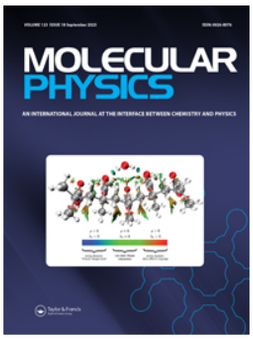


Taylor & Francis
Taylor & Francis Group

Molecular Physics

An International Journal at the Interface Between Chemistry and Physics

ISSN: 0026-8976 (Print) 1362-3028 (Online) Journal homepage: www.tandfonline.com/journals/tmph20

Benchmarking energy calculations using formal proofs

Ejike D. Ugwuanyi, Colin T. Jones, John Velkey & Tyler R. Josephson

To cite this article: Ejike D. Ugwuanyi, Colin T. Jones, John Velkey & Tyler R. Josephson (11 Aug 2025): Benchmarking energy calculations using formal proofs, Molecular Physics, DOI: 10.1080/00268976.2025.2539421

To link to this article: https://doi.org/10.1080/00268976.2025.2539421

Published online: 11 Aug 2025.

MOLECULAR PHYSICS e2539421
https://doi.org/10.1080/00268976.2025.2539421

Taylor & Francis
Taylor & Francis Group

THERMODYNAMICS 2024 CONFERENCE (BY INVITATION ONLY)

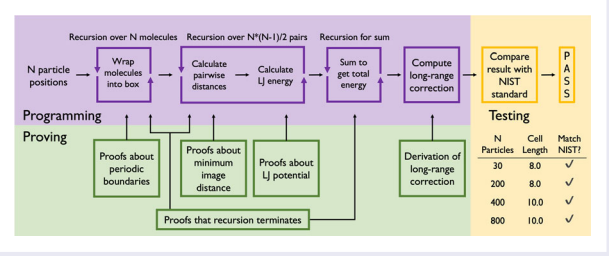
Benchmarking energy calculations using formal proofs

Ejike D. Ugwuanyi [a], Colin T. Jones [a], John Velkey [a] and Tyler R. Josephson [a,b]

[a]Department of Chemical, Biochemical, and Environmental Engineering, University of Maryland, Baltimore County, Baltimore, MD, USA;
[b]Department of Computer Science and Electrical Engineering, University of Maryland, Baltimore County, Baltimore, MD, USA

ABSTRACT

Traditional approaches for validating molecular simulations rely on making software open source and transparent, incorporating unit testing, and generally employing human oversight. We propose an approach that eliminates software errors using formal logic, providing proofs of correctness. We use the Lean theorem prover and programming language to create a rigorous, mathematically verified framework for computing molecular interaction energies. We demonstrate this in LeanLJ, a package of functions, proofs, and code execution software that implements Lennard-Jones energy calculations in periodic boundaries. We introduce a strategy that uses polymorphic functions and type classes to bridge formal proofs (about idealised Real numbers) and executable programs (over floating point numbers). Execution of LeanLJ matches the current gold standard NIST benchmarks, while providing even stronger guarantees, given LeanLJ's grounding in formal mathematics. This approach can be extended to formally verified molecular simulations in particular and formally verified scientific computing software, in general.



ARTICLE HISTORY
Received 20 May 2025
Accepted 15 July 2025

KEYWORDS
Formal verification; Lean 4; molecular simulations; functional programming

1. Introduction

Molecular simulations constitute an essential computational tool for understanding atomic-scale phenomena, playing a critical role in predicting the physicochemical properties underlying fields such as materials science, chemistry, and biophysics [1]. For instance, accurate simulations enable the study of molecular interactions that dictate gas adsorption behaviours in porous materials [2, 3], influence solvation dynamics in chemical solutions [4], and affect catalytic reactions on material surfaces [5]. Accurate modelling of particle interactions lies at the core of molecular simulations [6]. Among the most fundamental and extensively used models are the Lennard-Jones potential, describing van der Waals interactions, and Coulomb potentials, capturing electrostatic forces. The Lennard-Jones potential, specifically, finds significant application in representing non-bonded interactions in chemically relevant systems such as simple fluids [7], noble gas clusters [8], hydrocarbon fluids [9], and molecular adsorption phenomena in zeolites or metal-organic frameworks (MOFs) [10, 11]. It also describes the interaction between a pair of neutral atoms or molecules based on their distance [12–14], and efficiently captures the balance between attractive and repulsive forces [15, 16]. Coulomb interactions are particularly critical for describing electrolyte solutions [17], protein-ligand binding affinities [18], and charged colloidal systems [18, 19].

To accurately approximate infinite molecular systems from computationally manageable finite-sized simulations, periodic boundary conditions (PBC) are conventionally employed [20, 21]. In chemical simulations,

CONTACT Tyler R. Josephson tjo@umbc.edu

This article has been corrected with minor changes. These changes do not impact the academic content of the article.

Table 1. Errors in scientific computing software, and typical interventions. Our goal is to develop an approach to address syntax, runtime, and semantic errors in Lean at the 'editor' stage, before code is compiled.

Category of Error	Example	Intervention	Lean
Syntax	Not closing parentheses	Editor	Editor
Runtime	Accessing element in list that does not exist	Run program, program gives error message	Editor
Semantic	Missing a minus sign, transposing tensor indices	Human inspection of the code; test-driven development; observing anomalous behaviour	Editor
Floating-point/ Round-off	Subtracting small values from large values, ill-conditioned matrices	Modifying simulation methods, using double precision floats	–

PBC enables the study of bulk-phase properties without boundary effects, such as predicting phase behaviours in liquid water [22], ionic solutions [23], or polymer melts [24]. Furthermore, the minimum image convention ensures that computational resources are efficiently utilised by considering only the nearest periodic images in calculations [25], which is particularly important in simulations of dense chemical environments like ionic liquids or liquid crystal phases.

Software tools like LAMMPS and Gromacs [26–28] allow users to simulate the dynamics of large molecular systems. However, the sheer complexity of these software packages and the systems they intend to model presents challenges in making simulations transparent, reproducible, useable by others and extensible (TRUE) [29]. For example, the SAMPL Challenges (Statistical Assessment of the Modelling of Proteins and Ligands) [30] and the Industrial Fluid Properties Simulation Challenges [31, 32] task computational researchers to predict the solvation or binding free energies of small molecules or the thermophysical properties of fluids. Each year, researchers submit highly variable answers, reflecting differences in modelling choices by the researchers (e.g. force fields, simulation conditions, free-energy extrapolation strategies, etc.), as well as more hidden, subtle differences amongst software packages (e.g. default settings for managing Lennard-Jones cut-off and settings for Ewald summation). Projects such as the Molecular Simulation Design Framework (MoSDeF) [29, 33, 34] and the Molecular Sciences Software Institute (MolSSI) [35] address these issues by providing reproducible workflows for molecular simulation setup, and by teaching and promoting best practices in software development [36]. Simulation software can also be validated by comparing to benchmarks, such as those on the National Institute of Standards and Technology (NIST) Standard Reference Simulation Website (SRSW) [37].

We propose an alternative paradigm for improving reliability of molecular simulations. To illustrate, consider the taxonomy of programming errors in Table 1. The simplest are syntax errors: these are addressed immediately because the code cannot compile, the editor highlights the mistake, and the programmer fixes it. Runtime errors occur during code execution, and may arise when users run the program under conditions not anticipated by the software developers. Nonetheless, runtime errors typically provide a helpful error message pointing toward the source of the issue. The deepest issues are semantic errors in the *meaning* of the software: Python would not complain about misinterpreting a scientific principle or incorrectly transcribing math into code – it is simply not designed for that. Floating-point and round-off errors create numerical inaccuracies, since computers do not operate with infinite mathematical precision. These are addressed by judicious choices of simulation settings and algorithm choices, and by checking conditions like energy conservation after simulation completion [38].

In this work, we propose a strategy for catching syntax, runtime, and semantic errors at the 'editor' stage, namely, before the code is compiled. Our approach stems from the *formal methods* community in computer science, which seeks to prove when software is correct by construction before it is run (also known as static program analysis), unlike traditional testing, which checks for errors by running a program with different inputs. This approach is handy in areas where even small errors can have significant consequences, such as hardware design and critical software systems. A prominent example is the Pentium FDIV bug in Intel processors in the early 1990s, the subject of a multi-million dollar recall stemming from a few misplaced bits in chip software [39]. Now, formal verification approaches prove the correctness of such arithmetic operations in manufactured chips [40]. Our approach most closely resembles that of Selsam, *et al.*, who explored how formal methods can be applied to machine learning systems in Certigrad [41] (Figure 1). By proving the correctness of each step mathematically, this approach exposes errors that might otherwise slip through traditional empirical testing. They highlight the ability of theorem provers like Lean to eliminate entire classes of high-level errors that arise in complex software systems by enforcing correctness through formal reasoning. They demonstrate their approach by building a variational auto-encoder in Lean, proving properties about their implementation of stochastic gradient descent.

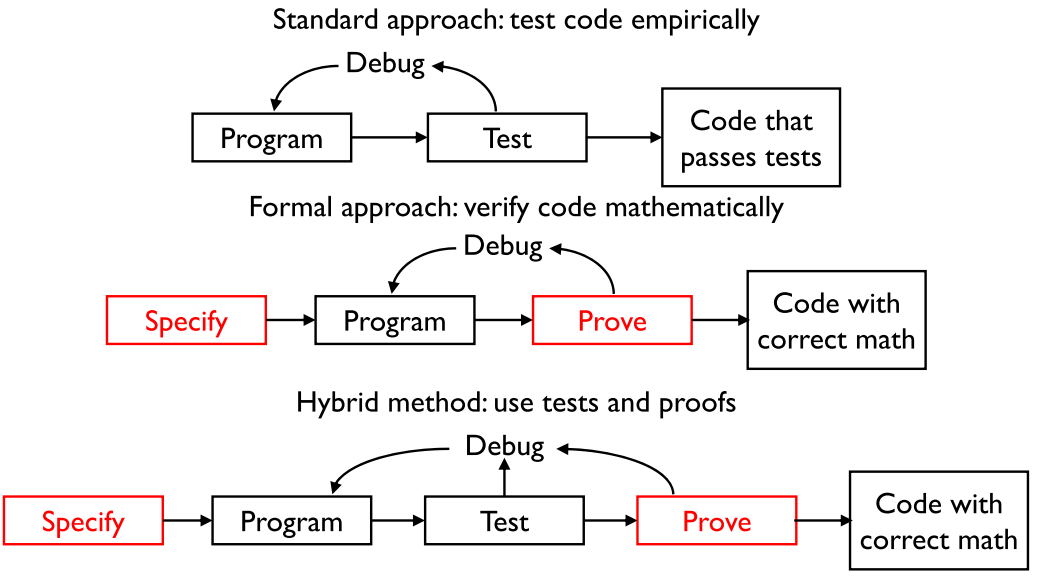


Figure 1. Comparison of code correctness approaches (adapted from [41]): the standard test-debug cycle, formal verification using proofs, and a hybrid method combining tests and formal proofs, that we adopt here.

Most prior work on formal methods has focussed on floating point operations [42]. In molecular simulations, these are typically insignificant, but they can lead to issues in certain settings, such as when programs are run with less precision to increase speed, or under extreme conditions. Tran and Wang [43], explored using interval arithmetic to model the propagation of these uncertainties in molecular dynamics simulations. Our work sets aside the imprecision of floating-point arithmetic, and instead focuses on verifying higher-level logic and mathematics. Incorporating interval arithmetic into our approach would in principle be possible, but these tools are currently in development [44].

Lean 4 is a theorem prover and functional programming language designed to write and verify mathematical proofs, as well as write formally-verified software [45]. Unlike traditional programming languages used for scientific computing (C, FORTRAN, Python, etc.), Lean provides a formally verified framework in which proofs of correctness can be explicitly constructed and checked [46]. We previously used Lean to formalise chemical physics [47]. Lean is also being used to formalise theories in high-energy physics [48]. We also recognise Tomáš Skřivan's ongoing SciLean project, which is working out methods for automated differentiation and efficient, array-based computations in Lean [49].

In our work [47], we showed how theories in science can be rigorously encoded using the Lean theorem prover, proving the correctness of the derivations, grounding them in the foundations of mathematics. We formalised derivations of the Langmuir and BET adsorption models, meticulously defining assumptions and derivations to ensure mathematical rigour. That work was limited to *proofs* in Lean – we extend that now to executable *programs* with formally-verified properties.

In this paper, we first present a familiar, informal description of Lennard-Jones energy calculations of periodic fluids (Section 3). We then highlight the proof components (especially definitions and theorems) for the formal implementation in Lean (Section 4). Section 5 describes how we implement these energy calculations in Lean, which requires novel approaches using functional programming, type polymorphism, and monads. Section 6 compares our calculations with the results from the NIST SRSW benchmarks [37].

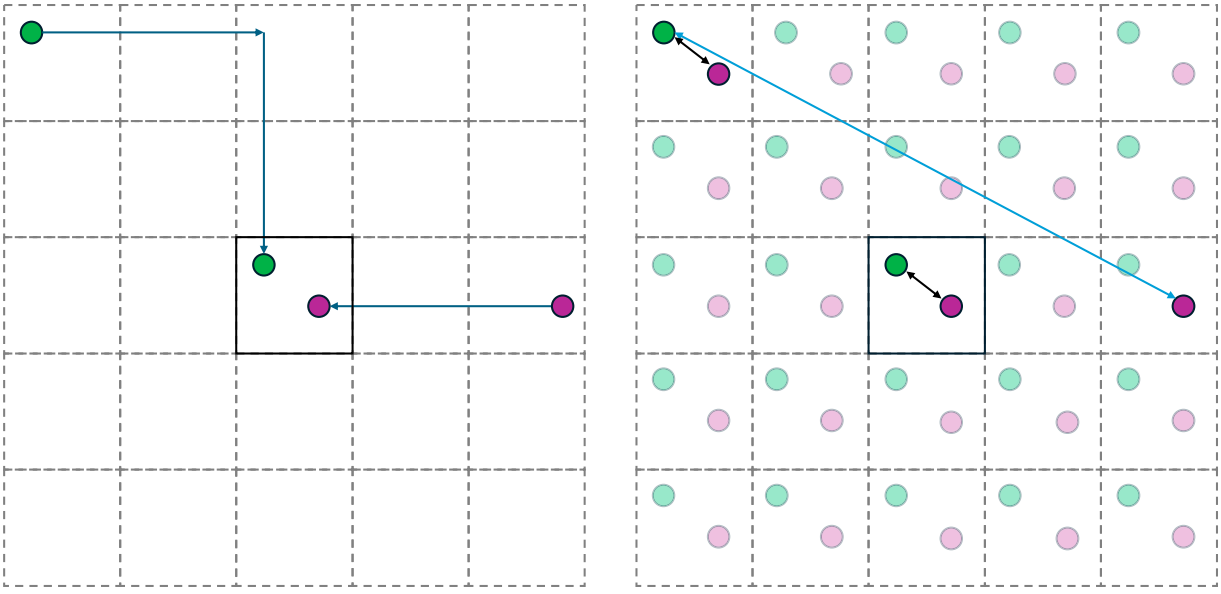
2. Methods

We implemented this using Lean version 4.16.0-rc2, Mathlib 4 at commit `e1a3d4c`, and Visual Studio Code version 1.96. The source code is available in LeanLJ Repository.

3. Informal description of the molecular simulation system

The Lennard-Jones system is modelled as a collection of N particles confined within a cubic simulation box of side length L. The position of each particle is represented as a vector in a three-dimensional space, $\mathbf{r}_i = (x_i, y_i, z_i)$, where $i = 1, 2, \ldots, N$. The interaction between particles is governed by the Lennard-Jones potential:

$$V_{\mathrm{LJ}}(r_{ij}) = 4\varepsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right] \qquad (1)$$



(a) Periodic boundary conditions. (b) Minimum image convention.

Figure 2. (a) Periodic boundary conditions: Particles outside the central cubic simulation box are wrapped back into it. Arrows represent the wrapping process along the directions. (b) Minimum image convention: Particles interact with the nearest periodic image, ensuring the shortest distance is used in calculations. The Euclidean distance (blue) is not used; the minimum image distance (black) is used instead, which is equivalent to the minimum image distance between wrapped particles.

where r_{ij} is the distance between particles i and j, ε represents the depth of the potential well, and σ is the characteristic length scale.

Periodic boundary conditions (PBCs) are applied to simulate an infinite system as shown in the equation for particle coordinates in the x, y, and z axes, respectively (Figure 2).

$$x_{i\_\text{wrapped}} = x_i - L \cdot \text{round}\left(\frac{x_i}{L}\right) \tag{2}$$

$$y_{i\_\text{wrapped}} = y_i - L \cdot \text{round}\left(\frac{y_i}{L}\right) \tag{3}$$

$$z_{i\_\text{wrapped}} = z_i - L \cdot \text{round}\left(\frac{z_i}{L}\right) \tag{4}$$

Because the LJ particles are in a system with PBCs, the distance between two particles is not the Euclidean distance, but the minimum image distance, the shortest pairwise distance considering the periodicity of the box as given in the equation below.

$$r_{ij} = \sqrt{\begin{aligned}&\left(\Delta x - L \cdot \text{round}\left(\frac{\Delta x}{L}\right)\right)^2 \\ &+\left(\Delta y - L \cdot \text{round}\left(\frac{\Delta y}{L}\right)\right)^2 \\ &+\left(\Delta z - L \cdot \text{round}\left(\frac{\Delta z}{L}\right)\right)^2\end{aligned}} \tag{5}$$

To improve computational efficiency, a cut-off radius r_c is introduced. Interactions are considered only for particle pairs that satisfy $r_{ij} \leq r_c$, with contributions beyond this radius set to zero. This truncation neglects a relatively minor contribution to the potential energy, depending on the cut-off radius r_c as shown in Figure 3.

$$V(r) = \begin{cases} V_{\text{LJ}}(r), & r \leq r_c \\ 0, & r > r_c \end{cases} \tag{6}$$

The Lennard Jones potential function is defined in part: When $r \leq r_c$, the potential is calculated as $4\epsilon[(\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^6]$, which captures both short-range repulsion and long-range attraction. For $r > r_c$, the potential is set to zero, reflecting the computational practice of truncating interactions beyond the cut-off to save resources. In addition, the inclusion of a cut-off distance makes the function practical for large-scale molecular systems.

The total internal energy U_{pair} is calculated by summing the energies of the pairs of particles interacting. This is given by the following equation, where $V(r_{ij})$ is the simulated pair potential:

$$U_{\text{pair}} = \sum_{i=1}^{N} \sum_{j=i+1}^{N} V(r_{ij}), \quad \text{where } r_{ij} \leq r_c. \tag{7}$$

The neglected part of the Lennard-Jones potential can be approximately included by incorporating a 'Long-Range

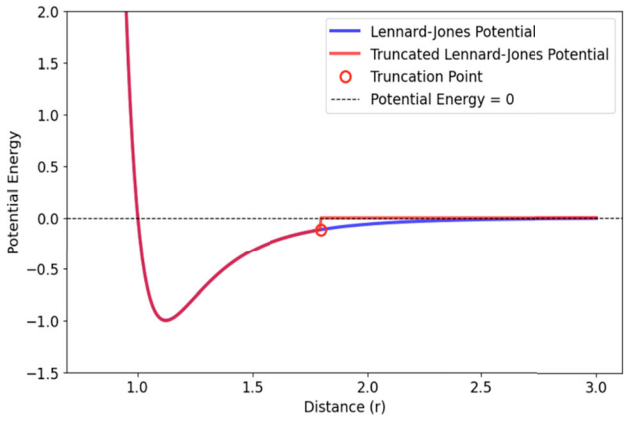


Figure 3. The Lennard-Jones potential, truncated at the cut-off.

Correction' (LRC), also known as 'tail corrections'. This incorporates the ensemble-averaged energy contribution of the particles beyond the cut-off radius, in a manner that only depends on the density of the system and does not require pairwise distance calculations [25]. The LRC is given by:

$$U_{LRC} = \frac{1}{2} 4\pi\rho \int_{r_c}^{\infty} r^2 V(r)\, \mathrm{d}r \tag{8}$$

where ρ is the density of the system, r_c is the cut-off radius, and $V(r)$ is the pairwise energy function.

When $V(r) = 4\epsilon((\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^{6})$, this integrates to:

$$U_{LRC} = \frac{1}{2} 4\pi\rho \int_{r_c}^{\infty} r^2 V_{LJ}(r)\, \mathrm{d}r \tag{9}$$

$$= \frac{1}{2} 4\pi\rho \int_{r_c}^{\infty} r^2 4\epsilon \left(\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right) \mathrm{d}r \tag{10}$$

$$= \frac{8\pi\epsilon\rho}{r_c^3} \left(\frac{\sigma^{12}}{9r_c^6} - \frac{\sigma^6}{3} \right) \tag{11}$$

4. Formally defining the mathematics

The previous section was an *informal* description of these concepts; now, we turn to a *formal* description, expressed as Lean code. Lean provides a structured framework to rigorously define the components of our system and prove their properties. Figure 4 illustrates our code and the means by which it is verified. In this section and the next, we describe the components of the system. We start by illustrating Lean's capabilities as a theorem prover.

4.1. Introduction to Lean syntax

Here are a few examples to illustrate the syntax of Lean 4. Lean's basic objects include types, tactics, definitions, and theorems; we do not introduce any custom types or tactics in this work, so we will focus on definitions and theorems. A definition has the following basic structure[1]:

```
def name_of_object (p1 : parameter1) ... : type_of_object := the_def_of_the_object
```

A theorem (or equivalently, a lemma) has the following basic structure:

```
theorem name_of_theorem (p1 : parameter1) ... (a1 : assumption1) ... :
    thing_to_be_proved := by
  proof
```

Lean's rich type system enables theorems to be stated and proved; while the user writes code, Lean effectively checks the types of the objects in the code for consistency. Type-checking a theorem object amounts to validating whether it is true. As the user writes the steps in a theorem's proof, Lean provides a concise overview of the current proof goal, as well as the current state of the assumptions and parameters. This information is presented in the 'Lean Infoview' in VS Code, in what is known as a *tactic state*, which is organised as follows:

```
p1 : parameter1
...
a1 : assumption1
⊢ current_state_of_goal
```

To learn more about Lean, we highly recommend the textbooks 'Mechanics of Proof' by Heather Macbeth [50] and 'Functional Programming in Lean' by David Christiansen [46].

4.2. Lennard-Jones potential

We can write the Lennard-Jones potential energy function in multiple ways. In every case, we aim to formally define Equation (6), using a function that takes four parameters (ϵ, σ, r, and r_c) and returns the energy between a pair of particles.

The first version of this is `lj`. In this version, all parameters are type $\mathbb{R}$, for the Real numbers. Lean requires this definition to be prefaced with the `noncomputable` keyword. Lean tries to generate executable bytecode for its functions, but this is not possible, in general for the Real numbers. `noncomputable` signals to Lean that this definition is only for Lean to reason about in proofs. We intend to execute other versions of this function – see functions `lj_Float` and `lj_p` for computable LJ functions in Section 5.

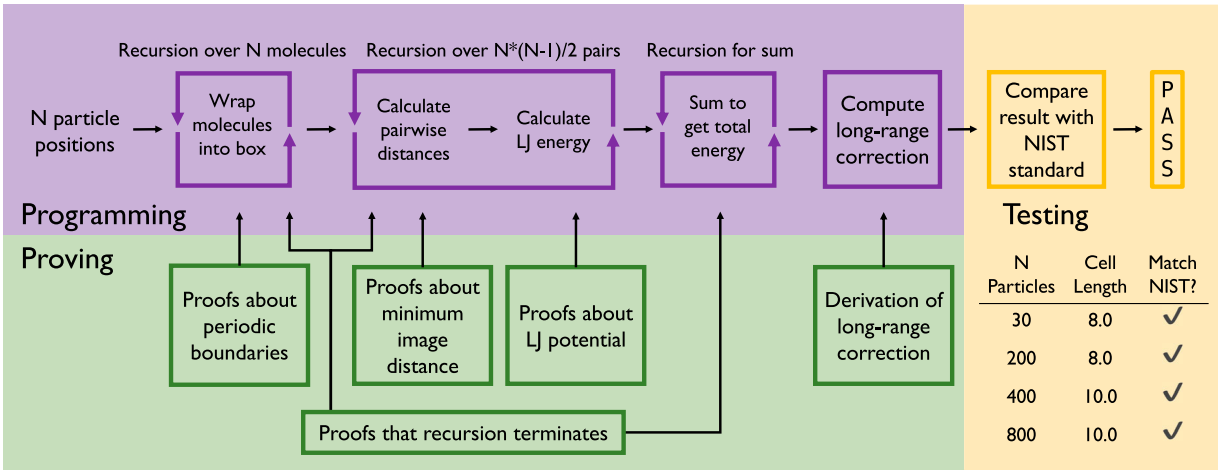


Figure 4. Workflow of the Lennard-Jones energy calculation using LeanLJ. The process involves recursive programming, formal proofs, and comparison to NIST benchmarks.

```
noncomputable def lj (r r_c ε σ : ℝ) : ℝ :=
  if r ≤ r_c then
    4 * ε * ((σ / r) ^ 12 - (σ / r) ^ 6)
  else
    0
```

While `lj` may be a natural way to write Equation (6), alternative formulations are typically used for efficient molecular simulations. For instance, r^{-3} can be computed first, which is then squared to obtain r^{-6}, which can be squared again to obtain r^{-12}. Our function `lj_Real` reflects this idea, using intermediate variables like `r6` and `r12`. Because `lj_Real` is also a function of Real numbers, it is also noncomputable.

```
noncomputable def lj_Real  (r r_c ε σ  : ℝ) : ℝ :=
  if r ≤ r_c then
    let r3 := (σ / r) ^ (3 : Nat)
    let r6 := r3 * r3
    let r12 := r6 * r6
    4 * ε * (r12 - r6)
  else
    0
```

Lean allows us to formally prove the equivalence of these two forms, as shown in the theorem `lj_eq`, allowing us to use either representations confidently. This capability enables not only correctness, but also flexibility in implementing the most efficient forms for simulation. Keep in mind that we *do not* address floating-point or round-off errors; this guarantee holds only for idealised functions over Real numbers, which have infinite precision. If a more-efficient version of a function is mathematically equivalent (over Reals) to a base case, but leads to more round-off errors, that would not be detected in our formulation. A more efficient version that is not mathematically equivalent (e.g. it invokes an approximation) would be shown to be distinct by this approach.

```
theorem lj_eq (r r_c ε σ : ℝ) : lj_Real r r_c ε σ = lj r r_c ε σ := by
  unfold lj_Real
  unfold lj
  simp
  ring_nf
```

The theorem `lj_eq` formally proves that `lj_Real r r_c εσ = lj r r_c εσ`. This illustrates the syntax of Lean functions: unlike Python, which uses parentheses to denote function application (e.g. `lj_Real(r, r_c, epsilon, sigma)`), Lean uses simple whitespace. In the expression `lj_Real r r_c εσ`, each argument is applied to the function from left to right, separated by spaces. Thus, `lj_Real r r_c εσ` represents 'apply the function `lj_Real` to these four arguments'. This compact syntax is helpful in mathematical reasoning, where function application is so pervasive.

We can also prove various mathematical properties of our LJ function. Theorem `cutoff_behaviour` states that for any $r > r_c$, the value of the Lennard-Jones potential is zero. (The way to read this theorem, is 'for Real numbers ϵ, σ, r, and r_c, assuming $r > r_c$, this function evaluates to zero'). This reflects the practice of truncating the potential beyond the cut-off distance.

```
theorem cutoff_behaviour (ε σ r r_c : ℝ) (h : r > r_c) :
    lj_Real ε σ r r_c = 0 := by
  unfold lj_Real
  simp [if_neg (not_le_of_gt h)]
```

Theorem `ljp_eq_le` establishes that, in $0 < r \leq r_c$, the Lennard-Jones potential is $4\epsilon[(\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^{6}]$. Lean

can use logical operators like ∀ (for all) for defining properties of functions.

```
theorem ljp_eq_le {r_c ε σ : ℝ} : ∀ r ∈ {r | r > 0 ∧ r ≤ r_c},
    lj_Real r r_c ε σ = 4 * ε * ((σ / r)^12 - (σ / r)^6) := by
  intro r hr
  have h_r_le_rc : r ≤ r_c := hr.2
  unfold lj_Real
  rw [if_pos h_r_le_rc]
  ring
```

We also prove the continuity of the function in this range, in Theorem `ljp_continuous_closed_domain` (for brevity, we just state the theorem here; the full proof is on GitHub). Continuity is essential in molecular dynamics simulations because forces are evaluated on the basis of energy gradients, and discontinuities can introduce artificial forces, destabilising numerical integration [7]. Importantly, we do not, indeed we *cannot*, prove that this function is continuous for the whole domain of r; the LJ function diverges at $r = 0$ and undergoes a step change at $r = r_c$. Researchers have implemented alternative truncation methods for the LJ function, such as the truncated and shifted LJ function or the linear force shift function, which would be continuous for all $0 < r$ [51]. These properties could be formalised in Lean, but in this work, we have focussed on the simple LJ function.

```
theorem lj_p_continuous_closed_domain (r_c ε σ : ℝ) :
    ContinuousOn (fun r => if r ≤ r_c then 4 * ε * (((σ / r) ^ 6) ^ 2 - (σ / r) ^ 6)
    else 0) {r | 0 < r ∧ r ≤ r_c} := by
```

4.3. Periodic boundaries

We follow the formulation in Allen and Tildesley [7] in defining functions for wrapping molecules according to periodic boundary conditions (PBCs) and calculating the minimum image distance. The periodic boundary function wraps a position from anywhere in space into the bounds of the simulation box. This function `pbc` takes in a one-dimensional position and box length and outputs a new position (all have type $\mathbb{R}$).

```
noncomputable def pbc_Real (pos boxLength : ℝ) : ℝ :=
  pos - boxLength * round (pos / boxLength)
```

We formally proved that the wrapped displacement produced by the periodic boundary condition function lies within the interval $[-L/2, L/2]$ for any Real coordinate p and positive box length L. This ensures that particles always interact with the nearest periodic image, which is a key assumption in molecular dynamics simulations. The proof was constructed in Lean by expressing the wrapped position as $L \cdot \delta$, where $\delta = \frac{p}{L} - \text{round}(\frac{p}{L})$, and rigorously showing that $|\delta| \leq \frac{1}{2}$, hence $|\text{pbc\_Real}(p, L)| \leq \frac{L}{2}$.

```
theorem abs_pbc_le (p L : ℝ) (hL : 0 < L) : |pbc_Real p L| ≤ L / 2 := by
  dsimp [pbc_Real]
  let δ := (p / L) - round (p / L)
  have h_eq : p - L * round (p / L) = L * δ := by
    rw [mul_sub]
    field_simp [hL.ne']
  rw [h_eq, abs_mul, abs_of_pos hL]
  have hδ : |δ| ≤ 1 / 2 := abs_diff_round_le_half (p / L)
  trans L * (1 / 2)
  · exact mul_le_mul_of_nonneg_left hδ hL.le
  · field_simp
```

4.4. Minimum image distance

In defining the minimum image distance, we found it more convenient to first define the squared minimum image distance, and then take the square root of that to obtain the minimum image distance. The box length `boxLength` and positions `posA` and `posB` are vectors in $\mathbb{R}^N$ (with $N = 3$) where each component corresponds to a coordinate in the respective dimension. This is specified using a vector type `Fin 3` $\to \mathbb{R}$.[2] The function iterates over each of the three dimension and computes a displacement, which is adjusted using the periodic boundary function `pbc_Real`. The adjusted displacements are squared and summed over all dimensions. In our `squaredminImageDistance_Real` function, the decide tactic is employed in each invocation of the vectors posB and posA, to prove to Lean that elements 0, 1, and 2 are in scope.

```
noncomputable def squaredminImageDistance_Real (posA posB : Fin 3 → ℝ) (boxLength :
    Fin 3 → ℝ) : ℝ :=
  let dx := pbc_Real (posB (0: Fin 3) - posA (0: Fin 3)) (boxLength (0: Fin 3))
  let dy := pbc_Real (posB (1: Fin 3) - posA (1: Fin 3)) (boxLength (1: Fin 3))
  let dz := pbc_Real (posB (2: Fin 3) - posA (2: Fin 3)) (boxLength (2: Fin 3))
  dx^2 + dy^2 + dz^2
```

We can prove a neat property of how the periodic boundaries interact with the minimum image distance – that the minimum image distance between arbitrary points in space is equivalent to the minimum image distance between those points, after being wrapped into the simulation box. This is stated in theorem `squaredminImageDistance_theorem`, which requires an inline invoking a λ function to iterate over the box dimensions (for brevity, the proof steps are omitted here, but available on GitHub). This only holds for non-zero box lengths.

```
theorem squaredminImageDistance_theorem (boxLength posA posB : Fin 3 → ℝ)
    (hL : ∀ i, boxLength i ≠ 0) squaredminImageDistance_Real boxLength posA posB =
    squaredminImageDistance_Real boxLength (λ i => pbc_Real (posA i) (boxLength i))
    (λ i => pbc_Real (posB i) (box_length i)) := by
    ...
```

Finally, the function `minImageDistance_Real` calls the `squaredminImageDistance` function,

and takes the square root to obtain the minimum image distance.

```
noncomputable def minImageDistance_Real (posA posB boxLength : Fin 3 → ℝ) : ℝ :=
  (squaredminImageDistance_Real posA posB boxLength).sqrt
```

We can also prove that computed distances between particles are guaranteed to be non-negative in all applications of the minimum image convention; this can be useful when non-negativity is invoked in proofs about energy calculations.

```
theorem minImageDistance_real_nonneg ( posA posB boxLength : Fin 3 → ℝ) :
    0 ≤ minImageDistance_Real  posA posB boxLength := by
  unfold minImageDistance_real
  apply Real.sqrt_nonneg
```

We also proved that the minimum image distance between a particle and itself is always zero (theorem `minImageDistance_self`).

```
theorem minImageDistance_real_self (pos boxLength : Fin 3 → ℝ) :
    minImageDistance_Real pos pos boxLength = 0 := by
  unfold minImageDistance_Real squaredminImageDistance_Real
  have h0 : pbc_Real (pos (0: Fin 3) - pos (0: Fin 3)) (boxLength (0: Fin 3)) = 0 := by
    simp [pbc_Real, sub_self, zero_div, round_zero, mul_zero, sub_zero]
  have h1 : pbc_Real (pos (1: Fin 3) - pos (1: Fin 3)) (boxLength (1: Fin 3)) = 0 := by
    simp [pbc_Real, sub_self, zero_div, round_zero, mul_zero, sub_zero]
  have h2 : pbc_Real (pos (2: Fin 3) - pos (2: Fin 3)) (boxLength (2: Fin 3)) = 0 := by
    simp [pbc_Real, sub_self, zero_div, round_zero, mul_zero, sub_zero]
  rw [h0, h1, h2]
  simp
```

While the above formulations of `pbc_Real` and `minImageDistance_Real` lead to valid computations and proofs, we are somewhat dissatisfied with the semantics. The `pbc_Real` function operates on *particle positions* (i.e. x_i), wrapping them inside the box from outside. When this function is applied in the `minImageDistance` function, it is being applied to a *difference* between particle positions (i.e. $x_j - x_i$). Lean does not complain, because in both cases, these are just real numbers, and everything checks out, but a displacement is nonetheless not the same thing as a position. There may be a way to make this even more rigorous, by defining a custom type for positions and restricting the `pbc_Real` function to only operate on such a type, but we kept our approach simpler for now.

4.5. Long-range corrections

The long-range correction, given in Equation (8), is computed using the function `U_LRC`, which depends on ρ, ϵ, σ, and rc.

```
noncomputable def U_LRC_Real (ρ ε σ rc : ℝ) : ℝ :=
  (8 * π * ρ * ε) * ((1/9) * (σ ^ 12 / rc ^ 9) - (1/3) * (σ ^ 6 / rc ^ 3))
```

We can prove that this function follows from the integral definition of U_{LRC}, Equation (11). The integral `∫ (r : ℝ) in Set.Ioi rc` is interpreted using measure theory, and refers to an integral over the set `Set.Ioi rc`, which is the open interval (r_c, ∞). We state the theorem here and omit the proof for brevity, the full proof is available on GitHub.

```
theorem long_range_correction_equality (hr : 0 < rc) (ρ ε σ : ℝ) :
    (2*π*ρ) * ∫ (r : ℝ) in Set.Ioi rc, 4*ε * (r^2 * (((σ / r)^12) - ((σ / r)^6))) =
    U_LRC ρ ε σ rc π := by
    ...
```

5. Code execution

Combining formal proofs with numerical computation is central to this work. In this section, we elaborate on three aspects of programming in Lean. Section 5.1 introduces the function for energy summation; in Lean, this must be recursive instead of based on traditional for loops. Section 5.2 highlights our approach for bridging computations and proofs using polymorphic functions. Section 5.3 describes Lean's approach to input and output.

5.1. Functional programming

Traditional molecular simulation software is implemented using imperative programming languages (like C and FORTRAN), but Lean is a functional programming language (like Haskell). Imperative programs are about 'doing' (following a step-by-step procedure), while functional programs are about 'being' (defining what a function *is*, which in Lean, ultimately enables proofs about it). Imperative programming is susceptible to 'side effects' that are avoided in functional programming, reducing security risks and improving rigour. Functional programming avoids mutable data types; rather than updating (mutating) existing variables, such as assigning `x = x+1`, when new things must be computed, new variables are assigned. Lean 4 does support some imperative design patterns, but to get guarantees that come from proofs, writing code in a functional style is generally preferred.

One of the most stark differences (and most relevant for molecular simulations) between imperative and functional programming is the use of for- and while-loops. Pairwise energy calculations typically first loop over particles i from 1 to N, then over particles j from $i+1$ to N [7, 51]. This 'double for loop' can be expressed in Lean as follows:

```
-- Imperative style, double for loop
def total_energy_loop (positions : List (Fin 3 → Float)) (boxLength : Fin 3 → Float)
    (cutoff ε σ : Float) : Float :=
  Id.run do
    let mut energy := 0.0
    for i in [0 : positions.length] do
      for j in [i+1 : positions.length] do
        let r := minImageDistance positions[i]! positions[j]! boxLength
        let e := lj_Float r cutoff ε σ
        energy := energy + e
    return energy
```

Lean sets up for loops using the keywords `Id.run`, `do`, and `for`, and uses `mut` to denote `energy` as a mutable variable. To be clear, Lean isn't actually running imperative code; `do` notation is 'syntax sugar' for purely functional code, which is guaranteed to be free of side effects. Lean 4's 'functional but in-place' paradigm for memory management makes it quite efficient compared to other functional programming languages [52]. Lean converts the above for loops into recursive functions. In the following function we illustrate, making the recursion explicit:

```
-- Recursive style
def total_energy_recursive (positions : List (Fin 3 → Float))
    (boxLength : Fin 3 → Float) (cutoff ε σ : Float) : Float :=
  let numAtoms := positions.length
  let rec energy : Nat → Nat → Float → Float
    | 0, _, acc => acc
    | i+1, 0, acc => energy i (i - 1) acc
    | i+1, j+1, acc =>
      let r := minImageDistance positions[i]! positions[j]! boxLength
      let e := lj_Float r cutoff ε σ
      energy (i+1) j (acc + e)
  energy numAtoms (numAtoms - 1) 0.0
```

Here, a recursive function `energy` is defined locally, as well an accumulation variable `acc`. In the central function call, `energy (i+1) j (acc + lj_Float r r_c ϵσ)`, `energy` adds one LJ energy contribution to the value of `acc`, using particle indices $i+1$ and j to obtain the distance r. The remaining conditions handle increments on the edge cases.

However, we found neither of these functions to be particularly amenable to proofs. We are particularly looking for a proof that the total number of pairs in the system is $N*(N-1)/2$. To do so, we defined a helper function that generates the pairs. Our function `pairs` returns a list of pairs of atom indexes (e.g. [(0,1), (0,2), (1,2)] when $n=3$).

```
-- Helper function to generate pairs
def pairs (n : Nat) : List (Nat × Nat) :=
  (List.range n).flatMap fun i =>
    (List.range' (i + 1) (n - (i + 1))).map fun j => (i, j)
```

We prove the length of this list is $N*(N-1)/2$ in `pairs_length_eq` in a long proof that uses mathematical induction (full proof available on LeanLJ GitHub). In the absence of test data, such a proof could provide confidence that the implementation is correct. We anticipate that such proofs would be useful for validating energy calculations over molecules with complex exclusion rules (like excluding adjacent up to 1-4 interactions in molecules), and higher-order interactions, like 3-body potentials.

```
theorem pairs_length_eq (n : Nat) : (pairs n).length = n * (n - 1) / 2 := by
```

We use `pairs` in `total_energy_pairs`, a function that uses a simpler version of the recursive style to accomplish this summation. This uses `foldl`, a function from functional programming that recursively applies a function to all elements of a list and accumulates the result, 'folding' them together from the left.

```
-- Total energy function using pairs
def total_energy_pairs (positions : List (Fin 3 → Float))
   (boxLength : Fin 3 → Float) (cutoff ε σ : Float) : Float :=
 let n := positions.length
 let indexPairs := pairs n
 indexPairs.foldl (fun acc (i, j) =>
   let r := minImageDistance_Float positions[i]! positions[j]! boxLength
   acc + lj_Float r cutoff ε σ
 ) 0.0
```

Lean automatically checks functions for termination, which is quite important for recursive functions, lest they get trapped in an infinite loop. These are verified formally using tactics that operate behind-the-scenes; only in more complicated cases, in which Lean cannot infer termination automatically, will the user be prompted to write a termination proof. Likewise, Lean also ensures that array indexing is safe, preventing all runtime errors associated with accessing out-of-bounds indices. In developing this code, we found execution and comparing to the NIST tests was valuable for developing the logic of the loops and the recursion, as it enabled quick feedback. Our code also incorporates tail recursion to facilitate efficient execution [46].

5.2. Polymorphism

In Lean, we can define functions specifically for Real numbers ($\mathbb{R}$), which allows us to prove mathematical properties, or for floating-point numbers (`Float`), which enables efficient numerical computation. However, these separate implementations create a trade-off: the `Real` version is *non-computable*, meaning it cannot be executed in actual simulations, while the `Float` version is not suitable for formal proofs, as floating-point arithmetic lacks the necessary mathematical structure (in the typical standard for floating point addition, IEEE 754, $0.1+0.2\neq 0.3$). To bridge this gap, *polymorphic functions* are used, allowing the same definition to work for multiple types (Figure 5). By introducing a generic type α that can subsume both `Real` and `Float`, we ensure that our function can operate on both Reals ($\mathbb{R}$) for proofs and floats (`Float`) for computations.

To illustrate, consider the function `pbc` (Figure 6), which wraps a particle's position into the simulation box using periodic boundary conditions. Section 4 showed `pbc_Real`, which operates on position and box length with type $\mathbb{R}$; `pbc` operates on position and box length with generic type α. We tell Lean more about what α

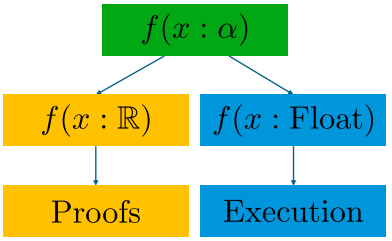


Figure 5. How polymorphic functions link proofs (over idealised Real numbers) with execution (over floating point numbers). The polymorphic function *f* is defined for *x* with generic type α; proofs about *f* can be written when *x* is Real, and computations with *f* can be executed when *x* is a float.

can be, using *type classes* and *instances*. Specially, `pbc` is defined for any type α that 'knows how to' subtract, multiply, divide, and round. These capabilities are provided through the type classes `HSub`, `HMul`, `HDiv`, and `HasRound`. For example, the type class `HSub [ααα]` requires that there exists a definition of subtraction between two members of α that would output a third member of α. `HSub`, `HMul`, and `HDiv` are all defined in Mathlib; for rounding, we defined a custom type class, since Mathlib did not already define that connection. This approach allows our definition of `pbc` to be used in two very different ways: with Real numbers for formal proofs, and with floating-point numbers for actual simulations.

Prefacing each function with a long list of instances is rather cumbersome. We can group all of these instances together into a single type class, `RealLike`, named thus because it is 'like' Real numbers. `RealLike` is ultimately compatible with Floats, but it doesn't have all the properties of the Real numbers (after all, it cannot use functions we did not attach to the type class). The `RealLike` type class captures the essential features of types that behave like real numbers, making it easier to write numeric code that works across different types such as `Float` or `Int`. It includes basic operations like addition, subtraction, multiplication, division, negation, and exponentiation with natural numbers that appear in our equations (1, 3, 9, etc.). It also provides comparisons ($\leq$, $<$) that return `Bool`, along with constants like zero and one, and ways to convert from natural and integer literals. In addition, `RealLike` extends two smaller type classes – `HasSqrt` for square roots (used in distance calculations) and `HasRound` for rounding (used in periodic boundaries) so that any type marked as `RealLike` is also expected to support those operations. This setup allows us to write clean, reusable, and type-safe numeric functions without tying them to one specific number type.

We note that this type class approach admits a small possibility of error that Lean will not catch. The `RealLike` type class manually links certain Real-typed and Float-typed functions, and mistakes in RealLike won't be flagged by Lean. For example, a subtle error in which one rounding function rounded 0.5 *down* while another rounded 0.5 *up* would not be detected by Lean. In fact, egregious errors linking a square root and a cube root are also technically possible, so human oversight remains necessary with this approach. Essentially, the `RealLike` type class defines which Real and Float functions are *semantically* equivalent, and the human is responsible for ensuring correct semantics. The Lean community is developing more systematic solutions, such as the ComputableReal project led by Alex Meiburg, which provides a more principled, computable representation of real numbers that integrates cleanly with existing numeric type classes [53].

```
class RealLike (α : Type) extends HasRound α, HasSqrt α where
  add : α → α → α
  sub : α → α → α
  mul : α → α → α
  div : α → α → α
  neg : α → α
  pow : α → Nat → α
  le : α → α → Bool
  lt : α → α → Bool
  zero : α
  one : α
  ofNat : Nat → α
  ofInt : Int → α
```

For some more examples, we provide the implementations for the Lennard-Jones potential in three forms: the *polymorphic* version (α), the *Real number* version ($\mathbb{R}$), and the *floating-point* version (`Float`).

```
-- Polymorphic version: Works for both ℝ and Float
def lj_p {α : Type} [RealLike α] (r r_c ε σ : α) : α :=
    if r ≤ r_c then
      let r3 := (σ / r) ^ (3 : Nat)
      let r6 := r3 * r3
      let r12 := r6 * r6
      4 * ε * (r12 - r6)
    else
      0
```

```
-- Real number version: Allows formal proofs but cannot compute
noncomputable def lj_Real  (r r_c ε σ  : ℝ) : ℝ :=
    if r ≤ r_c then
      let r3 := (σ / r) ^ (3 : Nat)
      let r6 := r3 * r3
      let r12 := r6 * r6
      4 * ε * (r12 - r6)
    else
      0
```

```
-- Floating-point version: Can compute but lacks proof capabilities
def lj_Float (r r_c ε σ : Float) : Float :=
    if r ≤ r_c then
      let r3 := (σ / r) ^ (3 : Nat)
      let r6 := r3 * r3
      let r12 := r6 * r6
      4 * ε * (r12 - r6)
    else
      0
```

```
def pbc (position boxLength : α)
    [HSub α α α] [HMul α α α] [HDiv α α α] [HasRound α] : α :=
    position - boxLength * (HasRound.pround (position / boxLength))
```

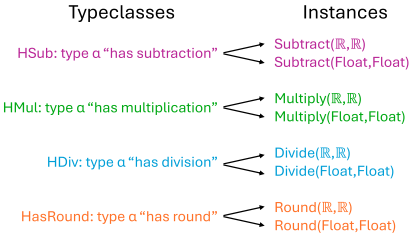


Figure 6. Explanation of the polymorphic `pbc` function. The function is defined over a generic type α, and the required operations – subtraction, multiplication, division, and rounding – are expressed through type classes: `HSub`, `HMul`, `HDiv`, and `HasRound`. Each type class specifies that the type α must support a given operation. For example, `HSub`$\alpha\alpha\alpha$ means α must support subtraction with two α inputs returning an α result. Concrete instances, such as `Float` and $\mathbb{R}$, implement these type classes to enable polymorphic behaviour. This allows `pbc` to work with different numeric types, as long as they satisfy the required operations.

Ultimately, we use `RealLike` to define polymorphic versions of *all* executable functions in the overall execution flow (Figure 4) and connect them to their Real counterparts. The polymorphic version (with type `RealLike`) of each function is that which is ultimately executed; the Real version or the polymorphic version is used in the proofs.

5.3. Input and output in Lean

Most of Lean is developed in terms of pure functions, whose behaviour can be guaranteed because the argument types limit the domain of the function inputs. By chaining pure functions with pure functions through-and-through, Lean guarantees there are no side effects. But input/output (IO) operations cannot have the same guarantees. For instance, if one writes a molecular configuration file to disk, then reads it back in, one cannot guarantee that some other process modified it in the meantime.

But to be a useful programming language, Lean must nonetheless have IO. Lean separates this cleanly from its pure functions and math libraries, implementing it in the `IO` monad. This essentially serves as a bridge between the messy, 'outside' world and the safe, pure functions inside Lean (Figure 7).

Monads are used to handle many kinds of computation patterns in a clean and consistent way, such as optional values, errors, and non-determinism. For example, the `Option` monad handles missing values, the `Except` monad deals with errors without crashing, and the `List` monad allows multiple possible results from

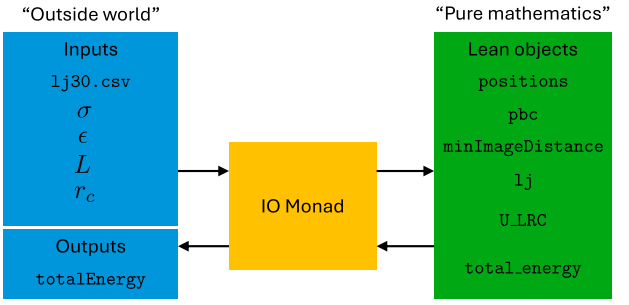


Figure 7. The IO Monad as a bridge that links the verified, pure functions in Lean with the messy real world, where data and simulation inputs reside. The CSV parser uses the IO Monad to read the particle coordinates from lj30.csv into the Lean object `positions`.

a single computation. They are central in functional programming, but are encountered less often in imperative languages; the interested reader can learn more here [46].

To import the configuration files from the NIST SRSW, we first saved them as comma-separated values (CSV) files. We adapted the CSV reader and used it to parse each configuration. In addition, users are asked to manually enter simulation parameters such as the cut-off radius, σ, ε, and the length of the box through the terminal. These user inputs and file reads are examples of interaction with the 'outside world', and are handled explicitly in Lean using the IO monad. This makes it clear which parts of the program remain exposed to sources of error – our setup does *not* provide guarantees against sources of error on the 'outside' of IO; if an incorrect value for σ were input, Lean would not catch it. This would be a form of semantic error (Table 1) that our current implementation does not avoid.

Proofs in Lean only provide guarantees about pure functions; errors in the I/O layer cannot be validated in this manner. This is why we advocate for both proofs and tests (Figure 1). For instance, while developing this application, our first approach for reading the configuration failed to read all atoms, leading to incorrect energy calculations. Our proofs do not catch bugs like these, but the tests do.

6. Results

To evaluate our implementation, we compare the pairwise interaction energy (U_{pair}) and long-range correction (LRC) values computed using our Lean code with the NIST Standard Reference Simulation Website (SRSW) benchmark values [37] for LJ particles in a cubic box (Table 2). The results show exact agreement for all four systems, within the number of digits provided by NIST.

Table 2. Comparison of LRC and U_{pair} energy calculations from NIST SRSW [37] and LeanLJ for various particle counts. Energies are reported in scientific notation (reduced units), with one more digit than NIST.

Particles	U_{pair} (Lean)	U_{pair} (NIST)	LRC (Lean)	LRC (NIST)
30	−1.67903E+01	−1.6790E+01	−5.45166E−01	−5.4517E−01
200	−6.90004E+02	−6.9000E+02	−2.42296E+01	−2.42296E+01
400	−1.14666E+03	−1.1467E+03	−4.96222E+01	−4.9622E+01
800	−4.35154E+03	−4.3515E+03	−1.98488E+02	−1.9849E+02

7. Discussion and outlook

In this study, we developed pairwise energy calculations in Lean and compared our results with the values provided by the NIST SRSW benchmark. Our calculations agree to machine precision with the NIST reference values. To be clear, our confidence in our system does *not* stem from its agreement with the NIST benchmark, rather from the theorems we have proved in Lean that certify that the functions in LeanLJ have those specified mathematical properties. For instance, the `pbc` function guarantees that all wrapped particles lie in the interval $[-L/2, L/2]$, and the derivation of the function computing long-range corrections is validated mathematically. We assert that LeanLJ is a more reliable benchmark than the NIST SRSW, at least for the components of the benchmark we have addressed. LeanLJ could be validated even further by adding to the list of proved theorems about current functions.

We consider it helpful to reflect on the remaining sources of uncertainty in our code – considering what we have verified, what could still be wrong? First, we are trusting in the axioms of mathematics, as expressed in Lean's core; errors here might compromise Mathlib, on which we depend. Second, our approach to polymorphism exposes us to mistakes in our `RealLike` type class, as we described in Section 5.2; Lean does not check to ensure that `RealLike` links the correct Float- and Real-type functions. Third, when we developed both polymorphic *and* floating-point versions of the same function for illustration purposes; we sometimes mistakenly called the floating-point version in execution instead of the polymorphic version. In this case, proofs for that function are not technically connected to the execution – since Lean does not require our polymorphic code design, it does not flag such mistakes (these could be eliminated by not defining Float and RealLike versions of the same functions). Fourth, we are still exposed to errors in input/output (Section 5.3), and in defining system-specific parameters, such as the force field parameters; these are mitigated by the testing, but do not prevent a user from inputting incorrect parameters for calculations outside the scope of the NIST benchmarks. A fifth source would be vulnerabilities in the broader operating system in which the code is executed. Nonetheless, traditional molecular simulation have far more possibilities for errors, such that many of these concerns are not considered in typical conversations about software correctness.

More broadly, this work demonstrates how Lean can provide a new paradigm for computational molecular simulations, where the results and the entire computational process are provably correct. Logical steps to build on this framework include implementing support for triclinic simulation boxes, Ewald summation for Coulomb interactions, neighbour lists to improve computational efficiency, and of course, integrating Newtons equations of motion to evolve particle trajectories. Some of these are matters of implementation (triclinic cells), but others will involve grappling yet-unresolved questions of how to handle various approximations in a formal environment, such as how to precisely describe the conditions under which neighbour lists can be trusted.

Nonetheless, we believe that Lean can be used (in principle) to implement molecular simulation software with all the features of the established packages used in daily practice. The user interface would not need to be more complex; the proofs and theorems would all be handled in the back-end. Furthermore, this could be integrated with software frameworks like MoSDeF [33, 54], that generate input files for multiple simulation engines. This would help address reproducibility and rigour at the 'outside world' layer (Figure 7) in which the force field parameters, molecular specifications, and configuration are defined. This would also enable rigorous evaluation of traditional software using Lean software as a benchmark; MoSDeF could create inputs for both verified Lean software as well as for LAMMPS, and deviations between the respective outputs may signal a bug in LAMMPS.

In our previous work, we showed Lean's broader utility for formalising derivations in science as math proofs [47], digitising key results in absorption theory, thermodynamics, and kinematics. Joseph Tooby-Smith is also developing derivations in the high-energy physics field [48, 55]. These early works showcase Lean's rigour and

versatility for building a library (or libraries) of formally-verified results in diverse areas of science, facilitating rigorous verification of scientific ideas in different disciplines. In the long term, we envision Lean being used to formalise not just interaction potentials or particle dynamics, but the very foundations of statistical mechanics itself. The mathematics of ensemble theory, such as definitions of the microcanonical (NVE) and canonical (NVT) ensembles and proofs about their properties, could be stated precisely, and then directly linked to formally verified simulation code. For instance, we imagine proving that a molecular dynamics simulation routine satisfies the conservation laws of the NVE ensemble, in the limit of infinitesimal time steps, and verifying the detailed balance condition of Monte Carlo moves.

LeanLJ demonstrates how *executable* scientific computing software can be tied to such proofs, using polymorphic functions. We believe this approach is quite general for reasoning about idealised Real-valued functions in scientific *theories*, while linking these to floating-point executions in scientific computing software. Certigrad's [41] approach is also worth considering; this verifies the high-level mathematics in Lean, and then links high-level functions to unverified, but efficient, linear algebra libraries written in C. Compared to our approach, Certigrad's 'bridge' between verified math and executable math consequently happens at a higher level; our polymorphic functions build this bridge at the level of individual math operators (e.g. addition, division) and constants (e.g. π).

Scientific computing benchmarks are typically based on human oversight and software best practices [36]; formal Lean verification offers an even more rigorous alternative, allowing rigorous mathematical proofs that the implemented software is correct. This shift from empirical validation to formal proof introduces a new level of confidence in molecular simulations, setting the stage for more reliable and mathematically sound scientific computing.

8. Conclusion

This work demonstrates how formal methods can complement molecular simulations by providing proven guarantees about the properties of the molecular simulation system. While still in its early stages, this approach opens a path toward mathematically-verified simulations.

Notes

1. This overview is inspired by the presentation in [48].
2. Lean can handle particularly rich mathematics through its use of *dependent types* – types that depend on a value. `Vector` is an example of this – it is a subtype of `List` that depends on a value, the length of the list, which in our case, is 3. This is one way in which Lean avoids runtime errors; before the code compiles, Lean can ensure that a function taking a vector of length `N` will always receive a vector of length `N`.

Acknowledgments

The authors thank the members of the Lean Zulip for helpful discussions, especially Tomáš Skřivan for his insights into polymorphic functions.

Data availability statement

All code, proofs, and benchmark files are on the ATOMS Lab Github.

Disclosure statement

No potential conflict of interest was reported by the author(s).

Funding

This material is based on work supported by the National Science Foundation (NSF) CAREER Award #2236769.

ORCID

Ejike D. Ugwuanyi http://orcid.org/0009-0006-4335-5428
Colin T. Jones http://orcid.org/0009-0007-9013-8036
John Velkey http://orcid.org/0000-0001-5156-7451
Tyler R. Josephson http://orcid.org/0000-0002-0100-0227

References

[1] H.A.L. Filipe and L.M.S. Loura, Molecules **27** (7), 2105 (2022). doi:10.3390/molecules27072105
[2] L. Sun, L. Yang, Y.-D. Zhang, Q. Shi, R.-F. Lu and W.-Q. Deng, J. Comput. Chem. **38** (23), 1991–1999 (2017). doi:10.1002/jcc.24832
[3] L. Sun and W. Deng, Acta Chim. Sinica **73** (6), 579 (2015). doi:10.6023/A15030192
[4] A.E. Mark and W.F. Van Gunsteren, J. Mol. Biol. **240** (2), 167–176 (1994). doi:10.1006/jmbi.1994.1430

[5] R.S. Alvim, Rev. Virtual Quím. **15** (2), 262–282 (2023). doi:10.21577/1984-6835.20220124

[6] A.R. Leach, *Molecular Modelling: Principles and Applications*, 2nd ed. (Prentice Hall, Harlow, 2001).

[7] M.P. Allen and D.J. Tildesley, *Computer Simulation of Liquids*, 2nd ed. (Oxford University Press, Oxford, 2017).

[8] L. Verlet, Phys. Rev. **159** (1), 98–103 (1967). doi:10.1103/PhysRev.159.98

[9] J.T. Horton, S. Boothroyd, P.K. Behara, D.L. Mobley and D.J. Cole, Digit. Discov. **2** (4), 1178–1187 (2023). doi:10.1039/D3DD00070B

[10] J. Zhang, *A Brief Review on Results and Computational Algorithms for Minimizing the Lennard-Jones Potential*, December 2010. arXiv:1101.0039 [physics]. doi:10.48550/arXiv.1101.0039

[11] J. Wang and S. Cheng, *Integrated Lennard-Jones Potential between a Sphere and a Thin Rod*, May 2024. arXiv:2405.03944 [cond-mat]. doi:10.48550/arXiv.2405.03944

[12] P. Schwerdtfeger, A. Burrows and O.R. Smits, J. Phys. Chem. A **125** (14), 3037–3057 (2021). doi:10.1021/acs.jpca.1c00012

[13] X. Wang, S. Ramírez-Hinestrosa, J. Dobnikar and D. Frenkel, Phys. Chem. Chem. Phys. **22** (19), 10624–10633 (2020). doi:10.1039/C9CP05445F

[14] A. Rahman, Phys. Rev. **136** (2A), A405–A411 (1964). doi:10.1103/PhysRev.136.A405

[15] C. Sun, Y. Zhang, C. Hou and W. Ge, Phys. Scr. **98** (1), 015702 (2023). doi:10.1088/1402-4896/aca443

[16] J.J. Nicolas, K.E. Gubbins, W.B. Streett and D.J. Tildesley, Mol. Phys. **37** (5), 1429–1454 (1979). doi:10.1080/00268977900101051

[17] W.R. Fawcett, *Liquids, Solutions, and Interfaces* (Oxford University Press, Oxford, 2004). doi:10.1093/oso/9780195094329.003.0007

[18] H.L. Wen, E. Allahyarov, C.N. Likos, R. Blaak, J. Dzubiella, A. Jusufi, N. Hoffmann and H.M. Harreis, J. Phys. A: Math. Gen. **36** (22), 5827–5834 (2003). doi:10.1088/0305-4470/36/22/301

[19] D.S. Dean, J. Dobnikar, A. Naji and R. Podgornik, editors, *Electrostatics of Soft and Disordered Matter* (Jenny Stanford Publishing, Singapore, 2014). doi:10.1201/b15597

[20] J.Z. Yang and X. Li, Phys. Rev. B **73** (22), 224111 (2006). doi:10.1103/PhysRevB.73.224111

[21] L. Mizzi, D. Attard, R. Gatt, K.K. Dudek, B. Ellul and J.N. Grima, Eng. Comput. **37** (3), 1765–1779 (2021). doi:10.1007/s00366-019-00910-1

[22] E. Palos, E.F. Bull-Vulpe, X. Zhu, H. Agnew, S. Gupta, S. Saha and F. Paesani, *Current Status of the MB-pol Data-Driven Many-Body Potential for Predictive Simulations of Water Across Different Phases*, September 2024. doi:10.26434/chemrxiv-2024-9r9gn-v2

[23] J.W. Lawson and J.B. Haskins, ECS Meeting Abstracts **MA2017-01** (22), 1147–1147 (2017). doi:10.1149/MA2017-01/22/1147

[24] J.P. Andrews and E. Blaisten-Barojas, *Workflow for investigating thermodynamic, structural and energy properties of condensed polymer systems from Molecular Dynamics*, version number: 1 (2022). doi:10.48550/ARXIV.2203.02819

[25] W.W. Wood and F.R. Parker, J. Chem. Phys. **27** (3), 720–733 (1957). doi:10.1063/1.1743822

[26] S. Plimpton, J. Comput. Phys. **117** (1), 1–19 (1995). doi:10.1006/jcph.1995.1039

[27] M.J. Abraham, T. Murtola, R. Schulz, S. Páll, J.C. Smith, B. Hess and E. Lindahl, SoftwareX**1–2**, 19–25 (2015). doi:10.1016/j.softx.2015.06.001

[28] D. Van Der Spoel, E. Lindahl, B. Hess, G. Groenhof, A.E. Mark and H.J.C. Berendsen, J. Comput. Chem. **26** (16), 1701–1718 (2005). doi:10.1002/jcc.20291

[29] M.W. Thompson, J.B. Gilmer, R.A. Matsumoto, C.D. Quach, P. Shamaprasad, A.H. Yang, C.R. Iacovella, C. McCabe and P.T. Cummings, Mol. Phys. **118** (9–10), e1742938 (2020). doi:10.1080/00268976.2020.1742938

[30] A. Nicholls, D.L. Mobley, J. Peter Guthrie, J.D. Chodera, C.I. Bayly, M.D. Cooper and V.S. Pande, J. Med. Chem. **51** (4), 769–779 (2008). doi:10.1021/jm070549+

[31] D.G. Friend, D.J. Frurip, J.W. Magee and J.D. Olson, Fluid Phase Equilib. **217** (1), 11–15 (2004). doi:10.1016/S0378-3812(03)00357-1

[32] F. Case, A. Chaka, D.G. Friend, D. Frurip, J. Golab, R. Johnson, J. Moore, R.D. Mountain, J. Olson, M. Schiller and J. Storer, Fluid Phase Equilib. **217** (1), 1–10 (2004). doi:10.1016/S0378-3812(03)00208-5

[33] N.C. Craven, R. Singh, C.D. Quach, J.B. Gilmer, B. Crawford, E. Marin-Rimoldi, R. Smith, R. DeFever, M.S. Dyukov, J.W. Fothergill, C. Jones, T.C. Moore, B.L. Butler, J.A. Anderson, C.R. Iacovella, E. Jankowski, E.J. Maginn, J.J. Potoff, S.C. Glotzer, P.T. Cummings, C. McCabe and J. Ilja Siepmann, J. Chem. Eng. Data **70** (6), 2178–2199 (2025). doi:10.1021/acs.jced.5c00010

[34] B. Crawford, U. Timalsina, C.D. Quach, N.C. Craven, J.B. Gilmer, C. McCabe, P.T. Cummings and J.J. Potoff, J. Chem. Inf. Model. **63** (4), 1218–1228 (2023). doi:10.1021/acs.jcim.2c01498

[35] J.A. Nash, M. Mostafanejad, T. Daniel Crawford and A.R. McDonald, Comput. Sci. Eng. **24** (3), 72–76 (2022). doi:10.1109/MCSE.2022.3165607

[36] M.W. Thompson, J.B. Gilmer, R.A. Matsumoto, C.D. Quach, P. Shamaprasad, A.H. Yang, C.R. Iacovella, C. McCabe and P.T. Cummings, Mol. Phys. **118** (9–10), e1742938 (2020). doi:10.1080/00268976.2020.1742938

[37] V.K. Shen, D.W. Siderius, W.P. Krekelberg and H.W. Hatch, editors, *NIST Standard Reference Simulation Website*. Number 173 in NIST Standard Reference Database (National Institute of Standards and Technology, Gaithersburg, MD, 2017). doi:10.18434/T4M88Q

[38] P.T. Merz and M.R. Shirts, PLoS. ONE **13** (9), e0202764 (2018). doi:10.1371/journal.pone.0202764

[39] D. Price, IEEE Micro **15** (2), 86–88 (1995). doi:10.1109/40.372360

[40] R. Kaivola and N. Narasimhan, in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '02, (IEEE Computer Society, USA, 2002), p. 20.

[41] D. Selsam, P. Liang and D.L. Dill, in *Proceedings of the 34th International Conference on Machine Learning*, Sydney, Australia (PMLR, 2017), pp. 3047–3056. https://proceedings.mlr.press/v70/selsam17a.html.

[42] J. Harrison, in *Formal Methods for Hardware Verification*, edited by M. Bernardo and A. Cimatti (Springer Berlin Heidelberg, Berlin, Heidelberg, 2006), Vol. 3965, pp. 211–242. Lecture Notes in Computer Science. doi:10.1007/11757283_8

[43] A.V. Tran and Y. Wang, Comput. Mater. Sci. **127**, 141–160 (2017). doi:10.1016/j.commatsci.2016.10.021
[44] G. Irving, *Conservative Interval Arithmetic in Lean*, March 2025. original-date: 2024-05-26T10:00:08Z. https://github.com/girving/interval.
[45] L. de Moura and S. Ullrich, in *Automated Deduction – CADE 28* (Springer International Publishing, Cham, 2021), Vol. 12699, pp. 625–635.doi:10.1007/978-3-030-79876-5_37
[46] D.T. Christiansen, *Functional Programming in Lean*. Leanprover Community, 2023. https://leanprover.github.io/functional_programming_in_lean/.
[47] M.P. Bobbin, S. Sharlin, P. Feyzishendi, A.H. Dang, C.M. Wraback and T.R. Josephson, Digit. Discov.**3** (2), 264–280 (2024). doi:10.1039/D3DD00077J
[48] J. Tooby-Smith, *HepLean: Digitalising High Energy Physics*, May 2024. arXiv:2405.08863 [hep-ph]. doi:10.48550/arXiv.2405.08863
[49] T. Skřivan, lecopivo/SciLean, April 2025. original-date: 2021-09-27T21:50:10Z. https://github.com/lecopivo/SciLean.
[50] H. Macbeth, *The Mechanics of Proof* , 2022. https://hrmacbeth.github.io/math2001/index.html.
[51] D. Frenkel and B. Smit, *Understanding Molecular Simulation: From Algorithms to Applications*, 2nd ed. Number 1 in Computational Science Series (Academic Press, San Diego, 2002).
[52] L. de Moura, S. Kong, J. Avigad, F. van Doorn and J. von Raumer, in *Automated Deduction – CADE-25*, edited by A.P. Felty and A. Middeldorp (Springer International Publishing, Cham, 2015). pp. 378–388. doi:10.1007/978-3-319-21401-6_26
[53] A. Meiburg and J. Commelin, *ComputableReal: A Lean Library for Computable Real Numbers*, 2025. https://github.com/Timeroot/ComputableReal.
[54] A.Z. Summers, J.B. Gilmer, C.R. Iacovella, P.T. Cummings and C. MCabe, J. Chem. Theory Comput.**16** (3), 1779–1793 (2020). doi:10.1021/acs.jctc.9b01183
[55] J. Tooby-Smith, *Formalization of Physics Index Notation in Lean 4*, November 2024. arXiv:2411.07667 [cs]. doi:10.48550/arXiv.2411.07667