CHAPTER

TWO

USE PYTHON WITH LAMMPS

These pages describe various ways that LAMMPS and Python can be used together.

2.1 Overview

The LAMMPS distribution includes a `python` directory with the Python code needed to run LAMMPS from Python. The `python/lammps` package contains *the "lammps" Python module* that wraps the LAMMPS C-library interface. This module makes it is possible to do the following either from a Python script, or interactively from a Python prompt:

- create one or more instances of LAMMPS
- invoke LAMMPS commands or read them from an input script
- run LAMMPS incrementally
- extract LAMMPS results
- and modify internal LAMMPS data structures.

From a Python script you can do this in serial or in parallel. Running Python interactively in parallel does not generally work, unless you have a version of Python that extends Python to enable multiple instances of Python to read what you type.

To do all of this, you must build LAMMPS in *"shared" mode* and make certain that your Python interpreter can find the `lammps` Python package and the LAMMPS shared library file.

The Python wrapper for LAMMPS uses the ctypes package in Python, which auto-generates the interface code needed between Python and a set of C-style library functions. Ctypes has been part of the standard Python distribution since version 2.5. You can check which version of Python you have by simply typing "python" at a shell prompt. Below is an example output for Python version 3.8.5.

```
$ python
Python 3.8.5 (default, Aug 12 2020, 00:00:00)
[GCC 10.2.1 20200723 (Red Hat 10.2.1-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Warning: The options described in this section of the manual for using Python with LAMMPS currently support either Python 2 or 3. Specifically version 2.7 or later and 3.6 or later. Since the Python community no longer maintains Python 2 (see this notice), we recommend use of Python 3 with LAMMPS. While Python 2 code should continue to work, that is not something we can guarantee long-term. If you notice Python code in the LAMMPS distribution that is not compatible with Python 3, please contact the LAMMPS developers or submit and issue on GitHub

LAMMPS can work together with Python in three ways. First, Python can wrap LAMMPS through the its *library interface*, so that a Python script can create one or more instances of LAMMPS and launch one or more simulations. In Python terms, this is referred to as "extending" Python with a LAMMPS module.

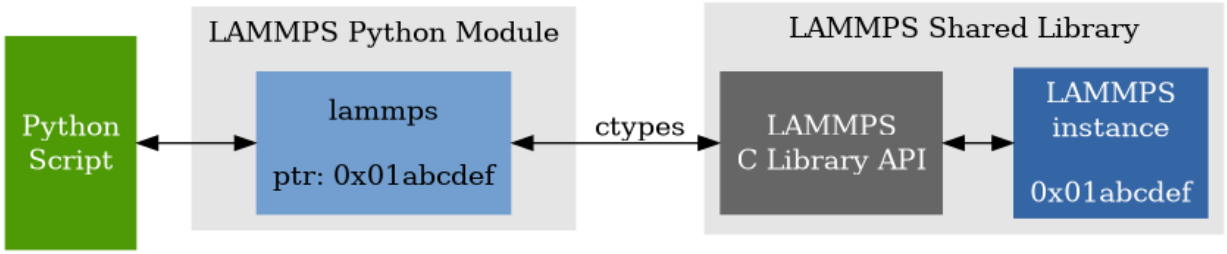


Fig. 1: Launching LAMMPS via Python

Second, the lower-level Python interface in the `lammps Python class` can be used indirectly through the provided `PyLammps` and `IPyLammps` wrapper classes, also written in Python. These wrappers try to simplify the usage of LAMMPS in Python by providing a more object-based interface to common LAMMPS functionality. They also reduce the amount of code necessary to parameterize LAMMPS scripts through Python and make variables and computes directly accessible.

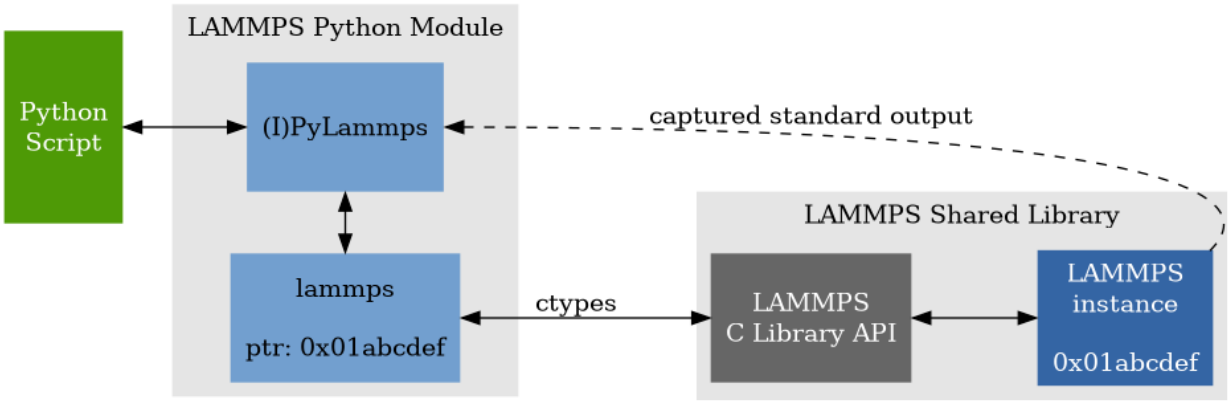


Fig. 2: Using the PyLammps / IPyLammps wrappers

Third, LAMMPS can use the Python interpreter, so that a LAMMPS input script or styles can invoke Python code directly, and pass information back-and-forth between the input script and Python functions you write. This Python code can also call back to LAMMPS to query or change its attributes through the LAMMPS Python module mentioned above. In Python terms, this is called "embedding" Python into LAMMPS. When used in this mode, Python can perform script operations that the simple LAMMPS input script syntax can not.

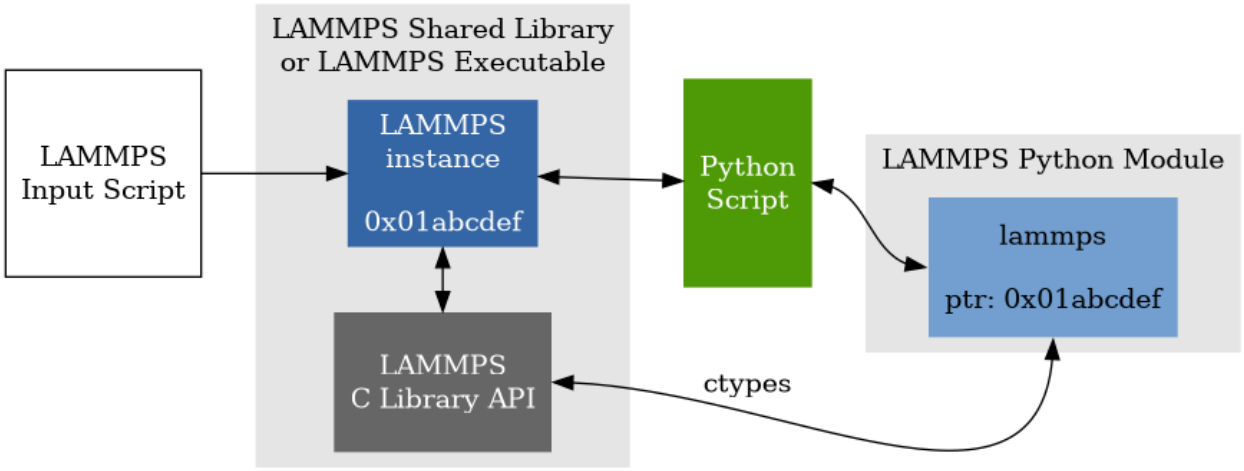


Fig. 3: Calling Python code from LAMMPS

2.2 Installation

The LAMMPS Python module enables calling the *LAMMPS C library API* from Python by dynamically loading functions in the LAMMPS shared library through the Python ctypes module. Because of the dynamic loading, it is required that LAMMPS is compiled in *"shared" mode*.

Two components are necessary for Python to be able to invoke LAMMPS code:

- The LAMMPS Python Package (`lammps`) from the `python` folder
- The LAMMPS Shared Library (`liblammps.so`, `liblammps.dylib` or `liblammps.dll`) from the folder where you compiled LAMMPS.

2.2.1 Installing the LAMMPS Python Module and Shared Library

Making LAMMPS usable within Python and vice versa requires putting the LAMMPS Python package (`lammps`) into a location where the Python interpreter can find it and installing the LAMMPS shared library into a folder that the dynamic loader searches or inside of the installed `lammps` package folder. There are multiple ways to achieve this.

1. Install both components into a Python `site-packages` folder, either system-wide or in the corresponding user-specific folder. This way no additional environment variables need to be set, but the shared library is otherwise not accessible.
2. Do an installation into a virtual environment.
3. Leave the files where they are in the source/development tree and adjust some environment variables.

Python package

Compile LAMMPS with either *CMake* or the *traditional make* procedure in *shared mode*. After compilation has finished, type (in the compilation folder):

```
make install-python
```

This will try to build a so-called (binary) wheel file, a compressed binary python package and then install it with the python package manager 'pip'. Installation will be attempted into a system-wide `site-packages` folder and

if that fails into the corresponding folder in the user's home directory. For a system-wide installation you usually would have to gain superuser privilege first, e.g. though `sudo`

File	Location	Notes
LAMMPS Python package	• `$HOME/.local/lib/pythonX.Y/site-packages/lammps`	`X.Y` depends on the installed Python version
LAMMPS shared library	• `$HOME/.local/lib/pythonX.Y/site-packages/lammps`	`X.Y` depends on the installed Python version

For a system-wide installation those folders would then become.

File	Location	Notes
LAMMPS Python package	• `/usr/lib/pythonX.Y/site-packages/lammps`	`X.Y` depends on the installed Python version
LAMMPS shared library	• `/usr/lib/pythonX.Y/site-packages/lammps`	`X.Y` depends on the installed Python version

No environment variables need to be set for those, as those folders are searched by default by Python or the LAMMPS Python package.

Changed in version 24Mar2022.

Note: If there is an existing installation of the LAMMPS python module, `make install-python` will try to update it. However, that will fail if the older version of the module was installed by LAMMPS versions until 17Feb2022. Those were using the distutils package, which does not create a "manifest" that allows a clean uninstall. The `make install-python` command will always produce a lammps-<version>-<python>-<abi>-<os>-<arch>.whl file (the 'wheel'). And this file can be later installed directly with `python -m pip install <wheel file>.whl` without having to type `make install-python` again and repeating the build step, too.

For the traditional make process you can override the python version to version x.y when calling `make` with `PYTHON=pythonX.Y`. For a CMake based compilation this choice has to be made during the CMake configuration step.

If the default settings of `make install-python` are not what you want, you can invoke `install.py` from the python directory manually as

```
python install.py -p <python package> -l <shared library> -v <version.h file> [-n]
```

- The `-p` flag points to the `lammps` Python package folder to be installed,
- the `-l` flag points to the LAMMPS shared library file to be installed,
- the `-v` flag points to the LAMMPS version header file to extract the version date,
- and the optional `-n` instructs the script to only build a wheel file but not attempt to install it.

Virtual environment

A virtual environment is a minimal Python installation inside of a folder. It allows isolating and customizing a Python environment that is mostly independent from a user or system installation. For the core Python environment, it uses symbolic links to the system installation and thus it can be set up quickly and will not take up much disk space. This gives you the flexibility to install (newer/different) versions of Python packages that would potentially conflict with already installed system packages. It also does not requite any superuser privileges. See PEP 405: Python Virtual Environments for more information.

To create a virtual environment in the folder `$HOME/myenv`, use the venv module as follows.

```
# create virtual environment in folder $HOME/myenv
python3 -m venv $HOME/myenv
```

For Python versions prior 3.3 you can use virtualenv command instead of "python3 -m venv". This step has to be done only once.

To activate the virtual environment type:

```
source $HOME/myenv/bin/activate
```

This has to be done every time you log in or open a new terminal window and after you turn off the virtual environment with the `deactivate` command.

When using CMake to build LAMMPS, you need to set `CMAKE_INSTALL_PREFIX` to the value of the `$VIRTUAL_ENV` environment variable during the configuration step. For the traditional make procedure, no additional steps are needed. After compiling LAMMPS you can do a "Python package only" installation with `make install-python` and the LAMMPS Python package and the shared library file are installed into the following locations:

File	Location	Notes
LAMMPS Python Module	• `$VIRTUAL_ENV/lib/pythonX.Y/site-packages/lammps`	`X.Y` depends on the installed Python version
LAMMPS shared library	• `$VIRTUAL_ENV/lib/pythonX.Y/site-packages/lammps`	`X.Y` depends on the installed Python version

In place usage

You can also *compile LAMMPS* as usual in *"shared" mode* leave the shared library and Python package inside the source/compilation folders. Instead of copying the files where they can be found, you need to set the environment variables `PYTHONPATH` (for the Python package) and `LD_LIBRARY_PATH` (or `DYLD_LIBRARY_PATH` on macOS

For Bourne shells (bash, ksh and similar) the commands are:

```
export PYTHONPATH=${PYTHONPATH}:${HOME}/lammps/python
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:${HOME}/lammps/src
```

For the C-shells like csh or tcsh the commands are:

```
setenv PYTHONPATH ${PYTHONPATH}:${HOME}/lammps/python
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:${HOME}/lammps/src
```

On macOS you may also need to set DYLD_LIBRARY_PATH accordingly. You can make those changes permanent by editing your $HOME/.bashrc or $HOME/.login files, respectively.

Note: The PYTHONPATH needs to point to the parent folder that contains the lammps package!

To verify if LAMMPS can be successfully started from Python, start the Python interpreter, load the lammps Python module and create a LAMMPS instance. This should not generate an error message and produce output similar to the following:

```
$ python
Python 3.8.5 (default, Sep  5 2020, 10:50:12)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import lammps
>>> lmp = lammps.lammps()
LAMMPS (18 Sep 2020)
using 1 OpenMP thread(s) per MPI task
>>>
```

Note: Unless you opted for "In place use", you will have to rerun the installation any time you recompile LAMMPS to ensure the latest Python package and shared library are installed and used.

Note: If you want Python to be able to load different versions of the LAMMPS shared library with different settings, you will need to manually copy the files under different names (e.g. liblammps_mpi.so or liblammps_gpu.so) into the appropriate folder as indicated above. You can then select the desired library through the *name* argument of the LAMMPS object constructor (see *Creating or deleting a LAMMPS object*).

2.2.2 Extending Python to run in parallel

If you wish to run LAMMPS in parallel from Python, you need to extend your Python with an interface to MPI. This also allows you to make MPI calls directly from Python in your script, if you desire.

We have tested this with MPI for Python (aka mpi4py) and you will find installation instruction for it below.

Installation of mpi4py (version 3.0.3 as of Sep 2020) can be done as follows:

- Via pip into a local user folder with:

```
pip install --user mpi4py
```

- Via dnf into a system folder for RedHat/Fedora systems:

```
# for use with OpenMPI
sudo dnf install python3-mpi4py-openmpi
```

(continues on next page)

(continued from previous page)

```
# for use with MPICH
sudo dnf install python3-mpi4py-openmpi
```

- Via `pip` into a virtual environment (see above):

```
$ source $HOME/myenv/activate
(myenv)$ pip install mpi4py
```

- Via `pip` into a system folder (not recommended):

```
sudo pip install mpi4py
```

For more detailed installation instructions and additional options, please see the mpi4py installation page.

To use `mpi4py` and LAMMPS in parallel from Python, you **must** make certain that **both** are using the **same** implementation and version of MPI library. If you only have one MPI library installed on your system this is not an issue, but it can be if you have multiple MPI installations (e.g. on an HPC cluster to be selected through environment modules). Your LAMMPS build is explicit about which MPI it is using, since it is either detected during CMake configuration or in the traditional make build system you specify the details in your low-level `src/MAKE/Makefile.foo` file. The installation process of `mpi4py` uses the `mpicc` command to find information about the MPI it uses to build against. And it tries to load "libmpi.so" from the `LD_LIBRARY_PATH`. This may or may not find the MPI library that LAMMPS is using. If you have problems running both mpi4py and LAMMPS together, this is an issue you may need to address, e.g. by loading the module for different MPI installation so that mpi4py finds the right one.

If you have successfully installed mpi4py, you should be able to run Python and type

```
from mpi4py import MPI
```

without error. You should also be able to run Python in parallel on a simple test script

```
mpirun -np 4 python3 test.py
```

where `test.py` contains the lines

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
print("Proc %d out of %d procs" % (comm.Get_rank(),comm.Get_size()))
```

and see one line of output for each processor you run on. Please note that the order of the lines is not deterministic

```
$ mpirun -np 4 python3 test.py
Proc 0 out of 4 procs
Proc 1 out of 4 procs
Proc 2 out of 4 procs
Proc 3 out of 4 procs
```

2.3 Run LAMMPS from Python

After compiling the LAMMPS shared library and making it ready to use, you can now write and run Python scripts that import the LAMMPS Python module and launch LAMMPS simulations through Python scripts. The following pages take you through the various steps necessary, what functionality is available and give some examples how to use it.

2.3.1 Running LAMMPS and Python in serial

To run a LAMMPS in serial, type these lines into Python interactively from the bench directory:

```
from lammps import lammps
lmp = lammps()
lmp.file("in.lj")
```

Or put the same lines in the file `test.py` and run it as

```
python3 test.py
```

Either way, you should see the results of running the `in.lj` benchmark on a single processor appear on the screen, the same as if you had typed something like:

```
lmp_serial -in in.lj
```

2.3.2 Running LAMMPS and Python in parallel with MPI

To run LAMMPS in parallel, assuming you have installed the mpi4py package as discussed *Extending Python to run in parallel*, create a `test.py` file containing these lines:

```
from mpi4py import MPI
from lammps import lammps
lmp = lammps()
lmp.file("in.lj")
me = MPI.COMM_WORLD.Get_rank()
nprocs = MPI.COMM_WORLD.Get_size()
print("Proc %d out of %d procs has" % (me,nprocs),lmp)
MPI.Finalize()
```

You can run the script in parallel as:

```
mpirun -np 4 python3 test.py
```

and you should see the same output as if you had typed

```
mpirun -np 4 lmp_mpi -in in.lj
```

Note that without the mpi4py specific lines from `test.py`

```
from lammps import lammps
lmp = lammps()
lmp.file("in.lj")
```

running the script with `mpirun` on *P* processors would lead to *P* independent simulations to run parallel, each with a single processor. Therefore, if you use the mpi4py lines and you see multiple LAMMPS single processor outputs, mpi4py is not working correctly.

Also note that once you import the mpi4py module, mpi4py initializes MPI for you, and you can use MPI calls directly in your Python script, as described in the mpi4py documentation. The last line of your Python script should be `MPI.finalize()`, to ensure MPI is shut down correctly.

2.3.3 Running Python scripts

Note that any Python script (not just for LAMMPS) can be invoked in one of several ways:

```
python script.py
python -i script.py
./script.py
```

The last command requires that the first line of the script be something like this:

```
#!/usr/bin/python
```

or

```
#!/usr/bin/env python
```

where the path in the first case needs to point to where you have Python installed (the second option is workaround for when this may change), and that you have made the script file executable:

```
chmod +x script.py
```

Without the `-i` flag, Python will exit when the script finishes. With the `-i` flag, you will be left in the Python interpreter when the script finishes, so you can type subsequent commands. As mentioned above, you can only run Python interactively when running Python on a single processor, not in parallel.

2.3.4 Creating or deleting a LAMMPS object

With the Python interface the creation of a *LAMMPS* instance is included in the constructors for the *lammps*, *PyLammps*, and *IPyLammps* classes. Internally it will call either *lammps_open()* or *lammps_open_no_mpi()* from the C library API to create the class instance.

All arguments are optional. The *name* argument allows loading a LAMMPS shared library that is named `liblammps_machine.so` instead of the default name of `liblammps.so`. In most cases the latter will be installed or used. The *ptr* argument is for use of the *lammps* module from inside a LAMMPS instance, e.g. with the *python* command, where a pointer to the already existing *LAMMPS* class instance can be passed to the Python class and used instead of creating a new instance. The *comm* argument may be used in combination with the mpi4py module to pass an MPI communicator to LAMMPS and thus it is possible to run the Python module like the library interface on a subset of the MPI ranks after splitting the communicator.

Here are simple examples using all three Python interfaces:

---

lammps API

```
from lammps import lammps

# NOTE: argv[0] is set by the lammps class constructor
args = ["-log", "none"]

# create LAMMPS instance
lmp = lammps(cmdargs=args)

# get and print numerical version code
print("LAMMPS Version: ", lmp.version())

# explicitly close and delete LAMMPS instance (optional)
lmp.close()
```

PyLammps API

The *PyLammps* class is a wrapper around the *lammps* class and all of its lower level functions. By default, it will create a new instance of *lammps* passing along all arguments to the constructor of *lammps*.

```
from lammps import PyLammps

# NOTE: argv[0] is set by the lammps class constructor
args = ["-log", "none"]

# create LAMMPS instance
L = PyLammps(cmdargs=args)

# get and print numerical version code
print("LAMMPS Version: ", L.version())

# explicitly close and delete LAMMPS instance (optional)
L.close()
```

PyLammps objects can also be created on top of an existing *lammps* object:

```
from lammps import lammps, PyLammps
...
# create LAMMPS instance
lmp = lammps(cmdargs=args)

# create PyLammps instance using previously created LAMMPS instance
L = PyLammps(ptr=lmp)
```

This is useful if you have to create the *lammps* instance is a specific way, but want to take advantage of the *PyLammps* interface.

IPyLammps API

The *IPyLammps* class is an extension of the *PyLammps* class. It has the same construction behavior. By default, it will create a new instance of *lammps* passing along all arguments to the constructor of *lammps*.

```
from lammps import IPyLammps

# NOTE: argv[0] is set by the lammps class constructor
args = ["-log", "none"]

# create LAMMPS instance
L = IPyLammps(cmdargs=args)

# get and print numerical version code
print("LAMMPS Version: ", L.version())

# explicitly close and delete LAMMPS instance (optional)
L.close()
```

You can also initialize IPyLammps on top of an existing *lammps* or PyLammps object:

```
from lammps import lammps, IPyLammps
...
# create LAMMPS instance
lmp = lammps(cmdargs=args)

# create PyLammps instance using previously created LAMMPS instance
L = PyLammps(ptr=lmp)
```

This is useful if you have to create the *lammps* instance is a specific way, but want to take advantage of the *IPyLammps* interface.

In all of the above cases, same as with the *C library API*, this will use the MPI_COMM_WORLD communicator for the MPI library that LAMMPS was compiled with.

The *lmp.close()* call is optional since the LAMMPS class instance will also be deleted automatically during the *lammps* class destructor. Instead of *lmp.close()* it is also possible to call *lmp.finalize()*; this will destruct the LAMMPS instance, but also finalized and release the MPI and/or Kokkos environment if enabled and active.

Note that you can create multiple LAMMPS objects in your Python script, and coordinate and run multiple simulations, e.g.

```
from lammps import lammps
lmp1 = lammps()
lmp2 = lammps()
lmp1.file("in.file1")
lmp2.file("in.file2")
```

2.3.5 Executing commands

Once an instance of the *lammps*, *PyLammps*, or *IPyLammps* class is created, there are multiple ways to "feed" it commands. In a way that is not very different from running a LAMMPS input script, except that Python has many more facilities for structured programming than the LAMMPS input script syntax. Furthermore it is possible to "compute" what the next LAMMPS command should be.

---

lammps API

Same as in the equivalent *C library functions*, commands can be read from a file, a single string, a list of strings and a block of commands in a single multi-line string. They are processed under the same boundary conditions as the C library counterparts. The example below demonstrates the use of `lammps.file()`, `lammps.command()`, `lammps.commands_list()`, and `lammps.commands_string()`:

```
from lammps import lammps
lmp = lammps()

# read commands from file 'in.melt'
lmp.file('in.melt')

# issue a single command
lmp.command('variable zpos index 1.0')

# create 10 groups with 10 atoms each
cmds = ["group g{} id {}:{}".format(i,10*i+1,10*(i+1)) for i in range(10)]
lmp.commands_list(cmds)

# run commands from a multi-line string
block = """
clear
region  box block 0 2 0 2 0 2
create_box 1 box
create_atoms 1 single 1.0 1.0 ${zpos}
"""
lmp.commands_string(block)
```

PyLammps/IPyLammps API

Unlike the lammps API, the PyLammps/IPyLammps APIs allow running LAMMPS commands by calling equivalent member functions of *PyLammps* and *IPyLammps* instances.

For instance, the following LAMMPS command

```
region box block 0 10 0 5 -0.5 0.5
```

can be executed using with the lammps API with the following Python code if `lmp` is an instance of *lammps*:

```
from lammps import lammps

lmp = lammps()
lmp.command("region box block 0 10 0 5 -0.5 0.5")
```

With the PyLammps interface, any LAMMPS command can be split up into arbitrary parts. These parts are then passed to a member function with the name of the *command*. For the *region* command that means the `region()` method can be called. The arguments of the command can be passed as one string, or individually.

```
from lammps import PyLammps

L = PyLammps()

# pass command parameters as one string
```

(continues on next page)

(continued from previous page)

```
L.region("box block 0 10 0 5 -0.5 0.5")

# OR pass them individually
L.region("box block", 0, 10, 0, 5, -0.5, 0.5)
```

In the latter example, all parameters except the first are Python floating-point literals. The member function takes the entire parameter list and transparently merges it to a single command string.

The benefit of this approach is avoiding redundant command calls and easier parameterization. In the lammps API parameterization needed to be done manually by creating formatted command strings.

```
lmp.command("region box block %f %f %f %f %f %f" % (xlo, xhi, ylo, yhi, zlo,␣
↪zhi))
```

In contrast, methods of PyLammps accept parameters directly and will convert them automatically to a final command string.

```
L.region("box block", xlo, xhi, ylo, yhi, zlo, zhi)
```

Using these facilities, the example shown for the lammps API can be rewritten as follows:

```
from lammps import PyLammps
L = PyLammps()

# read commands from file 'in.melt'
L.file('in.melt')

# issue a single command
L.variable('zpos', 'index', 1.0)

# create 10 groups with 10 atoms each
for i in range(10):
   L.group(f"g{i}", "id", f"{10*i+1}:{10*(i+1)}")

L.clear()
L.region("box block", 0, 2, 0, 2, 0, 2)
L.create_box(1, "box")
L.create_atoms(1, "single", 1.0, 1.0, "${zpos}")
```

2.3.6 System properties

Similar to what is described in *System properties*, the instances of `lammps`, `PyLammps`, or `IPyLammps` can be used to extract different kinds of information about the active LAMMPS instance and also to modify some of it. The main difference between the interfaces is how the information is exposed.

While the `lammps` is just a thin layer that wraps C API calls, `PyLammps` and `IPyLammps` expose information as objects and properties.

In some cases the data returned is a direct reference to the original data inside LAMMPS cast to `ctypes` pointers. Where possible, the wrappers will determine the `ctypes` data type and cast pointers accordingly. If `numpy` is installed arrays can also be extracted as numpy arrays, which will access the C arrays directly and have the correct dimensions to protect against invalid accesses.

Warning: When accessing per-atom data, please note that this data is the per-processor local data and indexed accordingly. These arrays can change sizes and order at every neighbor list rebuild and atom sort event as atoms are migrating between subdomains.

---

lammps API

```
from lammps import lammps

lmp = lammps()
lmp.file("in.sysinit")

natoms = lmp.get_natoms()
print(f"running simulation with {natoms} atoms")

lmp.command("run 1000 post no");

for i in range(10):
   lmp.command("run 100 pre no post no")
   pe = lmp.get_thermo("pe")
   ke = lmp.get_thermo("ke")
   print(f"PE = {pe}\nKE = {ke}")

lmp.close()
```

Methods:

- *version()*: return the numerical version id, e.g. LAMMPS 2 Sep 2015 -> 20150902
- *get_thermo()*: return current value of a thermo keyword
- *last_thermo()*: return a dictionary of the last thermodynamic output
- *get_natoms()*: total # of atoms as int
- *reset_box()*: reset the simulation box size
- *extract_setting()*: return a global setting
- *extract_global()*: extract a global quantity
- *extract_box()*: extract box info
- *create_atoms()*: create N atoms with IDs, types, x, v, and image flags

Properties:

- *last_thermo_step*: the last timestep thermodynamic output was computed

---

PyLammps/IPyLammps API

In addition to the functions provided by *lammps*, *PyLammps* objects have several properties which allow you to query the system state:

L.system
: Is a dictionary describing the system such as the bounding box or number of atoms

L.system.xlo, L.system.xhi
: bounding box limits along x-axis

L.system.ylo, L.system.yhi
bounding box limits along y-axis

L.system.zlo, L.system.zhi
bounding box limits along z-axis

L.communication
configuration of communication subsystem, such as the number of threads or processors

L.communication.nthreads
number of threads used by each LAMMPS process

L.communication.nprocs
number of MPI processes used by LAMMPS

L.fixes
List of fixes in the current system

L.computes
List of active computes in the current system

L.dump
List of active dumps in the current system

L.groups
List of groups present in the current system

Retrieving the value of an arbitrary LAMMPS expressions

LAMMPS expressions can be immediately evaluated by using the `eval` method. The passed string parameter can be any expression containing global *thermo command* values, variables, compute or fix data (see *Output from LAMMPS (thermo, dumps, computes, fixes, variables)*):

```
result = L.eval("ke") # kinetic energy
result = L.eval("pe") # potential energy

result = L.eval("v_t/2.0")
```

Example

```
from lammps import PyLammps

L = PyLammps()
L.file("in.sysinit")

print(f"running simulation with {L.system.natoms} atoms")

L.run(1000, "post no");

for i in range(10):
   L.run(100, "pre no post no")
   pe = L.eval("pe")
   ke = L.eval("ke")
   print(f"PE = {pe}\nKE = {ke}")
```

2.3.7 Per-atom properties

Similar to what is described in *Per-atom properties*, the instances of `lammps`, `PyLammps`, or `IPyLammps` can be used to extract atom quantities and modify some of them. The main difference between the interfaces is how the information is exposed.

While the `lammps` is just a thin layer that wraps C API calls, `PyLammps` and `IPyLammps` expose information as objects and properties.

In some cases the data returned is a direct reference to the original data inside LAMMPS cast to `ctypes` pointers. Where possible, the wrappers will determine the `ctypes` data type and cast pointers accordingly. If `numpy` is installed arrays can also be extracted as numpy arrays, which will access the C arrays directly and have the correct dimensions to protect against invalid accesses.

Warning: When accessing per-atom data, please note that this data is the per-processor local data and indexed accordingly. These arrays can change sizes and order at every neighbor list rebuild and atom sort event as atoms are migrating between subdomains.

---

lammps API

```
from lammps import lammps

lmp = lammps()
lmp.file("in.sysinit")

nlocal = lmp.extract_global("nlocal")
x = lmp.extract_atom("x")

for i in range(nlocal):
   print("(x,y,z) = (", x[i][0], x[i][1], x[i][2], ")")

lmp.close()
```

Methods:

- `extract_atom()`: extract a per-atom quantity

Numpy Methods:

- `numpy.extract_atom()`: extract a per-atom quantity as numpy array

PyLammps/IPyLammps API

All atoms in the current simulation can be accessed by using the `PyLammps.atoms` property. Each element of this list is a `Atom` or `Atom2D` object. The attributes of these objects provide access to their data (id, type, position, velocity, force, etc.):

```
# access first atom
L.atoms[0].id
L.atoms[0].type

# access second atom
L.atoms[1].position
L.atoms[1].velocity
L.atoms[1].force
```

Some attributes can be changed:

```
# set position in 2D simulation
L.atoms[0].position = (1.0, 0.0)

# set position in 3D simulation
L.atoms[0].position = (1.0, 0.0, 1.0)
```

2.3.8 Compute, fixes, variables

This section documents accessing or modifying data from objects like computes, fixes, or variables in LAMMPS using the *lammps* module.

---

lammps API

For *lammps.extract_compute()* and *lammps.extract_fix()*, the global, per-atom, or local data calculated by the compute or fix can be accessed. What is returned depends on whether the compute or fix calculates a scalar or vector or array. For a scalar, a single double value is returned. If the compute or fix calculates a vector or array, a pointer to the internal LAMMPS data is returned, which you can use via normal Python subscripting.

The one exception is that for a fix that calculates a global vector or array, a single double value from the vector or array is returned, indexed by I (vector) or I and J (array). I,J are zero-based indices. See the *Howto output* page for a discussion of global, per-atom, and local data, and of scalar, vector, and array data types. See the doc pages for individual *computes* and *fixes* for a description of what they calculate and store.

For *lammps.extract_variable()*, an *equal-style or atom-style variable* is evaluated and its result returned.

For equal-style variables a single `c_double` value is returned and the group argument is ignored. For atom-style variables, a vector of `c_double` is returned, one value per atom, which you can use via normal Python subscripting. The values will be zero for atoms not in the specified group.

lammps.numpy.extract_compute(), *lammps.numpy.extract_fix()*, and *lammps.numpy.extract_variable()* are equivalent NumPy implementations that return NumPy arrays instead of `ctypes` pointers.

The *lammps.set_variable()* method sets an existing string-style variable to a new string value, so that subsequent LAMMPS commands can access the variable.

Methods:

- *lammps.extract_compute()*: extract value(s) from a compute
- *lammps.extract_fix()*: extract value(s) from a fix
- *lammps.extract_variable()*: extract value(s) from a variable
- *lammps.set_variable()*: set existing named string-style variable to value

NumPy Methods:

- *lammps.numpy.extract_compute()*: extract value(s) from a compute, return arrays as numpy arrays
- *lammps.numpy.extract_fix()*: extract value(s) from a fix, return arrays as numpy arrays

- *lammps.numpy.extract_variable()*: extract value(s) from a variable, return arrays as numpy arrays

---

PyLammps/IPyLammps API

PyLammps and IPyLammps classes currently do not add any additional ways of retrieving information out of computes and fixes. This information can still be accessed by using the lammps API:

```
L.lmp.extract_compute(...)
L.lmp.extract_fix(...)
# OR
L.lmp.numpy.extract_compute(...)
L.lmp.numpy.extract_fix(...)
```

LAMMPS variables can be both defined and accessed via the *PyLammps* interface.

To define a variable you can use the *variable* command:

```
L.variable("a index 2")
```

A dictionary of all variables is returned by the *PyLammps.variables* property:

you can access an individual variable by retrieving a variable object from the `L.variables` dictionary by name

```
a = L.variables['a']
```

The variable value can then be easily read and written by accessing the value property of this object.

```
print(a.value)
a.value = 4
```

2.3.9 Scatter/gather operations

```
data = lmp.gather_atoms(name,type,count)  # return per-atom property of all atoms␣
↪gathered into data, ordered by atom ID
                                          # name = "x", "charge", "type", etc
data = lmp.gather_atoms_concat(name,type,count)  # ditto, but concatenated atom values␣
↪from each proc (unordered)
data = lmp.gather_atoms_subset(name,type,count,ndata,ids)  # ditto, but for subset of␣
↪Ndata atoms with IDs

lmp.scatter_atoms(name,type,count,data)   # scatter per-atom property to all atoms from␣
↪data, ordered by atom ID
                                          # name = "x", "charge", "type", etc
                                          # count = # of per-atom values, 1 or 3, etc

lmp.scatter_atoms_subset(name,type,count,ndata,ids,data)  # ditto, but for subset of␣
↪Ndata atoms with IDs
```

The gather methods collect peratom info of the requested type (atom coords, atom types, forces, etc) from all processors, and returns the same vector of values to each calling processor. The scatter functions do the inverse. They distribute

a vector of peratom values, passed by all calling processors, to individual atoms, which may be owned by different processors.

Note that the data returned by the gather methods, e.g. gather_atoms("x"), is different from the data structure returned by extract_atom("x") in four ways. (1) Gather_atoms() returns a vector which you index as x[i]; extract_atom() returns an array which you index as x[i][j]. (2) Gather_atoms() orders the atoms by atom ID while extract_atom() does not. (3) Gather_atoms() returns a list of all atoms in the simulation; extract_atoms() returns just the atoms local to each processor. (4) Finally, the gather_atoms() data structure is a copy of the atom coords stored internally in LAMMPS, whereas extract_atom() returns an array that effectively points directly to the internal data. This means you can change values inside LAMMPS from Python by assigning a new values to the extract_atom() array. To do this with the gather_atoms() vector, you need to change values in the vector, then invoke the scatter_atoms() method.

For the scatter methods, the array of coordinates passed to must be a ctypes vector of ints or doubles, allocated and initialized something like this:

```
from ctypes import c_double
natoms = lmp.get_natoms()
n3 = 3*natoms
x = (n3*c_double)()
x[0] = x coord of atom with ID 1
x[1] = y coord of atom with ID 1
x[2] = z coord of atom with ID 1
x[3] = x coord of atom with ID 2
...
x[n3-1] = z coord of atom with ID natoms
lmp.scatter_atoms("x", 1, 3, x)
```

The coordinates can also be provided as arguments to the initializer of x:

```
from ctypes import c_double
natoms = 2
n3 = 3*natoms
# init in constructor
x = (n3*c_double)(0.0, 0.0, 0.0, 1.0, 1.0, 1.0)
lmp.scatter_atoms("x", 1, 3, x)
# or using a list
coords = [1.0, 2.0, 3.0, -3.0, -2.0, -1.0]
x = (c_double*len(coords))(*coords)
```

Alternatively, you can just change values in the vector returned by the gather methods, since they are also ctypes vectors.

2.3.10 Neighbor list access

Access to neighbor lists is handled through a couple of wrapper classes that allows to treat it like either a python list or a NumPy array. The access procedure is similar to that of the C-library interface: use one of the "find" functions to look up the index of the neighbor list in the global table of neighbor lists and then get access to the neighbor list data. The code sample below demonstrates reading the neighbor list data using the NumPy access method.

```
from lammps import lammps
import numpy as np

lmp = lammps()
lmp.commands_string("""
region box block -2 2 -2 2 -2 2
```

(continues on next page)

(continued from previous page)

```
lattice fcc 1.0
create_box 1 box
create_atoms 1 box
mass 1 1.0
pair_style lj/cut 2.5
pair_coeff 1 1 1.0 1.0
run 0 post no""")

# look up the neighbor list
nlidx = lmp.find_pair_neighlist('lj/cut')
nl = lmp.numpy.get_neighlist(nlidx)
tags = lmp.extract_atom('id')
print("half neighbor list with {} entries".format(nl.size))
# print neighbor list contents
for i in range(0,nl.size):
    idx, nlist  = nl.get(i)
    print("\natom {} with ID {} has {} neighbors:".format(idx,tags[idx],nlist.size))
    if nlist.size > 0:
        for n in np.nditer(nlist):
            print("  atom {} with ID {}".format(n,tags[n]))
```

Another example for extracting a full neighbor list without evaluating a potential is shown below.

```
from lammps import lammps
import numpy as np

lmp = lammps()
lmp.commands_string("""
newton off
region box block -2 2 -2 2 -2 2
lattice fcc 1.0
create_box 1 box
create_atoms 1 box
mass 1 1.0
pair_style zero 1.0 full
pair_coeff * *
run 0 post no""")

# look up the neighbor list
nlidx = lmp.find_pair_neighlist('zero')
nl = lmp.numpy.get_neighlist(nlidx)
tags = lmp.extract_atom('id')
print("full neighbor list with {} entries".format(nl.size))
# print neighbor list contents
for i in range(0,nl.size):
    idx, nlist  = nl.get(i)
    print("\natom {} with ID {} has {} neighbors:".format(idx,tags[idx],nlist.size))
    if nlist.size > 0:
        for n in np.nditer(nlist):
            pass
            print("  atom {} with ID {}".format(n,tags[n]))
```

Methods:

- *lammps.get_neighlist()*: Get neighbor list for given index
- lammps.get_neighlist_size(): Get number of elements in neighbor list
- lammps.get_neighlist_element_neighbors(): Get element in neighbor list and its neighbors
- *lammps.find_pair_neighlist()*: Find neighbor list of pair style
- *lammps.find_fix_neighlist()*: Find neighbor list of pair style
- *lammps.find_compute_neighlist()*: Find neighbor list of pair style

NumPy Methods:

- *lammps.numpy.get_neighlist()*: Get neighbor list for given index, which uses NumPy arrays for its element neighbor arrays
- *lammps.numpy.get_neighlist_element_neighbors()*: Get element in neighbor list and its neighbors (as numpy array)

2.3.11 Configuration information

The following methods can be used to query the LAMMPS library about compile time settings and included packages and styles.

Listing 1: Example for using configuration settings functions

```
from lammps import lammps

lmp = lammps()

try:
    lmp.file("in.missing")
except Exception as e:
    print("LAMMPS failed with error:", e)

# write compressed dump file depending on available of options

if lmp.has_style("dump", "atom/zstd"):
    lmp.command("dump d1 all atom/zstd 100 dump.zst")
elif lmp.has_style("dump", "atom/gz"):
    lmp.command("dump d1 all atom/gz 100 dump.gz")
elif lmp.has_gzip_support():
    lmp.command("dump d1 all atom 100 dump.gz")
else:
    lmp.command("dump d1 all atom 100 dump")
```

Methods:

- *lammps.has_mpi_support*
- *lammps.has_exceptions*
- *lammps.has_gzip_support*
- *lammps.has_png_support*
- *lammps.has_jpeg_support*

- *lammps.has_ffmpeg_support*
- *lammps.installed_packages*
- *lammps.get_accelerator_config*
- *lammps.has_style()*
- *lammps.available_styles()*

2.4 The `lammps` Python module

The LAMMPS Python interface is implemented as a module called *lammps* which is defined in the `lammps` package in the `python` folder of the LAMMPS source code distribution. After compilation of LAMMPS, the module can be installed into a Python system folder or a user folder with `make install-python`. Components of the module can then loaded into a Python session with the `import` command.

There are multiple Python interface classes in the *lammps* module:

- the *lammps* class. This is a wrapper around the C-library interface and its member functions try to replicate the *C-library API* closely. This is the most feature-complete Python API.
- the *PyLammps* class. This is a more high-level and more Python style class implemented on top of the *lammps* class.
- the *IPyLammps* class is derived from *PyLammps* and adds embedded graphics features to conveniently include LAMMPS into Jupyter notebooks.

Version check

The *lammps* module stores the version number of the LAMMPS version it is installed from. When initializing the *lammps* class, this version is checked to be the same as the result from *lammps.version()*, the version of the LAMMPS shared library that the module interfaces to. If the they are not the same an AttributeError exception is raised since a mismatch of versions (e.g. due to incorrect use of the `LD_LIBRARY_PATH` or `PYTHONPATH` environment variables can lead to crashes or data corruption and otherwise incorrect behavior.

LAMMPS module global members:

`lammps.__version__`

Numerical representation of the LAMMPS version this module was taken from. Has the same format as the result of *lammps.version()*.

2.4.1 The `lammps` class API

The *lammps* class is the core of the LAMMPS Python interfaces. It is a wrapper around the *LAMMPS C library API* using the Python ctypes module and a shared library compiled from the LAMMPS sources code. The individual methods in this class try to closely follow the corresponding C functions. The handle argument that needs to be passed to the C functions is stored internally in the class and automatically added when calling the C library functions. Below is a detailed documentation of the API.

class lammps.**lammps**(*name=''*, *cmdargs=None*, *ptr=None*, *comm=None*)

Create an instance of the LAMMPS Python class.

This is a Python wrapper class that exposes the LAMMPS C-library interface to Python. It either requires that LAMMPS has been compiled as shared library which is then dynamically loaded via the ctypes Python module or that this module called from a Python function that is called from a Python interpreter embedded into a LAMMPS executable, for example through the *python invoke* command. When the class is instantiated it calls the *lammps_open()* function of the LAMMPS C-library interface, which in turn will create an instance of the *LAMMPS* C++ class. The handle to this C++ class is stored internally and automatically passed to the calls to the C library interface.

Parameters

- **name** (*string*) – "machine" name of the shared LAMMPS library ("mpi" loads `liblammps_mpi.so`, "" loads `liblammps.so`)
- **cmdargs** (*list*) – list of command line arguments to be passed to the *lammps_open()* function. The executable name is automatically added.
- **ptr** (*pointer*) – pointer to a LAMMPS C++ class instance when called from an embedded Python interpreter. None means load symbols from shared library.
- **comm** (*MPI_Comm*) – MPI communicator (as provided by mpi4py). `None` means use `MPI_COMM_WORLD` implicitly.

property numpy

Return object to access numpy versions of API

It provides alternative implementations of API functions that return numpy arrays instead of ctypes pointers. If numpy is not installed, accessing this property will lead to an ImportError.

Returns
instance of numpy wrapper object

Return type
numpy_wrapper

close()

Explicitly delete a LAMMPS instance through the C-library interface.

This is a wrapper around the *lammps_close()* function of the C-library interface.

finalize()

Shut down the MPI communication and Kokkos environment (if active) through the library interface by calling *lammps_mpi_finalize()* and *lammps_kokkos_finalize()*.

You cannot create or use any LAMMPS instances after this function is called unless LAMMPS was compiled without MPI and without Kokkos support.

error(*error_type*, *error_text*)

Forward error to the LAMMPS Error class.

Added in version 3Nov2022.

This is a wrapper around the *lammps_error()* function of the C-library interface.

Parameters

- **error_type** (*int*)
- **error_text** (*string*)

version()

Return a numerical representation of the LAMMPS version in use.

This is a wrapper around the *lammps_version()* function of the C-library interface.

Returns
version number

Return type
int

get_os_info()

Return a string with information about the OS and compiler runtime

This is a wrapper around the *lammps_get_os_info()* function of the C-library interface.

Returns
OS info string

Return type
string

get_mpi_comm()

Get the MPI communicator in use by the current LAMMPS instance

This is a wrapper around the *lammps_get_mpi_comm()* function of the C-library interface. It will return `None` if either the LAMMPS library was compiled without MPI support or the mpi4py Python module is not available.

Returns
MPI communicator

Return type
MPI_Comm

file(*path*)

Read LAMMPS commands from a file.

This is a wrapper around the *lammps_file()* function of the C-library interface. It will open the file with the name/path *file* and process the LAMMPS commands line by line until the end. The function will return when the end of the file is reached.

Parameters
path (*string*) – Name of the file/path with LAMMPS commands

command(*cmd*)

Process a single LAMMPS input command from a string.

This is a wrapper around the *lammps_command()* function of the C-library interface.

Parameters
cmd (*string*) – a single lammps command

commands_list(*cmdlist*)

Process multiple LAMMPS input commands from a list of strings.

This is a wrapper around the *lammps_commands_list()* function of the C-library interface.

Parameters
cmdlist (*list of strings*) – a single lammps command

commands_string(*multicmd*)

Process a block of LAMMPS input commands from a string.

This is a wrapper around the *lammps_commands_string()* function of the C-library interface.

Parameters
multicmd (*string*) – text block of lammps commands

get_natoms()

Get the total number of atoms in the LAMMPS instance.

Will be precise up to 53-bit signed integer due to the underlying *lammps_get_natoms()* function returning a double.

Returns
number of atoms

Return type
int

extract_box()

Extract simulation box parameters

This is a wrapper around the *lammps_extract_box()* function of the C-library interface. Unlike in the C function, the result is returned as a list.

Returns
list of the extracted data: boxlo, boxhi, xy, yz, xz, periodicity, box_change

Return type
[3*double, 3*double, double, double, 3*int, int]

reset_box(*boxlo*, *boxhi*, *xy*, *yz*, *xz*)

Reset simulation box parameters

This is a wrapper around the *lammps_reset_box()* function of the C-library interface.

Parameters

- **boxlo** (*list of 3 floating point numbers*) – new lower box boundaries
- **boxhi** (*list of 3 floating point numbers*) – new upper box boundaries
- **xy** (*float*) – xy tilt factor
- **yz** (*float*) – yz tilt factor
- **xz** (*float*) – xz tilt factor

get_thermo(*name*)

Get current value of a thermo keyword

This is a wrapper around the *lammps_get_thermo()* function of the C-library interface.

Parameters
name (*string*) – name of thermo keyword

Returns
value of thermo keyword

Return type
double or None

property last_thermo_step

Get the last timestep where thermodynamic data was computed

Returns
the timestep or a negative number if there has not been any thermo output yet

Return type
int

last_thermo()

Get a dictionary of the last thermodynamic output

This is a wrapper around the *lammps_last_thermo()* function of the C-library interface. It collects the cached thermo data from the last timestep into a dictionary. The return value is None, if there has not been any thermo output yet.

Returns
a dictionary containing the last computed thermo output values

Return type
dict or None

extract_setting(*name*)

Query LAMMPS about global settings that can be expressed as an integer.

This is a wrapper around the *lammps_extract_setting()* function of the C-library interface. Its documentation includes a list of the supported keywords.

Parameters
name (*string*) – name of the setting

Returns
value of the setting

Return type
int

extract_global_datatype(*name*)

Retrieve global property datatype from LAMMPS

This is a wrapper around the *lammps_extract_global_datatype()* function of the C-library interface. Its documentation includes a list of the supported keywords. This function returns `None` if the keyword is not recognized. Otherwise it will return a positive integer value that corresponds to one of the *data type* constants define in the *lammps* module.

Parameters
name (*string*) – name of the property

Returns
data type of global property, see *Data Types*

Return type
int

extract_global(*name*, *dtype=None*)

Query LAMMPS about global settings of different types.

This is a wrapper around the *lammps_extract_global()* function of the C-library interface. Since there are no pointers in Python, this method will - unlike the C function - return the value or a list of values. The *lammps_extract_global()* documentation includes a list of the supported keywords and their data types. Since Python needs to know the data type to be able to interpret the result, by default, this function will try to auto-detect the data type by asking the library. You can also force a specific data type. For that

purpose the *lammps* module contains *data type* constants. This function returns None if either the keyword is not recognized, or an invalid data type constant is used.

Parameters

- **name** (*string*) – name of the property
- **dtype** (*int, optional*) – data type of the returned data (see *Data Types*)

Returns
value of the property or list of values or None

Return type
int, float, list, or NoneType

map_atom(*id*)

Map a global atom ID (aka tag) to the local atom index

This is a wrapper around the *lammps_map_atom()* function of the C-library interface.

Parameters
id (*int*) – atom ID

Returns
local index

Return type
int

extract_atom_datatype(*name*)

Retrieve per-atom property datatype from LAMMPS

This is a wrapper around the *lammps_extract_atom_datatype()* function of the C-library interface. Its documentation includes a list of the supported keywords. This function returns None if the keyword is not recognized. Otherwise it will return an integer value that corresponds to one of the *data type* constants defined in the *lammps* module.

Parameters
name (*string*) – name of the property

Returns
data type of per-atom property (see *Data Types*)

Return type
int

extract_atom(*name*, *dtype=None*)

Retrieve per-atom properties from LAMMPS

This is a wrapper around the *lammps_extract_atom()* function of the C-library interface. Its documentation includes a list of the supported keywords and their data types. Since Python needs to know the data type to be able to interpret the result, by default, this function will try to auto-detect the data type by asking the library. You can also force a specific data type by setting dtype to one of the *data type* constants defined in the *lammps* module. This function returns None if either the keyword is not recognized, or an invalid data type constant is used.

Note: While the returned arrays of per-atom data are dimensioned for the range [0:nmax] - as is the underlying storage - the data is usually only valid for the range of [0:nlocal], unless the property of interest is also updated for ghost atoms. In some cases, this depends on a LAMMPS setting, see for example *comm_modify vel yes*.

Parameters

- **name** (*string*) – name of the property
- **dtype** (*int, optional*) – data type of the returned data (see *Data Types*)

Returns
requested data or None

Return type
ctypes.POINTER(ctypes.c_int32), ctypes.POINTER(ctypes.POINTER(ctypes.c_int32)), ctypes.POINTER(ctypes.c_int64), ctypes.POINTER(ctypes.POINTER(ctypes.c_int64)), ctypes.POINTER(ctypes.c_double), ctypes.POINTER(ctypes.POINTER(ctypes.c_double)), or NoneType

extract_compute(*cid*, *cstyle*, *ctype*)

Retrieve data from a LAMMPS compute

This is a wrapper around the *lammps_extract_compute()* function of the C-library interface. This function returns None if either the compute id is not recognized, or an invalid combination of *cstyle* and *ctype* constants is used. The names and functionality of the constants are the same as for the corresponding C-library function. For requests to return a scalar or a size, the value is returned, otherwise a pointer.

Parameters

- **cid** (*string*) – compute ID
- **cstyle** (*int*) – style of the data retrieve (global, atom, or local), see *Style Constants*
- **ctype** (*int*) – type or size of the returned data (scalar, vector, or array), see *Type Constants*

Returns
requested data as scalar, pointer to 1d or 2d double array, or None

Return type
c_double, ctypes.POINTER(c_double), ctypes.POINTER(ctypes.POINTER(c_double)), or NoneType

extract_fix(*fid*, *fstyle*, *ftype*, *nrow=0*, *ncol=0*)

Retrieve data from a LAMMPS fix

This is a wrapper around the *lammps_extract_fix()* function of the C-library interface. This function returns None if either the fix id is not recognized, or an invalid combination of *fstyle* and *ftype* constants is used. The names and functionality of the constants are the same as for the corresponding C-library function. For requests to return a scalar or a size, the value is returned, also when accessing global vectors or arrays, otherwise a pointer.

Note: When requesting global data, the fix data can only be accessed one item at a time without access to the whole vector or array. Thus this function will always return a scalar. To access vector or array elements the "nrow" and "ncol" arguments need to be set accordingly (they default to 0).

Parameters

- **fid** (*string*) – fix ID
- **fstyle** (*int*) – style of the data retrieve (global, atom, or local), see *Style Constants*
- **ftype** (*int*) – type or size of the returned data (scalar, vector, or array), see *Type Constants*
- **nrow** (*int*) – index of global vector element or row index of global array element

- **ncol** (*int*) – column index of global array element

Returns
requested data or None

Return type
c_double, ctypes.POINTER(c_double), ctypes.POINTER(ctypes.POINTER(c_double)), or NoneType

`extract_variable`(*name*, *group=None*, *vartype=None*)

Evaluate a LAMMPS variable and return its data

This function is a wrapper around the function *lammps_extract_variable()* of the C library interface, evaluates variable name and returns a copy of the computed data. The memory temporarily allocated by the C-interface is deleted after the data is copied to a Python variable or list. The variable must be either an equal-style (or equivalent) variable or an atom-style variable. The variable type can be provided as the `vartype` parameter, which may be one of several constants: `LMP_VAR_EQUAL`, `LMP_VAR_ATOM`, `LMP_VAR_VECTOR`, or `LMP_VAR_STRING`. If omitted or `None`, LAMMPS will determine its value for you based on a call to *lammps_extract_variable_datatype()* from the C library interface. The group parameter is only used for atom-style variables and defaults to the group "all".

Parameters

- **name** (*string*) – name of the variable to execute
- **group** (*string, only for atom-style variables*) – name of group for atom-style variable
- **vartype** (*int*) – type of variable, see *Variable Type Constants*

Returns
the requested data

Return type
c_double, (c_double), or NoneType

`flush_buffers`()

Flush output buffers

This is a wrapper around the *lammps_flush_buffers()* function of the C-library interface.

`set_variable`(*name*, *value*)

Set a new value for a LAMMPS string style variable

Deprecated since version 7Feb2024.

This is a wrapper around the *lammps_set_variable()* function of the C-library interface.

Parameters

- **name** (*string*) – name of the variable
- **value** (*any. will be converted to a string*) – new variable value

Returns
either 0 on success or -1 on failure

Return type
int

set_string_variable(*name*, *value*)

Set a new value for a LAMMPS string style variable

Added in version 7Feb2024.

This is a wrapper around the *lammps_set_string_variable()* function of the C-library interface.

Parameters

- **name** (*string*) – name of the variable
- **value** (*any. will be converted to a string*) – new variable value

Returns
either 0 on success or -1 on failure

Return type
int

set_internal_variable(*name*, *value*)

Set a new value for a LAMMPS internal style variable

Added in version 7Feb2024.

This is a wrapper around the *lammps_set_internal_variable()* function of the C-library interface.

Parameters

- **name** (*string*) – name of the variable
- **value** (*float or compatible. will be converted to float*) – new variable value

Returns
either 0 on success or -1 on failure

Return type
int

gather_bonds()

Retrieve global list of bonds

Added in version 28Jul2021.

This is a wrapper around the *lammps_gather_bonds()* function of the C-library interface.

This function returns a tuple with the number of bonds and a flat list of ctypes integer values with the bond type, bond atom1, bond atom2 for each bond.

Returns
a tuple with the number of bonds and a list of c_int or c_long

Return type
(int, 3*nbonds*c_tagint)

gather_angles()

Retrieve global list of angles

Added in version 8Feb2023.

This is a wrapper around the *lammps_gather_angles()* function of the C-library interface.

This function returns a tuple with the number of angles and a flat list of ctypes integer values with the angle type, angle atom1, angle atom2, angle atom3 for each angle.

Returns
a tuple with the number of angles and a list of c_int or c_long

Return type
(int, 4*nangles*c_tagint)

`gather_dihedrals()`

Retrieve global list of dihedrals

Added in version 8Feb2023.

This is a wrapper around the *lammps_gather_dihedrals()* function of the C-library interface.

This function returns a tuple with the number of dihedrals and a flat list of ctypes integer values with the dihedral type, dihedral atom1, dihedral atom2, dihedral atom3, dihedral atom4 for each dihedral.

Returns
a tuple with the number of dihedrals and a list of c_int or c_long

Return type
(int, 5*ndihedrals*c_tagint)

`gather_impropers()`

Retrieve global list of impropers

Added in version 8Feb2023.

This is a wrapper around the *lammps_gather_impropers()* function of the C-library interface.

This function returns a tuple with the number of impropers and a flat list of ctypes integer values with the improper type, improper atom1, improper atom2, improper atom3, improper atom4 for each improper.

Returns
a tuple with the number of impropers and a list of c_int or c_long

Return type
(int, 5*nimpropers*c_tagint)

`encode_image_flags`(*ix*, *iy*, *iz*)

convert 3 integers with image flags for x-, y-, and z-direction into a single integer like it is used internally in LAMMPS

This method is a wrapper around the *lammps_encode_image_flags()* function of library interface.

Parameters

- **ix** (*int*) – x-direction image flag
- **iy** (*int*) – y-direction image flag
- **iz** (*int*) – z-direction image flag

Returns
encoded image flags

Return type
lammps.c_imageint

`decode_image_flags`(*image*)

Convert encoded image flag integer into list of three regular integers.

This method is a wrapper around the *lammps_decode_image_flags()* function of library interface.

Parameters
image (*lammps.c_imageint*) – encoded image flags

Returns
list of three image flags in x-, y-, and z- direction

Return type
list of 3 int

create_atoms(*n*, *id*, *type*, *x*, *v=None*, *image=None*, *shrinkexceed=False*)

Create N atoms from list of coordinates and properties

This function is a wrapper around the *lammps_create_atoms()* function of the C-library interface, and the behavior is similar except that the *v*, *image*, and *shrinkexceed* arguments are optional and default to *None*, *None*, and *False*, respectively. With *None* being equivalent to a NULL pointer in C.

The lists of coordinates, types, atom IDs, velocities, image flags can be provided in any format that may be converted into the required internal data types. Also the list may contain more than *N* entries, but not fewer. In the latter case, the function will return without attempting to create atoms. You may use the *encode_image_flags* method to properly combine three integers with image flags into a single integer.

Parameters

- **n** (*int*) – number of atoms for which data is provided
- **id** (*list of lammps.tagint*) – list of atom IDs with at least n elements or None
- **type** (*list of int*) – list of atom types
- **x** (*list of float*) – list of coordinates for x-, y-, and z (flat list of 3n entries)
- **v** (*list of float*) – list of velocities for x-, y-, and z (flat list of 3n entries) or None (optional)
- **image** (*list of lammps.imageint*) – list of encoded image flags (optional)
- **shrinkexceed** (*bool*) – whether to expand shrink-wrap boundaries if atoms are outside the box (optional)

Returns
number of atoms created. 0 if insufficient or invalid data

Return type
int

property has_mpi_support

Report whether the LAMMPS shared library was compiled with a real MPI library or in serial.

This is a wrapper around the *lammps_config_has_mpi_support()* function of the library interface.

Returns
False when compiled with MPI STUBS, otherwise True

Return type
bool

property is_running

Report whether being called from a function during a run or a minimization

Added in version 9Oct2020.

Various LAMMPS commands must not be called during an ongoing run or minimization. This property allows to check for that. This is a wrapper around the *lammps_is_running()* function of the library interface.

Returns
True when called during a run otherwise false

Return type
bool

`force_timeout()`

Trigger an immediate timeout, i.e. a "soft stop" of a run.

Added in version 9Oct2020.

This function allows to cleanly stop an ongoing run or minimization at the next loop iteration. This is a wrapper around the *lammps_force_timeout()* function of the library interface.

`property has_exceptions`

Report whether the LAMMPS shared library was compiled with C++ exceptions handling enabled

This is a wrapper around the *lammps_config_has_exceptions()* function of the library interface.

Returns
state of C++ exception support

Return type
bool

`property has_gzip_support`

Report whether the LAMMPS shared library was compiled with support for reading and writing compressed files through `gzip`.

This is a wrapper around the *lammps_config_has_gzip_support()* function of the library interface.

Returns
state of gzip support

Return type
bool

`property has_png_support`

Report whether the LAMMPS shared library was compiled with support for writing images in PNG format.

This is a wrapper around the *lammps_config_has_png_support()* function of the library interface.

Returns
state of PNG support

Return type
bool

`property has_jpeg_support`

Report whether the LAMMPS shared library was compiled with support for writing images in JPEG format.

This is a wrapper around the *lammps_config_has_jpeg_support()* function of the library interface.

Returns
state of JPEG support

Return type
bool

`property has_ffmpeg_support`

State of support for writing movies with `ffmpeg` in the LAMMPS shared library

This is a wrapper around the *lammps_config_has_ffmpeg_support()* function of the library interface.

Returns
state of ffmpeg support

Return type
bool

has_package(*name*)

Report if the named package has been enabled in the LAMMPS shared library.

Added in version 3Nov2022.

This is a wrapper around the *lammps_config_has_package()* function of the library interface.

Parameters
name (*string*) – name of the package

Returns
state of package availability

Return type
bool

property accelerator_config

Return table with available accelerator configuration settings.

This is a wrapper around the *lammps_config_accelerator()* function of the library interface which loops over all known packages and categories and returns enabled features as a nested dictionary with all enabled settings as list of strings.

Returns
nested dictionary with all known enabled settings as list of strings

Return type
dictionary

property has_gpu_device

Availability of GPU package compatible device

This is a wrapper around the *lammps_has_gpu_device()* function of the C library interface.

Returns
True if a GPU package compatible device is present, otherwise False

Return type
bool

get_gpu_device_info()

Return a string with detailed information about any devices that are usable by the GPU package.

This is a wrapper around the *lammps_get_gpu_device_info()* function of the C-library interface.

Returns
GPU device info string

Return type
string

property installed_packages

List of the names of enabled packages in the LAMMPS shared library

This is a wrapper around the functions *lammps_config_package_count()* and :cpp:func`lammps_config_package_name` of the library interface.

:return

has_style(*category*, *name*)

Returns whether a given style name is available in a given category

This is a wrapper around the function *lammps_has_style()* of the library interface.

Parameters

- **category** (*string*) – name of category
- **name** (*string*) – name of the style

Returns
true if style is available in given category

Return type
bool

available_styles(*category*)

Returns a list of styles available for a given category

This is a wrapper around the functions *lammps_style_count()* and *lammps_style_name()* of the library interface.

Parameters
category (*string*) – name of category

Returns
list of style names in given category

Return type
list

has_id(*category*, *name*)

Returns whether a given ID name is available in a given category

Added in version 9Oct2020.

This is a wrapper around the function *lammps_has_id()* of the library interface.

Parameters

- **category** (*string*) – name of category
- **name** (*string*) – name of the ID

Returns
true if ID is available in given category

Return type
bool

available_ids(*category*)

Returns a list of IDs available for a given category

Added in version 9Oct2020.

This is a wrapper around the functions *lammps_id_count()* and *lammps_id_name()* of the library interface.

Parameters
category (*string*) – name of category

Returns
list of id names in given category

Return type
list

`available_plugins`(*category*)

Returns a list of plugins available for a given category

Added in version 10Mar2021.

This is a wrapper around the functions `lammps_plugin_count()` and `lammps_plugin_name()` of the library interface.

Returns
list of style/name pairs of loaded plugins

Return type
list

`set_fix_external_callback`(*fix_id*, *callback*, *caller=None*)

Set the callback function for a fix external instance with a given fix ID.

Optionally also set a reference to the calling object.

This is a wrapper around the *lammps_set_fix_external_callback()* function of the C-library interface. However this is set up to call a Python function with the following arguments.

- object is the value of the "caller" argument
- ntimestep is the current timestep
- nlocal is the number of local atoms on the current MPI process
- tag is a 1d NumPy array of integers representing the atom IDs of the local atoms
- x is a 2d NumPy array of doubles of the coordinates of the local atoms
- f is a 2d NumPy array of doubles of the forces on the local atoms that will be added

Changed in version 28Jul2021.

Parameters

- **fix_id** – Fix-ID of a fix external instance
- **callback** – Python function that will be called from fix external
- **caller** – reference to some object passed to the callback function

Type
string

Type
function

Type
object, optional

`fix_external_get_force`(*fix_id*)

Get access to the array with per-atom forces of a fix external instance with a given fix ID.

Added in version 28Jul2021.

This is a wrapper around the *lammps_fix_external_get_force()* function of the C-library interface.

Parameters
fix_id – Fix-ID of a fix external instance

Type
string

Returns
requested data

Return type
ctypes.POINTER(ctypes.POINTER(ctypes.double))

fix_external_set_energy_global(*fix_id*, *eng*)

Set the global energy contribution for a fix external instance with the given ID.

Added in version 28Jul2021.

This is a wrapper around the *lammps_fix_external_set_energy_global()* function of the C-library interface.

Parameters

- **fix_id** – Fix-ID of a fix external instance
- **eng** – potential energy value to be added by fix external

Type
string

Type
float

fix_external_set_virial_global(*fix_id*, *virial*)

Set the global virial contribution for a fix external instance with the given ID.

Added in version 28Jul2021.

This is a wrapper around the *lammps_fix_external_set_virial_global()* function of the C-library interface.

Parameters

- **fix_id** – Fix-ID of a fix external instance
- **eng** – list of 6 floating point numbers with the virial to be added by fix external

Type
string

Type
float

fix_external_set_energy_peratom(*fix_id*, *eatom*)

Set the per-atom energy contribution for a fix external instance with the given ID.

Added in version 28Jul2021.

This is a wrapper around the *lammps_fix_external_set_energy_peratom()* function of the C-library interface.

Parameters

- **fix_id** – Fix-ID of a fix external instance
- **eatom** – list of potential energy values for local atoms to be added by fix external

Type
string

Type
float

fix_external_set_virial_peratom(*fix_id*, *vatom*)

Set the per-atom virial contribution for a fix external instance with the given ID.

Added in version 28Jul2021.

This is a wrapper around the *lammps_fix_external_set_virial_peratom()* function of the C-library interface.

Parameters

- **fix_id** – Fix-ID of a fix external instance
- **vatom** – list of natoms lists with 6 floating point numbers to be added by fix external

Type
string

Type
float

fix_external_set_vector_length(*fix_id*, *length*)

Set the vector length for a global vector stored with fix external for analysis

Added in version 28Jul2021.

This is a wrapper around the *lammps_fix_external_set_vector_length()* function of the C-library interface.

Parameters

- **fix_id** – Fix-ID of a fix external instance
- **length** – length of the global vector

Type
string

Type
int

fix_external_set_vector(*fix_id*, *idx*, *val*)

Store a global vector value for a fix external instance with the given ID.

Added in version 28Jul2021.

This is a wrapper around the *lammps_fix_external_set_vector()* function of the C-library interface.

Parameters

- **fix_id** – Fix-ID of a fix external instance
- **idx** – 1-based index of the value in the global vector
- **val** – value to be stored in the global vector

Type
string

Type
int

Type
float

get_neighlist(*idx*)

Returns an instance of *NeighList* which wraps access to the neighbor list with the given index

See *lammps.numpy.get_neighlist()* if you want to use NumPy arrays instead of `c_int` pointers.

Parameters
idx (*int*) – index of neighbor list

Returns
an instance of *NeighList* wrapping access to neighbor list data

Return type
NeighList

get_neighlist_size(*idx*)

Return the number of elements in neighbor list with the given index

Parameters
idx (*int*) – neighbor list index

Returns
number of elements in neighbor list with index idx

Return type
int

get_neighlist_element_neighbors(*idx*, *element*)

Return data of neighbor list entry

Parameters

- **element** (*int*) – neighbor list index
- **element** – neighbor list element index

Returns
tuple with atom local index, number of neighbors and array of neighbor local atom indices

Return type
(int, int, POINTER(c_int))

find_pair_neighlist(*style*, *exact=True*, *nsub=0*, *reqid=0*)

Find neighbor list index of pair style neighbor list

Search for a neighbor list requested by a pair style instance that matches "style". If exact is True, the pair style name must match exactly. If exact is False, the pair style name is matched against "style" as regular expression or sub-string. If the pair style is a hybrid pair style, the style is instead matched against the hybrid sub-styles. If the same pair style is used as sub-style multiple types, you must set nsub to a value n > 0 which indicates the nth instance of that sub-style to be used (same as for the pair_coeff command). The default value of 0 will fail to match in that case.

Once the pair style instance has been identified, it may have requested multiple neighbor lists. Those are uniquely identified by a request ID > 0 as set by the pair style. Otherwise the request ID is 0.

Parameters

- **style** (*string*) – name of pair style that should be searched for
- **exact** (*bool, optional*) – controls whether style should match exactly or only must be contained in pair style name, defaults to True
- **nsub** (*int, optional*) – match nsub-th hybrid sub-style, defaults to 0

- **reqid** (*int, optional*) – list request id, > 0 in case there are more than one, defaults to 0

Returns
neighbor list index if found, otherwise -1

Return type
int

find_fix_neighlist(*fixid*, *reqid=0*)

Find neighbor list index of fix neighbor list

The fix instance requesting the neighbor list is uniquely identified by the fix ID. In case the fix has requested multiple neighbor lists, those are uniquely identified by a request ID > 0 as set by the fix. Otherwise the request ID is 0 (the default).

Parameters

- **fixid** (*string*) – name of fix
- **reqid** (*int, optional*) – id of neighbor list request, in case there are more than one request, defaults to 0

Returns
neighbor list index if found, otherwise -1

Return type
int

find_compute_neighlist(*computeid*, *reqid=0*)

Find neighbor list index of compute neighbor list

The compute instance requesting the neighbor list is uniquely identified by the compute ID. In case the compute has requested multiple neighbor lists, those are uniquely identified by a request ID > 0 as set by the compute. Otherwise the request ID is 0 (the default).

Parameters

- **computeid** (*string*) – name of compute
- **reqid** (*int, optional*) – index of neighbor list request, in case there are more than one request, defaults to 0

Returns
neighbor list index if found, otherwise -1

Return type
int

class lammps.numpy_wrapper.**numpy_wrapper**(*lmp*)

lammps API NumPy Wrapper

This is a wrapper class that provides additional methods on top of an existing *lammps* instance. The methods transform raw ctypes pointers into NumPy arrays, which give direct access to the original data while protecting against out-of-bounds accesses.

There is no need to explicitly instantiate this class. Each instance of *lammps* has a **numpy** property that returns an instance.

Parameters
lmp (lammps) – instance of the *lammps* class

extract_atom(*name*, *dtype=None*, *nelem=None*, *dim=None*)

Retrieve per-atom properties from LAMMPS as NumPy arrays

This is a wrapper around the `lammps.extract_atom()` method. It behaves the same as the original method, but returns NumPy arrays instead of `ctypes` pointers.

Note: The returned arrays of per-atom data are by default dimensioned for the range [0:nlocal] since that data is *always* valid. The underlying storage for the data, however, is typically allocated for the range of [0:nmax]. Whether there is valid data in the range [nlocal:nlocal+nghost] depends on whether the property of interest is also updated for ghost atoms. This is not often the case. In some cases, it depends on a LAMMPS setting, see for example *comm_modify vel yes*. By using the optional *nelem* parameter the size of the returned NumPy can be overridden. There is no check whether the number of elements chosen is valid.

Parameters

- **name** (*string*) – name of the property
- **dtype** (*int, optional*) – type of the returned data (see *Data Types*)
- **nelem** (*int, optional*) – number of elements in array
- **dim** (*int, optional*) – dimension of each element

Returns
requested data as NumPy array with direct access to C data or None

Return type
numpy.array or NoneType

extract_compute(*cid*, *cstyle*, *ctype*)

Retrieve data from a LAMMPS compute

This is a wrapper around the *lammps.extract_compute()* method. It behaves the same as the original method, but returns NumPy arrays instead of `ctypes` pointers.

Parameters

- **cid** (*string*) – compute ID
- **cstyle** (*int*) – style of the data retrieve (global, atom, or local), see *Style Constants*
- **ctype** (*int*) – type of the returned data (scalar, vector, or array), see *Type Constants*

Returns
requested data either as float, as NumPy array with direct access to C data, or None

Return type
float, numpy.array, or NoneType

extract_fix(*fid*, *fstyle*, *ftype*, *nrow=0*, *ncol=0*)

Retrieve data from a LAMMPS fix

This is a wrapper around the *lammps.extract_fix()* method. It behaves the same as the original method, but returns NumPy arrays instead of `ctypes` pointers.

Note: When requesting global data, the fix data can only be accessed one item at a time without access to the whole vector or array. Thus this function will always return a scalar. To access vector or array elements

the "nrow" and "ncol" arguments need to be set accordingly (they default to 0).

Parameters

- **fid** (*string*) – fix ID
- **fstyle** (*int*) – style of the data retrieve (global, atom, or local), see *Style Constants*
- **ftype** (*int*) – type or size of the returned data (scalar, vector, or array), see *Type Constants*
- **nrow** (*int*) – index of global vector element or row index of global array element
- **ncol** (*int*) – column index of global array element

Returns
requested data

Return type
integer or double value, pointer to 1d or 2d double array or None

`extract_variable`(*name*, *group=None*, *vartype=0*)

Evaluate a LAMMPS variable and return its data

This function is a wrapper around the function *lammps.extract_variable()* method. It behaves the same as the original method, but returns NumPy arrays instead of `ctypes` pointers.

Parameters

- **name** (*string*) – name of the variable to execute
- **group** (*string*) – name of group for atom-style variable (ignored for equal-style variables)
- **vartype** (*int*) – type of variable, see *Variable Type Constants*

Returns
the requested data or None

Return type
c_double, numpy.array, or NoneType

`gather_bonds`()

Retrieve global list of bonds as NumPy array

Added in version 28Jul2021.

This is a wrapper around *lammps.gather_bonds()* It behaves the same as the original method, but returns a NumPy array instead of a `ctypes` list.

Returns
the requested data as a 2d-integer numpy array

Return type
numpy.array(nbonds,3)

`gather_angles`()

Retrieve global list of angles as NumPy array

Added in version 8Feb2023.

This is a wrapper around *lammps.gather_angles()* It behaves the same as the original method, but returns a NumPy array instead of a `ctypes` list.

Returns
the requested data as a 2d-integer numpy array

Return type
numpy.array(nangles,4)

`gather_dihedrals`()

Retrieve global list of dihedrals as NumPy array

Added in version 8Feb2023.

This is a wrapper around *lammps.gather_dihedrals()* It behaves the same as the original method, but returns a NumPy array instead of a `ctypes` list.

Returns
the requested data as a 2d-integer numpy array

Return type
numpy.array(ndihedrals,5)

`gather_impropers`()

Retrieve global list of impropers as NumPy array

Added in version 8Feb2023.

This is a wrapper around *lammps.gather_impropers()* It behaves the same as the original method, but returns a NumPy array instead of a `ctypes` list.

Returns
the requested data as a 2d-integer numpy array

Return type
numpy.array(nimpropers,5)

`fix_external_get_force`(*fix_id*)

Get access to the array with per-atom forces of a fix external instance with a given fix ID.

Changed in version 28Jul2021.

This function is a wrapper around the *lammps.fix_external_get_force()* method. It behaves the same as the original method, but returns a NumPy array instead of a `ctypes` pointer.

Parameters
fix_id – Fix-ID of a fix external instance

Type
string

Returns
requested data

Return type
numpy.array

`fix_external_set_energy_peratom`(*fix_id*, *eatom*)

Set the per-atom energy contribution for a fix external instance with the given ID.

Added in version 28Jul2021.

This function is an alternative to *lammps.fix_external_set_energy_peratom()* method. It behaves the same as the original method, but accepts a NumPy array instead of a list as argument.

Parameters

- **fix_id** – Fix-ID of a fix external instance
- **eatom** – per-atom potential energy

Type
string

Type
numpy.array

`fix_external_set_virial_peratom`(*fix_id*, *vatom*)

Set the per-atom virial contribution for a fix external instance with the given ID.

Added in version 28Jul2021.

This function is an alternative to *lammps.fix_external_set_virial_peratom()* method. It behaves the same as the original method, but accepts a NumPy array instead of a list as argument.

Parameters

- **fix_id** – Fix-ID of a fix external instance
- **eatom** – per-atom potential energy

Type
string

Type
numpy.array

`get_neighlist`(*idx*)

Returns an instance of *NumPyNeighList* which wraps access to the neighbor list with the given index

Parameters
idx (*int*) – index of neighbor list

Returns
an instance of *NumPyNeighList* wrapping access to neighbor list data

Return type
NumPyNeighList

`get_neighlist_element_neighbors`(*idx*, *element*)

Return data of neighbor list entry

This function is a wrapper around the function *lammps.get_neighlist_element_neighbors()* method. It behaves the same as the original method, but returns a NumPy array containing the neighbors instead of a `ctypes` pointer.

Parameters

- **element** (*int*) – neighbor list index
- **element** – neighbor list element index

Returns
tuple with atom local index and numpy array of neighbor local atom indices

Return type
(int, numpy.array)

2.4.2 The PyLammps class API

The *PyLammps* class is a wrapper that creates a simpler, more "Pythonic" interface to common LAMMPS functionality. LAMMPS data structures are exposed through objects and properties. This makes Python scripts shorter and more concise. See the *PyLammps Tutorial* for an introduction on how to use this interface.

class lammps.**PyLammps**(*name=''*, *cmdargs=None*, *ptr=None*, *comm=None*, *verbose=False*)

This is a Python wrapper class around the lower-level *lammps* class, exposing a more Python-like, object-oriented interface for prototyping system inside of IPython and Jupyter notebooks.

It either creates its own instance of *lammps* or can be initialized with an existing instance. The arguments are the same of the lower-level interface. The original interface can still be accessed via `PyLammps.lmp`.

Parameters

- **name** (*string*) – "machine" name of the shared LAMMPS library ("mpi" loads `liblammps_mpi.so`, "" loads `liblammps.so`)
- **cmdargs** (*list*) – list of command line arguments to be passed to the *lammps_open()* function. The executable name is automatically added.
- **ptr** (*pointer*) – pointer to a LAMMPS C++ class instance when called from an embedded Python interpreter. None means load symbols from shared library.
- **comm** (*MPI_Comm*) – MPI communicator (as provided by mpi4py). `None` means use `MPI_COMM_WORLD` implicitly.
- **verbose** (*bool*) – print all LAMMPS output to stdout

Variables

- **lmp** (*lammps*) – instance of original LAMMPS Python interface
- **runs** – list of completed runs, each storing the thermo output

close()

Explicitly delete a LAMMPS instance

This is a wrapper around the *lammps.close()* of the Python interface.

version()

Return a numerical representation of the LAMMPS version in use.

This is a wrapper around the *lammps.version()* function of the Python interface.

Returns
version number

Return type
int

file(*file*)

Read LAMMPS commands from a file.

This is a wrapper around the *lammps.file()* function of the Python interface.

Parameters
path (*string*) – Name of the file/path with LAMMPS commands

property enable_cmd_history

Getter
Return whether command history is saved

Setter
Set if command history should be saved

Type
bool

write_script(*filepath*)

Write LAMMPS script file containing all commands executed up until now

Parameters
filepath (*string*) – path to script file that should be written

clear_cmd_history()

Clear LAMMPS command history up to this point

append_cmd_history(*cmd*)

Commands will be added to the command history but not executed.

Add *commands* only to the command history, but do not execute them, so that you can conveniently create Lammps input files, using *PyLammps.write_script()*.

command(*cmd*)

Execute LAMMPS command

If *PyLammps.enable_cmd_history* is set to True, commands executed will be recorded. The entire command history can be written to a file using *PyLammps.write_script()*. To clear the command history, use *PyLammps.clear_cmd_history()*.

Parameters
cmd – command string that should be executed

Type
cmd: string

run(**args*, ***kwargs*)

Execute LAMMPS run command with given arguments

Thermo data of the run is recorded and saved as new entry in PyLammps.runs. The latest run can be retrieved by *PyLammps.last_run*.

Note, for recording of all thermo steps during a run, the PYTHON package needs to be enabled in LAMMPS. Otherwise, it will only capture the final timestep.

property last_run

Return data produced of last completed run command

Getter
Returns an object containing information about the last run command

Type
dict

property atoms

All atoms of this LAMMPS instance

Getter
Returns a list of atoms currently in the system

Type
AtomList

property system

The system state of this LAMMPS instance

Getter
Returns an object with properties storing the current system state

Type
namedtuple

property communication

The communication state of this LAMMPS instance

Getter
Returns an object with properties storing the current communication state

Type
namedtuple

property computes

The list of active computes of this LAMMPS instance

Getter
Returns a list of computes that are currently active in this LAMMPS instance

Type
list

property dumps

The list of active dumps of this LAMMPS instance

Getter
Returns a list of dumps that are currently active in this LAMMPS instance

Type
list

property fixes

The list of active fixes of this LAMMPS instance

Getter
Returns a list of fixes that are currently active in this LAMMPS instance

Type
list

property groups

The list of active atom groups of this LAMMPS instance

Getter
Returns a list of atom groups that are currently active in this LAMMPS instance

Type
list

property variables

Returns a dictionary of all variables defined in the current LAMMPS instance

Getter
Returns a dictionary of all variables that are defined in this LAMMPS instance

Type
dict

eval(*expr*)

Evaluate expression

Parameters
expr (*string*) – the expression string that should be evaluated inside of LAMMPS

Returns
the value of the evaluated expression

Return type
float if numeric, string otherwise

lmp_print(*s*)

needed for Python2 compatibility, since print is a reserved keyword

class lammps.**AtomList**(*pylammps_instance*)

A dynamic list of atoms that returns either an *Atom* or *Atom2D* instance for each atom. Instances are only allocated when accessed.

Variables

- **natoms** – total number of atoms
- **dimensions** – number of dimensions in system

class lammps.**Atom**(*pylammps_instance*, *index*)

A wrapper class then represents a single atom inside of LAMMPS

It provides access to properties of the atom and allows you to change some of them.

property id

Return the atom ID

Type
int

property type

Return the atom type

Type
int

property mol

Return the atom molecule index

Type
int

property mass

Return the atom mass as a per-atom property. This returns either the per-type mass or the per-atom mass (AKA 'rmass') depending on what is available with preference for the per-atom mass.

Changed in version 17Apr2024: Support both per-type and per-atom masses. With per-type return "mass[type[i]]" else return "rmass[i]". Per-atom mass is preferred if available.

Type
float

property radius

Return the particle radius

Type
float

property position

Getter
Return position of atom

Setter
Set position of atom

Type
numpy.array (float, float, float)

property velocity

Getter
Return velocity of atom

Setter
Set velocity of atom

Type
numpy.array (float, float, float)

property force

Return the total force acting on the atom

Type
numpy.array (float, float, float)

property omega

Return the rotational velocity of the particle

Type
numpy.array (float, float, float)

property torque

Return the total torque acting on the particle

Type
numpy.array (float, float, float)

property angular_momentum

Return the angular momentum of the particle

Type
numpy.array (float, float, float)

property charge

Return the atom charge

Type
float

class lammps.**Atom2D**(*pylammps_instance*, *index*)

A wrapper class then represents a single 2D atom inside of LAMMPS

Inherits all properties from the *Atom* class, but returns 2D versions of position, velocity, and force.

It provides access to properties of the atom and allows you to change some of them.

property position

Access to coordinates of an atom

Getter
Return position of atom

Setter
Set position of atom

Type
numpy.array (float, float)

property velocity

Access to velocity of an atom :getter: Return velocity of atom :setter: Set velocity of atom :type: numpy.array (float, float)

property force

Access to force of an atom :getter: Return force of atom :setter: Set force of atom :type: numpy.array (float, float)

2.4.3 The IPyLammps class API

The *IPyLammps* class is an extension of *PyLammps*, adding additional functions to quickly display visualizations such as images and videos inside of IPython. See the *PyLammps Tutorial* for examples.

class lammps.**IPyLammps**(*name=''*, *cmdargs=None*, *ptr=None*, *comm=None*)

IPython wrapper for LAMMPS which adds embedded graphics capabilities to PyLammmps interface

It either creates its own instance of *lammps* or can be initialized with an existing instance. The arguments are the same of the lower-level interface. The original interface can still be accessed via `PyLammps.lmp`.

Parameters

- **name** (*string*) – "machine" name of the shared LAMMPS library ("mpi" loads `liblammps_mpi.so`, "" loads `liblammps.so`)
- **cmdargs** (*list*) – list of command line arguments to be passed to the *lammps_open()* function. The executable name is automatically added.
- **ptr** (*pointer*) – pointer to a LAMMPS C++ class instance when called from an embedded Python interpreter. None means load symbols from shared library.
- **comm** (*MPI_Comm*) – MPI communicator (as provided by mpi4py). `None` means use `MPI_COMM_WORLD` implicitly.

image(*filename='snapshot.png'*, *group='all'*, *color='type'*, *diameter='type'*, *size=None*, *view=None*, *center=None*, *up=None*, *zoom=1.0*, *background_color='white'*)

Generate image using write_dump command and display it

See *dump image* for more information.

Parameters

- **filename** (*string*) – Name of the image file that should be generated. The extension determines whether it is PNG or JPEG
- **group** (*string*) – the group of atoms write_image should use
- **color** (*string*) – name of property used to determine color
- **diameter** (*string*) – name of property used to determine atom diameter
- **size** (*tuple (width, height)*) – dimensions of image

- **view** (*tuple (theta, phi)*) – view parameters
- **center** (*tuple (flag, center_x, center_y, center_z)*) – center parameters
- **up** (*tuple (up_x, up_y, up_z)*) – vector pointing to up direction
- **zoom** (*float*) – zoom factor
- **background_color** (*string*) – background color of scene

Returns
Image instance used to display image in notebook

Return type
IPython.core.display.Image

video(*filename*)

Load video from file

Can be used to visualize videos from *dump movie*.

Parameters
filename (*string*) – Path to video file

Returns
HTML Video Tag used by notebook to embed a video

Return type
IPython.display.HTML

2.4.4 Additional components of the lammps module

The *lammps* module additionally contains several constants and the *NeighList* class:

Data Types

LAMMPS_INT, LAMMPS_INT_2D, LAMMPS_DOUBLE, LAMMPS_DOUBLE_2D, LAMMPS_INT64, LAMMPS_INT64_2D, LAMMPS_STRING

Constants in the *lammps* module to indicate how to cast data when the C library function returns a void pointer. Used in *lammps.extract_global()* and *lammps.extract_atom()*. See _LMP_DATATYPE_CONST for the equivalent constants in the C library interface.

Style Constants

LMP_STYLE_GLOBAL, LMP_STYLE_ATOM, LMP_STYLE_LOCAL

Constants in the *lammps* module to select what style of data to request from computes or fixes. See *_LMP_STYLE_CONST* for the equivalent constants in the C library interface. Used in *lammps.extract_compute()*, *lammps.extract_fix()*, and their NumPy variants *lammps.numpy.extract_compute()* and *lammps.numpy.extract_fix()*.

Type Constants

LMP_TYPE_SCALAR, LMP_TYPE_VECTOR, LMP_TYPE_ARRAY, LMP_SIZE_VECTOR, LMP_SIZE_ROWS, LMP_SIZE_COLS

Constants in the *lammps* module to select what type of data to request from computes or fixes. See *_LMP_TYPE_CONST* for the equivalent constants in the C library interface. Used in *lammps.extract_compute()*, *lammps.extract_fix()*, and their NumPy variants *lammps.numpy.extract_compute()* and *lammps.numpy.extract_fix()*.

Variable Type Constants

LMP_VAR_EQUAL, LMP_VAR_ATOM

Constants in the *lammps* module to select what type of variable to query when calling *lammps.extract_variable()*. See also: *variable command*.

Classes representing internal objects

class lammps.**NeighList**(*lmp*, *idx*)

This is a wrapper class that exposes the contents of a neighbor list.

It can be used like a regular Python list. Each element is a tuple of:

- the atom local index
- its number of neighbors
- and a pointer to an c_int array containing local atom indices of its neighbors

Internally it uses the lower-level LAMMPS C-library interface.

Parameters

- **lmp** (lammps) – reference to instance of *lammps*
- **idx** (*int*) – neighbor list index

property size

Returns
number of elements in neighbor list

get(*element*)

Access a specific neighbor list entry. “element” must be a number from 0 to the size-1 of the list

Returns
tuple with atom local index, number of neighbors and ctypes pointer to neighbor’s local atom indices

Return type
(int, int, ctypes.POINTER(c_int))

find(*iatom*)

Find the neighbor list for a specific (local) atom iatom. If there is no list for iatom, (-1, None) is returned.

Returns
tuple with number of neighbors and ctypes pointer to neighbor’s local atom indices

Return type
(int, ctypes.POINTER(c_int))

class lammps.numpy_wrapper.**NumPyNeighList**(*lmp*, *idx*)

This is a wrapper class that exposes the contents of a neighbor list.

It can be used like a regular Python list. Each element is a tuple of:

- the atom local index
- a NumPy array containing the local atom indices of its neighbors

Internally it uses the lower-level LAMMPS C-library interface.

Parameters

- **lmp** (lammps) – reference to instance of *lammps*
- **idx** (*int*) – neighbor list index

get(*element*)

Access a specific neighbor list entry. "element" must be a number from 0 to the size-1 of the list

Returns
tuple with atom local index, numpy array of neighbor local atom indices

Return type
(int, numpy.array)

find(*iatom*)

Find the neighbor list for a specific (local) atom iatom. If there is no list for iatom, None is returned.

Returns
numpy array of neighbor local atom indices

Return type
numpy.array or None

2.5 Extending the Python interface

As noted previously, most of the *lammps* Python class methods correspond one-to-one with the functions in the LAMMPS library interface in `src/library.cpp` and `library.h`. This means you can extend the Python wrapper by following these steps:

- Add a new interface function to `src/library.cpp` and `src/library.h`.
- Rebuild LAMMPS as a shared library.
- Add a wrapper method to `python/lammps/core.py` for this interface function.
- Define the corresponding `argtypes` list and `restype` in the `lammps.__init__()` function.
- Re-install the shared library and the python module, if needed
- You should now be able to invoke the new interface function from a Python script.

2.6 Calling Python from LAMMPS

LAMMPS has several commands which can be used to invoke Python code directly from an input script:

- *python*
- *variable python*
- *fix python/invoke*
- *pair_style python*

The *python* command which can be used to define and execute a Python function that you write the code for. The Python function can also be assigned to a LAMMPS python-style variable via the *variable* command. Each time the variable is evaluated, either in the LAMMPS input script itself, or by another LAMMPS command that uses the variable, this will trigger the Python function to be invoked.

The Python code for the function can be included directly in the input script or in an auxiliary file. The function can have arguments which are mapped to LAMMPS variables (also defined in the input script) and it can return a value to a LAMMPS variable. This is thus a mechanism for your input script to pass information to a piece of Python code, ask Python to execute the code, and return information to your input script.

Note that a Python function can be arbitrarily complex. It can import other Python modules, instantiate Python classes, call other Python functions, etc. The Python code that you provide can contain more code than the single function. It can contain other functions or Python classes, as well as global variables or other mechanisms for storing state between calls from LAMMPS to the function.

The Python function you provide can consist of "pure" Python code that only performs operations provided by standard Python. However, the Python function can also "call back" to LAMMPS through its Python-wrapped library interface, in the manner described in the *Python run* doc page. This means it can issue LAMMPS input script commands or query and set internal LAMMPS state. As an example, this can be useful in an input script to create a more complex loop with branching logic, than can be created using the simple looping and branching logic enabled by the *next* and *if* commands.

See the *python* page and the *variable* doc page for its python-style variables for more info, including examples of Python code you can write for both pure Python operations and callbacks to LAMMPS.

2.7 Output Readers

The Python package contains the `lammps.formats` module, which provides classes to post-process some of the output files generated by LAMMPS.

class lammps.formats.**LogFile**(*filename*)

Reads LAMMPS log files and extracts the thermo information

It supports the line, multi, and yaml thermo output styles.

Parameters
filename (*str*) – path to log file

Variables

- **runs** – List of LAMMPS runs in log file. Each run is a dictionary with thermo fields as keys, storing the values over time
- **errors** – List of error lines in log file

class lammps.formats.**AvgChunkFile**(*filename*)

Reads files generated by fix ave/chunk

Parameters
filename (*str*) – path to ave/chunk file

Variables

- **timesteps** – List of timesteps stored in file
- **total_count** – total count over time
- **chunks** – List of chunks. Each chunk is a dictionary containing its ID, the coordinates, and the averaged quantities

2.8 Example Python scripts

The `python/examples` directory has Python scripts which show how Python can run LAMMPS, grab data, change it, and put it back into LAMMPS.

These are the Python scripts included as demos in the `python/examples` directory of the LAMMPS distribution, to illustrate the kinds of things that are possible when Python wraps LAMMPS. If you create your own scripts, send them to us and we can include them in the LAMMPS distribution.

`trivial.py`	read/run a LAMMPS input script through Python
`demo.py`	invoke various LAMMPS library interface routines
`simple.py`	run in parallel, similar to `examples/COUPLE/simple/simple.cpp`
`split.py`	same as `simple.py` but running in parallel on a subset of procs
`gui.py`	GUI go/stop/temperature-slider to control LAMMPS
`plot.py`	real-time temperature plot with GnuPlot via Pizza.py
`viz_TOOL.py`	real-time viz via some viz package
`vizplotgui_TOOL.py`	combination of `viz_TOOL.py` and `plot.py` and `gui.py`

For the `viz_TOOL.py` and `vizplotgui_TOOL.py` commands, replace `TOOL` with `gl` or `atomeye` or `pymol` or `vmd`, depending on what visualization package you have installed.

Note that for GL, you need to be able to run the Pizza.py GL tool, which is included in the pizza subdirectory. See the Pizza.py doc pages for more info:

- https://lammps.github.io/pizza

Note that for AtomEye, you need version 3, and there is a line in the scripts that specifies the path and name of the executable. See the AtomEye web pages for more details:

- http://li.mit.edu/Archive/Graphics/A/
- http://li.mit.edu/Archive/Graphics/A3/A3.html

The latter link is to AtomEye 3 which has the scripting capability needed by these Python scripts.

Note that for PyMol, you need to have built and installed the open-source version of PyMol in your Python, so that you can import it from a Python script. See the PyMol web pages for more details:

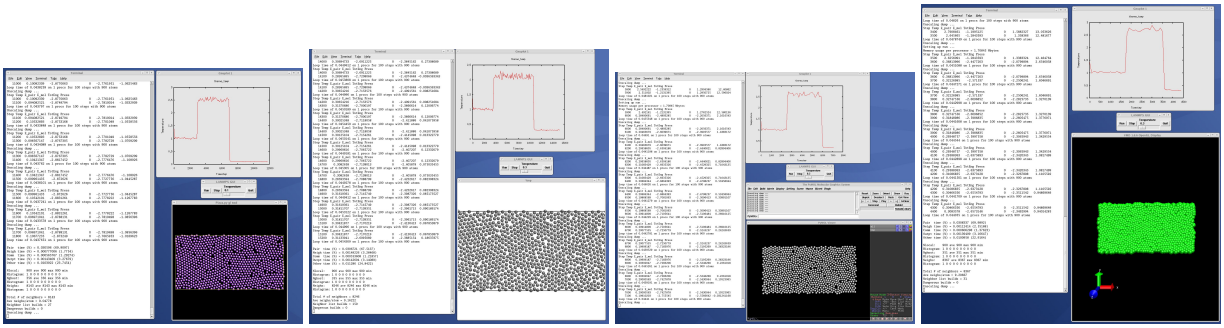
- https://www.pymol.org
- https://github.com/schrodinger/pymol-open-source

The latter link is to the open-source version.

Note that for VMD, you need a fairly current version (1.8.7 works for me) and there are some lines in the `pizza/vmd.py` script for 4 PIZZA variables that have to match the VMD installation on your system.

See the `python/README` file for instructions on how to run them and the source code for individual scripts for comments about what they do.

Here are screenshots of the `vizplotgui_tool.py` script in action for different visualization package options:



2.9 Handling LAMMPS errors

LAMMPS and the LAMMPS library are compiled with *C++ exception support* to provide a better error handling experience. LAMMPS errors trigger throwing a C++ exception. These exceptions allow capturing errors on the C++ side and rethrowing them on the Python side. This way LAMMPS errors can be handled through the Python exception handling mechanism.

```
from lammps import lammps, MPIAbortException

lmp = lammps()

try:
   # LAMMPS will normally terminate itself and the running process if an error
   # occurs. This would kill the Python interpreter.  The library wrapper will
   # detect that an error has occured and throw a Python exception

   lmp.command('unknown')
except MPIAbortException as ae:
   # Single MPI process got killed. This would normally be handled by an MPI abort
   pass
except Exception as e:
   # All (MPI) processes have reached this error
   pass
```

Warning: Capturing a LAMMPS exception in Python can still mean that the current LAMMPS process is in an illegal state and must be terminated. It is advised to save your data and terminate the Python instance as quickly as possible when running in parallel with MPI.

2.10 Troubleshooting

2.10.1 Testing if Python can launch LAMMPS

To test if LAMMPS is callable from Python, launch Python interactively and type:

```
>>> from lammps import lammps
>>> lmp = lammps()
```

If you get no errors, you're ready to use LAMMPS from Python. If the second command fails, the most common error to see is

```
OSError: Could not load LAMMPS dynamic library
```

which means Python was unable to load the LAMMPS shared library. This typically occurs if the system can't find the LAMMPS shared library or one of the auxiliary shared libraries it depends on, or if something about the library is incompatible with your Python. The error message should give you an indication of what went wrong.

If your shared library uses a suffix, such as `liblammps_mpi.so`, change the constructor call as follows (see *Creating or deleting a LAMMPS object* for more details):

```
>>> lmp = lammps(name='mpi')
```

You can also test the load directly in Python as follows, without first importing from the `lammps` module:

```
>>> from ctypes import CDLL
>>> CDLL("liblammps.so")
```

If an error occurs, carefully go through the steps in *Installing the LAMMPS Python Module and Shared Library* and on the *Build_basics* page about building a shared library.

If you are not familiar with Python, it is a powerful scripting and programming language which can do almost everything that compiled languages like C, C++, or Fortran can do in fewer lines of code. It also comes with a large collection of add-on modules for many purposes (either bundled or easily installed from Python code repositories). The major drawback is slower execution speed of the script code compared to compiled programming languages. But when the script code is interfaced to optimized compiled code, performance can be on par with a standalone executable, for as long as the scripting is restricted to high-level operations. Thus Python is also convenient to use as a "glue" language to "drive" a program through its library interface, or to hook multiple pieces of software together, such as a simulation code and a visualization tool, or to run a coupled multi-scale or multi-physics model.

See the *Coupling LAMMPS to other codes* page for more ideas about coupling LAMMPS to other codes. See the *LAMMPS Library Interfaces* page for a description of the LAMMPS library interfaces. That interface is exposed to Python either when calling LAMMPS from Python or when calling Python from a LAMMPS input script and then calling back to LAMMPS from Python code. The C-library interface is designed to be easy to add functionality to, thus the Python interface to LAMMPS is easy to extend as well.

If you create interesting Python scripts that run LAMMPS or interesting Python functions that can be called from a LAMMPS input script, that you think would be generally useful, please post them as a pull request to our GitHub site, and they can be added to the LAMMPS distribution or web page.