π-Stack Optimizer

User Guide and Technical Documentation

Arunima Ghosh, Susmita Barik, Roshan J. Singh and Sandeep K. Reddy

Version 1.0 – December 23, 2025

Abstract

The π**-Stack Optimizer** is a high-performance computational framework for discovering energetically favorable stacking configurations in molecular systems. Leveraging semi-empirical quantum chemistry (xTB) coupled with multiple global optimization algorithms (PSO, GA, GWO, PSO+Nelder-Mead), this tool provides researchers with a flexible, efficient, and reproducible workflow for π-stacking studies. The system features parallel energy evaluations, automatic symmetry detection, and comprehensive logging capabilities, making it suitable for both exploratory research and production computational chemistry pipelines.

Contents

Introduction

Pending note Verify SMILES-to-XYZ conversions produce the intended side-chain conformers before dropping monomers into this workflow, because any mistaken geometry is carried through unchanged.

The π-Stack Optimizer is a computational tool for finding the most stable arrangements of molecules stacked in one-dimensional columns. When molecules stack together, their relative positions—how far apart they are, how they're rotated, and how they're shifted—determine the stability of the assembly. Finding the best arrangement requires exploring many possible configurations and evaluating their energies, which becomes computationally expensive as the number of possible arrangements grows.

This tool automates the search for optimal stacking configurations by combining quantum chemistry calculations (xTB) with optimization algorithms. Starting from a single monomer structure, the tool systematically explores different stacking arrangements, evaluates their energies, and identifies the most stable configuration. The tool is designed for computational chemists and materials scientists who need to generate realistic starting structures for higher-level calculations, explore how stacking arrangements affect material properties, or study how molecular flexibility influences stacking stability.

The optimization explores a $(7 + N_{\text{torsion}})$-dimensional parameter space that controls how molecules are positioned in the stack:

$$\mathbf{x} = [c_\theta, s_\theta, T_x, T_y, T_z, C_x, C_y, \tau_1, \tau_2, \ldots, \tau_{N_{\text{torsion}}}] \in \mathbb{R}^{7+N_{\text{torsion}}} \tag{1}$$

The first 7 parameters control the rigid-body transformation (rotation, translation, and rotation center), while the remaining parameters control internal molecular flexibility through dihedral torsions. The optimizer finds parameter values that minimize the binding energy between molecules while avoiding unphysical overlaps.

The tool provides several key capabilities to make this optimization efficient and practical:

- Choose from Particle Swarm Optimization (PSO), Genetic Algorithm (GA), Grey Wolf Optimizer (GWO), or a PSO+Nelder-Mead hybrid, each suited to different problem characteristics.
- Uses a worker-pool architecture to run multiple quantum chemistry calculations simultaneously, significantly reducing computation time.
- Supports internal dihedral torsions, allowing flexible parts of molecules to rotate and adapt their shape to optimize stacking interactions.
- Identifies equivalent torsions in symmetric molecules, reducing the search space and computational cost without losing accuracy.
- Provides structured console output, trajectory files, and detailed result summaries for analysis and visualization.
- Clean separation between optimization, geometry handling, and energy evaluation, making it easy to extend or modify components.

Getting Started

This section will get you up and running with the π-Stack Optimizer in minutes.

Installation

Step 1: Clone the Repository

```
git clone https://github.com/sandeepgroup/pi-stack-optimizer.git
cd pi-stack-optimizer
```

Step 2: Create Python Virtual Environment

```
python3 -m venv .venv
source .venv/bin/activate
```

Step 3: Install Python Dependencies

If a requirements.txt file is included:

```
pip install --upgrade pip
pip install -r requirements.txt
```

Otherwise, install NumPy manually:

```
pip install numpy
```

Step 4: Enable CLI Wrappers (Recommended)

Source the helper script to expose the convenience launchers and ensure the repository is on your PYTHONPATH:

```
source ./activate_pi_stack.sh
pi-stack-generator --help    # Wrapper for pi-stack-generator.py
pi-hyperopt --help           # Wrapper for hyperparameter-opt/hyperopt.py
```

This step sets up shell functions so you can run the tools without typing python path/to/script.py. If you skip it, prefix the commands with python and provide the relative script path manually.

Step 5: Configure xTB Backend

Verify xTB is accessible:

```
which xtb
xtb --version
```

Set environment variables if needed:

```
export XTBHOME=/path/to/xtb
export OMP_NUM_THREADS=4
```

Step 6: Validate Installation

```
pi-stack-generator example.xyz --workers 1 --optimizer pso \
    --max-iters 1 --swarm-size 4
```

If successful, you should see initialization messages and a single optimization iteration. When not using the activation script, replace pi-stack-generator with python pi-stack-generator.py.

Quick Start: Your First Optimization

Let's optimize a simple benzene dimer geometry using PSO:

```
pi-stack-generator benzene.xyz --optimizer pso --workers 4 \
    --swarm-size 30 --max-iters 100
```

This assumes you already sourced activate_pi_stack.sh. Without it, run python pi-stack-generator.py ... instead.

extbfWhat happens:

1. The tool reads benzene.xyz and validates the geometry
2. Initializes 30 particles with random stacking parameters
3. Spawns 4 parallel xTB worker processes
4. Runs 100 PSO iterations (or until convergence)
5. Outputs optimized geometry to optimized_monomer_*.xyz
6. Saves full results to optimization_results.txt

Output files:

- optimized_monomer_*.xyz – Best monomer geometry
- optimized_stack_*.xyz – 10-layer demonstration stack
- optimization_results.txt – Detailed results with energies and parameters

Basic Usage

Input File Format

The primary input is a standard XYZ coordinate file:

```
12
Benzene monomer
C     0.000     1.396     0.000
C     1.209     0.698     0.000
C     1.209    -0.698     0.000
C     0.000    -1.396     0.000
C    -1.209    -0.698     0.000
C    -1.209     0.698     0.000
H     0.000     2.479     0.000
H     2.147     1.240     0.000
H     2.147    -1.240     0.000
H     0.000    -2.479     0.000
H    -2.147    -1.240     0.000
H    -2.147     1.240     0.000
```

Listing 1: Example benzene.xyz file

Common Workflows

Basic Rigid Monomer Optimization

No torsional flexibility; optimize only translation and rotation:

```
python main.py molecule.xyz --optimizer pso \
    --swarm-size 50 --max-iters 200 --workers 8
```

Optimization with Torsional Flexibility

Provide a torsions.json file defining rotatable bonds:

```
python main.py molecule.xyz --optimizer pso \
    --torsions-file torsions.json \
    --swarm-size 60 --max-iters 300
```

Example torsions.json:

```
{
  "indexing": "0-based",
  "torsions": [
    {
      "name": "phenyl_rotation",
      "atoms": [2, 5, 8, 11],
      "rotate_side": "d"
    }
  ]
}
```

Using Symmetry Detection

For molecules with symmetric torsions, reduce dimensionality automatically:

```
python main.py molecule.xyz \
    --torsions-file torsions.json \
    --enable-symmetric-torsions \
    --symmetric-torsion-tolerance 10.0
```

High-Accuracy Optimization

Use hybrid PSO+Nelder-Mead for tight convergence:

```
python main.py molecule.xyz --optimizer pso-nm \
    --swarm-size 80 --max-iters 400 \
    --hybrid-nm-max-iters 200 --hybrid-nm-tol 1e-6 \
    --workers 16
```

Comparing Multiple Algorithms

Run the same system with different optimizers:

```
# PSO
python main.py mol.xyz --optimizer pso --seed 42

# Genetic Algorithm
python main.py mol.xyz --optimizer ga --seed 42

# Grey Wolf Optimizer
python main.py mol.xyz --optimizer gwo --seed 42

# Hybrid
python main.py mol.xyz --optimizer pso-nm --seed 42
```

Understanding Output

Console Output

The tool prints structured blocks showing:

- System information (OS, CPU, memory)
- Input files and molecular properties
- Optimizer configuration
- xTB backend settings
- Iteration-by-iteration progress
- Final results summary

Result Files

optimization_results.txt
: Complete summary including:
  - Best objective value and binding energy
  - Optimized translation, rotation, and torsion parameters
  - Execution time and evaluation counts

optimized_monomer_*.xyz
: Best monomer geometry in XYZ format

optimized_stack_*.xyz
: Multi-layer stack for visualization

pso_trajectory.csv
: (Optional) Particle trajectories when –print-trajectories is enabled

Tips for Effective Optimization

Best Practices

- **Start Small:** Test with –max-iters 10 and small swarm/population to verify setup
- **Validate xTB:** Always run xtb –version before long optimizations
- **Match Workers to Cores:** Set –workers equal to available CPU cores
- **Use Trajectory Logging:** Enable –print-trajectories to diagnose convergence issues
- **Set Seeds:** Use –seed for reproducible results across runs
- **Monitor Convergence:** Check optimization_results.txt after each run

Command-Line Reference

This section provides a complete reference for all command-line options.

Synopsis

```
python main.py XYZ_FILE [OPTIONS]
```

Required Arguments

XYZ_FILE	Path to monomer XYZ coordinate file (*required positional argument*)

General Options

–n-layer	Number of layers for energy evaluation stack (default: 2)
–optimizer	Optimization method: pso, ga, gwo, or pso-nm (default: pso)
–max-iters	Maximum optimizer iterations/generations (default: 300)
–seed	Random seed for reproducible initialization (default: 42)
–verbose-every	Print progress every N iterations (default: 1)

xTB Backend Options

–workers	Number of parallel xTB worker processes (default: 4)
–threads	OpenMP threads per xTB calculation (default: 1)
–method	xTB method: gfn2, gfn1, gfn0, or gfnff (default: gfn2)
–charge	Total molecular charge (default: 0)
–mult	Spin multiplicity (default: 1)

Torsion and Symmetry Options

–torsions-file	Path to JSON file defining dihedral torsions (default: disabled)
–enable-symmetric-torsions	Enable automatic symmetry detection (default: disabled)

–symmetric-torsion-tolerance	Symmetry tolerance in degrees (default: 10.0°)

Penalty Function Options

–penalty-weight	Clash penalty weight (default: 2.0)
–clash-cutoff	Clash detection cutoff in Ångströms (default: 1.6 Å)
–intramol-penalty-weight	Intramolecular clash penalty weight (default: 5.0)
–intramol-cutoff	Intramolecular clash cutoff in Ångströms (default: 1.2 Å)

PSO-Specific Options

–swarm-size	Number of particles in swarm (default: 60)
–inertia	Inertia weight (default: 0.73)
–cognitive	Cognitive coefficient (default: 1.50)
–social	Social coefficient (default: 1.50)
–tol	Convergence tolerance for early stopping (default: 0.01)
–patience	Iterations to wait before early stopping (default: 20)
–print-trajectories	Enable CSV trajectory logging (default: False)

Genetic Algorithm Options

–ga-population	Population size (default: 80)
–ga-elite-fraction	Elite fraction (default: 0.10)
–ga-tournament-size	Tournament size (default: 3)
–ga-crossover-rate	Crossover probability (default: 0.90)
–ga-mutation-rate	Mutation probability (default: 0.10)
–ga-mutation-sigma	Mutation noise standard deviation (default: 0.30)
–ga-tol	Convergence tolerance for early stopping (default: 0.01)
–ga-patience	Generations to wait before early stopping (default: 20)

Grey Wolf Optimizer Options

–gwo-pack-size	Number of wolves (default: 50)
–gwo-a-start	Initial exploration parameter (default: 2.0)
–gwo-a-end	Final exploration parameter (default: 0.0)
–gwo-tol	Convergence tolerance for early stopping (default: 0.01)
–gwo-patience	Iterations to wait before early stopping (default: 20)

PSO+Nelder-Mead Hybrid Options

–hybrid-nm-max-iters	Maximum NM iterations (default: 200)
–hybrid-nm-initial-step	Initial simplex step size (default: 0.20)
–hybrid-nm-alpha	Reflection coefficient (default: 1.0)
–hybrid-nm-gamma	Expansion coefficient (default: 2.0)
–hybrid-nm-rho	Contraction coefficient (default: 0.50)
–hybrid-nm-sigma	Shrink coefficient (default: 0.50)
–hybrid-nm-tol	Convergence tolerance (default: 0.01)

Getting Help

```
pi-stack-generator --help
```

Use python pi-stack-generator.py –help if you have not sourced the activation script.

Workflow Overview

Before diving into implementation details, this section provides a high-level view of how the π-Stack Optimizer processes molecular inputs to discover optimal stacking configurations.

Conceptual Pipeline

The optimization workflow follows a clear data transformation pipeline from molecular input to optimized stack geometries:

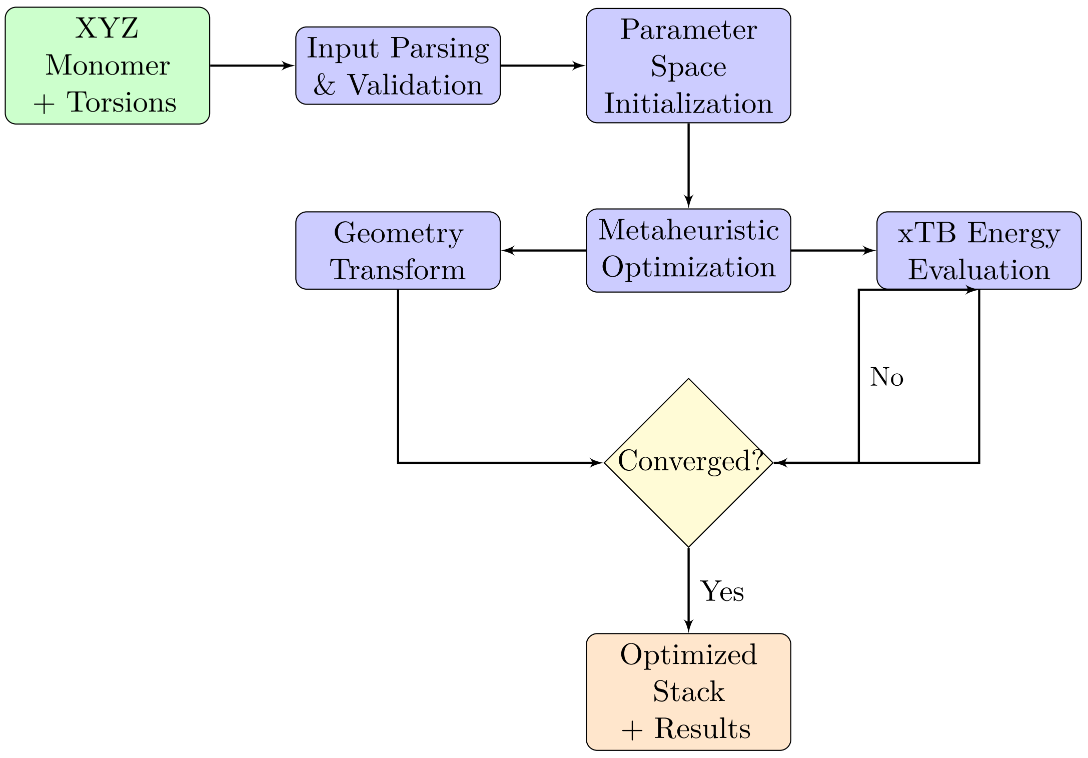
XYZ Monomer + Torsions
Input Parsing & Validation
Parameter Space Initialization
Geometry Transform
Metaheuristic Optimization
xTB Energy Evaluation
No
Converged?
Yes
Optimized Stack + Results

Figure 1: High-level workflow showing the iterative optimization loop between parameter updates and energy evaluations.

The workflow transforms a single monomer structure into an optimized supramolecular assembly through iterative refinement:

Input: Monomer XYZ + Optional Torsions

↓ *Parameter space definition*

Search: $\mathbf{x} = [c_\theta, s_\theta, T_x, T_y, T_z, C_x, C_y, \tau_1, \ldots, \tau_n]$

↓ *Geometry construction + Energy evaluation*

Objective: $f(\mathbf{x}) = E_{\text{bind}} + \text{Clash Penalties}$

↓ *Metaheuristic optimization*

Output: Optimized Stack + Analysis

This modular design ensures that new optimization algorithms, energy methods, or geometric constraints can be integrated with minimal code changes, making the framework suitable for both routine screening and methodological research.

Technical Details

Software Architecture

The system follows a modular pipeline architecture with clear separation of concerns:

$$\begin{tikzcd}
\text{pi-stack-generator.py} \arrow[d] & \\
\text{modules/xyz\_io.py} \arrow[d] & \\
\text{modules/geometry.py} \arrow[d] & \\
\text{modules/objective.py} \arrow[r] \arrow[d] & \text{modules/xtb\_workers.py} \\
\text{modules/optimizer.py} \arrow[d] & \\
\text{modules/reporting.py} &
\end{tikzcd}$$

Figure 2: High-level workflow pipeline showing data flow through core modules.

Module Responsibilities

pi-stack-generator.py
: Orchestrates workflow, parses CLI, initializes components, manages output

modules/xyz_io.py
: Validates XYZ files, reads/writes molecular coordinates

modules/geometry.py
: Handles coordinate transformations, torsion application, clash detection

modules/objective.py
: Defines objective function interface, implements batch evaluation

modules/xtb_workers.py
: Manages parallel xTB processes, handles scratch directories, implements performance optimizations

modules/optimizer.py
: Factory pattern for algorithm selection, implements PSO/GA/GWO/Hybrid

modules/reporting.py
: Generates result summaries and parameter formatting

Computational Workflow

Parameter Space Definition

The optimization explores a $(7 + N_{\text{torsion}})$-dimensional space consisting of 7 rigid-body transformation parameters and N_{torsion} internal dihedral angles:

$$\mathbf{x} = [c_\theta, s_\theta, T_x, T_y, T_z, C_x, C_y, \tau_1, \tau_2, \ldots, \tau_{N_{\text{torsion}}}] \in \mathbb{R}^{7+N_{\text{torsion}}} \tag{2}$$

where the first 7 parameters define a rotation-around-center transformation.

Seven-Dimensional Transformation Parameterization Instead of using Euler angles (which suffer from gimbal lock and discontinuities), the code employs a rotation-around-center parameterization with 7 parameters:

- c_θ, s_θ: Trigonometric components encoding rotation angle θ (dimensionless)
- T_x, T_y, T_z: Translation vector components (Ångströms)

- C_x, C_y: Center-of-rotation coordinates in the xy-plane (Ångströms)

Transformation Matrix The transformation combines rotation around an arbitrary center $(C_x, C_y, 0)$ with translation. We derive the transformation matrix step by step, starting from basic geometric operations.

Step 1: Rotation Angle Recovery The optimization parameters c_θ and s_θ encode the rotation angle without 2π periodicity issues. The actual rotation angle θ is recovered as:

$$\theta = \arctan 2(s_\theta, c_\theta) \tag{3}$$

From this angle, we compute the standard trigonometric values:

$$c = \cos\theta, \qquad s = \sin\theta \tag{4}$$

Step 2: Rotation Around Origin A rotation by angle θ around the z-axis (origin) transforms a point (x, y, z) to:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} cx - sy \\ sx + cy \\ z \end{bmatrix} \tag{5}$$

Step 3: Rotation Around Arbitrary Center To rotate around an arbitrary center $\mathbf{C} = (C_x, C_y, 0)$ instead of the origin, we:

1. Translate by $-\mathbf{C}$ to move the center to the origin
2. Apply the rotation around the origin
3. Translate back by $+\mathbf{C}$

This yields the transformation:

$$\mathbf{p}' = \mathbf{R}(\theta)(\mathbf{p} - \mathbf{C}) + \mathbf{C} = \mathbf{R}(\theta)\mathbf{p} + (\mathbf{C} - \mathbf{R}(\theta)\mathbf{C}) \tag{6}$$

where $\mathbf{R}(\theta)$ is the rotation matrix from Eq. (5). Expanding the center correction term:

$$\begin{aligned} \mathbf{C} - \mathbf{R}(\theta)\mathbf{C} &= \begin{bmatrix} C_x \\ C_y \\ 0 \end{bmatrix} - \begin{bmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} C_x \\ C_y \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} C_x \\ C_y \\ 0 \end{bmatrix} - \begin{bmatrix} cC_x - sC_y \\ sC_x + cC_y \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} C_x - (cC_x - sC_y) \\ C_y - (sC_x + cC_y) \\ 0 \end{bmatrix} \end{aligned} \tag{7}$$

Step 4: Adding Translation The optimization parameters include an additional translation vector $\mathbf{T} = (T_x, T_y, T_z)$. The complete transformation becomes:

$$\mathbf{p}' = \mathbf{R}(\theta)\mathbf{p} + (\mathbf{C} - \mathbf{R}(\theta)\mathbf{C}) + \mathbf{T} \tag{8}$$

Combining the center correction and translation terms, we obtain the effective displacement vector:

$$\begin{aligned}
d_x &= T_x + C_x - (cC_x - sC_y) = T_x + C_x - cC_x + sC_y \\
d_y &= T_y + C_y - (sC_x + cC_y) = T_y + C_y - sC_x - cC_y \\
d_z &= T_z
\end{aligned} \tag{9}$$

Step 5: Homogeneous Transformation Matrix Expressing the transformation in homogeneous coordinates (4D) allows us to represent both rotation and translation as a single matrix multiplication. The final 4×4 transformation matrix is:

$$\mathbf{M} = \begin{bmatrix} c & -s & 0 & d_x \\ s & c & 0 & d_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{10}$$

where $c = \cos\theta$ and $s = \sin\theta$ are computed from the optimization parameters via Eq. (3) and Eq. (4), and d_x, d_y are given by Eq. (9).

Step 6: Coordinate Transformation For a coordinate $\mathbf{p} = (x, y, z)^T$, the transformed coordinate in homogeneous form is:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \mathbf{M} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} c & -s & 0 & d_x \\ s & c & 0 & d_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} cx - sy + d_x \\ sx + cy + d_y \\ z + T_z \\ 1 \end{bmatrix} \tag{11}$$

The final 3D coordinates are obtained by extracting the first three components: $\mathbf{p}' = (x', y', z')^T$.

Multi-Layer Stack Construction For an N-layer stack, successive layers are generated by repeated application of $\mathbf{M}$:

$$\text{Layer } k: \quad \mathbf{p}_k = \mathbf{M}^k \mathbf{p}_0, \quad k = 0, 1, 2, \ldots, N-1 \tag{12}$$

where $\mathbf{p}_0$ is the base monomer coordinate and $\mathbf{M}^k = \underbrace{\mathbf{M} \cdot \mathbf{M} \cdots \mathbf{M}}_{k \text{ times}}$ is the k-th matrix power.

Optimization Bounds The parameter space is initialized with Gaussian distributions centered at reasonable physical values:

Parameter	Initialization	Physical Meaning
c_θ	$\mathcal{N}(0, 1)$	Cosine-like component (unbounded)
s_θ	$\mathcal{N}(0, 1)$	Sine-like component (unbounded)
T_x	$\mathcal{N}(0, 2.0)$ Å	x-translation
T_y	$\mathcal{N}(0, 2.0)$ Å	y-translation
T_z	$\mathcal{N}(3.5, 1.0)$ Å	z-translation (typical π-stacking distance)
C_x	$\mathcal{N}(0, 2.0)$ Å	Rotation center x-coordinate
C_y	$\mathcal{N}(0, 2.0)$ Å	Rotation center y-coordinate
τ_i	$\mathcal{N}(0, 0.5)$ rad	Torsion angle i, clipped to $[-\pi, \pi]$

Table 1: Parameter initialization distributions for swarm-based optimizers.

Rationale:

- Using (c_θ, s_θ) instead of θ directly avoids 2π periodicity and allows continuous exploration
- $T_z \sim \mathcal{N}(3.5, 1.0)$ centers the search around typical π-stacking distances (3–4 Å)
- In-plane parameters (T_x, T_y, C_x, C_y) use wider distributions to explore lateral shifts and rotation centers
- Torsion angles are clipped to $[-\pi, \pi]$ to maintain physical relevance

Parameter Bounds and Clamping To prevent the optimizer from exploring unphysical regions of parameter space, all optimization algorithms enforce hard bounds on parameters after each update step. These bounds are critical for preventing optimizer drift (particularly in PSO) that can lead to extreme parameter values and xTB calculation failures.

Parameter	Lower Bound	Upper Bound
c_θ	-1.0	$+1.0$
s_θ	-1.0	$+1.0$
T_x	-10.0 Å	$+10.0$ Å
T_y	-10.0 Å	$+10.0$ Å
T_z	$+2.0$ Å	$+6.0$ Å
C_x	-10.0 Å	$+10.0$ Å
C_y	-10.0 Å	$+10.0$ Å
τ_i	-2π rad	$+2\pi$ rad

Table 2: Hard parameter bounds enforced after each optimizer update. Parameters are clamped to these ranges using np.clip.

Implementation: After each position update in PSO (and similarly for other algorithms), the _clamp_positions function applies:

$$x_i \leftarrow \text{clip}(x_i, x_{i,\min}, x_{i,\max}) \tag{13}$$

This prevents accumulation of unrealistic values that could arise from unbounded velocity updates in PSO or mutation/crossover operations in GA. The bounds are defined in PARAM_BOUNDS dictionary in modules/optimizer.py.

Objective Function Evaluation

Evaluation Pipeline For each candidate vector $\mathbf{x} \in \mathbb{R}^{7+N_{\text{torsion}}}$ the optimizer executes a fixed series of transformations:

Parameter extraction
Partition $\mathbf{x}$ into rigid-body parameters $\mathbf{x}_{\text{rigid}} \in \mathbb{R}^7$ and torsions $\boldsymbol{\tau} \in \mathbb{R}^{N_{\text{torsion}}}$.

Torsion application
Apply each torsion in $\boldsymbol{\tau}$ to the relevant fragment, producing the modified monomer coordinates $\mathbf{p}_{\text{mono}}$.

Stack construction
Build the transformation matrix $\mathbf{M}$ from $\mathbf{x}_{\text{rigid}}$ and replicate the monomer by powers of $\mathbf{M}$ to obtain the N-layer stack.

Clash analysis
Evaluate inter-layer and intra-molecular overlaps to obtain preliminary penalty terms P_{inter} and P_{intra} (detailed below).

Energy sampling

Dispatch both the relaxed monomer and the stacked geometry to the xTB worker pool to retrieve E_{mono} and E_{stack}.

Energy Component The physics-based portion of the objective is the per-interface binding energy:

$$E_{\text{bind}} = \frac{E_{\text{stack}} - N \cdot E_{\text{mono}}}{N-1} \times 2625.5 \quad (\text{kJ/mol}) \tag{14}$$

which converts Hartree differences to kJ/mol and normalizes by the number of interfaces to make systems with different N directly comparable.

Clash Penalty Models Clash penalties prevent unphysical geometries where atoms approach closer than van der Waals radii. Two complementary models are applied:

Inter-layer penalty (P_{inter}): Captures overlaps between distinct layers in the stack with cutoff $r_{\text{inter}} = 1.6$ Å. The penalty uses geometric weights $w_{ij} = \frac{1}{j-i}$ so closer layers carry higher penalties.

Intra-molecular penalty (P_{intra}): Suppresses clashes inside the monomer caused by torsion rotations, with cutoff $r_{\text{intra}} = 1.2$ Å. The exclusion list removes bonded (1-2) and angle (1-3) pairs, ensuring only non-bonded contacts contribute.

Combined Objective and Failure Handling Binding energy alone cannot guarantee chemically valid structures, so the optimizer augments it with the clash models above. The first step is to accumulate a single scalar penalty:

$$P_{\text{total}} = P_{\text{inter}} + w_{\text{intra}} \cdot P_{\text{intra}} \tag{15}$$

where $w_{\text{intra}} = 5.0$ is the intra-molecular penalty weight. For successful xTB evaluations, the fitness is:

$$f(\mathbf{x}) = E_{\text{bind}} + c_{\text{clash}} \cdot P_{\text{total}} \tag{16}$$

where E_{bind} is the per-interface binding energy in kJ/mol and $c_{\text{clash}} = 10{,}000$ ensures clash penalties dominate when present.

Failed xTB calculations (e.g., SCF divergence caused by extreme clashes) receive a shifted objective

$$f_{\text{fail}}(\mathbf{x}) = c_{\text{base}} + c_{\text{clash}} \cdot P_{\text{total}} \tag{17}$$

with $c_{\text{base}} = 1{,}000{,}000$ (FAILURE_PENALTY_BASE). The additive $c_{\text{clash}}P_{\text{total}}$ term still differentiates degrees of failure, nudging the search toward less pathological geometries before re-entering the feasible region.

Parameter	Value	Purpose
w_{intra}	5.0	Weight for intra-molecular penalties in P_{total}
c_{clash}	10,000	Clash penalty multiplier (FAILURE_PENALTY_CLASH_MULT)
c_{base}	1,000,000	Base penalty for failed calculations (FAILURE_PENALTY_BASE)
r_{inter}	1.6 Å	Inter-molecular clash cutoff distance
r_{intra}	1.2 Å	Intra-molecular clash cutoff distance

Table 3: Clash penalty constants and their roles in the objective function.

Penalty Summary

Behavior Across Regimes Because $c_{\text{base}} \gg c_{\text{clash}} \gg |E_{\text{bind}}|$, the fitness landscape separates into three intuitive bands:

- **Failed geometries ($f > 10^6$):** Always worse than any converged point but still ranked by clash magnitude to steer the search toward milder failures.
- **Clash-prone geometries ($50 < f < 10^6$):** Dominated by $c_{\text{clash}} P_{\text{total}}$, forcing rapid movement toward feasible regions.
- **Valid geometries ($-50 < f < 50$):** Essentially clash-free so the optimizer compares solutions purely on binding energy.

xTB Parallel Execution

The xTB quantum chemistry calculations are parallelized using a worker pool architecture to maximize computational throughput. The implementation leverages Python's multiprocessing.Pool to spawn independent worker processes, each maintaining an isolated scratch directory for file I/O operations. The system uses only command-line xTB binaries (no tblite dependency) and includes several performance optimizations.

Worker Pool Architecture The parallel execution workflow proceeds as follows:

- At program startup, spawn N_{workers} independent processes. Each worker creates an isolated temporary directory to avoid file conflicts and remains idle, waiting for geometry submission requests.
- The main optimization loop submits batches of molecular geometries to the worker pool queue. Each submission includes atom types, Cartesian coordinates, and a unique job identifier. Jobs are distributed to available workers using a FIFO scheduling policy.
- Each worker receives a geometry, writes it to an XYZ file in its scratch directory, invokes the xTB binary with command xtb –gfn2 –sp for single-point GFN2-xTB energy calculation, and parses the output file to extract the total energy in Hartrees.
- Computed energies are returned to the main process via a result queue, where they are matched to their originating job IDs. If xTB reports an error (e.g., SCF convergence failure), the worker returns an error flag instead of an energy value.
- Upon optimization completion, all worker processes are terminated gracefully, and their temporary directories are recursively removed to avoid accumulating scratch files on disk.

Performance Considerations The number of workers N_{workers} should be chosen based on available CPU cores and xTB's threading behavior. Each xTB worker internally uses OpenMP threads (controlled by OMP_NUM_THREADS), so the total thread count is approximately:

$$N_{\text{threads,total}} \approx N_{\text{workers}} \times N_{\text{OMP}} \tag{18}$$

For optimal performance on a machine with N_{CPU} cores, typical configurations are:

- **High parallelism:** $N_{\text{workers}} = N_{\text{CPU}}$, $N_{\text{OMP}} = 1$ (many small jobs, minimal inter-thread synchronization overhead)

- **Balanced:** $N_{\text{workers}} = N_{\text{CPU}}/4$, $N_{\text{OMP}} = 4$ (moderate parallelism with some threading efficiency)
- **Low parallelism:** $N_{\text{workers}} = N_{\text{CPU}}/8$, $N_{\text{OMP}} = 8$ (few large jobs, higher per-job threading efficiency)

Symmetry Detection

Molecular symmetry can drastically reduce the search dimensionality by identifying torsion angles that should remain equivalent due to chemical equivalence. When the –enable-symmetric-torsions flag is activated, the optimizer applies an automated symmetry detection algorithm.

Symmetry Detection Algorithm The algorithm identifies groups of torsions that are chemically equivalent by analyzing local bonding environments:

Step	Procedure
1	**Pairwise Comparison:** For each pair of torsion definitions (T_i, T_j), compare the four defining atoms and their immediate bonding neighborhoods to detect structural similarity.
2	**Environment Matching:** Two torsions are considered equivalent if their central bond atoms have identical element types and bond orders, and their peripheral atoms (defining the dihedral) have matching connectivity patterns within a specified tolerance.
3	**Grouping:** Cluster all mutually equivalent torsions into symmetry groups. Each group $\mathcal{G}_k = \{T_{k,1}, T_{k,2}, \ldots, T_{k,m_k}\}$ contains m_k symmetric torsions that will share a single control parameter.
4	**Dimension Reduction:** Map the original N_{torsion} independent torsion angles to $N_{\text{reduced}} = \lvert\{\mathcal{G}_k\}\rvert$ control variables. The dimensionality reduction factor is $\Delta = N_{\text{torsion}} - N_{\text{reduced}}$.
5	**Expansion:** During objective function evaluation, expand the reduced parameter vector by applying the same angle to all torsions in each symmetry group, ensuring physical consistency.

Table 4: Five-step symmetry detection and dimension reduction algorithm.

Impact on Optimization Efficiency Symmetry detection provides substantial benefits for molecules with repeated structural motifs. Consider a benzene derivative with six equivalent methyl rotors: without symmetry detection, the optimizer explores a $(7 + 6) = 13$-dimensional space; with symmetry detection, this reduces to $(7+1) = 8$ dimensions, decreasing computational cost by approximately $\left(\frac{13}{8}\right)^2 \approx 2.6\times$ for swarm-based methods where cost scales roughly as $\mathcal{O}(d^2)$.

The symmetry tolerance parameter (default: 10^{-3} Å for distance comparisons) controls the strictness of environment matching. Tighter tolerances reduce false positives but may miss genuine symmetries in slightly distorted geometries.

Logging and Output

Console Output Structure

Structured ASCII blocks always appear in the same order. A banner opens the run and shows authors plus start time. System information follows with working directory, script path, Python version, and any detected xTB build. Next comes the input overview (XYZ/torsions) and geometry validation, then the molecular system, xTB configuration, and optimizer configuration blocks. The optimization settings block lists dimensionality, symmetry, and penalty weights before the progress log begins (cadence controlled by –verbose-every). When iterations finish, the script prints the results summary (best objective, binding energy, key parameters), lists the files written to disk, and closes with a footer showing end time and total elapsed seconds.

File Artifacts

- optimization_results.txt: Complete parameter dump and energies
- optimized_monomer_*.xyz: Best monomer geometry
- optimized_stack_*.xyz: Multi-layer visualization stack
- pso_trajectory.csv: Optional particle-level iteration log

Optimization Algorithms

The framework provides four distinct optimization algorithms, each with unique characteristics suited to different problem landscapes.

Particle Swarm Optimization (PSO)

Algorithm Overview

Particle Swarm Optimization, introduced by Kennedy and Eberhart [1], is a population-based stochastic optimization technique inspired by the social behavior of bird flocking and fish schooling. Each particle in the swarm represents a candidate solution that moves through the search space influenced by its own experience and that of its neighbors.

Mathematical Formulation

PSO maintains a swarm of particles exploring the parameter space, each tracking personal and global best positions. The velocity update equation combines three components:

$$\mathbf{v}_i^{(t+1)} = \omega \mathbf{v}_i^{(t)} + c_1 \mathbf{r}_1 \odot \left(\mathbf{p}_i - \mathbf{x}_i^{(t)}\right) + c_2 \mathbf{r}_2 \odot \left(\mathbf{g} - \mathbf{x}_i^{(t)}\right), \tag{19}$$

$$\mathbf{x}_i^{(t+1)} = \mathbf{x}_i^{(t)} + \mathbf{v}_i^{(t+1)}, \tag{20}$$

where:

- ω is the inertia weight controlling momentum (default: 0.73)
- c_1 and c_2 are cognitive and social acceleration coefficients (default: 1.50)
- $\mathbf{p}_i$ is the personal best position for particle i
- $\mathbf{g}$ is the global best position across all particles
- $\mathbf{r}_1, \mathbf{r}_2$ are random vectors sampled element-wise from $\mathcal{U}(0, 1)$
- $\odot$ denotes element-wise (Hadamard) product

The velocity is bounded: $\mathbf{v}_i \in [\mathbf{v}_{\min}, \mathbf{v}_{\max}]$ to prevent explosive growth.

Algorithm Workflow

The algorithm iteratively updates particle positions based on personal and global best positions, with early stopping when convergence is detected.

Key Parameters

- –swarm-size: Number of particles (default: 60)
- –inertia: Inertia weight (default: 0.73)
- –cognitive: Cognitive coefficient (default: 1.50)
- –social: Social coefficient (default: 1.50)
- –tol: Convergence tolerance (default: 0.01)
- –patience: Early stopping patience (default: 20)

PSO + Nelder-Mead Hybrid

The hybrid optimizer combines PSO for global exploration with Nelder-Mead for local refinement. After PSO completes, the best solution is refined using Nelder-Mead simplex optimization.

Key Parameters

- All PSO parameters plus:
- –hybrid-nm-max-iters: NM iteration limit (default: 200)
- –hybrid-nm-alpha, –hybrid-nm-gamma, –hybrid-nm-rho, –hybrid-nm-sigma: NM coefficients
- –hybrid-nm-tol: Convergence tolerance (default: 0.01)

Genetic Algorithm (GA)

Algorithm Overview

The Genetic Algorithm [3, 4] is a population-based metaheuristic inspired by biological evolution. Candidate solutions ("chromosomes") are iteratively refined through selection, crossover, and mutation operations. GA excels at exploring diverse regions of the search space due to its crossover mechanism, making it particularly effective for problems with multiple isolated optima.

Mathematical Formulation

GA maintains a population of N_{pop} candidate solutions. Each generation applies the following steps:

Selection. Individuals are selected for reproduction based on fitness using tournament selection: randomly draw k individuals and choose the best. This operator favors high-fitness candidates while preserving diversity.

Crossover. Two parent chromosomes generate offspring by combining genetic material. Single-point crossover randomly selects a crossover point $c \in [1, d-1]$ and swaps segments:

$$\text{Child}_1[i] = \begin{cases} \text{Parent}_1[i] & \text{if } i < c \\ \text{Parent}_2[i] & \text{otherwise} \end{cases} \tag{21}$$

Mutation. Each gene mutates with probability p_m:

$$x'_i = x_i + \mathcal{N}(0, \sigma_m) \quad \text{with probability } p_m \tag{22}$$

This operation maintains genetic diversity and prevents premature convergence to local minima.

Elitism. A fraction of the best individuals are preserved unchanged in the next generation to prevent loss of good solutions.

Early Stopping. Similar to PSO, GA monitors convergence by tracking improvement over a patience window. If the improvement $\Delta f = |f_{\text{best}}^{(t)} - f_{\text{best}}^{(t-p)}|$ falls below tolerance ϵ_{tol} for p consecutive generations, optimization terminates early. This prevents unnecessary evaluations once convergence is achieved.

Key Parameters

- –ga-population: Population size (default: 80)
- –ga-mutation-rate: Probability of gene mutation (default: 0.10)
- –ga-crossover-rate: Probability of crossover (default: 0.90)
- –ga-tournament-size: Tournament selection size (default: 3)
- –ga-tol: Convergence tolerance (default: 0.01)
- –ga-patience: Early stopping patience in generations (default: 20)
- –max-iters: Maximum number of generations

Grey Wolf Optimizer (GWO)

Algorithm Overview

The Grey Wolf Optimizer [5] is inspired by the social hierarchy and hunting behavior of grey wolves in nature. Wolves are organized into hierarchical packs with alpha (leader), beta (second-in-command), and delta (subordinates) individuals. The algorithm mimics this structure to balance exploration and exploitation, making GWO particularly robust for continuous optimization landscapes.

Mathematical Formulation

GWO maintains a population of N wolves, ranked by fitness into three tiers:

Wolf Hierarchy.

- **Alpha (α):** Best fitness solution; leads the hunt
- **Beta (β):** Second-best solution; assists alpha

- **Delta (δ):** Third-best solution; follows leaders
- **Omega (ω):** Remaining wolves; follow leaders' guidance

Position Update. Each wolf refines its position by averaging the pull from the best three leaders. For every leader $k \in \{\alpha, \beta, \delta\}$ the algorithm computes

$$\mathbf{A}_k = 2\,\mathbf{a} \odot \mathbf{r}_{1,k} - \mathbf{a}, \qquad \mathbf{C}_k = 2\,\mathbf{r}_{2,k} \tag{23}$$

$$\mathbf{D}_k = \left|\mathbf{C}_k \odot \mathbf{x}_k - \mathbf{x}^{(t)}\right|, \qquad \mathbf{X}_k = \mathbf{x}_k - \mathbf{A}_k \odot \mathbf{D}_k \tag{24}$$

followed by the averaged update

$$\mathbf{x}^{(t+1)} = \frac{\mathbf{X}_\alpha + \mathbf{X}_\beta + \mathbf{X}_\delta}{3}. \tag{25}$$

Here $\odot$ denotes element-wise multiplication, $\mathbf{r}_{1,k}, \mathbf{r}_{2,k} \sim \mathcal{U}(0,1)^d$, and $\mathbf{a}$ linearly decays from 2 to 0 so the exploration pressure gradually diminishes.

Early Stopping. GWO implements the same convergence detection mechanism as PSO and GA. The algorithm tracks the alpha wolf's fitness over a patience window and terminates when improvement falls below the tolerance threshold for consecutive iterations. This ensures efficient resource utilization by stopping when meaningful progress ceases.

Key Parameters

- –gwo-pack-size: Pack size (default: 50)
- –gwo-tol: Convergence tolerance (default: 0.01)
- –gwo-patience: Early stopping patience in iterations (default: 20)
- –max-iters: Maximum number of iterations
- The algorithm automatically decays the exploration parameter **a** from 2 to 0

Advanced Topics

Extensibility Guidelines

The optimizer factory pattern facilitates straightforward algorithm addition:

1. **Implement Configuration Dataclass**

```
@dataclass
class NewOptimizerConfig:
    param1: float = 1.0
    param2: int = 100
```

2. **Implement Optimizer Class**

```
class NewOptimizer:
    def __init__(self, config, bounds, dim):
        pass

    def optimize(self, objective_fn):
        # Return (best_params, best_value)
        pass
```

3. **Register in Factory**

   Add to SUPPORTED_METHODS and extend create_optimizer()

4. **Update CLI**

   Add command-line flags in parse_args()

Performance Tuning

Worker Count Selection

Optimal –workers depends on:

- **CPU Cores:** Set equal to physical core count for CPU-bound tasks
- **Memory:** Each xTB process requires ∼500 MB minimum
- **I/O:** Too many workers can saturate disk I/O for scratch files

Recommendation: Start with –workers=4, scale up while monitoring system load.

Algorithm Selection

Algorithm	Best For
PSO	Smooth landscapes with few local minima, requires moderate evaluations
GA	Discrete-continuous problems, good diversity maintenance
GWO	Robust general-purpose optimization, balanced exploration/exploitation
PSO+NM	High-accuracy requirements, when tight convergence tolerances needed

Table 5: Algorithm selection guidelines based on problem characteristics.

Best Practices

Production Workflow Recommendations

- **Validation Run:** Always test with –max-iters 10 before full optimization
- **xTB Check:** Verify xtb –version succeeds before batch jobs
- **Reproducibility:** Set –seed and log all CLI arguments
- **Convergence Monitoring:** Enable –print-trajectories for diagnostics
- **Resource Scaling:** Match –workers to available cores
- **Symmetry Detection:** Use for symmetric molecules to reduce dimensionality

Hyperparameter Optimization Framework

The π-Stack Optimizer includes a sophisticated hyperparameter optimization framework built on Optuna that automatically tunes optimization algorithm parameters across multiple molecular systems. This framework enables systematic exploration of hyperparameter spaces to find globally optimal configurations, significantly improving optimization performance compared to default parameter settings.

Metaheuristic optimization algorithms (PSO, GA, GWO, PSO-NM) contain numerous hyperparameters that critically affect convergence speed, solution quality, and robustness. Manual tuning of these parameters is time-consuming and often suboptimal, particularly when optimizing across diverse molecular systems with varying energy landscapes. The hyperparameter optimization framework addresses this challenge by providing automated parameter tuning using Optuna's Tree-structured Parzen Estimator, persistent caching to avoid redundant evaluations, support for both per-molecule and joint optimization strategies, warm-starting capabilities for transferring parameters between related systems, and comprehensive logging of trial histories and performance statistics.

The framework is implemented in hyperparameter-opt/hyperopt.py and consists of several key components:

HyperparameterCache
: SQLite-based persistent cache for hyperparameter evaluation results, enabling result memoization across optimization runs.

CachedHyperparameterOptimizer
: Main optimizer class that integrates caching with Optuna-based hyperparameter search, providing significant speedup through result reuse.

MoleculeCase
: Container for molecular input files (XYZ coordinates and optional torsion definitions) discovered automatically from directory structures.

create_cached_objective()
: Objective function factory that wraps molecule evaluation with caching, supporting both sequential and joint optimization modes.

Hyperparameter Specifications

The framework defines hyperparameter search spaces for four optimization algorithms.

Particle Swarm Optimization (PSO)

Table 6: PSO hyperparameter search space and default values.

Parameter	Description	Range	Default
swarm_size	Number of particles in swarm	[30, 120] (step 10)	60
inertia	Momentum coefficient	[0.4, 0.9] (step 0.05)	0.65
cognitive	Personal best attraction	[1.0, 2.5]	1.50
social	Global best attraction	[1.0, 2.5]	1.50

Genetic Algorithm (GA)

Table 7: GA hyperparameter search space and default values.

Parameter	Description	Range	Default
ga_population	Population size	[50, 200] (step 10)	80
ga_mutation_rate	Mutation probability	[0.01, 0.4] (step 0.05)	0.10
ga_mutation_sigma	Mutation noise std dev	[0.05, 0.6] (step 0.05)	0.30
ga_crossover_rate	Crossover probability	[0.6, 1.0] (step 0.05)	0.80
ga_elite_fraction	Elite preservation ratio	[0.05, 0.2]	0.1
ga_tournament_size	Tournament selection size	[2, 8] (step 1)	3

Grey Wolf Optimizer (GWO)

Table 8: GWO hyperparameter search space and default values.

Parameter	Description	Range	Default
gwo_pack_size	Number of wolves	[20, 120] (step 5)	50
gwo_a_start	Initial exploration parameter	[0.5, 3.0]	2.0
gwo_a_end	Final exploration parameter	[0.0, 1.0]	0.0

PSO + Nelder-Mead Hybrid (PSO-NM)

The hybrid optimizer combines all PSO parameters with additional Nelder-Mead specific parameters.

Table 9: PSO-NM hybrid hyperparameter search space and default values.

Parameter	Description	Range	Default
PSO Parameters (as above) plus:			
hybrid_nm_max_iters	NM maximum iterations	[50, 400] (step 10)	200
hybrid_nm_initial_step	Initial simplex step size	[0.05, 0.8]	0.20
hybrid_nm_alpha	Reflection coefficient	[0.5, 2.0]	1.0
hybrid_nm_gamma	Expansion coefficient	[1.0, 3.5]	2.0
hybrid_nm_rho	Contraction coefficient	[0.1, 0.9]	0.50
hybrid_nm_sigma	Shrink coefficient	[0.1, 0.9]	0.50
hybrid_nm_tol	Convergence tolerance	[1e-5, 5e-3] (log)	0.001

Optimization Modes

Sequential Mode (Default)

Sequential mode optimizes hyperparameters separately for each molecule, using warm-starting to transfer knowledge between related systems:

Algorithm 1 Sequential Hyperparameter Optimization

```
Input: Molecules $\mathcal{M} = \{M_1, M_2, \ldots, M_n\}$, trials per molecule $T$
Input: Hyperparameter specification $\mathcal{H}$, optimizer type $\mathcal{A}$
Output: Optimized parameters $\{\theta_1^*, \theta_2^*, \ldots, \theta_n^*\}$
1: Initialize $\theta_0 \leftarrow \mathcal{H}$.defaults          ▷ Start with default parameters
2: for $i = 1$ to $n$ do
3:     Create Optuna study $S_i$ for molecule $M_i$
4:     Enqueue warm-start trial: $S_i$.enqueue($\theta_{i-1}$)
5:     Define objective: $f_i(\theta) = \text{run\_pi\_stack}(M_i, \theta, \mathcal{A})$
6:     Optimize: $\theta_i^* = S_i.\text{optimize}(f_i, \text{n\_trials} = T)$
7:     Update warm-start: $\theta_i \leftarrow \theta_i^*$          ▷ Transfer to next molecule
8: end for
9: return $\{\theta_1^*, \theta_2^*, \ldots, \theta_n^*\}$
```

Advantages:

- Molecule-specific parameter optimization
- Knowledge transfer through warm-starting
- Independent studies allow parallel execution
- Detailed per-molecule performance analysis

Use Cases:

- Diverse molecular systems with different energy landscapes
- When molecule-specific optimal parameters are desired
- Exploratory studies comparing optimization performance across molecules

Joint Study Mode

Joint study mode finds globally optimal hyperparameters by evaluating all molecules in each trial and optimizing the average performance:

Algorithm 2 Joint Hyperparameter Optimization

```
Input: Molecules $\mathcal{M} = \{M_1, M_2, \ldots, M_n\}$, total trials $T$
Input: Hyperparameter specification $\mathcal{H}$, optimizer type $\mathcal{A}$
Output: Global optimal parameters $\theta^*$
1: Create single Optuna study $S$
2: Enqueue default trial: $S$.enqueue($\mathcal{H}$.defaults)
3: Define multi-molecule objective:
       $$f(\theta) = \frac{1}{n}\sum_{i=1}^{n} \text{run\_pi\_stack}(M_i, \theta, \mathcal{A})$$
4: Optimize: $\theta^* = S.\text{optimize}(f, \text{n\_trials} = T)$
5: return $\theta^*$, per-molecule breakdown
```

Advantages:

- Single globally optimal parameter set

- Robust performance across diverse molecular systems
- Reduced total computational cost (fewer total trials)
- Simplified deployment (one parameter set for all molecules)

Use Cases:

- Production workflows requiring consistent parameters
- Similar molecular systems with comparable energy landscapes
- Limited computational budget requiring efficient parameter search

Optuna Integration

Study Management

The framework leverages Optuna's persistent SQLite storage for robust study management. Sequential mode uses separate databases per molecule, while joint mode uses a single shared database. All studies use Optuna's Tree-structured Parzen Estimator (TPE) sampler for intelligent hyperparameter exploration.

Result Caching

The framework uses SQLite-based persistent caching with deterministic cache keys based on molecule name, optimizer method, and hyperparameters. This eliminates redundant evaluations and provides significant speedup for repeated parameter combinations.

Cache Benefits:

- Eliminates redundant evaluations of identical parameter combinations
- Provides significant speedup for repeated optimization runs
- Enables warm starting with previously computed results
- Tracks cache hit rates and performance statistics

Usage and Command-Line Interface

Basic Usage

Sequential Optimization

```
pi-hyperopt \
    --molecules-root molecules \
    --optimizer pso \
    --trials-per-molecule 50 \
    --progress
```

Joint Study Optimization

```
pi-hyperopt \
    --molecules-root molecules \
    --optimizer pso \
    --trials-per-molecule 100 \
    --joint-study \
    --progress
```

Both commands assume you sourced activate_pi_stack.sh. Without it, either run python hyperparameter-opt/hyperopt.py ... from the repository root or change into hyperparameter-opt and call python hyperopt.py ....

Command-Line Reference

–molecules-root	Directory containing per-molecule subdirectories (default: molecules)
–stack-script	Path to pi-stack-generator.py (default: ../pi-stack-generator.py)
–trials-per-molecule	Number of Optuna trials per molecule (default: 50)
–optimizer	Optimizer type: pso, ga, gwo, or pso-nm (default: pso)
–joint-study	Enable joint study mode (default: False)
–progress	Show progress bars and detailed output (default: False)
–reduced-iterations	Number of iterations for hyperparameter trials (default: 50)
–cache-dir	Directory for persistent result cache (default: cache_hyperopt)
–clear-cache	Clear cache before starting optimization
–cache-stats	Show cache statistics and exit
–study-dir	Directory for Optuna SQLite databases (default: studies)
–runs-dir	Directory for trial execution outputs (default: runs)
–keep-run-dirs	Prevent cleanup of per-trial run folders after success (default: False)
–results-dir	Directory for JSON result summaries (default: results)
–base-args	Additional arguments forwarded to pi-stack-generator.py. **NOTE:** This must be the last argument on the command line.

Worked Example

The repository ships with small BTA/BTA_68b samples under hyperparameter-opt/test. Use them to sanity-check the workflow end-to-end:

1. Copy (or symlink) the test molecules into an isolated workspace:

```
mkdir -p demo_molecules
cp -r hyperparameter-opt/test/BTA demo_molecules/
cp -r hyperparameter-opt/test/BTA_68b demo_molecules/
```

2. Ensure the activation script was sourced so pi-hyperopt is on your PATH.

3. Launch a short sequential study:

```
pi-hyperopt \
    --molecules-root demo_molecules \
    --optimizer pso \
    --trials-per-molecule 5 \
    --runs-dir runs/demo \
    --results-dir results/demo \
    --study-dir studies/demo \
    --progress
```

During execution you should see Optuna progress bars followed by summaries per molecule. After completion, inspect:

- studies/demo/BTA.db, studies/demo/BTA_68b.db: SQLite archives containing all trials.
- runs/demo/<molecule>/trial-*/: Raw stdout/stderr plus optimization_results.txt from the underlying π-stack generator.
- results/demo/*.json: Aggregated best hyperparameters and metrics for each molecule.

To re-run with fresh evaluations, pass –clear-cache. To forward specialized arguments (e.g., extra torsion controls) append them after –base-args. This miniature study mirrors the default directory layout described below while remaining fast enough for laptops.

Input Directory Structure

The framework expects a specific directory structure for molecular inputs:

```
molecules/
+-- BTA/
|    +-- monomer.xyz          # Required: molecular coordinates
|    \-- torsions.json        # Optional: torsion definitions
+-- BTA_68b/
|    +-- monomer.xyz
|    \-- torsions.json
\-- anthracene/
     \-- monomer.xyz          # No torsions for rigid molecules
```

Output and Results

File Structure

Successful optimization runs generate comprehensive output files:

```
hyperparameter-opt/
+-- studies/                  # Optuna SQLite databases
|    +-- BTA.db
|    +-- BTA_68b.db
|    \-- joint_study.db       # For joint mode
+-- runs/                     # Trial execution logs
|    +-- BTA/
|    |    +-- trial-20251120-051846-019797/
|    |    |    +-- stdout.log
|    |    |    +-- stderr.log
|    |    |    +-- optimization_results.txt
|    |    |    \-- trial_metadata.json
|    |    \-- trial-20251120-051846-200467/
|    \-- BTA_68b/
\-- results/                  # Summary JSON files
     +-- BTA.json
     +-- BTA_68b.json
     +-- joint_study_results.json
     \-- overall_summary.json
```

Result File Formats

Result files are JSON-formatted and include best objective values, optimal hyperparameters, trial numbers, and storage locations. Per-molecule results contain molecule-specific optimal parameters, while joint study results include average objectives and per-molecule breakdowns.

Advanced Features

Warm Starting Strategy

Sequential mode implements intelligent warm starting to transfer knowledge between molecules. The best hyperparameters from one molecule are used as the starting point for the next molecule, improving convergence speed for related systems.

Cache Management

The framework provides tools for managing the persistent result cache:

```
# View cache statistics
pi-hyperopt --cache-stats

# Clear cache before starting
pi-hyperopt \
    --clear-cache \
    --trials-per-molecule 100 \
    --optimizer pso
```

Custom Base Arguments

Additional arguments can be forwarded to the underlying pi-stack optimizer:

```
pi-hyperopt \
    --optimizer ga \
    --trials-per-molecule 50 \
    --base-args --workers 8 --threads 2 --n-layer 3 --method gfn1
```

Performance Considerations

Computational Scaling

The computational cost scales as:

$$\text{Cost} = N_{\text{molecules}} \times N_{\text{trials}} \times T_{\text{evaluation}} \tag{26}$$

where $T_{\text{evaluation}}$ is the time for a single pi-stack optimization run.

Optimization Strategies:

- Use result caching to eliminate redundant evaluations (can provide 2-10x speedup)
- Use joint study mode to reduce N_{trials} by factor of $N_{\text{molecules}}$
- Optimize $T_{\text{evaluation}}$ through worker count and iteration limits
- Use warm starting to improve convergence speed

Memory and Storage

- Each trial generates ∼1-10 MB of output files
- Optuna databases grow to ∼1-5 MB per 100 trials
- Cache database: ∼100-500 KB per 1000 cached entries
- Peak memory usage: ∼100 MB for framework + worker memory

Best Practices

Hyperparameter Optimization Best Practices

- **Start Small:** Begin with 10-20 trials to validate setup and estimate runtimes
- **Use Progress Monitoring:** Enable –progress for real-time feedback and cache statistics
- **Leverage Caching:** The persistent cache provides significant speedup on repeated runs; monitor cache hit rates
- **Choose Appropriate Mode:** Use sequential for diverse molecules, joint for similar systems
- **Monitor Resource Usage:** Ensure sufficient disk space for trial outputs and cache database
- **Validate Results:** Check results/ JSON files for convergence, performance, and cache statistics
- **Manage Cache:** Use –cache-stats to monitor cache efficiency and –clear-cache when needed

References

[1] Kennedy, J., & Eberhart, R. (1995). *Particle swarm optimization.* Proceedings of IEEE International Conference on Neural Networks, IV, 1942–1948.

[2] Clerc, M., & Kennedy, J. (2002). *The particle swarm - explosion, stability, and convergence in a multidimensional complex space.* IEEE Transactions on Evolutionary Computation, 6(1), 58–73.

[3] Holland, J. H. (1992). *Genetic algorithms.* Scientific American, 267(1), 66–73.

[4] Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Professional.

[5] Mirjalili, S., Mirjalili, S. M., & Lewis, A. (2014). *Grey wolf optimizer.* Advances in Engineering Software, 69, 46–61.

[6] Nelder, J. A., & Mead, R. (1965). *A simplex method for function minimization.* The Computer Journal, 7(4), 308–313.

[7] Fan, S.-K. S., & Zahara, E. (2007). *A hybrid simplex search and particle swarm optimization for unconstrained optimization.* European Journal of Operational Research, 181(2), 527–548.

End of Documentation

For issues or contributions, please contact the development team