Supporting Information: d-SEAMS: Deferred Structural Elucidation Analysis for Molecular Simulations

Rohit Goswami,[†,†,¶] Amrita Goswami,[‡,¶] and Jayant K. Singh[*,‡]

†*Currently at the Department of Chemistry, Indian Institute of Technology Kanpur, Kanpur, 208016, Uttar Pradesh, India*

‡*Department of Chemical Engineering, Indian Institute of Technology Kanpur, Kanpur, 208016, Uttar Pradesh, India*

¶*Contributed equally to this work*

E-mail: jayantks@iitk.ac.in

Phone: 0512-259 6141. Fax: 0512-259 0104

Design

The d-SEAMS framework is designed to be accessible to the end-user, while offering a powerful system of building blocks and generics for extensions. The engine itself is written in `C++` and is compiled to a binary. This binary accepts `Lua` input scripts to expose the functionality of the software such that the underlying data-structures and computations are abstracted away from the user. To facilitate reproduction of results and to prevent users from accessing conflicting or unphysical functional manipulations of the input data, `YAML` options mask certain functions from being exposed. The `YAML` workflows are completely reproducible in `Lua` scripts, but provide a way to share methodologies and also reduces the cognitive load of going through the complete API documentation.

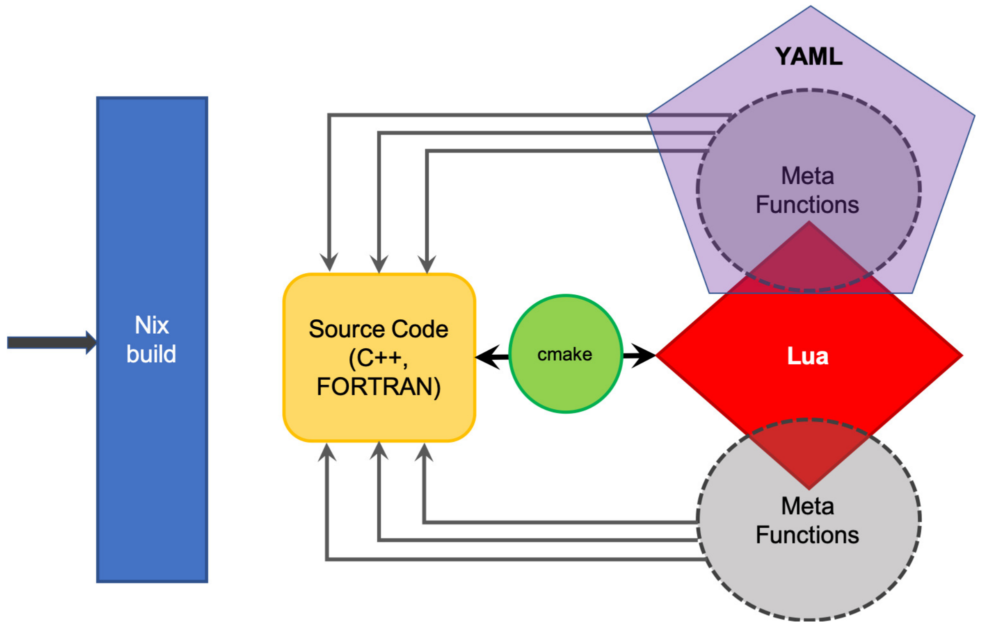


Figure S1: Work-flow of d-SEAMS. `Nix` uses cryptographic hashes to ensure reproducible builds over all systems. `Cmake` compiles and builds the source code, using the dependencies managed by `nix`. The `Lua` script provides an interface to the back-end functions. Combinations of these `C++` functions can be called the 'meta' functions. The `YAML` interface exposes only relevant back-end and `Lua` functions, corresponding to the user-determined pre-determined work-flow.

Figure S1 is a schematic of the overall design architecture of d-SEAMS. We have used `CMake` to compile and build the source code, with the dependencies managed by `nix`. Although the use of `nix` is recommended, users can directly manage the third-party dependencies themselves and build from source using `CMake`. The `Lua` scripting interface exposes `C`-like functions to create custom work-flows. The `YAML` configuration file provides options for pre-determined work-flows. Users with different requirements and experience can interact with `d-SEAMS`. The three main components of the code architecture are enumerated below:

- `YAML` **configuration file:** This contains options for pre-determined work-flows. For example, a user who enters the option for the confined quasi-one-dimensional ice determination will only be exposed to the relevant functions for INT prism determination. Multiple workflows can be selected at the same time.

- `Lua` **Interface:** C++ functions are registered as `Lua` functions, which are called from a `Lua` script. `Lua` is a `C`-like scripting language, enabling users to call the `Lua` functions without needing to learn a software-specific scripting convention. The advantage of using `Lua` over directly calling `C++` functions is that the users need not be concerned with pointers and clean-up of the `C++` structures. The `Lua` language also has a rich set of cross platform extensions for file handling, and is also supported by major editors for syntax highlighting.

- `C++` **Back-end:** The back-end is written in modern `C++`, employing common data structures, used uniformly throughout the code. Users can easily extend and write their own `C++` header files, and the documentation covers manipulating the build system to accept both user-defined and external headers. Registering custom `C++` functions as `Lua` functions, to be subsequently called in `Lua` scripts, is also documented. `GDB`[S1] can be used for code debugging, since the back-end is in `C++`.

From a user perspective, we have designed the `Lua` functions to mimic the mindset of a computational chemist, without burdening them with the software implementation. We have also ensured reproducibility, both as an aid to the science[S2] intended and also to allow for bugs to be dealt with more efficiently. This reproducibility is ensured during build, compile, and linking stages, by leveraging the functional, immutable binaries produced by nix.[S3] The dependencies are handled reproducibly, though for ease of extension by the wider community, most of the build system is in `CMake`. We use `nix` to ensure that the dependencies of the binary are fully reproducible, as a consequence of traversing the build graph defined by the nix-derivation. The binary itself has a server-client architecture, to ensure that the user can transparently interact with the code without needing a background in functional programming. Since the back-end server is written entirely in modern `C++`, the GDB debugger is usable throughout. The server-client nature of the system, though currently a bottleneck in terms of parallelism, allows for a single compiled binary to be used for the execution of multiple different `Lua` input scripts, with each script spawning a separate process.

Nix

The nix derivation provides a deterministic package-level lock on all dependencies and is written in `nix`, a lazy, dynamically typed, purely functional language.[S4] `Nix` manages the installation of libraries and extra tooling required. Through the `nix-shell`, we provide a reproducible environment for developers to ensure that their contributions are in keeping with the libraries and third-party tools used by the upstream developers. Given that `nix` also includes rollbacks, it allows for all dependencies to be reverted to arbitrary instances in the project history. However, if the user does not use `nix`, this would necessarily have to be handled manually.

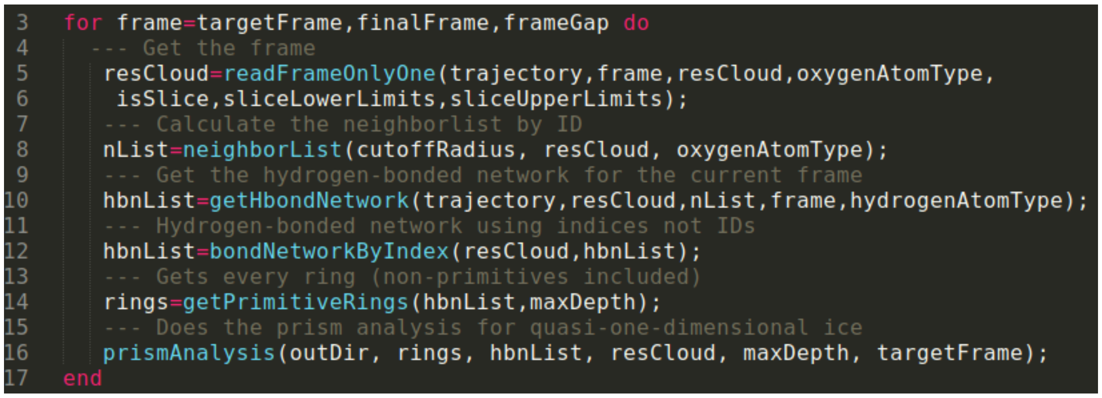
Lua

Existing molecular dynamics packages suffer from not having design parameters built-in, and with time, this has led to unique and non-standard syntax being used, as seen in the input scripts of LAMMPS[S5] and GROMACS,[S6] amongst others. Popular text editors do not offer syntax highlighting for these custom non-standard and software-specific syntaxes. For such software-specific syntaxes, the code is unusable without learning from the documentation. We also note that the version dependence of each internal segment of large and complex software systems can become intractable without continuous development, and as a result, these may spawn multiple language-specific errors. Thus, they tend to work best on the distribution on which the creators have worked.

We have opted to use `Lua` as the scripting interface, which has `C`-like functions. It is widely supported in terms of syntax highlighting, and can be interfaced with `C++` code. Furthermore, the error handling is such that it is amenable for arbitrarily complex GDB debugger workflows,[S1] and the rich standard library of `Lua`, along with user extensions, have no clashes. `Lua` is also user-friendly due to its `C`-like syntax. The rich table and object handling makes writing out image data convenient.

We have used the excellent FOSS (free and open source) `sol3` library (https://github.com/ThePhD/sol2
for generating C++ library bindings to Lua. It is a header-only library so in theory it is a lightweight dependency for those compiling without `nix`. The canonical bindings from `lua` are to `ANSI-C`. Leveraging `sol3` minimizes the creation of boiler-plate code and it also provides native support for handling class objects, unlike the standard bindings.

`Lua` has been the darling of the gaming development community, and has proven its worth in many related domains such as image handling. Apart from the user-friendly helper functions, our design has the `Lua` interface, which offers every core function to the user. This permits arbitarily complicated workflows to be used without re-compiliation, which is a boon for HPC cluster usage. We recommend strongly in the docs, that foreign code, once interfaced to the `C++` engine, should be bound in `Lua` for the end-users as well.

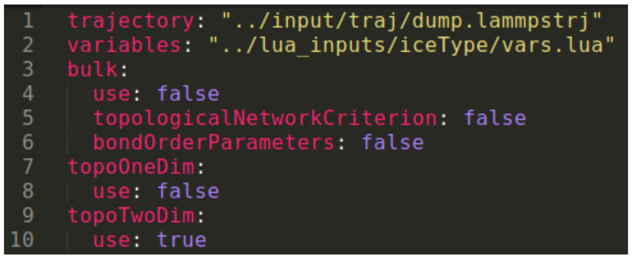


```lua
for frame=targetFrame,finalFrame,frameGap do
  --- Get the frame
   resCloud=readFrameOnlyOne(trajectory,frame,resCloud,oxygenAtomType,
    isSlice,sliceLowerLimits,sliceUpperLimits);
   --- Calculate the neighborlist by ID
   nList=neighborList(cutoffRadius, resCloud, oxygenAtomType);
   --- Get the hydrogen-bonded network for the current frame
   hbnList=getHbondNetwork(trajectory,resCloud,nList,frame,hydrogenAtomType);
   --- Hydrogen-bonded network using indices not IDs
   hbnList=bondNetworkByIndex(resCloud,hbnList);
   --- Gets every ring (non-primitives included)
   rings=getPrimitiveRings(hbnList,maxDepth);
   --- Does the prism analysis for quasi-one-dimensional ice
   prismAnalysis(outDir, rings, hbnList, resCloud, maxDepth, targetFrame);
end
```

Figure S2: The `Lua` input script, where the user is able to call any of the functions not voided by the options in the `YAML` file.

Figure S2 shows a typical `Lua` input script, which calls functions exposed by the current `YAML` file work-flow.

YAML



```yaml
trajectory: "../input/traj/dump.lammpstrj"
variables: "../lua_inputs/iceType/vars.lua"
bulk:
  use: false
  topologicalNetworkCriterion: false
  bondOrderParameters: false
topoOneDim:
  use: false
topoTwoDim:
  use: true
```

Figure S3: The `YAML` file, where boolean values are set to restrict functions exposed to the `Lua` scripting engine.

To improve usability and reduce the time required reading the API documentation, we have split the usage into a unique `YAML-Lua` design. The `Lua` interface is for power users, however, to reduce mistakes, options set in the `YAML` files will deactivate certain functions, in order to prevent incorrect manipulations of the internal data-structures. The `YAML` interface diverts the flow of functionality and code to different paths, and thus different algorithms. This also prevents name-clashes of similar functions for mutually exclusive work-flows. For example, an input system can either be a bulk system, a quasi-one-dimensional system or a quasi-two-dimensional system. The `YAML` file offers truthy options, an example of which is shown in Figure S3, and subsequently masks functions not applicable for the given system type.

In summary, the code architecture can be briefly described as follows:

- The d-SEAMS binary is built using `CMake`. To prevent dependency-clashes, library version-mismatch errors, and for ease of installation, users are recommended to install and build d-SEAMS using `nix`. A complete list of all the third-party libraries required is also provided (https://docs.dseams.info).

- The back-end is written in modern `C++17`. Functions for algorithms of separate work-flows (quasi-one- or quasi-two- dimensional, or bulk systems) are differentiated into distinct namespaces. The developer manual (https://docs.dseams.info/) provides API-level documentation of the `C++` back-end.

- User-facing `Lua` functions are registered on the `C++` side. The `sol3` library is used to generate `C++` library bindings to the `Lua` functions. The user-functions are documented in the d-SEAMS wiki page (https://wiki.dseams.info).
- The `YAML` file provides options for deciding the work-flow, which expose the functions valid for the particular work-flow selected. For example, the bulk radial distribution function would yield incorrect results for a monolayer system, for which the in-plane radial distribution function is appropriate. By setting the option for a quasi-two-dimensional system in the `YAML` file, the user is only exposed to the function for the in-plane radial distribution. The yaml-cpp library (https://github.com/jbeder/yaml-cpp) is used to parse the `YAML` options.
- The developer manual provides instructions for users to extend d-SEAMS, add to the `C++` back-end, and write their own `Lua` bindings.

Lua Input Files

In this section, we have provided the `Lua` input scripts used for generating the results described in the manuscript. For more complete descriptions and further guidance on how to use the scripts, we invite the reader to visit the relevant examples documented in the d-SEAMS wiki page (https://wiki.dseams.info).

Heterogenous Nucleation on an Ice-Promoting Surface

We have used the topological network criterion[S7] for identifying Double-Diamond Cages (DDCs) and Hexagonal Cages (HCs) and analyzing the stacking of ice on a smooth silver-iodide surface. This is similar to the example provided on the d-SEAMS wiki for homogenous bulk nucleation using the topological network criterion (https://wiki.dseams.info/examples/bulktopological

The *vars.lua* file contains user-defined variables required for the analysis.

vars.lua

```lua
cutoffRadius = 3.5; -- This is for H2O
oxygenAtomType = 4; -- This is assigned by LAMMPS
hydrogenAtomType = 1; -- Hydrogen atom type assigned (not used here)
targetFrame=1; -- The first frame (inclusive)
finalFrame=2392; -- This is inclusive
frameGap=1; -- The gap between frames
maxDepth = 6; -- The maximum depth upto which rings will be searched.
-- Slice Information
isSlice = false; -- This is true if the analysis is to be done only for a
    volume slice
sliceLowerLimits = {0,0,0}; -- Lower limit of the slice (for box dim, keep
    the values the same as 0)
sliceUpperLimits = {0,0,0}; -- Upper limit of the slice

-- Paths for the output directories and lua scipt
outDir="runOne/"; -- The subdirectory used;
functionScript="lua_inputs/iceType/functions.lua" -- This is relative to the
    binary location

-- Variable for the topological network criterion
printCages = false; -- Prints out every cage for every frame if true
```

The *functions.lua* file interacts with the `C++` backend using the `C`-like `lua` functions registered in the src/main.cpp. Users interested in extending d-SEAMS and adding new `Lua` functions are advised to refer to the documentation.

functions.lua

```lua
for frame=targetFrame,finalFrame,frameGap do
    resCloud=readFrameOnlyOne(trajectory,frame,resCloud,oxygenAtomType,isSlice,
     sliceLowerLimits,sliceUpperLimits) -- Get the frame
    nList=neighborList(cutoffRadius, resCloud, oxygenAtomType); -- Calculate
    the neighborlist by ID
    iceNeighbourList = bondNetworkByIndex(resCloud, nList); -- Calculate the
    neighborlist by index
    --
    -- Start of analysis using rings (by index from here onwards.)
    rings=getPrimitiveRings(iceNeighbourList,maxDepth); -- Gets every ring (
    non-primitives included)
    bulkTopologicalNetworkCriterion(outDir, rings, iceNeighbourList,
    clusterCloud, targetFrame, printCages); -- Finds DDCs and HCs
end
```

Homogenous Nucleation: Growth of the Largest Ice Cluster

Here, we have used a topological network criterion[S7] to analyze a successful nucleation event for a system of 4096 monoatomic (mW) water molecules.[S8] This is, in essence, the same as the example provided on the d-SEAMS wiki for homogenous bulk nucleation using the topological network criterion (https://wiki.dseams.info/examples/bulktopologicalcriterion). The trajectory provided in the example in the docs is shorter than the production run trajectory analyzed in the manuscript.

The *vars.lua* file contains user-defined variables required for the analysis.

vars.lua

```
cutoffRadius = 3.5; -- This is for H2O
oxygenAtomType = 2; -- This is assigned by LAMMPS
hydrogenAtomType = 1; -- Hydrogen atom type assigned (not used here)
targetFrame=1; -- The first frame (inclusive)
finalFrame=4000; -- This is inclusive
frameGap=1; -- The gap between frames
maxDepth = 6; -- The maximum depth upto which rings will be searched.
-- Slice Information
```

```
isSlice = false; -- This is true if the analysis is to be done only for a
    volume slice
sliceLowerLimits = {0,0,0}; -- Lower limit of the slice (for box dim, keep
    the values the same as 0)
sliceUpperLimits = {0,0,0}; -- Upper limit of the slice

-- Paths for the output directories and lua scipt
outDir="runOne/"; -- The subdirectory used;
functionScript="lua_inputs/iceType/functions.lua" -- This is relative to the
    binary location

-- Variable for the topological network criterion
printCages = false; -- Prints out every cage for every frame if true
```

The *functions.lua* has been provided below.

functions.lua

```
for frame=targetFrame,finalFrame,frameGap do
    resCloud=readFrameOnlyOne(trajectory,frame,resCloud,oxygenAtomType,isSlice,
    sliceLowerLimits,sliceUpperLimits) -- Get the frame
    nList=neighborList(cutoffRadius, resCloud, oxygenAtomType); -- Calculate
    the neighborlist by ID
    iceNeighbourList = bondNetworkByIndex(resCloud, nList); -- Calculate the
    neighborlist by index
    --
    -- Start of analysis using rings (by index from here onwards.)
    rings=getPrimitiveRings(iceNeighbourList,maxDepth); -- Gets every ring (
    non-primitives included)
    bulkTopologicalNetworkCriterion(outDir, rings, iceNeighbourList,
    clusterCloud, targetFrame, printCages); -- Finds DDCs and HCs
end
```

Quasi-Two-Dimensional Systems

We have used d-SEAMS to analyze flat Monolayer Square Ice (fMSI) by:

1. Creating LAMMPS data files, containing the connectivity information and particle classification types labelled by d-SEAMS using topological network criteria, specialized for quasi-two-dimensional ice.[S9] These data files are compatible with OVITO.[S10]

2. ASCII output files are written out by d-SEAMS, which contain the calculated coverage area percentage metric for each frame.

3. An ASCII output file is produced for the in-plane radial distribution function.

The system analyzed here is similar to the example provided in the wiki for monolayer ice classification (https://wiki.dseams.info/examples/monolayer). The *vars.lua* for the user-defined variables is provided below. Users should note that all units are the same as those in the input trajectory files. Here, all distance units are in Angstroms.

vars.lua

```lua
cutoffRadius = 3.5; -- This is for H2O
oxygenAtomType = 2; -- This is assigned by LAMMPS
hydrogenAtomType = 1; -- Hydrogen atom type assigned
targetFrame=1; -- The first frame (inclusive)
finalFrame=1000; -- This is inclusive
frameGap=1; -- The gap between frames
maxDepth = 4; -- The maximum depth upto which rings will be searched.
-- Slice Information
isSlice = true; -- This is true if the analysis is to be done only for a
    volume slice
```

```
sliceLowerLimits = {0,0,0}; -- Lower limit of the slice (for box dim, keep the values the same as 0)
sliceUpperLimits = {50,0,0}; -- Upper limit of the slice

-- Paths for the output directories and lua scipt
outDir="runOne/"; -- The subdirectory used;
functionScript="lua_inputs/iceType/functions.lua" -- This is relative to the binary location

-- Variables for the monolayer only:
confiningSheetArea = 50*50;

-- Variables for the RDF only
rdf = true; -- This should only be set to true if you want to calculate the RDF
rdfCutoff = 12; -- This should be less than half the box length
binwidth = 0.05; -- This is the binwidth or delta_r
```

The *functions.lua* input script containing the `Lua` functions is reproduced below. The wiki provides more information on using the functions for the radial distribution function (https://wiki.dseams.info/examples/rdf2d).

functions.lua

```
for frame=targetFrame,finalFrame,frameGap do
    resCloud=readFrameOnlyOne(trajectory,frame,resCloud,oxygenAtomType,isSlice,
     sliceLowerLimits,sliceUpperLimits) -- Get the frame
    nList=neighborList(cutoffRadius, resCloud, oxygenAtomType); -- Calculate
     the neighborlist by ID
    hbnList=getHbondNetwork(trajectory,resCloud,nList,frame,hydrogenAtomType)
     -- Get the hydrogen-bonded network for the current frame
    hbnList=bondNetworkByIndex(resCloud,hbnList) -- Hydrogen-bonded network
     using indices not IDs
    rings=getPrimitiveRings(hbnList,maxDepth); -- Gets every ring (non-
     primitives included)
    ringAnalysis(outDir, rings, hbnList, resCloud, maxDepth, confiningSheetArea
     , targetFrame); -- Does the ring analysis for quasi-two-dimensional ice
    -- RDF analysis
    calcRDF(outDir,rdf,resCloud,rdfCutoff,binwidth,targetFrame,finalFrame);
    --
end
```

Quasi-One-Dimensional Systems

In the manuscript, we have tracked the liquid-to-solid phase change of quasi-one-dimensional water, constrained by a smooth $(13,0)$ single walled nanotube (SWNT), using the $height_n\%$ order parameter. The example (https://wiki.dseams.info/examples/icenanotube) provides figshare links to input trajectory files and descriptions of a similar tetragonal ice nanotube.

vars.lua

```
cutoffRadius = 3.5; -- This is for H2O
oxygenAtomType = 2; -- This is assigned by LAMMPS
hydrogenAtomType = 1; -- Hydrogen atom type assigned
targetFrame=1; -- The first frame (inclusive)
finalFrame=1000; -- This is inclusive
frameGap=1; -- The gap between frames
maxDepth = 6; -- The maximum depth upto which rings will be searched.
-- Slice Information
isSlice = false; -- This is true if the analysis is to be done only for a
     volume slice
sliceLowerLimits = {0,0,0}; -- Lower limit of the slice (for box dim, keep
     the values the same as 0)
sliceUpperLimits = {0,0,0}; -- Upper limit of the slice

```

```
-- Paths for the output directories and lua scipt
outDir="runOne/"; -- The subdirectory used;
functionScript="lua_inputs/iceType/functions.lua" -- This is relative to the
    binary location
```

The *functions.lua* input script is also reproduced below.

functions.lua

```
for frame=targetFrame,finalFrame,frameGap do
  resCloud=readFrameOnlyOne(trajectory,frame,resCloud,oxygenAtomType,isSlice,
    sliceLowerLimits,sliceUpperLimits) -- Get the frame
  nList=neighborList(cutoffRadius, resCloud, oxygenAtomType); -- Calculate
    the neighborlist by ID
  hbnList=getHbondNetwork(trajectory,resCloud,nList,frame,hydrogenAtomType)
  -- Get the hydrogen-bonded network for the current frame
  hbnList=bondNetworkByIndex(resCloud,hbnList) -- Hydrogen-bonded network
    using indices not IDs
  rings=getPrimitiveRings(hbnList,maxDepth); -- Gets every ring (non-
    primitives included)
  prismAnalysis(outDir, rings, hbnList, resCloud, maxDepth, targetFrame); --
     Does the prism analysis for quasi-one-dimensional ice
end
```

Output Files Produced by d-SEAMS

d-SEAMS is capable of producing two types of output:

1. LAMMPS data files are produced for each frame or configuration read in. These data files contain the positional data of particles classified and labelled by d-SEAMS. The

connectivity information obtained from the application of topological network criteria is written out to the data files in the guise of bonds. Thus, each snapshot can be visualized directly in OVITO[S10] or VMD.[S11] We have used OVITO to generate the images in this work.

2. ASCII files are also written out, containing the order parameters calculated for each frame. The radial distribution function averaged over the input frames can also be optionally written out in the form of an ASCII text file. The columns of data are space-separated, and prefaced by a comment line.

Here, we provide a sample LAMMPS data file produced by d-SEAMS, which can be directly read in and visualized by OVITO. This data file was created by d-SEAMS, from the positional data of a single DDC. Here, the atom type or label 'dummy' refers to unclassified particles, while the atom types 'hc' and 'ddc' refer to particles comprising HCs and DDCs respectively. The atom type 'mixed' refers to atoms in mixed rings, which are shared between DDCs and HCs. In OVITO, the particles are labelled by the labels provided in the LAMMPS data file. The connectivity information is also preserved in the form of bonds of type 1.

```
Written out by D-SEAMS
14 atoms
18 bonds
0 angles
0 dihedrals
0 impropers
4 atom types
1 bond types
0 angle types
0 dihedral types
0 improper types
0 50.967 xlo xhi
0 50.967 ylo yhi
```

```
0 50.967 zlo zhi

Masses

1 15.999400 # dummy
2 15.999400 # hc
3 15.999400 # ddc
4 15.999400 # mixed

Atoms

1 1 3 0 25.4799996 25.4799996 19.1100006
2 2 3 0 25.4799996 25.4799996 25.4799996
3 3 3 0 25.4799996 22.2950001 22.2950001
4 4 3 0 25.4799996 28.6650009 22.2950001
5 5 3 0 22.2950001 22.2950001 25.4799996
6 6 3 0 22.2950001 25.4799996 22.2950001
7 7 3 0 28.6650009 25.4799996 22.2950001
8 8 3 0 23.8880006 20.7029991 23.8880006
9 9 3 0 23.8880006 27.073 23.8880006
10 10 3 0 27.073 27.073 20.7029991
11 11 3 0 20.7029991 23.8880006 23.8880006
12 12 3 0 27.073 23.8880006 23.8880006
13 13 3 0 23.8880006 23.8880006 20.7029991
14 1 3 0 23.8880006 23.8880006 27.073

Bonds

1 1 1 10
2 1 1 13
3 1 2 14
4 1 2 9
5 1 2 12
```

```
6 1 3 12
7 1 3 8
8 1 3 13
9 1 4 9
10 1 4 10
11 1 5 14
12 1 5 11
13 1 5 8
14 1 6 9
15 1 6 11
16 1 6 13
17 1 7 12
18 1 7 10
```

References

(S1) Stallman, R.; Pesch, R.; Shebs, S., et al. Debugging with GDB. *Free Software Foundation* **2002**, *51*, 02110–1301.

(S2) Mesirov, J. Accessible Reproducible Research. *Science* **2010**, *327*, 415–416.

(S3) Dolstra, E.; de Jonge, M.; Visser, E. Nix: A Safe and Policy-Free System for Software Deployment. Proceedings of the 18th USENIX Conference on System Administration. USA, 2004; p 79–92.

(S4) Dolstra, E.; Löh, A. NixOS. *SIGPLAN Not.* **2008**, *43*, 367.

(S5) Plimpton, S. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *J. Comput. Phys.* **1995**, *117*, 1–19.

(S6) Abraham, M. J.; Murtola, T.; Schulz, R.; Páll, S.; Smith, J. C.; Hess, B.; Lindahl, E. GROMACS: High Performance Molecular Simulations through Multi-Level Parallelism from Laptops to Supercomputers. *SoftwareX* **2015**, *1-2*, 19–25.

(S7) Haji-Akbari, A.; Debenedetti, P. G. Direct Calculation of Ice Homogeneous Nucleation Rate for a Molecular Model of Water. *Proc. Natl. Acad. Sci. U.S.A.* **2015**, *112*, 10582–10588.

(S8) Molinero, V.; Moore, E. B. Water Modeled As an Intermediate Element between Carbon and Silicon†. *J. Phys. Chem. B* **2009**, *113*, 4008–4016.

(S9) Goswami, A.; Singh, J. K. A General Topological Network Criterion for Exploring the Structure of Icy Nanoribbons and Monolayers. *Phys. Chem. Chem. Phys.* **2020**, *22*, 3800–3808.

(S10) Stukowski, A. Visualization and Analysis of Atomistic Simulation Data with OVITO–the Open Visualization Tool. *Modell. Simul. Mater. Sci. Eng.* **2009**, *18*, 015012.

(S11) Humphrey, W.; Dalke, A.; Schulten, K. VMD: Visual Molecular Dynamics. *J. Mol. Graphics* **1996**, *14*, 33–38.