LAMMPS-GUI Documentation

Release v3.1.2

Axel Kohlmeyer

akohlmey@gmail.com

LAMMPS-GUI Documentation

1 About LAMMPS-GUI **2**

2 About this document **3**

3 Citing LAMMPS-GUI **3**

4 User's Guide **5**
- 4.1 Installation . . . 5
- 4.2 Overview . . . 12
- 4.3 Basic usage of LAMMPS-GUI . . . 13
- 4.4 Monitoring LAMMPS output . . . 17
- 4.5 Visualization . . . 31
- 4.6 Editor Window . . . 47
- 4.7 Menus . . . 50
- 4.8 Dialogs . . . 55
- 4.9 Keyboard Shortcuts . . . 58

5 Programmer's Guide **60**
- 5.1 Overview . . . 60
- 5.2 Architecture . . . 61
- 5.3 API Reference . . . 65
- 5.4 Development Guidelines . . . 201
- 5.5 Testing . . . 204

Index **218**

1 About LAMMPS-GUI

LAMMPS-GUI is a graphical text editor with syntax highlighting, auto-completion, inline help, and indentation support for LAMMPS input files. It is programmed using the Qt Framework and customized for running, monitoring, and visualizing LAMMPS simulations. It calls LAMMPS directly using the LAMMPS library interface instead of launching an external LAMMPS executable. Therefore it can retrieve and display information from LAMMPS *while it is running* and *immediately* display visualizations created by a dump image command in the input.

The primary motivation for implementing LAMMPS-GUI is to facilitate teaching LAMMPS to beginners using only LAMMPS-GUI and to have a consistent behavior across major platforms like Linux, macOS, and Windows. This way one can focus on teaching LAMMPS and avoid having to spend time explaining different tools (for editing inputs, plotting graphs, visualizing systems) on the different platforms. Also, LAMMPS-GUI is fully integrated with a collection of LAMMPS tutorials.

Many of the features in LAMMPS-GUI are useful beyond working on tutorials. For instance, it can streamline the process of prototyping new simulation projects or debugging misbehaving simulations. It also has been found extremely useful in creating, debugging, and tweaking advanced visualizations with LAMMPS using the built-in visualization facility of the dump image command.

LAMMPS-GUI is Copyright (c) 2023 - 2026 by Axel Kohlmeyer, and its source code is distributed under the terms of the GNU General Public License v2.0 or later.

Custom rangeslider widget for Qt written by Hoyoung Lee and distributed under the CeCILL Free Software License.

Lepton library written by Peter Eastman and OpenMM contributors distributed under the MIT License.

The LAMMPS-GUI documentation is available under the Creative Commons Attribution 4.0 International License.

---

2 About this document

This document contains the documentation of LAMMPS-GUI and how to compile, install, use, configure, and modify it. Suggestions for new features and reports of bugs are always welcome. You can use the same channels as for LAMMPS itself for that purpose or submit bug reports or pull requests in the LAMMPS-GUI GitHub repository.

This document describes LAMMPS-GUI version 3.1.2.

---

3 Citing LAMMPS-GUI

If you use LAMMPS-GUI in your work, please cite the following paper in the Journal of Open Source Software:

> Kohlmeyer, A. (2026). LAMMPS-GUI: A Cross-Platform Graphical Tool to Learn and Explore Molecular Dynamics with LAMMPS. Journal of Open Source Software, 11(126), 11185. https://joss.theoj.org/papers/10.21105/joss.11185

or in BibTeX format:

```
@article{lammps_gui_joss,
  author       = {Kohlmeyer, Axel},
  title        = {{LAMMPS-GUI}: A Cross-Platform Graphical Tool to
                  Learn and Explore Molecular Dynamics with {LAMMPS}},
  journal      = {Journal of Open Source Software},
  publisher    = {The Open Journal},
  year         = {2026},
  volume       = {11},
  number       = {126},
  pages        = {11185},
  doi          = {10.21105/joss.11185},
  url          = {https://joss.theoj.org/papers/10.21105/joss.11185},
}
```

Also, starting with version 3.0.0 LAMMPS-GUI releases are automatically archived on Zenodo:

```
@software{lammps_gui_zenodo,
  author       = {Kohlmeyer, Axel},
  title        = {{LAMMPS-GUI}: A Cross-Platform Graphical Tool to
                  Learn and Explore Molecular Dynamics with {LAMMPS}},
  publisher    = {Zenodo},
  doi          = {10.5281/zenodo.21035505},
  url          = {https://doi.org/10.5281/zenodo.21035505},
}
```

The *Soft Matter* collection of tutorials in the *Tutorials menu* is published in LiveCoMS. Those tutorials teach how to use LAMMPS-GUI as much as how to use LAMMPS, so this publication is a suitable secondary citation:

Gravelle, S., Alvares, C. M. S., Gissinger, J. R., & Kohlmeyer, A. (2025). A Set of Tutorials for the LAMMPS Simulation Package [Article v1.0]. Living Journal of Computational Molecular Science, 6(1), 3037. https://doi.org/10.33011/livecoms.6.1.3037

or in BibTeX format:

```bibtex
@article{lammps_tutorials_2025,
  author={Gravelle, Simon and Alvares, Cecilia M. S. and Gissinger, Jacob R. and
↪Kohlmeyer, Axel},
  title={A Set of Tutorials for the {LAMMPS} Simulation Package [Article v1.0]},
  journal={Living Journal of Computational Molecular Science},
  pages={3037},
  volume={6},
  number={1},
  year={2025},
  month={Sep.},
  url={https://livecomsjournal.org/index.php/livecoms/article/view/v6i1e3037},
  DOI={10.33011/livecoms.6.1.3037}
}
```

4 User's Guide

4.1 Installation

LAMMPS-GUI is distributed as source code on GitHub and can be compiled as part of compiling LAMMPS, where it will be linked to the corresponding version of LAMMPS directly. Pre-compiled packages of LAMMPS with LAMMPS-GUI included are available for download (see below).

LAMMPS-GUI can also be compiled as a standalone package that loads the LAMMPS library dynamically at runtime. This enables using LAMMPS-GUI with customized, patched, or extended LAMMPS versions containing features not available in the official LAMMPS distribution packages. It also allows using LAMMPS-GUI with LAMMPS shared libraries compiled using the traditional makefile based build process (which does not support compiling LAMMPS-GUI directly). Pre-compiled packages of standalone LAMMPS-GUI versions with a LAMMPS shared library included are also available for download (see below).

Prerequisites and portability

LAMMPS-GUI is programmed in C++ based on the C++17 standard and using the Qt GUI framework. As of LAMMPS-GUI version 2.0.0 Qt version 6.2 or later is required. LAMMPS-GUI can switch between a "light" and a "dark" theme according to the settings of the desktop environment. Building LAMMPS-GUI from source requires CMake version 3.20 or later.

LAMMPS-GUI 3.1.2 has been successfully compiled and tested on

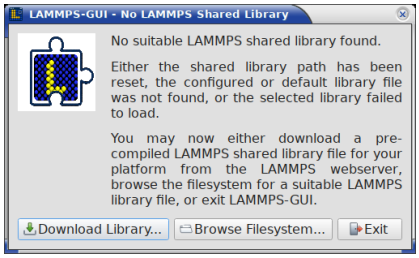
- Ubuntu Linux 22.04LTS x86_64 using GCC 11 and Qt version 6.2
- Ubuntu Linux 24.04LTS x86_64 using GCC 13 and Qt version 6.4
- AlmaLinux 9.8 x86_64 using GCC 11 and Qt version 6.6
- Fedora Linux 43 x86_64 using Clang 21 and Qt version 6.10
- Fedora Linux 43 x86_64 using GCC 15 and Qt version 6.10
- Apple macOS 12 (Monterey) with Xcode 14.2 / AppleClang 14 on arm64 and x86_64, Qt version 6.5
- Apple macOS 14 (Sonoma) with Xcode 16.4 / AppleClang 17 on arm64, Qt version 6.8
- Microsoft Windows Server 2025 x86_64 with Visual Studio 2022 and Visual C++ 14.40, Qt version 6.8
- Microsoft Windows 11 x86_64 with Visual Studio 2026 and Visual C++ 14.50, Qt version 6.10
- Microsoft Windows 11 x86_64 with MinGW / GCC 15.2 cross-compiler on Fedora 43, Qt version 6.10
- FreeBSD 15.1 x86_64 with Clang 19.1 and Qt version 6.11.1

Pre-compiled executables

Packages including a full LAMMPS version

For many users and especially for beginners learning to use LAMMPS, it is most convenient to install and use one of the pre-compiled packages that include both LAMMPS-GUI and the command-line version of LAMMPS. In these packages LAMMPS-GUI is linked directly to the included LAMMPS library and thus it *cannot* be changed in the *LAMMPS-GUI preferences dialog*. Such pre-compiled LAMMPS executable packages are available for download for Linux x86_64 (Red Hat Enterprise Linux 9.x or Ubuntu 22.04LTS or later and compatible), macOS (version 12 aka Monterey or later), and Windows (version 10 or later) from the LAMMPS releases page on GitHub. A backup download location is at https://download.lammps.org/static/ but may not always be up-to-date. Occasionally, also test version packages previewing recently added features are available at https://download.lammps.org/testing/ .

Standalone packages with a basic LAMMPS library



LAMMPS-GUI packages containing *only* LAMMPS-GUI compiled in plugin mode are available from the LAMMPS-GUI releases page on GitHub. Most of these packages include a LAMMPS shared library with some subset of LAMMPS' features that do not depend on additional libraries for improved portability.

If you want to override that choice of LAMMPS library, you can use the `-p` command line flag to tell LAMMPS-GUI which other LAMMPS shared library file you want it to load. By using `-p ""` you can also reset any previous choice and thus trigger loading the default library again. When resetting the LAMMPS shared library path or when the currently configured library file cannot be loaded or no longer exists, a dialog will appear that lets you re-download the default minimal LAMMPS shared library from the LAMMPS web server or browse the file system for a suitable custom shared library file. Once LAMMPS-GUI is running, you can also change the path to the LAMMPS shared library or re-download a pre-compiled copy from the *Preferences dialog*.

The flatpak version of the standalone LAMMPS-GUI package does not contain a pre-compiled library so it will directly show the download or browse dialog on the first invocation. When the flatpak version is updated, it may be required to reset the shared library location with `-p ""` and re-download the latest version.

Changed in version 3.1: The minimum LAMMPS version required by LAMMPS-GUI is now 2 September 2026

GPU support and MPI parallelization

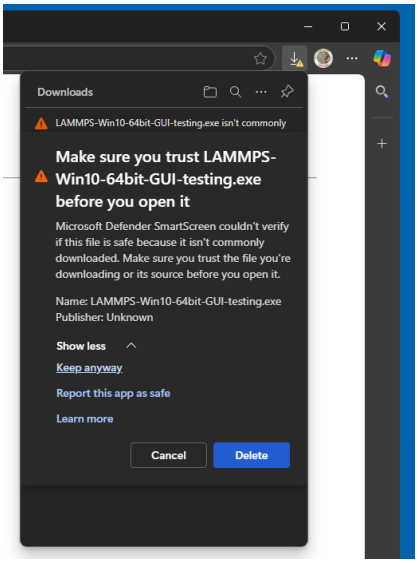
The pre-compiled packages include a LAMMPS version with support for GPUs through the GPU package using OpenCL in mixed precision. However, this requires that you have a compatible driver and the OpenCL runtime installed. This is not always available, and when using the LAMMPS flatpak bundle, the flatpak sandbox usually prevents accessing the GPU and thus the GPU package is disabled for that version. GPU support through the KOKKOS package is currently not available for technical reasons, but serial and OpenMP multi-threading use of KOKKOS is available.

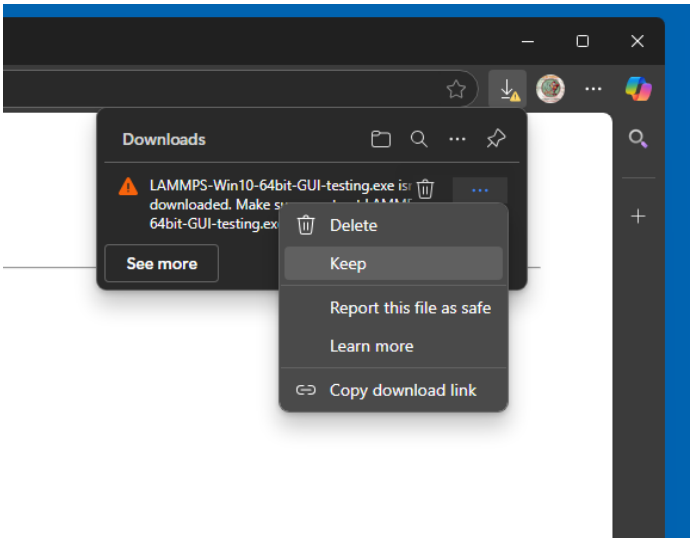
The design decisions for LAMMPS-GUI and how it launches LAMMPS conflict with parallel runs using MPI. You have to use a regular LAMMPS executable compiled with MPI support for that. For the use cases that LAMMPS-GUI has been conceived for (learning LAMMPS, testing or debugging LAMMPS inputs, prototyping new projects or complex workflows), this is not a significant limitation. Many supercomputing centers and high-performance computing clusters have parallel LAMMPS pre-installed.

Platform notes

Windows 10 and later

After downloading either the `LAMMPS-Win10-64bit-GUI-<LAMMPS version>.exe` or the `LAMMPS-GUI-Win10-x86_64-<LAMMPS-GUI version>.exe` installer package, you need to execute it, and start the installation process. Depending on your security settings of your web browser, you may have to explicitly tell it to download the file and then confirm **twice** to *keep the downloaded file* despite the claims that it may be dangerous and insecure. The main reason for that is that one needs to pay for being a registered developer and obtain a corresponding cryptographic signature to sign the binaries with.





Since this check is performed by the web browser, you can avoid it by opening a command prompt terminal window, change to the Downloads folder and then download the installer from the command line with:

`curl -OL https://github.com/lammps/lammps-gui/releases/download/<version>/LAMMPS-GUI-Win10-x86_64-<version>.exe` where you replace `<version>` with the version tag of the release, e.g. v3.1.0.

Managing Microsoft Defender SmartScreen protection

Since LAMMPS-GUI version 3.0.5 the Windows installer packages and the included executables are cryptographically signed with a *self-signed* certificate. You now have two options:

Option A: just ignore the warnings

When the message "Windows protected your PC" appears, click *More info*, and then *Run anyway*. You may also have to enable "Developer Mode" in the Windows System Settings to be able to run the installer. Using a cryptographic signature still guarantees the file has not been modified since we built it, so you have that extra protection. This may be needed since these installer packages sometimes cause false positives with anti-virus software and are reported as containing a trojan

Option B: install and trust our self-signed certificate

You can download our self-signed public certificate from https://download.lammps.org/lammps-gui/LAMMPS-GUI.cer (The SHA-256 hash of this file is `2a90ba8d0d3406fba6d67e6edb3f4904b1684756abf777bd2c58464ab1dff2cd`) and then open a "Command Prompt" with **Administrator** permissions. At the prompt you run the following two commands to import and trust the certificate.

```
certutil -addstore Root LAMMPS-GUI.cer
```

`certutil -addstore TrustedPublisher LAMMPS-GUI.cer` (if this step fails, it can be ignored)

Security note: Adding a certificate to the Root store means your computer will trust *anything* signed with the matching private key. Only install certificates from publishers you trust, obtained over HTTPS from their official site.

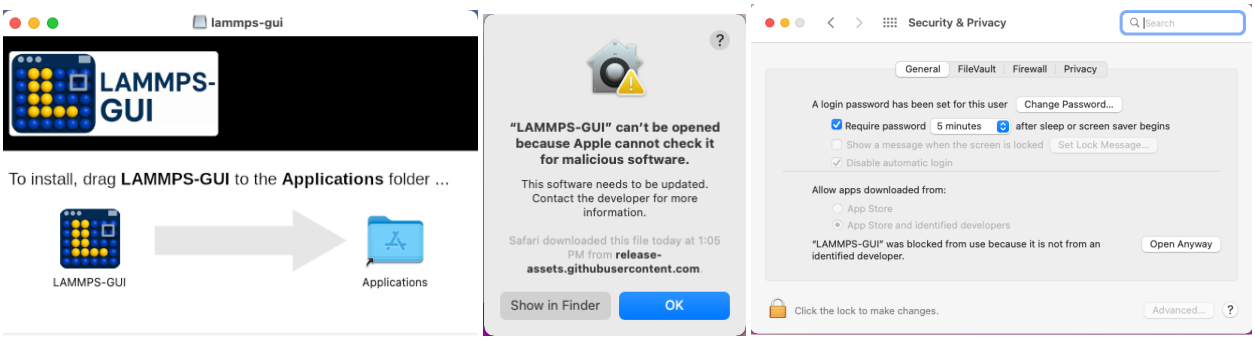
If you ever wish to *remove* this certificate, you can do it with the following commands.

```
certutil -delstore Root "The LAMMPS Developers"
```

```
certutil -delstore TrustedPublisher "The LAMMPS Developers"
```

MacOS 12 and later

After downloading the `LAMMPS-macOS-multiarch-GUI-<LAMMPS version>.dmg` or `LAMMPS-GUI-multiarch-<LAMMPS-GUI version>.dmg` application bundle disk image, you need to double-click it and then – in the window that opens – drag the app bundle as indicated into the "Applications" folder (see left image below). Afterwards, the disk image can be unmounted or ejected. For the `LAMMPS-macOS-multiarch-GUI-<LAMMPS version>.dmg` bundle, follow instructions in the "README.txt" file to get access to the other included command-line executables, if desired.



Because the executables are currently not cryptographically signed with a digital key provided by Apple, macOS will likely refuse to launch LAMMPS-GUI as shown in the the center image above. After this, the executable will be listed under `System Preferences -> Security & Privacy` where you can click on "Open Anyway" to override this (see right image). The details of this can change between different macOS versions (the screenshot images were created on macOS 12 Monterey), so you may need to look around or search the web to find the equivalent settings for your version.

A way to avoid this issue (verified with macOS 12 Monterey and macOS 26 Tahoe) is to download the disk image from a terminal window with `curl -OL https://github.com/lammps/lammps-gui/releases/download/<version>/LAMMPS-GUI-macOS-multiarch-<version>.dmg` where you replace `<version>` with the version tag, e.g. v3.1.0.

Linux on x86_64

For Linux with x86_64 CPU there are currently two variants of pre-compiled LAMMPS-GUI: 1) a tar file with binaries and a wrapper script and 2) a flatpak bundle. The first is currently compiled on AlmaLinux 9.8 or Ubuntu 22.04LTS (the oldest popular Linux distributions that provide the required C++17 compatibility out of the box and thus have the best chance that the pre-compiled binaries will also run on more recent Linux installations) and depends on the backward compatibility of the core libraries between different releases on Linux distributions. The second uses the flatpak sandbox environment to maintain binary compatibility across platforms; this uses a more recent build environment and Qt library release than the tar archive.

Linux binary tarball

After downloading and unpacking the `LAMMPS-Linux-x86_64-GUI-<LAMMPS version>.tar.gz` or the `LAMMPS-GUI-Linux-x86_64-<LAMMPS-GUI version>.tar.gz` package, you can switch into the "LAMMPS_GUI" folder and execute "./lammps-gui" directly:

```
$ cd ~/Downloads
$ tar -xzvvf LAMMPS-Linux-x86_64-GUI-30Mar2026.tar.gz
$ cd LAMMPS_GUI
$ ./lammps-gui &
```

The `LAMMPS_GUI` folder may also be moved around and added to the `PATH` environment variable so the executables will be found automatically.

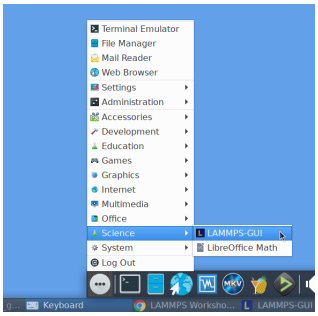
Installing required compatibility packages

Since software is constantly evolving, it may be required to install additional software packages for your Linux distribution to achieve compatibility with binaries compiled on older distributions. Using the flatpak bundle (see below) avoids these kind of issues by compiling and running the application in a standardized sandbox which is maintained by the flatpak software manager.

Linux flatpak bundle

The second Linux package variant uses flatpak software deployment environment and requires the flatpak management and runtime software to be installed. As with the binary tarball, there are two bundle variants: `LAMMPS-Linux-x86_64-GUI-<LAMMPS version>.flatpak` is built in the LAMMPS repository in linked mode and includes the LAMMPS console executable, while `LAMMPS-GUI-Linux-x86_64-<LAMMPS-GUI version>.flatpak` is built in the LAMMPS-GUI repository in plugin mode. After downloading either bundle, you can install it with:

```
$ cd ~/Downloads
$ flatpak install --user LAMMPS-Linux-x86_64-GUI-<version>.flatpak
```



After installation, LAMMPS-GUI should be integrated into your desktop environment under "Applications > Science" but also can be launched from the console with `flatpak run org.lammps.lammps-gui`. The flatpak bundle also includes the console LAMMPS executable `lmp` which can be launched to run simulations with, for example with:

```
flatpak run --command=lmp org.lammps.lammps-gui -in in.melt
```

Other bundled command-line executables are run the same way and can be listed with:

```
ls $(flatpak info --show-location org.lammps.lammps-gui)/files/bin
```

Compilation from source

History

The source for LAMMPS-GUI was included with the LAMMPS source code distribution until LAMMPS version 22 July 2025 in the folder `tools/lammps-gui`. Starting with LAMMPS-GUI version 1.8.0 and LAMMPS version 10 September 2025 the LAMMPS-GUI sources are distributed separately through its own git repository at https://github.com/lammps/lammps-gui.

LAMMPS-GUI can be built as part of a regular LAMMPS compilation. It will be automatically downloaded from its git repository and configured. This is usually the most convenient way to compile and install it. Since CMake is *required* to build LAMMPS-GUI, you need to build LAMMPS with CMake as well. To enable its compilation during compiling LAMMPS, the CMake variable `-D BUILD_LAMMPS_GUI=on` must be set when creating the CMake configuration. All other settings (compiler, flags, compile type) for LAMMPS-GUI are then inherited from the regular LAMMPS build. If the Qt library is installed as packaged for Linux distributions, then its location is typically auto-detected since the required CMake configuration files are stored in a location where CMake can find them without additional help. Otherwise, the location of the Qt library installation must be indicated by setting `-D Qt6_DIR=/path/to/qt6/lib/cmake/Qt6`, which is a path to a folder inside the Qt installation that contains the file `Qt6Config.cmake`.

The charts display is drawn by a self-contained native renderer (*`PlotWidget`*) built only on Qt Widgets and `QPainter`, so the build no longer depends on the Qt Charts or Qt Graphs modules. No extra CMake settings are required to select a chart backend.

The toolbar and menu icons are bundled in SVG format, so building and running LAMMPS-GUI also requires the **Qt Svg** module, which provides the SVG icon engine that Qt uses to render them. On Linux distributions this module is often packaged separately from the Qt base libraries (for example the `qt6-svg-dev` development package, which pulls in the `libqt6svg6` runtime, on Debian and Ubuntu). The CMake configuration requires it, and if the module – or, at run time, its icon-engine plugin – is missing, the icons render blank. The pre-compiled packages and installers already bundle it.

LAMMPS-GUI plugin version

It is possible to compile a standalone LAMMPS-GUI executable (e.g. when LAMMPS has been compiled with traditional make). Rather than linking to the LAMMPS library during compilation, it includes a plugin loader that will load a LAMMPS shared library file dynamically at runtime during the start of the GUI; e.g. `liblammps.so.0` or `liblammps.0.dylib` or `liblammps.dll` (depending on the operating system). This has the advantage that the LAMMPS library can be built from updated or modified LAMMPS source without having to (re-)compile the GUI.

The ABI of the LAMMPS C-library interface is very stable and generally backward compatible. However, features used in LAMMPS-GUI may require a minimum LAMMPS version of the library. LAMMPS-GUI will print a suitable error message and exit if an incompatible LAMMPS library is loaded. You can override the path to the LAMMPS library with the `-p <path>` or `--pluginpath <path>` command-line flag. This is usually auto-detected on the first run and can be changed in the LAMMPS-GUI *Preferences* dialog. The command-line flag lets you reset this path to a valid value in case the original setting has become invalid. An empty path ("") as argument restores the default setting.

It is also possible to link the standalone compiled LAMMPS-GUI version to the LAMMPS library directly. This feature is enabled by setting `-D LAMMPS_GUI_USE_PLUGIN=off` (default setting is on). This is also the setting for compilation within LAMMPS. In this case, the CMake configuration needs to be told where to find the LAMMPS headers and the LAMMPS library, via `-D LAMMPS_SOURCE_DIR=/path/to/lammps/src` and `-D LAMMPS_LIBRARY=/path/to/liblammps/file`.

Platform notes

macOS

When building on macOS, the build procedure will try to create a drag-n-drop installer, `LAMMPS-GUI-macOS-multiarch-<version>.dmg`, when using the 'dmg' target (i.e. `cmake --build <build dir> --target dmg` or `make dmg`). This requires the dmgbuild Python package, which the build script installs with `python3 -m pip install --user dmgbuild`, if it is missing. To use a different Python interpreter, set the `PYTHON` environment variable. Since dmgbuild sets up the Finder window layout of the disk image directly, no GUI session is needed, i.e. the installer can also be created when logged in remotely via ssh.

To build multi-arch executables on macOS that will run on both, arm64 and x86_64 architectures natively, it is necessary to set the CMake variable `-D CMAKE_OSX_ARCHITECTURES=arm64;x86_64`. To achieve wide compatibility with different macOS versions, you can also set `-D CMAKE_OSX_DEPLOYMENT_TARGET=12.0` which will set compatibility

to macOS 12 (Monterey) and later, even if you are compiling on a more recent macOS version. These are the settings currently used when building the pre-compiled LAMMPS-GUI packages.

Windows

On Windows either native compilation from within Visual Studio 2022 or Visual Studio 2026 with Visual C++ is supported and tested, or compilation with the MinGW / GCC cross-compiler environment on Fedora Linux. All pre-compiled LAMMPS-GUI packages for Windows are created with the MinGW64 cross-compiler; the native Visual C++ compilation is a development configuration without deployment or packaging support.

Since LAMMPS-GUI version 3.0.5, the build process includes generating cryptographically signed executables and installer packages. This is enabled by default but can be turned off with `-D CODE_SIGNING=no` during CMake configuration and setting the environment variable `SIGN_DISABLE` to 1.

Visual Studio

Using CMake and Ninja as the build system is required. Qt needs to be installed; a binary Qt package downloaded from https://www.qt.io was tested, which installs into the `C:\\Qt` folder by default. The compilation is verified by a GitHub action for every proposed change, and the compiled `lammps-gui.exe` executable is run directly from the build folder. Please note that the pre-compiled LAMMPS shared libraries downloaded by LAMMPS-GUI are built with the MinGW64 cross-compiler and use a different C runtime than Visual C++, so they are not compatible with a Visual C++ compiled executable; the download options are therefore disabled in this configuration. Instead, a LAMMPS shared library must also be compiled with Visual C++ and then either be selected manually in the `Preferences` dialog (plugin mode) or be linked directly (with `-D LAMMPS_GUI_USE_PLUGIN=no`).

In addition, LAMMPS-GUI and the LAMMPS shared library must be compiled with the *same* build configuration, i.e. both as `Release` or both as `Debug` builds. Debug and Release builds link different, mutually isolated Visual C++ runtime DLLs (`ucrtbased.dll` versus `ucrtbase.dll`), and capturing the LAMMPS screen output only works when the executable and the library share the same C runtime instance. With mismatched build configurations, simulations run normally, but the captured output remains empty because the LAMMPS library writes to the console of a different C runtime.

MinGW64 Cross-compiler

The standard CMake build procedure for cross-compilation can be applied. By using the `mingw64-cmake` wrapper the CMake configuration will automatically include a suitable CMake toolchain file (the regular cmake command can be used after that to modify the configuration settings, if needed). After building the libraries and executables, you can build the target 'nsis' (i.e. `cmake --build <build dir> --target nsis` or `make nsis`) to build a Nullsoft installer package executable that can be executed on a Windows 10 or later machine with x86_64 CPU and will then install LAMMPS-GUI including a basic LAMMPS shared library file and all required dependencies.

Linux

Binary tarball package

Version 6.2 or later of the Qt library is required. Those are provided by, e.g., Ubuntu 22.04LTS or later or RHEL 9.x or later. Thus older Linux distributions are not likely to be supported, while more recent ones will work, even for pre-compiled executables (see above). After compiling with `cmake --build <build folder>`, use `cmake --build <build folder> --target tgz` or `make tgz` to build a `LAMMPS-GUI-Linux-x86_64-<version>.tar.gz` file with the executables and their support libraries.

Flatpak bundle

It is also possible to build a flatpak bundle which is a way to distribute applications in a way that is compatible with most Linux distributions (provided the flatpak system is installed). Use the "flatpak" target to trigger a compile (`cmake --build <build folder> --target flatpak` or `make flatpak`). Please note that this will not build from the local sources but from the repository and branch listed in the `org.lammps.lammps-gui.yml` LAMMPS-GUI source folder.

4.2 Overview

LAMMPS-GUI is a graphical text editor customized for editing LAMMPS input files. It uses the LAMMPS C-language library interface and thus can run LAMMPS directly using the contents of the editor's text buffer. It can retrieve and display information from LAMMPS while it is running, display visualizations created with the dump image command, and is adapted specifically for editing LAMMPS input files through syntax highlighting, text completion, and reformatting, and linking to the online LAMMPS documentation for known LAMMPS commands and styles.

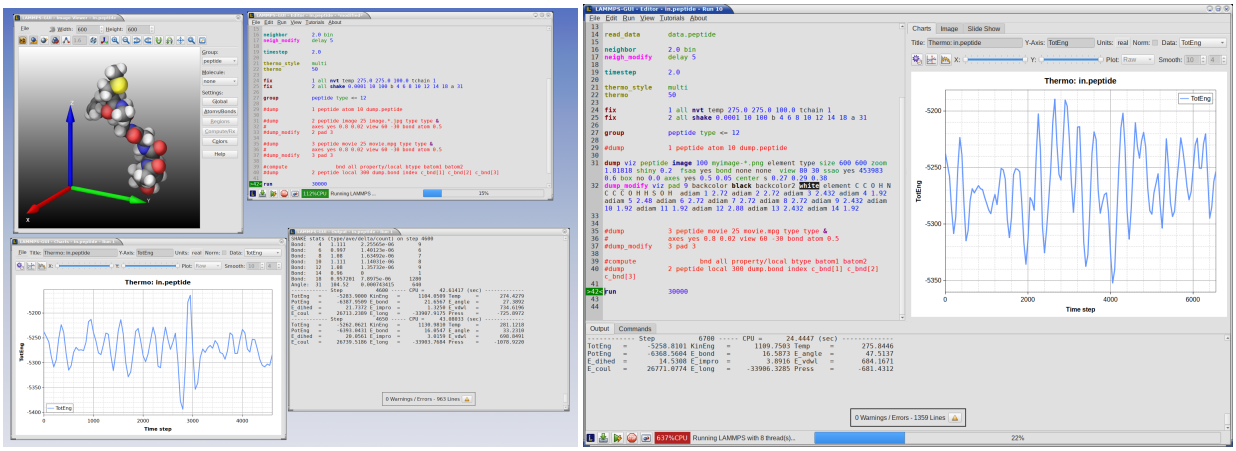
LAMMPS-GUI aims to support a workflow similar to the traditional experience of running LAMMPS using a text editor, a command-line window, launching the LAMMPS text-mode executable printing output to the screen, and post-processing and visualizing LAMMPS' output but just integrated into a single application.

LAMMPS-GUI integrates well with graphical desktop environments where the `.lmp` filename extension can be registered with LAMMPS-GUI as the executable to launch when double-clicking on such files using a file manager. LAMMPS-GUI will launch and read the file into its buffer. Input files can also be dropped into the editor window of the running LAMMPS-GUI application, which will close the current file and open the new file.

LAMMPS-GUI makes it easier for beginners to get started running LAMMPS and is well-suited for LAMMPS tutorials, since you only need to work with a single, ready-to-use program for most of the tasks. It is available for download as a pre-compiled package for popular operating systems (Linux, macOS, Windows). This saves time and allows users to focus on learning LAMMPS itself, without the need to learn how to compile LAMMPS, learn how to use the command line, or learn how to use a separate text editor, plotting or visualization program.

The tutorials at https://lammpstutorials.github.io/ are specifically designed for use with LAMMPS-GUI. Their tutorial materials can be downloaded and edited directly from within the GUI while automatically loading the matching tutorial instructions into a web browser.

While making it easy for beginners to get started with LAMMPS, it is expected that LAMMPS-GUI users will eventually transition to workflows that most experienced LAMMPS users employ. That traditional procedure is effective for people proficient in using the command line, as it allows them to use the tools for the individual steps that they are most comfortable with. In fact, it is often *required* to adopt this workflow when running LAMMPS simulations on high-performance computing facilities.



Most features in LAMMPS-GUI have been exposed to keyboard shortcuts, making it also appealing for experienced LAMMPS users for prototyping and testing simulation setups.

LAMMPS-GUI Key Features

A detailed discussion and explanation of all features and functionality are in the following pages. Here are a few highlights of LAMMPS-GUI:

- Individual windows or combined window viewing modes

- Text editor with line numbers, syntax highlighting, and find & replace, customized for LAMMPS
- Text editor features command completion and indentation for known commands and styles
- Input validation with a static pre-run check of the input script and a dry-run mode that executes only the setup phase of each command
- Switch its working directory to the folder of the file in the buffer
- Indicator for currently executed command and line that caused an error
- Progress bar indicates how far a run command has completed and how the CPUs are utilized
- Context-sensitive help for LAMMPS commands via the online documentation
- Auto-adapting to features and packages available in the LAMMPS library in use
- LAMMPS is running in a concurrent thread, so the GUI remains responsive
- LAMMPS can be started and stopped with a mouse click or a hotkey
- Screen output is captured in an *Output* window
- Many adjustable settings and preferences are persistent, including the 5 most recent files
- Thermodynamic output is captured and displayed as a line graph in a *Charts* window
- Export of thermodynamic data to CSV, YAML, and multi-column text format
- Create simple plots from imported CSV, YAML, and plain multi-column data
- Multiple post-processing options for plot data, and plot style customizations
- Interactive visualization of current state via calling dump image
- Capture of images created by dump image in the *Slide Show* window
- Export of *Slide Show* animations to movie files, or images to arbitrary formats
- Dialog to set variables, similar to the LAMMPS command-line flag '-v' / '-var'
- Support for GPU, INTEL, KOKKOS/OpenMP, OPENMP, and OPT accelerator packages
- Inspection of binary restart files created by LAMMPS
- View text, image, and animation/movie files
- Integration with LAMMPS tutorials
- Command prompt window for issuing shell commands with Tab completion for commands and filenames, scrollable history and aliases
- Update dynamically loaded LAMMPS library from LAMMPS download server

4.3 Basic usage of LAMMPS-GUI

Command-line options

LAMMPS-GUI supports the following command-line options:

```
lammps-gui [options] [file]
```

Option	Description
`-p <path>`, `--pluginpath <path>`	Set the path to the LAMMPS shared library (plugin mode only)
`-h`, `--help`	Print usage information and exit
`--help-all`	Same as above but also show generic Qt command-line options
`-v`, `--version`	Print version information and exit
`-x <width>`, `--width <width>`	Override the editor/main window width in pixels
`-y <height>`, `--height <height>`	Override the editor/main window height in pixels
`-s <style>`, `--style <style>`	Set the visual style of the application (default: `Fusion`)
`-c <file>`, `--chart <file>`	Open `file` directly in a standalone *Charts window*; a column-picker dialog is shown first
`-i <file>`, `--image <file>`	Open `file` in the *slide show viewer*; may be given multiple times to load several images at once
`-t <file>`, `--text <file>`	Open `file` in a standalone text viewer
`-j`, `--joined`	Selects the combined window mode for this run, whatever the preferences say
`-w`, `--windows`	Selects the individual window mode for this run, whatever the preferences say

The `-j` and `-w` flags select the window layout (see the *Preferences dialog*) for the process they are given to and nothing else: the stored preference is not changed, so the next start without a flag is back to what the *Preferences* dialog shows, and so is a relaunch (which passes no arguments on). Giving both at once is an error.

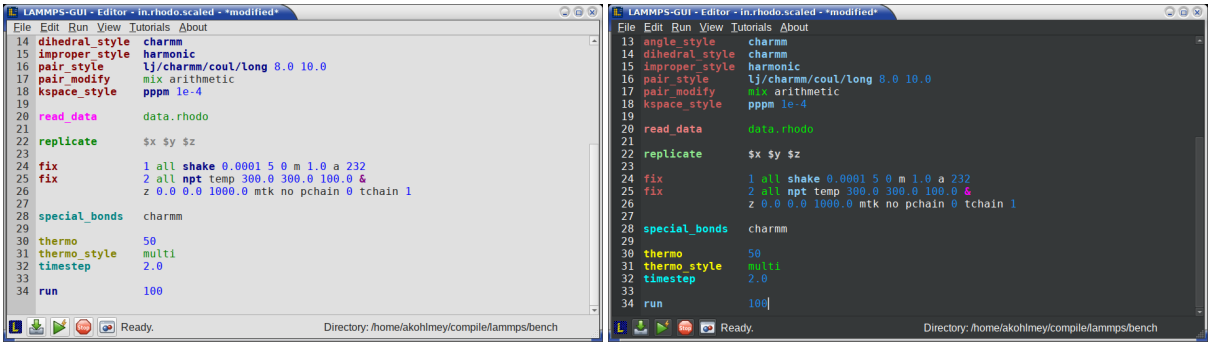
On Linux and other Unix-like systems except macOS, LAMMPS-GUI checks at startup whether a graphical display is available, i.e. whether either of the `DISPLAY` or `WAYLAND_DISPLAY` environment variables is set. Without one, the `-h`, `--help-all`, and `-v` flags still print their text (`--help-all` then lists only the generic Qt options that do not need a display) and `-p` still stores the library path, but LAMMPS-GUI itself cannot start: it prints an error message and exits. Selecting a Qt platform plugin explicitly, with `-platform <name>` or the `QT_QPA_PLATFORM` environment variable, skips this check. For example, `-platform vnc` makes LAMMPS-GUI act as a VNC server, so it can be used from a VNC viewer on another machine.

The optional `file` argument specifies a LAMMPS input file to open on startup. If no file is provided, LAMMPS-GUI starts with an empty editor buffer. Available choices for the visual style depend on the platform and Qt configuration. On most platforms, there is also the option `Windows` which is a visual style somewhat resembling Windows 95.

The `-c`, `-i`, and `-t` flags open a standalone viewer without the main editor window; they are mutually exclusive with each other and with the `file` positional argument. Such a viewer is always an individual window with a menu bar of its own, since there is no main window for it to be a panel of, whatever the layout preference or `-j` say. The `-i` flag does not support wildcards via filename globbing, but that can be emulated; for example on a Bourne shell with: `lammps-gui $(for f in image-*.png; do echo " -i $f"; done)`.

Launching LAMMPS-GUI

When LAMMPS-GUI starts, it shows the main window, labeled *Editor*, with either an empty buffer or the contents of the file used as argument. In the latter case it may look like the following:



There is the typical menu bar at the top, then the main editor buffer, and a status bar at the bottom. The input file contents are shown with line numbers on the left and the input is colored according to the LAMMPS input file syntax. The status bar shows the status of LAMMPS execution on the left (e.g. "Ready." when idle) and the current working directory on the right. The name of the current file in the buffer is shown in the window title; the word **modified** is added if the buffer edits have not yet been saved to a file. The geometry of the main window is stored when exiting and restored when starting again.

Opening and saving files

The LAMMPS-GUI application can be launched without command-line arguments and then starts with an empty buffer in the *Editor* window.

If a file argument is given, LAMMPS-GUI will use it as the file name for the *Editor* buffer and read its contents into the buffer, provided a file of that name exists; otherwise the buffer will be empty, but set up to save any added content to that file. Files can also be opened via the *File* menu, the *Ctrl-O* (*Command-O* on macOS) keyboard shortcut or by drag-and-drop of a file from a graphical file manager into the editor window. If a file extension (e.g. `.lmp`) has been registered with the graphical environment to launch LAMMPS-GUI, an existing input file can be launched with LAMMPS-GUI through double clicking.

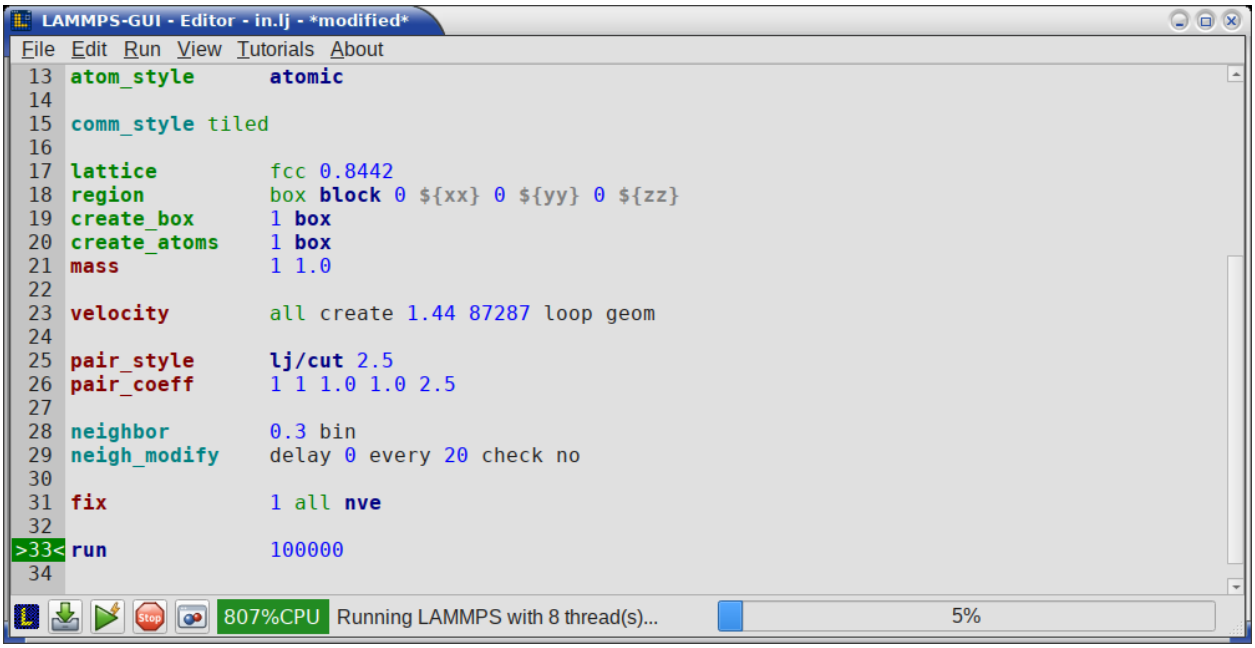
Only one file can be edited at a time, so opening a new file with a file already loaded into the buffer closes that buffer. If the buffer has unsaved modifications, you are asked to either cancel the operation, discard the changes, or save them. A buffer with modifications can be saved any time from the *File* menu, by the keyboard shortcut *Ctrl-S* (*Command-S* on macOS), or by clicking on the *Save* button at the very left in the status bar.

Running LAMMPS

From within the LAMMPS-GUI main window LAMMPS can be started either from the *Run* menu using the *Run LAMMPS from Editor Buffer* entry, by the keyboard shortcut *Ctrl-Enter* (*Command-Enter* on macOS), or by clicking on the green *Run* button in the status bar. All of these operations cause LAMMPS to process the entire input script in the editor buffer, which may contain multiple run or minimize commands.

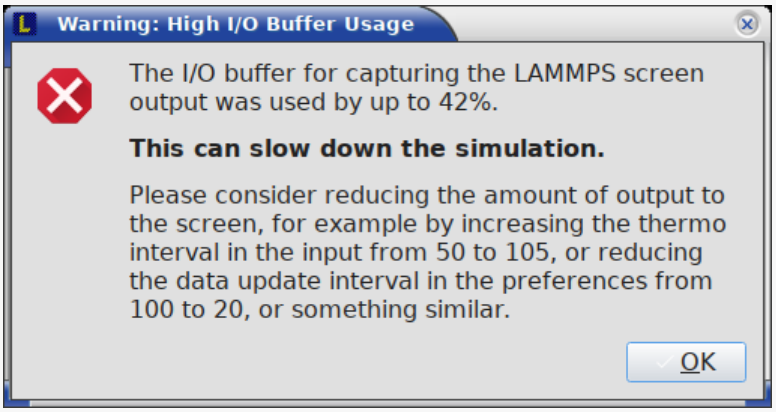
LAMMPS runs in a separate thread, so the GUI stays responsive and is able to interact with the running calculation and access data it produces. It is important to note that running LAMMPS this way uses the contents of the input buffer for the run (internally by calling the lammps_commands_string function of the LAMMPS C-library interface), and **not** the original file it was read from. Thus, if there are unsaved changes in the buffer, they *will* be used. As an alternative, it is also possible to run LAMMPS by reading the contents of a file from the *Run LAMMPS from File* menu entry or with *Ctrl-Shift-Enter*. This option may be required in some rare cases where the input uses some functionality that is not compatible with running LAMMPS from a string buffer. For consistency, any unsaved changes in the buffer must be either saved to the file or undone before LAMMPS can be run from a file.

The line number of the currently executed command is highlighted in green in the line number display for the *Editor* window.



While LAMMPS is running, the contents of the status bar change. The text fields that normally show "Ready." and the current working directory change into an area showing the CPU utilization in percent. Next to it is a text indicating that LAMMPS is running, which also indicates the number of active threads (in case thread-parallel acceleration was selected in the *Preferences* dialog). On the right side, a progress bar is shown that displays the estimated progress for the current run or minimize command.

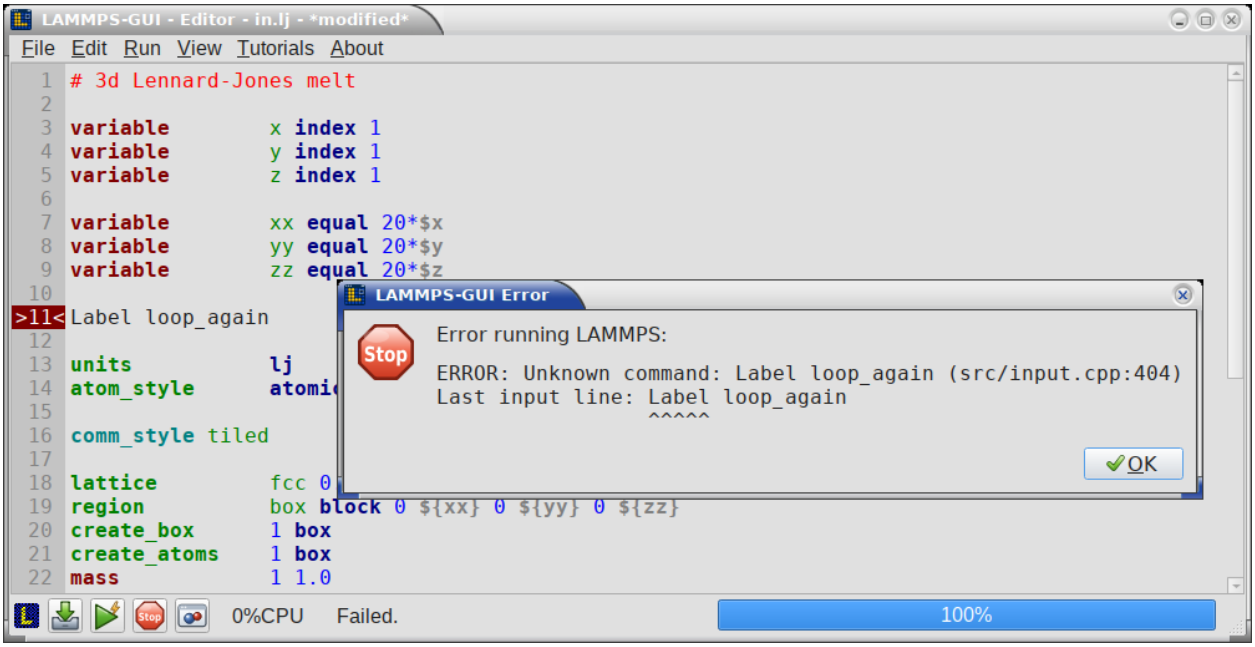
CPU Utilization



The CPU Utilization should ideally be close to 100% times the number of threads like in the screenshot image above. Since the GUI is running as a separate thread, the CPU utilization *may* be higher, for example when the GUI needs to work hard to keep up with the output produced by the simulation. This can occur when there is frequent thermo output or the simulation runs very fast. In the *Preferences* dialog, the polling interval for updating the *Output* and *Charts* windows can be adjusted. The intervals may need to be lowered to avoid missing data between *Chart* data updates or to avoid stalling when the thermo output is not transferred to the *Output* window fast enough. You could also make LAMMPS run slower by reducing or turning off thread parallelization. It is also possible to reduce the amount of data by increasing the thermo interval. LAMMPS-GUI detects if the associated I/O buffer is significantly full, and will print a warning *after* the run with suggested adjustments. The CPU utilization can also be lower than expected when some significant parts of the code paths in use are not multi-threaded, or when the simulation is slowed down by the GUI or other processes also running on the host computer and competing with

LAMMPS-GUI for resources.

If an error occurs (in the example below the command label was incorrectly capitalized as "Label"), an error message dialog is shown and the line of the input which triggered the error is highlighted in red. The state of LAMMPS in the status bar is set to "Failed." instead of "Ready." Note that since version 3.0.6 the input is checked for problems before a run by default (see the *Run menu*); a typo like this is normally caught by that check first, and the dialog shown here appears when choosing to run anyway or when the pre-run check is disabled.



Up to three additional windows or tabs may open during a run

- An *Output window* with the captured screen output from LAMMPS
- A *Charts window* with a line graph created from thermodynamic output of the run
- A *Slide Show window* with images created by a dump image command in the input

More information on those windows and how to adjust their behavior and contents is given in *the next pages*.

An active LAMMPS run can be stopped cleanly by using either the *Stop LAMMPS* entry in the *Run* menu, the keyboard shortcut *Ctrl-/* (*Command-/* on macOS), or by clicking on the red button in the status bar. This will cause the running LAMMPS process to complete the current timestep (or iteration for energy minimization) and then complete the processing of the buffer while skipping all run or minimize commands. This is equivalent to the input script command timer timeout 0 and is implemented by calling the lammps_force_timeout function of the LAMMPS C-library interface. Please see the corresponding documentation pages to understand the implications of this operation.

4.4 Monitoring LAMMPS output

LAMMPS-GUI Viewing Modes

Since version 3.1 LAMMPS-GUI supports two viewing modes: 1) individual window mode, where each output is displayed in a separate window and 2) combined window mode, where there is only one main window that may be

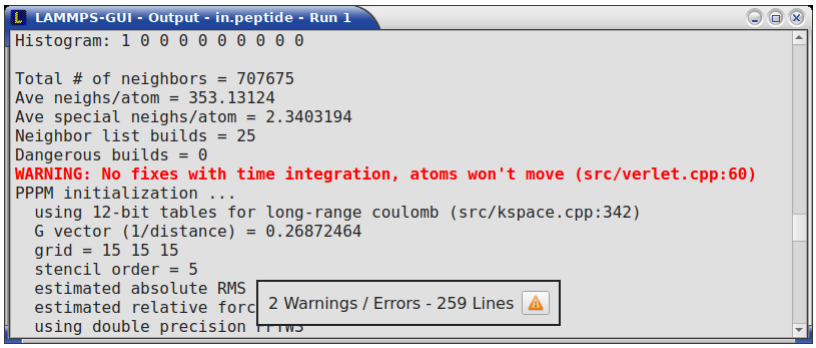
split once vertically and the upper part once more horizontally. The upper left section is the editor window and the upper right and bottom sections contain one or more tabs with the same content as the corresponding individual windows in mode 1). This viewing mode can be changed in the *Preferences dialog* or (temporarily) selected with the `-w` or `-j` *command-line options*.

Any reference to a "window" throughout the remainder of this documentation thus refers to either the corresponding individual window or the tab in the upper right or bottom part of the combined window. Screenshots are of the individual windows.

The behavior for both modes is largely the same with two exceptions:

1. There can be only one menu bar, so the displayed menu bar is that of the section / tab that currently has the focus. If access to the menu bar of a specific window is needed, e.g. to open a new LAMMPS input file, it may be needed to first click into the corresponding area or use the *F6* or *Shift-F6* keyboard shortcuts to switch focus.
2. The *Image Viewer* and *Slide Show* windows lose the auto-resize option to show the image without scroll bars if the screen size supports it.

Output Window



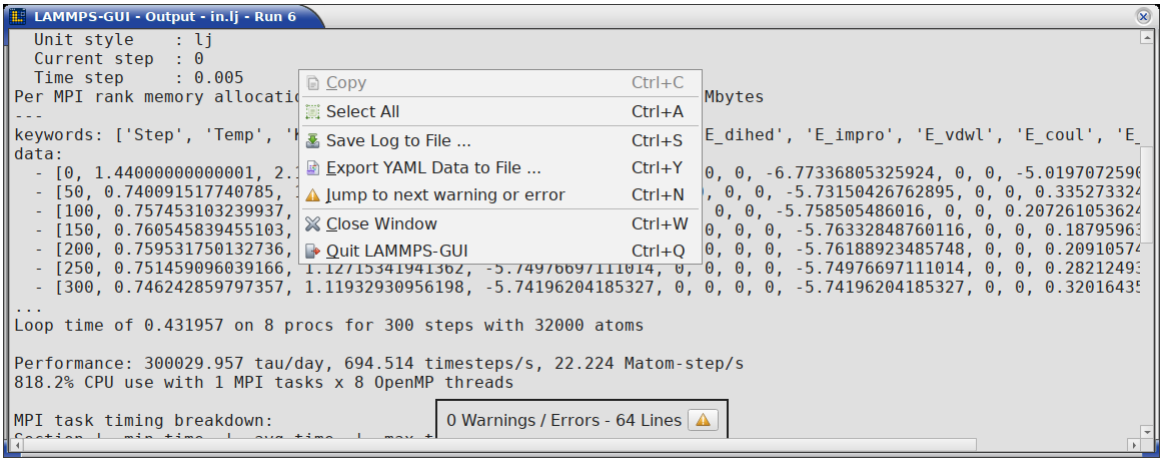
By default, when starting a run, an *Output* window opens that displays the screen output of the running LAMMPS calculation, as shown below. This text would normally be seen in the command-line window.

LAMMPS-GUI captures the screen output from LAMMPS as it is generated and updates the *Output* window regularly during a run. If there are any warnings or errors in the LAMMPS output, they are highlighted by using bold text colored in red. There is a small panel at the bottom center of the *Output* window showing how many warnings and errors were detected and how many lines the entire output has. By clicking on the button on the right with the warning symbol or by using the keyboard shortcut *Ctrl-N* (*Command-N* on macOS), you can jump to the next line with a warning or error. If there is a URL pointing to additional explanations in the online manual, that URL will be highlighted and double-clicking on it shall open the corresponding manual page in the web browser. The option is also available from the context menu.

The *Output* window is reused for every run: starting a run replaces its content. The runs are counted and the run number for the current run is displayed in the window title (with the *Combined Main Window* layout, in the title of the main window). Whether the *Output* window opens by default can be changed in the preferences dialog, and it can be shown or hidden at any time from the *View* menu.

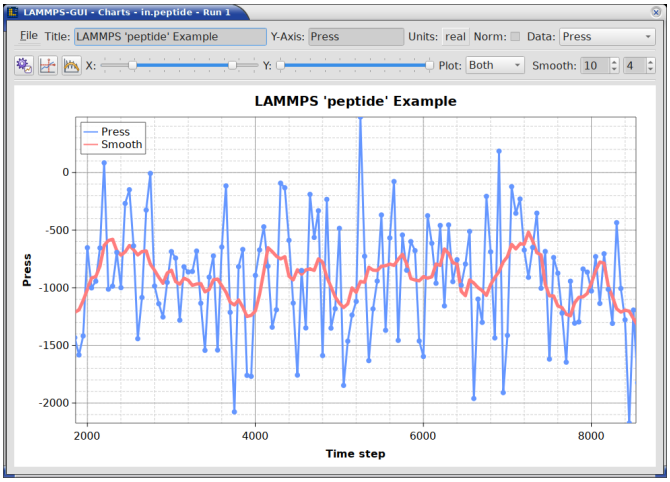
The text in the *Output* window is read-only and cannot be modified, but keyboard shortcuts to select and copy all or parts of the text can be used to transfer text to another program. Also, the keyboard shortcut *Ctrl-S* (*Command-S* on macOS) is available to save the *Output* buffer to a file. The window has a *File* menu carrying these functions, followed by the application-wide *Run*, *View*, *Tutorials*, and *About* menus that every output window shows, so a run can be started

or stopped from here as well. The “Select All” and “Copy” functions, as well as a “Save Log to File” option are also available from a context menu by clicking with the right mouse button into the *Output* window text area.



Should the *Output* window contain embedded YAML format text (see above for a demonstration), for example from using thermo_style yaml or thermo_modify line yaml, the keyboard shortcut *Ctrl-Y* (*Command-Y* on macOS) is available to save only the YAML parts to a file. This option is also available from a context menu by clicking with the right mouse button into the *Output* window text area.

Charts Window



By default, when starting a run, a *Charts* window opens that displays a plot of thermodynamic output of the LAMMPS calculation as shown below.

The “Data:” drop-down menu on the top right allows selection of different properties that are computed and written as thermodynamic output to the output window. Only one property can be shown at a time. The plots are updated regularly with new data as the run progresses, so they can be used to visually monitor the evolution of available properties. The update interval can be set in the *Preferences* dialog. By default, the raw data for the selected property is plotted as a blue graph. From the “Plot:” drop-down menu on the second row (immediately to the right of the *Chart Style...* and *Postprocess...* quick-access buttons), you can select whether to plot only raw data graph, only a smoothed data graph, or both graphs on top of each other. The smoothing process uses a Savitzky-Golay convolution filter. The convolution window width (left) and order (right) parameters can be set in the boxes next to the drop-down menu. Default settings are 10 and 4 which means that the smoothing window includes 10 points each to the left and the right of the current data point for a total of 21 points and a fourth order polynomial is fitted to the data in the window.

The "Title:" and "Y:" input boxes let you edit the text shown as the plot title and the y-axis label, respectively. The text entered in the "Title:" box is applied to *all* charts, while the "Y:" text changes only the y-axis label of the currently *selected* plot. In standalone plot mode (when plotting an external data file), an additional "X-Axis:" input box is shown to the right of "Y:" and sets the x-axis label for all charts.

The window title shows the current run number that this chart window corresponds to. Same as for the *Output* window, the chart window is reused and its charts are replaced on each new run.

From the *File* menu on the top left, it is possible to save an image of the currently displayed plot or export the data in either plain text columns (for use by plotting tools like gnuplot or grace), as CSV data which can be imported for further processing with Microsoft Excel, LibreOffice Calc, or with Python via pandas, or as YAML which can be imported into Python with PyYAML or pandas.

The export is not limited to the raw values: the results of the post-processing are written beside them, so that what was worked out in the chart window can be taken elsewhere. Each chart contributes its values, its error bars if it has any (as a `-err` column, or a `-errlo` and a `-errhi` column when they are asymmetric), its smoothed curve while smoothing is on (`-smooth`), any fit curve that is being shown, and any series added from a second file (`-added`). A flat table has a single x column, so everything in it is written against the x values of the data: a fit curve, which is sampled on a dense grid of its own, is written as the fitted function evaluated at each data point – which is also what makes it directly comparable to the values beside it. A series that carries x values of its own that do not match, as an added file generally does, is left out rather than resampled.

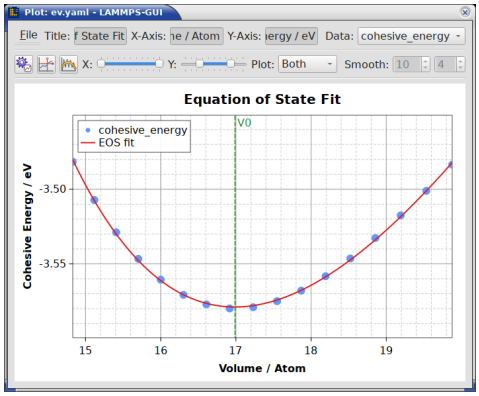
Thermo output data from successive run commands in the input script is combined into a single data set unless the format, number, or names of output columns are changed with a thermo_style or a thermo_modify command, or the current time step is reset with reset_timestep, or if a clear command is issued. This is where the YAML export from the *Charts* window differs from that of the *Output* window: here you get the compounded data set starting with the last change of output fields or timestep setting, while the export from the log will contain *all* YAML output but *segmented* into individual runs.

The *Preferences* dialog has a *Charts Settings* tab, where you can configure multiple chart-related settings, like the default title, colors for the graphs, default choice of the raw / smooth graph selection, whether the grid for the major and minor ticks is drawn, and the default chart graph size.

Slowdown of Simulations from Charts Data Processing

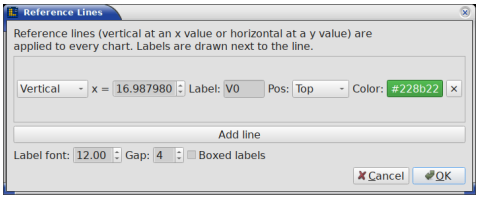
Using frequent thermo output during long simulations can result in a significant slowdown of that simulation since it is accumulating many data points for each of the thermo properties in the chart window to be redrawn with every update. The updates are consuming additional CPU time when smoothing is enabled. This slowdown can be confirmed when an increasing percentage of the total run time is spent in the "Output" or "Other" sections of the MPI task timing breakdown. It is thus recommended to use a large enough value as argument *N* for the thermo command and to select plotting only the "Raw" data in the *Charts* window during such simulations. It is always possible to switch between the different display styles for charts during the simulation and after it has finished.

Adjust chart style



The *Chart Style…* entry in the chart window's *File* menu, or the chart-style quick-access button at the far left of the second toolbar row, opens a dialog to change how the data is drawn. The *Raw data* and *Processed data* series each have independent settings for the display style (*Lines*, *Points*, or *Lines + Points*), the color, the line width, and the point size. This makes it possible, for example, to show the raw data as faint points and the smoothed curve as a bold line. The *Error bars* group sets the color and line width of the error bars of every series of the chart that has them (only imported data can, see *importing fix ave/* output*); they are drawn in their own color so that they stay readable where they overlap the curve they belong to. The defaults for all three groups are in the *Charts Settings* tab of the *Preferences* dialog. The *Legend* placement selector in the same dialog adds an in-plot legend that lists the visible named series; it can be turned *Off* or anchored to any of the four plot corners (*Top left*, *Top right*, *Bottom right*, or *Bottom left*). The selected placement is remembered across sessions.

Reference lines



The *Reference Lines…* entry in the chart window's *File* menu opens a dialog for adding straight annotation lines that are drawn on *every* chart in the window. Each line is either *Vertical* (at a chosen x value) or *Horizontal* (at a chosen y value) and can carry a text label and an individually chosen color. The label position along the line (*Top*, *Center*, or *Bottom* for vertical lines; *Left*, *Center*, or *Right* for horizontal lines) is selected per line, while the label font size, the gap between the label and the line, and whether the labels are drawn in an opaque box apply to the whole window. Reference lines are useful, for example, to mark a target temperature, a transition point, or a fitted value.

Post-process data

The *Postprocess…* entry in the chart window's *File* menu, or the quick-access button immediately to the right of the *Chart Style…* button, runs an analysis on the data of the currently selected property. The following analyses are available:

- *Autocorrelation* computes the normalized autocorrelation function of the selected data up to a chosen maximum lag and shows it in a new chart window (the abscissa becomes the lag). This is useful, for example, for estimating correlation times of fluctuating quantities.
- *Polynomial fit* performs a least-squares fit of a polynomial of a chosen degree, overlays the fitted curve on the chart, and reports the coefficients and the root-mean-square residual.
- *Birch-Murnaghan EOS fit* fits a 4-parameter Birch-Murnaghan equation of state to energy-versus-volume data (x = volume per unit cell, y = energy). A confirmation dialog lets you verify the x and y column assignments and enter the number of atoms in the conventional unit cell N (default 1; e.g. 4 for FCC, 2 for BCC or HCP; use N = 1 when the x data is already the conventional cell volume rather than volume per atom). The fit reports the equilibrium volume V_0, the derived lattice constant $a_0 = (N\,V_0)^{1/3}$, the equilibrium energy E_0, the bulk modulus B_0, and its pressure derivative B_0'.
- *Custom function* evaluates a user-supplied mathematical expression `f(x)` over the x range of the data and overlays it as a curve. The expression uses the variable `x` for the abscissa and supports the usual arithmetic operators and functions (for example `2*x^2 + 3*sin(x)`).
- *Custom fit* performs a nonlinear least-squares fit of a user-supplied expression to the data. In addition to the expression, you provide the fit parameters and their initial guesses as `name=value` pairs (for example `a=1, b=0.5`) and, optionally, a label for the fitted curve. The fit uses a Levenberg-Marquardt algorithm with analytic derivatives of the expression; on success the fitted curve is overlaid and the fitted parameters, the root-mean-square residual, and the number of iterations are reported.
- *Maxwell-Boltzmann fit* fits the distribution of the kinetic energy of d degrees of freedom, $f(E) = A\,E^{d/2-1}\exp(-E/k_BT)$, to the data. This is what a histogram of the per-atom kinetic energy is expected to follow, so importing a `fix ave/histo` file of `c_ke/atom` and fitting it reads the temperature off the shape of the distribution. The *Dimensions* selector sets d, the degrees of freedom **per atom** (three by default, which is the familiar $\sqrt{E}$ prefactor). Constrained and rigid molecules have fewer: a rigid 3-site water held by fix shake has six degrees of freedom per molecule, so two per atom, and fitting such a histogram with d = 3 returns a temperature that is wrong by a factor of two. The amplitude is fitted rather than derived, because a histogram carries an arbitrary normalization – raw counts, a normalized fraction, and a density differ by a constant that says nothing about the temperature. Reported are k_BT and the amplitude from the fitted shape, and next to them the measured mean energy $\langle E\rangle$ with the k_BT that follows from it through $\langle E\rangle = (d/2)k_BT$ alone. That second estimate makes no assumption about the shape – equipartition fixes it – so the two agreeing is a sign that the data really is the distribution being fitted, and the dialog says so when they differ by more than 10%. On a system the model describes – unconstrained atoms, and a histogram wide enough to hold the whole distribution – the two land within a fraction of a percent of each other, and the weighting then hardly matters either. They part company when d does not match the system, and when the histogram does not cover the whole distribution: energies beyond its range are missing from $\langle E\rangle$ and lower it, while the fitted shape is not affected. Note that k_BT comes out in the energy units of the plotted data, which the data file does not record; divide by the Boltzmann constant in those units to obtain a temperature. Points at $E \le 0$ are left out of the fit and counted in the report.

  A measured distribution is rarely a Maxwell-Boltzmann distribution exactly, and then the *Weighting* selector decides which part of it the one curve follows. *By bin population* (the default) counts every bin in proportion to the number of samples it holds, which keeps the fit on the bulk of the distribution and its peak; it is scale-free, so it behaves the same whether the histogram holds counts, fractions, or a density. *By error bars* is the textbook $1/\sigma^2$ weighting, which favors the points of smallest uncertainty – on a histogram usually the sparse tail, so it matches the peak *less* well. *None* weights every bin alike. Restricting the *Fit x-range* is the other way to say which part of a distribution matters.
- *Fourier transform* computes a transform of the data and shows it in a new chart window, with the angu-

lar frequency (rad per x unit) as the abscissa. Three kinds are offered: the one-sided *cosine* transform $2\int y(x)\cos(kx)\,dx$ – applied to a correlation function, for example imported `fix ave/correlate` output, this is the spectral density (Wiener-Khinchin theorem) – the matching *sine* transform, and the *power spectrum* $|\int y(x)e^{-ikx}\,dx|^2$, which does not care where the data starts on the x axis. The transform integral is evaluated directly (no FFT), so the x values need not be equally spaced and the output grid is free: it defaults to reaching the Nyquist limit of the mean sample spacing, and both its range and its resolution can be changed. The *Data x-range* restricts which part of the data is transformed, and the *Hann* window fades the data to zero toward the end of the range, which suppresses the ringing that truncating a correlation function before it has decayed would cause, at the price of some broadening.

- *Structure factor* computes the static structure factor $S(q) = 1 + 4\pi\rho\int r^2\,(g(r)-1)\,\frac{\sin(qr)}{qr}\,dr$ from radial distribution function data, such as the imported output of `compute rdf` written by `fix ave/time` in vector mode. It is applied to the plain $g(r)$ chart directly: the -1 shift is part of the formula, so no derived column is needed. *Density* is the number density N/V in the units of the r axis cubed; it scales $S(q)-1$, so getting it wrong stretches the structure away from 1 but moves no peak. The $\sin(qr)/(qr)$ form is regular at $q=0$, so the output grid may start at zero. Truncating $g(r)$ at the cutoff of the compute shows up as ringing in $S(q)$; the *Hann* window suppresses it.
- *Overlay other data column* copies the data of another column of the same chart window onto the current chart, for a direct comparison in one plot. The entry is only offered when the window has more than one column. The copy is a snapshot: it does not follow the source column afterwards, and it occupies the same overlay slot as a fitted curve, so the next fit or overlay replaces it.
- *Smoothed data (restore)* removes a fitted or overlaid curve again and returns the plot to the Savitzky-Golay smoothing of the raw data. The entry is only offered while a fit or overlay is in place; afterwards the "Plot:" drop-down reads "Smooth" again and the smoothing window and order controls are re-enabled.

The analyses whose result has a new x axis – autocorrelation, Fourier transform, and structure factor – show that result in a chart window of its own. With the *Combined Main Window* layout it does not open as a free window but joins the Charts tab group as a tab, as do any results computed from it in turn; closing the tab discards it.

The expressions for *Custom function* and *Custom fit* are parsed and evaluated with a bundled subset of the Lepton expression parser, the same library used by the LAMMPS Lepton-based styles, so the supported syntax matches.

When any fit or custom-function overlay is active, the "Plot:" drop-down treats the overlay as the "smoothed" series: selecting "Smoothed" or "Both" shows the overlay curve, while selecting "Raw" hides it. This applies uniformly to all analysis types (polynomial, EOS, custom function, and custom fit). To get the plain smoothed series back, select the *Smoothed data (restore)* entry the dialog offers while an overlay is active.

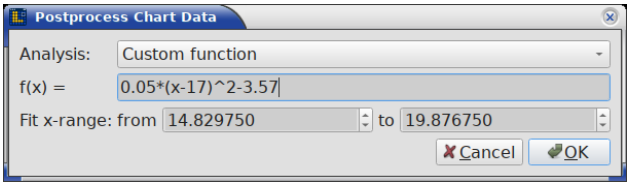


Fig. 1: The *Postprocess* dialog with a *Custom function* expression entered.

Plot imported data

The same *Charts* window is also used to plot data from an external file opened with *File -> Plot Data File…* (*Ctrl-Shift-P*, see *the File menu*); in that standalone mode there is no associated simulation, so the *Units* and *Norm* controls are hidden. The column-picker dialog shown before the chart opens lets you select which column provides the x axis and which columns to plot, and also allows renaming columns. A rename takes effect immediately: each column has exactly one name at any time, used in the column list, the preview, and the expressions described below alike. An "X-Axis:" label field in the first toolbar row (to the right of "Title:" and "Y:") lets you edit the x-axis label after the chart

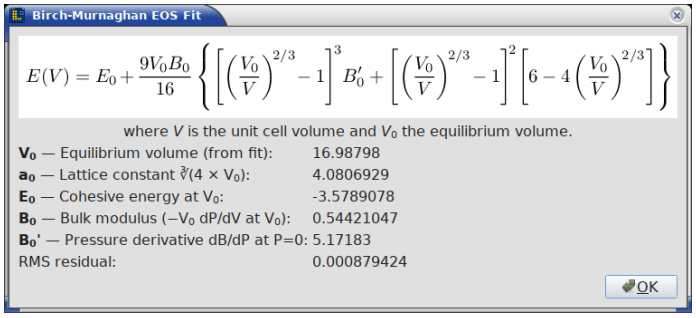


Fig. 2: An example post-processing result: a Birch-Murnaghan equation-of-state fit overlaid on energy-versus-volume data.

opens. All the styling, export, and post-processing features described above work the same way. The same column-picker and standalone chart window are also launched when LAMMPS-GUI is invoked from the command line with the `-c`/`--chart` flag (see *command-line options*).

The *Compute derived column* section of the column picker appends a new column computed row by row from an expression. Column values are referenced by name in braces, in the manner of Python format strings: `{name}` is the column's value in the current row, so an area-normalized energy is, for example, `{pe}/{area}*16021.766`. The braces end the name before the expression parser sees it, so names with special characters – `{c_rdf[2]}`, `{g(r)}`, or `{E / N}` – work the same way as plain ones. A colon inside the braces selects a per-column constant instead of the current-row value: `{name:first}`, `{name:last}`, `{name:min}`, `{name:max}`, and `{name:mean}`. Dividing by `{name:first}` scales a column relative to its initial value, and `{name}-{name:mean}` removes the mean. The variable `row` is the 0-based row index. Everything outside braces is never a column lookup, and the expression syntax is otherwise that of the bundled Lepton parser used by the post-processing analyses above. Because a colon in a column name would be ambiguous there, renames refuse names containing `{`, `}`, or `:` (a file may still supply such a name, but it has to be renamed before the column can be referenced). Renaming a column also rewrites the references in already added derived columns, so they keep meaning the same data.

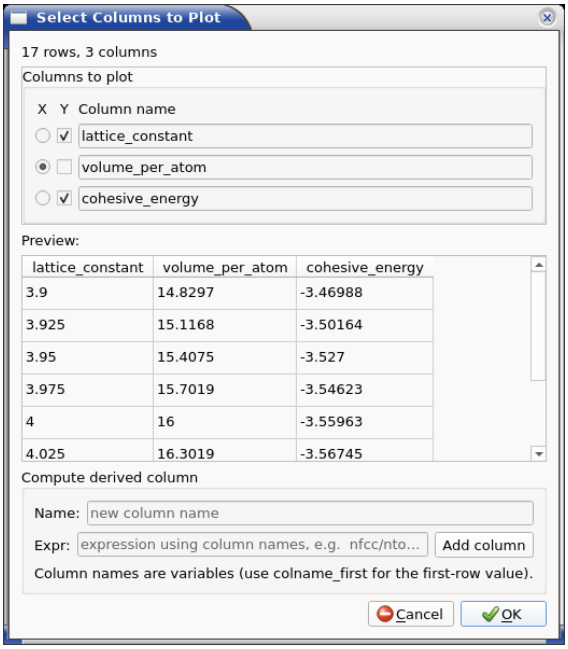


Fig. 3: The column-picker dialog shown when opening an external data file with *Plot Data File…*.

Import fix ave/* output files

The files written by fix ave/time in *vector* mode, fix ave/histo, fix ave/correlate, fix ave/correlate/long, and fix ave/chunk are not flat tables. Each is a sequence of blocks, one per output timestep, and each block is a small table of its own: the rows of a vector, the bins of a histogram, the time windows of a correlation function, or the chunks of a profile. Such a file is recognized when it is opened, and the column picker then grows a *Data blocks* group above the usual column grid, which reduces the blocks to the one flat table that grid refers to.

There are two ways to reduce the blocks:

Average blocks
: Average a range of blocks row by row and show the spread as error bars. The range covers the whole file, and the two spin boxes select a part of it; raising the first one is how a leading stretch of equilibration is left out. Nothing is dropped without being said: the line below reports how many blocks were averaged, and how many were dropped for having a different number of rows than the last block of the range.

Single block
: Show one block as it stands, by default the last one.

The error bars are the standard deviation of the blocks by default. The standard error of the mean, the *min/max of the blocks*, and no error bars at all are the other choices. Note that successive averaging windows are not strictly independent, so the standard error of the mean is a *lower bound* on the true uncertainty rather than the uncertainty itself. The min/max choice makes no statistical claim: the bar simply spans the smallest to the largest value any of the averaged blocks had for that row, so it usually reaches further in one direction than in the other. Error bars need at least two blocks to average.

How the bars are drawn – their color and line width – is set in the *Chart Style…* dialog of the chart window, and its defaults are in the *Charts Settings* tab of the *Preferences* dialog.

Which reduction the dialog starts on depends on the format. A histogram or a correlation function starts out averaged over the whole file, since its evolution over time is rarely what is wanted. A correlator that accumulates over the whole run does not: for `fix ave/correlate/long`, and for `fix ave/correlate` with `ave running`, every block is a successive estimate of the same quantity rather than an independent sample of it, so averaging the blocks would be statistically wrong and the last block is the answer. That case is recognized from a sample count that grows from block to block.

The columns that are preselected also follow from the format: the bin coordinate against the *normalized* bin count for a histogram (the per-block totals differ, so that is the column that may be averaged), the time delta against the correlation columns for a correlation function.

A `fix ave/chunk` file is preselected only when its chunks form a *profile*, that is when they vary along a single coordinate: a chart has one x axis, and a two- or three-dimensional grid of chunks has no meaningful projection onto it. That is decided from the coordinate values in the file rather than from the binning style, which the file does not record, so a `bin/1d` or `bin/sphere` profile always qualifies, and a `bin/cylinder` or `bin/2d` run that used a single bin in its other dimension qualifies as well – the varying coordinate becomes the x axis. Chunks that are a real grid, and chunks that are not bins at all (by molecule, by type, or from a compute), are still imported in full; they just get the same generic column defaults as any other block file.

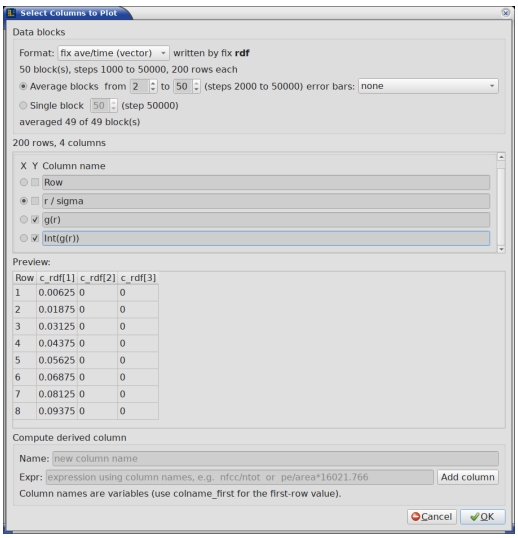
The *Format* combo shows what the file was recognized as, and can be corrected. Recognition uses the file's own header comments, which the `title1`, `title2` and `title3` keywords let you replace, so it can be wrong; when the headers are missing the format is recovered from the block structure instead. Correcting the format only moves the preselected reduction and columns. It never reinterprets the data, which was read before the dialog opened, and no reduction ever discards a column – a misrecognized file plots just as completely, only with different columns preselected.

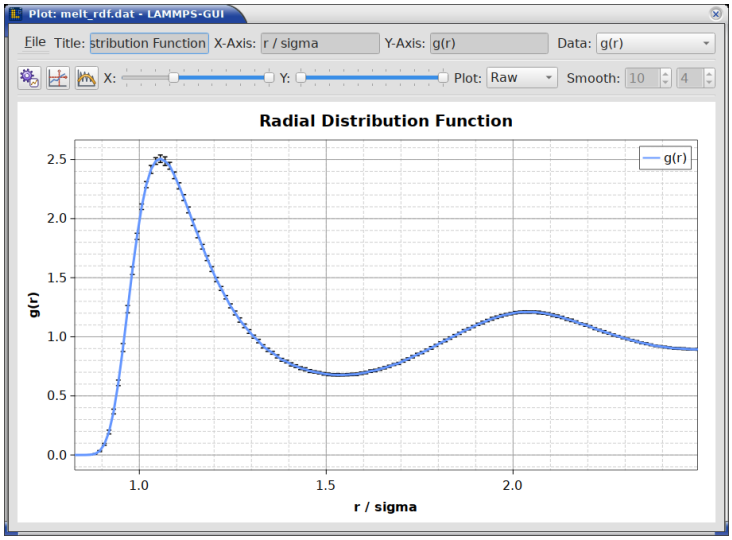
Changing the reduction rebuilds the column list below it. Column roles, edited names, and derived columns are kept across that: the derived columns are re-evaluated against the new table. Since any column can serve as the x axis and the *Compute derived column* section can scale one, a time-delta column in timesteps is turned into one in time units with an expression such as `{TimeDelta}*0.001`.

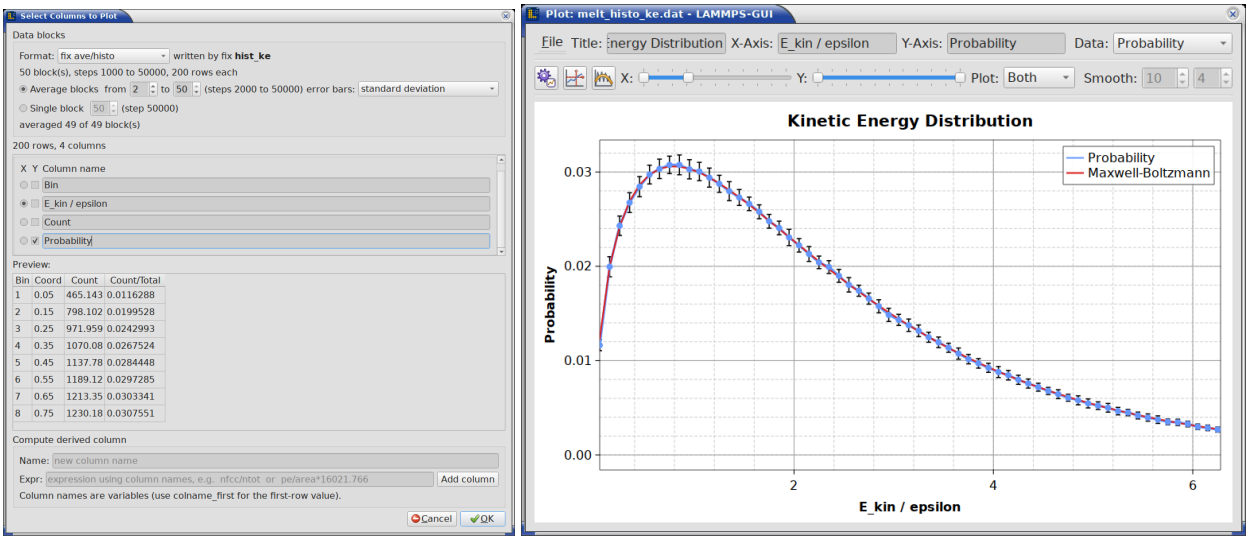
Error bars, once imported, behave like the rest of the chart data: smoothing operates on the values alone and leaves the bars on the raw series, the axis range covers them, and exporting the chart writes them as an extra `<name>-err` column next to the values they belong to. Reading such an exported file back in simply gives one more data column.

Here is a simple example for reproducing the radial distribution function g(r) and the Maxwell-Boltzmann distribution of the kinetic energy in a liquid LJ model. This uses the following input with fix ave/time and fix ave/histo where the first block of averaged data is skipped as equilibration data and the rest is presented as a plot of the average with standard deviation:

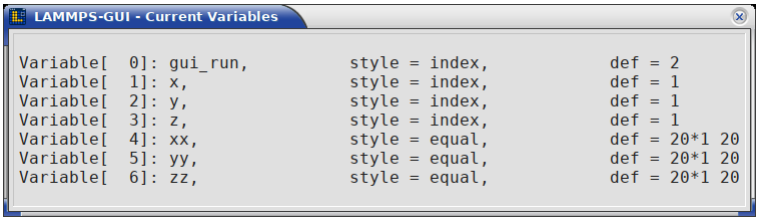
```
lattice         fcc 0.8442
region          box block 0 10 0 10 0 10
create_box      1 box
create_atoms    1 box
mass            1 1.0

velocity        all create 3.0 87287 loop geom

pair_style      lj/cut 2.5
pair_coeff      1 1 1.0 1.0 2.5

neighbor        0.3 bin
neigh_modify    every 20 delay 0 check no

fix             1 all nve

compute         rdf all rdf 200 1 1
fix             rdf all ave/time 100 10 1000 c_rdf[*] mode vector &
                   file melt_rdf.dat

compute         ke_atom all ke/atom
fix             hist_ke all ave/histo 100 10 1000  0.0 20.0 200  c_ke_atom &
                   file melt_histo_ke.dat mode vector

thermo_style    custom step temp pe press
thermo          1000
run             50000
```







Variable Info



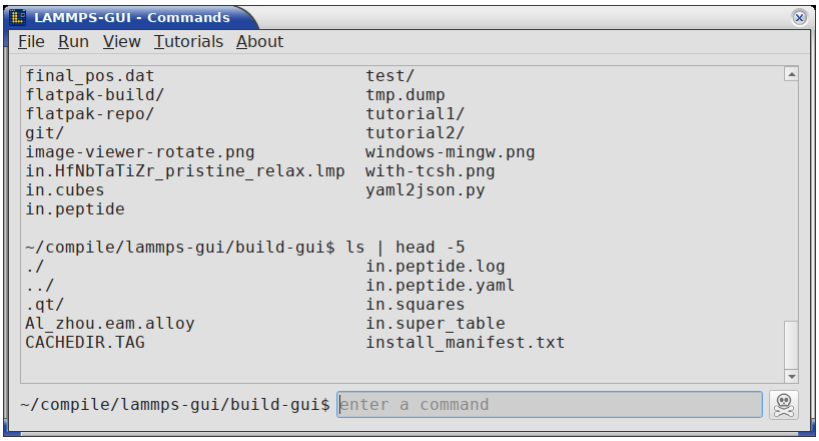
During a run, it may be of interest to monitor the value of input script variables, for example to monitor the progress of loops. This can be done by enabling the "Variables Window" in the *View* menu or by using the *Ctrl-Shift-W* keyboard shortcut. This shows info similar to the info variables command in a separate window as shown below.

Like for the *Output* and *Charts* windows, its content is continuously updated during a run. It will show "(none)" if there are no variables defined. Note that it is also possible to *set* index style variables, that would normally be set via command-line flags, via the "Set Variables…" dialog from the *Run* menu. LAMMPS-GUI automatically defines the variable "gui_run" to the current value of the run counter. That way it is possible to automatically record a separate log for each run attempt by using the command

```
log logfile-${gui_run}.txt
```

at the beginning of an input file. That would record logs to files `logfile-1.txt`, `logfile-2.txt`, and so on for successive runs.

Command window



The *Command window* is opened from the *Run* menu with *Open Command Window* or the *Ctrl-Shift-X* keyboard shortcut. It shows a shell prompt with a scrollback, for the ordinary work that surrounds a simulation: post-processing a dump file with a Python script, looking at what a run just wrote, calling a plotting tool, all without leaving LAMMPS-GUI.

Lines typed at the prompt are handed to a single shell process that is kept running between commands, so `cd`, `pushd`/`popd`, environment variables, and the rest of the shell state behave as they would in a terminal. The shell is started as an interactive one, so it reads the usual start-up file and the aliases and shell functions defined there are available – with one exception, noted below. The directory shown in front of the prompt follows the shell, however it was changed. A directory inside the home directory is shown with a leading ~, the way a shell prompt writes it, and hovering over it shows the path in full. The window starts in the directory of the current input file, which is where a run leaves its output, and stays independent afterwards: opening a different input file does not move a shell that may be busy. *File* > *Change to Input Directory* moves it to the current input file's directory on request. The up and down arrow keys walk through previously entered commands, which are remembered between sessions.

The *Tab* key completes the word it is in. On a word that starts a command – the first of the line, and equally the first after a `;`, `|`, `&&` or `||` – it offers whole lines entered before, sorted and without repeats, and then command names from the search path, so a few characters of a long command line bring the whole of it back, which is what a reverse history search is for. On any other word it offers the names in the directory the shell is in, with a `/` after the ones that are directories. Only that directory is offered, and only plain names: there is no completion across a path, and none of what the shell itself would complete, since the shell never sees the line until it is entered. The list is taken again whenever the shell changes directory, and each time completion starts on a new argument, so a file a command has just written is offered without reopening the panel.

When a single name matches, *Tab* completes it outright and no list appears. When several do, it offers them in a list; pressing *Tab* again walks through them and *Shift-Tab* walks back, with each entry appearing in the input line as it is reached. *Enter* takes the entry that is highlighted and goes no further, so the completed line is run by the next *Enter*; with nothing highlighted there is nothing to take and *Enter* runs the line as it stands.

The window adds three commands of its own, which hand files to LAMMPS-GUI rather than printing them, so that a whole project can be worked on without leaving the prompt. Each of them goes by two names, for the reason given below:

Command	Also	What it does with the files named after it
`open`	`gui-open`	Image and movie files go to a *slide show* viewer, all of the ones named in the same command together in one; any other file goes to a read-only text viewer.
`edit`	`gui-edit`	Loads the file into the editor, as *File* > *Open Input File* does – including the offer to save the current buffer first. The editor holds one file, so naming several says so and opens the first.
`plot`	`gui-plot`	Reads the file as a data file and asks which columns to draw, then opens the plot. With several files it asks for each in turn, and canceling stops the rest.

These are ordinary shell commands, so the shell expands their arguments before they run and wildcards, quoting, ~ and variables work there as they do anywhere else:

```
open melt-*.png                 # the whole sequence, in one slide show
edit in.melt                    # into the editor
plot log.lammps                 # pick columns, then plot
open $(ls -t *.png | head -1)   # the most recent image
gui-plot log.lammps             # the same as "plot", under its second name
```

The short name is defined only when nothing else on the system claims it, which is checked in the shell itself, so an alias or function from the start-up file counts as well as a program on the search path. On macOS `open` already exists and does much the same job, and GNU plotutils installs a `plot`; where that is the case the command that was there keeps working and only the second name is added. That is what the second name is for: nothing else is likely to be called `gui-open`, so it is defined whatever else is on the system, and a line that uses it works the same way on every machine. Both names do the same thing – the work itself is in a function called `__lgui_show`, `__lgui_edit` or `__lgui_plot`, which is what the names lead to and what turns up in a listing of what the shell has defined. None of this is available with `cmd.exe`, which has no way to define them.

The shell is the one selected in the *Preferences* dialog (*Command window shell*, offering the shells installed on the machine). Until one is selected there, it is the one named by the `SHELL` environment variable on Unix-like systems (falling back to `/bin/bash` and then `/bin/sh`) and by `COMSPEC` on Windows. Commands run with `TERM` set to `dumb` and with `PYTHONUNBUFFERED` set, so that the output of a Python script appears as it is produced rather than all at once when it exits. On macOS, where an application launched from the Finder inherits a minimal `PATH`, the common package manager locations (Homebrew, MacPorts) are appended to it, for the shell and for command completion alike. `COLUMNS` and `LINES` are set to the size of the panel and follow it as it is resized, so a program that formats its output to a width uses the width that is actually there rather than the 80 columns it would otherwise assume.

File > *Command Aliases…* lists aliases that are defined in every shell this window starts. Two things make them worth having. A section of the start-up file guarded by a test for a terminal never runs here, as described below, and on several distributions that is where `ls` and `ll` are defined. Programs also drop formatting they keep only for a terminal: `ls` lists one entry per line rather than in columns, because that is what it is required to do when its output is not a terminal. The list therefore starts out with

Alias	Expands to
`ls`	`ls -aCF`
`ll`	`ls -laCF`

which restores the multi-column, classified listing a terminal would have produced. Rows can be added, changed or removed, *Restore Defaults* puts the two back, and a change applies to the shell that is already running as well as to later ones. Aliases are not available with `cmd.exe`, which has no equivalent.

A command runs in the foreground and holds the shell until it finishes, exactly as it would in a terminal. Append & to start a program – a graphical one in particular – without waiting for it.

While a command is running the prompt says `running >` and refuses input. A line entered then could not be a command waiting its turn: it would go down the same pipe and be read by the running program, if it reads at all, and by the shell only once that program had finished.

There is no `Ctrl-Z` followed by `bg` to fall back on. Job control needs a controlling terminal, and there is none here, so the shell has no list of jobs for `bg` to act on and reports as much when it starts. To recover from a program started without `&` there are two choices. *File* > *Restart Shell* ends the shell and starts a fresh one in the same directory while whatever the shell had started keeps running, so a graphical application stays open and the prompt comes back – the outcome `Ctrl-Z` and `bg` would have produced. The kill button below ends the program itself.

The button with the skull at the right of the prompt, and *File* > *Kill Command*, end the running command outright. They ask it to quit and insist a moment later if it has not, so the shell becomes free again and the prompt returns; the button is only active while something is running. A command that started programs of its own leaves those behind, since without job control there is no group of processes to end in one go.

File > *Interrupt Command* is the gentler option and sends an interrupt instead. It is a best effort: without job control the shell starts its children with the interrupt signal ignored, so a program that does not install a handler of its own will sit through it.

It is also the way out of an unfinished multi-line construct. A line such as `if true; then` with nothing after it leaves the shell waiting for the rest of it, and what the shell reads next as part of that construct is the marker this window ends every command with – so the command never appears to finish, and since the prompt refuses input while a command runs, the closing `fi` cannot be typed either. Interrupting puts the shell back at a prompt and asks it for a fresh marker, which brings the prompt back with the session intact. Several commands on one line separated by `;` are otherwise nothing special: they run as they would in a terminal, and the exit status reported is the one of the last of them.

On Windows, neither killing nor interrupting a command is supported; *File* > *Restart Shell* is the way to get the prompt back there.

Output may only appear when a program exits

A program that writes to a terminal usually flushes each line, but when its output is a pipe – as it is here – the C runtime collects it into blocks instead and writes them out when the buffer fills or the program exits. Nothing is lost, but a long-running program can appear silent until it is done. Python is handled already, through `PYTHONUNBUFFERED`; for other programs, `stdbuf -oL` in front of the command asks for line buffering where that tool is available.

Start-up file sections that require a terminal are skipped

The shell reads the start-up file, but a section of it guarded by a test for a terminal, such as `[ ! -t 0 ] && return`, stops there, because the shell is reading a pipe and not a terminal. Most aliases and functions are unaffected; the ones defined in such a section are missing, which is confusing precisely because everything around them is there. On Fedora and related distributions this is where `ls`, `ll` and `l.` are defined, so those three are absent while the rest of the aliases are present. *File* > *Command Aliases…* is where to put them back, and it starts out holding exactly those.

This is NOT a terminal emulator

There is no pseudo terminal behind the prompt, only a pipe. Programs that need a real terminal – editors, pagers, anything using curses, anything asking for a password – will either report that the terminal is insufficient or misbehave. A program cannot be fed from the prompt either: input is refused while a command is running, precisely so that a line meant for the shell is not swallowed by whatever it started.

4.5 Visualization

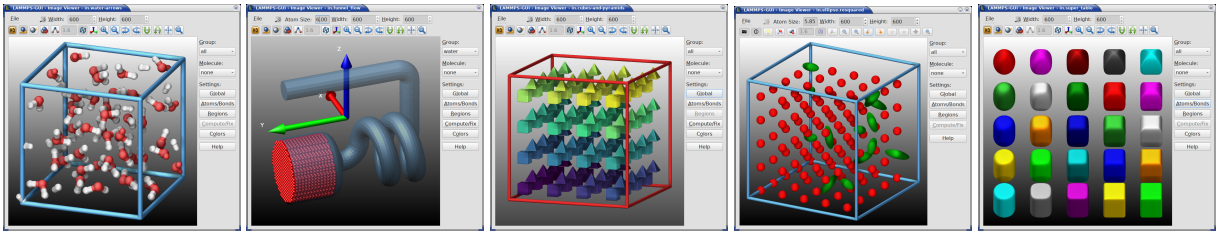
Snapshot Image Viewer

By selecting the *Create Image* entry in the *Run* menu, or by hitting the *Ctrl-I* (*Command-I* on macOS) keyboard shortcut, or by clicking on the "palette" button in the status bar of the *Editor window*, LAMMPS-GUI sends a custom write_dump image command to LAMMPS and reads back the resulting snapshot image with the current state of the system into an image viewer. This functionality is *not* available *during* an ongoing run. In case LAMMPS is not yet initialized, LAMMPS-GUI tries to identify the line with the first run or minimize command and execute all commands in the editor up to that line, and then executes a "run 0" command. This initializes the system so an image of the initial state of the system that can be rendered. If there was an error in that process, a dialog with the error message will appear.

Automatic settings

When possible, LAMMPS-GUI tries to detect which elements the atoms correspond to (via their mass) and then colorizes them in the image and sets their atom diameters accordingly. If this is not possible – for instance when using reduced (= 'lj') units – then LAMMPS-GUI will check the current pair style and if it is a Lennard-Jones type potential, it will extract the *sigma* parameter for each atom type and assign atom diameters from those numbers. When using an atom style where the atom diameters are set directly on a per-atom basis, LAMMPS will use that value. For cases where atom diameters are not auto-detected or you want to override the choice, you can configure it in the *Atom/Bond* settings dialog (see below). The default value is inferred from the x-direction lattice spacing.

For particles that use atom styles "body", "ellipsoid", "line", or "tri" LAMMPS will visualize the particles according to their atom style information by default. Other particle types will be visualized as spheres. In the *Atom/Bond* settings dialog, this can be further customized (or disabled).



It is also possible to visualize regions, graphics from computes and fixes, and have bonds computed dynamically for potentials, where the bonds are determined implicitly (like AIREBO). Please see the documentation of the dump image command for more details on these and other features and the LAMMPS Visualization Howto for more general discussions on how to generate advanced visualizations with LAMMPS directly.

If elements cannot be detected, the default sequence of colors of the dump image command is assigned to the different atom types.

Customizations

The Image Viewer controls described below support significant customization of the default visualization through the toolbar buttons, editable text fields, and additional dialogs. This covers a wide variety of possible customizations. After each change is applied, LAMMPS-GUI will have LAMMPS re-create the displayed image with the updated settings. The resulting image can be saved or copied to the clipboard and pasted into a compatible application.

Delays in updating the image

Some visualization settings, especially enabling SSAO and FSAA (see below) or massively enlarging the image size, can significantly increase the time required for LAMMPS to render the image. While in most cases the image

will be updated in a fraction of a second, complex visualizations may take multiple seconds. While LAMMPS is rendering an updated image, the small color palette icon in the menu bar is colored and will be grayed out when rendering is complete.

For further customization or making the visualization available when running the simulation with LAMMPS directly (e.g. when running on a cluster after enlarging the system), you can copy the current dump image and dump_modify commands to the clipboard so they can be pasted into a LAMMPS input file in either the included *text editor window* or some other text editor and adjusted according to the documentation.

The resulting images will be shown automatically in the *slide show viewer* when running the simulation with the thus modified input from LAMMPS-GUI. This strategy has been used to great effect to create many of the simulation snapshot images shown in this documentation and the LAMMPS manual.

Color customizations

LAMMPS-GUI uses two lists of colors. The first are the per-type atom colors that can be customized from a built-in initial assignment. This list is maintained by LAMMPS-GUI and information from them is passed to LAMMPS via LAMMPS commands while creating colors named "type#" where '#' is the atom type number. These color definitions can also be written to and loaded from *JSON files*. Recent LAMMPS versions have a dump_modify loadcolors and *savecolors* command that can read and write files in the same format. The second is the list of named colors that are maintained by the *dump image* command in LAMMPS. LAMMPS-GUI has the list of predefined color names and may define additional colors as needed, but will give them names that are specific to LAMMPS-GUI and does not attempt to overwrite any of the predefined colors.

Available controls

The Image Viewer window consists of three main areas: a menu/toolbar strip at the top, the rendered image in the center, and a settings panel on the right side. Following the general theme of LAMMPS-GUI of extensive keyboard shortcut support, you can select most text fields by using the *Alt* key and the underlined letter. For example the *File* menu is opened with *Alt-F* and its entries can be also selected the same way or using the cursor keys and *Enter*. Keyboard shortcuts starting with *Ctrl* usually work globally inside the image window, that is even when the corresponding menu item is not visible.

The **menu/toolbar strip** consists of two rows: the first row with the *File* menu, atom and bond size controls, image dimension controls, and a second row of toggle and action buttons.

The **menu bar row** has:

- **The File menu with the following entries:**
  - **Save As…** (*Ctrl-S*): Save the rendered image to a file. The file format is inferred from the file name extension. When the ImageMagick software is installed, additional file formats beyond those natively supported by the Qt library become available.
  - **Copy Image** (*Ctrl-C*): Copy the rendered image to the clipboard for pasting it into another application. This requires support from the receiving application, which is the case for many common applications like document editors and web browsers.
  - **Copy dump image command** (*Ctrl-D*): Copy the current dump image and dump_modify commands to the clipboard so they can be pasted into a LAMMPS input file in either the included *text editor window* or some other text editor. This allows the current visualization settings to be reproduced during a simulation run, including in the *slide show viewer*.

  - **Load Colors/Lights from JSON File...**: Load a list of definitions for per-type colors and settings for the four light sources from a *JSON format file*. The list of colors may contain either more or fewer definitions than the current system has atom types. In the latter case the colors "wrap around", that is colors are read from the list multiple times.
  - **Save Colors/Lights to JSON File...**: Save the currently used list of definitions for per-type colors and the current lighting settings to a *JSON format file*. The list may be loaded later to restore a previous color and lighting assignment, since these settings are reset when the Image Viewer dialog is restarted.
  - **Reset Colors**: Reset the list of per-type colors to a compiled in default list.
  - **Close** (*Ctrl-W*): Close the Image Viewer window.
  - **Quit** (*Ctrl-Q*): Quit the entire application.

- The **busy indicator**, a small palette icon that is colored while LAMMPS is rendering a new image and grayed out when rendering is complete.
- The **Atom size** text field, where the atom diameter can be adjusted. This field is only visible when the atom diameter is not automatically set.
- The **Bond size** text field, where the bond diameter can be adjusted. This field is only visible when the bond diameter is not automatically set and display of explicit or implicit bonds is enabled
- The **Width** spin box where the image width can be set. It can be accessed using the *Alt-W* keyboard shortcut.
- The **Height** spin box, where the image height can be set. It can be accessed using the *Alt-H* keyboard shortcut.

The **toolbar buttons** row below the menu bar provide quick access to several rendering options and view manipulations. From left to right there are:

- **SSAO** (toggle): Enable or disable Screen Space Ambient Occlusion rendering for a more spatial, depth-shaded appearance, at the expense of more CPU time.
- **Anti-aliasing** (toggle): Render the image at double resolution and scale down for smoother edges. Full Scene Anti-Aliasing (FSAA) produces higher quality images at the expense of more CPU time. It is particularly recommended in combination with any transparent objects.
- **Depth cueing** (toggle): Enable or disable depth cueing, which fades distant objects toward a fog color for a stronger sense of depth. The intensity, fog color, and start position are configured in the *Global image settings dialog*.
- **Defocus** (toggle): Enable or disable defocusing, which blurs distant objects as if the camera were focused on the front of the scene, while leaving their colors untouched. This can be combined with depth cueing. The intensity and start position are configured in the *Global image settings dialog*.
- **Shininess** (toggle): Switch between shiny and matte surface rendering of graphics objects like atoms and bonds.
- **VDW style** (toggle): Switch between space-filling (Van der Waals) sphere representation of atoms and the smaller ball-and-stick style of atoms and bonds.
- **Dynamic bonds** (toggle): Automatically compute bonds from atom distances. This is useful for force fields with implicit bonds. When enabled, the adjacent text field allows setting the bond cutoff distance. This feature depends on existing neighbor list data and thus may not always work as expected when the system has explicit bonds and thus neighbors may be automatically excluded from neighbor lists due to the special_bonds settings
- **Box** (toggle): Show or hide the simulation box drawn as colored cylinders.
- **Axes** (toggle): Show or hide the labeled coordinate axes arrows.
- **Zoom in / Zoom out**: Adjust the zoom level in 10 percent increments between 0.1x and 10.0x.
- **Rotate left / Rotate right**: Rotate the view horizontally by 10 degrees per click.
- **Rotate up / Rotate down**: Rotate the view vertically by 10 degrees per click.

- **Recenter**: Recenter the view on the center of mass of the currently selected group.
- **Reset**: Reset the view to the default orientation and zoom level.
- **Fit window**: Resize the window so the image is shown at its full size, without scroll bars or unused space. This undoes a manual resize of the window; the window is never grown beyond a fraction of the screen, so scroll bars remain for very large images.

The default image size, some default image quality settings, the view style and some colors can be changed in the *Preferences* dialog window. From the image viewer window further adjustments can be made: actual image size, high-quality (SSAO) rendering, anti-aliasing, view style, display of box or axes, zoom factor. The view of the system can be rotated horizontally and vertically.

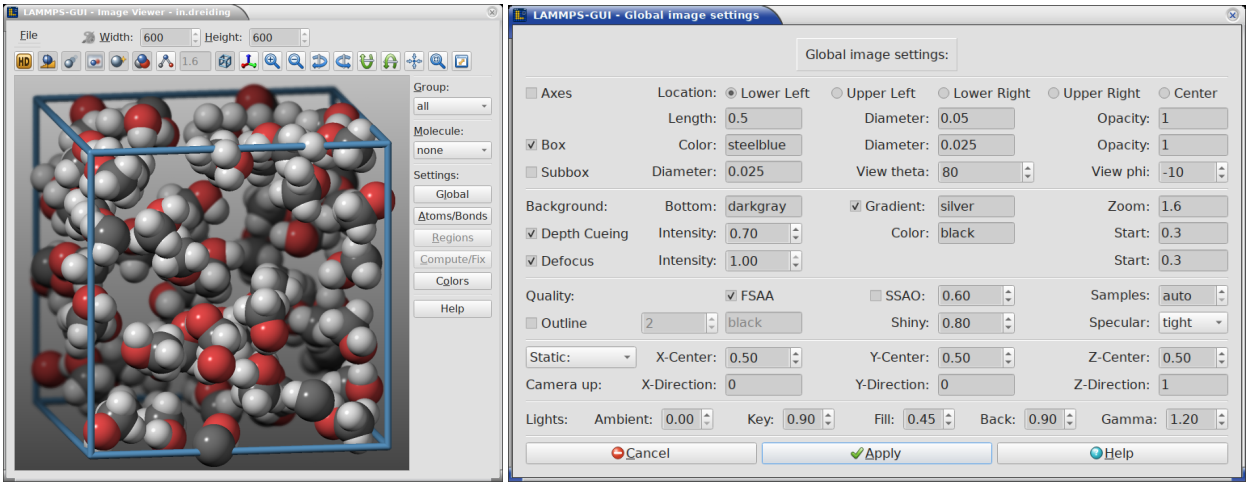
The **settings panel** on the right side of the window provides additional controls (most are explained in detail below):

- **Group**: A drop-down list to select which group of atoms to display (default is "all"). Only atoms belonging to the selected group are rendered.
- **Molecule**: A drop-down list to select a molecule to visualize (default is "none"). When a molecule is selected, it is shown at the center of the simulation box, and the group selection is disabled. Selecting "none" restores normal group-based display.
- **Global**: Opens the *Global image settings* dialog for fine-grained control of axes, box, background, quality, view, center, and camera settings.
- **Atoms/Bonds**: Opens the *Atom and bond settings* dialog for detailed atom, bond, VDW, and special atom style visualization options.
- **Regions**: Opens the *Region settings* dialog to configure visualization of regions defined in the simulation.
- **Compute/Fix**: Opens the *Compute and fix graphics* dialog to enable and configure extra graphics objects provided by selected compute and fix styles.
- **Colors**: Opens the *Atom Type Color Customization* dialog where the current list of colors used for per-type coloring can be customized and saved or loaded from a file.
- **Help**: Opens this online documentation page for the visualization features in LAMMPS-GUI in a web browser.

The image is re-rendered after each change to the buttons, text fields or settings dialogs, and when there are many atoms to render and high quality images with anti-aliasing are requested, re-rendering may take several seconds. Some time consuming rendering steps are multi-threaded, but there is no GPU acceleration.

Global image settings

While some persistent default settings for the image output can be configured in the "Snapshot Image" tab of the *Preferences dialog*, more fine-grained configuration is possible by opening the "Global image settings" dialog. However, settings not stored by the preferences are reset when the image viewer window is closed. This dialog is opened by pressing the "Global" button in the settings panel or by using the *Alt-L* keyboard shortcut. The settings in this dialog correspond to options of the LAMMPS dump image and dump_modify commands.



The dialog is organized into the following sections:

Axes

Controls the display of coordinate axes arrows in the image.

- **Axes** (checkbox): Enable or disable rendering of coordinate axes.
- **Location** (radio buttons): Select where the axes are drawn in the image. Possible choices are: *Lower Left* (default), *Lower Right*, *Upper Left*, *Upper Right*, or *Center*.
- **Length**: The length of the axes arrows as a fraction of the box size (range: 0.00001 – 5.0).
- **Diameter**: The diameter of the axes arrows as a fraction of the box size (range: 0.00001 – 5.0).
- **Opacity**: The transparency of the axes (range: 0.0 – 1.0, where 1.0 is fully opaque and 0.0 is fully transparent).

Box

Controls the display of the simulation box.

- **Box** (checkbox): Enable or disable rendering of the simulation box.
- **Color**: The color used to draw the box edges. Accepts named colors.
- **Diameter**: The diameter of the box edge sticks as fraction of the box size (range: 0.000001 – 5.0).
- **Opacity**: The transparency of the box edges (range: 0.0 – 1.0, where 1.0 is fully opaque and 0.0 is fully transparent)

Subbox

Controls the display of the per-processor sub-domain boxes (relevant for MPI parallel simulations, will coincide with the regular box for LAMMPS-GUI runs).

- **Subbox** (checkbox): Enable or disable rendering of the sub-domain box.
- **Diameter**: The diameter of the sub-domain box edge sticks as fraction of the box size (range: 0.00001 – 5.0).
- **View theta**: The viewing angle in degrees away from the positive z-axis (default: 60). Disabled for 2d systems, where LAMMPS always looks down the z-axis.
- **View phi**: The azimuthal viewing angle in degrees around the z-axis (default: 30). Disabled for 2d systems.

Background

Sets the background color(s) of the rendered image.

- **Bottomcolor**: The background color at the bottom of the image.

- **Topcolor**: The background color at the top of the image. If the two colors differ, a vertical gradient is applied from bottom to top.
- **Zoom**: The zoom factor of the view (range: 0.1 – 10.0, where values larger than 1.0 zoom in). This is the same setting that the zoom in/out buttons of the settings panel change in steps of 10 percent.

Depth Cueing

Controls depth cueing, which fades distant objects toward a fog color, similar to looking through fog. This is perceived as depth and helps to visually untangle dense systems.

- **Depth Cueing** (checkbox): Enable or disable depth cueing. This is the same setting that the depth cueing toolbar button toggles.
- **Intensity**: The strength of the fading (range: 0.0 – 1.0). At 1.0 the most distant objects blend completely into the fog color.
- **Color**: The fog color. Accepts named colors or "auto" (the default), which fades toward the background color, following the background gradient when one is set.
- **Start**: Where the fading starts, as a fraction of the simulation box along the view direction (0.0 = side nearest to the camera, 1.0 = far side), or "auto" (the default) to start at the nearest rendered object.

Defocus

Controls defocusing, which blurs distant objects as if the camera were focused on the front of the scene. Like depth cueing this draws the attention to the objects in front, but it leaves their colors untouched, and the two effects may be combined. There is no color setting for this effect, so that field is left empty.

- **Defocus** (checkbox): Enable or disable defocusing. This is the same setting that the defocus toolbar button toggles.
- **Intensity**: The strength of the blur (range: 0.0 – 1.0). At 1.0 the most distant objects are blurred over a radius of 1 percent of the image height. Blurring over much more than the size of the rendered particles turns them into a uniform haze, so moderate values usually give the best result.
- **Start**: Where the blurring starts, as a fraction of the simulation box along the view direction (0.0 = side nearest to the camera, 1.0 = far side), or "auto" (the default) to start at the nearest rendered object, which keeps the front of the scene sharp.

Quality

Controls rendering quality options.

- **FSAA** (checkbox): Enable or disable full-scene anti-aliasing.
- **SSAO** (checkbox): Enable or disable Screen Space Ambient Occlusion for depth-shaded rendering.
- **SSAO strength**: The strength of the SSAO effect (range: 0.0 – 1.0).
- **Samples**: The number of SSAO sampling directions (range: 4 – 64). More samples produce smoother shading; fewer samples render proportionally faster. With "auto" (the default) the number is derived from the SSAO strength in the copied `dump image` command, while interactive renders in the Image Viewer use a reduced, fixed sample count to stay responsive. An explicitly set number applies to both, so a high-quality image can be saved directly from the viewer without going through a simulation run.
- **Outline** (checkbox): Enable or disable drawing outlines along the visible edges of rendered objects for a flat, illustration-like appearance. The two fields to the right of the checkbox set the width of the outlines in pixels (range: 1 – 16) and their color.
- **Shiny**: The shininess factor for surface rendering (range: 0.0 – 1.0, where 0.0 is matte and 1.0 is fully shiny).
- **Specular**: The width of the specular highlights: *auto* (the default, derived from the shiny factor), *none* (highlights off for a matte appearance), *wide*, *narrow*, or *tight* (small sharp highlights with a plastic-like appearance).

Metal Effect

Controls metallic shading, which renders objects as if they were made of metal rather than of colored plastic. Metal reflects light back directly and colors it in the process, so the rendering approximates the surroundings with a bright sky above and a dark ground below; a dark background makes the effect look the most convincing. The color assigned to an object tints its reflections.

- **Metal Effect** (checkbox): Enable or disable metallic shading.
- **Intensity**: How metallic the objects appear (range: 0.0 – 1.0). At 1.0 the objects are rendered as bare metal, smaller values blend toward the default appearance.
- **Metal Finish**: The surface finish of metallic objects: *satin* (the default) has a broad soft sheen and resembles brushed metal, *polished* concentrates the sheen into a narrower streak, and *mirror* reflects the surroundings the way a curved mirror does, making spheres look like polished ball bearings.

Center

Adjusts the center point of the rendered view. The drop-down list selects between **Center (static)**, where the center fractions refer to the simulation box, and **Center (dynamic)**, where they refer to the bounding box of the currently displayed atoms in each frame. The dynamic setting is mainly useful for the copied `dump image` command when creating a movie of a system that drifts or expands.

- **X-direction**, **Y-direction**, **Z-direction**: Fractional coordinates (range: 0.0 – 1.0) specifying the center of the view relative to the simulation box (static) or the bounding box of the displayed atoms (dynamic).

Camera up

Sets the direction that points up in the rendered image.

- **X-direction**, **Y-direction**, **Z-direction**: The components of the camera's up vector. The vector does not need to be normalized, but it must not be all zeros, or the values are ignored. The default is 0 0 1 for 3d systems and 0 1 0 for 2d systems, where the Z-direction entry is disabled.

Lighting

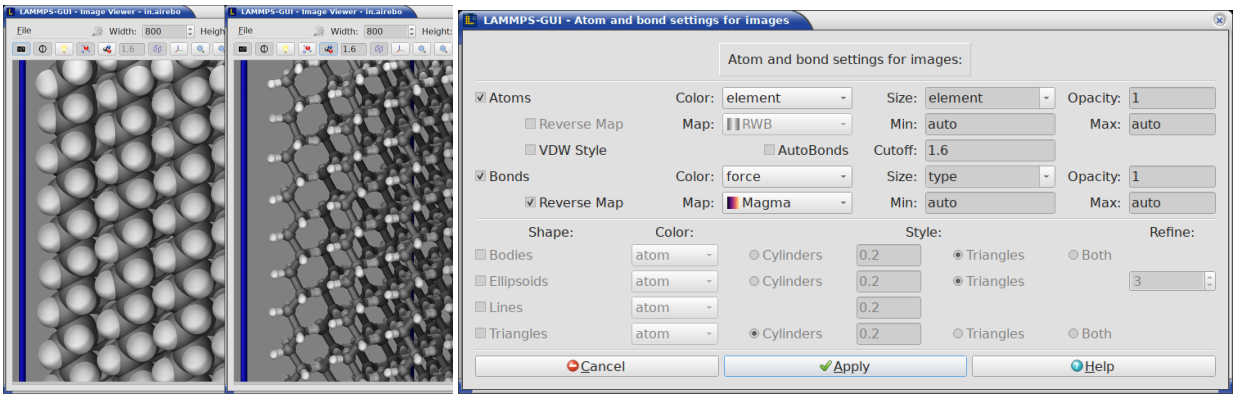
Adjusts the settings for the four light sources used in the rendering and the tonal balance of the result. The intensity of each light source is a floating point number in the range 0.0 – 1.0.

- **Ambient**: The intensity of the uniform, non-directional base lighting that illuminates all parts of the scene equally.
- **Key**: The intensity of the primary, directional light source that provides the main illumination and highlights.
- **Fill**: The intensity of the secondary directional light source that fills in the shadows created by the key light.
- **Back**: The intensity of the tertiary directional light source that illuminates the back of the objects, helping to separate them from the background.
- **Gamma**: The gamma adjustment applied to the summed up light contributions of the rendered objects (range: 0.1 – 10.0, default 1.0 = unchanged). Values above 1.0 lighten the image, most strongly in the darker regions, and thus can bring out shading detail on the dimly lit side of objects; values below 1.0 darken the image and increase its contrast. The background colors are not affected.

Press **Apply** to apply the current settings and re-render the image, or **Cancel** to discard changes. The **Help** button opens the LAMMPS dump image documentation.

Atoms/bonds settings

This dialog offers more detailed customizations for atom and bond visualization that are not directly accessible from the main Image Viewer toolbar. It is opened by pressing the "Atoms/Bonds" button in the settings panel or by using the *Alt-A* keyboard shortcut.



The dialog contains the following sections:

Atoms

Controls how atoms are rendered.

- **Atoms** (checkbox): Enable or disable rendering of atoms.
- **Color**: Select the per-atom property used for coloring. Options include *type*, *element* (if detected), *mol*, *q* (charge), *diameter*, *id*, *mass*, *x*/*y*/*z* (coordinates), as well as atom-style variables (*v_<name>*), per-atom compute results (*c_<name>* or *c_<name>[col]*), and per-atom fix results (*f_<name>* or *f_<name>[col]*). The contents of the list depends on which are available in the current simulation state. When selecting *type* or *element* the colors are fixed and can be only changed manually for `dump_image` output in the input using `dump_modify acolor` or `dump_modify element`, respectively. Otherwise, the colors are determined by the color map selection described below.
- **Size**: Select the property used for atom sizing. Options include *auto* (when element, diameter, or sigma data is available), *type*, *element*, atom-style variables (*v_<name>*) defined in the current simulation state, and a few pre-defined choices for custom atom diameters. The text field can be edited and a different custom diameter entered. Selecting an atom-style variable uses its per-atom value directly as the atom diameter; this allows, for example, to apply a scaling factor to a per-atom property. Per-atom data from computes and fixes are not offered directly, but they can be referenced from an atom-style variable, which then also allows to scale them to suitable diameters.
- **Opacity**: The transparency of atoms (range: 0.0 – 1.0, where 1.0 is fully opaque and 0.0 is fully transparent). Bonds have their own Opacity setting in the **Bonds** section below.
- **Map**: Select the color map used for coloring by a per-atom property. This option is *not* available for atom color selections *type* and *element*. Currently available continuous color maps are: *RWB* (red-white-blue), *PWT* (purple-white-teal), *BWG* (blue-white-green), *BGR* (blue-green-red), *Grayscale* (black-white), *Viridis* (from matplotlib), *Plasma* (from matplotlib), *Inferno* (from matplotlib), *Magma* (from matplotlib), *Cividis* (from matplotlib), *Turbo* (from matplotlib), *Teal*, and *Rainbow*. *Sequential*, *Landscape*, and *Basic* are maps with discrete colors. These are pre-defined color map settings and currently cannot be adjusted from LAMMPS-GUI directly. As for *all* image settings, further customizations can be realized by copying the dump image command line as customized by the Image Viewer to the editor and then run LAMMPS and observe the resulting images in the Slide Show window. Then the color map setting can be fully customized according to the dump_modify colormap documentation.
- **Reverse** (checkbox): Mirror the selected color map so its low and high ends are swapped (for example, *RWB* becomes blue-white-red). This replaces the former *BWR* entry, which is exactly *RWB* reversed. Enabled together with the **Map** selector.
- **Min / Max**: Set the range of the color map. Use *auto* to have LAMMPS determine the range automatically or specify an explicit numeric value.
- **VDW style** (checkbox): Enable or disable space-filling sphere rendering. When unchecked, the ball-and-stick style is used. This toggle shares a line with the **AutoBonds** control of the **Bonds** section below; the

two are mutually exclusive.

The color maps available for coloring atoms and bonds by value are shown below; the continuous maps are interpolated between color stops, while *Sequential*, *Landscape*, and *Basic* use discrete colors.

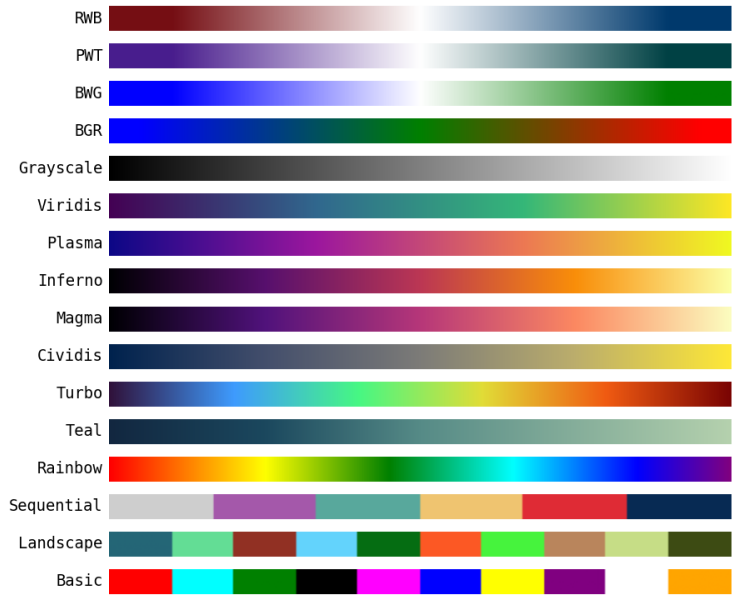


Fig. 4: The dump-image color maps offered for coloring atoms and bonds by value.

These color maps are defined in a single table in the C++ source, which makes them simple to add or modify; see *Adding or modifying a color map* in the Programmer's Guide for step-by-step instructions.

Bonds

Controls bond visualization.

- **Bonds** (checkbox): Enable or disable bond rendering. This option is only available when the atom style supports explicit bonds.
- **Color**: Select the bond coloring mode. The basic choices are *atom* (each bond half is colored by the atom type at its end) and *type* (a uniform color per bond type). The list also offers a set of per-bond properties computed by `compute bond/local` – *dist*, *dx*, *dy*, *dz*, *engpot*, *force*, *fx*, *fy*, *fz*, *engvib*, *engrot*, *engtrans*, *omega*, and *velvib*. Selecting one of these colors the bonds by that per-bond value using the bond color map (see **Map** below); LAMMPS-GUI creates the required `compute bond/local` automatically.
- **Size**: Select bond diameter mode. Options include *atom*, *type*, and a few pre-defined choices for custom bond diameters. The text field can be edited and a different custom diameter entered.
- **Opacity**: The transparency of bonds (range: 0.0 – 1.0), set independently from the atom opacity.
- **AutoBonds** (checkbox): Automatically determine bonds from atom distances, useful for many-body force fields with implicit bonds like AIREBO or Tersoff. This feature depends on existing neighbor list data and thus may not always work as expected when the system has explicit bonds and thus neighbors may be automatically excluded from neighbor lists due to the special_bonds settings
- **Cutoff**: The distance cutoff used for automatic bond detection (range: 0.001 – 10.0 in distance units), in the text field next to the AutoBonds checkbox. Only available when auto-bonds are enabled.
- **Reverse / Map / Min / Max**: Select the color map and value range used when coloring bonds by a per-bond value (see **Color** above). The same color maps as the atom **Map** are offered, and the **Reverse** checkbox mirrors the chosen map exactly like its atom counterpart. These fields are only enabled when a per-bond

property is selected as the bond color. Use *auto* for **Min / Max** to let LAMMPS determine the range automatically.

Bodies

Controls visualization of body particles (when present in the simulation).

- **Bodies** (checkbox): Enable or disable rendering of body particle shapes. When disabled, the particles are rendered as spheres like regular atoms.
- **Color** (selection): Use coloring by the *atom* color choice, the body *index*, or the atom *type* of the body particles.
- **Style** (radio buttons): Select the body rendering style – *Cylinders*, *Triangles*, or *Both*. For cylinders – when used for body particle rendering – their diameter can also be set (range: 0.1 – 10.0).

Ellipsoids

Controls visualization of aspherical particles (when present in the simulation). Particles flagged as ellipsoids are represented as a triangle mesh, others as spheres.

- **Ellipsoids** (checkbox): Enable or disable rendering of ellipsoid particle shapes. When disabled, the particles are rendered as spheres like regular atoms.
- **Color** (selection): Use coloring by the *atom* color choice, the ellipsoid *index*, or the atom *type* of the ellipsoid particles.
- **Style** (radio buttons): Select the ellipsoid rendering style – *Cylinders*, *Triangles*, or *Both*. For cylinders – when used for ellipsoid particle rendering – their diameter can also be set (range: 0.1 – 10.0).
- **Refine** (spinbox): Level of triangle mesh refinement. At level 1 the ellipsoids are represented by a deformed octahedron. With a level increase, each triangle is replaced by 4 triangles following the ellipsoid shape more closely (max: 6). At the maximum level each ellipsoid is represented by 8192 triangles. At high refinement level, there may be artifacts from rounding due to limitations of the image rasterizer included in LAMMPS. These can be made less prominent by enabling anti-aliasing.

Lines

Controls visualization of line segment particles (when present in the simulation).

- **Lines** (checkbox): Enable or disable rendering of line segment particle shapes as connected cylinders. When disabled, the particles are rendered as spheres like regular atoms. Also the cylinder diameter can be set (range: 0.1 – 10.0).
- **Color** (selection): Use coloring by the *atom* color choice, the line *index*, or the atom *type* of the line particles.

Triangles

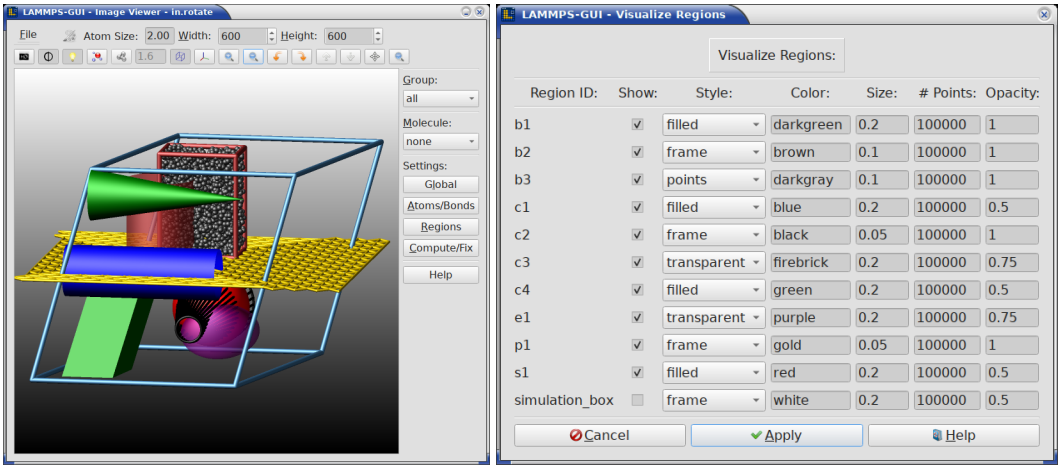
Controls visualization of triangulated particles (when present in the simulation).

- **Triangles** (checkbox): Enable or disable rendering of triangulated particle shapes. When disabled, the particles are rendered as spheres like regular atoms.
- **Color** (selection): Use coloring by the *atom* color choice, the triangulated particle *index*, or the atom *type* of the triangulated particles.
- **Style** (radio buttons): Select the particle rendering style – *Cylinders*, *Triangles*, or *Both*. For cylinders – when used for triangle particle rendering – their diameter can also be set (range: 0.1 – 10.0).

Press **Apply** to apply the settings and re-render the image, or **Cancel** to discard changes. The **Help** button opens the LAMMPS dump image documentation.

Region settings

This dialog allows enabling and configuring the visualization of regions defined in the LAMMPS input script. It is opened by pressing the "Regions" button in the settings panel or by using the *Alt-R* keyboard shortcut. The dialog only appears when at least one region is defined in the current simulation.



For each region, the following settings can be adjusted:

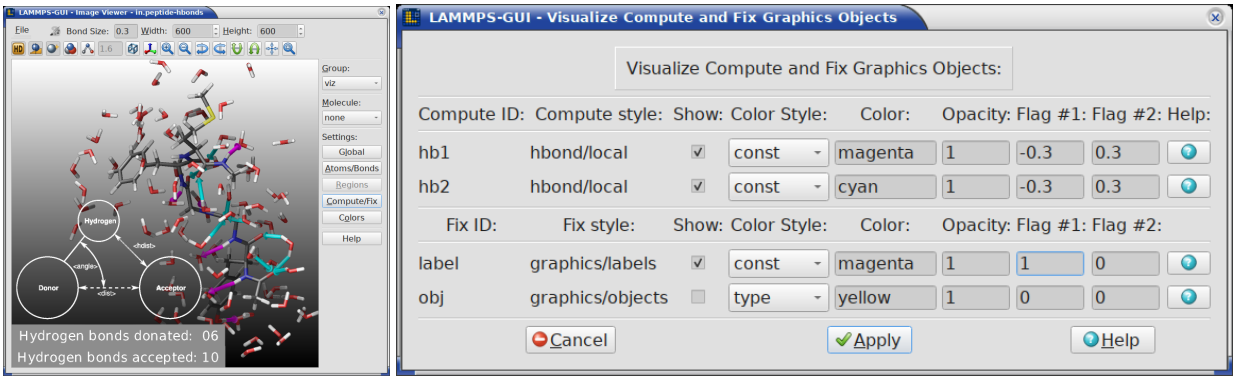
- **Region ID**: The identifier of the region (read-only).
- **Show** (checkbox): Enable or disable visualization of this region.
- **Style**: The rendering style for the region surface – *frame* (wireframe), *filled* (solid), *transparent* (see-through solid), or *points* (point cloud).
- **Color**: The color used to render the region. Accepts named colors or hex color values.
- **Size**: The diameter of the lines (for frame style) or points (for points style).
- **# Points**: The number of points used to approximate the region volume (range: 100 – 1,000,000). Higher values reproduce the volume better, but may obscure other details of the image.
- **Opacity**: The transparency of the region rendering (range: 0.0 – 1.0, where 1.0 is fully opaque and 0.0 fully transparent).

Press **Apply** to apply the settings and re-render the image, or **Cancel** to discard changes. The **Help** button opens the LAMMPS visualization howto documentation and jumps to the section discussing visualizing regions.

Graphics from computes and fixes

Some compute and fix styles can prepare lists of graphics objects for inclusion into visualizations generated by the dump image command. This command is used internally by LAMMPS-GUI to create the snapshot image.

The "Visualize Compute and Fix Graphics Objects" dialog allows enabling these graphics objects and adjusting their settings. The dialog is opened by pressing the "Compute/Fix" button in the settings panel or by using the *Alt-C* keyboard shortcut. The dialog only appears when at least one compute or fix with graphics capabilities is defined.



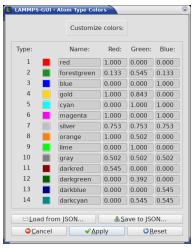
For each compute or fix that supports graphics output, the following settings can be adjusted:

- **Compute/Fix ID**: The identifier of the compute or fix (read-only).
- **Style**: The compute or fix style name (read-only).
- **Show** (checkbox): Enable or disable visualization of the graphics objects from this compute or fix.
- **Color Style**: Select the coloring mode – *type*, *element*, or *const* (constant color).
- **Color**: The color to use when *const* color style is selected. Accepts named colors or hex color values.
- **Opacity**: The opacity of the graphics objects (range: 0.0 – 1.0).
- **Flag #1 / Flag #2**: Style-specific numeric flags that control additional rendering options. Their meaning depends on the specific compute or fix style.

Each row also has a **Help** button that opens the LAMMPS documentation page for the corresponding compute or fix style, jumping directly to the "Dump image info" section that describes the available graphics objects and flag settings.

Press **Apply** to apply the settings and re-render the image, or **Cancel** to discard changes. The general **Help** button at the bottom opens the LAMMPS dump image documentation.

Atom Type Color Customization



This dialog allows customizing the current color definitions used for per-type coloring, reset them to the default settings, and load or save them using JSON format files.

The dialog contains as many color rows as the current system has atom types and it is initialized from the current list of colors. If there are fewer types, then only the first part of that list is used. If there are more types, then the list of colors is used multiple times and wraps around. When the list is too long to fit into the dialog window, it can be scrolled up and down as needed.

The changes are applied to the image only after the "Apply" button is clicked and the dialog closed. At this step, the list of colors is updated with the colors in the dialog and expanded, if needed. When the "Cancel" button is pressed, the edits are discarded and the original list of colors retained. Clicking on the "Reset" button will reset the list of colors to its default values.

With the "Load from JSON" button a list of definitions for per-type colors is loaded from a JSON format file. The list may contain either more or fewer definitions than the current system has atom types. The "Save to JSON" button instead saves the edited list of definitions to a file. The list may be loaded later to restore a previous color assignment.

JSON file format for colors and lighting definitions

The JSON format file for color and lighting definitions has the following structure and is compatible with the format used by the LAMMPS commands dump_modify savecolors and dump_modify loadcolors. The file can be validated with the JSON schema file at https://download.lammps.org/json/color-schema.json.

The "application", "format", "revision" entries are *required* and are checked for on reading so that files without them are rejected. Also the "colors" list is *required* with color definitions of three entries each: "red", "green", and "blue" that have the value of the corresponding color component given as a floating point number in the range from 0.0 to 1.0 inclusive. The "lights" object is *optional* (LAMMPS-GUI will display a warning if it is missing) and contains the intensity settings for the four light sources: "ambient", "key", "fill", and "back" also in the range from 0.0 to 1.0 inclusive.

Here is an example with just two colors (red and green) and default lighting settings:

```
{
    "application": "LAMMPS",
    "format": "colors",
    "revision": 1,
    "schema": "https://download.lammps.org/json/color-schema.json",
    "colors": [
       {
           "blue": 0,
           "green": 0,
           "red": 0.9
       },
       {
           "blue": 0,
           "green": 0.9,
           "red": 0
       }
    ],
    "lights": {
       "ambient": 0.2,
       "back": 0.4,
       "fill": 0.4,
       "key": 0.4
    }
}
```

Image Slide Show

When running a LAMMPS input containing a dump image command with LAMMPS-GUI, the "Slide Show" window opens to load and display the images created by LAMMPS as they are written. This is a convenient way to visually monitor the progress of the simulation. It also can be used as an effective way to refine visualizations created with the *Snapshot Image Viewer*.

Warning

When two or more `dump image` commands are active at the same time, the slide show picks up the images from all of them and displays them interleaved in the order they are written. This is usually not intended, but cannot be detected by LAMMPS-GUI before the run has started and the images have already been mixed. To avoid it, make sure that only one `dump image` command is active at any time during a run, for example by removing a no longer needed dump with an undump command.

The same window can also display existing image files that were not created by the current session: select one or more files with *File -> View Image or Movie File(s)...* (see *the File menu*) to review images produced by an external (for example large parallel) simulation, or to revisit images from an earlier run without rerunning it. Image formats that Qt cannot read natively are converted on demand with ImageMagick if it is available. Each such file is converted only once and the converted copy is reused while the window is open, so displaying it repeatedly neither repeats the conversion nor repeats any complaint its format may provoke from Qt (the Targa/TGA decoder is a common source of those). A file that can be read by neither is reported once on the console and then skipped. When the slide show is opened this way, the controls that act on a running simulation (such as stopping the run or sending images to the trash) are hidden.

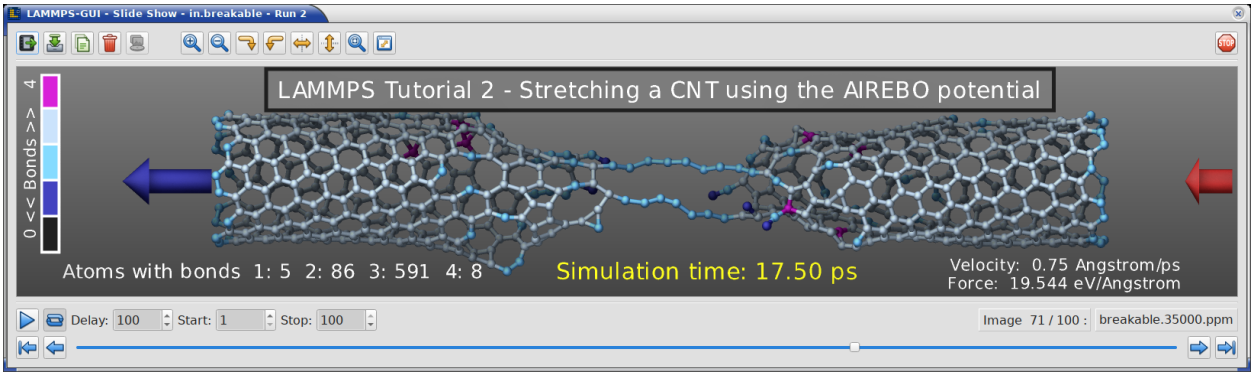
To load many imagefiles from the command line, you can use on Linux and macOS a command line like the following:

```
lammps-gui $(for f in image-*.png; do echo -n " -i $f"; done)
```

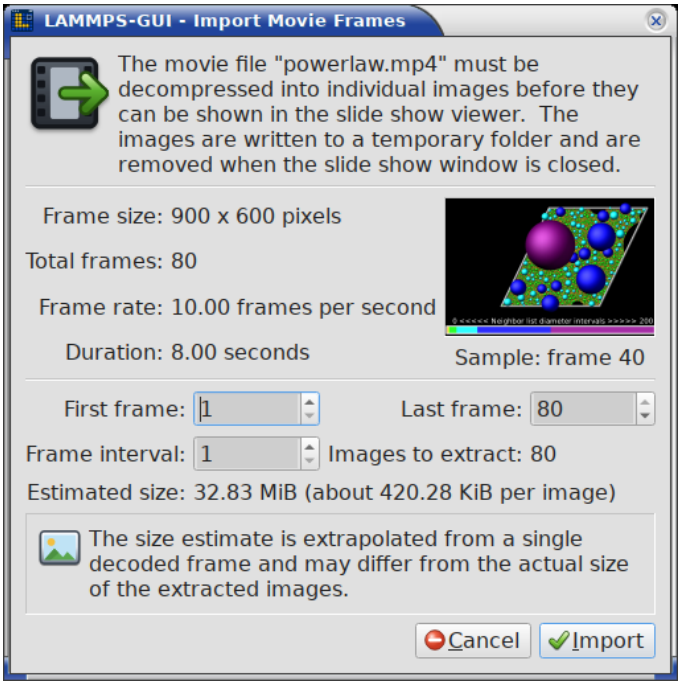
Movie files can be selected in the same dialog; their frames are then extracted into individual images as described in *Importing movie files* below.

From the slide show window the following global keyboard shortcuts are supported: *Ctrl-W*: close window, *Ctrl-Q*: quit application, *Ctrl-/*: stop running simulation. Other keyboard shortcuts are connected to some of the controls and listed in their documentation below.

The visualization shown below is created by the in.breakable GRAPHICS package example



Importing movie files



Movie files (`.mp4`, `.mkv`, `.webm`, `.avi`, `.mov`, and so on, as well as animated GIF files) can be opened with *File -> View Image or Movie File(s)...* just like image files. Since the slide show viewer displays individual images, the frames of a movie must first be decompressed into a sequence of image files. This requires the FFmpeg programs `ffmpeg` and `ffprobe`; it is the inverse of the movie export described below.

When a movie file is selected, a dialog reports its properties and asks for confirmation before any frames are extracted:

- **First frame** and **Last frame** select the range of the movie to extract.
- **Frame interval** thins out that range: an interval of 1 extracts every frame, 2 every other frame, and so on. This is useful to skim a long movie without decompressing all of it.
- **Estimated size** is how much temporary disk space the extracted images are expected to need. It is obtained by decoding a single sample frame near the middle of the selected range and multiplying its size by the number of selected frames, so it is an approximation. The sample frame is shown as a thumbnail next to the movie properties. When the middle of the selected range moves away from the sampled frame by more than a tenth of the movie, a new sample frame is decoded after a moment, and the thumbnail and the estimate are refreshed from it. A highlighted warning appears when the estimate exceeds one gigabyte, when it would use up most of the free space on the volume holding the temporary folder, or when more than 1000 images would be extracted.

Because the frames are stored as individual images and not as a compressed video stream, they usually take up substantially more space than the movie file itself. The extracted frames are written to a temporary folder and are deleted again when the slide show window is closed. Below the navigation slider each extracted image is labeled with the name of the movie and its frame number in it.

Slide show controls

There are controls and displays above and below the image. If you are uncertain about the function of a specific button, you can place the cursor on top of it and a descriptive tooltip will appear.

The **toolbar** at the top of the Slide Show window provides the following controls, organized from left to right:

- **Export to movie** (*Ctrl-E*): Export the active range of images (see the **Start** and **Stop** controls below) to a movie file or animated GIF file. This requires that either the FFmpeg program or the ImageMagick software are installed.

Supported output formats include MP4, MKV, AVI, MPG, MPEG, WEBM, and animated GIF. The file format is determined by the file name extension. Any active image transformations (rotation, mirroring, see below) are applied to the exported movie.

- **Save current image** (*Ctrl-S*): Save the currently displayed image to a file, including any applied transformations (rotation, mirroring, see below). The file format is inferred from the file name extension. When the ImageMagick software is installed, additional file formats beyond those natively supported by the Qt library become available.
- **Copy to clipboard** (*Ctrl-C*): Copy the current image to the system clipboard for pasting it into another application. This requires support from the receiving application, which is the case for many common applications like document editors and web browsers.
- **Delete selected images**: Remove the image files in the currently selected range (see the **Start** and **Stop** controls below) from disk. A confirmation dialog reports how many files will be removed before they are deleted. With the default range (first to last image) this removes the entire sequence. Since the number of image files can be large for long simulations, this provides a safe way to clean up the working directory without risk of accidentally deleting other files. This will, however, only delete images of the last run. If that was stopped before completion or the output filename has changed, older images created by previous runs will not be deleted. Deleting an image also discards its converted copy from the image cache described next.
- **Image cache**: An indicator that is grayed out while the image cache is empty and shown in color once it holds anything. Its tooltip reports how many converted images and extracted movie frames are cached and how much temporary disk space they occupy. Pressing it discards the converted images after a confirmation; they are converted again the next time they are displayed, so nothing is lost but time. Extracted movie frames are never discarded this way, since re-creating them requires running FFmpeg over the movie again, and the button is therefore disabled when the cache holds nothing but frames. The entire cache is removed when the Slide Show window is closed.
- **Zoom in**: Increase the displayed image size by scaling it up. Every click on the button increases the zoom factor by 10 percent.
- **Zoom out**: Decrease the displayed image size by scaling it down. Every click on the button decreases the zoom factor by 10 percent until a minimum zoom factor of 0.1.
- **Rotate clockwise**: Rotate the displayed image 90 degrees clockwise.
- **Rotate counter-clockwise**: Rotate the displayed image 90 degrees counter-clockwise.
- **Mirror horizontally**: Flip the displayed image along the vertical axis.
- **Mirror vertically**: Flip the displayed image along the horizontal axis.
- **Reset image**: Reset the displayed image to the original image. This reverts all zoom, rotate, and mirror operations.
- **Fit window**: Resize the window so the displayed image is shown at its full size, without scroll bars or unused space. This undoes a manual resize of the window; the window is never grown beyond a fraction of the screen, so scroll bars remain for very large images.
- **Stop Simulation** (*Ctrl-/*): Stop a running simulation.

These image transformations are useful when the simulation images need to be adjusted for presentation purposes. The same transformations are also applied when exporting images or movies.

The **playback controls** below the image let you select the displayed image, restrict the active range, and control the slideshow settings:

- **Play**: Start playing the animation, advancing through the active range defined by the **Start** and **Stop** controls.
- **Loop**: Toggle continuous looping of the animation. When enabled, playback wraps around from the last image of the active range back to the first.
- **Delay**: Set the delay in milliseconds between frames during animation playback.

- **Start**: First image of the active range. Animation, single stepping, movie export, and image deletion are all restricted to images at or after this position. Defaults to the first image.
- **Stop**: Last image of the active range. Defaults to the last image and keeps following the growing sequence while a simulation produces new images, unless it has been set to a specific value.
- At the right end of the row, a counter gives the position of the displayed image in the sequence, followed by its file name. Only the name is shown and not the directory holding it: the directory is the same for every image of a sequence and says nothing about which image this is, while a long path would hold the window open to its width. Hovering over the name shows the full path. Frames extracted from a movie file are named after the movie and their frame number in it instead.
- **First**: Jump to the first image of the active range.
- **Previous**: Step back to the previous image.
- The **slider control** selects a frame in the image sequence by moving the slider position. Its track is colored to indicate the active range: positions inside the **Start** to **Stop** range are drawn in blue, while the skipped images outside it are drawn in red.
- **Next**: Step forward to the next image.
- **Last**: Jump to the last image of the active range.

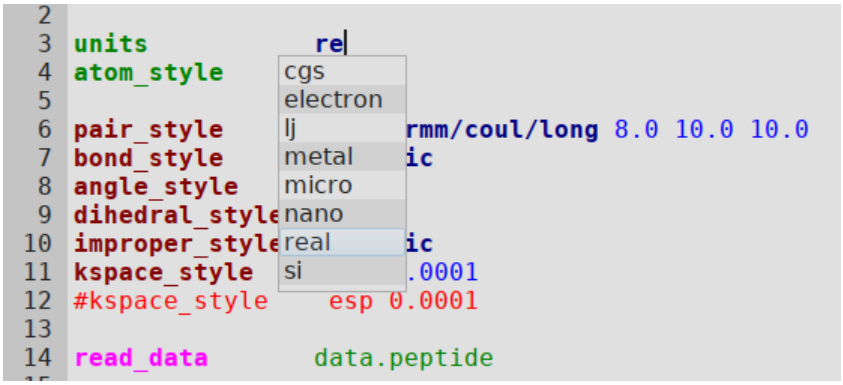
4.6 Editor Window

The *Editor* window of LAMMPS-GUI has most of the usual functionality that similar programs have: text selection via mouse or with cursor moves while holding the Shift key, Cut (*Ctrl-X*), Copy (*Ctrl-C*), Paste (*Ctrl-V*), Undo (*Ctrl-Z*), Redo (*Ctrl-Shift-Z*), Select All (*Ctrl-A*). When trying to exit the editor with a modified buffer, a dialog will pop up asking whether to cancel the exit operation, or to save or not save the buffer contents to a file.

The editor has an auto-save mode that can be enabled or disabled in the *Preferences* dialog. In auto-save mode, the editor buffer is automatically saved before running LAMMPS or before exiting LAMMPS-GUI.

Context Specific Word Completion

By default, LAMMPS-GUI displays a small pop-up frame with possible choices for LAMMPS input script commands or styles after 2 characters of a word have been typed.



The word can then be completed through selecting an entry by scrolling up and down with the cursor keys and selecting with the 'Enter' key or by clicking on the entry with the mouse. The automatic completion pop-up can be disabled in the *Preferences* dialog, but the completion can still be requested manually by either hitting the *Shift+TAB* key or by right-clicking with the mouse and selecting the option from the context menu. Most of the completion information is retrieved from the active LAMMPS instance and thus it shows only available options that have been enabled when compiling LAMMPS. That list, however, excludes accelerated styles and commands; for improved clarity, only the non-suffix versions of styles are shown.

Like the syntax highlighting, the completion and the context-specific help are aware of & line continuations: on a continuation line, completions and help are offered for the command that is being continued.

Syntax Highlighting

The editor highlights LAMMPS input scripts following the same parsing rules as LAMMPS itself: comments, single, double, and triple quoted strings, $ variable substitutions, and numbers are recognized, and command names are colored by category. Arguments are colored by their role in the command, for example the ID, group-ID, and style name of a fix command. IDs use the same color whether they are being defined (`fix`, `compute`, `dump`) or referenced (`fix_modify`, `unfix`, and so on). A few commands embed a second command in their argument list; the keyword that starts the embedded part is shown in the color of the command it stands for, that is the `modify` keyword of the write_dump command is colored like `dump_modify` and the `dump` keyword of the rerun command like `read_dump`. The arguments following such a keyword are colored as arguments of the embedded command, also when the command is spread over multiple lines with & line continuations. Lines joined with the & line continuation character are highlighted in the context of the command they continue, including style names or quoted strings that are split across lines.

Sub-styles of hybrid styles are recognized the same way LAMMPS parses them – by checking against the known styles of the loaded library – and are shown in their own color, both in the arguments of the `*_style` command and in the sub-style position of the corresponding `*_coeff` commands, where they also auto-complete. Words in those positions that are not a known style are treated as arguments of the sub-styles and are never marked as unknown. On dump image lines the dump image keywords are marked, and on both `dump image` and `dump_modify` lines color names are displayed in their actual color; keywords and color names auto-complete in the positions where they are expected. Color names with too little contrast against the editor background (for example `yellow` on a light theme) are placed on a small dark or light chip so they remain readable.

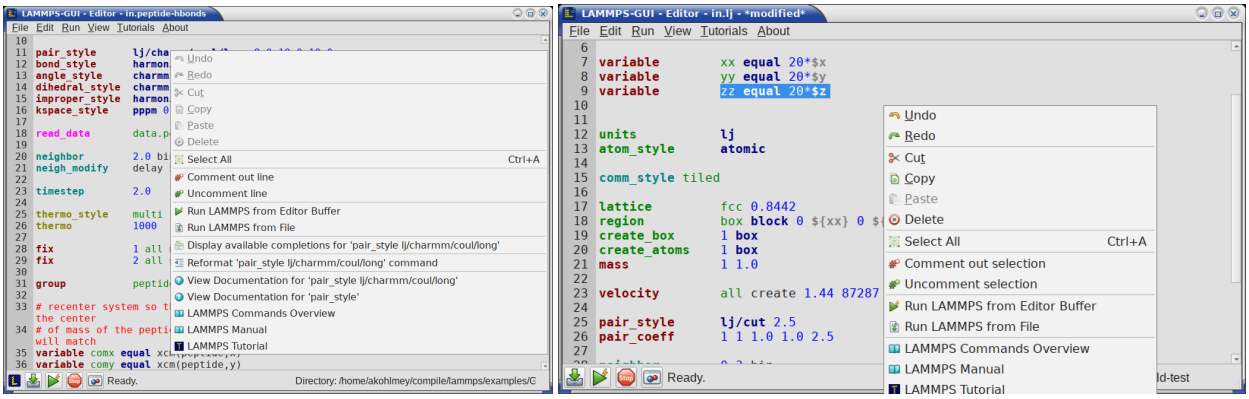
Command and style names that are not known to the LAMMPS library that LAMMPS-GUI has loaded – for example due to a typo or because the corresponding package was not included when the library was compiled – are marked with a wavy underline. The marker is not shown for the word at the cursor position, so partially typed names are not flagged while typing, and it is suppressed for names constructed with $ substitutions. When LAMMPS-GUI is running without a usable LAMMPS library (for example in plugin mode before a library has been selected), the known-name information is unavailable and no words are marked.

Line Reformatting

The editor supports reformatting lines according to the syntax in order to have consistently aligned lines. This primarily means adding whitespace padding to commands, type specifiers, IDs and names. This reformatting is performed manually by hitting the 'Tab' key. It is also possible to have this done automatically when hitting the 'Enter' key to start a new line. This feature can be turned on or off in the *Preferences* dialog for *Editor Settings* with the "Reformat with 'Enter'" checkbox. The amount of padding for multiple categories can be adjusted in the same dialog.

Internally this functionality is achieved by splitting the line into "words" and then putting it back together with padding added where the context can be detected; otherwise a single space is used between words.

Context Specific Help



A unique feature of LAMMPS-GUI is the option to look up the LAMMPS documentation for the command in the current line. This can be done by either clicking the right mouse button or by using the *Ctrl-?* keyboard shortcut. When using the mouse, there are additional entries in the context menu that open the corresponding documentation page in the online LAMMPS documentation in a web browser window. When using the keyboard, the first of those entries is chosen.

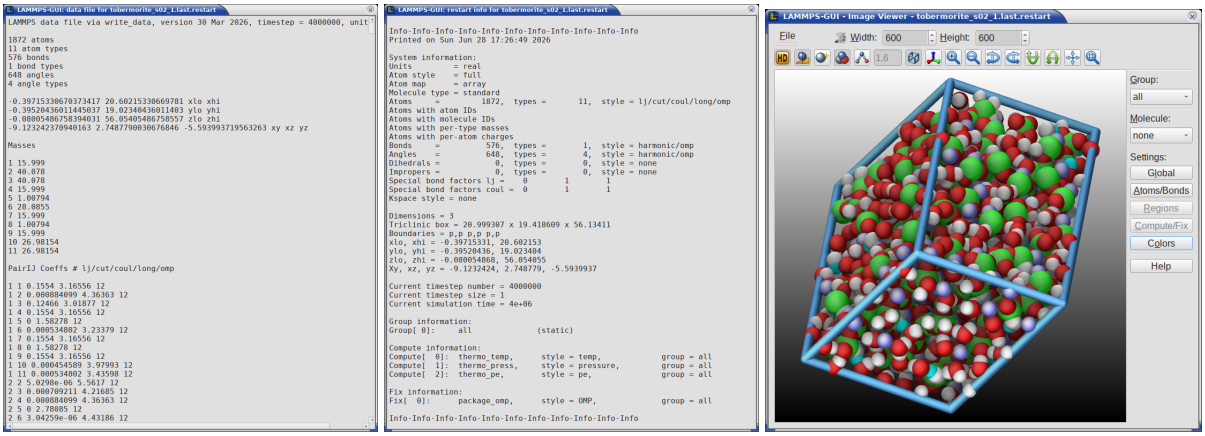
If the current line defines an index style variable, the context menu also has a *Set Variables...* entry that opens the corresponding *dialog* from the *Edit* menu.

If the word under the cursor is a file, then additionally the context menu has an entry to open the file in a read-only text viewer window. If the file is a LAMMPS restart file, instead the menu entry offers to *inspect the restart*.

The text viewer is a convenient way to view the contents of files that are referenced in the input. The file viewer also supports on-the-fly decompression based on the file name suffix in a similar fashion as available with LAMMPS. If the necessary decompression program is missing or the file cannot be decompressed, the viewer window will contain a corresponding message.

Inspecting a Restart file

When LAMMPS-GUI is asked to "Inspect a Restart", it will read the restart file into a LAMMPS instance and then open three different windows. The first window is a text viewer with the output of an info command with system information stored in the restart. The second window is a text viewer containing a data file generated with a write_data command. The third window is a *Snapshot Image Viewer* containing a visualization of the system in the restart.



Large Restart Files

If the restart file is larger than 250 MBytes, a dialog will ask for confirmation before continuing, since large restart files may require large amounts of RAM: the entire system must be read into memory. Thus restart files for large simulations that have been run on an HPC cluster may overload a laptop or local workstation. The *Show Details...* button will display a rough estimate of the additional memory required.

4.7 Menus

The menu bar has entries *File*, *Edit*, *Run*, *View*, *Tutorials*, and *About*. Instead of using the mouse to click on them, the individual menus can also be activated by hitting the *Alt* key together with the corresponding underlined letter, that is *Alt-F* activates the *File* menu. For the corresponding activated sub-menus, the key corresponding to the underlined letters can be used to select entries instead of using the mouse.

LAMMPS-GUI Combined Window Mode

In combined window mode there can be only one menu bar, so the displayed menu bar is that of the section / tab that currently has the focus. If access to the menu bar of a specific window is needed, e.g. to open a new LAMMPS input file, it may be needed to either first click into the corresponding area or use the *F6* or *Shift-F6* keyboard shortcuts to switch focus.

File

The *File* menu offers the usual options

- *New Input File* clears the current buffer and resets the file name to `*unknown*`
- *Open Input File* opens a dialog to select a new file for editing in the *Editor*
- *Save Input File* saves the current file; if the file name is `*unknown*` a dialog will open to select a new file name
- *Save Input File As* opens a dialog to select a new file name (and folder, if desired) and saves the buffer to it. Writing the buffer to a different folder will also switch the current working directory to that folder.
- *View Text File* opens a dialog to select a file for viewing in a *separate* window (read-only) with support for on-the-fly decompression. If the selected file appears to be an image, a movie, or a binary file, a warning is shown instead; use *View Image or Movie File(s)...* for those.
- *View Image or Movie File(s)...* opens a dialog to select one or more image files and shows them together in a standalone *slide show* window. This is useful for reviewing images created by an external (e.g. large parallel) simulation, or for revisiting images from an earlier run without rerunning it. Image formats that Qt cannot read natively are converted on demand with ImageMagick if it is available, and each file is converted only once. Movie files may be selected as well: their frames are extracted into individual images with FFmpeg after confirming a dialog that also selects the frame range and interval, as explained under *Importing movie files*.
- *Plot Data File...* opens a dialog to select a file with column-oriented numeric data and plots it in a standalone *Charts window* without running a simulation. See the description below for details.
- *Inspect Restart File* opens a dialog to select a file. If that file is a LAMMPS restart three windows with *information about the file are opened*.
- *Write Restart File...* opens a dialog to select a file name and then writes a LAMMPS restart file with the current state of the system. This requires a system state, e.g. from running an input. A typical use case

is to preserve the state of a run that was interrupted with the *Stop LAMMPS* entry of the *Run menu* before extending it.

- *Quit* exits LAMMPS-GUI. If there are unsaved changes, a dialog will appear to either cancel the operation, or to save, or to not save the modified buffer.

In addition, up to 5 recent file names will be listed after the *Open Input File* entry that allows re-opening recently opened files. This list is stored when quitting and recovered when starting again.

Files that are not what they are opened as. Each of these entries expects a certain kind of file, and the file dialogs offer *All files* as well, so a file can be picked that does not fit: a binary file for the editor or the text viewer, something that is not a picture for the slide show, an image for the plotter. Rather than refuse it or fill a window with unreadable content, LAMMPS-GUI asks – *"… does not look like a text file. Do you want to open it anyway?"* – and the answer defaults to *No*, since the usual reason to see the question is a name that was mistyped or a file that was picked by mistake. Answering *Yes* opens it regardless, which is occasionally what is wanted: an input file with a stray null byte in it can still be edited. A file that looks right is never asked about. The same check applies to the `edit`, `open` and `plot` commands of the *command window*.

Plotting external data files. The *Plot Data File…* entry (*Ctrl-Shift-P*) opens a dialog to select a file with column-oriented numeric data and plots it in a standalone *Charts window* without running a simulation. Supported formats are whitespace-separated columns (`.dat`), comma-separated values (`.csv`), YAML (including the segmented thermo output that LAMMPS itself writes), and JSON; the format is recognized from the file name extension or, failing that, from the content. After the file is read, a dialog lets you pick which column provides the x axis and which columns to plot; column names can also be edited at this point. The block-structured output of the `fix ave/*` styles is recognized as such, and that dialog then also offers to average the blocks with error bars or to show a single one (see *importing fix ave/* output*). Because there is no associated simulation, the *Units* and *Norm* controls are hidden in such a standalone chart window. All the post-processing and export features described for the *Charts window* are available here as well.

Edit

The *Edit* menu offers the usual editor functions like *Undo*, *Redo*, *Cut*, *Copy*, *Paste*, and a *Find and Replace* dialog (keyboard shortcut *Ctrl-F*).

Changed in version 3.1: The *Preferences* dialog and the option to reset all settings to their defaults have moved to the *View* menu, which is where the window layout they configure is controlled.

Run

The *Run* menu has options to start and stop a LAMMPS process. Rather than calling the LAMMPS executable as a separate executable, the LAMMPS-GUI is linked to the LAMMPS library and thus can run LAMMPS internally through the LAMMPS C-library interface in a separate thread.

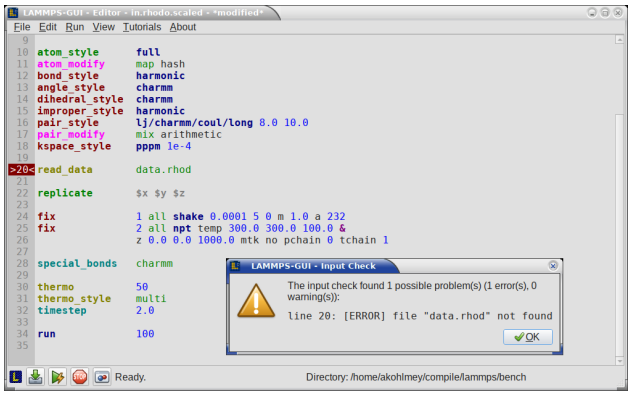
Specifically, a LAMMPS instance will be created by calling lammps_open_no_mpi (through the `LammpsWrapper` C++ adapter). The buffer contents are then executed by calling lammps_commands_string. Certain commands and features are only available after a LAMMPS instance is created. Its presence is indicated by a small LAMMPS `L` logo in the status bar at the bottom left of the main window. As an alternative, it is also possible to run LAMMPS using the contents of the edited file by reading the file. This is mainly provided as a fallback option in case the input uses some feature that is not available when running from a string buffer.

The LAMMPS calculations are run in a concurrent thread so that the GUI can stay responsive and be updated during the run. The GUI can retrieve data from the running LAMMPS instance and tell it to stop at the next timestep. The *Stop LAMMPS* entry will do this by calling the lammps_force_timeout library function, which is equivalent to a timer timeout 0 command.

The *Extend Run…* entry (keyboard shortcut *Ctrl-E*) opens a dialog asking for a number of steps and then continues the previous run for that many more steps without clearing the system. This requires a system state, e.g. from a previous run or an inspected restart file. The continuation executes a timer timeout off command, which resets the expired timer

in case the previous run was interrupted with *Stop LAMMPS*, followed by `run <steps> pre yes post no`. The captured output and thermodynamic data are appended to the existing *Output* and *Charts* windows after a message noting the extension. Typical use cases are continuing a run that was stopped (e.g. after writing a restart file first), or extending a run that did not produce enough frames for a smooth animation or enough data for a plot.

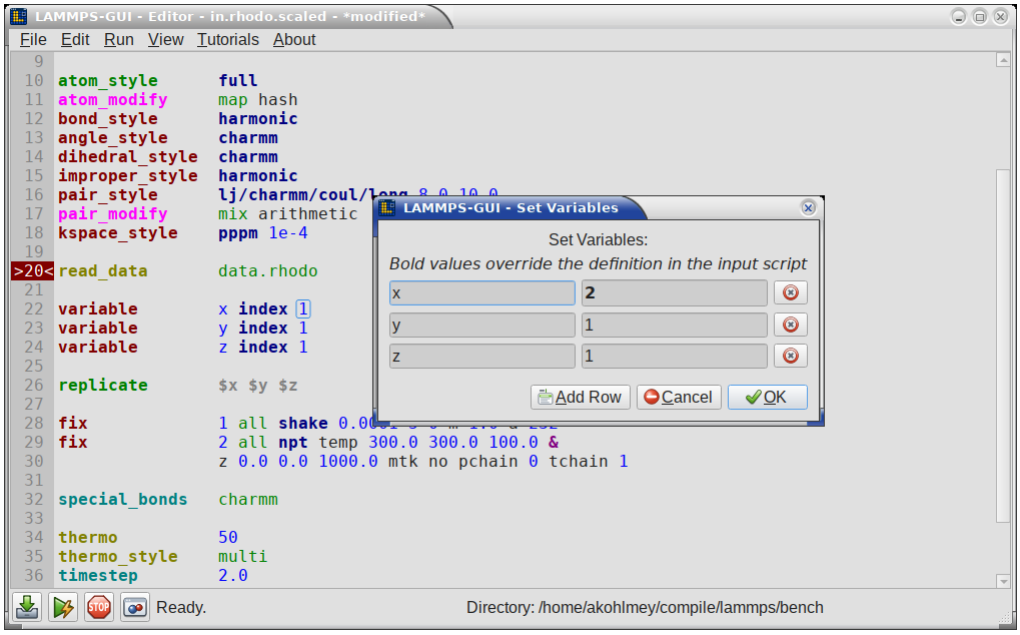
The *Relaunch LAMMPS Instance* will destroy the current LAMMPS thread and free its data and then create a new thread with a new LAMMPS instance. This is usually not needed, since LAMMPS-GUI tries to detect when this is needed and does it automatically. This is available in case it missed something and LAMMPS behaves in unexpected ways.



The *Check Input via Heuristics* entry (keyboard shortcut *Ctrl-K*) runs a fast static check of the editor buffer and reports its findings in a dialog: unknown commands and style names (validated against the loaded LAMMPS library), unbalanced quotes, dangling line continuations, variables used before they are defined, references to undefined groups, references to computes, fixes, or variables that are defined nowhere in the buffer, missing input files, missing required arguments, and non-numeric arguments where strictly numeric values are required. When no problems are found, a corresponding message is shown; otherwise the cursor moves to the first finding. The same check runs automatically before every run (this can be disabled in the *Editor Settings* of the *Preferences* dialog); in that case only error-level findings trigger a dialog asking whether to run anyway, while warnings are only noted in the status bar. The checker is designed to avoid false alarms: any word containing a $ substitution is exempt from checking, and script features that make static analysis unreliable (include files, jump loops, if/then commands, python scripting, shell commands, restart files, runtime plugins) disable the affected groups of checks.

The *Check Input via Dry Run* entry (keyboard shortcut *Ctrl-Shift-K*) validates the buffer by actually executing it: the equivalent of the -skiprun command-line flag is applied, so LAMMPS parses every command and executes the setup phase of every run and minimize command without computing any timesteps. This is a much deeper check than the static one, but it takes as long as the setup of a real run and has the same side effects: output files may be created or overwritten and shell commands are executed, which is why the action first asks for confirmation. The captured output is shown in an *Output* window; errors are reported with the usual error dialog and the offending line is highlighted in the editor. On success, a dialog confirms that the input passed and points to the *Output* window for any LAMMPS warnings.

The *Set Variables...* entry opens a dialog box where index style variables can be set. Those variables are passed to the LAMMPS instance when it is created and are thus set *before* a run is started.



The *Set Variables* dialog will be pre-populated with entries that are set as index variables in the input and any variables that are used but not defined, if the built-in parser can detect them. New rows for additional variables can be added through the *Add Row* button and existing rows can be deleted by clicking on the *X* icons on the right.

The dialog follows edits to the input script: when a `variable ... index` command in the editor is changed, the dialog picks up the new value the next time it is opened or a run is started, even if the value had been changed in the dialog before. A value edited in the dialog so that it differs from the input script is shown in bold with a tooltip listing the script value, and the overridden value in the editor is surrounded by a thin frame as a reminder that the input script line is not what LAMMPS will use. Values from the dialog are passed to LAMMPS before the input script runs, so they take precedence over `variable ... index` commands in the input, exactly like the `-var` command line flag to LAMMPS.

The *Create Image* entry will send a dump image command to the LAMMPS instance, read the resulting file, and show it in an *Image Viewer* window.

The *Open Command Window* entry opens the *Command window*, a shell prompt for the ordinary work that surrounds a run: post-processing a dump file with a script, looking at what a run just wrote, calling a plotting tool. It is *not* a terminal emulator and cannot run full-screen programs such as `vim` or `top`.

The *View in OVITO* entry will launch OVITO with a data file containing the current state of the system. This option is only available if LAMMPS-GUI can find the OVITO executable in the system path.

The *View in VMD* entry will launch VMD with a data file containing the current state of the system. This option is only available if LAMMPS-GUI can find the VMD executable in the system path.

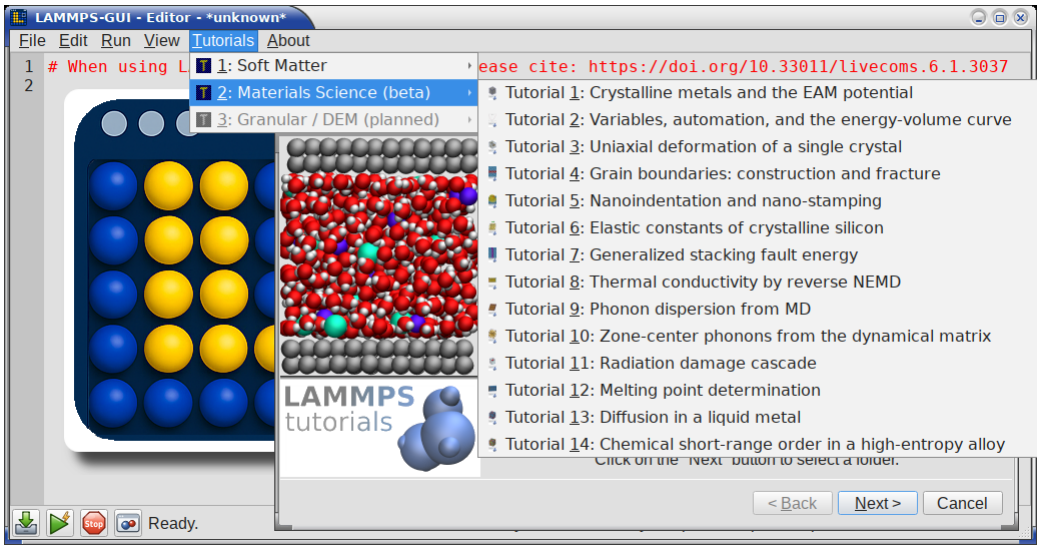
View

The *View* menu offers to show or hide additional windows with log output, charts, slide show, variables, snapshot images, or the *Command window*. With the *Combined Main Window* layout these are panels docked into the main window instead of windows of their own, and the same entries show and hide the panels. The *Image Window* entry first creates a snapshot image, like *Create Image* in the *Run* menu, when there is none yet. The default settings for their visibility can be changed in the *Preferences* dialog.

With that layout the menu also has the *Next Panel* (*F6*) and *Previous Panel* (*Shift-F6*) entries, which move the keyboard focus between the editor and the panels around it; see *the keyboard shortcuts* for what they walk through. They are not shown with individual windows, where switching windows is the window manager's job.

The *View* menu is also where the *Preferences* dialog is opened (keyboard shortcut *Ctrl-P*), and where all stored preferences and settings can be deleted, so they are reset to their default values. Resetting the preferences also deletes a LAMMPS shared library that was previously downloaded into the configuration folder; the library files for all supported platforms are removed in case the configuration folder is shared between multiple computers.

Tutorials



The *Tutorials* menu supports several collections of LAMMPS tutorials for beginners and intermediate LAMMPS users. The menu has one submenu per collection, for example *Soft Matter* (the molecular tutorials documented in *Gravelle1*), *Materials Science*, and *Granular / DEM*. Each submenu lists its individual tutorial sessions; selecting one begins that session.

Collections are released incrementally. A collection that is not yet fully published is labeled in the menu with its status, e.g. *(coming soon)* or *(planned)*. Within such a collection only the tutorials that are already available can be launched; the remaining entries are shown (so you can preview what is coming) but are disabled. A collection with no tutorials available yet appears as a single disabled submenu.

Selecting an available tutorial opens a 'wizard' dialog where you can choose in which folder you want to work, whether you want that folder to be wiped from *any* files, whether you want to download the solution files (which can be large) to a `solution` sub-folder, and whether you want the corresponding tutorial's online version opened in your web browser. The dialog will then start downloading the files requested (download progress is reported in the status line) and load the first input file for the selected session into LAMMPS-GUI.

Should individual tutorial files fail to download, the remaining files are still fetched, and a dialog afterwards lists the files that are missing. That dialog offers a *Report Issue* button that opens the issue tracker of the tutorial's file repository in a web browser, so the missing files can be reported; alternatively they can be reported by email to akohlmey@gmail.com.

About

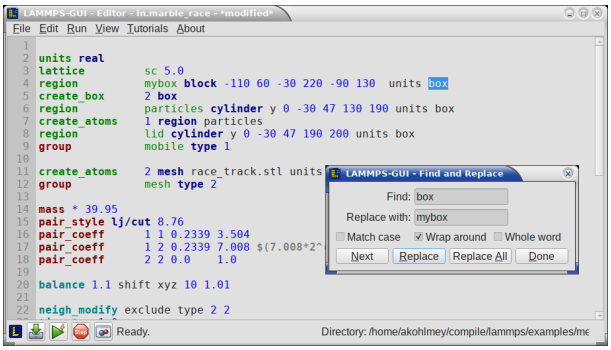
The *About* menu finally offers a couple of dialog windows and an option to launch the LAMMPS online documentation in a web browser. The *About LAMMPS-GUI* entry displays a dialog with a summary of the configuration settings of the LAMMPS library in use and the version number of LAMMPS-GUI itself. The *Quick Help* displays a dialog with a minimal description of LAMMPS-GUI. The *LAMMPS-GUI Documentation* entry will open the LAMMPS-GUI online documentation website https://lammps-gui.lammps.org in a web browser window. The *LAMMPS Manual* entry will open the main page of the LAMMPS online documentation in a web browser window. The *LAMMPS Tutorial* entry will open the main page of the set of LAMMPS tutorials authored and maintained by Simon Gravelle at https://lammpstutorials.github.io/ in a web browser window. The *Check for LAMMPS update* entry – available only in the plugin version of LAMMPS-GUI – compares the downloaded LAMMPS shared library with the latest version

available online and offers to download and install an update when a newer version is found; LAMMPS-GUI is then relaunched to activate it. The checksum of the downloaded file is verified before it replaces the current library, which is renamed to a backup name first; leftover backup files and partial downloads in the configuration folder are cleaned up on the next launch of LAMMPS-GUI.

(Gravelle1) Gravelle, Alvares, Gissinger, Kohlmeyer, Living Journal of Computational Molecular Science, 6(1), 3037. https://doi.org/10.33011/livecoms.6.1.3037 (2025)

4.8 Dialogs

Find and Replace



The *Find and Replace* dialog allows searching for and replacing text in the *Editor* window.

The dialog can be opened either from the *Edit* menu or with the keyboard shortcut *Ctrl-F*. You can enter the text to search for.

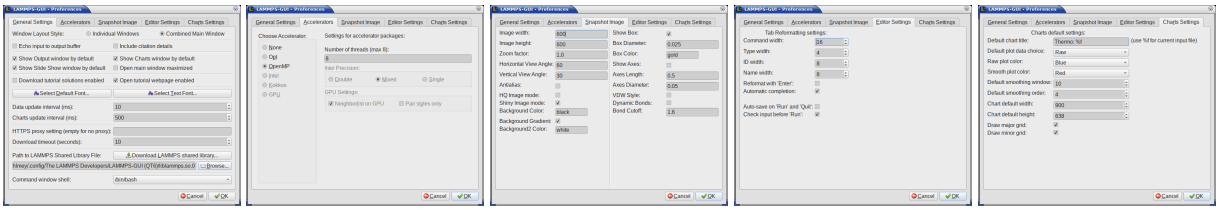
Through three checkboxes the search behavior can be adjusted

- If checked, "Match case" does a case-sensitive search; otherwise the search is case-insensitive.
- If checked, "Wrap around" starts searching from the start of the document, if there is no match found from the current cursor position until the end of the document; otherwise the search will stop.
- If checked, the "Whole word" setting only finds full word matches (white space and special characters are word boundaries).

Clicking on the *Next* button will search for the next occurrence of the search text and select / highlight it. Clicking on the *Replace* button will replace an already highlighted search text and find the next one. If no text is selected, or the selected text does not match the selection string, then the first click on the *Replace* button will only search and highlight the next occurrence of the search string. Clicking on the *Replace All* button will replace all occurrences from the cursor position to the end of the file; if the *Wrap around* box is checked, then it will replace **all** occurrences in the **entire** document. Clicking on the *Done* button will dismiss the dialog.

Preferences

The *Preferences* dialog allows customization of the behavior and look of LAMMPS-GUI. The settings are grouped and each group is displayed within a tab.



General Settings

The following settings are available in this tab

- **Echo input to output buffer:** when checked, all input commands, including variable expansions, are echoed to the *Output window*. This is equivalent to using `-echo screen` on the command-line. There is no log *file* produced by default, since LAMMPS-GUI uses `-log none`.
- **Individual Windows / Combined Main Window:** selects how the Output, Charts, Image, Slide Show, Variables, and Command window views are presented. With *Individual Windows* each of them is a window of its own, placed and stacked freely. With *Combined Main Window* they become panels docked around the editor: the editor keeps the center, the Charts, Image, and Slide Show views share a tabbed group on the right, and the Output, Variables, and Command window views share a group across the full width at the bottom. The panels have fixed places and are not dragged around; they are shown and hidden from the *View* menu, and the splitters between them can be moved to change their proportions. The windows opened on demand join them rather than floating above: a text viewer, an inspected data file or image, a slide show of image or movie files opened with *View Image or Movie File(s)...*, and a plot made with *Plot Data File...* all become further tabs of the group on the right, and are closed from their own *File* menu. A panel that shares its area with another is named by its tab, one that is alone by a title bar. The proportions are kept when the main window is resized and restored in the next session, and the main window remembers a size of its own for each of the two layouts. Changing this setting relaunches LAMMPS-GUI. A single session can be started in either layout without changing this setting, with the `-j` or `-w` *command-line flag*. In the combined window the views do not remember individual window sizes, and a keyboard shortcut that a view shares with the main window (for example `Ctrl+S`) is left to the main window, so only its menu entry remains for the view.
- **Include citation details:** when checked, full citation info will be included in the Output window. This is equivalent to using `-cite screen` on the command-line.
- **Show Output window by default:** when checked, the screen output of a LAMMPS run will be collected in an Output window during the run.
- **Show Charts window by default:** when checked, the thermodynamic output of a LAMMPS run will be collected and displayed in a Charts window as line graphs.
- **Show Slide Show window by default:** when checked, a Slide Show window will be shown with images from a dump image command, if present, in the LAMMPS input.
- **Open main window maximized:** when checked, LAMMPS-GUI starts with its main window filling the screen instead of restoring the size it had when it was last closed. This applies to the combined main window only and is disabled for individual windows, where a maximized main window would cover the very windows it is meant to sit beside.
- **Download tutorial solutions enabled:** this controls whether the "Download solutions" option is enabled by default when setting up a tutorial.

- **Open tutorial webpage enabled:** this controls whether the "Open tutorial webpage in web browser" option is enabled by default when setting up a tutorial.
- **Select Default Font:** Opens a font selection dialog where the type and size for the default font (used for everything but the editor and log) of the application can be set.
- **Select Text Font:** Opens a font selection dialog where the type and size for the text editor and log font of the application can be set.
- **Data update interval:** Allows the user to set, in milliseconds, the time interval between data updates during a LAMMPS run. The default is to update the data (for the Charts and Output windows) every 10 milliseconds. This is good for many cases. Set this to 100 milliseconds or more if LAMMPS-GUI consumes too many resources during a run. For LAMMPS runs that run *very* fast (for example in tutorial examples), however, data may be missed; this can be corrected by lowering this interval. However, this will make the GUI use more resources. This setting may be changed to a value between 1 and 1000 milliseconds.
- **Charts update interval:** Allows the user to set, in milliseconds, the time interval between redrawing the plots in the *Charts window*. The default is to redraw the plots every 500 milliseconds. This is just for the drawing; data collection is managed with the previous setting.
- **HTTPS proxy setting:** Allows the user to enter a URL for an HTTPS proxy. This may be needed when the LAMMPS input contains geturl commands or for downloading tutorial files from the *Tutorials* menu. If the `https_proxy` environment variable was set externally, its value is displayed but cannot be changed.
- **Download timeout:** Sets the time, in seconds, after which a download (of tutorial files or the LAMMPS shared library) is aborted with an error message when no data has arrived. The default is 10 seconds. Users with a slow internet connection may want to increase this value so that the download of larger files, for example the LAMMPS shared library, is not canceled prematurely.
- **Path to LAMMPS Shared Library File:** this option is only visible when LAMMPS-GUI was compiled to load the LAMMPS library at runtime instead of being linked to it directly. Using the *Browse...* button or by changing the text, a different shared library file with a different compilation of LAMMPS with different settings or from a different version can be loaded. The accompanying *Download LAMMPS shared library...* button retrieves a pre-built LAMMPS shared library from the LAMMPS web server. After changing this setting, LAMMPS-GUI needs to be re-launched. This setting is remembered separately for each compiler toolchain LAMMPS-GUI was built with, so for example an MSVC build and a MinGW build on the same machine each keep their own library: a library built against a different C runtime loads and runs, but its screen output would silently bypass the Output window.
- **Command window shell:** selects the command interpreter that the *Command window* starts, from a list of the shells installed on the machine: on Unix-like systems the entries of `/etc/shells` that are actual shells (not `nologin` or a terminal multiplexer), on Windows `cmd.exe`, PowerShell, and a `bash.exe` from an installation such as Git for Windows, when present. A shell listed under more than one path (for example under both `/bin` and `/usr/bin`) is offered only once. The default is the user's default shell (`SHELL` on Unix-like systems, `COMSPEC` on Windows). A change takes effect when the next shell starts: when the Command window is first opened, or on its *File* > *Restart Shell*.

Accelerators

This tab enables selection of an accelerator package and modification of some of its settings for use when running LAMMPS. This is equivalent to using the -sf and -pk flags on the command-line. Only settings supported by the LAMMPS library and local hardware are available. The *Number of threads* field allows setting the number of threads for the accelerator packages that support using threads (OPENMP, INTEL, KOKKOS, and GPU). Furthermore, the precision mode (double, mixed, or single) for the INTEL package can be selected, and for the GPU package, whether the neighbor lists are built on the GPU or the host (required for pair style hybrid) and whether only pair styles should be accelerated (i.e., run PPPM entirely on the CPU, which sometimes leads to better overall performance). Whether settings can be changed depends on which accelerator package is chosen (or "None").

Snapshot Image

This tab allows setting defaults for the snapshot images displayed in the *Image Viewer window*, such as its dimensions, the zoom factor, and view angles. The **Antialias** switch will render images with double the number of pixels for width and height and then smoothly scale the image back to the requested size. This produces higher quality images with smoother edges at the expense of requiring more CPU time to render an initial image four times the size. The **HQ Image mode** option turns on "Screen Space Ambient Occlusion (SSAO)" mode when rendering images. This is also more time consuming, but produces a more 'spatial' representation of the system with shading of atoms by their depth. The **Shiny Image mode** option will render objects with a shiny surface when enabled. Otherwise, the surfaces will be matte. The **Show Box** option selects whether the system box is drawn as a colored set of sticks. Furthermore, the diameter of the sticks and their color can be set. Similarly, the **Show Axes** option selects whether a representation of the three system axes will be drawn or not (as colored and labeled arrows). In addition, the axes length and diameter can be set in fractions of the image size. The **VDW Style** checkbox selects whether atoms are represented by space filling spheres when checked or by smaller spheres and sticks. The **Dynamic Bonds** checkbox selects whether bonds between atoms shall be automatically determined from the atom distances (for instance to visualize simulations using force fields with implicit bonds), and the corresponding "Bond Cutoff" text field allows the user to set the cutoff used for that feature. Finally, there are a couple of text fields to select the two **Background Colors**. If the two colors differ, there will be a vertical background gradient starting with the "Background" color at the bottom and ending with the "Background2" color at the top.

These settings correspond to the available settings for the LAMMPS dump image and corresponding dump_modify commands.

Editor Settings

This tab allows adjusting settings of the *editor window*. Specifically, the amount of padding to be added to LAMMPS commands, types or type ranges, IDs (e.g., for fixes), and names (e.g., for groups). The value set is the minimum width for the text element and it can be chosen in the range between 1 and 32.

The three settings which follow enable or disable the automatic reformatting when hitting the 'Enter' key, the automatic display of the completion pop-up window, and whether auto-save mode is enabled. In auto-save mode, the editor buffer is saved before a run or before exiting LAMMPS-GUI.

The last setting enables or disables (default: enabled) the static input check (see the *Check Input via Heuristics* entry of the *Run menu*) that runs automatically before every run. Only error-level findings, which would make LAMMPS reject the input, open a dialog asking whether to run anyway; warnings are only noted in the status bar.

Charts Settings

This tab allows adjusting settings of the *Charts window*. Specifically, one can set the default chart title (if the title contains '%f' it will be replaced with the name of the current input file), one can select whether by default the raw data, the smoothed data, or both will be plotted, the default smoothing parameters, the default size of the chart graph in pixels, and whether you want to display major and minor grid lines.

The *Raw data*, *Processed data*, and *Error bars* groups hold the defaults of the per-chart *Chart Style...* dialog (see *Adjust chart style*): the display style (*Lines*, *Points*, or *Lines + Points*), the color (one of ten preset colors), the line width, and the point size of the two data series, plus the color and line width of the error bars that imported data can carry. Changing a color here also applies to charts that are already open, since a chart only stores a color of its own once one is picked in its *Chart Style...* dialog; the widths and display styles are picked up by charts created afterwards.

4.9 Keyboard Shortcuts

Almost all functionality is accessible from the menu of the editor window or through keyboard shortcuts. The following shortcuts are available (On macOS use the Command key instead of Ctrl/Control).

Shortcut	Function	Shortcut	Function	Shortcut	Function
Ctrl+N	New File	Ctrl+Z	Undo edit	Ctrl+Enter	Run Input
Ctrl+O	Open File	Ctrl+Shift+Z	Redo edit	Ctrl+/	Stop Active Run
Ctrl+Shift+F	View Text File	Ctrl+C	Copy text	Ctrl+Shift+V	Set Variables
Ctrl+S	Save File	Ctrl+X	Cut text	Ctrl+I	Snapshot Image
Ctrl+Shift+S	Save File As	Ctrl+V	Paste text	Ctrl+L	Slide Show
Ctrl+Q	Quit Application	Ctrl+A	Select All	Ctrl+F	Find and Replace
Ctrl+W	Close Window	TAB	Reformat line	Shift+TAB	Show Completions
Ctrl+Shift+Enter	Run File	Ctrl+Shift+W	Show Variables	Ctrl+P	Preferences
Ctrl+Shift+A	About GUI	Ctrl+Shift+H	Quick Help	Ctrl+Shift+G	LAMMPS-GUI Docs
Ctrl+Shift+R	Inspect Restart File	Ctrl+Shift+O	View in OVITO	Ctrl+Shift+D	View in VMD
Ctrl+Shift+L	Output Window	Ctrl+Shift+C	Charts Window	Ctrl+Shift+I	Image Window
Ctrl+Shift+M	LAMMPS Manual	Ctrl+?	Context Help	Ctrl+Shift+T	LAMMPS Tutorial
Ctrl+Shift+U	Update LAMMPS	Ctrl+Home	Go to Home	Ctrl+End	Go to End
Ctrl+Shift+J	View Image or Movie File(s)	Ctrl+Shift+P	Plot Data File	Ctrl+E	Extend Run
Ctrl+K	Check Input via Heuristics	Ctrl+Shift+K	Check Input via Dry Run	Ctrl+Shift+X	Command Window

Further keybindings of the editor window are documented with the Qt documentation. In case of conflicts the list above takes precedence.

All other windows only support a subset of keyboard shortcuts listed above. Typically, the shortcuts *Ctrl-/* (Stop Run), *Ctrl-W* (Close Window), and *Ctrl-Q* (Quit Application) are supported. Some sub-windows also rebind shortcuts for window-specific actions:

- *Output* window: *Ctrl-N* jumps to the next warning or error, *Ctrl-S* saves the captured log to a file, *Ctrl-Y* exports any embedded YAML data, *Ctrl-Enter* runs the current input buffer.
- *Image Viewer* window: *Ctrl-S* saves the rendered image, *Ctrl-C* copies it to the clipboard, *Ctrl-D* copies the `dump image` / `dump_modify` commands to the clipboard.
- *Slide Show* window: *Ctrl-S* saves the currently displayed image, *Ctrl-C* copies it to the clipboard, *Ctrl-E* exports the image sequence to a movie file.

With the *Combined Main Window* layout the sub-windows are panels docked into the main window, and a shortcut a panel shares with the main window (for example *Ctrl-S*) is left to the main window; the panel's own function remains available from its menu entry.

That layout also has keyboard shortcuts for moving between the panels:

- *F6* moves the keyboard focus to the next panel and *Shift-F6* to the previous one, walking the editor, the group on the right and the group across the bottom in turn and wrapping around at either end. A panel that shares a tab group with others is reached this way when its tab is the one in front; the transient panels – a text viewer, an inspected file, a plot of a data file – are walked like any other.
- The shortcut that opens a panel also returns to it: pressing it while the panel is on screen but is not the one being worked in moves the focus there instead of hiding it, and pressing it again – with the focus in the panel by then – hides it as before. With individual windows the shortcut toggles the window as it always has, since there the stacking and the focus belong to the window manager.

---

5 Programmer's Guide

This guide provides documentation for developers who want to understand the internals of LAMMPS-GUI or contribute to its development.

AI Generated Content

The initial version of the Programmer's Guide section was created by the GitHub Copilot Coding Agent and not everything has been carefully checked. It is therefore possible that it contains errors where the LLM has misinterpreted the LAMMPS-GUI source code. If you spot any such errors or inconsistencies, please submit a bug report issue to point them out or – even better – submit a pull request with the necessary corrections.

5.1 Overview

LAMMPS-GUI is built using C++17 and the Qt Framework (Qt 6.2+). The application follows object-oriented design principles with separation of concerns between different components:

- **Editor Components**: Handle text editing, syntax highlighting, and auto-completion
- **LAMMPS Interface**: Wraps the LAMMPS C library API
- **Visualization**: Displays images, charts, and simulation output
- **GUI Framework**: Main window, dialogs, and preferences

LAMMPS Interface

LAMMPS-GUI can operate in two modes: **Plugin Mode** and **Linked Mode**. The mode is controlled by the `-D LAMMPS_GUI_USE_PLUGIN=(ON|OFF)` CMake configuration option.

Plugin Mode (default)
: LAMMPS is loaded dynamically at runtime from a shared library file (.so, .dll, .dylib). This allows using different LAMMPS builds with different compilation settings and different LAMMPS versions without recompiling the GUI. The library loading is handled in *`LammpsWrapper`* using platform-specific dynamic loading functions (`dlopen()` on Unix/Linux/macOS, `LoadLibrary()` on Windows). The path to the shared library file is auto-detected or configured via command line or preferences.

Linked Mode
: The LAMMPS library is linked at compile time. Used by default when building LAMMPS-GUI as part of a LAMMPS CMake build with `-D BUILD_LAMMPS_GUI=on`. For standalone builds, the `-D LAMMPS_SOURCE_DIR=<path to LAMMPS' src folder>` and `-D LAMMPS_LIBRARY=<path to LAMMPS shared or static library file>` settings are also required when configuring with CMake. It may be necessary to adjust environment variables to find shared libraries (`LD_LIBRARY_PATH` on Linux, `DYLD_LIBRARY_PATH` on macOS, or `PATH` on Windows) when linked to a shared library.

Qt Integration

LAMMPS-GUI makes extensive use of Qt features:

Signals and Slots
: Used for inter-component communication, especially between GUI components and background threads.

Qt Resource System
: Icons and resources embedded via `resources/lammpsgui.qrc`. The icons are SVG and are rendered through the Qt6 Svg module (`Qt6::Svg`).

Qt Models
: Used for data display in various viewers and inspectors.

LAMMPS-GUI requires the Qt application development and GUI framework version 6.2 or later, including the Widgets, Network, and Svg modules. See the Qt Documentation for more details.

5.2 Architecture

Main Components

The application architecture consists of several key components organized into functional groups:

Main Window

LammpsGui (lammpsgui.h/.cpp)
: The main window class that coordinates all other components. It manages the editor, handles file operations, controls LAMMPS execution, and manages the overall application state. This is the central hub of the application that integrates all other components. The UI is built programmatically in `setupUi()`, which delegates menu construction to `createFileMenu()`, `createEditMenu()`, `createRunMenu()`, `createViewMenu()`, `createTutorialMenu()`, `createAboutMenu()`, and the status bar to `createStatusBar()`. Plugin discovery and accelerator setup are handled by `setupPlugin()` and `setupAccelerators()`. See *LammpsGui*

Editor Components

CodeEditor (codeeditor.h/.cpp)
: Custom text editor widget based on QPlainTextEdit, providing LAMMPS-specific features including syntax highlighting, auto-completion, line numbers, and context-sensitive help. The main editing surface for LAMMPS input scripts. See *CodeEditor*

LineNumberArea (linenumberarea.h)
: Widget that displays line numbers in the left margin of the CodeEditor. Updates dynamically as text is added or removed. See *LineNumberArea*

Highlighter (highlighter.h/.cpp)
: Syntax highlighter for LAMMPS input scripts. Categorizes and colors different types of commands, keywords, variables, and comments using Qt's QSyntaxHighlighter framework. See *Highlighter*

FindAndReplace (findandreplace.h/.cpp)
: Dialog for searching and replacing text in the editor. Supports case-sensitive search, wrap-around search, and whole-word matching options. See *FindAndReplace*

LAMMPS Interface

LammpsWrapper (lammpswrapper.h/.cpp)
: C++ wrapper around the LAMMPS C library interface. Provides a clean C++ API and handles dynamic library loading in plugin mode. Manages LAMMPS initialization, command execution, and error handling. See *LammpsWrapper*

LammpsRunner (lammpsrunner.h)
: Worker thread for executing LAMMPS simulations without blocking the GUI. Uses Qt's threading facilities to run simulations in the background, allowing the UI to remain responsive during long calculations. See *LammpsRunner*

Visualization Components

ImageViewer (imageviewer.h/.cpp)
: Dialog for viewing and manipulating LAMMPS snapshot images created by the `dump image` command. Supports interactive control of visualization parameters such as zoom, rotation, atom size, coloring, and rendering op-

tions. Changes can be applied to regenerate the image using the LAMMPS library interface. See *ImageViewer*. This uses two internal helper classes:

- **ImageInfo** - Stores settings for displaying graphics from a LAMMPS compute or fix in snapshot images.
- **RegionInfo** - Stores settings for displaying a region in snapshot images.

ChartWindow (chartviewer.h/.cpp)
Window for displaying thermodynamic data as charts. Supports line plots and multiple data series. It owns one `ChartColumn` per thermo column, which holds the neutral `PlotSeries` data objects of that column, and a single *ChartViewer* that is rebound to whichever column is selected. See *ChartWindow*

ChartViewer (chartviewer.h/.cpp)
Custom chart view widget that provides interactive features like zooming, smoothing, and panning for data visualization. ChartViewer is a view over the active `ChartColumn` of its window and renders it with *PlotWidget*. See *ChartViewer*.

PlotDataDialog (plotdatadialog.h/.cpp)
Column picker for data files opened for plotting: selects the x and the y columns, renames columns, adds derived columns computed from expressions, and, for block-structured files, reduces the blocks to a single one or to the average of a range. `PlotDataDialog::fromFile()` reads a file and builds the dialog matching its format. See *PlotDataDialog*

PlotBlockData (plotblockdata.h/.cpp)
Parsers for the block-structured output files of the `fix ave/*` styles and their reduction to a flat *PlotData* table. The helpers shared with the flat-file parsers of `plotdata.cpp` are declared in `plotdata_internal.h`. See *PlotBlockData*

PlotWidget (plotwidget.h/.cpp)
Native `QWidget` + `QPainter` 2D line/scatter chart renderer. It is the only chart backend and depends only on Qt Widgets – no Qt Charts, Qt Graphs, or QML. Axis-layout math (nice ticks, label formatting) lives in the Qt-free `plotaxismath` helpers. See *PlotWidget*.

SlideShow (slideshow.h/.cpp)
Dialog for viewing multiple images as a slideshow or animation with navigation controls. Supports converting an animation to a movie file when FFmpeg or ImageMagick is available. See *SlideShow*

RangeSlider (thirdparty/rangeslider/rangeslider.h/.cpp) Custom
slider widget with two handles for selecting a range of values. This is code written by Hoyoung Lee and distributed under the CeCILL Free Software License. Used in *ChartWindow* for selecting x- and y-direction plot ranges. See *RangeSlider*

RangeBandSlider (rangebandslider.h/.cpp)
Horizontal `QSlider` that paints an active sub-range on its track, distinct from the third-party *RangeSlider*. See *RangeBandSlider*

Dialog and Utility Components

LogWindow (logwindow.h/.cpp)
Window displaying captured output from LAMMPS simulations. Updates in real-time as the simulation progresses and highlights warning and error messages. Provides navigation to jump between warnings. See *LogWindow*

Preferences (preferences.h/.cpp)
Dialog for configuring application settings including accelerator packages, editor appearance, snapshot settings, and chart preferences. Settings are made persistent across LAMMPS-GUI sessions using the QSettings class. See *Preferences*. The dialog is organized into five tabs, each implemented as a separate widget class:

- *GeneralTab* - General settings (LAMMPS library path, fonts, etc.)

- *AcceleratorTab* - LAMMPS accelerator package configuration
- *SnapshotTab* - Snapshot image viewer defaults
- *EditorTab* - Editor appearance and behavior settings
- *ChartsTab* - Chart viewer display settings

SetVariables (setvariables.h/.cpp)
Dialog for editing LAMMPS index-style variable definitions. Allows users to define name-value pairs that are substituted in input scripts using `${varname}` syntax. See *SetVariables*

FileViewer (fileviewer.h/.cpp)
Read-only text viewer dialog for displaying file contents. Used for viewing auxiliary files without allowing modifications. See *FileViewer*

CommandWindow (commandwindow.h/.cpp)
Shell prompt with scrollback next to the simulation. It runs one persistent shell process without a terminal, frames each command with a sentinel line that reports the exit status and the working directory, and offers commands that hand files to the slide show, the editor, or a plot. See *CommandWindow*. It uses two helper classes:

- *ShellPrompt* - The input line with Tab completion of commands, file names, and earlier command lines.
- *ShellAliases* - The configurable aliases, which are defined in the shell at start since a non-interactive shell reads no startup files.

TutorialWizard (tutorialwizard.h/.cpp)
Wizard dialog for interactive LAMMPS tutorials. Guides users through setting up tutorial directories and files, providing a structured learning experience. See *TutorialWizard*

AboutDialog (aboutdialog.h/.cpp)
Custom About dialog that displays version information, LAMMPS configuration details, and available styles in two scrollable text areas. The dialog automatically scrolls down when the content exceeds the visible area, pauses at the bottom, and then returns back to the top. See *AboutDialog*

Support Components

WindowLayout (windowlayout.h/.cpp)
Mediator between the main window and its output views that implements the two window layouts: individual windows, or docked panels in a single main window with their proportions kept across resizes and sessions. The views themselves do not know which layout they are in. See *WindowLayout*

URLDownloader (urldownloader.h/.cpp)
Utility class for downloading files over HTTPS. Provides a synchronous download interface using `QNetworkAccessManager` with `QEventLoop`. Respects the `https_proxy` preference setting and the `https_proxy` environment variable. After downloading a file, it checks for a `SHA256SUMS` file in the same remote directory and verifies the SHA-256 checksum if available. See *URLDownloader*

StdCapture (stdcapture.h/.cpp)
Utility class that captures stdout output from LAMMPS. Redirects the C-level stdout file descriptor through a pipe to allow capturing output from the LAMMPS library without blocking the GUI thread. See *StdCapture*

FlagWarnings (flagwarnings.h/.cpp)
Syntax highlighter for LAMMPS warning and error messages in log output. Detects and highlights WARNING/ERROR lines and URLs for documentation links. Maintains a count of warnings and updates a summary label. See *FlagWarnings*

QHline (qaddon.h/.cpp)
Simple horizontal line widget for visual separation in dialogs and forms. See *QHline*

QColorCompleter (qaddon.h/.cpp)
: Auto-completer for color name inputs, suggesting valid Qt color names as the user types. See *QColorCompleter*

QColorValidator (qaddon.h/.cpp)
: Validator for color input fields, ensuring they contain valid color names or hex color codes. See *QColorValidator*

VerticalLabel (qaddon.h/.cpp)
: Label widget that renders text rotated 90 degrees for vertical display. See *VerticalLabel*

Helper Functions

The *helpers module* provides utility functions used throughout the application. The `chartstyle.h/.cpp` module next to it holds the palette of preset series colors and the widget builders that the Chart Style dialog and the charts preferences tab share, so that both offer the same choices. The helper functions include:

- Date comparison (`dateCompare` for version comparisons)
- Command-line parsing (`splitLine` with quote handling)
- System utilities (`hasExe` for executable detection)
- UI utilities (`isLightTheme` for theme detection, `showUnsavedChangesDialog` for standardized unsaved-changes prompts)
- Menu construction (`addMenuAction` builds a menu action with an optional icon and a `triggered()` handler in one call; used to build the context and tool menus across the widget classes)
- Stdout management (`silenceStdout`/`restoreStdout` for suppressing LAMMPS library output, with the *StdoutSilencer* RAII guard for scope-based silencing, coordinated with *StdCapture* via `isStdoutSilenced` and `notifyCaptureState`)

Constants (constants.h)
: The `Cfg` namespace centralizes application-wide magic numbers and repeated string literals, such as default buffer sizes, minimum window dimensions, file limits, resource paths, and version constants, while the `Keys` namespace holds every persisted `QSettings` key and group name. Using named constants avoids typos and makes maintenance easier.

Data Flow

1. **User Input**: User edits LAMMPS input in CodeEditor with syntax highlighting
2. **Execution Request**: User triggers execution via menu or button
3. **Preparation**: LammpsGui creates/configures LammpsWrapper and prepares variables
4. **Threading**: Commands sent to LammpsRunner thread to avoid UI blocking
5. **Execution**: LammpsRunner executes commands via LammpsWrapper
6. **Output Capture**: Output captured via StdCapture for display
7. **Visualization**: Results displayed in LogWindow, ImageViewer, or ChartWindow
8. **Completion**: UI updated when execution completes, progress indicators cleared

Settings and State Management

The application uses Qt's QSettings mechanism to persist:

- Recent files list
- Window geometry and state

- Editor preferences (font, colors)
- Accelerator settings
- LAMMPS plugin path
- Tutorial preferences

Settings are stored in platform-specific locations (the application name includes the Qt major version, e.g. `LAMMPS-GUI (QT6)`):

- Linux: `~/.config/The LAMMPS Developers/LAMMPS-GUI (QT6).conf`
- macOS: `~/Library/Preferences/org.lammps.LAMMPS-GUI (QT6).plist`
- Windows: Registry under `HKEY_CURRENT_USER\Software\The LAMMPS Developers\LAMMPS-GUI (QT6)`

Threading Model

The application uses Qt's event-driven architecture with threading:

- **Main Thread**: Handles all UI operations and user interactions
- **LAMMPS Thread**: LammpsRunner executes LAMMPS in a separate QThread
- **Communication**: Signals/slots for thread-safe communication between threads

This design keeps the UI responsive even during long-running simulations.

5.3 API Reference

The following sections provide detailed API documentation for the main classes in LAMMPS-GUI. Documentation is generated from Doxygen comments in the source code.

Main Window

LammpsGui Class

class **LammpsGui** : public QMainWindow

Main application window for LAMMPS-GUI.

LammpsGui is the central component of the LAMMPS-GUI application, serving as the main window that coordinates all other components. It manages:

- The code editor for LAMMPS input scripts with syntax highlighting
- File operations (open, save, recent files)
- LAMMPS simulation execution and control
- Visualization windows (images, charts, log output)
- Application preferences and settings
- Tutorial wizard for interactive LAMMPS learning

The class integrates with Qt's main window framework and provides menu actions, toolbars, and status bar components. It uses a *LammpsWrapper* to interface with the LAMMPS library and *LammpsRunner* to execute simulations in a separate thread.

See also

CodeEditor for the text editor component

See also

ChartWindow for the charts window component

See also

LogWindow for the log output window component

See also

ImageViewer for the snapshot image window component

See also

SlideShow for the slide show viewer window component

See also

Preferences for the preferences window component

See also

LammpsRunner for simulation execution in a separate thread

See also

LammpsWrapper for LAMMPS library interface

Public Functions

LammpsGui(QWidget *parent = nullptr, const QString &filename = QString(), int width = 0, int height = 0)

Construct the main application window.

Initializes the main window, sets up the UI components, loads preferences, initializes the LAMMPS library, and optionally opens a file if provided.

Parameters

- **parent** – Parent widget (typically nullptr for main window)

- **filename** – Optional file to open on startup
- **width** – Optional main editor window width override
- **height** – Optional main editor window height override

~LammpsGui() override

Destructor.

Cleans up resources including dynamically created widgets and LAMMPS instances.

LammpsGui() = delete

LammpsGui(const *LammpsGui*&) = delete

LammpsGui(*LammpsGui*&&) = delete

LammpsGui &**operator=**(const *LammpsGui*&) = delete

LammpsGui &**operator=**(*LammpsGui*&&) = delete

void **openFile**(const QString &filename)

Load a file into the editor.

Ends a running simulation and closes the output windows, and offers to save the current buffer first if it was changed. Also reachable from the command window's "edit".

Parameters
filename – File to edit; nothing happens if it is empty

bool **plotFile**(const QString &fileName)

Plot the columns of a data file, asking which ones.

The return value lets a caller with several files to plot stop at the first cancel rather than ask again for each of the rest. Also reachable from the command window's "plot".

Parameters
fileName – Data file to read

Returns
false only if the user canceled the column dialog

void **viewFile**(const QString &filename)

Open a file in a read-only text viewer.

Refuses an image or a movie file, which belong in *openImageFiles()*, and a binary one, which belongs nowhere. Also reachable from the command window's "open".

Parameters
filename – File to show; nothing happens if it is empty

void **openImageFiles**(const QStringList &files)

Open image or movie files in a slide show viewer.

The list is taken as given: openImages() collects it from a file dialog, the command window's "open" from a shell that has expanded it. Movie files are offered for frame extraction as they are added, which is why the viewer is shown before they are.

Parameters
files – Images and movies to show together in one viewer

QList<QMenu*> **sharedMenus**() const

The menus that act on the application rather than on one view.

These are owned by the main window but shown by every window that has a menu bar: a QMenu can be added to more than one QMenuBar, which puts the *same* actions there rather than duplicates. That is what lets a run be started or stopped from any window, and it is also why the accelerators stay unambiguous – one action matches once, however many menus show it.

Returns
Run, View, Tutorials and About, in the order they should appear

Public Slots

void **quit**()

Quit the application.

void **stopRun**()

Stop a running LAMMPS simulation.

inline void **runBuffer**()

Run LAMMPS with content from editor buffer.

Protected Functions

void **inspectFile**(const QString &filename)

Read a restart file into LAMMPS and open the inspection windows.

void **writeFile**(const QString &filename)

Write current editor content to a file.

void **updateEditorTitle**(const QString &file)

Set the editor window title from the current file and run number.

In the docked layout the views are named by their dock tab and no longer carry a window title with the run number, so the editor title shows it.

void **updateRecents**(const QString &filename = "")

Update the recent files list.

void **clearVariables**()

Delete all variables defined in the LAMMPS instance.

void **updateVariables**()

Rebuild the variables list from the editor buffer.

void **refreshVariables**()

Fold input script edits into the variables list.

Re-parses the editor buffer and merges the result into the current variables list: a changed index variable definition in the input wins over a previous value, an unchanged one keeps the value set in the Set Variables dialog. Also updates the override markers shown in the editor. Called before the variables list is consumed (run setup, input check, Set Variables dialog).

void **doRun**(bool use_buffer, bool dryrun = false)

Execute a LAMMPS simulation.

Parameters

- **use_buffer** – If true, runs from editor buffer; if false, saves and runs from file
- **dryrun** – If true, checks the input via a dry run: LAMMPS executes the setup of every command but no timesteps (the equivalent of the -skiprun command line flag)

bool **hasSystemState**()

Check whether the LAMMPS instance holds a usable system state.

Guard for operations that act on the current system state outside of a run, like writing a restart file or extending the previous run.

Returns

true if LAMMPS is open, idle, and a simulation box is defined

void **launchRunner**(std::string input, std::string file, bool clearfirst)

Create and start a *LammpsRunner* thread and the log update timer.

Shared tail of *doRun()* and extendRun(): dispatches the given input to a new runner thread, connects its completion to *runDone()*, and starts the periodic polling of captured output and thermo data.

Parameters

- **input** – String of LAMMPS commands to execute (can be empty)
- **file** – Input file path to execute (can be empty)
- **clearfirst** – If true, wipe the current LAMMPS system state first

void **startLammps**()

Initialize and start a new LAMMPS instance.

void **populateSyntax**()

Fill the syntax registry from LAMMPS library introspection.

Queries the command and style name lists from the running LAMMPS instance into `syntax` and re-highlights the editor. Does nothing when no LAMMPS instance is available (the registry then stays unpopulated and unknown-name marking is disabled).

QStringList **presetVariableNames**() const

Variable names that are defined before the input runs.

The names from the Set Variables dialog plus the always-present `gui_run` variable; passed to the input checker as preset names.

bool **confirmLintIssues**()

Run the pre-run input lint and ask about found errors.

Called from *doRun()* when the pre-run check preference is enabled. Warning-level findings never block; they are noted in the status bar. Error-level findings pop up a “run anyway?” dialog.

Returns

true when the run should proceed

void **runDone**()

Handle completion of a LAMMPS run.

void **setDocver**()

Set the documentation version string for help links.

void **autoSave**()

Perform an auto-save of the current file.

void **setFont**(const QFont &newfont)

Update the editor font.

Parameters
newfont – The font to apply to the editor

QWizardPage ***tutorialIntro**(int collection, int ntutorial, const QString &infotext)

Create an introduction page for a tutorial.

Parameters

- **collection** – Tutorial collection index
- **ntutorial** – Tutorial number within the collection
- **infotext** – Information text to display

Returns
Wizard page with tutorial introduction

QWizardPage ***tutorialDirectory**(int collection, int ntutorial)

Create a directory selection page for a tutorial.

Parameters

- **collection** – Tutorial collection index
- **ntutorial** – Tutorial number within the collection

Returns
Wizard page for tutorial directory selection

void **setupTutorial**(int collection, int tutno, const QString &dir, bool purgedir, bool getsolution, bool openwebpage)

Set up files and resources for a tutorial.

Parameters

- **collection** – Tutorial collection index
- **tutno** – Tutorial number within the collection
- **dir** – Directory to create files in
- **purgedir** – Whether to clean the directory first
- **getsolution** – Whether to include solution files
- **openwebpage** – Whether to open the tutorial web page

void **purgeInspectList**()

Clean up the inspect file dialog list.

bool **eventFilter**(QObject *watched, QEvent *event) override

Event filter for handling special events.

Parameters

- **watched** – Object being watched
- **event** – Event to filter

Returns

true if event was handled

Protected Attributes

int **nthreads**

Number of threads for parallel execution.

int **mainx**

Override value for main editor window width or 0.

int **mainy**

Override value for main editor window height or 0.

bool **hasClipboard**

true if Qt was configured with Clipboard support, otherwise false

---

TutorialWizard Class

class **TutorialWizard** : public QWizard

Wizard dialog for interactive LAMMPS tutorials.

TutorialWizard provides a step-by-step wizard interface for setting up and running LAMMPS tutorials. It guides users through directory selection, file preparation, and launching tutorial exercises.

Public Functions

TutorialWizard(int collection, int ntutorial, *LammpsGui* *lammpsgui, QWidget *parent = nullptr)

Construct a tutorial wizard.

Parameters

- **collection** – Tutorial collection index
- **ntutorial** – Tutorial number within the collection
- **lammpsgui** – Pointer to *LammpsGui* for sending signals
- **parent** – Parent widget

void **accept**() override

Accept the wizard and set up the tutorial.

Called when the user completes the wizard. Sets up tutorial files and opens the tutorial in the main window.

Tutorial Collections

The `TutorialCollection` struct (`src/tutorials.h`) is the single source of truth for the tutorial collections offered in the *Tutorials* menu. Each entry describes one independently hosted collection (its files repository, web pages, per-tutorial titles and blurbs, and how much of it is released). The menu and the `TutorialWizard` consume the table through the `tutorialCollections()` and `tutorialCollection()` accessors. See *Adding or updating a tutorial collection* for how to add or update a tutorial or a whole collection.

struct **TutorialCollection**

Metadata for one collection of LAMMPS-GUI tutorials.

LAMMPS-GUI ships support for multiple, independently hosted tutorial collections (e.g. the molecular soft-matter set and the materials-science set). Each collection is a numbered series of tutorials laid out identically – one `files/tutorial<N>/` folder per tutorial with a `.manifest` and an optional `solution/` subfolder – so the only thing that varies between collections is this metadata. The single source of truth is the table in tutorials.cpp, consumed by the Tutorials menu and the *TutorialWizard*.

Public Functions

inline int **count**() const

Number of tutorials in this collection.

inline QString **logoFor**(int n) const

Logo resource for tutorial `n` (1-based); resolves a "%1" pattern.

Public Members

QString **key**

stable identifier, e.g. "softmatter", "matsci"

QString **name**

display name for the menu and wizard

QString **dirPrefix**

working-directory folder prefix (e.g. "tutorial")

QString **author**

short author/attribution line for the intro page

QString **filesUrl**

raw files base, "…tutorial%1/%2" (number, filename)

QString **filesRepoUrl**

human-facing repository URL shown on the intro page

QString **webUrl**

per-tutorial web page pattern, "…%1/%2.html" (number, slug)

QString **siteUrl**

collection landing page (fallback web page)

QString **logo**

logo resource; may contain "%1" for per-tutorial logos

QStringList **titles**

short per-tutorial titles (*count()* entries)

QStringList **slugs**

per-tutorial web-page slugs (*count()* entries, or empty)

QStringList **blurbs**

per-tutorial HTML descriptions (*count()* entries)

bool **published**

true once the whole collection is released (hides the teaser label)

QString **status**

menu label for an unpublished collection ("coming soon", "planned")

int **available** = 0

number of leading tutorials that are launchable now; the remaining *count()*-available entries appear as disabled teasers

---

const QList<*TutorialCollection*> &**tutorialCollections**()

The available tutorial collections, in display order.

const *TutorialCollection* &**tutorialCollection**(int index)

Collection at `index`, clamped to a valid entry (0 if out of range)

Editor Components

CodeEditor Class

class **CodeEditor** : public QPlainTextEdit

Custom text editor with LAMMPS syntax support and auto-completion.

The *CodeEditor* class extends QPlainTextEdit to provide specialized features for editing LAMMPS input scripts:

- Line numbers in a margin area
- Syntax highlighting via *Highlighter*
- Context-aware auto-completion for LAMMPS commands
- Automatic indentation and formatting
- Context menu with LAMMPS-specific help
- Line highlighting for error visualization

Public Functions

CodeEditor(QWidget *parent = nullptr)

Constructor.

Parameters
parent – Parent widget (typically the main window)

~CodeEditor() override

Destructor.

CodeEditor() = delete

CodeEditor(const *CodeEditor*&) = delete

CodeEditor(*CodeEditor*&&) = delete

CodeEditor &**operator=**(const *CodeEditor*&) = delete

CodeEditor &**operator=**(*CodeEditor*&&) = delete

inline void **setSyntax**(const *LammpsSyntax* *newsyntax)

Set the syntax registry used for completion and help lookups.

Parameters
newsyntax – Syntax registry (not owned)

void **lineNumberAreaPaintEvent**(QPaintEvent *event)

Paint line numbers in the line number area.

Parameters
event – Paint event to handle

int **lineNumberAreaWidth**()

Calculate width needed for line number area.

Returns
Width in pixels

void **setFont**(const QFont &newfont)

Set editor font.

Parameters
newfont – Font to use for editor text

void **setCursor**(int block)

Set cursor to specific text block.

Parameters
block – Block number (line number) to position cursor

void **setHighlight**(int block, bool error)

Highlight a specific line (used for error indication)

Parameters

- **block** – Block number to highlight
- **error** – true for the error (red) highlight, false for the normal (green) one

inline void **setReformatOnReturn**(bool flag)

Enable/disable automatic reformatting on Enter key.

Parameters
flag – true to enable, false to disable

inline void **setAutoComplete**(bool flag)

Enable/disable automatic completion popup.

Parameters
flag – true to enable, false to disable

QString **reformatLine**(const QString &line)

Reformat a line with proper indentation.

Parameters
line – Line to reformat

Returns
Reformatted line

void **setCommandList**(const QStringList &words)

Set word list for LAMMPS command completion.

Parameters
words – List of command names

void **setFixList**(const QStringList &words)

Set word list for fix style completion.

Parameters
words – List of fix style names

void **setComputeList**(const QStringList &words)

Set word list for compute style completion.

Parameters
words – List of compute style names

void **setDumpList**(const QStringList &words)

Set word list for dump style completion.

Parameters
words – List of dump style names

void **setAtomList**(const QStringList &words)

Set word list for atom style completion.

Parameters
words – List of atom style names

void **setPairList**(const QStringList &words)

Set word list for pair style completion.

Parameters
words – List of pair style names

void **setBondList**(const QStringList &words)

Set word list for bond style completion.

Parameters
words – List of bond style names

void **setAngleList**(const QStringList &words)

Set word list for angle style completion.

Parameters
words – List of angle style names

void **setDihedralList**(const QStringList &words)

Set word list for dihedral style completion.

Parameters
words – List of dihedral style names

void **setImproperList**(const QStringList &words)

Set word list for improper style completion.

Parameters
words – List of improper style names

void **setKspaceList**(const QStringList &words)

Set word list for kspace style completion.

Parameters
words – List of kspace style names

void **setRegionList**(const QStringList &words)

Set word list for region style completion.

Parameters
words – List of region style names

void **setIntegrateList**(const QStringList &words)

Set word list for integration style completion.

Parameters
words – List of integration style names

void **setMinimizeList**(const QStringList &words)

Set word list for minimization style completion.

Parameters
words – List of minimization style names

void **setVariableList**(const QStringList &words)

Set word list for variable style completion.

Parameters
words – List of variable style names

void **setUnitsList**(const QStringList &words)

Set word list for units style completion.

Parameters
words – List of units style names

void **setExtraList**(const QStringList &words)

Set extra word list for completion.

Parameters
words – List of extra words

void **setColorList**(const QStringList &words)

Set list of color names for completion.

Parameters
words – List of color names

void **setImageKwList**(const QStringList &words)

Set list of dump image keywords for completion.

Parameters
words – List of dump image keywords

void **setGroupList**()

Update group ID list from the editor buffer.

void **setVarNameList**()

Update variable name list from the editor buffer and LAMMPS instance.

void **setComputeIDList**()

Update compute ID list from the editor buffer.

void **setFixIDList**()

Update fix ID list from the editor buffer.

void **setFileList**()

Update file list from current directory.

void **setVariableOverrides**(const QList<*VariableEntry*> &vars)

Set the variables list used to mark overridden index variable values.

Entries whose value overrides the definition in the input script (see isOverridden()) get a thin frame drawn around the value text of their definition line and a tooltip showing the overriding value.

Parameters
vars – Current variable entries (parse results plus dialog edits)

Public Static Attributes

static constexpr int **NO_HIGHLIGHT** = 1 << 30

Constant for disabled highlighting.

Protected Functions

void **resizeEvent**(QResizeEvent *event) override

Handle resize events to update line number area.

Parameters
event – The resize event

void **paintEvent**(QPaintEvent *event) override

Paint the editor and frame overridden index variable values.

Parameters
event – The paint event

bool **event**(QEvent *event) override

Handle tooltip events for overridden index variable values.

Parameters
event – The event to handle

Returns
true if the event was handled

bool **canInsertFromMimeData**(const QMimeData *source) const override

Check if MIME data can be inserted (for drag-and-drop)

Parameters
source – The MIME data to check

Returns
true if data can be inserted

void **dragEnterEvent**(QDragEnterEvent *event) override

Handle drag enter events.

Parameters
event – The drag enter event

void **dragLeaveEvent**(QDragLeaveEvent *event) override

Handle drag leave events.

Parameters
event – The drag leave event

void **dropEvent**(QDropEvent *event) override

Handle drop events.

Parameters
event – The drop event

void **contextMenuEvent**(QContextMenuEvent *event) override

Handle context menu events.

Parameters
event – The context menu event

void **keyPressEvent**(QKeyEvent *event) override

Handle key press events (for auto-completion and formatting)

Parameters
event – The key event

void **setDocver**()

Set LAMMPS documentation version for help links.

LineNumberArea Class

class **LineNumberArea** : public QWidget

Widget displaying line numbers alongside the code editor.

This widget is placed in the margin of the *CodeEditor* to show line numbers. All painting is delegated back to the *CodeEditor* class.

Public Functions

inline explicit **LineNumberArea**(*CodeEditor* *editor)

Constructor.

Parameters
editor – Pointer to the associated *CodeEditor*

~LineNumberArea() override = default

Destructor.

LineNumberArea() = delete

LineNumberArea(const *LineNumberArea*&) = delete

LineNumberArea(*LineNumberArea*&&) = delete

LineNumberArea &**operator=**(const *LineNumberArea*&) = delete

LineNumberArea &**operator=**(*LineNumberArea*&&) = delete

inline QSize **sizeHint**() const override

Get the ideal size for the line number area.

Returns
QSize with width from editor, height 0 (fills available height)

Protected Functions

inline void **paintEvent**(QPaintEvent *event) override

Paint event handler - delegates to *CodeEditor*.

Parameters
event – The paint event

Highlighter Class

class **Highlighter** : public QSyntaxHighlighter

Syntax highlighter for LAMMPS input scripts.

This class extends QSyntaxHighlighter to provide syntax highlighting for LAMMPS input files in the *CodeEditor*. Lines are lexed with the shared syntax engine (tokenizeLine()), so multi-line constructs – '&' line continuations (including mid-word joins), triple-quoted strings, and quoted strings spanning continuations – are tracked through the QSyntaxHighlighter block states and continuation lines are highlighted in the context of their logical command. Command words keep the historically grown per-category colors; arguments are colored by their role from the *LammpsSyntax* command spec table plus lexical classes (numbers, strings, comments, variable references, special words). Keywords that embed a second command in a command line – the "modify" of `write_dump` and the "dump" of `rerun` – are shown in the color of the command they stand for (`dump_modify` and `read_dump`, respectively) and the arguments after them are colored as arguments of that command. Command and style names that are unknown to the populated syntax registry are flagged with a wavy underline, except for the word currently being edited at the cursor.

Public Functions

Highlighter(const *LammpsSyntax* *syntax, QTextDocument *parent)

Constructor.

Parameters

- **syntax** – Syntax registry with name sets and command specs (not owned, may not be null)
- **parent** – Parent text document to highlight

~Highlighter() override = default

Destructor.

Highlighter() = delete

Highlighter(const *Highlighter*&) = delete

Highlighter(*Highlighter*&&) = delete

Highlighter &**operator=**(const *Highlighter*&) = delete

Highlighter &**operator=**(*Highlighter*&&) = delete

Public Slots

void **setCursorPos**(int blockNumber, int column)

Track the editing cursor to suppress unknown-name marking there.

The unknown-name underline is not shown for the word the cursor is on, so a partially typed name is not flagged. Re-highlights the previously active and the new block.

Parameters

- **blockNumber** – block (line) number of the cursor position
- **column** – column of the cursor within the block

Protected Functions

void **highlightBlock**(const QString &text) override

Highlight a single block (line) of text.

Parameters

text – The text to highlight

LammpsSyntax Class

class **LammpsSyntax**

Registry of LAMMPS syntax data for highlighting, completion, and checking.

Holds the sets of valid command and style names for each StyleCat plus the per-command argument role table. The name sets are populated by injection (normally from LAMMPS library introspection right after the LAMMPS instance is created; from literal lists in unit tests), so this class has no dependency on the LAMMPS library. The role table is loaded from the command spec resource table; additional tables can be loaded on top and override earlier entries.

Public Functions

LammpsSyntax()

Constructor; seeds the static keyword sets.

~LammpsSyntax() = default

Destructor.

LammpsSyntax(const *LammpsSyntax*&) = delete

LammpsSyntax(*LammpsSyntax*&&) = delete

LammpsSyntax &**operator=**(const *LammpsSyntax*&) = delete

LammpsSyntax &**operator=**(*LammpsSyntax*&&) = delete

void **setStyles**(*StyleCat* cat, const QStringList &names)

Set the full (unfiltered) list of valid names for one category.

The unfiltered set is used for name validity checks; a sorted list with accelerator-suffixed variants removed is derived for completions.

Parameters

- **cat** – category to populate
- **names** – full list of valid names for the category

inline void **setCommands**(const QStringList &names)

convenience alias for setStyles(StyleCat::Command, names)

bool **loadCommandSpecs**(const QString &path)

Load a command spec table from a file or Qt resource.

Parameters
path – file system path or Qt resource path of the table

Returns
true if the file could be read and contained no malformed entries

bool **loadCommandSpecsFromString**(const QString &text)

Load a command spec table from a string (tests, future overlays)

Later entries override earlier entries with the same command name. Malformed lines are skipped.

Parameters
text – complete table text

Returns
true if no malformed entries were found

inline bool **isPopulated**() const

true once command names have been populated; gates unknown-name checks

inline bool **knownCommand**(const QString &word) const

check whether a word is a known command name

bool **knownStyle**(*StyleCat* cat, const QString &name) const

check whether a name is in the full style set of a category

inline bool **isSpecialWord**(const QString &word) const

check for the special argument words (INF, EDGE, NULL, SELF, if/then/else/elif)

CmdCat **commandCategory**(const QString &cmd) const

display category for a command word (CmdCat::Other if not in the table)

inline int **commandIndex**(const QString &cmd) const

stable spec table index for a command (-1 if the command has no spec entry)

const *CommandSpec* ***spec**(int index) const

spec table entry by index (nullptr for invalid index)

ArgSpec **argSpec**(int cmdIndex, int argIndex) const

role of argument position `argIndex` (>= 1) of command spec `cmdIndex`

QStringList **completionList**(*StyleCat* cat, bool withNone) const

Sorted completion word list for a category.

Accelerator-suffixed style variants are filtered out.

Parameters

- **cat** – category of the word list
- **withNone** – prepend the "none" style (pair/bond/… styles)

CompletionTarget **completionTarget**(int prevBlockState, const QString &line, int cursorCol) const

Determine which completion applies at a cursor position.

Tokenizes the line with the given previous block state, locates the word under the cursor, and classifies it: the command completer on word 0 of a fresh logical line, and per-argument completers from the command spec table (style categories, group IDs, file names) plus the v_/c_/f_ reference prefixes and the read_data extra keywords. Because the previous block state carries the active command across '&' line continuations, completion works on continuation lines as well.

Parameters

- **prevBlockState** – block state after the previous line (-1 or 0 for none)
- **line** – physical line the cursor is on
- **cursorCol** – column of the cursor within the line

Returns

completer kind, style category, and the word under the cursor

---

InputScanner Class

class **InputScanner**

Assemble logical LAMMPS commands from the physical lines of a buffer.

Feeds physical lines through tokenizeLine() while threading the multi-line state, joins '&' continuations (including mid-word joins and quoted strings spanning lines), and reports each completed logical command as a list of words with their original line numbers. Comment and empty lines produce no commands. Tokenizer-level problems (unbalanced quotes, an unclosed triple-quoted block, or a dangling '&' on the final line) are collected as diagnostics. This is the input-scanning building block for whole-buffer consumers such as the pre-run syntax checker; the highlighter works on the per-block state directly instead.

Public Types

enum class **Diag** : quint8

tokenizer-level problems found while scanning

Values:

enumerator **UnbalancedQuote**

a single/double quote was left unclosed

enumerator **UnclosedTriple**

a triple-quoted block was still open at the end

enumerator **DanglingContinuation**

the final line ended with a '&' continuation

Public Functions

InputScanner() = default

Constructor.

~InputScanner() = default

Destructor.

InputScanner(const *InputScanner*&) = default

Copy constructor.

InputScanner(*InputScanner*&&) = default

Move constructor.

InputScanner &**operator=**(const *InputScanner*&) = default

Copy assignment.

InputScanner &**operator=**(*InputScanner*&&) = default

Move assignment.

void **scan**(const QString &buffer)

scan a whole buffer (split on newlines, 1-based line numbers); results accumulate, so use a fresh scanner for every buffer

inline const QVector<*LogicalCommand*> &**commands**() const

completed logical commands in buffer order

inline const QVector<*Diagnostic*> &**diagnostics**() const

diagnostics in detection order

struct **Diagnostic**

one diagnostic with the line it was detected on

Public Members

Diag **what** = *Diag*::*UnbalancedQuote*

kind of problem

int **line** = 0

1-based line number

struct **LogicalCommand**

one logical command with continuations joined

Public Members

int **firstLine** = 0

line number of the first physical line

int **lastLine** = 0

line number of the last physical line

QVector<*Word*> **words**

words in command order; words[0] is the command

struct **Word**

one whitespace-separated word of a logical command

Public Members

QString **text**

word text; surrounding matching quotes stripped

int **line** = 0

line number of the word's first character

bool **quoted** = false

word contains (or is) a quoted section

bool **hasSubst** = false

word contains a '$' variable reference

SyntaxChecker Class

class **SyntaxChecker**

Static lint checks for LAMMPS input scripts.

Scans an input buffer with the syntax engine's *InputScanner* and applies a set of deterministic checks against the introspected name registry: unknown commands and style names, quoting and line continuation problems, references to undefined variables and groups, references to computes, fixes, or variables defined nowhere in the buffer, missing input files, missing required arguments, and non-numeric arguments in a curated list of strictly numeric positions.

The design priority is the absence of false positives: any word containing a '$' substitution is exempt from every check, a substitution anywhere on a line disables its argument-count check, and script features that make

static analysis unreliable disable whole rule groups. Any of include files, jump/label loops, if/then commands, or python scripting disables all definition-tracking rules (variable, group, and reference checks); jump/label additionally downgrades unknown commands to warnings; shell, geturl, and include disable file-existence checks downstream; read_restart disables group and ID checks downstream; plugin disables unknown-name errors downstream. Only ERROR severity findings should gate a run.

Public Functions

inline explicit **SyntaxChecker**(const *LammpsSyntax* *syntax)

Constructor.

Parameters
syntax – Syntax registry with name sets and command specs (not owned, may not be null)

~SyntaxChecker() = default

SyntaxChecker() = delete

SyntaxChecker(const *SyntaxChecker*&) = delete

SyntaxChecker(*SyntaxChecker*&&) = delete

SyntaxChecker &**operator=**(const *SyntaxChecker*&) = delete

SyntaxChecker &**operator=**(*SyntaxChecker*&&) = delete

QList<*LintIssue*> **check**(const QString &buffer, const QStringList &presetVariables, const QString &cwd) const

Check an input buffer.

Parameters

- **buffer** – complete input script text
- **presetVariables** – variable names that are defined outside the script before the run (the *SetVariables* dialog entries and the always-present gui_run variable)
- **cwd** – directory that relative file names are resolved against

Returns
findings sorted by line number

Public Static Functions

static int **countErrors**(const QList<*LintIssue*> &issues)

number of ERROR severity findings in a result list

static QString **formatIssues**(const QList<*LintIssue*> &issues, int maxShown = -1)

Format findings as a plain text list, one finding per line.

Parameters

- **issues** – findings to format
- **maxShown** – maximum number of findings listed (-1 = all); when truncated, a final "… and N more" line is appended

Returns
the formatted list

struct **LintIssue**

one finding of the pre-run input check

Public Members

int **line** = 0

1-based physical line number

LintSeverity **severity** = *LintSeverity*::*Warning*

how serious the finding is

QString **message**

human readable description

enum class **LintSeverity** : quint8

severity of a single lint finding

Values:

enumerator **Warning**

suspicious, but may be intentional

enumerator **Error**

LAMMPS will reject this input.

Syntax Engine Functions

Functions

LineTokens **tokenizeLine**(const QString &text, int prevState = 0)

Split one physical line of a LAMMPS input into tokens with spans.

Mirrors the parsing rules of the LAMMPS Input class (lammps/src/input.cpp): trailing whitespace is trimmed before the '&' continuation check, '#' starts a comment unless it appears inside a quoted string, single, double, and triple quotes group text, and quoted strings may span '&' continuations. Multi-line state (open triple quote, continuation, open quote) is carried in the state integer, see *SyntaxState*.

The command spec index bits of the returned state are preserved from `prevState` while a logical line continues and reset to "none" when a new logical line starts; callers that track the active command patch the state with *SyntaxState::withCommand()* before storing it.

Parameters

- **text** – physical line (no trailing newline)
- **prevState** – block state after the previous line (-1 or 0 for none)

Returns

tokens with spans and the state after this line

QVector<*Token*> **findVarRefs**(const QString &text)

Find $x, ${name}, and variable references.

Follows the LAMMPS substitution rules: a '$' preceded by a backslash is escaped, '${' extends to the closing brace, '$(' extends to the balanced closing parenthesis (an optional ':format' suffix is part of the expression), otherwise the single following character names the variable. References are reported everywhere including inside quoted strings, since commands like print, if, and python substitute their quoted arguments.

Parameters
text – text to scan

Returns
VarRef tokens with spans in scan order

QHash<int, QString> **argumentTexts**(const *LineTokens* &lt, const QString &line)

Collect the full text of each argument of a tokenized line.

Joins the tokens belonging to the same argument (words may consist of several tokens when they contain quoted sections), skipping comment and continuation tokens.

Parameters

- **lt** – result of tokenizeLine() for the line
- **line** – the tokenized line

Returns
map from argument index to the argument's full text

bool **isNumberWord**(const QString &word)

Check whether a word is a number the way the LAMMPS-GUI editor colors them.

Accepts integers, floating point numbers with optional e/E/d/D exponent, and integer ranges / type wildcards built from digits, ':', and '*'.

Parameters
word – word to test

Returns
true if the word is number-like

Syntax Engine Types

enum class **StyleCat** : quint8

Style-list categories known to the syntax registry.

The first group mirrors the categories that can be enumerated through the LAMMPS library interface (lammps_style_count() / lammps_style_name()); Variable, Units, Extra, and ImageKw are static keyword sets that LAMMPS does not expose through introspection, and Color is injected from the shared color name list (see lammpsImageColors()).

Values:

enumerator **Command**

command names (input commands + registered command styles)

enumerator **Fix**

fix styles

enumerator **Compute**

compute styles

enumerator **Dump**

dump styles

enumerator **Atom**

atom styles

enumerator **Pair**

pair styles

enumerator **Bond**

bond styles

enumerator **Angle**

angle styles

enumerator **Dihedral**

dihedral styles

enumerator **Improper**

improper styles

enumerator **Kspace**

kspace styles

enumerator **Region**

region styles

enumerator **Integrate**

integrator styles (run_style)

enumerator **Minimize**

minimizer styles (min_style)

enumerator **Variable**

variable command styles (static set)

enumerator **Units**

units command arguments (static set)

enumerator **Extra**

read_data / create_box “extra/…” keywords (static set)

enumerator **Color**

color names accepted by the dump image command (injected)

enumerator **ImageKw**

keywords of the dump image command (static set)

enumerator **None**

no style category (used in *ArgSpec* for non-style roles)

enum class **CmdCat** : quint8

Display category of a command word.

These are the historically grown highlight classes of the LAMMPS-GUI editor; the mapping of command names to categories is read from the command spec table and must be kept consistent with the colors users are accustomed to.

Values:

enumerator **Lattice**

system setup / lattice / geometry commands

enumerator **Output**

file I/O and output commands

enumerator **Modify**

*_modify commands

enumerator **Particle**

particle, force field, and definition commands

enumerator **Run**

run-like commands

enumerator **Setup**

settings commands

enumerator **Special**

undo / flow control commands

enumerator **Other**

recognized command without an explicit category entry

enum class **ArgRole** : quint8

Role of an argument position within a command.

Values:

enumerator **Any**

no specific role; only lexical highlighting applies

enumerator **DefineId**

an ID defined by this command (fix/compute/dump/region/variable ID)

enumerator **GroupId**

reference to an atom group ID

enumerator **Style**

a style name; the category is in *ArgSpec::cat*

enumerator **SubStyle**

may name a sub-style of a hybrid style; recognized when the word is a member of the category's style set, like LAMMPS itself separates hybrid sub-styles from their arguments

enumerator **File**

a file name argument

enumerator **Int**

an integer number is expected (informational in the spec table for now; the checker uses its own vetted numeric list)

enumerator **Number**

a floating point number is expected (informational, see Int)

enumerator **Keyword**

a keyword-like argument

enumerator **Label**

reference to an ID defined elsewhere (unfix/jump/label targets)

enum class **TokType** : quint8

Type of a token produced by tokenizeLine() / findVarRefs()

Values:

enumerator **Word**

unquoted (part of a) word

enumerator **String**

single- or double-quoted string span

enumerator **TripleString**

triple-quoted string span

enumerator **Comment**

comment from '#' to end of line

enumerator **Continuation**

trailing '&' line continuation character

enumerator **VarRef**

$x, ${name}, or variable reference (findVarRefs() only)

enum class **CompleterKind** : quint8

Which completion word list applies at a completion location.

Values:

enumerator **None**

no completion at this location

enumerator **Command**

command names (word 0 of a logical line)

enumerator **Style**

style names; the category is in *CompletionTarget::cat*

enumerator **Group**

group IDs

enumerator **VarName**

variable name references (v_…)

enumerator **ComputeId**

compute ID references (c_…)

enumerator **FixId**

fix ID references (f_…)

enumerator **File**

file names

enumerator **Extra**

read_data / create_box "extra/…" keywords

struct **ArgSpec**

Role and (for ArgRole::Style) style category of one argument position.

Public Members

ArgRole **role** = *ArgRole*::*Any*

role of this argument position

StyleCat **cat** = *StyleCat*::*None*

style category when role == ArgRole::Style

struct **CommandSpec**

Per-command syntax information loaded from the command spec table.

Public Members

QString **name**

command name

CmdCat **category** = *CmdCat*::*Other*

display category of the command word

int **minArgs** = 0

minimum number of required arguments

QVector<*ArgSpec*> **args**

roles of argument positions 1, 2, ...

bool **repeatLast** = false

repeat the last role for excess arguments

struct **Token**

One token within a physical line of a LAMMPS input.

Adjacent tokens with the same argIndex belong to the same argument (words may contain embedded quoted sections).

Public Members

int **start** = 0

offset of the first character within the line

int **length** = 0

number of characters

TokType **type** = *TokType*::*Word*

token type

int **argIndex** = -1

0 = command word, > 0 argument position, -1 = not an argument

bool **isNumber** = false

word parses as a LAMMPS number, range, or type wildcard

bool **hasSubst** = false

word contains a '$' variable reference

bool **fragment** = false

word is split across a mid-word '&' continuation

struct **LineTokens**

Result of tokenizing one physical line.

Public Members

QVector<*Token*> **tokens**

tokens in line order

int **outState** = 0

block state after this line (see *SyntaxState*)

bool **unbalancedQuote** = false

a single/double quote was left unclosed

struct **CompletionTarget**

Result of *LammpsSyntax::completionTarget()*

Public Members

CompleterKind **kind** = *CompleterKind*::*None*

which completer applies

StyleCat **cat** = *StyleCat*::*None*

style category when kind == Style

int **wordStart** = -1

start column of the word under the cursor

int **wordLength** = 0

length of the word under the cursor

namespace **SyntaxState**

Packing helpers for the per-block syntax state integer.

The state is stored in QSyntaxHighlighter block states and threaded through tokenizeLine() so multi-line constructs (triple-quoted strings, '&' line continuations, quoted strings spanning continuations) are lexed consistently. Layout: bits 0-5 flags, bits 6-18 command spec index + 1 (0 = no spec), bits 19-26 number of arguments consumed so far.

Functions

inline int **flags**(int state)

extract the flag bits from a block state

inline int **cmdIndex**(int state)

extract the command spec index from a block state (-1 if none)

inline int **argsUsed**(int state)

extract the number of arguments consumed so far from a block state

inline int **pack**(int flagbits, int cmdidx, int nargs)

combine flags, command spec index (-1 for none), and argument count into a block state

inline int **withCommand**(int state, int cmdidx)

replace the command spec index in a block state (-1 for none)

inline int **withArgs**(int state, int nargs)

replace the number of arguments consumed so far in a block state; used to renumber the arguments when a command embeds a second command (the “modify” section of write_dump, the “dump” section of rerun)

inline bool **logicalContinues**(int state)

true when a line with this previous-line state continues the logical line (an open continuation, triple-quoted block, or quoted string)

Variables

constexpr int **TRIPLE** = 1 << 0

inside an open triple-quoted string

constexpr int **CONTINUE** = 1 << 1

previous line ended with ‘&’: logical line continues

constexpr int **MIDWORD** = 1 << 2

the ‘&’ directly followed a word character

constexpr int **COMMENT** = 1 << 3

the continuation is inside a comment

constexpr int **SINGLEQ** = 1 << 4

inside an open single-quoted string

constexpr int **DOUBLEQ** = 1 << 5

inside an open double-quoted string

constexpr int **FLAG_MASK** = 0x3f

mask covering all flag bits

constexpr int **CMD_SHIFT** = 6

bit position of the command spec index

constexpr int **CMD_MASK** = 0x1fff

13 bits for the command spec index + 1

constexpr int **ARG_SHIFT** = 19

bit position of the argument counter

constexpr int **ARG_MASK** = 0xff

8 bits for the argument counter

constexpr int **ARG_MAXX** = *ARG_MASK*

saturation value of the argument counter

LAMMPS Interface

LammpsWrapper Class

class **LammpsWrapper**

C++ wrapper for the LAMMPS C library interface.

This class provides a C++-oriented interface to the LAMMPS library, routing all library calls through a unified API. It manages the LAMMPS instance and handles dynamic loading of the library when the application is built in plugin mode.

Public Types

enum **StyleConst**

Constants for variable styles.

Values:

enumerator **EQUAL_STYLE**

enumerator **ATOM_STYLE**

enumerator **VECTOR_STYLE**

enumerator **STRING_STYLE**

enum **ScopeConst**

Constants for data scopes.

Values:

enumerator **GLOBAL_STYLE**

enumerator **DUMMY**

enumerator **LOCAL_STYLE**

enum **TypeConst**

Constants for data types.

Values:

enumerator **SCALAR_TYPE**

enumerator **VECTOR_TYPE**

enumerator **ARRAY_TYPE**

enumerator **NUM_ROWS**

enumerator **NUM_COLS**

Public Functions

LammpsWrapper()

Constructor - initializes wrapper.

~LammpsWrapper()

Destructor.

Does not close an open LAMMPS instance. Callers must invoke *close()* explicitly before destroying the wrapper. In plugin mode the handle to the dynamically loaded LAMMPS library is released.

LammpsWrapper(const *LammpsWrapper*&) = delete

LammpsWrapper(*LammpsWrapper*&&) = delete

LammpsWrapper &**operator=**(const *LammpsWrapper*&) = delete

LammpsWrapper &**operator=**(*LammpsWrapper*&&) = delete

void **open**(int nargs, char **args)

Create a new LAMMPS instance.

Parameters

- **nargs** – Number of command-line arguments
- **args** – Command-line arguments array

void **close**()

Close the LAMMPS instance.

void **finalize**()

Finalize MPI (if used) and close LAMMPS.

void **file**(const QString &fname)

Process commands from a LAMMPS input file.

Parameters
fname – Filename as Qt-style QString

void **command**(const QString &cmd)

Execute a single LAMMPS command.

Parameters
cmd – Command string as Qt-style QString

void **commandsString**(const QString &cmd)

Execute multiple LAMMPS commands from a string.

Parameters
cmd – Commands string with newlines as Qt-style QString

void **forceTimeout**()

Force a timeout condition in LAMMPS.

int **version**()

Get LAMMPS version number.

Returns
Version number as integer (YYYYMMDD format)

int **extractSetting**(const char *keyword)

Extract a global setting from LAMMPS.

Parameters
keyword – Setting name to extract

Returns
Integer value of the setting

void ***extractGlobal**(const char *keyword)

Extract a pointer to global data from LAMMPS.

Parameters
keyword – Name of global data to extract

Returns
Pointer to the data

void ***extractPair**(const char *keyword)

Extract pair style data from LAMMPS.

Parameters
keyword – Name of pair data to extract

Returns
Pointer to the pair data cast to a void pointer

void ***extractAtom**(const char *keyword)

Extract atom data from LAMMPS.

Parameters
keyword – Name of atom data to extract

Returns
Pointer to the atom data cast to void pointer

void *extractCompute(const QString &id, int style, int type)

Extract data from a compute from LAMMPS.

Parameters

- **id** – compute id as a QString
- **style** – style of data to extract
- **type** – type of data to extract

Returns

data cast to a void pointer

void *extractFix(const QString &id, int style, int type, int nrow, int ncol)

Extract data from a fix from LAMMPS.

Parameters

- **id** – fix id as a QString
- **style** – style of data to extract
- **type** – type of data to extract
- **nrow** – row index (only for global)
- **ncol** – column index (only for global)

Returns

data cast to a void pointer. Must be freed for global elements

double extractVariable(const char *keyword)

Extract a variable value from LAMMPS.

Parameters

keyword – Variable name to extract

Returns

Value of the variable as double

int extractVariableDatatype(const QString &keyword)

Extract style of a variable from LAMMPS.

Parameters

keyword – variable name as a QString

Returns

Value type of variable as integer

int hasId(const char *idtype, const char *id)

Check if a compute/fix/variable ID exists.

Parameters

- **idtype** – Type of ID (“compute”, “fix”, “variable”)
- **id** – The ID to check

Returns

1 if exists, 0 otherwise

int **idCount**(const char *idtype)

Get count of IDs of a specific type.

Parameters
idtype – Type of ID (“compute”, “fix”, “variable”, “group”)

Returns
Number of IDs of that type

QString **idName**(const char *idtype, int idx)

Get the name of an ID by index.

Parameters

- **idtype** – Type of ID (“compute”, “fix”, “variable”, “group”, …)
- **idx** – Index of the ID

Returns
The ID name, or an empty string on error

int **styleCount**(const char *keyword)

Get count of styles of a specific type.

Parameters
keyword – Type of style (“compute”, “fix”, “pair”, etc.)

Returns
Number of available styles

QString **styleName**(const char *keyword, int idx)

Get the name of a style by index.

Parameters

- **keyword** – Type of style (“command”, “pair”, “fix”, …)
- **idx** – Index of the style

Returns
The style name, or an empty string on error

QString **variableInfo**(int idx)

Get the name of a variable by index.

Parameters
idx – Variable index

Returns
The variable name, or an empty string on error

double **getThermo**(const char *keyword)

Get current value of a thermodynamic quantity.

Parameters
keyword – Thermo keyword

Returns
Value of the thermo quantity

void ***lastThermo**(const char *keyword, int idx)

Get a specific value from last thermo output.

Parameters

- **keyword** – Thermo keyword
- **idx** – Index for vector quantities

Returns
Pointer to the value

template<typename **T**>
inline *T* **lastThermoAs**(const char *keyword, int idx)

Typed read of a cached last-thermo value.

Template Parameters
T – scalar type the thermo value points to (int, int64_t, double)

Parameters

- **keyword** – Thermo keyword (“step”, “num”, “type”, “data”, …)
- **idx** – Index for vector quantities

Returns
The dereferenced value, or a value-initialized T if the query returns null

inline QString **lastThermoString**(const char *keyword, int idx)

Read a string-valued cached last-thermo entry.

Parameters

- **keyword** – Thermo keyword that returns text (e.g. “keyword”, “imagename”)
- **idx** – Index for vector quantities

Returns
The value as a QString (empty if unavailable)

inline bool **isOpen**() const

Check if LAMMPS instance is open.

Returns
true if LAMMPS is initialized, false otherwise

bool **isRunning**()

Check if LAMMPS is currently executing a run.

Returns
true if running, false otherwise

bool **hasError**() const

Check if LAMMPS has encountered an error.

Returns
true if error occurred, false otherwise

int **getLastErrorMessage**(char *errorbuf, int buflen)

Get the last error message from LAMMPS.

Parameters

- **errorbuf** – Buffer to store error message
- **buflen** – Length of buffer

Returns
Error type code

QString **lastErrorMessage**()

Get the last error message from LAMMPS as a string.

Convenience wrapper around *getLastErrorMessage()* that manages the character buffer internally. Retrieving the message also clears the pending error state in LAMMPS.

Returns
The error text, or an empty string if no error is pending

bool **configAccelerator**(const char *package, const char *category, const char *setting) const

Check if an accelerator package is available.

Parameters

- **package** – Package name
- **category** – Category name
- **setting** – Setting name

Returns
true if available, false otherwise

bool **configHasPackage**(const char *pkg) const

Check if a package is included in LAMMPS build.

Parameters
pkg – Package name

Returns
true if included, false otherwise

bool **configHasCurlSupport**() const

Check if LAMMPS was built with CURL support.

Returns
true if CURL is available, false otherwise

bool **configHasOmpSupport**() const

Check if LAMMPS was built with OpenMP support.

Returns
true if OpenMP is available, false otherwise

bool **configHasPngSupport**() const

Check if LAMMPS was compiled with PNG format image support.

Returns
true if PNG image format support is available, false if not

bool **configHasJpegSupport**() const

Check if LAMMPS was compiled with JPEG format image support.

Returns
true if JPEG image format support is available, false if not

bool **hasGpuDevice**() const

Check if GPU device is available for GPU package.

Returns
true if GPU device found, false otherwise

bool **loadLib**(const QString &fname)

Load LAMMPS shared library (plugin mode)

Parameters
fname – Library filename (QString version)

Returns
true on success, false on failure

bool **hasPlugin**() const

Check if running in plugin mode.

Returns
true if plugin mode enabled, false if linked mode

LammpsRunner Class

class **LammpsRunner** : public QThread

Worker thread for executing LAMMPS simulations.

This class runs LAMMPS simulations in a background thread to maintain UI responsiveness during long calculations. It executes either a LAMMPS command string or a full input file and emits a signal upon completion.

Input is passed using std::string, ensuring clean ownership transfer.

Public Functions

explicit **LammpsRunner**(QObject *parent = nullptr)

Constructor.

Parameters
parent – Parent QObject

~LammpsRunner() override = default

Destructor.

LammpsRunner(const *LammpsRunner*&) = delete

LammpsRunner(*LammpsRunner*&&) = delete

LammpsRunner &**operator=**(const *LammpsRunner*&) = delete

LammpsRunner &**operator=**(*LammpsRunner*&&) = delete

void **setupRun**(*LammpsWrapper* *_lammps, std::string _input, std::string _file = {}, bool _clearfirst = true)

Prepare the runner thread with LAMMPS instance and commands.

Sets up the runner with the LAMMPS instance and input. Clears any previous LAMMPS state with the "clear" command, unless _clearfirst is false, which continues from the current state (used to extend a previous run). Either input or file should be provided, not both.

Parameters

- **_lammps** – Pointer to *LammpsWrapper* instance
- **_input** – String of LAMMPS commands to execute (can be empty)

- **_file** – Input file path to execute (can be empty)
- **_clearfirst** – If true, wipe the current LAMMPS system state first

Signals

void **resultReady**()

Signal emitted when LAMMPS execution completes.

Protected Functions

void **run**() override

Thread execution function - runs LAMMPS commands or input file.

This function executes in the worker thread. It processes either a string of LAMMPS commands or an input file, then signals completion.

Visualization Components

ChartWindow Class

class **ChartWindow** : public QWidget

Window for displaying and managing multiple time-series charts.

ChartWindow provides a GUI for displaying and manipulating multiple charts showing time-series data from LAMMPS simulations (thermodynamic output). It supports data smoothing, zooming via range sliders, and export to various formats (PNG, CSV, YAML, DAT).

Public Functions

explicit **ChartWindow**(const QString &filename, *LammpsGui* *lammpsgui = nullptr, QWidget *parent = nullptr)

Constructor.

Parameters

- **filename** – Path to the log file containing the data
- **lammpsgui** – Pointer to *LammpsGui* for sending signals (optional; nullptr for a standalone window with no live simulation)
- **parent** – Parent widget

inline int **numCharts**() const

Get the number of charts currently displayed.

Returns
Number of charts

inline bool **hasTitle**(const QString &title, int index) const

Check if a chart at given index has the specified title.

Parameters

- **title** – Title to check
- **index** – Chart index

Returns
true if chart has the title, false otherwise

int **getStep**() const

Get the current simulation step number.

Returns
Current step

void **resetCharts**()

Reset all charts to initial state.

void **reset**(const QString &filename)

Clear the window for a new run.

Drops all charts and reference lines and re-reads the chart preferences, so the window can be reused instead of destroyed and recreated for every run. Keeps the position and size the window currently has on screen.

Parameters
filename – Name of the log/input file the new run belongs to

void **resetZoom**()

Manually update chart display zoom status.

This is needed at an end of a run when the run finishes too quickly and the regular chart update has not yet triggered.

void **addChart**(const QString &title, int index)

Add a new chart to the window.

Parameters

- **title** – Chart title (thermodynamic property name)
- **index** – Chart index

void **addData**(int step, double value, int index)

Add a data point to a chart.

Parameters

- **step** – Simulation step number
- **data** – Data value
- **index** – Chart index

void **setUnits**(const QString &_units)

Set the units displayed for thermodynamic quantities.

Parameters
_units – Units string (e.g., “real”, “metal”, “lj”)

void **setNorm**(bool norm)

Enable/disable data normalization.

Parameters
norm – true to normalize data, false otherwise

void **setRangeEnabled**(bool enabled)

Enable/Disable range sliders.

Parameters

enabled – true to enable sliders and false to disable them

void **loadData**(const *PlotData* &table, int xcol, const QList<int> &ycols, const *PlotErrors* &yerrs = {})

Populate the window from an external data table (file plotting)

Replaces any existing charts; each selected y column becomes a chart titled by its column name, with the x axis labeled by the x column. Unlike the live thermo feed this loads all rows in one shot.

Parameters

- **data** – Parsed column data
- **xcol** – Index of the column to use as the shared x axis
- **ycols** – Indices of the columns to plot, one chart each
- **yerrs** – Optional error bars, indexed like the columns of `table`; an empty or wrongly sized entry means that column has none. Their lower half is used where it is filled in

Signals

void **resultWindowCreated**(*ChartWindow* *window, const QString &title)

A post-processing analysis opened a new chart window.

Emitted for the analyses whose result has a new x axis (autocorrelation, Fourier transform, structure factor) and therefore opens a chart window of its own. The receiver decides how to present it – *LammpsGui* hands it to its *WindowLayout*, so in the docked layout it becomes a tab beside the charts instead of a free window. Without a receiver the window shows itself.

Parameters

- **window** – The new window (self-deleting on close), not yet shown
- **title** – Short label for a tab in a combined layout

Protected Functions

void **closeEvent**(QCloseEvent *event) override

Handle window close event.

Parameters

event – Close event

ChartColumn Struct

`ChartWindow` owns one `ChartColumn` per thermo column: a plain, move-only data holder (`src/chartviewer.h`) that bundles the column's `PlotSeries` objects, cached data bounds, smoothing parameters, display style, and overlay/reference-line state. The single `ChartViewer` is rebound (via `setColumn()`) to whichever column is currently selected.

struct **ChartColumn**

The per-column data and display state of one chart.

Holds everything specific to a single plotted column: its neutral *PlotSeries* objects, cached data bounds, smoothing parameters, display style, and the overlay/reference-line state. It is a plain (non-QWidget) value type, move-only because of its unique_ptr<PlotSeries> members, so that the single shared renderer can be pointed at any column on demand.

Public Members

int **index** = -1

Chart index (thermo column id)

double **lastX** = -1.0

Last (largest) x value appended.

double **rawXmin** = 1.0e100

Running min x of the raw series (live path)

double **rawXmax** = -1.0e100

Running max x of the raw series.

double **rawYmin** = 1.0e100

Running min y of the raw series.

double **rawYmax** = -1.0e100

Running max y of the raw series.

int **window** = 10

Smoothing window.

int **order** = 4

Smoothing polynomial order.

std::unique_ptr<*PlotSeries*> **series**

Raw data series (always present)

std::unique_ptr<*PlotSeries*> **smooth**

Smoothed data series (created on demand)

std::unique_ptr<*PlotSeries*> **scatter**

Raw data as points (created on demand)

std::unique_ptr<*PlotSeries*> **smoothScatter**

Processed data as points (created on demand)

std::unique_ptr<*PlotSeries*> **fit**

Optional fit-curve overlay (created on demand)

QElapsedTimer **lastUpdate**

Time since the last chart update.

bool **doRaw** = true

Show raw data series.

bool **doSmooth** = false

Show smoothed data series.

bool **custom** = false

True when a custom curve (fit/function/overlay) takes the processed-series slot (visibility follows doSmooth)

ChartDisplayMode **dispmode** = ChartDisplayMode::Lines

How the raw series is drawn.

QColor **rawColor**

Raw series color override (invalid = configured default)

qreal **rawWidth** = Cfg::LINE_WIDTH_DEFAULT

Raw series line width.

qreal **rawPointSize** = Cfg::POINT_SIZE_DEFAULT

Raw series marker diameter.

ChartDisplayMode **smoothmode** = ChartDisplayMode::Lines

How the processed series is drawn.

QColor **smoothcolor**

Processed series color (invalid = configured default)

qreal **smoothwidth** = Cfg::LINE_WIDTH_DEFAULT

Processed series line width.

qreal **smoothpointsize** = Cfg::POINT_SIZE_DEFAULT

Processed series marker diameter.

QColor **errColor**

Error bar color of every series of this column (invalid = configured)

qreal **errWidth** = Cfg::ERR_WIDTH_DEFAULT

Error bar line width.

std::vector<std::unique_ptr<*PlotSeries*>> **overlaySeries**

Extra series from secondary files.

std::vector<std::unique_ptr<*PlotSeries*>> **vlines**

Reference line series (decorative)

QList<RefLine> **reflineDefs**

Reference line definitions (parallel to vlines)

QString **yTitle**

This column's Y-axis label (restored on the shared plot)

QString **procLabel** = QStringLiteral("Smooth")

Label of the processed-series slot in the Plot combo ("Smooth", or a fit/function name)

ChartViewer Class

class **ChartViewer** : public QWidget

Rebindable view of a single *ChartColumn*.

ChartViewer renders whichever *ChartColumn* it is currently bound to (via *setColumn()*) with its *PlotWidget*, supporting both raw and smoothed data display. The column data objects are owned by *ChartWindow*; this class is only the view.

See also

PlotWidget, *PlotSeries*, *ChartColumn*

Public Functions

explicit **ChartViewer**(QWidget *parent = nullptr)

Constructor – creates the renderer; no column is bound yet.

Parameters
parent – Parent widget

~ChartViewer() override

Destructor.

ChartViewer(const *ChartViewer*&) = delete

ChartViewer(*ChartViewer*&&) = delete

ChartViewer &**operator=**(const *ChartViewer*&) = delete

ChartViewer &**operator=**(*ChartViewer*&&) = delete

void **setColumn**(*ChartColumn* *c)

Bind this view to a column (or nullptr) and render it.

Unregisters the previous column's series from the shared plot and (re)attaches and draws `c`, restoring its Y-axis label. The column is owned by *ChartWindow*, not by the viewer.

void **addPoint**(double x, double y)

Append an (x, y) data point to the chart.

The point is appended only if **x** is greater than the last appended x; this keeps the live thermo feed monotonic in the step number.

Parameters

- **x** – X value (e.g. the simulation step, or an arbitrary abscissa)
- **y** – Y value

QRectF **getMinMax**() const

Get the min/max bounds of the data.

Returns
Rectangle containing data bounds

void **setXAxisRange**(double min, double max)

Set the displayed X-axis range (used by the range sliders)

void **setYAxisRange**(double min, double max)

Set the displayed Y-axis range (used by the range sliders)

void **resetZoom**()

Reset zoom to show all data.

void **updateSmooth**()

Recalculate and update smoothed data.

inline int **getCount**() const

Get number of data points.

Returns
Number of points in series

QString **getName**() const

Get the series (data column) name.

This is the per-column identifier (e.g. the thermo keyword), distinct from the shared plot title. Used for data-export column headers.

Returns
The property/column name this chart was created with

inline double **getStep**(int index) const

Get step number at given index.

Parameters
index – Data point index

Returns
Step number (X value)

inline double **getData**(int index) const

Get data value at given index.

Parameters
index – Data point index

Returns
Data value (Y value)

inline double **getError**(int index) const

Get the error bar half-height at a given index.

Asymmetric bars are reported by their mean half-height, which is what a fit weighted by the uncertainty can use as a standard deviation.

Parameters
index – Data point index

Returns
Half the extent of the error bar, or 0 if the series has none

void **setTLabel**(const QString &tlabel)

Set chart title.

Parameters
tlabel – New title

void **setYLabel**(const QString &ylabel)

Set Y-axis label.

Parameters
ylabel – New Y-axis label

void **setXLabel**(const QString &xlabel)

Set the X-axis label.

Parameters
xlabel – New X-axis label

void **setXLabelFormat**(const QString &fmt)

Set the X-axis tick label format.

Call after *setXLabel()* when the x-axis carries non-integer data (e.g. lattice constants from a Plot Data file). The thermo live-feed path leaves the default integer format in place.

Parameters
fmt – printf-style format string (e.g. "%.6g" for floating-point, "%d" for integer steps)

void **addOverlaySeries**(const QList<QPointF> &pts, const QString &name, const QColor &color, const QList<double> &yerr = {}, const QList<double> &yerrLo = {})

Add an overlay data series from a second file.

Overlay series are always shown in full (no smoothing); they are included in the axis range calculation.

Parameters

- **pts** – (x, y) data points

- **name** – Series name (shown as a tooltip / legend entry)
- **color** – Line color
- **yerr** – Optional error bars, one per point (ignored if the size differs)
- **yerrLo** – Optional lower half of asymmetric error bars (empty = symmetric)

inline int **overlaySeriesCount**() const

Number of overlay series currently displayed.

void **setReferenceLines**(const QList<RefLine> &lines)

Set reference lines (replaces any existing set)

Each line spans the full data range perpendicular to its orientation and is updated on every zoom reset. Lines are described by their orientation, position, label (series name), anchor, and color.

Parameters

lines – List of reference line descriptors

void **setLegendPos**(LegendPos pos)

Set the in-plot legend placement (corner, or off)

void **setRefLabelStyle**(double pointSize, double distance, bool boxed)

Set the window-wide reference-label style (font size, gap, boxed)

void **setDisplayStyle**(ChartDisplayMode mode, const QColor &color, qreal width, qreal pointSize)

Set how the raw data series is displayed.

Parameters

- **mode** – Lines, points, or both
- **color** – Series color (invalid color falls back to the theme default)
- **width** – Line width (used for the line and lines+points modes)
- **pointSize** – Marker diameter (used for the points and lines+points modes)

inline ChartDisplayMode **displayMode**() const

Current display mode.

inline QColor **displayColor**() const

Current series color (may be invalid, meaning the theme default)

inline qreal **displayWidth**() const

Current line width.

inline qreal **displayPointSize**() const

Current marker diameter.

void **setSmoothStyle**(ChartDisplayMode mode, const QColor &color, qreal width, qreal pointSize)

Set how the processed (smoothed) data series is displayed.

Parameters

- **mode** – Lines, points, or both
- **color** – Series color (invalid color falls back to the theme default)
- **width** – Line width (used for the line and lines+points modes)
- **pointSize** – Marker diameter (used for the points and lines+points modes)

inline ChartDisplayMode **smoothMode**() const

Current processed-series display mode.

inline QColor **smoothColor**() const

Current processed-series color (may be invalid, meaning the theme default)

inline qreal **smoothWidth**() const

Current processed-series line width.

inline qreal **smoothPointSize**() const

Current processed-series marker diameter.

void **setErrorStyle**(const QColor &color, qreal width)

Set how the error bars of this chart are drawn.

The style applies to every series of the chart that carries error bars, so that the bars read as one annotation layer rather than as part of the curve they belong to.

Parameters

- **color** – Bar color (invalid color falls back to the color of the series)
- **width** – Bar line width; the end caps grow with it

inline QColor **errorColor**() const

Current error bar color (may be invalid, meaning the series color)

inline qreal **errorWidth**() const

Current error bar line width.

void **setFitCurve**(const QList<QPointF> &points, const QString &name = QString())

Put a custom curve into the processed-series slot of the chart.

The curve – a fitted curve, an evaluated custom function, or a column overlay – takes the place of the Savitzky-Golay smooth while it is set: its visibility follows the Raw/Smoothed/Both choice (hidden in Raw mode), and *clearFitCurve()* returns the slot to the smooth.

Parameters

- **points** – Curve points (x, y) drawn as an overlay line; created on the first call and replaced on subsequent calls
- **name** – Optional series name for the overlay (e.g. the fitted expression or a user label); shown wherever series names are surfaced

inline bool **hasCustom**() const

True when a custom curve occupies the processed-series slot.

void **clearFitCurve**()

Remove the fit-curve overlay set with *setFitCurve()*

The overlay curve is emptied and hidden and the processed-series slot falls back to the Savitzky-Golay smooth of the raw data, which is recomputed on the redraw this triggers.

QString **getXLabel**() const

Get X-axis label.

Returns

X-axis label

QString **getYLabel**() const

Get Y-axis label.

Returns
Y-axis label

PlotWidget Class

Native `QWidget` + `QPainter` 2D line/scatter chart renderer (`src/plotwidget.h`). It is the sole chart backend; `ChartViewer` feeds it neutral `PlotSeries` / `PlotAxis` model objects plus the Qt-free axis-layout helpers from `plotaxismath` – no Qt Charts, Graphs, or QML.

class **PlotWidget** : public QWidget

QPainter-based renderer for the neutral chart model.

Series objects are referenced, not owned: the caller owns each *PlotSeries* and registers / unregisters it here, then calls update() after changing its data or style.

Public Functions

explicit **PlotWidget**(QWidget *parent = nullptr)

Constructor.

Parameters
parent – Parent widget

void **setTitle**(const QString &title)

Set the chart title (drawn centered at the top)

void **setXTitle**(const QString &title)

Set the X-axis title.

QString **xTitle**() const

Get the X-axis title.

void **setYTitle**(const QString &title)

Set the Y-axis title.

QString **yTitle**() const

Get the Y-axis title.

void **setXRange**(double min, double max)

Set the displayed X-axis range.

void **setYRange**(double min, double max)

Set the displayed Y-axis range.

void **setXLabelFormat**(const QString &fmt)

Set the printf-style tick label format of the X-axis.

void **setGrid**(bool major, bool minor)

Toggle major and minor gridline visibility on both axes.

void **setLegendPos**(LegendPos pos)

Place a legend in one of the plot corners (or turn it off)

The legend lists each visible, named data series (raw / processed / fit / overlays); reference lines and unnamed marker-only mirrors are excluded.

void **setRefLabelStyle**(double pointSize, double distance, bool boxed)

Style applied to all reference-line labels.

The per-line label position along the line comes from each reference series' RefAnchor; this sets the window-wide font/offset/box appearance.

Parameters

- **pointSize** – Label font point size (<= 0 keeps the default size)
- **distance** – Perpendicular gap between the label and its line, in px
- **boxed** – Draw a frame + opaque background behind each label

void **addSeries**(const *PlotSeries* *series)

Register a series for drawing (non-owning; ignored if already present)

Parameters

series – Series owned by the caller

void **removeSeries**(const *PlotSeries* *series)

Remove a previously registered series (does not delete it)

bool **hasSeries**(const *PlotSeries* *series) const

Whether a series is currently registered.

void **clearSeries**()

Unregister all series (does not delete them)

Protected Functions

void **paintEvent**(QPaintEvent *event) override

Paint the chart onto the widget.

QSize **sizeHint**() const override

Recommended size.

Chart Model and Axis Math

Neutral chart model value types (`src/plotseries.h`) consumed by `PlotWidget`, and the Qt-free axis-layout helpers (`src/plotaxismath.h`: nice-number ticks, tick values, and printf-style label formatting).

Enums

enum class **PlotSeriesType**

Whether a *PlotSeries* is drawn as a connected line or as markers.

Values:

enumerator **Line**

enumerator **Scatter**

enum class **RefAnchor**

Where a reference-line label sits along its line.

For a vertical line: Start = top, Center = middle, End = bottom. For a horizontal line: Start = left, Center = center, End = right.

Values:

enumerator **Start**

enumerator **Center**

enumerator **End**

struct **PlotSeries**

#include <plotseries.h> One data series in the neutral chart model.

Carries the points plus the minimal styling the native renderer needs.

Public Functions

inline void **append**(double x, double y)

Append one (x, y) point.

inline void **replace**(const QList<QPointF> &p)

Replace all points (and drop any error bars, which no longer fit)

inline bool **hasErrors**() const

Whether the series carries usable error bars.

inline bool **hasAsymErrors**() const

Whether the bars extend by different amounts up and down.

inline double **errHigh**(int i) const

How far the bar of point `i` reaches above the value.

inline double **errLow**(int i) const

How far the bar of point `i` reaches below the value.

inline int **count**() const

Number of points.

inline QPointF **at**(int i) const

Point at index i.

inline void **setVisible**(bool v)

Set visibility.

inline bool **isVisible**() const

Whether the series is visible.

Public Members

PlotSeriesType **type** = *PlotSeriesType*::*Line*

line vs. scatter rendering

QList<QPointF> **points**

data points in axis coordinates

QList<double> **yerr**

upper y error per point (empty = none)

QList<double> **yerrLo**

lower y error per point (empty = symmetric)

QColor **errColor**

error bar color (invalid = series color)

qreal **errWidth** = 1.5

error bar line width

QColor **color** = Qt::black

line / marker color

qreal **width** = 1.0

line width (Line series)

Qt::PenStyle **style** = Qt::SolidLine

line style, e.g. dashed reference lines

qreal **markerSize** = 8.0

marker diameter (Scatter series)

QString **name**

series label (shown in the legend)

bool **visible** = true

whether the series is drawn

bool **isReference** = false

draw as a labeled reference line

QString **refLabel**

text drawn next to a reference line

RefAnchor **refAnchor** = *RefAnchor*::*Start*

where the label sits along the line

struct **PlotAxis**

#include <plotseries.h> One linear value axis in the neutral chart model.

Public Members

double **min** = 0.0

lower end of the displayed range

double **max** = 1.0

upper end of the displayed range

QString **title**

axis title

QString **labelFormat** = QStringLiteral("%g")

printf-style tick label format

int **subTicks** = 4

minor subdivisions between major ticks

bool **gridVisible** = true

draw major gridlines

bool **minorGridVisible** = true

draw minor gridlines

namespace **PlotAxisMath**

ImageViewer Class

class **ImageViewer** : public QDialog

Dialog for viewing and manipulating LAMMPS snapshot images.

This class provides an image viewer dialog for displaying LAMMPS snapshots created by the `dump image` command. It allows interactive manipulation of visualization parameters such as zoom, rotation, atom size, coloring, and rendering options. Changes can be applied to regenerate the image using the LAMMPS library interface.

dump-image command preparation used by createImage()

Gather widget state and LAMMPS-derived data into a *DumpImageParams* snapshot

dialog row builders/readers used by the *Settings() slots

Build one compute/fix table row per map entry (shared by fixSettings)

Public Functions

explicit **ImageViewer**(const QString &fileName, *LammpsWrapper* *lammps, *LammpsGui* *lammpsgui, QWidget *parent = nullptr)

Constructor.

Parameters

- **fileName** – Path to the image file to display
- **lammps** – Pointer to *LammpsWrapper* for regenerating images
- **lammpsgui** – Pointer to *LammpsGui* for sending signals
- **parent** – Parent widget

~ImageViewer() override

Destructor.

ImageViewer() = delete

ImageViewer(const *ImageViewer*&) = delete

ImageViewer(*ImageViewer*&&) = delete

ImageViewer &**operator=**(const *ImageViewer*&) = delete

ImageViewer &**operator=**(*ImageViewer*&&) = delete

void **createImage**()

Generate image using current settings.

Constructs and executes a LAMMPS dump image command with current visualization parameters and updates the displayed image.

Protected Functions

bool **eventFilter**(QObject *watched, QEvent *event) override

Intercept Alt-keystrokes.

void **showEvent**(QShowEvent *event) override

Redo the initial window fit once shown.

Dump Image Command Builder

The `DumpImageParams` struct and the `buildDumpImageCommand()` free function (`src/dumpimage.h`) form a GUI-free, unit-testable core that assembles the LAMMPS `write_dump ... image ...` command from a snapshot of the viewer state. `ImageViewer` populates the struct (resolving all LAMMPS queries up front) and then calls the pure builder, which returns a `DumpImageCommand` holding the render and `dump_modify` argument strings; `toWriteDumpCommand()` composes the final one-shot `write_dump` command from those pieces.

struct **DumpImageParams**

Pre-resolved inputs for assembling a LAMMPS `dump image` command.

`ImageViewer::gatherDumpImageParams()` populates this struct from the widget state and from a set of LAMMPS library queries (atom/setting/global data) before the image is rendered. All LAMMPS-derived values are captured here as plain data so that buildDumpImageCommand() can run as a pure function – it never touches the GUI or a live LAMMPS instance and is therefore unit-testable on its own.

The `computes/fixes/regions` maps hold non-owning pointers to *ImageInfo* and *RegionInfo* objects owned elsewhere (by *ImageViewer* at runtime, or by the test fixture in unit tests); they only need to outlive the call.

Public Members

QString **group**

atom group to render

QString **dumpfile**

full path of the temporary `.ppm` output file

bool **atomcustom**

use custom atom color/diameter properties

bool **useelements**

element data available (color/diameter by element)

bool **usediameter**

per-atom diameter (radius) attribute available

bool **usesigma**

Lennard-Jones sigma usable as atom radius.

bool **showatoms**

draw atoms

QString **atomcolor**

custom atom color property

QString **atomdiam**

custom atom diameter property (attribute name, a "v_" atom-style variable reference, or a numeric diameter)

double **vdwfactor**

van der Waals radius scaling factor

double **atomSize**

explicit atom size (radius)

QString **elements**

pre-built `element <X> <Y> ...` argument string

QString **adiams**

pre-built `adiam <i> <d> ...` argument string

bool **showbodies**

draw body particles

int **body_flag**

LAMMPS body_flag setting.

QString **bodycolor**

body color property

double **bodydiam**

body diameter

int **bodyflag**

body draw flag (triangle, cylinder, or both)

bool **showlines**

draw line particles

int **line_flag**

LAMMPS line_flag setting.

QString **linecolor**

line color property

double **linediam**

line diameter

bool **showtris**

draw triangle particles

int **tri_flag**

LAMMPS tri_flag setting.

QString **tricolor**

triangle color property

int **triflag**

triangle draw flag (triangle, cylinder, or both)

double **tridiam**

triangle diameter

bool **showellipsoids**

draw ellipsoid particles

int **ellipsoid_flag**

LAMMPS ellipsoid_flag setting.

QString **ellipsoidcolor**

ellipsoid color property

int **ellipsoidflag**

ellipsoid draw flag (triangle or cylinder)

int **ellipsoidlevel**

ellipsoid refinement level

double **ellipsoiddiam**

ellipsoid diameter

int **nbondtypes**

number of bond types

int **bond_flag**

LAMMPS bond_flag setting.

bool **showbonds**

draw bonds

QString **bondcolor**

bond color property (or "c_<id>" when bondbyvalue)

bool **bondbyvalue**

color bonds by a per-bond compute value (emit bmap)

QString **bonddiam**

bond diameter property

bool **autobond**

derive bonds from a distance cutoff

bool **haspairstyle**

a pair style other than "none" is defined

double **bondcutoff**

autobond distance cutoff

int **xsize**

rendered image width in pixels

int **ysize**

rendered image height in pixels

double **zoom**

zoom level

double **shinyfactor**

shininess / specular factor

bool **antialias**

enable full-scene anti-aliasing

int **dimension**

system dimension (2 or 3)

int **hrot**

horizontal rotation angle

int **vrot**

vertical rotation angle

bool **usessao**

enable screen-space ambient occlusion

double **ssaoval**

SSAO strength.

int **ssaosamples**

SSAO sampling directions, 0 = derived from the SSAO strength.

bool **usedepthcue**

enable depth cueing

double **depthcuefactor**

depth cueing strength (0.0 - 1.0)

QString **depthcuecolor**

fog color name, or “auto” = fade toward the background

QString **depthcuestart**

fading start as a box fraction along the view direction, or “auto”

bool **usedefocus**

enable defocusing of distant objects

double **defocusfactor**

defocus blur strength (0.0 - 1.0)

QString **defocusstart**

blurring start as a box fraction along the view direction, or “auto”

bool **useoutline**

draw outlines at depth jumps

int **outlinewidth**

outline width in pixels (1 - 16)

QString **outlinecolor**

outline color name

QString **specular**

specular preset “none”/”wide”/”narrow”/”tight”, or “auto” = highlight width derived from the shiny factor

bool **usemetal**

enable metallic surface shading

double **metalfactor**

how metallic the objects appear (0.0 - 1.0)

QString **metalfinish**

metal surface finish “satin”/”polished”/”mirror”

bool **showbox**

draw simulation box

double **boxdiam**

box edge diameter

bool **showsubbox**

draw subdomain boxes

double **subboxdiam**

subbox edge diameter

bool **showaxes**

draw coordinate axes

QString **axesloc**

axes location

double **axeslen**

axes length

double **axesdiam**

axes diameter

bool **dynamiccenter**

use the dynamic (“d”) instead of the static (“s”) center flavor

double **xcenter**

view center x coordinate

double **ycenter**

view center y coordinate

double **zcenter**

view center z coordinate

double **xup**

camera up vector x component

double **yup**

camera up vector y component

double **zup**

camera up vector z component

int **ntypes**

number of atom types

QList<QPair<QString, QColor>> **color_list**

per-type atom color table

QString **boxcolor**

box / subbox color

QString **backcolor**

lower background color

QString **backcolor2**

upper background color

bool **usegradient**

draw a vertical gradient

double **axestrans**

axes transparency

double **boxtrans**

box / subbox transparency

double **atomtrans**

atom transparency

double **bondtrans**

bond transparency

double **ambientlight**

ambient light setting

double **keylight**

key light setting

double **filllight**

fill light setting

double **backlight**

back light setting

double **gammaval**

gamma adjustment of rendered objects (0.1 - 10.0), 1.0 = unchanged

int **version**

LAMMPS version (date) id.

QString **colormap**

name of the selected atom color map

QString **mapmin**

minimum-value choice for the atom color map

QString **mapmax**

maximum-value choice for the atom color map

bool **revcolormap**

reverse (mirror) the atom color map

QString **bondcolormap**

name of the selected bond color map

QString **bondmapmin**

minimum-value choice for the bond color map

QString **bondmapmax**

maximum-value choice for the bond color map

bool **revbondcolormap**

reverse (mirror) the bond color map

std::map<std::string, *ImageInfo**> **computes**

per-compute graphics settings

std::map<std::string, *ImageInfo**> **fixes**

per-fix graphics settings

std::map<std::string, *RegionInfo**> **regions**

per-region display settings

DumpImageCommand **buildDumpImageCommand**(const *DumpImageParams* &p)

Assemble the render and dump_modify argument strings for a dump image.

Pure function: depends only on `p`, performs no I/O and no LAMMPS calls, and is therefore exercised directly by the unit tests.

Parameters

p – Fully populated parameter struct (no GUI or LAMMPS access required)

Returns

The two argument strings (see *DumpImageCommand*)

---

struct **DumpImageCommand**

The two argument strings of an assembled `dump image` command.

The render options (everything that follows `... image <N> <file>`) and the `dump_modify` options (colors, color maps, lighting, ...) are kept separate so the caller can compose either an explicit `dump`/`dump_modify` pair or a one-shot `write_dump` (via toWriteDumpCommand()), and so each builder step can append to whichever string is logical. Both strings begin with a leading space.

Public Members

QString **dumpargs**

render options after `image <N> <file>`

QString **modifyargs**

options for `dump_modify <id>` / after `modify`

bool **dofixes**

a fix/compute graphic is active (write_dump omits `noinit`)

QString **toWriteDumpCommand**(const *DumpImageCommand* &c, const QString &group, const QString &file)

Compose a one-shot `write_dump ... image ...` command from the pieces.

Parameters

- **c** – the two argument strings from buildDumpImageCommand()
- **group** – atom group to render
- **file** – output image file name

Returns

A complete `write_dump` command string

Color Maps

The dump-image color maps are defined once, as a table of `ColorMapDef` entries in `src/colormaps.cpp`. Both the command builder (`appendColorMapArgs()` in `src/dumpimage.cpp`) and the settings-dialog preview swatches (`addColorMapItems()` in `src/imageviewersettings.cpp`) consume this single source of truth, so they cannot drift apart. See *Adding or modifying a color map* for how to add or modify a map.

Functions

const *ColorMapDef* &**colorMapDef**(const QString &name)

Look up a color-map definition by name.

Parameters

name – color-map name (e.g. "Viridis"); unknown names fall back to "BWR"

Returns

reference to the (statically stored) definition

const QStringList &**colorMapNames**()

The selectable color-map names, in display order.

Returns

reference to the (statically stored) ordered name list

struct **ColorMapStop**

#include <colormaps.h> A single color stop of a dump-image color map.

A stop is either a LAMMPS named color (*name* non-empty, used verbatim in the generated command and resolved with QColor for the preview) or an explicit RGB color (*name* empty, *r* / *g* / *b* used). Storing the RGB as floats keeps the generated command identical to the values LAMMPS renders and lets the preview use the very same numbers via QColor::fromRgbF(). *pos* is the stop position in [0,1] and is only meaningful for continuous maps (the first stop maps to the `min` value, the last to `max`).

Public Members

double **pos**

position in [0,1] (continuous maps); 0 = min, 1 = max

QString **name**

LAMMPS named color, or empty when an explicit RGB is given.

double **r**

explicit red in [0,1], used when *name* is empty

double **g**

explicit green in [0,1], used when *name* is empty

double **b**

explicit blue in [0,1], used when *name* is empty

struct **ColorMapDef**

#include <colormaps.h> Definition of a named dump-image color map.

This is the single source of truth shared by the command builder (`appendColorMapArgs()` in dumpimage.cpp, which emits `dump_modify amap/bmap`) and the settings-dialog preview swatches (`addColorMapItems()` in imageviewersettings.cpp), so the swatch a user picks matches what LAMMPS renders.

Public Members

bool **continuous**

true: interpolated (`cf`); false: discrete sequence (`sa`)

QList<*ColorMapStop*> **stops**

ordered color stops

SlideShow Class

class **SlideShow** : public QDialog

Slideshow viewer for displaying sequences of images.

SlideShow provides a dialog for viewing and navigating through sequences of images, typically from LAMMPS dump image commands. It supports manual navigation (first/prev/next/last), automatic playback with configurable timing, looping, and zoom controls. Images can be exported as a movie file.

Public Functions

explicit **SlideShow**(const QString &fileName, *LammpsGui* *lammpsgui = nullptr, QWidget *parent = nullptr)

Constructor.

Parameters

- **fileName** – Path to first image file

- **lammpsgui** – Pointer to *LammpsGui* for sending signals (optional; nullptr for a standalone viewer with no live simulation)
- **parent** – Parent widget

~SlideShow() override = default

Destructor.

SlideShow() = delete

SlideShow(const *SlideShow*&) = delete

SlideShow(*SlideShow*&&) = delete

SlideShow &**operator=**(const *SlideShow*&) = delete

SlideShow &**operator=**(*SlideShow*&&) = delete

void **addImage**(const QString &filename, const QString &label = QString())

Add an image to the slideshow sequence.

Parameters

- **filename** – Path to image file to add
- **label** – Text shown in place of the file name (optional)

int **addMovie**(const QString &filename)

Extract the frames of a movie file and add them as images.

Probes the movie, asks the user to confirm the extraction and to select a frame range and interval, and decodes the selected frames into PNG files inside the image cache, where they are removed together with the rest of the cache when the slide show window is closed.

Parameters
filename – Path to the movie file

Returns
Number of images added; 0 when canceled or on failure

inline int **imageCount**() const

Number of images currently in the slideshow sequence.

void **clear**()

Clear all images from slideshow.

Protected Functions

void **showEvent**(QShowEvent *event) override

Redo the initial window fit once shown.

ImageCache Class

`SlideShow` owns an `ImageCache` (`src/imagecache.h`) that holds the temporary files it creates: the PNG copies of image formats that Qt cannot decode natively, and the frames extracted from imported movie files. A source file is converted at most once, since the cache entries are validated against the modification time and the size of the source file, and the whole cache directory is removed when the slide show window is closed.

class **ImageCache**

Cache of images converted to a format that Qt can decode.

Qt reads only a subset of the image formats that LAMMPS and other tools can write. For the remaining ones (tga, eps, sgi, ...) *readImage()* shells out to ImageMagick and converts the file to a temporary PNG. That conversion is expensive, so the PNG is kept and reused: a file is converted at most once, unless it changes on disk. Cache entries are keyed by the absolute source path and validated against its modification time and size, so a file that is rewritten (a growing dump image sequence, for example) is converted again.

An entry therefore exists only for a file that Qt cannot decode, and a fresh entry answers a request without handing the source file to Qt again. That matters beyond the saved work: several of Qt's image format plugins print a warning for every file they reject (the TGA plugin is a notorious example), which would otherwise be repeated on every single display of the image. The one unavoidable attempt is made under a *QtMessageSilencer*, and its output is reported only if ImageMagick cannot rescue the file either. A file that cannot be read at all is remembered as such, so it is neither converted nor complained about twice.

All temporary files live in a single QTemporaryDir that is created on first use and removed, with everything in it, when the cache is destroyed. The directory also hosts the frames extracted from imported movie files, for which *makeSubDir()* hands out private subdirectories.

The class is not thread-safe and must only be used from the GUI thread.

Public Functions

ImageCache()

Constructor. The temporary directory is created lazily.

~ImageCache()

Destructor. Removes the temporary directory and its contents.

ImageCache(const *ImageCache*&) = delete

ImageCache(*ImageCache*&&) = delete

ImageCache &**operator=**(const *ImageCache*&) = delete

ImageCache &**operator=**(*ImageCache*&&) = delete

QImage **readImage**(const QString &filename)

Read an image file, converting it first if Qt cannot decode it.

Files that Qt understands are decoded directly and never cached. For the others ImageMagick (`magick` or `convert`) is used, if available, and the resulting PNG is cached for subsequent calls. A file that neither Qt nor ImageMagick can read is reported once, on standard error, and then remembered as unreadable.

Parameters
filename – Path to the image file

Returns
The decoded image, or a null QImage if it cannot be read

QSize **imageSize**(const QString &filename)

Dimensions of an image file, without decoding its pixels.

Reads the header only, so it is much cheaper than *readImage()* when just the dimensions are needed. For a file with a cached conversion the header of the converted PNG is read instead of the original, so a format that Qt cannot decode is not handed to it a second time.

Parameters
filename – Path to the image file

Returns
Image size, or an invalid QSize if it cannot be determined

QString **makeSubDir**(const QString &prefix)

Create a private subdirectory inside the cache directory.

Each call creates a new directory, so two imports of the same movie do not overwrite each other's frames.

Parameters
prefix – Name hint; characters outside [A-Za-z0-9_-] are replaced

Returns
Absolute path of the new directory, or an empty string on failure

void **registerFrames**(const QString &subdir)

Account for the files a caller has written into a subdirectory.

The cache does not create movie frames itself, so it has to be told about them before it can report them in *frameImages()* and *frameBytes()*.

Parameters
subdir – Directory from *makeSubDir()* that now holds extracted frames

void **forget**(const QString &filename)

Drop what the cache knows about a single source file.

Deletes its converted copy, if any. Call this when the source file is removed from disk, since its conversion is then useless and would occupy the cache directory until the cache is destroyed.

Parameters
filename – Path to the source file

void **purgeConversions**()

Delete every converted image, keeping the extracted movie frames.

A conversion can be redone from its source file on demand, so discarding it only costs time. Movie frames cannot be re-created without running FFmpeg over the movie again and are therefore left alone, as are the records of files that could not be read at all: forgetting those would only make the cache report them a second time.

QString **path**()

Path of the temporary cache directory, creating it if needed.

Returns
Absolute path, or an empty string if the directory cannot be made

inline int **count**() const

Number of source files Qt cannot decode that the cache knows about.

Counts both the files with a converted copy and those remembered as unreadable. Files that Qt decodes directly are never given an entry.

inline int **conversions**() const

Number of times ImageMagick has been run to convert a file.

A cache that works keeps this at one run per file, however often the file is displayed.

inline int **cachedImages**() const

Number of converted images currently held in the cache directory.

inline qint64 **cachedBytes**() const

Total size in bytes of the converted images.

inline int **frameImages**() const

Number of extracted movie frames registered with *registerFrames()*

inline qint64 **frameBytes**() const

Total size in bytes of the extracted movie frames.

inline bool **isEmpty**() const

Whether the cache directory holds any images at all.

void **clear**()

Drop all cached conversions and delete the temporary directory.

Movie Frame Import

Movie files are turned into a sequence of images before they can be shown in the slide show viewer. `probeMovie()` collects the properties of the video stream by running `ffprobe`, `MovieImportDialog` lets the user confirm the extraction and select a frame range and interval, and `extractMovieFrames()` runs `ffmpeg` to decode the selected frames into the `ImageCache`. The parsing and frame-counting helpers (`src/movieimport.h`) are free functions so that they can be unit tested without running either program.

struct **MovieInfo**

Properties of the first video stream of a movie file.

Filled in by probeMovie(). When `valid` is false, `error` holds a message suitable for display in a dialog and all other members are meaningless.

Public Members

bool **valid** = false

True when the movie was probed successfully.

int **width** = 0

Frame width in pixels.

int **height** = 0

Frame height in pixels.

int **frames** = 0

Number of frames in the video stream.

double **fps** = 0.0

Nominal frame rate in frames per second.

double **duration** = 0.0

Duration of the movie in seconds.

QString **error**

Reason why the movie could not be probed.

---

class **MovieImportDialog** : public QDialog

Dialog to confirm and configure the import of movie frames as images.

Shows the properties of the movie, lets the user pick a frame range and a frame interval, and estimates how much temporary disk space the extracted images will need. The estimate is the size of a single decoded sample frame times the number of selected frames. When it exceeds Cfg::MOVIE_WARN_BYTES, Cfg::MOVIE_WARN_FRAMES frames, or most of the free space on the volume holding the temporary directory, a highlighted warning is displayed.

The sample frame is shown as a thumbnail next to the movie properties. When the middle of the selected range moves away from the sampled frame (see sampleOutdated()), a frame near the new middle is decoded after a short delay, and the thumbnail and the size estimate are refreshed from it.

Public Functions

MovieImportDialog(const QString &filename, const *MovieInfo* &info, QWidget *parent = nullptr)

Constructor.

Parameters

- **filename** – Path to the movie file, used for the labels only
- **info** – Movie properties from probeMovie(); must be valid
- **parent** – Parent widget

~MovieImportDialog() override = default

Destructor.

MovieImportDialog() = delete

MovieImportDialog(const *MovieImportDialog*&) = delete

MovieImportDialog(*MovieImportDialog*&&) = delete

MovieImportDialog &**operator=**(const *MovieImportDialog*&) = delete

MovieImportDialog &**operator=**(*MovieImportDialog*&&) = delete

int **firstFrame**() const

First selected frame, counted from 1.

int **lastFrame**() const

Last selected frame (inclusive), counted from 1.

int **frameInterval**() const

Selected frame interval; 1 selects every frame.

MovieInfo **probeMovie**(const QString &filename)

Determine the properties of a movie file by running ffprobe.

Parameters

filename – Path to the movie file

Returns

Movie properties, with *MovieInfo::valid* indicating success

MovieInfo **parseProbeOutput**(const QByteArray &json)

Extract the movie properties from the JSON output of ffprobe.

Separated from probeMovie() so that the parsing can be tested without running ffprobe.

Parameters

json – Output of "ffprobe -of json" for the first video stream

Returns

Movie properties; *MovieInfo::frames* is 0 when the frame count is not stored in the container and must be determined separately

double **parseFrameRate**(const QString &rate)

Convert an FFmpeg frame rate to a number.

Parameters

rate – Rate as a rational ("30000/1001") or plain number ("25")

Returns

Frames per second, or 0.0 if `rate` cannot be parsed

int **selectedFrameCount**(int first, int last, int interval)

Number of frames selected by a range and a stride.

The frame numbers may be counted from either 0 or 1, the result is the same.

Parameters

- **first** – First frame of the range
- **last** – Last frame of the range (inclusive)
- **interval** – Stride; 1 selects every frame, 2 every other one, ...

Returns

Number of selected frames, or 0 if the arguments are inconsistent

QStringList **extractMovieFrames**(QWidget *parent, const QString &filename, const QString &outdir, int first, int last, int interval, QString &error)

Extract selected frames of a movie into individual PNG files.

Runs FFmpeg with a progress dialog that lets the user abort a long extraction. On abort or failure the partial output is removed and an empty list is returned.

Parameters

- **parent** – Parent widget of the progress dialog
- **filename** – Path to the movie file
- **outdir** – Existing directory that receives the PNG files
- **first** – First frame to extract, counted from 1
- **last** – Last frame to extract (inclusive), counted from 1
- **interval** – Stride; 1 extracts every frame, 2 every other one, ...
- **error** – Set to a message when the result is empty

Returns

Paths of the extracted PNG files in movie order, empty on failure

---

Dialog Components

FindAndReplace Class

class **FindAndReplace** : public QDialog

Find and Replace dialog for the code editor.

FindAndReplace provides a dialog for searching and replacing text in the *CodeEditor*. It supports case-sensitive/insensitive search, whole word matching, text wrapping, and batch replace operations. The dialog is non-modal so users can continue editing while searching.

Public Functions

explicit **FindAndReplace**(*CodeEditor* *_editor, QWidget *parent = nullptr)

Constructor.

Parameters

- **_editor** – Pointer to the *CodeEditor* to search in
- **parent** – Parent widget

~FindAndReplace() override = default

Destructor.

FindAndReplace() = delete

FindAndReplace(const *FindAndReplace*&) = delete

FindAndReplace(*FindAndReplace*&&) = delete

FindAndReplace &**operator=**(const *FindAndReplace*&) = delete

FindAndReplace &**operator=**(*FindAndReplace*&&) = delete

SetVariables Class

class **SetVariables** : public QDialog

Dialog for editing LAMMPS index-style variable definitions.

SetVariables provides a dialog for managing name-value pairs that will be used as index-style variables in LAMMPS input scripts. Users can add, delete, and edit variable definitions. The dialog shows the variables that the input script defines or uses; values that override a differing definition in the input script are shown in bold with a tooltip listing the script value.

Public Functions

explicit **SetVariables**(QList<*VariableEntry*> &vars, QWidget *parent = nullptr)

Constructor.

Parameters

- **vars** – Reference to list of variable entries (modified in place)
- **parent** – Parent widget

~SetVariables() override = default

Destructor.

SetVariables() = delete

SetVariables(const *SetVariables*&) = delete

SetVariables(*SetVariables*&&) = delete

SetVariables &**operator=**(const *SetVariables*&) = delete

SetVariables &**operator=**(*SetVariables*&&) = delete

ShellAliases Class

class **ShellAliases** : public QDialog

Editor for the aliases the command window defines in its shell.

The shell the command window starts reads the user's start-up file, but a section of that file guarded by a test for a terminal stops before it runs, because the shell is reading a pipe. On several distributions that is where `ls`, `ll` and `l`. are defined, so those go missing while every other alias is present. Programs behave differently for the same reason: `ls` lists one entry per line rather than in columns unless it is told otherwise.

Rather than give the shell a terminal, this dialog lets those few aliases be stated once and defines them in every shell the command window starts. The defaults cover the common case – `ls` and `ll` with the arguments that restore the output a terminal would have produced.

Only what is in the table is sent, and only when a shell starts or the table is accepted; nothing else ever reaches the shell without being typed.

See also

CommandWindow

Public Functions

explicit **ShellAliases**(QWidget *parent = nullptr)

Constructor.

Parameters
parent – Parent widget

~ShellAliases() override = default

ShellAliases() = delete

ShellAliases(const *ShellAliases*&) = delete

ShellAliases(*ShellAliases*&&) = delete

ShellAliases &**operator=**(const *ShellAliases*&) = delete

ShellAliases &**operator=**(*ShellAliases*&&) = delete

Public Slots

void **accept**() override

Store the table and close.

Public Static Functions

static QList<ShellAlias> **aliases**()

The aliases to define, in the order they are listed.

The defaults are returned until the dialog has been accepted once, so a new installation starts with `ls` and `ll` already useful.

Returns
Name and definition of each alias

static QList<ShellAlias> **defaults**()

The aliases a new installation starts with.

Returns
Name and definition of each alias

ShellPrompt Class

class **ShellPrompt** : public QLineEdit

Input line that completes with Tab, the way a shell prompt does.

A plain QLineEdit gives Tab away to the focus chain and completes with nothing, which is the wrong half of both. This takes the key back: Tab offers what matches the word being typed, Tab again walks the offers, and a word with only one match is completed without a list appearing at all.

Both interceptions have to happen in *event()* rather than in keyPressEvent() or an event filter, for two separate reasons:

- `QWidget::event()` hands the focus on when it sees Tab, before `keyPressEvent()` is ever reached.
- While the completion popup is up, QCompleter delivers keys to this widget by calling *`event()`* on it directly instead of sending them through the event loop, so an event filter installed on the line edit never sees them.

The same override is what keeps Enter from doing two things at once; see *event()* for what Qt does with it when left alone.

The widget completes from whatever QCompleter it was given with setCompleter(). What it does not know is which list the word being typed should be completed from – in a shell that depends on where in the line the word is – so it emits *completing()* first and leaves that choice to its owner.

Public Functions

inline explicit **ShellPrompt**(QWidget *parent = nullptr)

Constructor.

Parameters
parent – Parent widget (optional)

~ShellPrompt() override = default

Destructor.

ShellPrompt(const *ShellPrompt*&) = delete

ShellPrompt(*ShellPrompt*&&) = delete

ShellPrompt &**operator=**(const *ShellPrompt*&) = delete

ShellPrompt &**operator=**(*ShellPrompt*&&) = delete

Signals

void **completing**(const QString &text)

Emitted before completions are computed, so the completer can be pointed at the list this word should be completed from.

Parameters
text – The line as it currently stands

Protected Functions

bool **event**(QEvent *event) override

Filter out the keys a completing prompt has to own.

Parameters
event – Event to handle

Returns
true if the event was consumed

Index Variable Helpers

The parse, merge, and override-detection helpers behind the *Set Variables* dialog and the editor's override markers are free functions over plain value types (`src/inputvariables.h`).

struct **VariableEntry**

One index-style variable managed via the Set Variables dialog.

The effective value is what LAMMPS-GUI defines before a run (like the -var command line flag would); the script value records what the input script currently assigns so that edits to the input can be told apart from overrides entered in the Set Variables dialog.

Public Members

QString **name**

variable name

QString **value**

effective value defined before a run; empty if unset

QString **scriptValue**

value assigned in the input script; empty if not defined there

struct **IndexVariableMatch**

Result of matching a line against an index variable definition.

The value offsets refer to the unmodified line text so they can be used to locate the value in a text editor for visual markup.

Public Members

bool **valid** = false

true if the line defines an index style variable

QString **name**

variable name

QString **value**

assigned value with surrounding whitespace removed

int **valueStart** = 0

offset of the value in the line

int **valueLength** = 0

length of the (trimmed) value in the line

Functions

IndexVariableMatch **matchIndexVariable**(const QString &line)

Match a single input script line against an index variable definition.

Parameters
line – One line of input script text

Returns
Match result with the name, value, and value position on success

QList<*VariableEntry*> **parseInputVariables**(const QString &text)

Collect index-style variables from an input script.

Records every index variable definition (first definition wins, as in LAMMPS) with its assigned value and every use of an otherwise undefined variable with an empty value.

Parameters
text – Complete input script text

Returns
List of variable entries in order of appearance

QList<*VariableEntry*> **mergeInputVariables**(const QList<*VariableEntry*> &parsed, const QList<*VariableEntry*> &previous)

Fold a fresh parse of the input script into an existing variable list.

A changed, non-empty script value is the most recent user edit and replaces the current value, while an unchanged one preserves the value (which may be an override from the Set Variables dialog). Entries no longer present in the script are kept as long as they have a value (they may apply to included files) and dropped otherwise.

Parameters

- **parsed** – Entries from parsing the current input script
- **previous** – Current variable list (parse results plus dialog edits)

Returns
Merged list: script order first, then retained leftover entries

bool **isOverridden**(const *VariableEntry* &entry)

Check whether an entry overrides the definition in the input script.

Parameters
entry – Variable entry to check

Returns
true if the entry's value replaces a different script definition

---

PlotDataDialog Class

class **PlotDataDialog** : public QDialog

Dialog to choose which columns of a *PlotData* to plot.

Presents the parsed columns of an external data file and lets the user assign a role to each column: exactly one column is the shared x-axis (exclusive radio buttons), any number of columns are plotted on the y-axis (checkboxes), and columns with neither selected are ignored. When the x-axis selection moves to a different column, the previous x-axis column becomes a y-axis column. A small preview of the first rows is shown to help identify column content.

A "Compute derived column" section at the bottom lets the user add derived columns from expressions that reference the existing columns by name in braces, like Python format strings (e.g. `{nfcc}/{ntot}` or `{load}*1.602176634`). `{name:first}`, `{name:last}`, `{name:min}`, `{name:max}`, and `{name:mean}` are per-column constants, and `row` is the 0-based row index. Each column has exactly one live name: renaming it takes effect immediately, updates the preview, and rewrites the references in already added derived columns. See substituteColumnRefs() in customfunc.h.

For a block-structured `fix ave/\*` file (see plotblockdata.h) the dialog grows a "Data blocks" group above the column grid, which reduces the blocks to the single flat table the columns below then refer to: either one chosen block, or the average of a range of them with error bars. Changing the reduction rebuilds the column grid, keeping the roles and names the user has already assigned and re-evaluating any derived columns against the new table.

Public Functions

explicit **PlotDataDialog**(const *PlotData* &data, QWidget *parent = nullptr)

Constructor for a flat data table.

Parameters

- **data** – Parsed column data to choose from (stored as a working copy)
- **parent** – Parent widget

explicit **PlotDataDialog**(const *PlotBlockData* &blocks, QWidget *parent = nullptr)

Constructor for a block-structured fix ave/* file.

Starts out on the reduction that suits the detected format, which the "Data blocks" group then lets the user change.

Parameters

- **blocks** – Parsed blocks (stored as a working copy)
- **parent** – Parent widget

~PlotDataDialog() override = default

PlotDataDialog() = delete

PlotDataDialog(const *PlotDataDialog*&) = delete

PlotDataDialog(*PlotDataDialog*&&) = delete

PlotDataDialog &**operator=**(const *PlotDataDialog*&) = delete

PlotDataDialog &**operator=**(*PlotDataDialog*&&) = delete

int **xColumn**() const

Index of the column used as the x-axis.

Returns the index of the column whose x-axis radio button is selected. Falls back to 0 if no column is selected as the x-axis. Indices refer to *buildData()* columns.

Returns
Column index

QList<int> **yColumns**() const

Indices of the columns selected to plot on the y-axis.

Indices refer to *buildData()* columns.

Returns
List of column indices (all columns with a checked y checkbox)

PlotData **buildData**() const

Return the working data with renames and derived columns applied.

Includes any columns added via the "Compute derived column" section; renames are already committed when they are made, so this is the working copy as it stands.

Returns
Updated *PlotData* ready for plotting

PlotErrors **buildErrors**() const

Error bars belonging to the columns of *buildData()*

Only a block average produces them; everything else returns entries that are all empty. Indexed like the columns of *buildData()*.

Returns
Per-column error bars

Public Static Functions

static std::unique_ptr<*PlotDataDialog*> **fromFile**(const QString &fileName, QWidget *parent, QString *error)

Read a data file and build the column dialog for it.

The block-structured output of the fix ave/time family is tried first, since it is not a flat table and gets the dialog that can reduce it to one; anything else goes through the flat-file parsers.

Parameters

- **fileName** – File to read
- **parent** – Parent widget of the dialog
- **error** – Receives what the parsers reported, or the file name when they reported nothing; untouched on success

Returns
The dialog, or nullptr when the file could not be read

Preferences Class

class **Preferences** : public QDialog

Preferences/Settings dialog for LAMMPS-GUI.

This dialog provides a tabbed interface for configuring various aspects of LAMMPS-GUI including:

- General settings (LAMMPS library path, plugins, etc.)
- Accelerator package settings
- Image viewer defaults
- Editor appearance and behavior
- Chart viewer settings

Settings are persisted using QSettings and loaded on startup.

Public Functions

explicit **Preferences**(*LammpsWrapper* *lammps, *LammpsGui* *lammpsgui, QWidget *parent = nullptr)

Constructor.

Parameters

- **lammps** – Pointer to *LammpsWrapper* for querying LAMMPS configuration
- **lammpsgui** – Pointer to *LammpsGui* for sending signals
- **parent** – Parent widget

~Preferences() override

Destructor.

Preferences() = delete

Preferences(const *Preferences*&) = delete

Preferences(*Preferences*&&) = delete

Preferences &**operator=**(const *Preferences*&) = delete

Preferences &**operator=**(*Preferences*&&) = delete

inline void **setRelaunch**(bool val)

Set flag indicating application needs restart.

Some settings require restarting the application to take effect.

Parameters

val – true if restart needed, false otherwise

inline void **requestRelaunch**(const QString &reason)

Request a restart and record why.

The reasons are collected and shown together in the dialog that announces the relaunch, so a single visit to the preferences that changes several restart-only settings explains all of them.

Parameters

reason – One sentence naming the setting that changed

GeneralTab Class

class **GeneralTab** : public QWidget

Preferences Tab for General LAMMPS-GUI Settings.

Public Functions

explicit **GeneralTab**(QSettings *settings, *LammpsWrapper* *lammps, *LammpsGui* *lammpsgui, QWidget *parent = nullptr)

Constructor.

Parameters

- **settings** – Pointer to QSettings for storing preferences
- **lammps** – Pointer to *LammpsWrapper* for querying LAMMPS configuration
- **lammpsgui** – Pointer to *LammpsGui* for sending signals
- **parent** – Parent widget

AcceleratorTab Class

class **AcceleratorTab** : public QWidget

Preferences Tab for LAMMPS Accelerator settings.

Public Types

enum **AccelType**

Constants for selecting LAMMPS accelerator package

Values:

enumerator **None**

no accelerator

enumerator **Opt**

OPT package.

enumerator **OpenMP**

OPENMP package.

enumerator **Intel**

INTEL package.

enumerator **Kokkos**

KOKKOS package.

enumerator **Gpu**

GPU package.

enum **AccelPrec**

Constants for selecting LAMMPS accelerator precision

Values:

enumerator **Double**

full double precision

enumerator **Mixed**

only accumulators in double precision, rest in single precision

enumerator **Single**

full single precision

Public Functions

explicit **AcceleratorTab**(QSettings *settings, *LammpsWrapper* *lammps, QWidget *parent = nullptr)

Constructor.

Parameters

- **settings** – Pointer to QSettings for storing preferences
- **lammps** – Pointer to *LammpsWrapper* for querying available accelerator packages
- **parent** – Parent widget

SnapshotTab Class

class **SnapshotTab** : public QWidget

Preferences Tab for Snapshot Viewer Settings.

Public Functions

explicit **SnapshotTab**(QSettings *settings, QWidget *parent = nullptr)

Constructor.

Parameters

- **settings** – Pointer to QSettings for storing preferences
- **parent** – Parent widget

EditorTab Class

class **EditorTab** : public QWidget

Preferences Tab for LAMMPS-GUI Editor Settings.

Public Functions

explicit **EditorTab**(QSettings *settings, QWidget *parent = nullptr)

Constructor.

Parameters

- **settings** – Pointer to QSettings for storing preferences
- **parent** – Parent widget

ChartsTab Class

class **ChartsTab** : public QWidget

Preferences Tab for LAMMPS-GUI Charts Viewer Settings.

Public Functions

explicit **ChartsTab**(QSettings *settings, QWidget *parent = nullptr)

Constructor.

Parameters

- **settings** – Pointer to QSettings for storing preferences
- **parent** – Parent widget

AboutDialog Class

class **AboutDialog** : public QDialog

Custom About dialog for LAMMPS-GUI.

AboutDialog displays version information, LAMMPS configuration details, and available styles in scrollable text areas. The dialog automatically scrolls down when the content exceeds the visible area, pauses at the bottom, and then returns back to the top. When style information is available, the dialog allocates 2/3 of the combined scroll area space to the configuration information text and 1/3 to the style information text.

The style information text uses the configured fixed-width font from the QSettings keys "monofamily" and "monosize" while the rest uses the application's default (variable width) font.

Public Functions

AboutDialog(const QString &version, const QString &info, const QString &details, int minwidth, QWidget *parent = nullptr)

Constructor.

Parameters

- **version** – Version information text displayed at the top
- **info** – LAMMPS configuration info displayed in a scroll area
- **details** – Style information displayed in a scroll area with fixed-width font
- **minwidth** – minimum width of dialog
- **parent** – Parent widget

~AboutDialog() override = default

AboutDialog() = delete

AboutDialog(const *AboutDialog*&) = delete

AboutDialog(*AboutDialog*&&) = delete

AboutDialog &**operator=**(const *AboutDialog*&) = delete

AboutDialog &**operator=**(*AboutDialog*&&) = delete

Protected Functions

void **showEvent**(QShowEvent *event) override

Event handler for widget show events; implements the auto-scroll functionality.

Parameters

event – The show event

Utility Components

CommandWindow Class

class **CommandWindow** : public QWidget

A shell prompt with a scrollback, next to the simulation.

CommandWindow forwards typed lines to one long-lived shell process and streams what it writes back into a read-only scrollback. It exists for the ordinary work that surrounds a run – post-process a dump file with a Python script, look at what a run just wrote, call a plotting tool – without leaving the GUI.

It is deliberately **not a terminal emulator**: there is no pseudo terminal, so no escape sequence interpretation, no cursor addressing, no color, and no curses program. `TERM` is set to `dumb` precisely so that a program needing more than a stream of bytes finds out and says so, rather than writing escape sequences into a scrollback that cannot interpret them.

The shell is kept alive between commands, which is what makes `cd`, `pushd` / `popd`, environment variables and the rest of the shell state work at all. This class does not implement any of them; it only observes the result, by appending a sentinel to every command that reports the exit status and the shell's working directory.

Public Functions

explicit **CommandWindow**(*LammpsGui* *lammpsgui, QWidget *parent = nullptr)

Constructor.

Parameters

- **lammpsgui** – Pointer to the main window, which provides quit() and the shared menus (may be nullptr in standalone use)
- **parent** – Parent widget

~CommandWindow() override

Destructor; ends the shell process.

CommandWindow() = delete

CommandWindow(const *CommandWindow*&) = delete

CommandWindow(*CommandWindow*&&) = delete

CommandWindow &**operator=**(const *CommandWindow*&) = delete

CommandWindow &**operator=**(*CommandWindow*&&) = delete

void **changeDirectory**(const QString &dir)

Move the shell to a directory.

Sent as a command, so the shell stays the one place that knows where it is and the prompt is updated from its answer like any other change. Held back until the shell is at a prompt when a command is running.

Parameters
dir – Directory to change to

Public Static Functions

static QString **preferredShell**()

The command interpreter that will be run.

The shell selected in the preferences, if one was. Otherwise `$SHELL` on Unix-like systems, falling back to `/bin/bash` and then `/bin/sh`; `COMSPEC%` on Windows, falling back to `cmd.exe`.

Returns
Path of the user's preferred shell

static QStringList **availableShells**()

The command interpreters installed on this machine.

On Unix-like systems the entries of `/etc/shells` that exist and are not a `nologin`, plus `$SHELL`. On Windows `cmd.exe`, PowerShell and any `bash.exe` found on the search path or in a Git for Windows installation. Each shell *name* appears once: a shell listed under several paths (e.g. `/bin` and `/usr/bin`) is offered only under the first of them, with the spelling of `$SHELL` taking precedence.

Returns
Sorted list of shells the preferences can offer

Protected Functions

bool **eventFilter**(QObject *watched, QEvent *event) override

Filter the prompt's key presses for the history keys.

Parameters

- **watched** – Object being watched
- **event** – Event to filter

Returns
true if the event was consumed

WindowLayout Class

enum class **ViewSlot**

The output views *LammpsGui* presents alongside the editor.

One enumerator per view that *LammpsGui* keeps for the lifetime of a session (as opposed to the transient file viewers and inspection windows, which are created and closed on demand). Used to address a view in *WindowLayout* without the layout having to know the concrete widget classes.

Values:

enumerator **Log**

Output window with the captured LAMMPS log.

enumerator **Chart**

Charts window with the thermo data of the current run.

enumerator **Image**

Snapshot image viewer.

enumerator **SlideShow**

Slide show viewer for dump image sequences.

enumerator **Variables**

Variables window listing the active index variables.

enumerator **Command**

Shell prompt with a scrollback.

enumerator **Count**

Number of slots; not a view itself.

enum class **LayoutMode**

How the output views are presented.

Values:

enumerator **Windows**

Each view is an individual, freely placed top-level window.

enumerator **Docked**

The views are docked into the main window around the editor.

class **WindowLayout** : public QObject

Presentation policy for the output views of the main window.

WindowLayout is the single place that decides *how* an output view is put in front of the user. *LammpsGui* creates and owns the view widgets and keeps its typed pointers to them, but does not call *show()*, *hide()* or *isVisible()* on them directly: it hands each widget to the layout with *place()* and then addresses it by its ViewSlot.

Two policies are available, chosen once at construction from the user preference:

- LayoutMode::Windows keeps every view an individual top-level window, freely placed and stacked, which is what the application has always done.
- LayoutMode::Docked puts the views into dock areas around the editor, which stays the central widget: the charts, image and slide show views share a tabbed group on the right, the log, the variables view and the command window share a group across the full width at the bottom.

In docked mode the layout owns one QDockWidget per slot, created up front so that a saved arrangement can be restored before the views themselves exist. *place()* only swaps the content of the dock, so a view that is destroyed and rebuilt (as the image viewer is on every render) keeps its position and its place in the tab order.

The layout never owns the view widgets themselves. It watches them for destruction, so a slot whose widget is deleted elsewhere empties itself and never hands out a dangling pointer.

See also

LammpsGui for the owner of both the layout and the *view* widgets

Public Functions

WindowLayout(QMainWindow *mainwindow, *LayoutMode* mode)

Constructor.

In docked mode the dock widgets are created and a previously saved arrangement is restored here, before any view exists.

Parameters

- **mainwindow** – Main window the views belong to; also becomes the parent object, so the layout is deleted along with it
- **mode** – Presentation policy to apply

~WindowLayout() override

Destructor.

WindowLayout() = delete

WindowLayout(const *WindowLayout*&) = delete

WindowLayout(*WindowLayout*&&) = delete

WindowLayout &**operator=**(const *WindowLayout*&) = delete

WindowLayout &**operator=**(*WindowLayout*&&) = delete

inline *LayoutMode* **mode**() const

The policy this layout applies.

Returns

The mode passed to the constructor

void **place**(*ViewSlot* slot, QWidget *view)

Put a view widget into a slot.

Replaces whatever the slot held before without deleting it – the widgets stay owned by *LammpsGui*. Safe to call again with the same widget, which is what a reused Output or Charts window does.

Parameters

- **slot** – Slot the widget belongs to
- **view** – Widget to present; may be nullptr to empty the slot

QWidget ***view**(*ViewSlot* slot) const

Widget currently in a slot.

Parameters
slot – Slot to query

Returns
The widget, or nullptr if the slot is empty

void **show**(*ViewSlot* slot)

Show the view in a slot.

Does nothing when the slot is empty.

Parameters
slot – Slot to show

void **hide**(*ViewSlot* slot)

Hide the view in a slot.

Does nothing when the slot is empty.

Parameters
slot – Slot to hide

void **raise**(*ViewSlot* slot)

Show the view in a slot and bring it to the front.

Use for an explicit request from the user. Unlike *show()*, this pulls the view to the front of its tab group, which is not wanted for the periodic updates during a run.

Parameters
slot – Slot to raise

void **setVisible**(*ViewSlot* slot, bool visible)

Show or hide the view in a slot.

Parameters

- **slot** – Slot to update
- **visible** – true to show the view, false to hide it

bool **toggle**(*ViewSlot* slot)

Flip the visibility of the view in a slot.

Docked, a view that is on screen without holding the keyboard focus is raised and focused rather than hidden, so the key that opened a panel is also the key that goes back to it; a second press, with the focus in it by then, hides it as before.

Persists the new state for the slots that have a “show by default” preference (Output and Charts), so the next session starts the way the session ended.

Parameters
slot – Slot to toggle

Returns
Visibility of the view after the call (false for an empty slot)

void **focusNextPane**(bool forward)

Move the keyboard focus to the neighboring pane.

The panes are the editor and the panels that are on screen, walked in the order they are arranged around it and wrapping at either end. A panel behind a tab is not a pane of its own: it is reached with the key that opens it, which raises it within its group. Does nothing with individual windows, where the window manager has a key for this.

Parameters
forward – true for the next pane, false for the previous one

bool **isVisible**(*ViewSlot* slot) const

Check whether the view in a slot is visible.

Parameters
slot – Slot to query

Returns
true if the slot holds a widget and that widget is visible

void **addAuxiliaryView**(QWidget *view, *ViewSlot* group, const QString &title)

Show a transient view as a tab beside an existing panel.

With individual windows this just shows the widget. Docked, it gets a dock of its own tabbed into that group, which is removed again when the widget is destroyed.

Parameters

- **view** – Widget to show; it keeps its own lifetime
- **group** – Slot whose dock the new tab joins
- **title** – Short label for the tab – the window title of a viewer is far too long to sit in one

void **saveState**() const

Store the current dock arrangement in the settings.

Does nothing in windowed mode, where the views carry their own geometry. Call before the main window is destroyed.

Signals

void **viewActivated**(QWidget *view)

A view was brought to the front of its group.

The combined layout shows one menu bar for the whole window, so whoever owns it needs to know which panel the user just asked to see.

Parameters
view – The view now in front, or nullptr if the slot was empty

Protected Functions

bool **eventFilter**(QObject *watched, QEvent *event) override

Track the docked views and their containers.

Parameters

- **watched** – Object being watched: the main window (resizes keep the dock proportions), a dock (a dragged splitter updates them), or a view (its close is redirected to its dock)
- **event** – Event to inspect

Returns

true if the event was consumed

URLDownloader Class

class **URLDownloader**

Utility class for downloading files over HTTPS.

URLDownloader provides a synchronous interface for downloading files from HTTPS URLs. It respects the "https_proxy" preference setting (or the https_proxy environment variable) when configured.

See also

Preferences for the proxy setting

Public Functions

explicit **URLDownloader**(QWidget *parent = nullptr)

Construct a new *URLDownloader*.

Parameters

parent – Optional parent widget for progress dialogs

~URLDownloader()

Destructor.

URLDownloader(const *URLDownloader*&) = delete

URLDownloader(*URLDownloader*&&) = delete

URLDownloader &**operator=**(const *URLDownloader*&) = delete

URLDownloader &**operator=**(*URLDownloader*&&) = delete

bool **download**(const QString &url, const QString &file, bool showDialog = false, bool keepBackup = false)

Download a file from the given HTTPS URL to a local file.

Performs a synchronous (blocking) download of the resource at `url`, writing the result to the local file `file`. Respects the configured HTTPS proxy setting. Optionally display a dialog with the downloaded URL and the location of the downloaded file.

When a SHA256SUMS file is available in the same remote directory, the checksum of the downloaded data is verified *before* it replaces an existing file, so a corrupted download never clobbers a working file.

Parameters

- **url** – The HTTPS URL to download from
- **file** – The local file path to write to
- **showDialog** – Display a dialog with the downloaded URL and target file location while downloading
- **keepBackup** – Rename an existing target file to a backup name instead of replacing it in place; required to update a shared library that is currently loaded on Windows, where a loaded library can be renamed but not deleted or overwritten. The caller is responsible for removing the backup file eventually (LAMMPS-GUI does this at launch).

Returns
true if the download completed successfully, false otherwise

inline QString **errorString**() const

Return the last error message.

Returns
Human-readable error description or empty string

void **abort**()

Abort the current download and any further ones on this instance.

Safe to call from a slot triggered while *download()* blocks in its event loop (e.g. the Cancel button of a progress dialog). The in-flight request is aborted and all subsequent *download()* calls on this instance fail immediately, so one cancellation stops a whole batch of downloads.

inline bool **wasAborted**() const

Return whether the download was canceled via *abort()*

QString **getRemoteChecksum**(const QString &url)

Return the remote SHA-256 checksum for a given URL.

Fetches the SHA256SUMS file from the same remote directory as the resource at `url`, and returns the expected hex-hash for the resource's filename.

Parameters
url – The HTTPS URL of the resource

Returns
Hex-hash string or empty string if not found or on error

Public Static Functions

static QString **getLocalChecksum**(const QString &file)

Compute the local SHA-256 checksum for a given file.

Parameters
file – The local file path

Returns
Hex-hash string or empty string on error

DownloadProgress Class

class **DownloadProgress** : public QDialog

Splash-style transient dialog showing the progress of a batch download.

Shows a logo, a headline, a single activity line, a dedicated progress bar, and a Cancel button while files are downloaded. The bar is indeterminate (busy indicator) while the number of files is not yet known (e.g. during the initial fetch of a manifest) and switches to determinate per-file progress afterwards. Canceling (button, escape, or closing the dialog) emits QDialog::rejected(); the caller connects that to aborting the download. When the batch ends (success or failure), the caller must end the dialog with accept() – QDialog::close() implies reject() and would be indistinguishable from a user cancellation.

The setters show the dialog and process pending events, so the updated state is painted even when the caller immediately blocks in a synchronous download loop afterwards.

Public Functions

explicit **DownloadProgress**(const QString &headline, const QPixmap &logo, QWidget *parent = nullptr)

Create the dialog.

Parameters

- **headline** – Bold headline describing the batch (e.g. the tutorial name)
- **logo** – Image shown to the left of the text (may be null)
- **parent** – Parent widget

~DownloadProgress() override = default

DownloadProgress() = delete

DownloadProgress(const *DownloadProgress*&) = delete

DownloadProgress(*DownloadProgress*&&) = delete

DownloadProgress &**operator=**(const *DownloadProgress*&) = delete

DownloadProgress &**operator=**(*DownloadProgress*&&) = delete

void **setBusy**(const QString &text)

Show indeterminate (busy) progress with the given activity text.

void **setProgress**(const QString &text, int value, int maximum)

Show determinate progress with the given activity text.

Parameters

- **text** – Current activity (e.g. name of the file being downloaded)
- **value** – Number of completed items
- **maximum** – Total number of items

FileViewer Class

class **FileViewer** : public QPlainTextEdit

Read-only text viewer for displaying file contents.

FileViewer provides a simple read-only text window for viewing file contents. It's used in the context menu of the code editor to view files referenced in LAMMPS input scripts (data files, potential files, etc.). The viewer supports keyboard shortcuts for closing and stopping the simulation.

Public Functions

explicit **FileViewer**(const QString &filename, *LammpsGui* *lammpsgui, const QString &title = "", QWidget *parent = nullptr)

Constructor.

Parameters

- **filename** – Path to file to display
- **lammpsgui** – Pointer to *LammpsGui* for sending signals
- **title** – Window title (defaults to filename if empty)
- **parent** – Parent widget

~FileViewer() override = default

Destructor.

FileViewer() = delete

FileViewer(const *FileViewer*&) = delete

FileViewer(*FileViewer*&&) = delete

FileViewer &**operator=**(const *FileViewer*&) = delete

FileViewer &**operator=**(*FileViewer*&&) = delete

Protected Functions

void **resizeEvent**(QResizeEvent *event) override

Keep the menu bar across the top of the viewport.

Parameters
event – Resize event

void **changeEvent**(QEvent *event) override

Keep the fixed-width document font when the widget font changes.

Parameters
event – Change event

LogWindow Class

class **LogWindow** : public QPlainTextEdit

Text viewer for LAMMPS log output with warning/error detection.

LogWindow specializes QPlainTextEdit for LAMMPS log viewing. It highlights warnings and errors, detects embedded YAML data for extraction, provides navigation between warnings, and makes error URLs clickable.

Public Functions

LogWindow(const QString &filename, *LammpsGui* *lammpsgui, QWidget *parent = nullptr)

Constructor.

Parameters

- **filename** – Name of the input file the run belongs to (used for default save-file names)
- **lammpsgui** – Pointer to *LammpsGui* for sending signals
- **parent** – Parent widget

~LogWindow() override

Destructor.

void **reset**(const QString &filename)

Clear the window for a new run.

Discards the collected log text and the warning/error counters so the window can be reused instead of destroyed and recreated for every run. Keeps the position and size the window currently has on screen.

Parameters

filename – Name of the input file the new run belongs to

LogWindow() = delete

LogWindow(const *LogWindow*&) = delete

LogWindow(*LogWindow*&&) = delete

LogWindow &**operator=**(const *LogWindow*&) = delete

LogWindow &**operator=**(*LogWindow*&&) = delete

Protected Functions

void **closeEvent**(QCloseEvent *event) override

Handle window close event.

Parameters

event – Close event

void **mouseDoubleClickEvent**(QMouseEvent *event) override

Handle double-click to open URLs.

Parameters

event – Mouse event

void **contextMenuEvent**(QContextMenuEvent *event) override

Show context menu with log-specific actions.

Parameters
event – Context menu event

void **changeEvent**(QEvent *event) override

Keep the fixed-width document font when the inherited font changes.

Parameters
event – Change event

void **resizeEvent**(QResizeEvent *event) override

Keep the menu bar across the top of the viewport.

Parameters
event – Resize event

bool **checkYaml**()

Check if log contains embedded YAML data.

Returns
true if YAML data detected, false otherwise

FlagWarnings Class

class **FlagWarnings** : public QSyntaxHighlighter

Syntax highlighter for LAMMPS warning and error messages.

FlagWarnings extends QSyntaxHighlighter to detect and highlight warning and error messages in LAMMPS log output. It also detects and highlights URLs, enabling easy navigation and documentation access. The class maintains a count of warnings and updates a summary label.

Public Functions

explicit **FlagWarnings**(QLabel *label = nullptr, QTextDocument *parent = nullptr)

Constructor.

Parameters

- **label** – Optional label to display warning count summary
- **parent** – Text document to apply highlighting to

~FlagWarnings() override = default

Destructor.

FlagWarnings() = delete

FlagWarnings(const *FlagWarnings*&) = delete

FlagWarnings(*FlagWarnings*&&) = delete

FlagWarnings &**operator=**(const *FlagWarnings*&) = delete

FlagWarnings &**operator=**(*FlagWarnings*&&) = delete

inline int **getNWarnings**() const

Get the current number of warnings detected.

Returns
Number of warnings found in the document

void **reset**()

Clear the warning and line counters.

The counters are accumulated across calls to *highlightBlock()* and are never decremented, so they must be cleared explicitly when the attached document is reused for new content. Also resets the summary label to its empty-document text.

Public Static Functions

static QString **summaryText**(int nwarnings, int nlines)

The text of the summary label for the given counters.

The one place the format lives; also used to seed the label before any highlighting has run.

Parameters

- **nwarnings** – Number of warnings/errors counted
- **nlines** – Number of lines counted

Returns
Formatted summary text

static const QRegularExpression &**warningPattern**()

The pattern that marks a line as an error or warning.

Shared with the log window's search for the next warning, so the two cannot disagree about what counts as one.

static const QRegularExpression &**errorUrlPattern**()

The pattern that finds an error-explanation URL in a line.

Shared with the log window's context menu and double-click handling. The URL is the first capture group.

Protected Functions

void **highlightBlock**(const QString &text) override

Highlight a single block (line) of text.

Searches for warning/error patterns and URLs, applies formatting, and updates warning count.

Parameters
text – Text to highlight

ImageInfo Class

class **ImageInfo**

Store settings for displaying graphics from a fix or compute in a LAMMPS snapshot image.

Public Functions

ImageInfo() = delete

inline **ImageInfo**(bool _enabled, const QString &_style, int _colorstyle, const QString &_color, double _opacity, double _flag1, double _flag2)

Custom constructor

Public Members

bool **enabled**

display graphics if true

QString **style**

name of style

int **colorstyle**

color style for graphics: TYPE, ELEMENT, CONSTANT

QString **color**

custom color of graphics objects for style == CONSTANT

double **opacity**

opacity of graphics objects for style == CONSTANT

double **flag1**

Flag #1 for graphics.

double **flag2**

Flag #2 for graphics.

RegionInfo Class

class **RegionInfo**

Store settings for displaying a region in a LAMMPS snapshot image.

Public Functions

RegionInfo() = delete

inline **RegionInfo**(bool _enabled, int _style, const QString &_color, double _diameter, double _opacity, int _npoints)

Custom constructor

Public Members

bool **enabled**

display region if true

int **style**

style of region object: FRAME, FILLED, TRANSPARENT, or POINTS

QString **color**

color of region display

double **diameter**

diameter value for POINTS and FRAME

double **opacity**

opacity for TRANSPARENT

int **npoints**

number of points to be used for POINTS style region display

StdCapture Class

class **StdCapture**

Capture stdout output to a string buffer.

This class provides functionality to redirect and capture standard output (stdout) into a string buffer. Used to capture output from LAMMPS library calls for display in the GUI.

Public Functions

StdCapture()

Constructor - initializes capture buffers.

StdCapture(const *StdCapture*&) = delete

StdCapture(*StdCapture*&&) = delete

StdCapture &**operator=**(const *StdCapture*&) = delete

StdCapture &**operator=**(*StdCapture*&&) = delete

~StdCapture()

Destructor - closes the capture pipe descriptors.

void **beginCapture**()

Start capturing stdout.

Redirects stdout to an internal pipe for capture

bool **endCapture**()

Stop capturing stdout and restore original stdout.

Returns
true if capture was active, false otherwise

std::string **getCapture**()

Get all captured output and clear the buffer.

Returns
String containing all captured output

std::string **getChunk**()

Get a chunk of captured output without clearing.

Returns
String containing new output since last getChunk call

double **getBufferUse**() const

Get the buffer usage as a fraction of max buffer size.

Returns
Value between 0.0 and 1.0 indicating buffer fullness

inline bool **isUsable**() const

Whether stdout could be redirected at all.

False means nothing the library writes can ever be shown, and *diagnostic()* says why. Worth asking: the failure is otherwise completely silent, because printf() reports success either way and the runtime simply drops the bytes.

Returns
true when captured output can reach the caller

inline const std::string &**diagnostic**() const

Why capture is not available.

Returns
One line naming the step that failed, empty while capture works

inline size_t **totalRead**() const

Bytes retrieved through *getChunk()* since the capture began.

Returns
Byte count; zero after a whole run means the output was lost

std::string **probeRunEnd**()

Decide which side lost the output of a run that captured nothing.

Performs one more marker round trip while the capture is still active: a marker that comes back proves the redirect held for the whole run (the library's output went elsewhere), one that does not means stdout was re-pointed while the run was underway. Call before *endCapture()*; any real output drained alongside the marker is preserved and delivered through the following *endCapture()*/getCapture().

Returns
One line stating whether the redirect still worked at run end, empty when there is no active, usable capture to probe

Qt Helper Widgets

class **QHline** : public QFrame

Horizontal line widget for visual separation.

Provides a simple horizontal line widget for grouping UI elements in forms or dialogs.

Public Functions

QHline(QWidget *parent = nullptr)

Constructor.

Parameters
parent – Parent widget

---

class **QColorCompleter** : public QCompleter

Auto-completer for color name inputs.

Provides auto-completion for color names in text fields, suggesting valid color names for use with the LAMMPS dump image command.

Public Functions

QColorCompleter(QWidget *parent = nullptr)

Constructor.

Parameters
parent – Parent widget

---

class **QColorValidator** : public QValidator

Validator for color name inputs.

Ensures color inputs are names from the LAMMPS color list. Fix-up trims whitespace and lowercases the input.

Public Functions

QColorValidator(QWidget *parent = nullptr)

Constructor.

Parameters
parent – Parent widget

void **fixup**(QString &input) const override

Attempt to fix invalid color input.

Parameters
input – String to fix (modified in place)

QValidator::State **validate**(QString &input, int &pos) const override

Validate color input string.

Parameters

- **input** – String to validate
- **pos** – Cursor position (unused)

Returns
Validation state (Invalid, Intermediate, Acceptable)

---

class **RangeSlider** : public QSlider

Custom Qt Widget implementing a variant of QSlider that has two instead of one handle.

Public Functions

RangeSlider(Qt::Orientation ot = Qt::Horizontal, QWidget *parent = nullptr)

Constructor.

Copyright Hoyoung Lee (2024-07-14).

hoyoung.yi@gmail.com

This software is a computer program whose purpose is to provide "Qt widget of range slider with two handles".

This software is governed by the CeCILL-A license under French law and abiding by the rules of distribution of free software. You can use, modify and/or redistribute the software under the terms of the CeCILL-A license as circulated by CEA, CNRS and INRIA at the following URL: "http://www.cecill.info".

As a counterpart to the access to the source code and rights to copy, modify and redistribute granted by the license, users are provided only with a limited warranty and the software's author, the holder of the economic rights, and the successive licensors have only limited liability.

In this respect, the user's attention is drawn to the risks associated with loading, using, modifying and/or developing or reproducing the software by the user in light of its specific status of free software, that may mean that it is complicated to manipulate, and that also therefore means that it is reserved for developers and experienced professionals having in-depth computer knowledge. Users are therefore encouraged to load and test the software's suitability as regards their requirements in conditions enabling the security of their systems and/or data to be ensured and, more generally, to use and operate it in the same conditions as regards security.

The fact that you are presently reading this means that you have had knowledge of the CeCILL-A license and that you accept its terms.

Parameters

- **ot** – Slider orientation (default: Horizontal)
- **parent** – Parent widget

inline int **low**() const

Return the rangeslider's current low handle value.

inline void **setLow**(int low_limit)

Set the rangeslider's current low handle value.

inline int **high**() const

Return the rangeslider's current high handle value.

inline void **setHigh**(int high_limit)

Set the rangeslider's current high handle value.

Signals

void **sliderMoved**(int, int)

This signal is emitted when sliderDown is true and one of the two sliders moves.

Protected Functions

void **paintEvent**(QPaintEvent *ev) override

handle paint event

void **mousePressEvent**(QMouseEvent *ev) override

handle mouse press event

void **mouseMoveEvent**(QMouseEvent *ev) override

handle mouse move event

inline int **pick**(QPoint const &pt) const

Extract the relevant coordinate from a point based on slider orientation.

int **pixelPosToRangeValue**(int pos)

Convert a pixel position along the slider to a range value.

Protected Attributes

int **lowLimit**

position of rangeslider's lower handle

int **highLimit**

position of rangeslider's upper handle

QStyle::SubControl **pressed_control**

currently pressed sub-control (handle)

int **tick_interval**

interval between tick marks

QSlider::TickPosition **tick_position**

position of tick marks relative to slider

QStyle::SubControl **hover_control**

sub-control currently under the mouse cursor

int **click_offset**

offset from handle center to click position

int **active_slider**

index of the currently active slider handle

---

class **RangeBandSlider** : public QSlider

Horizontal QSlider that colors an active sub-range of its track.

RangeBandSlider behaves like an ordinary horizontal QSlider for selecting the current value (e.g. an image index), but paints its track in two colors: values inside the active [low, high] range use the "active" color, while values outside it (the skipped images) use the "skipped" color. The handle is still drawn by the platform style, so it keeps its native appearance. The active range is expressed in the same units as the slider value and is updated with *setActiveRange()*.

Public Functions

inline explicit **RangeBandSlider**(QWidget *parent = nullptr)

Constructor.

Parameters
parent – Parent widget (optional)

~RangeBandSlider() override = default

Destructor.

RangeBandSlider(const *RangeBandSlider*&) = delete

RangeBandSlider(*RangeBandSlider*&&) = delete

RangeBandSlider &**operator=**(const *RangeBandSlider*&) = delete

RangeBandSlider &**operator=**(*RangeBandSlider*&&) = delete

void **setActiveRange**(int low, int high)

Set the active value range to highlight.

Parameters

- **low** – First in-range value (inclusive, in slider value units)
- **high** – Last in-range value (inclusive, in slider value units)

Protected Functions

void **paintEvent**(QPaintEvent *event) override

Paint a two-color track plus the native handle.

Parameters
event – The paint event (unused)

---

class **VerticalLabel** : public QWidget

Widget displaying text rotated 90 degrees counter-clockwise.

Renders text vertically, optimized for y-axis titles in charts where horizontal space is limited.

Public Functions

explicit **VerticalLabel**(const QString &text, QWidget *parent = nullptr)

Constructor.

Parameters

- **text** – Text to display
- **parent** – Parent widget

void **setText**(const QString &text)

Update the displayed text.

Parameters

text – New text to display

QString **text**() const

Get the displayed text.

Returns

Current text

Protected Functions

void **paintEvent**(QPaintEvent *event) override

event handler for requests to draw all or parts of the custom widget

QSize **sizeHint**() const override

return the recommended size for the widget

QSize **minimumSizeHint**() const override

return the recommended minimum size for the widget

StdoutSilencer Class

class **StdoutSilencer**

RAII guard that silences stdout for the duration of its scope.

Calls silenceStdout() on construction and restoreStdout() on destruction, so stdout is restored on every exit path from the enclosing scope, including early returns and exceptions. Use in place of manual silenceStdout() / restoreStdout() pairs.

> **Note**
>
> Not thread-safe. Must only be used from the main thread.

Public Functions

inline **StdoutSilencer**()

inline **~StdoutSilencer**()

StdoutSilencer(const *StdoutSilencer*&) = delete

StdoutSilencer(*StdoutSilencer*&&) = delete

StdoutSilencer &**operator=**(const *StdoutSilencer*&) = delete

StdoutSilencer &**operator=**(*StdoutSilencer*&&) = delete

QtMessageSilencer Class

class **QtMessageSilencer**

RAII guard that collects Qt log messages instead of printing them.

Installs a Qt message handler on construction and restores the previous one on destruction. Debug, info, and warning messages emitted while the guard is alive are collected and can be retrieved with *messages()*; they are never printed. Critical and fatal messages are passed on to the previous handler, since those must not be swallowed.

The intended use is a scope whose failure is expected and handled, such as asking Qt to decode an image in a format it may not support: some of Qt's image format plugins print a warning for every file they reject, which would otherwise be repeated for each file and each attempt. Retrieve the collected text with *messages()* and report it once if the operation fails for good.

> **Note**
>
> The Qt message handler is process-wide, so the guard also captures messages emitted by other threads while it is alive. Keep the guarded scope short and use it only from the main thread.

Public Functions

QtMessageSilencer()

~QtMessageSilencer()

QtMessageSilencer(const *QtMessageSilencer*&) = delete

QtMessageSilencer(*QtMessageSilencer*&&) = delete

QtMessageSilencer &**operator=**(const *QtMessageSilencer*&) = delete

QtMessageSilencer &**operator=**(*QtMessageSilencer*&&) = delete

QString **messages**() const

The messages collected so far, one per line.

Returns

Collected text, or an empty string if nothing was captured

PlotData and File Parsers

Column-oriented numeric data model (`src/plotdata.h`) and the parsers for external data files (whitespace/`.dat`, CSV, LAMMPS YAML, and JSON) used to plot data from files.

Functions

PlotData **parsePlotCsv**(const QString &text, QString *error = nullptr)

Parse comma-separated values into a *PlotData*.

The first line is treated as a header of column names unless every field parses as a number, in which case generic names are generated and the line is treated as data.

Parameters

- **text** – File contents
- **error** – Optional out-parameter set to a message on failure

Returns

Parsed table (empty on failure)

PlotData **parsePlotWhitespace**(const QString &text, QString *error = nullptr)

Parse whitespace-separated columns (gnuplot / LAMMPS `.dat`) into a *PlotData*.

Lines beginning with # are comments; the last comment line before the data is used for column names if its field count matches the data.

Parameters

- **text** – File contents
- **error** – Optional out-parameter set to a message on failure

Returns

Parsed table (empty on failure)

PlotData **parsePlotYaml**(const QString &text, QString *error = nullptr)

Parse YAML (LAMMPS thermo `keywords:+data:` or a sequence of maps) into a *PlotData*.

Parameters

- **text** – File contents
- **error** – Optional out-parameter set to a message on failure

Returns

Parsed table (empty on failure)

PlotData **parsePlotJson**(const QByteArray &bytes, QString *error = nullptr)

Parse JSON into a *PlotData*.

Supports two simple shapes only: an array of equally long numeric rows (`[[...],[...]]`, generic column names) and an object mapping names to numeric arrays (`{"a":[...],"b":[...]}`).

Parameters

- **bytes** – File contents
- **error** – Optional out-parameter set to a message on failure

Returns

Parsed table (empty on failure)

PlotData **loadPlotData**(const QString &filename, QString *error = nullptr)

Load a file into a *PlotData*, choosing the parser by file extension or content.

Parameters

- **filename** – Path to the data file (.csv, .yaml/.yml, .json; else the whitespace format, with a content-sniffing fallback to YAML/JSON)
- **error** – Optional out-parameter set to a message on failure

Returns

Parsed table (empty on failure)

QString **writePlotCsv**(const *PlotData* &data)

Format a *PlotData* as comma-separated values.

Round-trips through parsePlotCsv().

Parameters

data – Table to format

Returns

CSV text: a header row of column names, then one row per data point

QString **writePlotDat**(const *PlotData* &data, const QString &source = QString())

Format a *PlotData* as whitespace-separated columns (gnuplot style)

Parameters

- **data** – Table to format
- **source** – Optional description placed in the leading comment line

Returns

Text with a # comment header carrying the column names, then the numeric columns; round-trips through parsePlotWhitespace()

QString **writePlotYaml**(const *PlotData* &data)

Format a *PlotData* as a LAMMPS-style thermo YAML document.

Parameters

data – Table to format

Returns

YAML text with a quoted `keywords:` list and a `data:` list of rows; round-trips through parsePlotYaml()

class **PlotData**

#include <plotdata.h> Column-oriented table of named numeric columns.

Holds a set of equally long columns of doubles, each with a name. It is the GUI-free data model shared by the file parsers and the chart window when plotting external data.

Public Functions

PlotData() = default

~PlotData() = default

PlotData(const *PlotData*&) = default

Copy constructor.

PlotData(*PlotData*&&) = default

Move constructor.

PlotData &**operator=**(const *PlotData*&) = default

Copy assignment.

PlotData &**operator=**(*PlotData*&&) = default

Move assignment.

inline int **columnCount**() const

Number of columns.

inline int **rowCount**() const

Number of rows (length of the columns; 0 if empty)

inline bool **isEmpty**() const

True if there are no columns or no rows.

inline const QString &**columnName**(int c) const

Name of a column.

Parameters
c – Column index

Returns
Column name

inline const QStringList &**columnNames**() const

All column names.

inline const std::vector<double> &**column**(int c) const

Read access to a column's values.

Parameters
c – Column index

Returns
Reference to the column data

void **setColumnNames**(const QStringList &columnNames)

Reset the table to a fresh set of (empty) named columns.

Parameters
columnNames – Names of the columns to create

void **renameColumns**(const QStringList &newNames)

Rename columns in place without clearing data.

Parameters
newNames – New names; entries beyond *columnCount()* are ignored, missing ones keep the old name

bool **appendRow**(const std::vector<double> &row)

Append one row of values, one per column.

Parameters
row – Values to append (size must equal *columnCount()*)

Returns
true on success, false if the size does not match

void **addColumn**(const QString &name, std::vector<double> data)

Append a complete named column.

Used by the column-wise parsers (e.g. JSON object-of-arrays).

Parameters

- **name** – Column name
- **data** – Column values

Private Members

QStringList **names**

per-column names

std::vector<std::vector<double>> **cols**

column-major numeric payload

struct **PlotErrors**

#include <plotdata.h> Per-column error bars, parallel to the columns of a *PlotData*.

Entry `i` of *upper* holds the (upper) error of column `i`, or is empty when that column has no error bars. Errors are kept beside the table rather than as extra columns so that they never appear as plottable columns of their own, and so that columns added later (e.g. a derived column) simply have none.

lower is only filled for bars that are not symmetric around the value – the min/max spread of a set of blocks is the case that needs it. While it is empty the bars extend by *upper* in both directions.

Public Functions

inline bool **isEmpty**() const

True if no column carries error bars.

inline bool **isAsymmetric**() const

True if the bars extend by different amounts up and down.

inline std::size_t **columnCount**() const

Number of columns covered.

inline void **clear**()

Drop all error bars.

inline void **resize**(std::size_t ncol)

Grow or shrink to `ncol` columns, keeping the existing ones.

inline void **appendEmpty**()

Append one column without error bars.

Public Members

std::vector<std::vector<double>> **upper**

upper (or symmetric) error per column

std::vector<std::vector<double>> **lower**

lower error per column (empty = symmetric)

Block-Structured fix ave/* Files

Data model and parsers (`src/plotblockdata.h`) for the block-structured output files of `fix ave/time` in vector mode, `fix ave/histo` and `fix ave/correlate`, which are reduced to a flat *PlotData* before they are plotted.

struct **PlotDataBlock**

One per-timestep block of a fix ave/* output file.

A block is the table LAMMPS writes for a single output timestep: the rows of a vector, the bins of a histogram, or the time windows of a correlation function. *scalars* carries the extra per-block numbers some styles put on the block header line (for `fix ave/histo` the total and missing counts and the value range); their names are in *PlotBlockData::scalarNames*.

Public Members

long long **step** = 0

timestep this block was written for

std::vector<double> **scalars**

per-block extras from the block header line

PlotData **rows**

the block's table of named columns

struct **PlotBlockData**

A parsed block-structured fix ave/* output file.

All blocks of a well-formed file have the same columns; blocks of differing shape can still occur when a file was appended to by a second fix, and are dropped by the reduction step rather than by the parser.

Public Functions

inline int **blockCount**() const

Number of blocks.

inline bool **isEmpty**() const

True if no block carrying data was found.

inline QStringList **columnNames**() const

Column names of the first block (empty if there is none)

Public Members

AveFileKind **kind** = *AveFileKind*::*Unknown*

detected format

QString **fixId**

fix ID from the default header, if present

QStringList **scalarNames**

names for *PlotDataBlock::scalars*

std::vector<*PlotDataBlock*> **blocks**

the per-timestep blocks, in file order

Functions

QString **aveFileKindName**(*AveFileKind* kind)

Human-readable name of a file kind, as shown in the import dialog.

Parameters
kind – Detected or user-selected kind

Returns
Descriptive name, e.g. “fix ave/histo”

PlotBlockData **parseAveBlocks**(const QString &text, QString *error = nullptr)

Parse the native (whitespace-separated) fix ave/* block format.

Recognizes both block layouts LAMMPS writes: a numeric block header line (`step nrows ...`) followed by `nrows` data rows, and the `# Timestep: N` comment delimiter of `fix ave/correlate/long`. Comment lines anywhere in the file re-synchronize the scanner, so a file that a second fix instance appended its own header to still parses.

Parameters

- **text** – File contents
- **error** – Optional out-parameter set to a message on failure

Returns
Parsed blocks (empty on failure)

PlotBlockData **parseAveBlocksYaml**(const QString &text, QString *error = nullptr)

Parse the vector-mode YAML variant `fix ave/time` writes.

The shape is a `keywords:` list followed by `data:` as a map of timestep keys, each holding a list of flow-style rows. A leading `Row` column is synthesized so that the table matches the native format, whose rows carry an explicit row index. Returns nothing for the scalar-mode YAML shape (a plain list of rows), which parsePlotYaml() already handles.

Parameters

- **text** – File contents
- **error** – Optional out-parameter set to a message on failure

Returns
Parsed blocks (empty on failure)

bool **looksLikeAveBlocks**(const QString &text)

Test whether a file's contents are block-structured fix ave/* output.

Requires an actual block structure, not just a matching header comment: the row-index column of every block has to count 1, 2, … n. A flat table is therefore never mistaken for a block file, whatever its comments say.

Parameters
text – File contents

Returns
true if the file should be imported through the block path

PlotBlockData **loadPlotBlockData**(const QString &filename, QString *error = nullptr)

Load a file as block-structured data, choosing the parser by extension or content.

Parameters

- **filename** – Path to the data file
- **error** – Optional out-parameter set to a message on failure

Returns
Parsed blocks; empty if the file is not block structured

QString **blockErrorTypeName**(*BlockErrorType* type)

Name of an error type as offered in the import dialog.

Parameters
type – Error type

Returns
Descriptive name, e.g. "standard deviation"

PlotData **singleBlock**(const *PlotBlockData* &data, int index)

Extract the rows of a single block (import mode "single block")

Parameters

- **data** – Parsed blocks
- **index** – Block index; clamped to the available range

Returns
That block's table, or an empty table if there are no blocks

BlockAverage **averageBlocks**(const *PlotBlockData* &data, int first, int last, *BlockErrorType* type)

Average a contiguous range of blocks row by row (import mode "average")

The last block of the range sets the expected shape; blocks of a different shape are dropped and counted in BlockAverage::skippedBlocks rather than silently reinterpreted. Error bars need at least two contributing blocks, and are computed in two passes so that a small spread on top of a large mean does not lose its significant digits.

Note that successive averaging windows are not strictly independent, so the standard error is a lower bound on the true uncertainty. BlockErrorType::MinMax makes no statistical claim at all: it just marks the range the blocks covered, which is why it is the one error type with an asymmetric result.

Parameters

- **data** – Parsed blocks
- **first** – First block of the range (inclusive)
- **last** – Last block of the range (inclusive)
- **type** – Which uncertainty to report as error bars

Returns
Means, error bars, and how many blocks contributed

AveImportDefaults **aveImportDefaults**(const *PlotBlockData* &data)

Pick the import defaults that suit a parsed file.

Histograms and correlation functions default to the block average, because their time evolution is rarely what is wanted. A file whose correlator accumulates over the whole run defaults to the last block instead: there every block is a successive estimate of the same quantity, so averaging them would be statistically wrong.

Parameters
data – Parsed blocks

Returns
Reduction mode and column roles to preselect in the dialog

Enums

enum class **AveFileKind**

Which LAMMPS fix wrote a block-structured data file.

The kind only selects the sensible import defaults (which column is the x axis, whether averaging blocks is meaningful); the parser itself is generic, so an unrecognized file still imports as `Unknown` without losing data.

Values:

enumerator **Unknown**

block structure recognized, but not the writing fix

enumerator **AveTimeVector**

`fix ave/time` in vector mode

enumerator **AveHisto**

`fix ave/histo` (and `fix ave/histo/weight`)

enumerator **AveCorrelate**

`fix ave/correlate`

enumerator **AveCorrelateLong**

`fix ave/correlate/long`

enumerator **AveChunk**

`fix ave/chunk`

enum class **BlockErrorType**

Which uncertainty a block average reports.

Values:

enumerator **None**

no error bars

enumerator **StdDev**

standard deviation of the blocks

enumerator **StdError**

standard error of the mean, sigma/sqrt(N)

enumerator **MinMax**

full spread: the bar spans the smallest to the largest value

Least-Squares Toolkit

Self-contained (Qt-free) dense linear-algebra and least-squares routines (`src/leastsquares.h`) used by the chart smoothing and reusable for polynomial and equation-of-state fits.

Typedefs

using **float_vect** = std::vector<double>

Dense vector of doubles used by the least-squares routines

using **int_vect** = std::vector<int>

Dense vector of integers used for LU pivot bookkeeping

Functions

float_mat **transpose**(const *float_mat* &a)

Return the transpose of a matrix.

Return the transpose of a matrix.

Parameters
a – Input matrix

Returns
Transposed matrix

float_mat **operator***(const *float_mat* &a, const *float_mat* &b)

Matrix-matrix multiplication.

Matrix-matrix multiplication.

Parameters

- **a** – Left operand
- **b** – Right operand (must have as many rows as a has columns)

Returns
Product matrix

float_mat **lin_solve**(const *float_mat* &A, const *float_mat* &a)

Solve the linear system A*X = a via in-place LU decomposition.

Setting a to the identity matrix yields the inverse of **A**.

Solve the linear system A*X = a via in-place LU decomposition.

Parameters

- **A** – Coefficient matrix
- **a** – Right-hand side(s); each column is an independent system

Returns
Solution matrix X with the same shape as a

float_mat **invert**(const *float_mat* &A)

Invert a square matrix using LU decomposition.

Invert a square matrix using LU decomposition.

Parameters
A – Square matrix to invert

Returns
Inverse of **A**

float_vect **sg_smooth**(const *float_vect* &v, std::size_t width, int deg)

Smooth a data vector with a Savitzky-Golay filter.

Fits a polynomial of degree deg to a sliding window of width 2*width+1 by least squares; non-symmetric windows are used near the borders.

Smooth a data vector with a Savitzky-Golay filter.

This method means fitting a polynomial of degree 'deg' to a sliding window of width 2w+1 throughout the data. The needed coefficients are generated dynamically by doing a least squares fit on a "symmetric" unit vector of size 2w+1, e.g. for w=2 b=(0,0,1,0,0). evaluating the polynomial yields the sg-coefficients. at the border non symmetric vectors b are used.

Parameters

- **v** – Input samples (assumed equally spaced)
- **width** – Half-window size; the filter window spans 2*width+1 points
- **deg** – Polynomial degree fitted in each window (0 = moving average)

Returns
Smoothed vector with the same length as v

class **float_mat** : public std::vector<*float_vect*>

#include <*leastsquares.h*> Simple dense matrix of doubles backed by a vector of rows.

Elements are indexed [row][column] with 0-based indices (C style). Because the storage is row-major, iterating over rows is cheaper than over columns. Used as the working type for the LU solver, matrix inverse, and the Savitzky-Golay coefficient generation.

Public Functions

float_mat() = delete

float_mat(*float_mat*&&) = default

Move constructor.

~**float_mat**() = default

float_mat &**operator=**(const *float_mat*&) = delete

float_mat &**operator=**(*float_mat*&&) = delete

float_mat(std::size_t rows, std::size_t cols, double def = 0.0)

Construct a rows x cols matrix filled with a default value.

Parameters

- **rows** – Number of rows
- **cols** – Number of columns
- **def** – Initial value for every element (default 0.0)

float_mat(const *float_mat* &m)

Copy constructor.

float_mat(const *float_vect* &v)

Construct a single-row matrix from a vector.

Parameters

v – Row contents

inline std::size_t **nr_rows**() const

Number of rows.

inline std::size_t **nr_cols**() const

Number of columns (size of the first row)

Post-Processing Analyses

Self-contained (Qt-free) post-processing analyses (`src/analysis.h`) used by the chart post-processing dialog.

Enums

enum class **FourierKind**

Kind of Fourier transform computed by fourierTransform()

Values:

enumerator **Cosine**

one-sided cosine transform 2 * integral y(x) cos(kx) dx

enumerator **Sine**

one-sided sine transform 2 * integral y(x) sin(kx) dx

enumerator **Power**

power spectrum |integral y(x) exp(-ikx) dx|^2

enum class **FourierWindow**

Window (taper) applied to the data before a Fourier transform.

Values:

enumerator **None**

transform the data as it stands

enumerator **Hann**

Hann taper: 1 at the first sample decaying to 0 at the last.

Functions

std::vector<double> **autocorrelation**(const std::vector<double> &y, int maxlag)

Normalized autocorrelation function (ACF) of a data series.

Uses the standard biased estimator $\mathrm{ACF}(k) = \frac{\sum_{i=0}^{N-1-k}(y_i-\bar{y})(y_{i+k}-\bar{y})}{\sum_{i=0}^{N-1}(y_i-\bar{y})^2}$.

Parameters

- **y** – Input samples (assumed equally spaced)
- **maxlag** – Largest lag to compute; values <= 0 or >= y.size() are clamped to y.size()-1

Returns

ACF values for lags 0..maxlag (length maxlag+1), normalized so that the lag-0 value is 1; an empty vector if the input has fewer than two samples or zero variance (a constant series)

std::vector<double> **fourierTransform**(const std::vector<double> &x, const std::vector<double> &y, const std::vector<double> &k, *FourierKind* kind, *FourierWindow* window = *FourierWindow*::*None*)

Fourier transform of sampled data by direct quadrature.

Computes the transform integral over the sampled x range with the trapezoidal rule, evaluated at each requested k value. Working on the integral directly keeps arbitrary (also non-uniformly spaced) x grids and arbitrary output grids possible; with the modest series lengths of chart data the O(N*M) cost is irrelevant. The cosine and sine transforms are the one-sided conventions with their factor 2, so the cosine transform of an autocorrelation function is the spectral density (Wiener-Khinchin); k is an angular frequency (rad per x unit). The Hann taper is 1 at the first sample and 0 at the last, which suppresses the ringing of data truncated before it has decayed to zero.

Parameters

- **x** – Sample positions, ascending

- **y** – Sample values (same length as x)
- **k** – Angular frequencies / wavenumbers to evaluate at
- **kind** – Which transform to compute
- **window** – Taper applied to y before transforming

Returns

One value per entry of k; empty if the input has fewer than two samples or the lengths of x and y differ

std::vector<double> **structureFactor**(const std::vector<double> &r, const std::vector<double> &g, double rho, const std::vector<double> &q, *FourierWindow* window = *FourierWindow*::*None*)

Static structure factor from a radial distribution function.

Computes $S(q) = 1 + 4\pi\rho \int_0^R r^2 \, (g(r) - 1) \, \frac{\sin(qr)}{qr} \, dr$ by the trapezoidal rule over the sampled r range. The $\sin(qr)/(qr)$ form is regular at $q = 0$ (where it is 1), so the q grid may start at zero. The -1 shift is applied here, so g is the plain radial distribution function as imported. The optional Hann taper is applied to $g(r) - 1$ to suppress the ringing from truncating it at a finite R where it has not fully decayed.

Parameters

- **r** – Radial sample positions, ascending
- **g** – Radial distribution function values (same length as r)
- **rho** – Number density N/V, in the units of the r axis cubed
- **q** – Wavenumbers to evaluate at (angular, rad per r unit)
- **window** – Taper applied to g(r)-1 before transforming

Returns

S(q), one value per entry of q; empty if the input has fewer than two samples or the lengths of r and g differ

Variables

constexpr double **MY_PI_CONST** = 3.14159265358979323846

Pi, spelled out: M_PI needs feature-test macros on some of the platforms the packaging cross-compiles for

---

Curve Fitting

Linear-least-squares curve fits (src/fitting.h) – polynomial and 4-parameter Birch-Murnaghan equation of state – built on the leastsquares toolkit and used by the chart post-processing dialog.

Functions

PolynomialFit **polynomialFit**(const std::vector<double> &x, const std::vector<double> &y, int degree)

Least-squares polynomial fit of y(x)

Parameters

- **x** – Abscissa values
- **y** – Ordinate values (same length as x)
- **degree** – Polynomial degree (>= 0)

Returns
Fit result; ok is false if the sizes mismatch or there are fewer than degree+1 points

double **evalPolynomial**(const std::vector<double> &coeffs, double x)

Evaluate a polynomial at a point.

Parameters

- **coeffs** – Coefficients c0..cn (y = sum_k c_k x^k)
- **x** – Evaluation point

Returns
Polynomial value

EosFit **birchMurnaghanFit**(const std::vector<double> &v, const std::vector<double> &e)

4-parameter Birch-Murnaghan EOS fit of energy versus volume

Parameters

- **v** – Volumes (must be positive)
- **e** – Energies (same length as v)

Returns
Fit result; ok is false if the sizes mismatch, there are fewer than four points, a volume is non-positive, or no physical minimum exists

double **evalBirchMurnaghan**(const *EosFit* &fit, double v)

Evaluate the fitted Birch-Murnaghan energy at a volume.

Parameters

- **fit** – Fit result (uses its a/b/c/d coefficients)
- **v** – Volume

Returns
Energy E(V)

struct **PolynomialFit**

#include <fitting.h> Result of a least-squares polynomial fit.

Public Members

std::vector<double> **coeffs**

coefficients c0..cn, i.e. y = sum_k c_k x^k

double **rms** = 0.0

root-mean-square residual

bool **ok** = false

true if the fit succeeded

struct **EosFit**

#include <fitting.h> Result of a 4-parameter Birch-Murnaghan equation-of-state fit.

The fitted model is the linear-in-coefficients form $E(V) = a + b\,V^{-2/3} + c\,V^{-4/3} + d\,V^{-2}$, from which the physical quantities are derived.

Public Members

double **a** = 0.0

constant coefficient

double **b** = 0.0

coefficient of V^(-2/3)

double **c** = 0.0

coefficient of V^(-4/3)

double **d** = 0.0

coefficient of V^(-2)

double **v0** = 0.0

equilibrium volume

double **e0** = 0.0

equilibrium energy

double **b0** = 0.0

bulk modulus (in energy/volume units)

double **b0prime** = 0.0

pressure derivative of the bulk modulus

double **rms** = 0.0

root-mean-square residual

bool **ok** = false

true if the fit succeeded and a minimum was found

Nonlinear Least Squares

Compact, self-contained (Qt-free) Levenberg-Marquardt solver (`src/levmar.h`) for nonlinear least-squares fits. The model is supplied as a residual/Jacobian callback, so the core is independent of how the model is expressed; it is driven by the custom-fit code with LeptonMini expressions and their symbolic derivatives, and solves the damped normal equations with the leastsquares LU solver. LeptonMini is based on the Lepton library by Peter Eastman and OpenMM contributors distributed under the MIT License.

Typedefs

using **LevmarModel** = std::function<bool(const std::vector<double> &params, std::vector<double> &residuals, std::vector<std::vector<double>> &jacobian)>

Callback evaluating residuals and the Jacobian at a parameter vector.

Given the current `params` (length n), it must fill `residuals` (length m) with model(x_i) - y_i and `jacobian` (m rows of n columns) with d model(x_i) / d p_j. Returning false signals an evaluation failure (e.g. a domain error) for that parameter vector; the solver treats the trial step as rejected rather than aborting.

Functions

LevmarResult **levmarFit**(int numResiduals, int numParams, const std::vector<double> &initial, const *LevmarModel* &model, int maxIterations = 200, double tolerance = 1.0e-12)

Levenberg-Marquardt nonlinear least-squares minimization.

Parameters

- **numResiduals** – Number of data points m (must be >= numParams)
- **numParams** – Number of free parameters n (> 0)
- **initial** – Initial parameter guess (length numParams)
- **model** – Residual/Jacobian callback
- **maxIterations** – Maximum number of outer iterations
- **tolerance** – Relative cost-change convergence threshold

Returns

Fit result; ok is false (with a message) on bad dimensions or a failed initial evaluation

struct **LevmarResult**

#include <levmar.h> Result of a Levenberg-Marquardt nonlinear least-squares fit.

Public Members

std::vector<double> **params**

fitted parameter values (empty on failure)

double **rms** = 0.0

root-mean-square residual at the solution

int **iterations** = 0

number of outer iterations performed

bool **ok** = false

true if the fit ran to a usable solution

std::string **message**

human-readable status/diagnostic

Custom-Function Evaluation and Fitting

Evaluation and nonlinear fitting of user-supplied mathematical expressions (`src/customfunc.h`) via the vendored LeptonMini parser, used for custom-function plotting and custom curve fits in the chart post-processing dialog. The fit builds its Jacobian from LeptonMini's analytic derivatives and minimizes with the Levenberg-Marquardt solver. LeptonMini is based on the Lepton library by Peter Eastman and OpenMM contributors distributed under the MIT License.

Functions

ColumnRefExpr **substituteColumnRefs**(const QString &expr, const QStringList &names)

Replace `{name}` column references with generated Lepton variables.

Column values are referenced in braces, like Python format strings: `{name}` is the column's value in the current row, and `{name:first}`, `{name:last}`, `{name:min}`, `{name:max}`, and `{name:mean}` are per-column constants. A colon inside braces is always the accessor separator, so a column whose name contains a colon (or a brace) cannot be referenced; the dialogs refuse such names for user-entered columns, and a file-supplied one has to be renamed before it can be used. Text outside braces is passed through untouched and is never a column lookup.

Each distinct (column, accessor) pair is replaced by one generated variable and reported in *ColumnRefExpr::refs* so the caller can bind it. An unknown name, an unknown accessor, an empty span, or an unmatched brace fails with a message naming the offender and the available columns.

Parameters

- **expr** – Expression with braced column references
- **names** – Column names, referenced by exact (case-sensitive) match

Returns
Rewritten expression plus the references, or an error

QString **renameColumnRefs**(const QString &expr, const QString &oldName, const QString &newName)

Rewrite the `{name}` references of a renamed column.

Replaces `{oldName}` and `{oldName:accessor}` spans with the same reference to `newName`, leaving everything else (including references to other columns) untouched. This is what keeps stored derived-column expressions valid when a column is renamed.

Parameters

- **expr** – Expression with braced column references
- **oldName** – Column name to rewrite
- **newName** – Replacement column name

Returns
The rewritten expression

double **columnAccessor**(const std::vector<double> &column, const QString &accessor)

Evaluate a per-column accessor constant.

Parameters

- **column** – Column values
- **accessor** – One of "first", "last", "min", "max", "mean"

Returns
The accessor value, or NaN for an empty column or unknown accessor

CustomCurve **evalCustomCurve**(const QString &expression, double xmin, double xmax, int nsamples, const QString &variable = QStringLiteral("x"))

Parse and evaluate a single-variable expression over an x range.

Parses `expression` with the vendored LeptonMini parser, optimizes it, and evaluates it at `nsamples` + 1 equally spaced points spanning [`xmin`, `xmax`]. Points whose value is not finite (NaN/inf) are omitted.

Parameters

- **expression** – Math expression in the variable `variable` (LeptonMini syntax)
- **xmin** – Lower bound of the sampling range
- **xmax** – Upper bound of the sampling range
- **nsamples** – Number of sub-intervals (clamped to >= 1); nsamples+1 points
- **variable** – Name of the independent variable in `expression` (default "x")

Returns

Curve result; on a parse/evaluation error *CustomCurve::ok* is false and *CustomCurve::error* describes the problem

CustomFit **fitCustomCurve**(const QString &expression, const QList<*FitParam*> &initialParams, const std::vector<double> &xdata, const std::vector<double> &ydata, double xmin, double xmax, int nsamples, const QString &variable = QStringLiteral("x"), const std::vector<double> &weights = {})

Nonlinear least-squares fit of a custom expression to (x, y) data.

Fits `expression` – a function of the independent variable `variable` and the named parameters in `initialParams` – to the data (`xdata`, `ydata`) by Levenberg-Marquardt. The Jacobian is built from analytic derivatives of the expression with respect to each parameter (via LeptonMini's symbolic differentiation). On success the fitted model is sampled at `nsamples` + 1 points over [`xmin`, `xmax`] for overlaying on the chart.

Parameters

- **expression** – Math expression in `variable` and the parameter names
- **initialParams** – Parameters with their initial guesses (at least one)
- **xdata** – Independent-variable data
- **ydata** – Dependent-variable data (same length as `xdata`)
- **xmin** – Lower bound for sampling the fitted curve
- **xmax** – Upper bound for sampling the fitted curve
- **nsamples** – Number of sub-intervals (clamped to >= 1); nsamples+1 points
- **variable** – Name of the independent variable (default "x")
- **weights** – Optional per-point weights w_i for a weighted fit, which minimizes sum w_i (model_i - y_i)^2. Empty (the default) or a wrongly sized vector means every point counts the same; negative entries are treated as zero. Weighting decides which part of the data a model that cannot describe all of it will follow

Returns

Fit result; on a parse/dimension/evaluation error *CustomFit::ok* is false and *CustomFit::error* describes the problem. *CustomFit::rms* is the plain, unweighted residual either way, so that fits with different weightings stay comparable

class **CompiledExpression**

#include <customfunc.h> A parsed and compiled LeptonMini expression with QString error reporting.

Confines the LeptonMini (std::string) parsing boundary to one translation unit, the way *LammpsWrapper* confines the LAMMPS C API. Construct from a QString expression, check *isValid* / *error*, then call *evaluate* repeatedly with a name -> value variable map. *evaluate* propagates the LeptonMini exception thrown for an unbound variable, so callers that may reference variables not present in the map should evaluate inside a try block.

Public Functions

explicit **CompiledExpression**(const QString &expression)

Parse, optimize, and compile `expression` (a LeptonMini string)

~CompiledExpression()

CompiledExpression() = delete

CompiledExpression(const *CompiledExpression*&) = delete

CompiledExpression(*CompiledExpression*&&) = delete

CompiledExpression &**operator=**(const *CompiledExpression*&) = delete

CompiledExpression &**operator=**(*CompiledExpression*&&) = delete

inline bool **isValid**() const

True if the expression parsed and compiled successfully.

inline const QString &**error**() const

Parse-error message (empty when *isValid* is true)

double **evaluate**(const std::map<std::string, double> &variables) const

Evaluate with the given variable bindings (may throw on unbound vars)

Private Members

std::unique_ptr<*LeptonMini*::ExpressionProgram> **program**

compiled program (null if invalid)

bool **valid** = false

parse/compile succeeded

QString **errorMsg**

parse error (empty when valid)

struct **ColumnRef**

#include <customfunc.h> One resolved `{name}` or `{name:accessor}` column reference.

Produced by substituteColumnRefs(). An empty *accessor* means the reference stands for the column's value in the current row and must be rebound for every row; otherwise it is one of the per-column constants ("first", "last", "min", "max", "mean") and can be bound once.

Public Members

int **column** = -1

index of the referenced column in the name list

QString **accessor**

empty = current-row value, else the accessor name

QString **variable**

generated variable the braced span was replaced with

struct **ColumnRefExpr**

#include <customfunc.h> Result of rewriting the `{name}` column references of an expression.

Public Members

bool **ok** = false

true if every braced span resolved to a column

QString **error**

human-readable error message when *ok* is false

QString **expr**

rewritten expression ready for the Lepton parser

QList<*ColumnRef*> **refs**

the distinct references, one per generated variable

struct **CustomCurve**

#include <customfunc.h> Result of sampling a custom expression over an x range.

Public Members

bool **ok** = false

true if the expression parsed and evaluated

QString **error**

human-readable error message when *ok* is false

QList<QPointF> **points**

sampled (x, y) points; non-finite y values are skipped

struct **FitParam**

#include <customfunc.h> A named nonlinear-fit parameter.

Carries the initial guess on input to fitCustomCurve() and the fitted value on output.

Public Members

QString **name**

parameter name as it appears in the expression

double **value** = 0.0

initial guess (input) / fitted value (output)

struct **CustomFit**

#include <customfunc.h> Result of a nonlinear least-squares fit of a custom expression.

Public Members

bool **ok** = false

true if the fit produced a usable solution

QString **error**

human-readable error message when *ok* is false

QList<*FitParam*> **params**

fitted parameters, in the input order

QList<QPointF> **curve**

fitted model sampled over the x range

double **rms** = 0.0

root-mean-square residual at the solution

int **iterations** = 0

Levenberg-Marquardt iterations performed.

namespace **LeptonMini**

Helper Functions

Functions

QColor **chartPaletteColor**(int index)

The palette color for a configured series color index.

Parameters

index – Index into the palette; the first entry when out of range

Returns

The color

QColor **configuredChartColor**(const QString &key, int fallback)

The configured default color of a series kind.

Parameters

- **key** – Settings key of the color index in the charts group (e.g. `Keys::RAWBRUSH`)
- **fallback** – Index to use when the key is not set

Returns

The palette color for the configured index

QComboBox *makeChartColorCombo(int current)

Combo box offering the palette colors by name.

Parameters
current – Index of the preselected color

Returns
The new widget, without a parent

QComboBox *makeChartModeCombo(int current)

Combo box offering the display modes in ChartDisplayMode order.

Each item carries its mode as integer item data, so a choice can be read back through `currentData()` as well as through `currentIndex()`.

Parameters
current – Index of the preselected mode

Returns
The new widget, without a parent

QComboBox *makePlotChoiceCombo(int current)

Combo box offering the plot data choices Raw, Smooth, and Both.

Parameters
current – Index of the preselected choice

Returns
The new widget, without a parent

QDoubleSpinBox *makeLineWidthSpin(double value)

Spin box for a series line width within the configured limits.

Parameters
value – Preset width

Returns
The new widget, without a parent

QDoubleSpinBox *makePointSizeSpin(double value)

Spin box for a marker diameter within the configured limits.

Parameters
value – Preset diameter

Returns
The new widget, without a parent

Functions

QFont monoFontFromSettings()

Build the configured fixed-width font from the application settings.

Returns
Fixed-pitch QFont with the family and point size stored in the settings, falling back to the platform default GUI_MONOFONT

void reassertMonoFont(QTextDocument *document)

Re-assert the configured fixed-width font on a text view's document.

Docked, a text view is a child of the main window and inherits its proportional font; QPlainTextEdit reacts to the resulting FontChange event by adopting the widget font as the document font, and the view loses its fixed

pitch. A view that must keep it calls this from its changeEvent() override on QEvent::FontChange. Setting only the document font does not feed back into the widget font, so this cannot recurse.

Parameters
document – The view's text document (no-op if null)

int **dateCompare**(const QString &one, const QString &two)

Compare two date strings in LAMMPS "DD MMM YYYY" format (e.g. "22 Jul 2025")

Parameters

- **one** – First date string
- **two** – Second date string

Returns
-1 if one < two, 0 if equal, 1 if one > two

QStringList **splitLine**(const QString &text)

Split a string into words while respecting quotes.

Parameters
text – The string to split

Returns
List of words extracted from the string

void **setDialogIcons**(QMessageBox &mb, const QString &iconPath)

Apply the bundled SVG icons to a QMessageBox.

Replaces the standard large icon of the message box with the given bundled SVG icon and sets the window icon and a styled standard "Ok" button consistently. Custom buttons can be added after this call.

Parameters

- **mb** – Message box to style
- **iconPath** – Resource path of the SVG icon used as the large dialog icon

void **information**(QWidget *parent, const QString &title, const QString &text1, const QString &text2 = QString())

Provide standardized information dialog.

Parameters

- **parent** – Pointer to parent widget
- **title** – Information dialog title
- **text1** – Information message part 1
- **text2** – Information message part 2 (optional)

void **critical**(QWidget *parent, const QString &title, const QString &text1, const QString &text2 = QString())

Provide standardized critical error dialog.

Parameters

- **parent** – Pointer to parent widget
- **title** – Error dialog title
- **text1** – Error message summary
- **text2** – Detailed error message (optional)

void **warning**(QWidget *parent, const QString &title, const QString &text1, const QString &text2 = QString())

Provide standardized custom warning dialog.

Parameters

- **parent** – Pointer to parent widget
- **title** – Warning dialog title
- **text1** – Warning message summary
- **text2** – Detailed warning message (optional)

QString **getLammpsLibName**()

Provide platform specific name of a LAMMPS shared library.

Returns

String with the filename or an empty string if compiled without plugin support

QString **getLammpsDownloadUrl**()

Provide platform specific URL for downloading a LAMMPS shared library.

Returns

String with the URL or an empty string if compiled without plugin support or with a compiler that is incompatible with the pre-compiled libraries (MSVC)

void **exportImage**(QWidget *parent, QImage *image, const QString &title, const QString &defaultname)

Save image directly or convert with ImageMagick.

Parameters

- **parent** – Pointer to parent widget
- **image** – Pointer to image class
- **title** – Warning dialog title if failed
- **defaultname** – Default file name offered by the save dialog (resolved relative to the current working directory)

QString **defaultFileStem**(const QString &filename)

Derive the default save-file name stem from an input or data file name.

Strips any directory part, a leading "in." prefix, and any trailing known file extensions (input, plottable data, log, restart, image, and movie formats), so "in.melt", "melt.lmp", or "melt.lmp.txt" all yield "melt". Falls back to "lammps" when nothing remains.

Parameters

filename – Name of the file the stem is derived from (may include a path)

Returns

the stem to build default save-file names from

QString **ensureFileSuffix**(const QString &filename, const QString &suffix)

Append a default suffix to a file name that has no suffix.

Parameters

- **filename** – File name selected in a save dialog
- **suffix** – Default suffix (without the leading dot)

Returns
the file name with the default suffix appended if it had none

bool **hasExe**(const QString &exe)

Check if an executable is in the executable search path.

Uses findExe(), so the macOS package manager fallback locations apply.

Parameters
exe – The executable name to search for

Returns
true if executable is found, false otherwise

QString **findExe**(const QString &exe)

Find an executable in the executable search path.

On macOS, an application launched from the Finder inherits a minimal PATH without the common package manager locations, so the Homebrew (Arm and Intel macs) and MacPorts binary folders are searched as a fallback. Launch external helper programs with the path returned by this function rather than the bare executable name, so they are also found in that case.

Parameters
exe – The executable name to search for

Returns
Full path to the executable or an empty string when not found

QString **renameToBackup**(const QString &file)

Rename a file to a backup name with the Cfg::BACKUP_SUFFIX suffix.

An existing backup file is replaced. When it cannot be removed (on Windows a loaded shared library is locked against deletion), a numbered backup name is used instead. Windows does permit renaming a locked file, so this is the way to move a loaded shared library out of the way before an update. The caller is responsible for removing the backup file eventually.

Parameters
file – Path of the file to rename

Returns
Path of the backup file or an empty string on failure

bool **isImageFile**(const QString &filename)

Check whether a file is (likely) an image.

Recognizes the formats Qt can decode plus common ImageMagick-only formats (e.g. tga, eps, sgi) so callers can route them through a conversion step.

Parameters
filename – Path to the file

Returns
true if the extension is a known image type, or the file exists and QImageReader recognizes its contents as an image

bool **isMovieFile**(const QString &filename)

Check whether a file is a movie (video) file.

An animated GIF is both an image and a movie, and isImageFile() also claims it. Callers that route a file to either destination must therefore test isMovieFile() first.

Parameters
filename – Path to the file

Returns
true if the extension is a known movie type, or the file is an animated GIF with more than one frame

bool **isRestartFile**(const QString &filename)

Check whether a file is a LAMMPS binary restart file.

Parameters
filename – Path to the file

Returns
true if the file exists and begins with the LAMMPS restart magic string

bool **looksLikeBinaryFile**(const QString &filename)

Heuristic check whether a file is binary rather than text.

Null bytes are essentially absent from text (ASCII, UTF-8, Latin-1) but ubiquitous in binary formats (IEEE floats, packed integers, padding). This is the same heuristic used by git, grep, and the POSIX file utility. Returns false for files that cannot be opened (callers handle that separately).

Parameters
filename – Path to the file

Returns
true if the first 8 KB of the file contains a null byte

void **relaunchApplication**()

Re-exec the current LAMMPS-GUI process in place (e.g. to reload the plugin)

Replaces the running process with a fresh launch of the same executable. On success it does not return; it returns only if the re-exec failed, so callers must handle that case (typically warn and/or exit).

void **relaunchOrExit**(QWidget *parent)

Relaunch LAMMPS-GUI, or report the failure and exit.

Wraps `relaunchApplication()` with the handling every caller needs: when the re-exec fails, the user is told that the saved settings take effect on the next start and the application exits, since continuing in a state that no longer matches the settings is worse than stopping.

Parameters
parent – Parent widget for the error dialog

void **purgeDirectory**(const QString &dir)

Recursively delete all files in a directory.

Parameters
dir – The directory to purge

bool **isLightTheme**()

Determine if the current Qt theme is light or dark.

Returns
true if light theme, false if dark theme

int **showUnsavedChangesDialog**(QWidget *parent, const QString &filename, const QString &question)

Show a standardized "unsaved changes" confirmation dialog.

Provides a consistent confirmation dialog used whenever the user may lose unsaved changes (opening a new file, quitting, running LAMMPS, etc.).

Parameters

- **parent** – Pointer to the parent widget
- **filename** – Name of the file with unsaved changes
- **question** – Informative text explaining the context (e.g. "save before opening?")

Returns
QMessageBox::Yes, QMessageBox::No, or QMessageBox::Cancel

bool **confirmUnexpectedFile**(QWidget *parent, const QString &filename, const QString &kind)

Ask before opening a file that is not what it is being opened as.

For the cases where opening the wrong file is a mistake rather than an error: a binary in the editor, a picture handed to the plotter. Answering No is what Return and Escape do, because the usual reason to see this is a name that was mistyped or a file that was mis-picked. Ask only when the file fails the test for its kind – a file that looks right must never produce a dialog.

Parameters

- **parent** – Pointer to the parent widget
- **filename** – File about to be opened; only its name is shown
- **kind** – What it was expected to be, worded to follow "a": "text", "data", "image or movie"

Returns
true if the user wants to go ahead

void **styleDialogButtons**(QDialogButtonBox *box)

Apply the bundled SVG icons to a dialog button box's standard buttons.

Qt fetches `QDialogButtonBox` standard-button icons (Ok, Cancel, ...) via `QIcon::fromTheme()`, which falls back to the desktop theme because the bundled `lammpsgui` icon theme only carries the editor context-menu actions. This applies our own `dialog-ok` / `dialog-cancel` / `dialog-no` / `window-close` SVGs to whichever standard buttons the box contains, so every dialog button looks the same regardless of platform or desktop theme.

Parameters
box – The button box whose standard buttons should be re-iconed

void **styleMessageBoxButtons**(QMessageBox &mb)

Apply the bundled SVG icons to the standard buttons of a message box.

The `QMessageBox` counterpart of `styleDialogButtons()`: the same icons for the same buttons, so every dialog of the application looks alike. Buttons the box does not have are skipped, so this is called once the standard buttons are set.

Parameters
mb – Message box with its standard buttons already set

void **silenceStdout**()

Silence stdout by redirecting it to the null device.

Redirects stdout to /dev/null (Unix) or NUL: (Windows) to suppress all output. Does nothing if *StdCapture* is currently active or if stdout is already silenced.

> **Note**
> Not thread-safe. Must only be called from the main thread.

void **restoreStdout**()

Restore stdout after it was silenced.

Restores the original stdout file descriptor that was saved by silenceStdout(). Does nothing if stdout is not currently silenced.

> **Note**
> Not thread-safe. Must only be called from the main thread.

bool **isStdoutSilenced**()

Check if stdout is currently silenced.

Returns
true if silenceStdout() is active, false otherwise

void **notifyCaptureState**(bool active)

Notify the silence/restore system about *StdCapture* state changes.

Called by *StdCapture* to indicate whether it is actively capturing output. While capture is active, silenceStdout() becomes a no-op to avoid interfering with the capture pipe.

Parameters
active – true when *StdCapture* starts capturing, false when it stops

QImage **grayscaleImage**(const QImage &src)

Fade an image into an unmistakably inactive version of itself.

Removes the color and, in addition, pulls the gray levels towards Cfg::GRAYSCALE_MIDPOINT, keeping only Cfg::GRAYSCALE_CONTRAST of the original contrast: a merely desaturated icon retains all of its structure and still reads as active next to its colored counterpart.

Parameters
src – Image to convert

Returns
Grayscale, low-contrast copy of `src` with the original transparency

QPixmap **grayscalePixmap**(const QPixmap &src)

Fade a pixmap into an unmistakably inactive version of itself.

Applies grayscaleImage() to the pixmap. Used for the “inactive” state of a status icon, so the widget does not depend on the disabled-widget visual, which does not refresh reliably on all platforms (e.g. macOS 12) and cannot be applied to an enabled widget at all.

Parameters
src – Pixmap to convert

Returns
Grayscale, low-contrast copy of src with the original transparency

QSize **toolButtonSize**(const QAbstractButton *sample)

Square size for a toolbar/status-bar button from a sample’s size hint.

Implements the shared tool-button sizing policy: take the sample button’s minimum size hint height, enlarge it by Cfg::TOOLBAR_BUTTON_MARGIN, and return a square of that side. Compute this once per button row and reuse it for the row’s buttons (via styleToolButtons()) and any adjacent widgets (line edits, labels, spin boxes) that should share the button height.

Parameters
sample – Representative button (typically the first in the row)

Returns
Square button size in logical pixels

void **styleToolButtons**(const QSize &size, std::initializer_list<QAbstractButton*> buttons)

Apply the shared tool-button policy to a set of buttons.

Fixes every button to size (a square from toolButtonSize()) and gives it the standard Cfg::TOOLBAR_ICON_SIZE icon, so only a small, uniform padding remains between icon and frame. The image viewer, slide show, chart window, and editor status bar all use this so their button rows look identical.

Parameters

- **size** – Square button size (from toolButtonSize())
- **buttons** – Buttons to size and assign the standard icon size

void **applyWindowFlags**(QWidget *window)

Apply the shared window-manager hints to a top-level output window.

Strips the dialog property (so the window is an independent top-level window, not a transient that stays above its parent) and removes the minimize button. The maximize button is also removed, except on macOS where it is kept because removing it makes the window non-resizable there. Used for the log, chart, image, slide-show, file-viewer and variables windows so they share one frame policy.

> **Note**
> setWindowFlags() re-shows a hidden widget, so call this before a final hide() when the window must start hidden.

Parameters
window – Top-level window to adjust (no-op if null)

void **retireViewMenuBar**(QMenuBar *menubar)

Retire an output view’s own menu bar in the combined layout.

Docked, a view does not show a menu bar of its own: the main window puts the view’s *File* menu into its menu bar while the view has the focus. The menu bar object still exists, because it is where the menu was built, so it is simply hidden – which is enough on every platform but one.

On macOS a QMenuBar is not a widget in the window but a handle on the system-wide menu bar, and the last one to claim a window replaces the one before it. Docked, both the main window's menu bar and the view's live in the same window, so opening the first panel handed the system menu bar to a menu bar that is hidden and empty: everything but the application menu that macOS assembles itself disappeared. Detaching the view's menu bar from the platform gives the window back to the main window's, and costs nothing elsewhere, where a QMenuBar is an ordinary widget already.

In the individual-window layout each view is a window of its own and claims its own menu bar legitimately, so this is only for the docked case.

Parameters

menubar – Menu bar of a docked output view (no-op if null)

bool **installViewMenuBar**(QMenuBar *menubar, QMenu *file, const QList<QMenu*> &shared)

Give a view window its menu bar, or retire it in the combined layout.

In the individual-windows layout the bar shows the view's own menu followed by the application-wide menus of the main window, so a run can be started or stopped from the view without a second set of actions to keep in step (and without a second binding for their accelerators). In the combined layout the main window carries one menu bar for all views and puts the view's menu at its front while the view has the focus, so the view's own bar is retired instead (see `retireViewMenuBar()`).

Parameters

- **menubar** – The view's menu bar (no-op if null)
- **file** – The view's own menu
- **shared** – The main window's application-wide menus; empty standalone

Returns

true when the bar is in use, false when it was retired

void **layoutViewMenuBar**(QWidget *view, QMenuBar *menubar)

Stretch a view's menu bar across the top of the view.

A QPlainTextEdit has no layout slot for a menu bar, so the text views place theirs over the viewport (with the bar's height reserved as a viewport margin) and re-place it here on every resize. Does nothing while the bar is hidden, i.e. retired in the combined layout.

Parameters

- **view** – The view
- **menubar** – Its menu bar

QSize **viewerFitSize**(const QSize &content, const QSize &budget, int frame, int sbext)

Compute the scroll area size that shows the given content, within a budget.

Pure size computation behind fitViewerWindow(). The natural size is the content plus the scroll area frame, with no scroll bars. An axis whose natural size exceeds the budget is clamped and gets a scroll bar, which consumes `sbext` pixels of viewport on the *other* axis, so that axis is enlarged accordingly (still within its budget). The result depends only on the arguments – never on the current scroll bar visibility – so repeated calls with the same input always yield the same size.

Parameters

- **content** – Size of the displayed content (image) in pixels
- **budget** – Largest allowed outer scroll area size (e.g. a screen fraction)
- **frame** – Total scroll area frame thickness (2 * frameWidth())
- **sbext** – Scroll bar thickness (QStyle::PM_ScrollBarExtent)

Returns
Outer scroll area size that best fits the content within the budget

QSize **fitViewerWindow**(QWidget *window, QScrollArea *area, const QSize &content, const QSize &budget, const QSize &lastFit)

Resize a viewer window so its scroll area just fits the displayed image.

Shared auto-resize policy of the image viewer and the slide show: the scroll area is sized via viewerFitSize() and the window is resized around it. The scroll area's minimum size is pinned only for the duration of the resize, so the user can freely shrink the window afterwards. When the computed size equals `lastFit`, the window is left untouched; passing the previous return value back in keeps navigating a sequence of equally-sized images from ever moving or resizing the window.

While `window` is still hidden its layout uses unpolished style metrics, so the applied size is only approximate: the fit is applied but an invalid QSize is returned instead of the memoized size. The viewers use that in their showEvent() overrides to apply the fit once more on the shown window.

Parameters

- **window** – Top-level viewer window to resize
- **area** – Scroll area inside `window` that shows the content
- **content** – Size of the displayed content (image) in pixels
- **budget** – Largest allowed outer scroll area size (e.g. a screen fraction)
- **lastFit** – Return value of the previous call (default QSize() initially)

Returns
The scroll area size applied, or an invalid QSize while `window` is hidden; pass this value back as `lastFit` on the next call

template<typename **Recv**, typename **Func**>
QAction ***addMenuAction**(QMenu *menu, const QString &text, const QString &icon, *Recv* *receiver, *Func* slot)

Append an action with an optional icon and a triggered() handler to a menu.

Collapses the recurring "addAction() + setIcon() + connect()" idiom used when building context menus and tool menus across the widget classes.

Parameters

- **menu** – Menu to append the new action to
- **text** – Action label text
- **icon** – Resource path for the action icon (empty for no icon)
- **receiver** – Object that owns the slot/callable
- **slot** – Member function pointer or callable invoked on trigger

Returns
The created action, for any further configuration by the caller (e.g. setData())

bool **dockedLayout**()

Whether the output views are docked into the main window.

The layout is chosen once at startup (*WindowLayout* applies it), so this only reads the stored preference, or what forceLayout() was given instead. Widgets consult it for the things that make no sense in a dock, such as remembering their own window size.

Returns
true when the docked layout is in effect for this session

void **forceLayout**(bool docked)

Override the stored layout preference for this session.

What the -j/–joined and -w/–windows command-line flags do. The preference itself is left alone: the choice applies to the process it was given to and nothing else, which is also why a relaunch (which passes no arguments on) goes back to what the preferences say. Call before the first window is built, since that is when the layout is decided.

Parameters
docked – true for the combined main window, false for individual windows

void **setMainWindowShortcuts**(const QList<QKeySequence> &keys)

Record the key sequences the main window's menus already use.

Called once by the main window after its menus are built. A view that ends up as a dock panel lives inside that same window, so a sequence it binds would match at the same time as the menu does and Qt would fire neither. *WindowLayout* consults this when a widget actually becomes a panel; a view that stays a window of its own keeps all of its shortcuts.

Parameters
keys – Every shortcut reachable from the main window's menu bar

bool **isMainWindowShortcut**(const QKeySequence &keys)

Whether a key sequence belongs to the main window's menus.

Parameters
keys – Sequence to check

Returns
true if the main window already binds it

void **scopeShortcut**(QWidget *widget, QAction *action, const QKeySequence &keys)

Give a menu action a keyboard shortcut that is scoped to one widget.

A menu action is only associated with the menu it was added to, and a menu is a popup that never holds the keyboard focus. Associating the action with `widget` as well is therefore what makes Qt::WidgetWithChildrenShortcut usable here at all: without it the shortcut would never match. See addShortcut() for why the output windows want focus scope in the first place.

Parameters

- **widget** – Widget the shortcut belongs to
- **action** – Action to bind the shortcut to
- **keys** – Key sequence to bind

template<typename **Recv**, typename **Func**>
QShortcut ***addShortcut**(QWidget *widget, const QKeySequence &keys, *Recv* *receiver, *Func* slot)

Add a keyboard shortcut that is scoped to one widget.

The shortcut uses Qt::WidgetWithChildrenShortcut, so it fires only while the keyboard focus is inside `widget` rather than anywhere in its window. That is what keeps the per-window shortcuts of the output windows (several of which repeat main window accelerators such as Ctrl+S, Ctrl+Q or Ctrl+/) from becoming ambiguous overloads once those windows are docked into the main window instead of being windows in their own right.

Parameters

- **`widget`** – Widget the shortcut belongs to and that owns the QShortcut
- **`keys`** – Key sequence to bind
- **`receiver`** – Object that owns the slot/callable
- **`slot`** – Member function pointer or callable invoked on activation

Returns

The created shortcut, for any further configuration by the caller

5.4 Development Guidelines

Code Style

The project follows these coding conventions:

- **Indentation**: 4 spaces (no tabs)
- **Line length**: Maximum 100 characters
- **Formatting**: Enforced by `.clang-format` configuration (LLVM-based)
- **Comments**: Use Doxygen-style documentation comments
- **Naming**: - Classes: CamelCase (e.g., `CodeEditor`, `ChartWindow`) - Qt overrides: camelCase (e.g., `setFont`, `eventFilter`) - Custom methods/slots: camelCase (e.g., `stopRun`, `saveAs`) - Members: camelCase (e.g., `reformatOnReturn`)

Documentation

Added functionality should be documented both in the User's Guide and the Programmer's Guide section of the documentation. The documentation should have suitable `.. index::` directives to populate the index with suitable keywords. The documentation is written in reStructuredText and imports documentation of the source code from Doxygen using the Breathe Sphinx plugin. The documentation should translate cleanly to HTML and PDF using the `html` or `pdf` build targets. Additionally the build targets `spelling` and `linkcheck` are available to run a spell checker on the documentation and validate external links.

All public classes and functions should have Doxygen documentation as demonstrated below:

```
/**
 * @brief Brief description
 *
 * Detailed description if needed
 *
 * @param param1 Description of parameter
 * @return Description of return value
 */
int myFunction(int param1);
```

Building

See *Installation* for detailed build instructions. For development:

```
# Debug build with Qt6
cmake -S . -B build -DCMAKE_BUILD_TYPE=Debug \
      -DLAMMPS_GUI_USE_PLUGIN=ON -DBUILD_DOC=OFF
cmake --build build --parallel 2
```

Before submitting a pull request, run `clang-format` on any modified source files so they conform to the project's `.clang-format` configuration:

```
clang-format -i src/*.cpp src/*.h
```

When adding new `.cpp` or `.h` files to `src/`, also add them to the `PROJECT_SOURCES` list in `cmake/Sources.cmake` so they are picked up by the build (Qt's `AUTOMOC` handles `moc` generation automatically once the file is listed). The top-level `CMakeLists.txt` contains only the configuration options and the executable target; the remaining build logic is organized into include files in the `cmake/` folder.

All documentation should be written in American English using plain ASCII characters (no typographic quotes, em-dashes written as `--`, etc.).

Adding or modifying a color map

The dump-image color maps shown in the *Atoms/bonds settings dialog* are defined once, as a table in `src/colormaps.cpp`. A single `ColorMapDef` – a `continuous` flag plus a list of `ColorMapStop` entries (each a position in `[0,1]` and either a LAMMPS named color or an explicit RGB triple in `[0,1]`) – drives both the `dump_modify` command emitted by `appendColorMapArgs()` (in `src/dumpimage.cpp`) and the preview swatch built by `addColorMapItems()` (in `src/imageviewersettings.cpp`), so the two cannot disagree.

To add a color map:

1. Add an entry to the `maps` table in `src/colormaps.cpp`. Use `rc(pos, r, g, b)` for an explicit RGB stop (components in `[0,1]`) or `nm(pos, "name")` for a LAMMPS named color. Set the `ColorMapDef` flag to `true` for an interpolated (continuous) map or `false` for a discrete sequence. Use the same 0..1 floating-point values that LAMMPS renders so the preview matches exactly.
2. Add the map's name to `colorMapNames()` in the same file, at the position where it should appear in the selection menu.
3. Regenerate the documentation preview image and add the new map to the color-map list in `doc/visualization.rst`:

   ```
   python3 doc/colormaps_preview.py
   ```

4. Optionally extend `test/test_dumpimage.cpp` to pin the new map's stops.

Modifying an existing map is just editing its stops in the table; nothing else needs to change. Canonical stops for a matplotlib color map can be resampled with a few lines of Python:

```
import matplotlib.cm as cm
m = cm.get_cmap("cividis")
n = 5  # number of stops
print([tuple(round(c, 3) for c in m(i / (n - 1))[:3]) for i in range(n)])
```

Adding or updating a tutorial collection

The *Tutorials menu* is driven by a single table of `TutorialCollection` entries in `src/tutorials.cpp`. Each collection is built by a small factory function (`molecular()`, `matsci()`, `granular()`) and registered in the `collections()` list; the menu and the `TutorialWizard` read everything they need from that table, so adding or updating a tutorial is a data change in one file – no menu or wizard code has to be touched.

How the files are hosted

Each collection's input and solution files live in their own public (e.g. GitHub) repository, laid out as one `tutorial<N>/` folder per tutorial. The `filesUrl` field is a two-argument download pattern where `%1` is the tutorial number and `%2` is the file name, for example `.../matsci-tutorials-inputs/main/tutorial%1/%2`.

Every `tutorial<N>/` folder contains a `.manifest` text file that lists the files to download, one per line (`#` comments and blank lines are ignored):

- The **first** plain file name is the initial template that is auto-loaded into the editor when the tutorial starts.
- Other plain file names are support files downloaded alongside it.
- Lines under `solution/` are downloaded only when the user requests the solution files.

Large data files that are shared between the tutorial folder and its `solution/` subfolder do not need to be duplicated: store the copy in `solution/` as a symbolic link to the parent-directory file. The raw file server then serves that link as a one-line text file containing the link target (e.g. `../Al_zhou.eam.alloy`); `downloadTutorialFiles()` detects a `../<same-name>` payload and copies the already-downloaded parent file in place of the placeholder.

A manifest entry that is missing on the server (e.g. after a file was renamed without updating the `.manifest`) does not abort the download: the remaining files are still fetched and a dialog then lists the missing files, with a button that opens the repository's issue tracker (`filesRepoUrl + /issues`) so users can report them. Only a failure to download the `.manifest` itself or a cancellation by the user stops the setup.

Incremental rollout

Collections are released tutorial by tutorial. Three fields on `TutorialCollection` control how an unfinished collection appears:

- `available` – the number of leading tutorials that are launchable now. Entries at or beyond this index are shown in the menu but disabled, acting as a preview of what is coming.
- `published` – when `false`, the submenu title gets the `status` text appended as a suffix.
- `status` – that suffix text, e.g. `"coming soon"` or `"planned"`.

A collection with no tutorials defined yet (`count() == 0`) is shown as a single disabled submenu.

Common changes

1. **Enable the next tutorial in a collection that is rolling out:** bump `c.available` in that collection's factory function once the tutorial's text and files are ready.
2. **Update a tutorial's menu/wizard text:** edit the corresponding entry in the `titles`, `slugs`, and `blurbs` lists (all parallel, one entry per tutorial) in the factory function.
3. **Add a whole new collection:** add a factory function that fills in `key`, `name`, `dirPrefix`, `author`, `filesUrl`, `filesRepoUrl`, `webUrl`/`siteUrl`, `logo`, the `titles` / `slugs` / `blurbs` lists, and the rollout fields, then append it to the `collections()` list. Host the files in the `tutorial<N>/` + `.manifest` layout described above.
4. **Fully publish a collection:** set `available` to the tutorial count and `published = true` so the teaser suffix and the disabled entries go away.

Contributing

To contribute to LAMMPS-GUI:

1. Fork the repository on GitHub
2. Create a feature branch
3. Make your changes with proper documentation
4. Ensure code compiles with Qt 6.2 or later
5. Test your changes thoroughly
6. Submit a pull request

All contributions must:

- Follow the existing code style and pass `clang-format`
- Include Doxygen documentation for new public APIs
- Register new source files in `PROJECT_SOURCES` in `cmake/Sources.cmake`
- Not break existing functionality
- Have GPG-signed commits

5.5 Testing

The `test` directory contains some tests for the LAMMPS-GUI project using either the GoogleTest framework or the Python unittest framework.

Overview

The test suite uses CMake's CTest front end to select and run the tests. Tests implemented with GoogleTest are automatically discovered and can be run individually or as a complete suite for each test program. Tests running LAMMPS-GUI itself use the "virtual frame buffer" X server (Xvfb) and are written in Python using the `unittest` Python module and the PyAutoGUI module

Building the Tests

Tests are built as part of the main project build when using `-D ENABLE_TESTING=ON` during CMake configuration (default setting is *OFF*). Due to technical requirements, testing is currently only enabled for native Linux builds of LAMMPS-GUI. In any other build environment, the `-D ENABLE_TESTING=ON` setting is ignored.

Quick Build

For running the tests, it is not necessary to build the documentation, so its build can be skipped during configuration.

```
cmake -S . -B build -D LAMMPS_GUI_USE_PLUGIN=ON -D BUILD_DOC=OFF -D ENABLE_TESTING=ON
cmake --build build --parallel 2
```

Disable Tests

Tests are disabled by default. If they have been enabled during CMake configuration they can be disabled at a later point with:

```
cmake -S . -B build -D ENABLE_TESTING=OFF
```

Running Tests

Below are some frequently used command line examples for running tests. These examples assume that LAMMPS-GUI was compiled in the folder `build`. CTest can also be invoked from the top of the build tree (`--test-dir build`); both forms work and select the same set of tests.

Run All Tests

```
ctest --test-dir build/test
```

Run Tests with Verbose Output

```
ctest --test-dir build/test -V
```

List Available Tests

The list of the names of all available tests can be obtained with:

```
ctest --test-dir build/test -N
```

Run Specific Tests

Individual tests can be selected in different ways. Most common is the use of regular expressions to select (`-R`) or exclude (`-E`) tests. It is also possible to select tests by a range of Test numbers (`-I`) from the -N test list output. Examples:

```
ctest --test-dir build/test -R HelpersTest
ctest --test-dir build/test -E Frame
ctest --test-dir build/test -I 20,25
```

Current Test Coverage

The unit tests cover the utility functions, the widget-scoped keyboard shortcuts, the stdout capture, the log window warning highlighter, the dump-image command builder, the movie import and image cache of the Slide Show window, the docked window layout, the Tab completion of the command window's prompt, the plot data model with its file parsers and writers, the block-structured `fix ave/*` file parsers and their reduction to a plottable table, the column picker dialog, the chart axis-layout math, and the Qt-free math toolkit (least squares and smoothing, autocorrelation, curve fitting, the Levenberg-Marquardt solver, the vendored LeptonMini expression parser, and the custom-function layer on top of them). Command-line tests validate basic executable behavior, and PyAutoGUI-based tests exercise the GUI itself. Future expansion will include more GUI component testing and integration tests.

Test Organization

Tests are organized into three main categories:

1. **Unit Tests**: Using GoogleTest framework to test individual functions
2. **Command-Line Tests**: Using command-line to validate basic executable behavior
3. **GUI Tests**: Tests using the Python unittest framework and PyAutoGUI to run LAMMPS-GUI inside a virtual frame buffer in a "remote controlled fashion".

Unit Tests

test_helpers.cpp

Comprehensive tests for functions in `src/helpers.h` and `src/helpers.cpp`. This module contains test cases covering the utility functions used throughout the application.

Date Comparison (dateCompare)
Tests for the dateCompare function that compares version date strings in LAMMPS date format (e.g., "22 Jul 2025"):

- Same dates (returns 0)
- Different years (returns positive/negative)
- Different months (returns positive/negative)
- Different days (returns positive/negative)
- Full month names vs. abbreviations
- Invalid date formats
- Edge cases (year boundaries, month boundaries)
- December month ordering
- Both dates invalid
- Dates with "- Update" suffix

Line Splitting (splitLine)
Tests for the splitLine function that parses command-line style input with proper quote handling:

- Simple whitespace-separated tokens
- Single-quoted strings
- Double-quoted strings
- Escaped quotes within strings
- Mixed quoting styles
- Triple-nested quotes
- Multiple consecutive whitespace characters
- Empty input
- Quotes at string boundaries
- Single word without spaces
- Hash comment handling
- Special characters

Executable Detection (hasExe)
Tests for the hasExe function that checks if an executable exists in PATH:

- Common system commands (sh, ls on Unix; cmd on Windows)
- Non-existent commands
- Empty string input
- bash executable

- Platform-specific behavior (conditional compilation)

Theme Detection (isLightTheme)
Tests for the isLightTheme function that determines if the current Qt theme is light or dark:

- Boolean return value validation
- Consistency across calls
- No crashes on theme query

Stdout Silencing (silenceStdout/restoreStdout)
Tests for the stdout silencing functions that redirect stdout to suppress LAMMPS library output:

- Silencing actually suppresses stdout
- Restoring when not silenced is a no-op
- Nested silencing and restoring
- Silencing skipped during active StdCapture
- Capture restores silenced stdout state
- Silence and restore preserves stdout

Directory Purging (purgeDirectory)
Tests for the purgeDirectory function that recursively removes directory contents:

- Purging directory with files
- Non-existent directory (no crash)
- Empty directory

Image File Detection (isImageFile)
Test for the extension-based check that decides whether a file is loaded as an image by the Slide Show window.

Grayscale Conversion (grayscaleImage)
Tests for the helper that renders an image in grayscale, used for the "inactive" state of icons:

- Color is removed while the alpha channel is preserved
- Contrast is faded towards the midpoint

Qt Message Silencing (QtMessageSilencer)
Tests for the `QtMessageSilencer` RAII guard that collects Qt warning messages instead of printing them to the console:

- Emitted warnings are collected instead of printed
- Nothing is collected when nothing is emitted
- The previous message handler is restored and guards can be nested

Viewer Window Fit (viewerFitSize)
Tests for the pure window-fit computation used by the Image Viewer and Slide Show window auto-resize:

- Content within the screen budget is fitted exactly (content plus frame)
- An exact fit adds no scroll bar allowance
- Width or height overflow clamps that axis and adds scroll bar room on the other axis
- Overflow in both directions clamps to the budget
- The added scroll bar room never exceeds the budget
- A negative budget clamps to zero

test_shortcuts.cpp

Tests for the `addShortcut()` and `scopeShortcut()` helpers in `src/helpers.{h,cpp}`. The output windows repeat several main window accelerators, which is harmless while each is a window of its own but becomes an ambiguous binding once they are docked into the main window, so both helpers scope a shortcut to its owning widget. Test cases cover:

- A shortcut added with `addShortcut()` firing only while its widget has the focus
- `scopeShortcut()` binding an action's shortcut to its widget and tolerating null arguments
- Scoped shortcuts on sibling widgets not colliding with each other
- A standalone window keeping the shortcuts the main window also binds

test_windowlayout.cpp

Tests for the *`WindowLayout`* mediator in `src/windowlayout.{h,cpp}`, which docks the output views into the main window in the combined layout. Test cases cover:

- Dock title bars reporting a valid size hint
- Closing a docked view closing its dock, closing a transient view removing its dock, and a floating view being left alone
- The saved dock arrangement being keyed by the Qt version, with the legacy key dropped when saving
- Closing a panel keeping the proportions of the remaining ones
- A transient view surviving its dock being torn down first
- A docked view keeping its monospace document font when it inherits the main window's font

test_shellprompt.cpp

Tests for the *`ShellPrompt`* input line of the command window in `src/shellprompt.{h,cpp}`, driven together with its `QCompleter` the way the window manager would. Test cases cover:

- Tab keeping the focus, offering the matches, taking a single match without a list, and walking the matches forward and (with Shift-Tab) backward
- Enter taking a chosen match without running it and running the line when nothing is chosen or no list is up
- The completion request being announced before matching, so the list to complete from can be chosen for the line
- Tab on an empty line, on a read-only prompt, and with a modifier held offering nothing

test_stdcapture.cpp

Tests for the *`StdCapture`* class in `src/stdcapture.{h,cpp}`, which redirects the C-level stdout file descriptor through a pipe so that LAMMPS library output can be collected and displayed in the *Output* window. Test cases cover:

- Capturing simple single-line output
- Capturing multiple lines of output
- Behavior with empty output
- Re-using a single `StdCapture` instance for several capture cycles
- `getChunk()` returning incremental output while a capture is active
- `getChunk()` returning an empty string when not capturing

- Multiple successive `getChunk()` calls during one capture
- `endCapture()` being a safe no-op when no capture is active
- `getBufferUse()` reporting zero before any capture activity
- `getBufferUse()` reflecting the size of the most recent chunk
- The original `stdout` file descriptor being restored when capture ends

test_flagwarnings.cpp

Tests for the *`FlagWarnings`* syntax highlighter used in the *Output* window to flag lines beginning with `WARNING` or `ERROR` and to update a running count displayed in the summary label. Test cases cover:

- Constructor initializes the warning count to zero
- `WARNING` and `ERROR` lines are correctly detected
- Normal output lines are not flagged
- The summary `QLabel` is updated when warnings appear
- Empty documents produce no spurious warnings
- Warnings are detected when they appear at the start of a line
- The running count is correct for multiple warnings

test_dumpimage.cpp

Tests for the GUI-free dump-image command builder (`src/dumpimage.{h,cpp}`) that assembles the LAMMPS `write_dump ... image` command from a `DumpImageParams` snapshot of the Image Viewer state. Test cases cover:

- Basic structure of the generated render and `dump_modify` arguments
- Pruning of all settings that match the LAMMPS defaults (color tables, transparency, background, up direction, sub-box style, axes center)
- The default BWR color map, named and reversed continuous maps, discrete bond color maps, and the canonical stops of the perceptual maps
- Element-based vs. type-based coloring (no color map for type coloring)
- Bond rendering: suppressed while VDW spheres are active, drawn when requested, auto-bond generation only when a pair style is defined
- `noinit` suppressed while a fix is active; disabled fixes are ignored
- Region outline points and bond coloring by computed values

test_movieimport.cpp

Tests for the parsing and frame-counting helpers of the movie import (`src/movieimport.{h,cpp}`). These are free functions, so the tests run without `ffprobe` or `ffmpeg` installed. Test cases cover:

- `parseFrameRate()`: rational (`30000/1001`) and plain numbers, invalid input
- `selectedFrameCount()`: full range, range with interval, invalid ranges
- `parseProbeOutput()`: frame count taken from the container, missing frame count or frame size, absent video stream, malformed JSON, and numeric fields given as JSON numbers instead of strings

test_imagecache.cpp

Tests for the *ImageCache* class (`src/imagecache.{h,cpp}`), the temporary-directory backed store for converted images and extracted movie frames owned by the Slide Show window. Test cases cover:

- Qt-readable formats are loaded directly and never cached
- Exotic formats are converted at most once; changed source files are converted again
- Missing files return a null image; unreadable files are reported once and not retried until the file changes
- Usage totals track the converted images
- `forget()` drops the conversion of a deleted file; purging conversions keeps extracted movie frames and failure records
- Cache subdirectories are unique and sanitized
- `clear()` and the destructor remove the temporary directory

test_plotdata.cpp

Tests for the column-oriented `PlotData` model and the parsers and writers for external data files (`src/plotdata.{h,cpp}`). Test cases cover:

- Appending rows and columns to the model
- CSV import with and without a header line
- Whitespace-separated (`.dat`) import with a LAMMPS-style header
- LAMMPS YAML thermo output, including trailing commas, interleaved log lines, and a sequence of maps
- JSON import as array-of-rows and object-of-arrays, with error handling for unequal columns and malformed input
- Dispatch by file extension and content-based YAML detection in log files
- CSV, `.dat`, and YAML export round-trips, including YAML quoting rules

test_plotblockdata.cpp

Tests for the parsers of the block-structured output files written by the `fix ave/*` styles and for their reduction to a flat table (`src/plotblockdata.{h,cpp}`). Test cases cover:

- The native format of `fix ave/time` in vector mode, `fix ave/histo`, `fix ave/correlate`, and the comment-delimited blocks of `fix ave/correlate/long`, including the column and per-block scalar names taken from the default header comments
- A single value in vector mode, whose block header is exactly as wide as its data rows
- `overwrite` (a single block), a header written again mid-file by a second fix instance, a last block cut short by an interrupted run, and blocks whose row count changes
- Format recognition from the block structure alone, for files whose headers were replaced with `title1`/`title2`/`title3`
- `fix ave/chunk`: a one-dimensional profile and a compressed-chunk-ID file getting the coordinate as their x axis, a cylindrical binning with a single axial bin recognized as one-dimensional from its values, and a real two-dimensional grid and non-binned chunks keeping the generic defaults
- Rejecting flat tables, including a `fix ave/time` scalar file whose header comment reads like a block file's
- The vector-mode YAML variant, with its synthesized row index, and the scalar-mode YAML shape being left to the flat parser

- Single-block and block-range reduction with hand-computed means, standard deviations, standard errors, and min/max spreads (whose bars are asymmetric and end exactly on the extreme values), clamped and reversed ranges, and blocks dropped for having a different shape
- The per-format import defaults, including a running average being recognized so that its blocks are not averaged

test_plotdatadialog.cpp

Tests for the *PlotDataDialog* column picker in `src/plotdatadialog.{h,cpp}`, run with the `offscreen` platform plugin. Test cases cover:

- Derived columns referencing columns as `{name}`, including names with brackets, and a bare name being rejected
- Renaming a column rewriting the references in already added derived columns, and unusable names being refused
- The block reduction keeping renames and derived columns
- `PlotDataDialog::fromFile()` picking the block or the flat parser for a file and reporting a file that neither can read

test_plotaxismath.cpp

Tests for the Qt-free chart axis-layout helpers (`src/plotaxismath.{h,cpp}`). Test cases cover:

- `niceTickInterval()`: 1-2-5-10 interval selection, scaling across powers of ten, degenerate ranges, and non-positive tick targets
- `tickValues()`: even spacing, anchor alignment, snapping zero, endpoint inclusion despite floating-point rounding, reversed ranges, custom anchors
- `tickDecimals()`: decimal counts for integer and fractional spacings
- `formatAxisLabel()`: printf-style integer and floating-point specifiers, length-modifier normalization, literal prefix and suffix text, and fallback behavior for empty or placeholder-free formats

test_leastsquares.cpp

Tests for the dense linear-algebra and smoothing routines (`src/leastsquares.{h,cpp}`). Test cases cover:

- Matrix transpose, multiplication, and inversion
- LU linear solve with single and multi-column right-hand sides
- Savitzky-Golay smoothing: moving-average behavior for constant data, exact preservation of linear and quadratic data at matching polynomial degrees, and noise reduction around a line

test_analysis.cpp

Tests for the post-processing analyses (`src/analysis.{h,cpp}`). Test cases cover the normalized autocorrelation function: an exact small case, lag zero being one, empty results for constant or too-short series, clamping of the maximum lag, and anticorrelation of an alternating series.

test_fitting.cpp

Tests for the linear-least-squares curve fits (`src/fitting.{h,cpp}`). Test cases cover recovering known polynomial models and evaluating the fitted polynomial, recovering a known Birch-Murnaghan equation-of-state model, and the failure paths for too few data points and non-positive volumes.

test_levmar.cpp

Tests for the Levenberg-Marquardt nonlinear least-squares solver (`src/levmar.{h,cpp}`). Test cases cover recovering linear, exponential-decay, and Gaussian models, fitting noisy data, rejecting underdetermined problems (more parameters than residuals), and reporting a failing initial model evaluation as an error instead of crashing.

test_lepton.cpp

Tests for the vendored LeptonMini expression parser (`thirdparty/lepton_mini`). Test cases cover expression evaluation, error handling for invalid expressions, verification of symbolic derivatives, custom functions, and expression optimization.

test_customfunc.cpp

Tests for the custom-function evaluation and fitting layer (`src/customfunc.{h,cpp}`) built on LeptonMini. Test cases cover:

- Evaluating user expressions (polynomials, trigonometric functions, constants, custom variables) over a sample range
- Skipping non-finite points and clamping the sample count
- Error handling for empty expressions, syntax errors, and undefined variables
- Nonlinear custom fits recovering exponential-decay and quadratic models
- Fit-setup validation: variable/parameter clashes, duplicate parameters, undeclared symbols, and too few data points

Command-Line Tests

These tests validate the `lammps-gui` executable behavior without starting the full GUI. They run quickly and are useful for CI/CD pipelines.

CommandLine.GetVersion

Purpose: Verify version reporting consistency

This test runs:

```
lammps-gui --platform offscreen -v
```

and validates that:

- The executable launches successfully
- Version output includes "LAMMPS-GUI (QT6)"
- Version number matches the `PROJECT_VERSION` CMake variable
- Process exits cleanly with status 0

Environment: `OMP_NUM_THREADS=1` to ensure consistent behavior

CommandLine.HasPlugin

Purpose: Verify build configuration is reflected in help text

This test runs:

```
lammps-gui --platform offscreen -h
```

and validates that help text is consistent with CMake configuration:

- **Plugin Mode** (`LAMMPS_GUI_USE_PLUGIN=ON`): Help text includes "-p, –pluginpath <path>" option
- **Linked Mode** (`LAMMPS_GUI_USE_PLUGIN=OFF`): Help text omits plugin path option

Environment: `OMP_NUM_THREADS=1` to ensure consistent behavior

CommandLine.NoDisplayHelp

Purpose: Verify that the help text is printed without a display

This test runs:

```
lammps-gui -h
```

with the `DISPLAY`, `WAYLAND_DISPLAY`, and `QT_QPA_PLATFORM` environment variables set to empty values, and validates that the help text is printed instead of Qt aborting for lack of a platform plugin.

CommandLine.NoDisplayStart

Purpose: Verify that a start without a display fails cleanly

This test runs:

```
lammps-gui
```

in the same environment as `CommandLine.NoDisplayHelp` and validates that LAMMPS-GUI exits with an error message stating that no graphical display was found, instead of aborting with a core dump.

GUI Tests

These tests validate LAMMPS-GUI functionality using PyAutoGUI and Xvfb (virtual frame buffer). They run the actual GUI application in a headless X server environment, allowing automated interaction and screenshot capture.

Important Note: The argument for the screen number flag `-n` for `xvfb-run` *must* be different for each test, so that the tests may run in parallel.

Framebuffer.CreateScreenshot (test_shooter.py)

Purpose: Test the screenshot wrapper utility that abstracts different screenshooter applications

Test File: `test/test_shooter.py`

This test validates the `shooter` wrapper script that provides a unified interface to various Linux screenshot utilities (ImageMagick's `import`, `magick import`, `xfce4-screenshooter`, `gnome-screenshot`).

The test runs:

```
xvfb-run -n 11 -s "-screen 0 1024x768x24" -w 1 python test_shooter.py
```

within a virtual frame buffer and validates:

ScreenshotChecks.testCreateImage

- The `shooter` command executes without errors
- A PNG file is created at the specified path

- The image dimensions match the virtual frame buffer size (1024x768)
- The image format is PNG
- The screenshot captures an all-black screen (expected for empty Xvfb)
- Specific pixel values at multiple locations are (0,0,0) RGB

Dependencies:

- PyAutoGUI - for screen size detection
- Pillow (PIL) - for image file validation
- One of: ImageMagick (`import` or `magick`), `xfce4-screenshooter`, or `gnome-screenshot`

Setup/Teardown:

- `setUp()`: Removes leftover `shot.png` from previous runs
- `tearDown()`: Cleans up `shot.png` after test completion

Environment: Virtual frame buffer at 1024x768x24, `PYTHONUNBUFFERED=1`, `PYTHONDONTWRITEBYTECODE=1`, `OMP_NUM_THREADS=1`

Framebuffer.CheckSize (test_xvfbsize.py)

Purpose: Verify PyAutoGUI functionality and Xvfb screen size configuration

Test File: `test/test_xvfbsize.py`

This test validates that PyAutoGUI can properly interact with the virtual frame buffer created by Xvfb, which is essential for GUI automation tests.

The test runs:

```
xvfb-run -n 12 -s "-screen 0 1024x768x24" -w 1 python test_xvfbsize.py
```

within a virtual frame buffer and validates:

PyAutoGUIChecks.testScreenSize

- PyAutoGUI correctly detects the screen dimensions
- Screen width is 1024 pixels
- Screen height is 768 pixels

PyAutoGUIChecks.testMousePosition

- PyAutoGUI can detect the mouse cursor position
- Initial mouse position is at screen center (512, 384)
- `pyautogui.moveTo()` can move cursor to absolute positions
- `pyautogui.moveRel()` can move cursor by relative offsets
- Position queries return expected coordinates after moves

Dependencies:

- PyAutoGUI - for screen size detection and mouse control

Environment: Virtual frame buffer at 1024x768x24, `PYTHONUNBUFFERED=1`, `PYTHONDONTWRITEBYTECODE=1`, `OMP_NUM_THREADS=1`

Framebuffer.GUIEditorChecks (test_gui_edit.py)

Purpose: Test basic LAMMPS-GUI editor functionality using automated GUI interactions

Test File: `test/test_gui_edit.py`

This test validates fundamental editor operations in LAMMPS-GUI by launching the application inside a virtual frame buffer and automating user interactions with PyAutoGUI. The test uses screenshot comparison to verify visual state.

The test runs:

```
xvfb-run -n 13 -s "-screen 0 1024x768x24" -w 1 python test_gui_edit.py
```

within a virtual frame buffer and validates:

GUIEditorChecks.testExitShortcut

- LAMMPS-GUI launches and displays a white editor background
- The `Ctrl-Q` keyboard shortcut exits the application cleanly
- The process exits with status 0
- Screenshots confirm the window was displayed and then closed

GUIEditorChecks.testExitMenu

- The `Alt-F` menu shortcut opens the File menu
- The `Q` key selects the Quit entry
- The application exits cleanly with status 0
- Screenshots confirm expected visual state

GUIEditorChecks.testExitModCancelNo

- Text can be typed into the editor buffer
- Exiting with a modified buffer shows a confirmation dialog
- The `Cancel` option returns to the editor without exiting
- The `No` option exits without saving

Dependencies:

- PyAutoGUI - for keyboard and mouse automation
- Pillow (PIL) - for screenshot validation
- A supported screenshooter application

Setup/Teardown:

- `setUp()`: Launches LAMMPS-GUI, cleans up leftover test files
- `tearDown()`: Terminates the LAMMPS-GUI process

Environment: Virtual frame buffer at 1024x768x24, `PYTHONUNBUFFERED=1`, `PYTHONDONTWRITEBYTECODE=1`, `OMP_NUM_THREADS=1`, `LAMMPS_GUI=<path to executable>`

Test Fixtures and Utilities

HelpersTest Fixture
Base test fixture that creates a `QCoreApplication` instance for tests that require Qt functionality. The application is created once per test suite and reused across tests for efficiency. Similar lightweight fixtures (`StdCaptureTest`, `FlagWarningsTest`) are used by the matching unit test programs.

Platform-Specific Testing
Tests use conditional compilation (`#ifdef _WIN32`) to adapt to platform differences in:

- Path separators
- Line endings
- Available system executables
- Default shell commands

Future Test Expansion

Planned additions to the test suite include:

GUI Component Tests

- CodeEditor text manipulation
- Syntax highlighter accuracy
- Find/replace functionality
- Auto-completion behavior

LAMMPS Integration Tests

- LammpsWrapper command execution
- Variable substitution
- Error handling
- Output capture

File I/O Tests

- File opening/saving
- Recent files management
- Auto-save functionality
- Data file inspection

Preferences Tests

- Settings persistence
- Default value initialization
- Migration between versions

Tutorial Tests

- Tutorial file generation
- Directory setup
- Resource extraction

Adding Tests

Create a New Test File

1. Create a new test file in the `test/` directory (e.g., `test_newfile.cpp`)
2. Add the test executable to `test/CMakeLists.txt`:

```
add_executable(test_newfile
  test_newfile.cpp
  ${CMAKE_SOURCE_DIR}/src/newfile.cpp
)

target_include_directories(test_newfile PRIVATE
  ${CMAKE_SOURCE_DIR}/src
)

target_link_libraries(test_newfile
  GTest::gtest_main
  Qt6::Widgets
)

gtest_discover_tests(test_newfile)
```

Add Tests to Existing File

Add new test cases using GoogleTest macros:

```
TEST_F(HelpersTest, NewTestName)
{
    // Arrange
    std::string input = "test data";

    // Act
    auto result = function_to_test(input);

    // Assert
    EXPECT_EQ(result, expected_value);
}
```

Dependencies

- **GoogleTest**: Automatically fetched via CMake FetchContent (v1.17.0)
- **Qt6**: Required for Qt-dependent functions (Widgets component)
- **CTest**: Part of CMake, used for test execution

Notes

- Tests that require a Qt application context use a `HelpersTest` fixture that creates a `QCoreApplication` instance.
- Platform-specific tests (e.g., `has_exe`) use conditional compilation to test appropriate commands on different operating systems.
- The test suite is designed to be easily extended with additional test files and test cases.

- GoogleTest is fetched automatically during CMake configuration, so no manual installation is required.

CI Integration

The test suite integrates with existing CI workflows: - Tests run as part of the standard build process when `ENABLE_TESTING=ON` - CTest provides standard output for CI systems - Tests can be disabled for documentation-only builds

---

Index

A

`About menu`, 54
`AboutDialog` (*C++ class*), 146
`AboutDialog::~AboutDialog` (*C++ function*), 146
`AboutDialog::AboutDialog` (*C++ function*), 146
`AboutDialog::operator=` (*C++ function*), 146
`AboutDialog::showEvent` (*C++ function*), 147
`accelerators`, 57
`AcceleratorTab` (*C++ class*), 144
`AcceleratorTab::AcceleratorTab` (*C++ function*), 145
`AcceleratorTab::AccelPrec` (*C++ enum*), 144
`AcceleratorTab::AccelPrec::Double` (*C++ enumerator*), 144
`AcceleratorTab::AccelPrec::Mixed` (*C++ enumerator*), 144
`AcceleratorTab::AccelPrec::Single` (*C++ enumerator*), 144
`AcceleratorTab::AccelType` (*C++ enum*), 144
`AcceleratorTab::AccelType::Gpu` (*C++ enumerator*), 144
`AcceleratorTab::AccelType::Intel` (*C++ enumerator*), 144
`AcceleratorTab::AccelType::Kokkos` (*C++ enumerator*), 144
`AcceleratorTab::AccelType::None` (*C++ enumerator*), 144
`AcceleratorTab::AccelType::OpenMP` (*C++ enumerator*), 144
`AcceleratorTab::AccelType::Opt` (*C++ enumerator*), 144
`addMenuAction` (*C++ function*), 199
`addShortcut` (*C++ function*), 200
`animation`, 43
`applyWindowFlags` (*C++ function*), 197
`ArgRole` (*C++ enum*), 89
`ArgRole::Any` (*C++ enumerator*), 90
`ArgRole::DefineId` (*C++ enumerator*), 90
`ArgRole::File` (*C++ enumerator*), 90
`ArgRole::GroupId` (*C++ enumerator*), 90
`ArgRole::Int` (*C++ enumerator*), 90
`ArgRole::Keyword` (*C++ enumerator*), 90
`ArgRole::Label` (*C++ enumerator*), 90
`ArgRole::Number` (*C++ enumerator*), 90
`ArgRole::Style` (*C++ enumerator*), 90
`ArgRole::SubStyle` (*C++ enumerator*), 90
`ArgSpec` (*C++ struct*), 91
`ArgSpec::cat` (*C++ member*), 92
`ArgSpec::role` (*C++ member*), 92
`argumentTexts` (*C++ function*), 87
`atom settings`, 37
`auto-completion`, 47

auto-save, 47
autocorrelation, 21
autocorrelation (*C++ function*), 180
AveFileKind (*C++ enum*), 176
AveFileKind::AveChunk (*C++ enumerator*), 176
AveFileKind::AveCorrelate (*C++ enumerator*), 176
AveFileKind::AveCorrelateLong (*C++ enumerator*), 176
AveFileKind::AveHisto (*C++ enumerator*), 176
AveFileKind::AveTimeVector (*C++ enumerator*), 176
AveFileKind::Unknown (*C++ enumerator*), 176
aveFileKindName (*C++ function*), 174
aveImportDefaults (*C++ function*), 176
averageBlocks (*C++ function*), 175

B

basic usage, 13
Birch-Murnaghan EOS, 21
birchMurnaghanFit (*C++ function*), 182
block-structured data, 24
BlockErrorType (*C++ enum*), 176
BlockErrorType::MinMax (*C++ enumerator*), 177
BlockErrorType::None (*C++ enumerator*), 177
BlockErrorType::StdDev (*C++ enumerator*), 177
BlockErrorType::StdError (*C++ enumerator*), 177
blockErrorTypeName (*C++ function*), 175
bond settings, 37
buildDumpImageCommand (*C++ function*), 126

C

chart style, 21
ChartColumn (*C++ struct*), 105
ChartColumn::custom (*C++ member*), 107
ChartColumn::dispmode (*C++ member*), 107
ChartColumn::doRaw (*C++ member*), 107
ChartColumn::doSmooth (*C++ member*), 107
ChartColumn::errColor (*C++ member*), 107
ChartColumn::errWidth (*C++ member*), 107
ChartColumn::fit (*C++ member*), 106
ChartColumn::index (*C++ member*), 106
ChartColumn::lastUpdate (*C++ member*), 107
ChartColumn::lastX (*C++ member*), 106
ChartColumn::order (*C++ member*), 106
ChartColumn::overlaySeries (*C++ member*), 107
ChartColumn::procLabel (*C++ member*), 108
ChartColumn::rawColor (*C++ member*), 107
ChartColumn::rawPointSize (*C++ member*), 107
ChartColumn::rawWidth (*C++ member*), 107
ChartColumn::rawXmax (*C++ member*), 106
ChartColumn::rawXmin (*C++ member*), 106
ChartColumn::rawYmax (*C++ member*), 106
ChartColumn::rawYmin (*C++ member*), 106
ChartColumn::reflineDefs (*C++ member*), 108
ChartColumn::scatter (*C++ member*), 106
ChartColumn::series (*C++ member*), 106

ChartColumn::smooth (*C++ member*), 106
ChartColumn::smoothcolor (*C++ member*), 107
ChartColumn::smoothmode (*C++ member*), 107
ChartColumn::smoothpointsize (*C++ member*), 107
ChartColumn::smoothScatter (*C++ member*), 106
ChartColumn::smoothwidth (*C++ member*), 107
ChartColumn::vlines (*C++ member*), 108
ChartColumn::window (*C++ member*), 106
ChartColumn::yTitle (*C++ member*), 108
chartPaletteColor (*C++ function*), 189
charts settings, 58
charts window, 19
ChartsTab (*C++ class*), 146
ChartsTab::ChartsTab (*C++ function*), 146
ChartViewer (*C++ class*), 108
ChartViewer::~ChartViewer (*C++ function*), 108
ChartViewer::addOverlaySeries (*C++ function*), 110
ChartViewer::addPoint (*C++ function*), 109
ChartViewer::ChartViewer (*C++ function*), 108
ChartViewer::clearFitCurve (*C++ function*), 112
ChartViewer::displayColor (*C++ function*), 111
ChartViewer::displayMode (*C++ function*), 111
ChartViewer::displayPointSize (*C++ function*), 111
ChartViewer::displayWidth (*C++ function*), 111
ChartViewer::errorColor (*C++ function*), 112
ChartViewer::errorWidth (*C++ function*), 112
ChartViewer::getCount (*C++ function*), 109
ChartViewer::getData (*C++ function*), 110
ChartViewer::getError (*C++ function*), 110
ChartViewer::getMinMax (*C++ function*), 109
ChartViewer::getName (*C++ function*), 109
ChartViewer::getStep (*C++ function*), 109
ChartViewer::getXLabel (*C++ function*), 112
ChartViewer::getYLabel (*C++ function*), 112
ChartViewer::hasCustom (*C++ function*), 112
ChartViewer::operator= (*C++ function*), 108
ChartViewer::overlaySeriesCount (*C++ function*), 111
ChartViewer::resetZoom (*C++ function*), 109
ChartViewer::setColumn (*C++ function*), 108
ChartViewer::setDisplayStyle (*C++ function*), 111
ChartViewer::setErrorStyle (*C++ function*), 112
ChartViewer::setFitCurve (*C++ function*), 112
ChartViewer::setLegendPos (*C++ function*), 111
ChartViewer::setReferenceLines (*C++ function*), 111
ChartViewer::setRefLabelStyle (*C++ function*), 111
ChartViewer::setSmoothStyle (*C++ function*), 111
ChartViewer::setTLabel (*C++ function*), 110
ChartViewer::setXAxisRange (*C++ function*), 109
ChartViewer::setXLabel (*C++ function*), 110
ChartViewer::setXLabelFormat (*C++ function*), 110
ChartViewer::setYAxisRange (*C++ function*), 109
ChartViewer::setYLabel (*C++ function*), 110
ChartViewer::smoothColor (*C++ function*), 112
ChartViewer::smoothMode (*C++ function*), 112

ChartViewer::smoothPointSize (*C++ function*), 112
ChartViewer::smoothWidth (*C++ function*), 112
ChartViewer::updateSmooth (*C++ function*), 109
ChartWindow (*C++ class*), 103
ChartWindow::addChart (*C++ function*), 104
ChartWindow::addData (*C++ function*), 104
ChartWindow::ChartWindow (*C++ function*), 103
ChartWindow::closeEvent (*C++ function*), 105
ChartWindow::getStep (*C++ function*), 104
ChartWindow::hasTitle (*C++ function*), 103
ChartWindow::loadData (*C++ function*), 105
ChartWindow::numCharts (*C++ function*), 103
ChartWindow::reset (*C++ function*), 104
ChartWindow::resetCharts (*C++ function*), 104
ChartWindow::resetZoom (*C++ function*), 104
ChartWindow::resultWindowCreated (*C++ function*), 105
ChartWindow::setNorm (*C++ function*), 104
ChartWindow::setRangeEnabled (*C++ function*), 104
ChartWindow::setUnits (*C++ function*), 104
Check Input, 52
CMake, 5
CMake configuration, 9
CmdCat (*C++ enum*), 89
CmdCat::Lattice (*C++ enumerator*), 89
CmdCat::Modify (*C++ enumerator*), 89
CmdCat::Other (*C++ enumerator*), 89
CmdCat::Output (*C++ enumerator*), 89
CmdCat::Particle (*C++ enumerator*), 89
CmdCat::Run (*C++ enumerator*), 89
CmdCat::Setup (*C++ enumerator*), 89
CmdCat::Special (*C++ enumerator*), 89
code completion, 47
code formatting, 48
code formatting preferences, 58
CodeEditor (*C++ class*), 73
CodeEditor::~CodeEditor (*C++ function*), 74
CodeEditor::canInsertFromMimeData (*C++ function*), 78
CodeEditor::CodeEditor (*C++ function*), 74
CodeEditor::contextMenuEvent (*C++ function*), 78
CodeEditor::dragEnterEvent (*C++ function*), 78
CodeEditor::dragLeaveEvent (*C++ function*), 78
CodeEditor::dropEvent (*C++ function*), 78
CodeEditor::event (*C++ function*), 77
CodeEditor::keyPressEvent (*C++ function*), 78
CodeEditor::lineNumberAreaPaintEvent (*C++ function*), 74
CodeEditor::lineNumberAreaWidth (*C++ function*), 74
CodeEditor::NO_HIGHLIGHT (*C++ member*), 77
CodeEditor::operator= (*C++ function*), 74
CodeEditor::paintEvent (*C++ function*), 77
CodeEditor::reformatLine (*C++ function*), 75
CodeEditor::resizeEvent (*C++ function*), 77
CodeEditor::setAngleList (*C++ function*), 76
CodeEditor::setAtomList (*C++ function*), 75
CodeEditor::setAutoComplete (*C++ function*), 75

CodeEditor::setBondList (*C++ function*), 75
CodeEditor::setColorList (*C++ function*), 76
CodeEditor::setCommandList (*C++ function*), 75
CodeEditor::setComputeIDList (*C++ function*), 77
CodeEditor::setComputeList (*C++ function*), 75
CodeEditor::setCursor (*C++ function*), 74
CodeEditor::setDihedralList (*C++ function*), 76
CodeEditor::setDocver (*C++ function*), 78
CodeEditor::setDumpList (*C++ function*), 75
CodeEditor::setExtraList (*C++ function*), 76
CodeEditor::setFileList (*C++ function*), 77
CodeEditor::setFixIDList (*C++ function*), 77
CodeEditor::setFixList (*C++ function*), 75
CodeEditor::setFont (*C++ function*), 74
CodeEditor::setGroupList (*C++ function*), 77
CodeEditor::setHighlight (*C++ function*), 74
CodeEditor::setImageKwList (*C++ function*), 77
CodeEditor::setImproperList (*C++ function*), 76
CodeEditor::setIntegrateList (*C++ function*), 76
CodeEditor::setKspaceList (*C++ function*), 76
CodeEditor::setMinimizeList (*C++ function*), 76
CodeEditor::setPairList (*C++ function*), 75
CodeEditor::setReformatOnReturn (*C++ function*), 74
CodeEditor::setRegionList (*C++ function*), 76
CodeEditor::setSyntax (*C++ function*), 74
CodeEditor::setUnitsList (*C++ function*), 76
CodeEditor::setVariableList (*C++ function*), 76
CodeEditor::setVariableOverrides (*C++ function*), 77
CodeEditor::setVarNameList (*C++ function*), 77
color customization, 42
color map, 39, 202
colorMapDef (*C++ function*), 127
ColorMapDef (*C++ struct*), 128
ColorMapDef::continuous (*C++ member*), 128
ColorMapDef::stops (*C++ member*), 128
colorMapNames (*C++ function*), 127
ColorMapStop (*C++ struct*), 127
ColorMapStop::b (*C++ member*), 128
ColorMapStop::g (*C++ member*), 128
ColorMapStop::name (*C++ member*), 128
ColorMapStop::pos (*C++ member*), 128
ColorMapStop::r (*C++ member*), 128
columnAccessor (*C++ function*), 185
ColumnRef (*C++ struct*), 187
ColumnRef::accessor (*C++ member*), 187
ColumnRef::column (*C++ member*), 187
ColumnRef::variable (*C++ member*), 187
ColumnRefExpr (*C++ struct*), 188
ColumnRefExpr::error (*C++ member*), 188
ColumnRefExpr::expr (*C++ member*), 188
ColumnRefExpr::ok (*C++ member*), 188
ColumnRefExpr::refs (*C++ member*), 188
command window, 28
command-line arguments, 13

command-line options, 13
CommandSpec (C++ *struct*), 92
CommandSpec::args (C++ *member*), 92
CommandSpec::category (C++ *member*), 92
CommandSpec::minArgs (C++ *member*), 92
CommandSpec::name (C++ *member*), 92
CommandSpec::repeatLast (C++ *member*), 92
CommandWindow (C++ *class*), 147
CommandWindow::~CommandWindow (C++ *function*), 147
CommandWindow::availableShells (C++ *function*), 148
CommandWindow::changeDirectory (C++ *function*), 147
CommandWindow::CommandWindow (C++ *function*), 147
CommandWindow::eventFilter (C++ *function*), 148
CommandWindow::operator= (C++ *function*), 147
CommandWindow::preferredShell (C++ *function*), 148
compilation, 5
    from source, 9
    plugin mode, 10
CompiledExpression (C++ *class*), 186
CompiledExpression::~CompiledExpression (C++ *function*), 187
CompiledExpression::CompiledExpression (C++ *function*), 187
CompiledExpression::error (C++ *function*), 187
CompiledExpression::errorMsg (C++ *member*), 187
CompiledExpression::evaluate (C++ *function*), 187
CompiledExpression::isValid (C++ *function*), 187
CompiledExpression::operator= (C++ *function*), 187
CompiledExpression::program (C++ *member*), 187
CompiledExpression::valid (C++ *member*), 187
CompleterKind (C++ *enum*), 91
CompleterKind::Command (C++ *enumerator*), 91
CompleterKind::ComputeId (C++ *enumerator*), 91
CompleterKind::Extra (C++ *enumerator*), 91
CompleterKind::File (C++ *enumerator*), 91
CompleterKind::FixId (C++ *enumerator*), 91
CompleterKind::Group (C++ *enumerator*), 91
CompleterKind::None (C++ *enumerator*), 91
CompleterKind::Style (C++ *enumerator*), 91
CompleterKind::VarName (C++ *enumerator*), 91
CompletionTarget (C++ *struct*), 93
CompletionTarget::cat (C++ *member*), 93
CompletionTarget::kind (C++ *member*), 93
CompletionTarget::wordLength (C++ *member*), 93
CompletionTarget::wordStart (C++ *member*), 93
compute graphics, 41
configuration, 56
configuredChartColor (C++ *function*), 189
confirmUnexpectedFile (C++ *function*), 195
context help, 49
critical (C++ *function*), 191
CSV export, 20
curve fitting, 21
custom fit, 21
custom function, 21
CustomCurve (C++ *struct*), 188

CustomCurve::error (*C++ member*), 188
CustomCurve::ok (*C++ member*), 188
CustomCurve::points (*C++ member*), 188
CustomFit (*C++ struct*), 188
CustomFit::curve (*C++ member*), 189
CustomFit::error (*C++ member*), 189
CustomFit::iterations (*C++ member*), 189
CustomFit::ok (*C++ member*), 189
CustomFit::params (*C++ member*), 189
CustomFit::rms (*C++ member*), 189

D

data export, 20
data visualization, 19
dateCompare (*C++ function*), 191
defaultFileStem (*C++ function*), 192
dialogs, 55
    Find and Replace, 55
    Preferences, 56
dockedLayout (*C++ function*), 199
documentation
    inline help, 49
    online, 49, 54
DownloadProgress (*C++ class*), 155
DownloadProgress::~DownloadProgress (*C++ function*), 155
DownloadProgress::DownloadProgress (*C++ function*), 155
DownloadProgress::operator= (*C++ function*), 155
DownloadProgress::setBusy (*C++ function*), 155
DownloadProgress::setProgress (*C++ function*), 155
dump image, 31
dump_modify, 34
DumpImageCommand (*C++ struct*), 126
DumpImageCommand::dofixes (*C++ member*), 126
DumpImageCommand::dumpargs (*C++ member*), 126
DumpImageCommand::modifyargs (*C++ member*), 126
DumpImageParams (*C++ struct*), 118
DumpImageParams::adiams (*C++ member*), 119
DumpImageParams::ambientlight (*C++ member*), 125
DumpImageParams::antialias (*C++ member*), 122
DumpImageParams::atomcolor (*C++ member*), 119
DumpImageParams::atomcustom (*C++ member*), 119
DumpImageParams::atomdiam (*C++ member*), 119
DumpImageParams::atomSize (*C++ member*), 119
DumpImageParams::atomtrans (*C++ member*), 125
DumpImageParams::autobond (*C++ member*), 121
DumpImageParams::axesdiam (*C++ member*), 124
DumpImageParams::axeslen (*C++ member*), 123
DumpImageParams::axesloc (*C++ member*), 123
DumpImageParams::axestrans (*C++ member*), 124
DumpImageParams::backcolor (*C++ member*), 124
DumpImageParams::backcolor2 (*C++ member*), 124
DumpImageParams::backlight (*C++ member*), 125
DumpImageParams::body_flag (*C++ member*), 120
DumpImageParams::bodycolor (*C++ member*), 120

DumpImageParams::bodydiam (*C++ member*), 120
DumpImageParams::bodyflag (*C++ member*), 120
DumpImageParams::bond_flag (*C++ member*), 121
DumpImageParams::bondbyvalue (*C++ member*), 121
DumpImageParams::bondcolor (*C++ member*), 121
DumpImageParams::bondcolormap (*C++ member*), 125
DumpImageParams::bondcutoff (*C++ member*), 121
DumpImageParams::bonddiam (*C++ member*), 121
DumpImageParams::bondmapmax (*C++ member*), 126
DumpImageParams::bondmapmin (*C++ member*), 125
DumpImageParams::bondtrans (*C++ member*), 125
DumpImageParams::boxcolor (*C++ member*), 124
DumpImageParams::boxdiam (*C++ member*), 123
DumpImageParams::boxtrans (*C++ member*), 125
DumpImageParams::color_list (*C++ member*), 124
DumpImageParams::colormap (*C++ member*), 125
DumpImageParams::computes (*C++ member*), 126
DumpImageParams::defocusfactor (*C++ member*), 122
DumpImageParams::defocusstart (*C++ member*), 123
DumpImageParams::depthcuecolor (*C++ member*), 122
DumpImageParams::depthcuefactor (*C++ member*), 122
DumpImageParams::depthcuestart (*C++ member*), 122
DumpImageParams::dimension (*C++ member*), 122
DumpImageParams::dumpfile (*C++ member*), 119
DumpImageParams::dynamiccenter (*C++ member*), 124
DumpImageParams::elements (*C++ member*), 119
DumpImageParams::ellipsoid_flag (*C++ member*), 120
DumpImageParams::ellipsoidcolor (*C++ member*), 121
DumpImageParams::ellipsoiddiam (*C++ member*), 121
DumpImageParams::ellipsoidflag (*C++ member*), 121
DumpImageParams::ellipsoidlevel (*C++ member*), 121
DumpImageParams::filllight (*C++ member*), 125
DumpImageParams::fixes (*C++ member*), 126
DumpImageParams::gammaval (*C++ member*), 125
DumpImageParams::group (*C++ member*), 119
DumpImageParams::haspairstyle (*C++ member*), 121
DumpImageParams::hrot (*C++ member*), 122
DumpImageParams::keylight (*C++ member*), 125
DumpImageParams::line_flag (*C++ member*), 120
DumpImageParams::linecolor (*C++ member*), 120
DumpImageParams::linediam (*C++ member*), 120
DumpImageParams::mapmax (*C++ member*), 125
DumpImageParams::mapmin (*C++ member*), 125
DumpImageParams::metalfactor (*C++ member*), 123
DumpImageParams::metalfinish (*C++ member*), 123
DumpImageParams::nbondtypes (*C++ member*), 121
DumpImageParams::ntypes (*C++ member*), 124
DumpImageParams::outlinecolor (*C++ member*), 123
DumpImageParams::outlinewidth (*C++ member*), 123
DumpImageParams::regions (*C++ member*), 126
DumpImageParams::revbondcolormap (*C++ member*), 126
DumpImageParams::revcolormap (*C++ member*), 125
DumpImageParams::shinyfactor (*C++ member*), 122
DumpImageParams::showatoms (*C++ member*), 119

DumpImageParams::showaxes (C++ member), 123
DumpImageParams::showbodies (C++ member), 119
DumpImageParams::showbonds (C++ member), 121
DumpImageParams::showbox (C++ member), 123
DumpImageParams::showellipsoids (C++ member), 120
DumpImageParams::showlines (C++ member), 120
DumpImageParams::showsubbox (C++ member), 123
DumpImageParams::showtris (C++ member), 120
DumpImageParams::specular (C++ member), 123
DumpImageParams::ssaosamples (C++ member), 122
DumpImageParams::ssaoval (C++ member), 122
DumpImageParams::subboxdiam (C++ member), 123
DumpImageParams::tri_flag (C++ member), 120
DumpImageParams::tricolor (C++ member), 120
DumpImageParams::tridiam (C++ member), 120
DumpImageParams::triflag (C++ member), 120
DumpImageParams::usedefocus (C++ member), 122
DumpImageParams::usedepthcue (C++ member), 122
DumpImageParams::usediameter (C++ member), 119
DumpImageParams::useelements (C++ member), 119
DumpImageParams::usegradient (C++ member), 124
DumpImageParams::usemetal (C++ member), 123
DumpImageParams::useoutline (C++ member), 123
DumpImageParams::usesigma (C++ member), 119
DumpImageParams::usessao (C++ member), 122
DumpImageParams::vdwfactor (C++ member), 119
DumpImageParams::version (C++ member), 125
DumpImageParams::vrot (C++ member), 122
DumpImageParams::xcenter (C++ member), 124
DumpImageParams::xsize (C++ member), 121
DumpImageParams::xup (C++ member), 124
DumpImageParams::ycenter (C++ member), 124
DumpImageParams::ysize (C++ member), 121
DumpImageParams::yup (C++ member), 124
DumpImageParams::zcenter (C++ member), 124
DumpImageParams::zoom (C++ member), 122
DumpImageParams::zup (C++ member), 124
dynamic library loading, 10

E

Edit menu, 51
editing features, 47
editor settings, 58
editor window, 47
EditorTab (C++ class), 145
EditorTab::EditorTab (C++ function), 145
ensureFileSuffix (C++ function), 192
EosFit (C++ struct), 182
EosFit::a (C++ member), 183
EosFit::b (C++ member), 183
EosFit::b0 (C++ member), 183
EosFit::b0prime (C++ member), 183
EosFit::c (C++ member), 183
EosFit::d (C++ member), 183

EosFit::e0 (*C++ member*), 183
EosFit::ok (*C++ member*), 183
EosFit::rms (*C++ member*), 183
EosFit::v0 (*C++ member*), 183
equation of state, 21
evalBirchMurnaghan (*C++ function*), 182
evalCustomCurve (*C++ function*), 185
evalPolynomial (*C++ function*), 182
exportImage (*C++ function*), 192
extractMovieFrames (*C++ function*), 134

F

features, 12
file inspection, 49
File menu, 50
file operations, 15
FileViewer (*C++ class*), 156
FileViewer::~FileViewer (*C++ function*), 156
FileViewer::changeEvent (*C++ function*), 156
FileViewer::FileViewer (*C++ function*), 156
FileViewer::operator= (*C++ function*), 156
FileViewer::resizeEvent (*C++ function*), 156
Find and Replace, 51, 55
FindAndReplace (*C++ class*), 135
FindAndReplace::~FindAndReplace (*C++ function*), 135
FindAndReplace::FindAndReplace (*C++ function*), 135
FindAndReplace::operator= (*C++ function*), 135
findExe (*C++ function*), 193
findVarRefs (*C++ function*), 86
fitCustomCurve (*C++ function*), 186
FitParam (*C++ struct*), 188
FitParam::name (*C++ member*), 188
FitParam::value (*C++ member*), 188
fitViewerWindow (*C++ function*), 199
fix ave import, 24
fix graphics, 41
FlagWarnings (*C++ class*), 158
FlagWarnings::~FlagWarnings (*C++ function*), 158
FlagWarnings::errorUrlPattern (*C++ function*), 159
FlagWarnings::FlagWarnings (*C++ function*), 158
FlagWarnings::getNWarnings (*C++ function*), 158
FlagWarnings::highlightBlock (*C++ function*), 159
FlagWarnings::operator= (*C++ function*), 158
FlagWarnings::reset (*C++ function*), 159
FlagWarnings::summaryText (*C++ function*), 159
FlagWarnings::warningPattern (*C++ function*), 159
flatpak, 9
float_mat (*C++ class*), 178
float_mat::~float_mat (*C++ function*), 179
float_mat::float_mat (*C++ function*), 179
float_mat::nr_cols (*C++ function*), 179
float_mat::nr_rows (*C++ function*), 179
float_mat::operator= (*C++ function*), 179
float_vect (*C++ type*), 177

forceLayout (*C++ function*), 200
fourier transform, 21
FourierKind (*C++ enum*), 179
FourierKind::Cosine (*C++ enumerator*), 179
FourierKind::Power (*C++ enumerator*), 180
FourierKind::Sine (*C++ enumerator*), 180
fourierTransform (*C++ function*), 180
FourierWindow (*C++ enum*), 180
FourierWindow::Hann (*C++ enumerator*), 180
FourierWindow::None (*C++ enumerator*), 180
full LAMMPS packages, 5

G

general settings, 56
GeneralTab (*C++ class*), 143
GeneralTab::GeneralTab (*C++ function*), 143
getLammpsDownloadUrl (*C++ function*), 192
getLammpsLibName (*C++ function*), 192
getting started, 13
global image settings, 34
GPU acceleration, 57
GPU support, 6
grayscaleImage (*C++ function*), 196
grayscalePixmap (*C++ function*), 196

H

hasExe (*C++ function*), 193
help, 54
Highlighter (*C++ class*), 79
Highlighter::~Highlighter (*C++ function*), 80
Highlighter::highlightBlock (*C++ function*), 80
Highlighter::Highlighter (*C++ function*), 80
Highlighter::operator= (*C++ function*), 80
Highlighter::setCursorPos (*C++ function*), 80
hotkeys, 58

I

image computes, 41
image export, 43
image fixes, 41
image rendering, 58
image sequence, 43
image settings, 34
image viewer, 31
image viewer controls, 32
ImageCache (*C++ class*), 129
ImageCache::~ImageCache (*C++ function*), 130
ImageCache::cachedBytes (*C++ function*), 132
ImageCache::cachedImages (*C++ function*), 132
ImageCache::clear (*C++ function*), 132
ImageCache::conversions (*C++ function*), 132
ImageCache::count (*C++ function*), 131
ImageCache::forget (*C++ function*), 131
ImageCache::frameBytes (*C++ function*), 132

ImageCache::frameImages (*C++ function*), 132
ImageCache::ImageCache (*C++ function*), 130
ImageCache::imageSize (*C++ function*), 130
ImageCache::isEmpty (*C++ function*), 132
ImageCache::makeSubDir (*C++ function*), 131
ImageCache::operator= (*C++ function*), 130
ImageCache::path (*C++ function*), 131
ImageCache::purgeConversions (*C++ function*), 131
ImageCache::readImage (*C++ function*), 130
ImageCache::registerFrames (*C++ function*), 131
ImageInfo (*C++ class*), 159
ImageInfo::color (*C++ member*), 160
ImageInfo::colorstyle (*C++ member*), 160
ImageInfo::enabled (*C++ member*), 160
ImageInfo::flag1 (*C++ member*), 160
ImageInfo::flag2 (*C++ member*), 160
ImageInfo::ImageInfo (*C++ function*), 160
ImageInfo::opacity (*C++ member*), 160
ImageInfo::style (*C++ member*), 160
ImageViewer (*C++ class*), 117
ImageViewer::~ImageViewer (*C++ function*), 118
ImageViewer::createImage (*C++ function*), 118
ImageViewer::eventFilter (*C++ function*), 118
ImageViewer::ImageViewer (*C++ function*), 118
ImageViewer::operator= (*C++ function*), 118
ImageViewer::showEvent (*C++ function*), 118
indentation, 48
IndexVariableMatch (*C++ struct*), 139
IndexVariableMatch::name (*C++ member*), 139
IndexVariableMatch::valid (*C++ member*), 139
IndexVariableMatch::value (*C++ member*), 139
IndexVariableMatch::valueLength (*C++ member*), 139
IndexVariableMatch::valueStart (*C++ member*), 139
information (*C++ function*), 191
input script variables, 27
InputScanner (*C++ class*), 82
InputScanner::~InputScanner (*C++ function*), 83
InputScanner::commands (*C++ function*), 83
InputScanner::Diag (*C++ enum*), 83
InputScanner::Diag::DanglingContinuation (*C++ enumerator*), 83
InputScanner::Diag::UnbalancedQuote (*C++ enumerator*), 83
InputScanner::Diag::UnclosedTriple (*C++ enumerator*), 83
InputScanner::Diagnostic (*C++ struct*), 83
InputScanner::Diagnostic::line (*C++ member*), 83
InputScanner::Diagnostic::what (*C++ member*), 83
InputScanner::diagnostics (*C++ function*), 83
InputScanner::InputScanner (*C++ function*), 83
InputScanner::LogicalCommand (*C++ struct*), 84
InputScanner::LogicalCommand::firstLine (*C++ member*), 84
InputScanner::LogicalCommand::lastLine (*C++ member*), 84
InputScanner::LogicalCommand::words (*C++ member*), 84
InputScanner::operator= (*C++ function*), 83
InputScanner::scan (*C++ function*), 83
InputScanner::Word (*C++ struct*), 84

InputScanner::Word::hasSubst (*C++ member*), 84
InputScanner::Word::line (*C++ member*), 84
InputScanner::Word::quoted (*C++ member*), 84
InputScanner::Word::text (*C++ member*), 84
installation, 5
    pre-compiled packages, 5
installViewMenuBar (*C++ function*), 198
int_vect (*C++ type*), 177
invert (*C++ function*), 178
isImageFile (*C++ function*), 193
isLightTheme (*C++ function*), 194
isMainWindowShortcut (*C++ function*), 200
isMovieFile (*C++ function*), 193
isNumberWord (*C++ function*), 87
isOverridden (*C++ function*), 140
isRestartFile (*C++ function*), 194
isStdoutSilenced (*C++ function*), 196

J

JSON color file, 43
JSON format, 43

K

keybindings, 58
keyboard shortcuts, 15, 50, 58
KOKKOS package, 6

L

LAMMPS documentation, 49, 54
LAMMPS execution, 15, 51
LAMMPS library interface, 51
LAMMPS tutorials, 54
LammpsGui (*C++ class*), 65
LammpsGui::~LammpsGui (*C++ function*), 67
LammpsGui::autoSave (*C++ function*), 70
LammpsGui::clearVariables (*C++ function*), 68
LammpsGui::confirmLintIssues (*C++ function*), 69
LammpsGui::doRun (*C++ function*), 68
LammpsGui::eventFilter (*C++ function*), 70
LammpsGui::hasClipboard (*C++ member*), 71
LammpsGui::hasSystemState (*C++ function*), 69
LammpsGui::inspectFile (*C++ function*), 68
LammpsGui::LammpsGui (*C++ function*), 66, 67
LammpsGui::launchRunner (*C++ function*), 69
LammpsGui::mainx (*C++ member*), 71
LammpsGui::mainy (*C++ member*), 71
LammpsGui::nthreads (*C++ member*), 71
LammpsGui::openFile (*C++ function*), 67
LammpsGui::openImageFiles (*C++ function*), 67
LammpsGui::operator= (*C++ function*), 67
LammpsGui::plotFile (*C++ function*), 67
LammpsGui::populateSyntax (*C++ function*), 69
LammpsGui::presetVariableNames (*C++ function*), 69
LammpsGui::purgeInspectList (*C++ function*), 70

LammpsGui::quit (*C++ function*), 68
LammpsGui::refreshVariables (*C++ function*), 68
LammpsGui::runBuffer (*C++ function*), 68
LammpsGui::runDone (*C++ function*), 69
LammpsGui::setDocver (*C++ function*), 70
LammpsGui::setFont (*C++ function*), 70
LammpsGui::setupTutorial (*C++ function*), 70
LammpsGui::sharedMenus (*C++ function*), 68
LammpsGui::startLammps (*C++ function*), 69
LammpsGui::stopRun (*C++ function*), 68
LammpsGui::tutorialDirectory (*C++ function*), 70
LammpsGui::tutorialIntro (*C++ function*), 70
LammpsGui::updateEditorTitle (*C++ function*), 68
LammpsGui::updateRecents (*C++ function*), 68
LammpsGui::updateVariables (*C++ function*), 68
LammpsGui::viewFile (*C++ function*), 67
LammpsGui::writeFile (*C++ function*), 68
LammpsRunner (*C++ class*), 102
LammpsRunner::~LammpsRunner (*C++ function*), 102
LammpsRunner::LammpsRunner (*C++ function*), 102
LammpsRunner::operator= (*C++ function*), 102
LammpsRunner::resultReady (*C++ function*), 103
LammpsRunner::run (*C++ function*), 103
LammpsRunner::setupRun (*C++ function*), 102
LammpsSyntax (*C++ class*), 80
LammpsSyntax::~LammpsSyntax (*C++ function*), 81
LammpsSyntax::argSpec (*C++ function*), 82
LammpsSyntax::commandCategory (*C++ function*), 81
LammpsSyntax::commandIndex (*C++ function*), 82
LammpsSyntax::completionList (*C++ function*), 82
LammpsSyntax::completionTarget (*C++ function*), 82
LammpsSyntax::isPopulated (*C++ function*), 81
LammpsSyntax::isSpecialWord (*C++ function*), 81
LammpsSyntax::knownCommand (*C++ function*), 81
LammpsSyntax::knownStyle (*C++ function*), 81
LammpsSyntax::LammpsSyntax (*C++ function*), 81
LammpsSyntax::loadCommandSpecs (*C++ function*), 81
LammpsSyntax::loadCommandSpecsFromString (*C++ function*), 81
LammpsSyntax::operator= (*C++ function*), 81
LammpsSyntax::setCommands (*C++ function*), 81
LammpsSyntax::setStyles (*C++ function*), 81
LammpsSyntax::spec (*C++ function*), 82
LammpsWrapper (*C++ class*), 95
LammpsWrapper::~LammpsWrapper (*C++ function*), 96
LammpsWrapper::close (*C++ function*), 96
LammpsWrapper::command (*C++ function*), 97
LammpsWrapper::commandsString (*C++ function*), 97
LammpsWrapper::configAccelerator (*C++ function*), 101
LammpsWrapper::configHasCurlSupport (*C++ function*), 101
LammpsWrapper::configHasJpegSupport (*C++ function*), 101
LammpsWrapper::configHasOmpSupport (*C++ function*), 101
LammpsWrapper::configHasPackage (*C++ function*), 101
LammpsWrapper::configHasPngSupport (*C++ function*), 101
LammpsWrapper::extractAtom (*C++ function*), 97

LammpsWrapper::extractCompute (*C++ function*), 97
LammpsWrapper::extractFix (*C++ function*), 98
LammpsWrapper::extractGlobal (*C++ function*), 97
LammpsWrapper::extractPair (*C++ function*), 97
LammpsWrapper::extractSetting (*C++ function*), 97
LammpsWrapper::extractVariable (*C++ function*), 98
LammpsWrapper::extractVariableDatatype (*C++ function*), 98
LammpsWrapper::file (*C++ function*), 96
LammpsWrapper::finalize (*C++ function*), 96
LammpsWrapper::forceTimeout (*C++ function*), 97
LammpsWrapper::getLastErrorMessage (*C++ function*), 100
LammpsWrapper::getThermo (*C++ function*), 99
LammpsWrapper::hasError (*C++ function*), 100
LammpsWrapper::hasGpuDevice (*C++ function*), 101
LammpsWrapper::hasId (*C++ function*), 98
LammpsWrapper::hasPlugin (*C++ function*), 102
LammpsWrapper::idCount (*C++ function*), 98
LammpsWrapper::idName (*C++ function*), 99
LammpsWrapper::isOpen (*C++ function*), 100
LammpsWrapper::isRunning (*C++ function*), 100
LammpsWrapper::LammpsWrapper (*C++ function*), 96
LammpsWrapper::lastErrorMessage (*C++ function*), 100
LammpsWrapper::lastThermo (*C++ function*), 99
LammpsWrapper::lastThermoAs (*C++ function*), 100
LammpsWrapper::lastThermoString (*C++ function*), 100
LammpsWrapper::loadLib (*C++ function*), 101
LammpsWrapper::open (*C++ function*), 96
LammpsWrapper::operator= (*C++ function*), 96
LammpsWrapper::ScopeConst (*C++ enum*), 95
LammpsWrapper::ScopeConst::DUMMY (*C++ enumerator*), 95
LammpsWrapper::ScopeConst::GLOBAL_STYLE (*C++ enumerator*), 95
LammpsWrapper::ScopeConst::LOCAL_STYLE (*C++ enumerator*), 95
LammpsWrapper::StyleConst (*C++ enum*), 95
LammpsWrapper::StyleConst::ATOM_STYLE (*C++ enumerator*), 95
LammpsWrapper::StyleConst::EQUAL_STYLE (*C++ enumerator*), 95
LammpsWrapper::StyleConst::STRING_STYLE (*C++ enumerator*), 95
LammpsWrapper::StyleConst::VECTOR_STYLE (*C++ enumerator*), 95
LammpsWrapper::styleCount (*C++ function*), 99
LammpsWrapper::styleName (*C++ function*), 99
LammpsWrapper::TypeConst (*C++ enum*), 96
LammpsWrapper::TypeConst::ARRAY_TYPE (*C++ enumerator*), 96
LammpsWrapper::TypeConst::NUM_COLS (*C++ enumerator*), 96
LammpsWrapper::TypeConst::NUM_ROWS (*C++ enumerator*), 96
LammpsWrapper::TypeConst::SCALAR_TYPE (*C++ enumerator*), 96
LammpsWrapper::TypeConst::VECTOR_TYPE (*C++ enumerator*), 96
LammpsWrapper::variableInfo (*C++ function*), 99
LammpsWrapper::version (*C++ function*), 97
LayoutMode (*C++ enum*), 149
LayoutMode::Docked (*C++ enumerator*), 149
LayoutMode::Windows (*C++ enumerator*), 149
layoutViewMenuBar (*C++ function*), 198
legend, 21
LeptonMini (*C++ type*), 189
levmarFit (*C++ function*), 184

LevmarModel (*C++ type*), 183
LevmarResult (*C++ struct*), 184
LevmarResult::iterations (*C++ member*), 184
LevmarResult::message (*C++ member*), 184
LevmarResult::ok (*C++ member*), 184
LevmarResult::params (*C++ member*), 184
LevmarResult::rms (*C++ member*), 184
lin_solve (*C++ function*), 178
line continuation, 48
line reformatting, 48
LineNumberArea (*C++ class*), 78
LineNumberArea::~LineNumberArea (*C++ function*), 79
LineNumberArea::LineNumberArea (*C++ function*), 79
LineNumberArea::operator= (*C++ function*), 79
LineNumberArea::paintEvent (*C++ function*), 79
LineNumberArea::sizeHint (*C++ function*), 79
LineTokens (*C++ struct*), 93
LineTokens::outState (*C++ member*), 93
LineTokens::tokens (*C++ member*), 93
LineTokens::unbalancedQuote (*C++ member*), 93
LintIssue (*C++ struct*), 85
LintIssue::line (*C++ member*), 86
LintIssue::message (*C++ member*), 86
LintIssue::severity (*C++ member*), 86
LintSeverity (*C++ enum*), 86
LintSeverity::Error (*C++ enumerator*), 86
LintSeverity::Warning (*C++ enumerator*), 86
Linux installation, 8
loadPlotBlockData (*C++ function*), 175
loadPlotData (*C++ function*), 169
log window, 18
LogWindow (*C++ class*), 157
LogWindow::~LogWindow (*C++ function*), 157
LogWindow::changeEvent (*C++ function*), 158
LogWindow::checkYaml (*C++ function*), 158
LogWindow::closeEvent (*C++ function*), 157
LogWindow::contextMenuEvent (*C++ function*), 157
LogWindow::LogWindow (*C++ function*), 157
LogWindow::mouseDoubleClickEvent (*C++ function*), 157
LogWindow::operator= (*C++ function*), 157
LogWindow::reset (*C++ function*), 157
LogWindow::resizeEvent (*C++ function*), 158
looksLikeAveBlocks (*C++ function*), 175
looksLikeBinaryFile (*C++ function*), 194

M

macOS installation, 8
main window, 13
makeChartColorCombo (*C++ function*), 189
makeChartModeCombo (*C++ function*), 190
makeLineWidthSpin (*C++ function*), 190
makePlotChoiceCombo (*C++ function*), 190
makePointSizeSpin (*C++ function*), 190
matchIndexVariable (*C++ function*), 139

menu bar, 50
menus, 50
    About, 54
    Edit, 51
    File, 50
    Run, 51
    Tutorials, 54
    View, 53
mergeInputVariables (*C++ function*), 140
monoFontFromSettings (*C++ function*), 190
movie export, 43
movie import, 45
MovieImportDialog (*C++ class*), 133
MovieImportDialog::~MovieImportDialog (*C++ function*), 133
MovieImportDialog::firstFrame (*C++ function*), 133
MovieImportDialog::frameInterval (*C++ function*), 134
MovieImportDialog::lastFrame (*C++ function*), 133
MovieImportDialog::MovieImportDialog (*C++ function*), 133
MovieImportDialog::operator= (*C++ function*), 133
MovieInfo (*C++ struct*), 132
MovieInfo::duration (*C++ member*), 133
MovieInfo::error (*C++ member*), 133
MovieInfo::fps (*C++ member*), 133
MovieInfo::frames (*C++ member*), 132
MovieInfo::height (*C++ member*), 132
MovieInfo::valid (*C++ member*), 132
MovieInfo::width (*C++ member*), 132
MPI parallelization, 6
MY_PI_CONST (*C++ member*), 181

N

notifyCaptureState (*C++ function*), 196

O

OpenCL, 6
opening files, 15
operator* (*C++ function*), 177
output window, 18
overview, 12

P

parseAveBlocks (*C++ function*), 174
parseAveBlocksYaml (*C++ function*), 174
parseFrameRate (*C++ function*), 134
parseInputVariables (*C++ function*), 140
parsePlotCsv (*C++ function*), 169
parsePlotJson (*C++ function*), 169
parsePlotWhitespace (*C++ function*), 169
parsePlotYaml (*C++ function*), 169
parseProbeOutput (*C++ function*), 134
per-type colors, 42
platform notes, 6
plot data file, 23
PlotAxis (*C++ struct*), 116

PlotAxis::gridVisible (*C++ member*), 117
PlotAxis::labelFormat (*C++ member*), 117
PlotAxis::max (*C++ member*), 117
PlotAxis::min (*C++ member*), 117
PlotAxis::minorGridVisible (*C++ member*), 117
PlotAxis::subTicks (*C++ member*), 117
PlotAxis::title (*C++ member*), 117
PlotAxisMath (*C++ type*), 117
PlotBlockData (*C++ struct*), 173
PlotBlockData::blockCount (*C++ function*), 173
PlotBlockData::blocks (*C++ member*), 174
PlotBlockData::columnNames (*C++ function*), 173
PlotBlockData::fixId (*C++ member*), 174
PlotBlockData::isEmpty (*C++ function*), 173
PlotBlockData::kind (*C++ member*), 174
PlotBlockData::scalarNames (*C++ member*), 174
PlotData (*C++ class*), 170
PlotData::~PlotData (*C++ function*), 170
PlotData::addColumn (*C++ function*), 172
PlotData::appendRow (*C++ function*), 171
PlotData::cols (*C++ member*), 172
PlotData::column (*C++ function*), 171
PlotData::columnCount (*C++ function*), 171
PlotData::columnName (*C++ function*), 171
PlotData::columnNames (*C++ function*), 171
PlotData::isEmpty (*C++ function*), 171
PlotData::names (*C++ member*), 172
PlotData::operator= (*C++ function*), 171
PlotData::PlotData (*C++ function*), 170, 171
PlotData::renameColumns (*C++ function*), 171
PlotData::rowCount (*C++ function*), 171
PlotData::setColumnNames (*C++ function*), 171
PlotDataBlock (*C++ struct*), 173
PlotDataBlock::rows (*C++ member*), 173
PlotDataBlock::scalars (*C++ member*), 173
PlotDataBlock::step (*C++ member*), 173
PlotDataDialog (*C++ class*), 140
PlotDataDialog::~PlotDataDialog (*C++ function*), 141
PlotDataDialog::buildData (*C++ function*), 141
PlotDataDialog::buildErrors (*C++ function*), 142
PlotDataDialog::fromFile (*C++ function*), 142
PlotDataDialog::operator= (*C++ function*), 141
PlotDataDialog::PlotDataDialog (*C++ function*), 141
PlotDataDialog::xColumn (*C++ function*), 141
PlotDataDialog::yColumns (*C++ function*), 141
PlotErrors (*C++ struct*), 172
PlotErrors::appendEmpty (*C++ function*), 172
PlotErrors::clear (*C++ function*), 172
PlotErrors::columnCount (*C++ function*), 172
PlotErrors::isAsymmetric (*C++ function*), 172
PlotErrors::isEmpty (*C++ function*), 172
PlotErrors::lower (*C++ member*), 173
PlotErrors::resize (*C++ function*), 172
PlotErrors::upper (*C++ member*), 173

PlotSeries (*C++ struct*), 115
PlotSeries::append (*C++ function*), 115
PlotSeries::at (*C++ function*), 115
PlotSeries::color (*C++ member*), 116
PlotSeries::count (*C++ function*), 115
PlotSeries::errColor (*C++ member*), 116
PlotSeries::errHigh (*C++ function*), 115
PlotSeries::errLow (*C++ function*), 115
PlotSeries::errWidth (*C++ member*), 116
PlotSeries::hasAsymErrors (*C++ function*), 115
PlotSeries::hasErrors (*C++ function*), 115
PlotSeries::isReference (*C++ member*), 116
PlotSeries::isVisible (*C++ function*), 115
PlotSeries::markerSize (*C++ member*), 116
PlotSeries::name (*C++ member*), 116
PlotSeries::points (*C++ member*), 116
PlotSeries::refAnchor (*C++ member*), 116
PlotSeries::refLabel (*C++ member*), 116
PlotSeries::replace (*C++ function*), 115
PlotSeries::setVisible (*C++ function*), 115
PlotSeries::style (*C++ member*), 116
PlotSeries::type (*C++ member*), 116
PlotSeries::visible (*C++ member*), 116
PlotSeries::width (*C++ member*), 116
PlotSeries::yerr (*C++ member*), 116
PlotSeries::yerrLo (*C++ member*), 116
PlotSeriesType (*C++ enum*), 114
PlotSeriesType::Line (*C++ enumerator*), 114
PlotSeriesType::Scatter (*C++ enumerator*), 115
plotting, 19
plotting external data, 23
plotting preferences, 58
PlotWidget (*C++ class*), 113
PlotWidget::addSeries (*C++ function*), 114
PlotWidget::clearSeries (*C++ function*), 114
PlotWidget::hasSeries (*C++ function*), 114
PlotWidget::paintEvent (*C++ function*), 114
PlotWidget::PlotWidget (*C++ function*), 113
PlotWidget::removeSeries (*C++ function*), 114
PlotWidget::setGrid (*C++ function*), 113
PlotWidget::setLegendPos (*C++ function*), 113
PlotWidget::setRefLabelStyle (*C++ function*), 114
PlotWidget::setTitle (*C++ function*), 113
PlotWidget::setXLabelFormat (*C++ function*), 113
PlotWidget::setXRange (*C++ function*), 113
PlotWidget::setXTitle (*C++ function*), 113
PlotWidget::setYRange (*C++ function*), 113
PlotWidget::setYTitle (*C++ function*), 113
PlotWidget::sizeHint (*C++ function*), 114
PlotWidget::xTitle (*C++ function*), 113
PlotWidget::yTitle (*C++ function*), 113
plugin mode, 6
polynomial fit, 21
polynomialFit (*C++ function*), 181

PolynomialFit (*C++ struct*), 182
PolynomialFit::coeffs (*C++ member*), 182
PolynomialFit::ok (*C++ member*), 182
PolynomialFit::rms (*C++ member*), 182
post-processing, 21
pre-compiled executables, 5
pre-compiled packages, 5
preferences, 56
    accelerators, 57
    charts, 58
    editor, 58
    general, 56
    snapshot image, 58
Preferences (*C++ class*), 142
Preferences::~Preferences (*C++ function*), 143
Preferences::operator= (*C++ function*), 143
Preferences::Preferences (*C++ function*), 143
Preferences::requestRelaunch (*C++ function*), 143
Preferences::setRelaunch (*C++ function*), 143
prerequisites, 5
probeMovie (*C++ function*), 134
purgeDirectory (*C++ function*), 194

Q

QColorCompleter (*C++ class*), 163
QColorCompleter::QColorCompleter (*C++ function*), 163
QColorValidator (*C++ class*), 163
QColorValidator::fixup (*C++ function*), 163
QColorValidator::QColorValidator (*C++ function*), 163
QColorValidator::validate (*C++ function*), 163
QHline (*C++ class*), 163
QHline::QHline (*C++ function*), 163
Qt framework, 5
QtMessageSilencer (*C++ class*), 168
QtMessageSilencer::~QtMessageSilencer (*C++ function*), 168
QtMessageSilencer::messages (*C++ function*), 168
QtMessageSilencer::operator= (*C++ function*), 168
QtMessageSilencer::QtMessageSilencer (*C++ function*), 168

R

RangeBandSlider (*C++ class*), 166
RangeBandSlider::~RangeBandSlider (*C++ function*), 166
RangeBandSlider::operator= (*C++ function*), 166
RangeBandSlider::paintEvent (*C++ function*), 166
RangeBandSlider::RangeBandSlider (*C++ function*), 166
RangeBandSlider::setActiveRange (*C++ function*), 166
RangeSlider (*C++ class*), 164
RangeSlider::active_slider (*C++ member*), 165
RangeSlider::click_offset (*C++ member*), 165
RangeSlider::high (*C++ function*), 164
RangeSlider::highLimit (*C++ member*), 165
RangeSlider::hover_control (*C++ member*), 165
RangeSlider::low (*C++ function*), 164
RangeSlider::lowLimit (*C++ member*), 165

RangeSlider::mouseMoveEvent (*C++ function*), 165
RangeSlider::mousePressEvent (*C++ function*), 165
RangeSlider::paintEvent (*C++ function*), 165
RangeSlider::pick (*C++ function*), 165
RangeSlider::pixelPosToRangeValue (*C++ function*), 165
RangeSlider::pressed_control (*C++ member*), 165
RangeSlider::RangeSlider (*C++ function*), 164
RangeSlider::setHigh (*C++ function*), 165
RangeSlider::setLow (*C++ function*), 164
RangeSlider::sliderMoved (*C++ function*), 165
RangeSlider::tick_interval (*C++ member*), 165
RangeSlider::tick_position (*C++ member*), 165
reassertMonoFont (*C++ function*), 190
RefAnchor (*C++ enum*), 115
RefAnchor::Center (*C++ enumerator*), 115
RefAnchor::End (*C++ enumerator*), 115
RefAnchor::Start (*C++ enumerator*), 115
reference lines, 21
region settings, 41
region visualization, 41
RegionInfo (*C++ class*), 160
RegionInfo::color (*C++ member*), 161
RegionInfo::diameter (*C++ member*), 161
RegionInfo::enabled (*C++ member*), 161
RegionInfo::npoints (*C++ member*), 161
RegionInfo::opacity (*C++ member*), 161
RegionInfo::RegionInfo (*C++ function*), 160
RegionInfo::style (*C++ member*), 161
relaunchApplication (*C++ function*), 194
relaunchOrExit (*C++ function*), 194
renameColumnRefs (*C++ function*), 185
renameToBackup (*C++ function*), 193
restart file inspection, 49
restart files, 49
restoreStdout (*C++ function*), 196
retireViewMenuBar (*C++ function*), 197
reversible color map, 39
rigid bodies, 37
Run menu, 51
running LAMMPS, 15

S

saving files, 15
Savitzky-Golay filter, 19
scopeShortcut (*C++ function*), 200
screen output, 18
selectedFrameCount (*C++ function*), 134
setDialogIcons (*C++ function*), 191
setMainWindowShortcuts (*C++ function*), 200
settings, 56
SetVariables (*C++ class*), 136
SetVariables::~SetVariables (*C++ function*), 136
SetVariables::operator= (*C++ function*), 136
SetVariables::SetVariables (*C++ function*), 136

sg_smooth (*C++ function*), 178
shell, 28
ShellAliases (*C++ class*), 136
ShellAliases::~ShellAliases (*C++ function*), 137
ShellAliases::accept (*C++ function*), 137
ShellAliases::aliases (*C++ function*), 137
ShellAliases::defaults (*C++ function*), 137
ShellAliases::operator= (*C++ function*), 137
ShellAliases::ShellAliases (*C++ function*), 137
ShellPrompt (*C++ class*), 137
ShellPrompt::~ShellPrompt (*C++ function*), 138
ShellPrompt::completing (*C++ function*), 138
ShellPrompt::event (*C++ function*), 138
ShellPrompt::operator= (*C++ function*), 138
ShellPrompt::ShellPrompt (*C++ function*), 138
showUnsavedChangesDialog (*C++ function*), 195
silenceStdout (*C++ function*), 196
singleBlock (*C++ function*), 175
slideshow, 43
SlideShow (*C++ class*), 128
slideshow controls, 45
SlideShow::~SlideShow (*C++ function*), 129
SlideShow::addImage (*C++ function*), 129
SlideShow::addMovie (*C++ function*), 129
SlideShow::clear (*C++ function*), 129
SlideShow::imageCount (*C++ function*), 129
SlideShow::operator= (*C++ function*), 129
SlideShow::showEvent (*C++ function*), 129
SlideShow::SlideShow (*C++ function*), 128, 129
smoothing, 19
snapshot image settings, 58
snapshot viewer, 31
SnapshotTab (*C++ class*), 145
SnapshotTab::SnapshotTab (*C++ function*), 145
spectral density, 21
splitLine (*C++ function*), 191
standalone packages, 6
StdCapture (*C++ class*), 161
StdCapture::~StdCapture (*C++ function*), 161
StdCapture::beginCapture (*C++ function*), 161
StdCapture::diagnostic (*C++ function*), 162
StdCapture::endCapture (*C++ function*), 161
StdCapture::getBufferUse (*C++ function*), 162
StdCapture::getCapture (*C++ function*), 162
StdCapture::getChunk (*C++ function*), 162
StdCapture::isUsable (*C++ function*), 162
StdCapture::operator= (*C++ function*), 161
StdCapture::probeRunEnd (*C++ function*), 162
StdCapture::StdCapture (*C++ function*), 161
StdCapture::totalRead (*C++ function*), 162
StdoutSilencer (*C++ class*), 167
StdoutSilencer::~StdoutSilencer (*C++ function*), 167
StdoutSilencer::operator= (*C++ function*), 168
StdoutSilencer::StdoutSilencer (*C++ function*), 167

structure factor, 21
structureFactor (*C++ function*), 181
StyleCat (*C++ enum*), 87
StyleCat::Angle (*C++ enumerator*), 88
StyleCat::Atom (*C++ enumerator*), 88
StyleCat::Bond (*C++ enumerator*), 88
StyleCat::Color (*C++ enumerator*), 89
StyleCat::Command (*C++ enumerator*), 87
StyleCat::Compute (*C++ enumerator*), 88
StyleCat::Dihedral (*C++ enumerator*), 88
StyleCat::Dump (*C++ enumerator*), 88
StyleCat::Extra (*C++ enumerator*), 88
StyleCat::Fix (*C++ enumerator*), 87
StyleCat::ImageKw (*C++ enumerator*), 89
StyleCat::Improper (*C++ enumerator*), 88
StyleCat::Integrate (*C++ enumerator*), 88
StyleCat::Kspace (*C++ enumerator*), 88
StyleCat::Minimize (*C++ enumerator*), 88
StyleCat::None (*C++ enumerator*), 89
StyleCat::Pair (*C++ enumerator*), 88
StyleCat::Region (*C++ enumerator*), 88
StyleCat::Units (*C++ enumerator*), 88
StyleCat::Variable (*C++ enumerator*), 88
styleDialogButtons (*C++ function*), 195
styleMessageBoxButtons (*C++ function*), 195
styleToolButtons (*C++ function*), 197
substituteColumnRefs (*C++ function*), 185
syntax highlighting, 47, 48
SyntaxChecker (*C++ class*), 84
SyntaxChecker::~SyntaxChecker (*C++ function*), 85
SyntaxChecker::check (*C++ function*), 85
SyntaxChecker::countErrors (*C++ function*), 85
SyntaxChecker::formatIssues (*C++ function*), 85
SyntaxChecker::operator= (*C++ function*), 85
SyntaxChecker::SyntaxChecker (*C++ function*), 85
SyntaxState (*C++ type*), 93
SyntaxState::ARG_MASK (*C++ member*), 95
SyntaxState::ARG_MAXX (*C++ member*), 95
SyntaxState::ARG_SHIFT (*C++ member*), 95
SyntaxState::argsUsed (*C++ function*), 94
SyntaxState::CMD_MASK (*C++ member*), 94
SyntaxState::CMD_SHIFT (*C++ member*), 94
SyntaxState::cmdIndex (*C++ function*), 94
SyntaxState::COMMENT (*C++ member*), 94
SyntaxState::CONTINUE (*C++ member*), 94
SyntaxState::DOUBLEQ (*C++ member*), 94
SyntaxState::FLAG_MASK (*C++ member*), 94
SyntaxState::flags (*C++ function*), 94
SyntaxState::logicalContinues (*C++ function*), 94
SyntaxState::MIDWORD (*C++ member*), 94
SyntaxState::pack (*C++ function*), 94
SyntaxState::SINGLEQ (*C++ member*), 94
SyntaxState::TRIPLE (*C++ member*), 94
SyntaxState::withArgs (*C++ function*), 94

SyntaxState::withCommand (*C++ function*), 94

T

text editor, 47
text search, 55
thermodynamic output, 19
thread parallelization, 57
Token (*C++ struct*), 92
Token::argIndex (*C++ member*), 92
Token::fragment (*C++ member*), 93
Token::hasSubst (*C++ member*), 93
Token::isNumber (*C++ member*), 92
Token::length (*C++ member*), 92
Token::start (*C++ member*), 92
Token::type (*C++ member*), 92
tokenizeLine (*C++ function*), 86
TokType (*C++ enum*), 90
TokType::Comment (*C++ enumerator*), 91
TokType::Continuation (*C++ enumerator*), 91
TokType::String (*C++ enumerator*), 90
TokType::TripleString (*C++ enumerator*), 90
TokType::VarRef (*C++ enumerator*), 91
TokType::Word (*C++ enumerator*), 90
toolButtonSize (*C++ function*), 197
toWriteDumpCommand (*C++ function*), 127
transpose (*C++ function*), 177
tutorial, 203
tutorial wizard, 54
tutorialCollection (*C++ function*), 73
TutorialCollection (*C++ struct*), 72
TutorialCollection::author (*C++ member*), 72
TutorialCollection::available (*C++ member*), 73
TutorialCollection::blurbs (*C++ member*), 73
TutorialCollection::count (*C++ function*), 72
TutorialCollection::dirPrefix (*C++ member*), 72
TutorialCollection::filesRepoUrl (*C++ member*), 72
TutorialCollection::filesUrl (*C++ member*), 72
TutorialCollection::key (*C++ member*), 72
TutorialCollection::logo (*C++ member*), 72
TutorialCollection::logoFor (*C++ function*), 72
TutorialCollection::name (*C++ member*), 72
TutorialCollection::published (*C++ member*), 73
TutorialCollection::siteUrl (*C++ member*), 72
TutorialCollection::slugs (*C++ member*), 73
TutorialCollection::status (*C++ member*), 73
TutorialCollection::titles (*C++ member*), 73
TutorialCollection::webUrl (*C++ member*), 72
tutorialCollections (*C++ function*), 73
Tutorials menu, 54
TutorialWizard (*C++ class*), 71
TutorialWizard::accept (*C++ function*), 71
TutorialWizard::TutorialWizard (*C++ function*), 71

U

unknown commands, 48
URLDownloader (C++ *class*), 153
URLDownloader::~URLDownloader (C++ *function*), 153
URLDownloader::abort (C++ *function*), 154
URLDownloader::download (C++ *function*), 153
URLDownloader::errorString (C++ *function*), 154
URLDownloader::getLocalChecksum (C++ *function*), 154
URLDownloader::getRemoteChecksum (C++ *function*), 154
URLDownloader::operator= (C++ *function*), 153
URLDownloader::URLDownloader (C++ *function*), 153
URLDownloader::wasAborted (C++ *function*), 154

V

variable info window, 27
VariableEntry (C++ *struct*), 138
VariableEntry::name (C++ *member*), 139
VariableEntry::scriptValue (C++ *member*), 139
VariableEntry::value (C++ *member*), 139
variables, 27
VDW style, 37
VerticalLabel (C++ *class*), 166
VerticalLabel::minimumSizeHint (C++ *function*), 167
VerticalLabel::paintEvent (C++ *function*), 167
VerticalLabel::setText (C++ *function*), 167
VerticalLabel::sizeHint (C++ *function*), 167
VerticalLabel::text (C++ *function*), 167
VerticalLabel::VerticalLabel (C++ *function*), 167
View menu, 53
viewerFitSize (C++ *function*), 198
ViewSlot (C++ *enum*), 148
ViewSlot::Chart (C++ *enumerator*), 149
ViewSlot::Command (C++ *enumerator*), 149
ViewSlot::Count (C++ *enumerator*), 149
ViewSlot::Image (C++ *enumerator*), 149
ViewSlot::Log (C++ *enumerator*), 149
ViewSlot::SlideShow (C++ *enumerator*), 149
ViewSlot::Variables (C++ *enumerator*), 149
visual style, 13
visualization, 31

W

warning (C++ *function*), 191
window size, 13
window visibility, 53
WindowLayout (C++ *class*), 149
WindowLayout::~WindowLayout (C++ *function*), 150
WindowLayout::addAuxiliaryView (C++ *function*), 152
WindowLayout::eventFilter (C++ *function*), 153
WindowLayout::focusNextPane (C++ *function*), 152
WindowLayout::hide (C++ *function*), 151
WindowLayout::isVisible (C++ *function*), 152
WindowLayout::mode (C++ *function*), 150

WindowLayout::operator= (*C++ function*), 150
WindowLayout::place (*C++ function*), 150
WindowLayout::raise (*C++ function*), 151
WindowLayout::saveState (*C++ function*), 152
WindowLayout::setVisible (*C++ function*), 151
WindowLayout::show (*C++ function*), 151
WindowLayout::toggle (*C++ function*), 151
WindowLayout::view (*C++ function*), 151
WindowLayout::viewActivated (*C++ function*), 152
WindowLayout::WindowLayout (*C++ function*), 150
word completion, 47
writePlotCsv (*C++ function*), 170
writePlotDat (*C++ function*), 170
writePlotYaml (*C++ function*), 170

Y

YAML export, 20