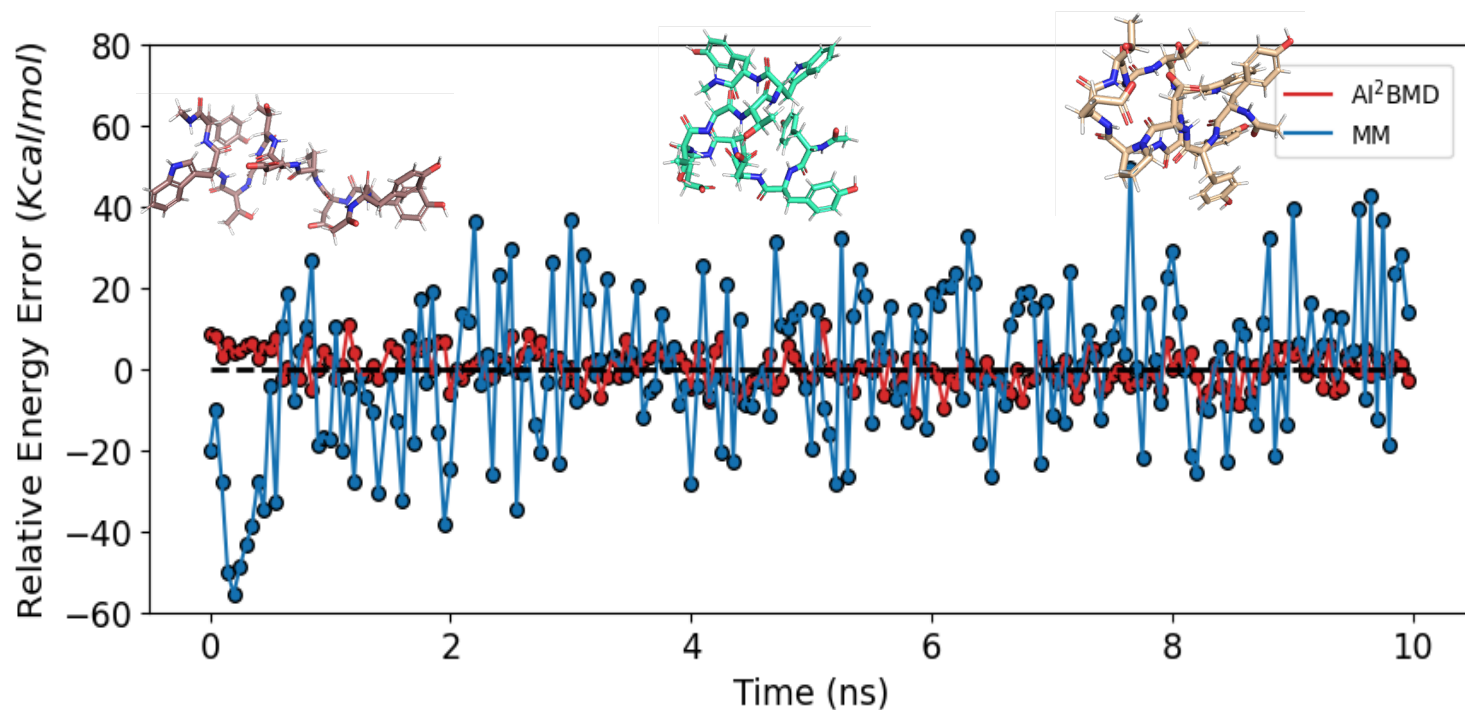


Formed between water and the side chain oxygen

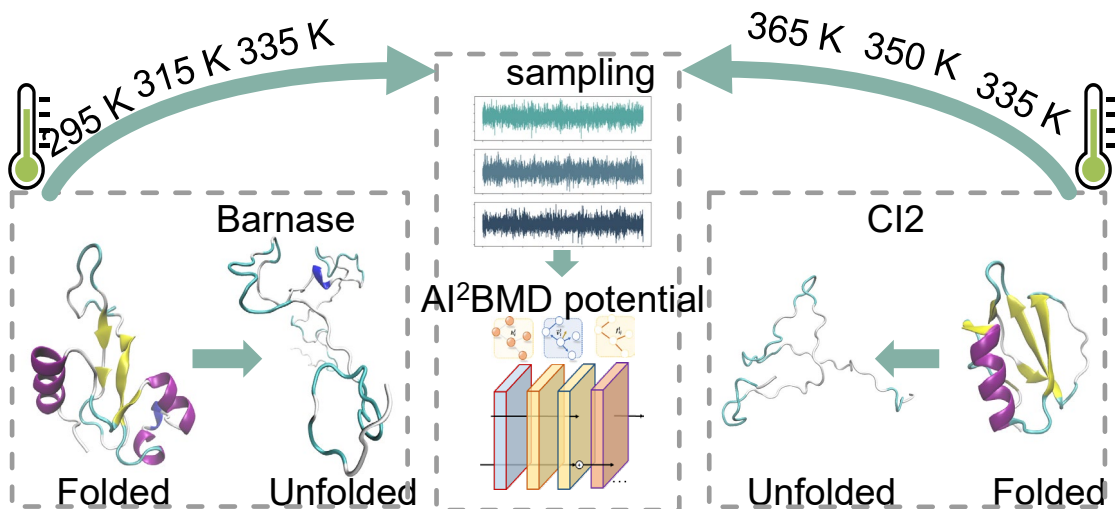


AI-based simulation applications

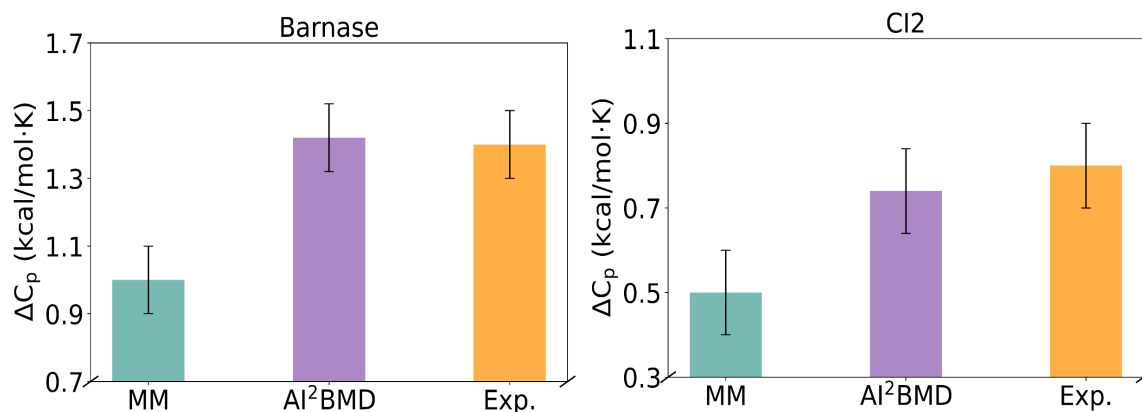
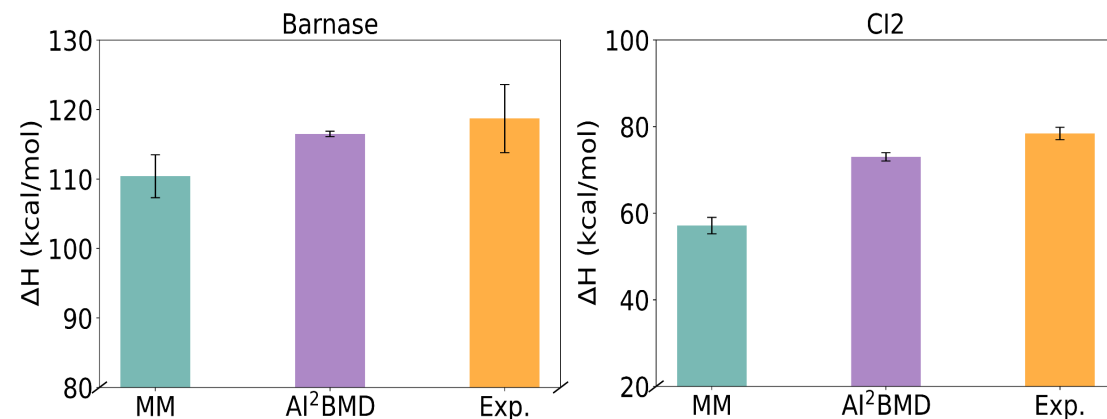
Protein folding (Chignolin 10ns (10 million simulation steps) simulation in explicit solvent)



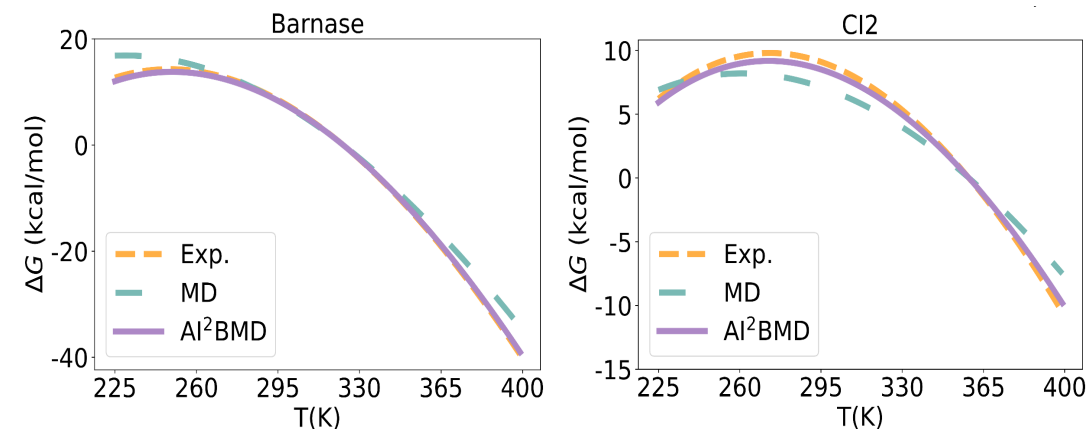
AI-based simulation applications



Calculation pipeline



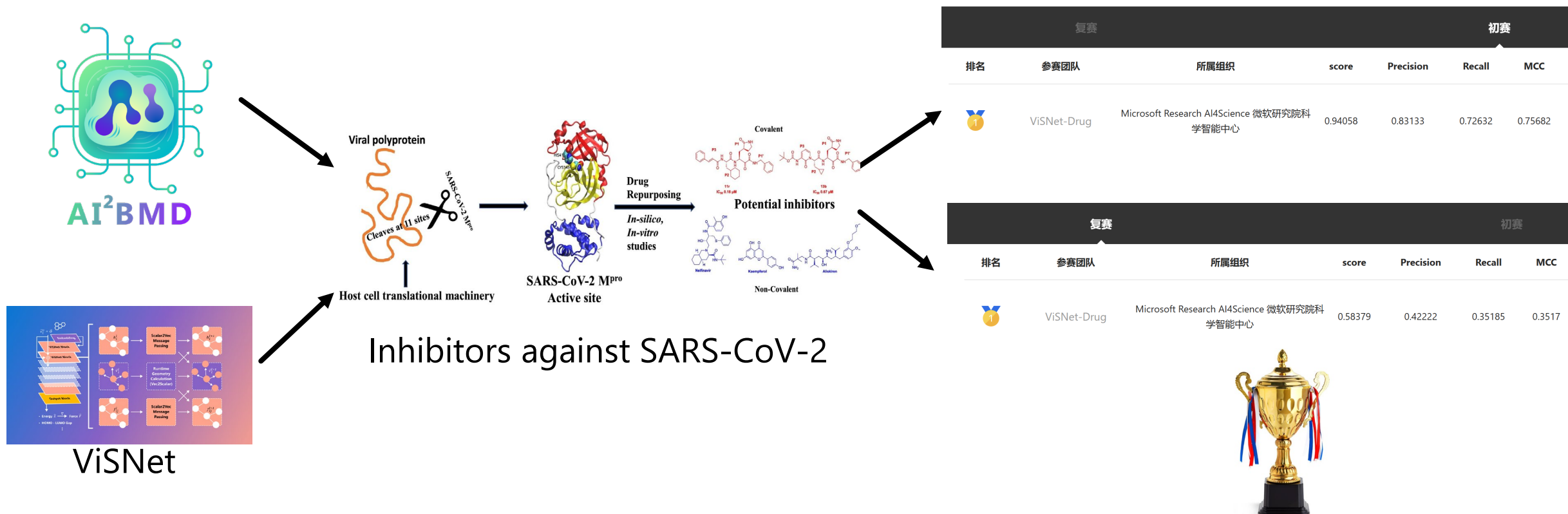
Heat capacity upon protein unfolding



AI-based simulation applications

Drug repurposing in the First Global AI Drug Development Competition

- Predict whether a drug candidate bind to and act as the inhibitor against M^{pro} of SARS-CoV-2
- 1,105 teams around the world took part in the competition

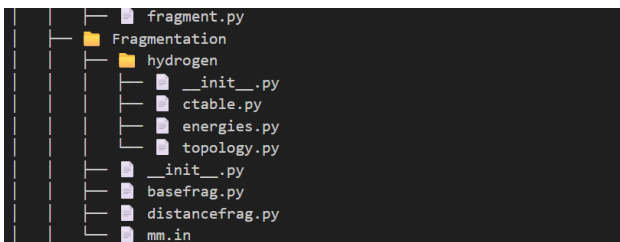


AI²BMD code overview

```
AI2BMD
├── examples
│   ├── chig_preprocessed
│   │   ├── chig_preeq-nowat.pdb
│   │   └── chig_preeq.pdb
│   ├── abd.pdb
│   ├── chig.pdb
│   ├── trpcage.pdb
│   └── ww.pdb
├── src
│   ├── AIMD
│   │   ├── __init__.py
│   │   ├── arguments.py
│   │   ├── fragment.py
│   │   ├── geometry_optimization.py
│   │   ├── preprocess.py
│   │   ├── protein.py
│   │   └── simulator.py
│   ├── Calculators
│   │   ├── __init__.py
│   │   ├── async_utils.py
│   │   ├── bonded.py
│   │   ├── calculator.py
│   │   ├── combiner.py
│   │   ├── device_strategy.py
│   │   ├── fragment.py
│   │   ├── nonbonded.py
│   │   ├── pme.py
│   │   ├── qmmm.py
│   │   ├── tinkert_async.py
│   │   └── visnet_calculator.py
│   ├── Fragmentation
│   │   ├── hydrogen
│   │   │   ├── __init__.py
│   │   │   ├── ctable.py
│   │   │   ├── energies.py
│   │   │   └── topology.py
│   │   ├── __init__.py
│   │   ├── basefrag.py
│   │   ├── distancefrag.py
│   │   └── mm.in
│   ├── VisNet
│   │   ├── checkpoints
│   │   │   ├── visnet-uni-2ef43f29ec78fa5fef0b3de832bfada9.ckpt
│   │   │   └── visnet-uni-de11d1421ccda37ffab07d7403c8f5bb.ckpt
│   │   ├── model
│   │   │   ├── __init__.py
│   │   │   ├── output_modules.py
│   │   │   ├── priors.py
│   │   │   ├── utils.py
│   │   │   ├── visnet.py
│   │   │   ├── visnet_block.py
│   │   │   └── __init__.py
│   │   └── main.py
├── ai2bmd.sif
└── run.sh
```

- Fragmentation
- Calculators
- Model
- Simulation
- Geometry Optimization

AI²BMD Fragmentation



```
class BaseFragment(ABC):
    """
    Basic fragment Object.
    """

    def __init__(self) -> None:
        pass

    @abstractmethod
    def fragment(self, *args, **kwargs):
        """
        The basic fragment method. Fragment the protein into specific
        sub-structures (e.g. to generate dipeptides and ACE-NMEs).
        Please rewrite this function in subclass according to your specific
        fragment method.
        """
        pass

class DipeptideFragment(BaseFragment):
    """
    A subclass of BaseFragment to generate dipeptides and ACE-NMEs. All the
    dipeptides and ACE-NMEs are generated by the same method, i.e., @method
    get_fragment_index, but with different postprocessing in @method fragment.
    The @method fragment must invoke @method get_fragment_index.
    """

    def __init__(self) -> None:
        super().__init__()

    @abstractmethod
    def get_fragments(self, prot: Protein) -> FragmentData:
        pass

    @staticmethod
    def get_fragments_index(prot: Atoms) -> tuple[list[int], list[int]]:
        """
        Fragment the protein into dipeptides/ACE-NMEs and return their indices
        in the original protein.

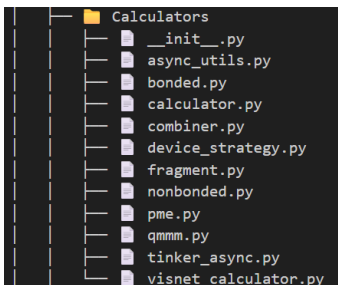
        Parameters:
        -----
        prot: Atoms
            The protein to be fragmented.

        Returns:
        -----
        """
```

basefrag.py

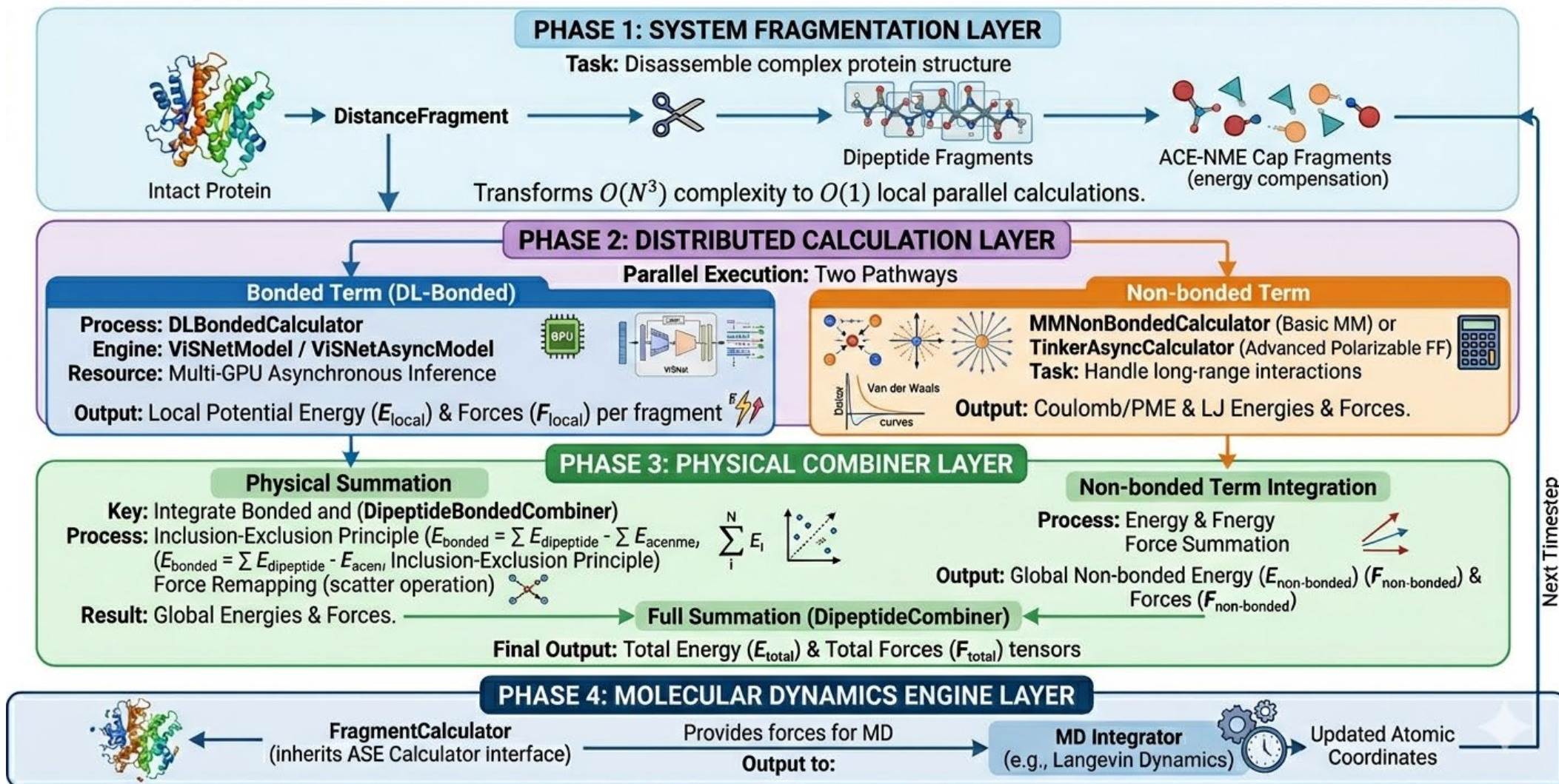
- **Protein Fragmentation:** Deconstructs large protein chains into Dipeptides and ACE-NMEs units
- **Overlapping Strategy:** Ensures physical connectivity by sharing key backbone atoms (CA, C, N) across adjacent fragments.
- **Index Mapping:** Efficiently maps global atom indices from the original ASE.Atoms object to specific sub-structures.
- **Topological Maintenance:** Correctly inserts sidechain atoms into the backbone sequence to preserve the chemical identity of each residue.

AI²BMD Calculators



AI2BMD: CORE ARCHITECTURE & WORKFLOW

AI2BMD Core Architecture
Divide-and-Conquer Hybrid Simulation



AI²BMD Calculators

```
class FragmentCalculator(Calculator):
    """
    FragmentCalculator is a universal calculator for dipeptide fragments,
    including Bonded Calculator, QM/DL Calculator, and MM (Non-Bonded) Calculator.

    Parameters:
    -----
    properties: list[str]
        Targets of the calculation.
    """

    implemented_properties = ["energy", "forces"]

    def __init__(self, is_root_calc, nbcalc_type, **kwargs):
        super().__init__(**kwargs)

        self.is_root_calc = is_root_calc

        if self.is_root_calc is True:
            self.work_queue = WorkQueue()

        # * set bonded calculator
        self.bonded_calculator = DLBondedCalculator(**kwargs)

        # * set non-bonded calculator
        device = DeviceStrategy.get_non_bonded_device()
        self.nonbonded_calculator = nonbonded_calcs[nbcalc_type](device=device)

        # * set fragment strategy
        self.concurrent = DeviceStrategy.fragment_strategy()

        # * set combiner
        self.combiner = DipeptideCombiner()

    def calculate(self, atoms, properties, system_changes):
        if self.is_root_calc:
            Calculator.calculate(self, atoms, properties, system_changes)
            self.work_queue.drain()

        f_args = [
            (self.bonded_calculator, atoms),
            (self.nonbonded_calculator, atoms),
        ]

        energy, forces = zip(*execution_wrapper(f_args, self.concurrent))

        energy = self.combiner.energy_combine(*energy)
        forces = self.combiner.forces_combine(*forces)

        self.results = {
            "energy": energy,
            "forces": forces,
        }
```

fragment.py

- **Universal Orchestrator:** Integrates multiple computational backends, including deep learning-based bonded calculators and classical MM/PME non-bonded calculators.
- **Hybrid Energy Scoring:** Combines local quantum-level accuracy (via fragments) with long-range classical electrostatics.
- **Result Consolidation:** Combine energy and force contributions from overlapping dipeptide fragments back into a global molecular state.

```
class DipeptideBondedCombiner:
    """
    Combine the energies and forces of the dipeptides and ACE-NMEs.
    """

    @staticmethod
    def energy_combine(
        dipeptides_energies: torch.Tensor,
        acenmes_energies: torch.Tensor,
    ) -> np.ndarray:
        """
        Combine the energies of the dipeptides and ACE-NMEs.
        """
        energy = torch.sum(dipeptides_energies) - torch.sum(acenmes_energies)

        return energy.detach().cpu().numpy()

    @staticmethod
    def forces_combine(
        num_atoms: int,
        dipeptides_forces: torch.Tensor,
        acenmes_forces: torch.Tensor,
        select_index: torch.Tensor,
        origin_index: torch.Tensor,
    ) -> np.ndarray:
        """
        Combine the forces of the dipeptides and ACE-NMEs, removing the extra
        forces that correspond to the added hydrogens, and assigning the
        selected forces to their original indices in the protein.
        """

        # negate ACE-NME forces
        forces = torch.cat([dipeptides_forces, -acenmes_forces])[select_index]
        forces = torch_scatter.scatter(forces, origin_index, dim=0, dim_size=num_atoms, reduce="sum")

        return forces.detach().cpu().numpy()

class DipeptideCombiner:
    @staticmethod
    def energy_combine(
        bonded_energy: np.float32, nonbonded_energy: np.float32
    ) -> np.float32:
        return bonded_energy + nonbonded_energy

    @staticmethod
    def forces_combine(
        bonded_forces: np.ndarray, nonbonded_forces: np.ndarray
    ) -> np.ndarray:
```

combiner.py

- **Fragmentation Correction:** Implements an inclusion-exclusion principle ($E_{\text{dipeptide}} - E_{\text{cap}}$) to correct for overlapping regions created during protein fragmentation.
- **Force Remapping:** Projects localized forces from multiple sub-fragments back onto the global coordinates of the original protein using scatter operations.
- **Physics Integration:** Combine Bonded (DL-based) and Non-bonded (MM-based) contributions.

AI²BMD Calculators

```
class DLBonedCalculator:
    """
    DLBonedCalculator is a dipeptide bonded calculator based on
    DL calculations supported by ViSNet.
    """

    def __init__(
        self,
        ckpt_path: str,
        ckpt_type: str,
        **kwargs,
    ) -> None:
        self.models: list[ViSNetModelLike] = []
        self.ckpt_path = ckpt_path
        self.ckpt_type = ckpt_type

        # * set fragment method and combiner
        self.fragment_method = DistanceFragment()
        self.combiner = DipeptideBondedCombiner()

        print("Loading models...")
        model_path = osp.join(self.ckpt_path, f"visnet-uni-{self.ckpt_type}.ckpt")
        self.models = [
            get_visnet_model(model_path, device)
            for device in DeviceStrategy.get_bonded_devices()
        ]

    def _inference_impl(
        self, data: list[FragmentData], model: ViSNetModelLike
    ) -> tuple[list[np.ndarray], list[np.ndarray]]:
        return zip(*[model.dl_potential_loader(unit) for unit in data])

    def calculate(
        self, fragments: FragmentData
    ) -> tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray]:
        """
        Calculate the energy and forces of the dipeptide.
        The basic function of DLBonedCalculator.

        Parameters:
        -----
            fragments: FragmentData
                combined dipeptide and ACE-NME fragments.
        """

        # retrieve devices and work assignment
        devices = DeviceStrategy.get_bonded_devices()
        work = DeviceStrategy.get_work_partitions()

        # work execution
        n_devices = len(devices)
        partitions = [[] for _ in range(n_devices)]
        for idx, start, end in work:
            partitions[idx].append(fragments[start:end])
```

bonded.py

- **Automated Workflow:** Encapsulates the entire "fragment-calculate-combine" cycle, making deep learning calculations seamless for the user.
- **Result Integration:** Transform per-fragment outputs into global protein energy and force tensors.
- **Hardware Efficiency:** Dynamically manages device assignments and work partitioning to maximize available computational resources.

```
class MMNonBondedCalculator:
    """
    MMNonBondedCalculator is a non-bonded calculator based on MM calculations.
    """

    def __init__(self, device="cpu") -> None:
        super().__init__()
        self.device = device
        self.k = 1 / (4 * pi * _eps0) * 10e6 * mol * C ** (-2)
        self.sigmas = None
        self.epsilon = None
        self.charges = None
        self.src = None
        self.dst = None

    def set_parameters(self, prot: Protein) -> None:
        self.sigmas = torch.tensor(prot.sigmas, dtype=torch.float, device=self.device)
        self.epsilon = torch.tensor(prot.epsilon, dtype=torch.float, device=self.device)
        self.charges = torch.tensor(prot.charges, dtype=torch.float, device=self.device)

        src, dst = prot.initial_mm_adjmatrix()
        self.src = src.to(self.device)
        self.dst = dst.to(self.device)

    def __call__(self, prot: Protein) -> tuple[np.float32, np.ndarray]:
        """
        Using non-bonded atom pairs to calculate the non-bonded energy and
        force. Non-bonded forces are calculated by calculating the gradient of
        the non-bonded energy with respect to the atom positions.
        """
        pos = torch.tensor(prot.get_positions(), dtype=torch.float, device=self.device)

        vec = pos[self.dst] - pos[self.src]
        d2 = (vec**2).sum(-1)
        d = torch.sqrt(d2)

        # LJ
        sigma_ij = 0.5 * (self.sigmas[self.src] + self.sigmas[self.dst]) * nm
        epsilon_ij = torch.sqrt(self.epsilon[self.src] * self.epsilon[self.dst])
        c6 = (sigma_ij**2 / d2) ** 3
        c12 = c6**2
        energy_lj = 4 * epsilon_ij * (c12 - c6)
        force_lj = (24 * epsilon_ij * (2 * c12 - c6) / d2).unsqueeze(-1) * vec

        # Coulomb
        energy_coulomb = self.k * self.charges[self.src] * self.charges[self.dst] / d
        force_coulomb = (energy_coulomb / d2).unsqueeze(-1) * vec

        # Combine
        energy = energy_lj.sum() + energy_coulomb.sum()
        force = force_lj + force_coulomb
        force = scatter_add(force, self.dst, dim=0, dim_size=len(prot))
        energy = energy.cpu().item() * (kJ / mol) / 2
        force = force.cpu().numpy() * (kJ / mol)

        return energy, force
```

nonbonded.py

- **Classical Force Field Integration:** Implements Lennard-Jones and Coulomb potentials to account for long-range van der Waals and electrostatic interactions.
- **Adjacency-Based Selection:** Calculates interactions only for valid non-bonded atom pairs defined by the protein's topology.
- **Force Aggregation:** Utilizes efficient scatter operations to sum individual pair-wise forces into a global force tensor for the entire protein system.

```
class ViNetModel:
    """
    Calculate the energy and forces of the system
    using deep learning model, i.e., ViNet.

    Parameters:
    -----
    model:
        Deep learning model.
    device: cpu | cuda
        Device to use for calculation.
    """
    implemented_properties = ["energy", "forces"]

    def __init__(self, model, device="cpu"):
        self.model = model
        self.model.eval()
        self.device = device
        self.stream = (
            torch.cuda.Stream(device=device)
            if device.startswith('cuda')
            else None
        )
        self.model.to(self.device)

    def collate(self, frag: FragmentData):
        z = numpy_to_torch(frag.z, self.device)
        pos = numpy_to_torch(frag.pos, self.device)
        batch = numpy_to_torch(frag.batch, self.device)

        return dict(z=z, pos=pos, batch=batch)

    def dl_potential_loader(self, frag_data: FragmentData):
        # note: this context is a no-op if self.stream is None
        with torch.cuda.stream(self.stream):
            with torch.set_grad_enabled(True):
                e, f = self.model(self.collate(frag_data))

                e = e.detach().cpu().reshape(-1, 1).numpy()
                f = f.detach().cpu().reshape(-1, 3).numpy()

        return e, f

    @classmethod
    def from_file(cls, **kwargs):
        if "model_path" not in kwargs:
            raise ValueError("model_path must be provided")

        model_path = kwargs["model_path"]
        device = kwargs.get("device", "cpu")

        model = load_model(model_path)
        out = cls(model, device=device)
        return out
```

visnet_calculator.py

- **Model Wrapping:** Encapsulates ViNet with native PyTorch optimizations like CUDA streams for low-latency inference.
- **Process-level Parallelism:** Implements an AsyncModel proxy using subprocesses and Socket communication to bypass Python's GIL and maximize multi-GPU throughput.
- **ASE Integration:** Provides a standard ViNetCalculator to make deep learning potentials compatible with the ASE ecosystem.
- **Resource Management:** Features a factory pattern (get_visnet_model) to manage model instances across different devices efficiently.

```
class TinkerAsyncCalculator(Calculator):
    implemented_properties = ["energy", "forces"]

    def __init__(self, pdb_file, utils_dir, devices: list[str], **kwargs):
        super().__init__(**kwargs)
        self.pdb_file = os.path.abspath(pdb_file)
        self.utils_dir = utils_dir
        self.prot_name = os.path.basename(pdb_file)[:4]
        self.devices = devices
        self._tinker_proc = None
        self.atoms = Atoms

        global _tinker_instance_id
        self.instance_id = _tinker_instance_id
        _tinker_instance_id += 1

        self.server = AsyncServer("tinker")
        self.logger = get_logger(f"Tinker-Proxy-{self.instance_id}")

        if any(map(lambda x: x.startswith('cuda'), devices)):
            self.command_dir = '/usr/local/gpu-m'
        else:
            self.command_dir = '/usr/local/cpu-m'
        atexit.register(self._shutdown)

    def _start_tinker(self):
        self.logger.debug("Initializing tinker...")
        self._write_key(self.atoms)
        self.logger.debug('Key file written!')
        if not os.path.exists(f'{self.prot_name}.xyz'):
            self._generate_xyz_template()
            self.logger.debug('XYZ template generated!')

        # bind devices
        envs = os.environ.copy()
        gpus = []
        for device in self.devices:
            if device.startswith("cuda"):
                _, nr = device.split(':')
                gpus.append(nr)

        if len(gpus) > 1:
            assert self.instance_id < len(gpus)
            envs["CUDA_VISIBLE_DEVICES"] = gpus[self.instance_id]
        elif len(gpus) == 1:
            envs["CUDA_VISIBLE_DEVICES"] = gpus[0]
```

tinker_async.py

Use tinker for solvent simulation

```
class BaseSimulator(ABC):
    def __init__(
        self, prot: Protein, log_path: str, preeq_steps: int = 200, temp_k: int = 300
    ) -> None:
        self.prot = prot
        self.log_path = log_path
        self.simulation_save_path = os.path.join(log_path, "SimulationResults")
        os.makedirs(self.simulation_save_path, exist_ok=True)
        self.nowat_pdb = self.prot.nowater_PDB_path
        self.preeq_steps = preeq_steps
        self.prot.set_pbc(True)
        self.temp_k = temp_k

    def get_qm_idx(self):
        return list(range(len(read_protein(self.nowat_pdb))))

    def need_fragmentation(self):
        return isinstance(self.qmcalc, FragmentCalculator)

    def initialize_fragcalc(self):
        if self.need_fragmentation():
            self.qmcalc.bonded_calculator.fragment_method.fragment(self.qmatoms)
            self.qmcalc.nonbonded_calculator.set_parameters(self.qmatoms)

            start, end = self.qmatoms.fragments_start, self.qmatoms.fragments_end
        else:
            start, end = [0], [len(self.qmatoms)]

        # set work partitions based on dipeptides/ACE-NMEs
        DeviceStrategy.set_work_partitions(start, end)

    def set_calculator(self, **kwargs) -> None:
        os.chdir(self.simulation_save_path)
        self.make_calculator(**kwargs)
        self.initialize_fragcalc()

    @abstractmethod
    def make_calculator(self, **kwargs) -> Calculator:
        pass

    def make_fragment_calculator(self, is_root_calc: bool, **kwargs) -> FragmentCalculator:
        mode = arguments.get().mode
        if mode == "fragment":
            return FragmentCalculator(is_root_calc=is_root_calc, **kwargs)
        if mode == "visnet":
            return ViSNetCalculator(is_root_calc=is_root_calc, **kwargs)

    def simulate(
        self, prot_name: str, simulation_steps: int, time_step: float,
        record_per_steps: int, hydrogen_constraints: bool,
        seed: int, restart: bool, build_frames: bool
    ):
        restart_traj_path = os.path.join(self.log_path, f"{prot_name}-traj.traj")
        if restart:
```

simulator.py

- Modular Architecture:**
 The system uses a base class to handle both SolventSimulator and NoSolventSimulator .
- Divide-and-Conquer:**
 Break large proteins into overlapping dipeptide fragments, transforming complex $O(N^3)$ calculations into efficient, parallel $O(1)$ tasks.
- Hybrid Engines:**
 Integrates ViSNet with Tinker9 through an asynchronous, multi-GPU processing strategy.

```
#Simulation task specific arguments
parser.add_argument(
    "--temp-k",
    type=int,
    default=300,
    help="Simulation temperature in Kelvin",
)
parser.add_argument(
    "--timestep",
    type=float,
    default=1,
    help="TimeStep (fs) for simulation",
)
parser.add_argument(
    "--sim-steps",
    type=int,
    default=1000,
    help="Simulation steps for simulation",
)
parser.add_argument(
    "--preeq-steps",
    type=int,
    default=2000,
    help="Pre-equilibration simulation steps for each constraint",
)
parser.add_argument(
    "--max-cyc",
    type=int,
    default=100,
    help="Maximum energy minimization cycles in preprocessing",
)
parser.add_argument(
    "--constraints",
    action=argparse.BooleanOptionalAction,
    default=False,
    help="Constrain hydrogen bonds",
)
parser.add_argument(
    "--solvent",
    action=argparse.BooleanOptionalAction,
    default=False,
    help="Use solvent or not",
)
parser.add_argument(
    "--write-solvent",
    action=argparse.BooleanOptionalAction,
    default=False,
    help="Write coordinates of solvent atoms in output",
)
parser.add_argument(
    "--mm-method",
    type=str,
    default="tinker-GPU",
    choices=["tinker", "tinker-GPU"],
    help="MM calculator for the nonbonded energy",
)
parser.add_argument(
    "--seed",
    type=int,
    default=0,
    help="Random seed for simulation",
```

Simulation arguments

command:

```
#以chignolin为例进行模拟
python ./src/main.py \
    --prot-file ./examples/chig.pdb \
    --task simulation \
    --base-dir ./simulation/chig \
    --sim-steps 1000 \
    --temp-k 300 \
    --timestep 1 \
    --preeq-steps 0 \
    --record-per-steps 1
```

AI²BMD Geometry Optimization

```
class AI2BMDOptimizer:
    def __init__(self, atoms, fmax=0.015, max_cycles=100, max_step_size=0.1):
        self.atoms = atoms
        self.fmax = fmax
        self.max_cycles=max_cycles
        self.max_step_size = max_step_size
        self.traj_file = "optimization_trace.xyz"
        self.best_file = "best_structure.xyz"

    def run(self):
        all_best_f = 999.0
        for cycle in range(self.max_cycles):
            print(f"\n🚀 [Cycle {cycle+1}] 启动精细优化 (BFGS)...")

            curr_max_step_size = self.max_step_size
            if cycle >= 0:
                opt = LBFGS([self.atoms, memory=20, alpha=70.0, maxstep=curr_max_step_size])
            else:
                opt = BFGSLineSearch(self.atoms, alpha=40.0, maxstep=curr_max_step_size)

            stagnant_steps = 0
            best_f = 999.0

            for step in opt.irun(fmax=self.fmax, steps=100):
                write(self.traj_file, self.atoms, append=True)
                curr_f = self.cal_fmax()

            if curr_f < best_f:
                best_f = curr_f
                curr_max_step_size = curr_max_step_size/2
                print(f"💡更新了局部极小值(fmax={best_f:.4f})")
                if best_f < all_best_f:
                    all_best_f = best_f
                    #curr_max_step_size = curr_max_step_size/10
                    write(self.best_file, self.atoms)
                    print(f"💡更新了整体极小值(fmax={all_best_f:.4f})")

            # 检查是否进入平台期
            if (curr_f-best_f) > 1e-6:
                stagnant_steps += 1
            else:
                stagnant_steps = 0
```

command:

```
#以chignolin为例进行几何优化
python ./src/main.py \
    --prot-file ./examples/chig.pdb \
    --task optimization \
    --base-dir ./optimization/chig \
    --fmax 0.005 \
    --max-cycles 100 \
    --max-step-size 0.1
```

geometry_optimization.py

- **Adaptive Optimization:** Combines **LBFGS** for high-precision local convergence with a custom **SGD-based "shake-and-push"** mechanism to escape local minima.
- **Parallel Acceleration:** Fully supports the **AI2BMD fragmentation** scheme, distributing fragment calculations across multiple GPUs/CPU.
- **Stability Mechanisms:** Implements dynamic step-size scaling and coordinate reconstruction to handle high-force regions safely.
- **Post-Optimization Analysis:** Integrated with **ASE Vibrations** to perform Hessian calculations and frequency analysis upon convergence.

```
#Optimization task specific arguments
parser.add_argument(
    "--fmax",
    type=float,
    default=0.015,
    help="Convergence threshold of RMS gradient.",
)
parser.add_argument(
    "--max-cycles",
    type=int,
    default=10,
    help="Max cycles of optimization.",
)
parser.add_argument(
    "--max-step-size",
    type=float,
    default=0.05,
    help="Max size of a sing optimization step.",
)
parser.add_argument(
    "--vib",
    action=argparse.BooleanOptionalAction,
    default=False,
    help="Do vibrations.",
)
parser.add_argument(
    "--sequence-dict-path",
    type=str,
    default='NONE',
    help="The path to sequence-dict for fragment.",
)
```

Geometry optimization arguments

Life is nothing but an electron looking for a place to rest.

Albert SzentGyörgyi

