Pizza.py Users Manual

Pizza.py Toolkit

http://pizza.sandia.gov - Sandia National Laboratories

Copyright (2005) Sandia Corporation. This software and manual is distributed under the GNU General Public License.

Table of Contents

Table of Contents

Pizza.py Documentation

Version info:

The Pizza.py "version" is the date when it was released, such as 1 May 2010. Pizza.py is updated continuously. Whenever we fix a bug or add a feature, we release it immediately, and post a notice on this page of the WWW site. Each dated copy of Pizza.py contains all the features and bug-fixes up to and including that version date. The version date is printed to the screen and logfile every time you run Pizza.py. It is also in the file src/version.h and in the Pizza.py directory name created when you unpack a tarball.

- If you browse the HTML or PDF doc pages on the Pizza.py WWW site, they always describe the most current version of Pizza.py.
- If you browse the HTML or PDF doc pages included in your tarball, they describe the version you have.

Pizza.py is a loosely integrated collection of tools written in Python, many of which provide pre- and post-processing capability for the LAMMPS molecular dynamics, ChemCell cell simulator, and SPARTA Direcct Simulation Monte Carlo (DSMC) packages. There are tools to create input files, convert between file formats, process log and dump files, create plots, and visualize and animate simulation snapshots.

The name Pizza.py is meant to evoke the aroma of a collection of "toppings" that the user can combine in different ways on a "crust" of basic functionality, with Python as the "cheese" that glues everything together.

The maintainer of Pizza.py is Steve Plimpton at Sandia National Laboratories, a US Department of Energy (DOE) laboratory. Many of the tools were written by Matt Jones, a BYU student who spent a summer at Sandia. The Pizza.py WWW Site at http://pizza.sandia.gov has more information about Pizza.py and its uses.

The Pizza.py documentation is organized into the following sections. If you find errors or omissions in this manual or have suggestions for useful information to add, please send an email to sjplimp@sandia.gov so we can improve the Pizza.py documentation.

PDF file of the entire manual, generated by htmldoc

List of tools in Pizza.py:

animate.py	Animate a series of image files
bdump.py	Read LAMMPS dump local files with bond info
cdata.py	Read, write, manipulate ChemCell data files
chain.py	Create bead-spring chains for LAMMPS input
cfg.py	Convert LAMMPS snapshots to CFG format
data.py	Read, write, manipulate LAMMPS data files
dump.py	Read, write, manipulate dump files and particle attributes
ensight.py	Convert LAMMPS snapshots to Ensight format
gl.py	3d interactive visualization via OpenGL
gnu.py	Create plots via GnuPlot plotting program
histo.py	Particle density histogram from a dump
image.py	View and manipulate images
ldump.py	Read LAMMPS dump files with line info
log.py	Read LAMMPS log files and extract thermodynamic data
matlab.py	Create plots via MatLab numerical analysis program
mdump.py	Read, write, manipulate mesh dump files
olog.py	Read other log files (ChemCell, SPPARKS, SPARTA) and extract time-series data
pair.py	Compute LAMMPS pairwise energies
patch.py	Create patchy Lennard-Jones particles for LAMMPS input
pdbfile.py	Read, write PDB files in combo with LAMMPS snapshots
plotview.py	Plot multiple vectors from a data set
rasmol.py	3d visualization via RasMol program
raster.py	3d visualization via Raster3d program
sdata.py	Read, write, manipulate SPARTA surface files
svg.py	3d visualization via SVG files
tdump.py	Read LAMMPS dump files with triangle info
vcr.py	VCR-style GUI for 3d interactive OpenGL visualization
vec.py	Create numeric vectors from columns in file or list of vecs
vtk.py	Convert LAMMPS snapshots to VTK format
vmd.py	Wrapper on VMD visualization package
xyz.py	Convert LAMMPS snapshots to XYZ format

1. Introduction

These sections describe what Pizza.py is, what it means to be open-source software, and acknowledge the funding and people who have contributed to Pizza.py.

1.1 What is Pizza.py

Pizza.py is a loosely integrated collection of tools, many of which provide pre- and post-processing capabilities for the LAMMPS molecular dynamics and ChemCell cell simulator packages.

There are tools to create input files, convert between file formats to connect to other codes, process log and dump files, plot output, and visualize and animate simulation snapshots.

Python is used in Pizza.py in 3 ways:

- to provide an interactive and scripting interface to the tools
- as a language for writing tools
- to wrap existing stand-alone codes

Python makes it easy for users of Pizza.py to:

- experiment with tools interactively
- automate tasks as script files of commands
- extend tools or create new ones

The topmost level of Pizza.py adds a modest bit of functionality to the Python interpreter to make it easier to invoke tools and pass data between them. As such, Python is an ideal "framework" or "glue" language that enables various tools to be hooked together, while also providing a rich programming environment of its own.

1.2 Open source distribution

Pizza.py comes with no warranty of any kind. As each source file states in its header, it is distributed free-of-charge, under the terms of the GNU Public License (GPL). This is often referred to as open-source distribution - see www.gnu.org or www.opensource.org for more details. The legal text of the GPL is in the LICENSE file that is included in the Pizza.py distribution.

Here is a summary of what the GPL means for Pizza.py users:

(1) Anyone is free to use, modify, or extend Pizza.py in any way they choose, including for commercial purposes.

(2) If you distribute a modified version of Pizza.py, it must remain open-source, meaning you distribute it under the terms of the GPL. You should clearly annotate such a code as a derivative version of Pizza.py.

(3) If you release any code that includes Pizza.py source code, then it must also be open-sourced, meaning you distribute it under the terms of the GPL.

(4) If you give Pizza.py to someone else, the GPL LICENSE file and source file headers (including the GPL notices) should remain part of the code.

In the spirit of an open-source code, these are various ways you can contribute to making Pizza.py better. You can send email to sjplimp@sandia.gov on any of these items.

- If you write a Pizza.py script that is generally useful or illustrates how to do something cool with Pizza.py, it can be added to the Pizza.py distribution. Ditto for a picture or movie that can be added to the Pizza.py WWW site.
- If you add a new method to one of the tools or create a new tool that is useful for Pizza.py or LAMMPS or ChemCell users, it can be added to the Pizza.py distribution. See the ToDo list at the beginning of the src/*.py files for features that haven't yet been implemented.
- If you find a bug, report it.
- Report if you find an error or omission in the Pizza.py documentation or on the Pizza.py WWW Site, or have a suggestion for something to clarify or include.
- Point prospective users to the Pizza.py WWW Site or link to it from your WWW site.

1.3 Acknowledgements

Pizza.py has been developed at Sandia National Laboratories which is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Funding for Pizza.py development has come from the US Department of Energy (DOE), through its LDRD and Genomes-to-Life programs. The latter effort has been funded by DOE's OASCR and OBER offices as part of the US Department of Energy's Genomics:GTL program (www.doegenomestolife.org) under the project, "Carbon Sequestration in Synechococcus Sp.: From Molecular Machines to Hierarchical Modeling".

The maintainer of Pizza.py is Steve Plimpton.

Matt Jones, a BYU student who was a summer intern at Sandia, wrote several of the coolest tools in Pizza.py and about half the code in the initial version.

Others who have written tools or scripts that are part of the Pizza.py distribution are listed on the Pizza.py WWW site.

2. Installing Pizza.py

Unpack the Pizza.py distribution by typing the following:

```
gunzip pizza.tar.gz
tar xvf pizza.tar
```

The Pizza.py main directory should then contain the following directories:

README	initial instructions
LICENSE	the GNU open-source license
doc	HTML documentation for Pizza.py
examples	scripts and data sets that exercise Pizza.py tools
scripts	various Pizza.py script files
src	source code for Pizza.py and its tools

Because Python is an interpreted language, there is no need to compile or "make" Pizza.py. You run Pizza.py by executing the src/pizza.py file directly, as described below. However there are 3 issues to consider: (a) you must have Python installed on your machine, (b) some Pizza.py tools require your Python to be extended with additional Python packages, and (c) some Pizza.py tools are wrappers on other software which needs to be available on your system.

If you don't plan to use a particular Pizza.py tool, you don't need to install additional Python packages or other software it requires.

Note that we cannot provide help on installing the various software packages described here. If you have problems, you'll need to talk to a local expert who can help you with your machine. If you find that instructions on this page are incorrect or incomplete or you can provide a better description of the install procedure, please send an email to sjplimp@sandia.gov.

Installing Python

Python is open-source software available for Unix, Macintosh, and Windows machines. If you have a Linux box or Mac it is probably already installed. If the python executable is in your path, typing "python" should give you a Python prompt ">>>" and tell you what version you are running. Version 2.4 or newer is sufficiently current to run Pizza.py, though older versions may work as well.

If Python is not installed on your machine, go to www.python.org to download a binary or source-code version and then build and/or install it.

On my Linux box, this was as easy as

```
./configure; make; sudo make install
```

To use the Pizza.py tools that have GUIs, you need to insure your Python supports Tcl/Tk via the Tkinter module. You can test if this is the case by typing the following from your Python prompt:

```
>>> import Tkinter
>>> Tkinter._test()
```

If this fails, see further instructions below for Tkinter.

IMPORTANT NOTE: If you are installing a new version of Python, read the Tkinter section first, since it describes how to make sure the Tcl/Tk and Togl libraries are installed. If you want to use the Pizza.py tools that require them, you need to do this step before building Python.

Running Pizza.py

Typically Pizza.py should be run from the directory where your LAMMPS or other data files are. Like any Python program, Pizza.py can be run in one of 2 ways, by typing either

```
python -i ~/pizza/src/pizza.py
~/pizza/src/pizza.py
```

where the last argument is the full pathname of the pizza.py file.

The -i switch leaves Python in interactive mode (instead of exiting) after the pizza.py script is run. In the 2nd case, if the src dir is in your path, just pizza.py could be typed. For the 2nd case, you must also set src/pizza.py to be an executable file (chmod +x pizza.py) and edit the 1st line of pizza.py to reflect where Python lives on your system (find it by typing "which python"), e.g.

```
#!/usr/local/bin/python -i
```

Putting a definition like one of the following in your .cshrc file (or equivalent syntax for other Unix shell start-up files) will let you simply type "pizza" at the command-line to start Pizza.py.

alias pizza python -i ~/pizza/src/pizza.py alias pizza ~/pizza/src/pizza.py

Pizza.py accepts several command-line options; they are described in this section of the documentation.

When Pizza.py starts, it reads a few values from the src/DEFAULTS.py file (described below) and imports the *.py files from the src directory as Python modules. These are the Pizza.py tools. Error messages will be printed if your Python has not been extended with a Python package that a tool requires. If you don't plan to use the tool you can ignore the message, or exclude that tool via the command-line switch "-x".

Once all tools have been loaded and any initial scripts and commands have been run (via command-line arguments) you should see the Pizza.py ">" prompt. From this point on, everything you type is a Python command. Python interprets what you type, operates on your data, and produces output or error messages, just as if you were typing in response to Python's interactive prompt ">>>".

You can also type special commands that have been added to the Python interpreter by Pizza.py or commands that invoke Pizza.py tools. More details about these options are explained in this section of the documentation.

As with Python, type Ctrl-D to exit Pizza.py at any time.

Setting up the DEFAULTS.py file

When Pizza.py starts it reads 3 values from the src/DEFAULTS.py file:

PIZZA_TOOLS	directories that contain additional Pizza.py tools
PIZZA_SCRIPTS	directories that contain additional Pizza.py scripts
PIZZA_EXCLUDE	Python files that are not loaded, since they are not tools

These are designed to allow users to augment Pizza.py with their own tools and scripts, which need not be stored in the directories of the Pizza.py pacakge. Follow the instructions in the DEFAULTS.py file for using these options.

The DEFAULTS.py files also contains various variables that specify the name and path of programs that Pizza.py tools will execute on your system. In some cases the variables contain settings that are used by these programs. Read the comments in the DEFAULTS.py file for more information.

The following table lists the keywords in the DEFAULTS.py, the program or setting that will be used by default if the keyword line is commented out, and the Pizza.py tools that use the keyword. If the program is not in your path or you wish to use an alternate program or setting, you must edit the DEFAULTS.py file accordingly. If you don't plan to use any tool that needs the keyword, you can ignore its setting.

Keyword	Default Value	Purpose	Tools that Use it
DISPLAY	display	display image files (ImageMagick)	rasmol, raster, svg
CONVERT	convert	convert image files (ImageMagick)	image
MONTAGE	montage	montage image files (ImageMagick)	image
GNUPLOT	gnuplot	Gnu Plotting package	gnu
GNUTERM	x11	GnuPlot terminal	gnu
GUNZIP	gunzip	unzip a compressed *.gz file	dump, log
LABEL3D	label3d	put a label on a Raster3D image	raster
MATLAB	matlab	MatLab numerical analysis & plotting package	matlab
RASMOL	rasmol	RasMol molecular vizualization package	rasmol
RENDER	render	Raster3D vizualization rendering engine	raster
VMDNAME	vmd	VMD visualization package	vmd
VMDDIR	/usr/local/lib/vmd	VMD visualization package	vmd
VMDDEV	win	VMD visualization package	vmd
VMDARCH	LINUX	VMD visualization package	vmd

Installing additional Python packages

This is the list of extra Python packages various Pizza.py tools require. If a tool is not listed it requires no extra packages. Instructions on where to find the Python extensions and how to install them are listed below.

Package	Tools that Use it
Numeric or Numpy	dump, mdump, bdump, ldump, tdump
pexpect	vmd
PIL	animate, gl, image
Pmw	image
PyOpenGL	gl, vcr

readline	Pizza.py itself
Tkinter and Togl	animate, image, plotview, vcr

Numeric or Numpy

Numeric and its follow-on NumPy enables Python to process vectors and arrays of numbers efficiently, both in terms of memory and CPU speed. It's an extremely useful extension to have in your Python if you do any numerical work on large data sets. Pizza.py can use either Numeric or NumPy.

If Numeric or NumPy is already installed in your Python, you should be able to type one of the following without getting an error:

```
>>> import Numeric
>>> import numpy
```

Numeric can be downloaded from this site on SourceForge and NumPy from this site. Numeric version 24.2 is fine for Pizza.py as is a current version of NumPy. Once unpacked, you can type the following from the Numeric or NumPy directories to install it in your Python.

```
sudo python setup.py install
```

Note: on my Linux box, when Numeric installed itself under the Python lib in /usr/local, it did not set all file permsissions correctly to allow a user to import it. So I also needed to do this:

```
sudo chmod -R 755 /usr/local/lib/python2.5/site-packages/Numeric
```

pexpect

Pexpect in a pure Python implementation of the Expect capability of the Tcl language. It allows Python to send commands to another program, and handshake the interaction between them, one command at a time.

If pexpect is already installed in your Python, you should be able to type the following without getting an error:

```
>>> import pexpect
```

Pexpect can be downloaded from this site. As of Nov 2010, Version 2.4 is fine. On my Linux box, this command installed it:

```
sudo python setup.py install
```

PIL

The PIL (Python Imaging Library) allows Python to read image files, manipulate them, and convert between several common image formats.

If PIL is already installed in your Python, you should be able to type the following without getting an error:

```
>>> import Image,ImageTk
```

PIL can be downloaded from this site. As of July 2007, Version 1.1.6 is fine. On my Linux box, this command installed it:

```
sudo python setup.py install
```

with a notice that Tkinter and ZLIB (PNG) support were enabled. If you want the Pizza.py tools to recognize other image formats (e.g. JPEG), then look at the README for further details, if the PIL build did not find the correct libraries.

Pmw

Pmw (Python mega-widgets) is a common Tkinter extension that provides a host of more powerful GUI widgets.

If Pmw is already installed in your Python, you should be able to type the following without getting an error:

```
>>> import Pmw
```

Pmw can be downloaded from this site. As of July 2007, Version 1.2 is fine. On my Linux box, these commands installed it:

```
cd Pmw.1.3.2/src
sudo python setup.py install
```

PyOpenGL

The PyOpenGL package is a wrapper on the ubiquitous OpenGL graphics library and allows a Python program to make graphics calls in standard OpenGL syntax. It also includes Togl support for opening a Tk OpenGL widget, assuming your Python has Tkinter intstalled and that Tkinter was built with Togl. The Pizza.py tools that use PyOpenGL require this capability.

If PyOpenGL is already installed in your Python, you should be able to type the following without getting an error:

```
>>> import OpenGL
```

If your PyOpenGL supports Togl, you should be able to type the following without getting an error:

```
>>> from OpenGL.Tk import *
>>> from OpenGL.GLUT import *
```

PyOpenGL can be downloaded from this site on SourceForge. You want the latest PyOpenGL release (not OpenGLContext). As of July 2007, Version 3.0.0a6 is what I used.

IMPORTANT NOTE: I had many problems installing earlier versions of PyOpenGL on various boxes. But the 3.0 release was much easier to install on my Linux box, as outlined here. Note that version 3.0 requires Python 2.4 or later.

IMPORTANT NOTE #2: By default, your Python may or may not have Tkinter installed. Even if Tkinter is installed, it probably was not built with Togl. If this is the case, you should install them first before installing PyOpenGl, otherwise you may get errors when you try to use the Pizza.py tools that use PyOpenGL, because they require Tkinter and Togl. See the Tkinter section below for instructions on installing Tkinter and Togl in your Python.

Once Tcl/Tk and Togl were in place, Installing PyOpenGL on my Linux box was as simple as:

```
sudo python setup.py install
```

(Sep 2012) A user reports, you can also then do this:

cd src sudo python togl.py

which should enable Togl with PyOpenGL

---

readline

The readline library is part of Python but is not supported on all systems. If it works in your Python, then Pizza.py (and Python) prompts are more shell-like and should support arrow keys, Emacs-style editing, command history, etc. If you get an error "No module named readline" when Pizza.py starts up, you can ignore it, or comment out the line "import readline" in src/pizza.py.

If readline is already installed in your Python, you should be able to type the following without getting an error:

```
>>> import readline
```

The readline library can be downloaded from this site. After building the library, I believe you then have to rebuild Python itself.

Tkinter and Togl

The Tkinter package is part of Python but is not always enabled when Python is built, typically due to not finding the Tcl/Tk libraries. If you can type the following without an error message in your Python, then Tkinter is operational in your Python:

```
>>> import Tkinter
>>> Tkinter._test()
```

If this fails, you need to rebuild Python and either insure it finds the Tcl/Tk libraries or build those libraries first as described here. Since I wanted a more current Python then was installed with Linux on my box, I download the latest Python (Python 2.5.1 as of July 2007) and installed it under /usr/local.

IMPORTANT NOTE: If you want to use any of the Pizza.py tools that use PyOpenGL, you also need the Togl library installed in the same directory as your Tcl/Tk libraries, so that Tcl/Tk can find it when it runs. Thus you should also install Togl, after building the Tcl/Tk libraries, and do both before re-building Python, as outlined in the following 3 steps.

(1) Building Tcl/Tk libraries

You can download the latest versions of Tcl and Tk here:

As of Oct 2011, version 8.5 is fine. After unpacking the two tarballs I did the following:

```
cd tcl8.5.10/unix
./configure; make; sudo make install
cd tk8.5.10/unix
./configure; make; sudo make install
```

(2) Installing Togl

To intstall Togl, download it from this site on SourceForge. As of Oct 2011, Version 2.0 is fine.

After unpacking the tarball, try typing the following to install it:

./configure

The first time I did this, it complained about not finding the Tcl and Tk strings it needed to point to Tcl/Tk. I fixed this by installing the tcl-devel package via yum.

I then noticed that "configure" was finding the system Tcl/Tk instead of the one I had installed. So I typed this instead:

```
configure --with-tcl=/usr/local/lib/tcl8.5 --with-tk=/usr/local/lib/tk8.5
```

You should also make sure that "configure" prints a "prefix" string that points to the correct directory where you want Togl installed. For example, in my case, I did not want it installed in /usr/lib to use with the system Python, but in /usr/local/lib to be loaded by my own installed Python version.

Using the with Tcl/Tk options on the configure command fixed this as well.

(3) Rebuilding Python

Now you can re-build Python (as described above) and it should find the correct Tcl/Tk libraries under /usr/local, so that you have Tkinter and Togl working with your Python as well. Note that when you run ./configure as the first step in building Python, it will tell you what it found for Tcl/Tk.

Installing other software

Some Pizza.py tools invoke other software which must be installed on your system for the tool to work. This is an alphabetic list of the needed software. Except where noted, it is freely available for download on the WWW. The Pizza.py tools that use this software are listed above in this section. To see if you already have the software on your box, type "which command", e.g. "which display".

ImageMagick display, convert, montage commands

Several Pizza.py tools display image files. The ImageMagick "display" program can be used for this purpose. Likewise, the ImageMagick "convert" and "montage" comands are used by the image tool. The ImageMagick toolkit can be downloaded from this site and contains a variety of useful image conversion and manipulation software.

GnuPlot

The Pizza.py gnu tool is a wrapper on the open-source GnuPlot program. GnuPlot can be downloaded from this site.

Gunzip

Gunzip is invoked by Python to read compressed (*.gz) data and dump files. It is almost certainly on your Unix system. If not if can be downloaded from this site.

Label3d

The Pizza.py tool raster uses the label3d and render programs from the Raster3d visualization package to produce high-quality ray-traced images. See the description of "Render" below for information about Raster3d.

MatLab

The Pizza.py matlab tool is a wrapper on MatLab which is a widely-used commercial numerical analysis package that also produces nice plots. Further information is available at the MathWorks WWW site. When MatLab is installed on your system and the appropriate environment variables are set, the command "matlab" should launch the program.

RasMol

The Pizza.py rasmol tool invokes the RasMol visualization package to view molecular systems and produce nice images. RasMol can be downloaded from this site, which is for the original open-source version of RasMol, not the Protein Explorer derivative version of RasMol.

Note that when using RasMol on a Mac, you will need to launch X11 first (or run Pizza.py from an X11 xterm) to get RasMol to display properly.

Render

The Pizza.py tool raster uses the render and label3d programs from the Raster3d visualization package to produce high-quality ray-traced images. Raster3d can be downloaded from this site.

For Macs, Raster3d is available for download via Fink as an unstable package.

VMD

The Pizza.py vmd tool is a simple wrapper on the VMD visualization package which is a popular tool for visualizing the output of molecular dynamics simulations. VMD can be downloaded from this site.

3. Basics of using Pizza.py

The previous section describes how to install and run Pizza.py and the various software it uses. After Pizza.py has started you should see a ">" prompt. The following sections describe what comes next:

3.1 Python syntax

Aside from its tools, Pizza.py itself simply adds a bit of functionality to the Python interpreter to enable it to more easily launch shell commands and scripts and invoke its tools. Pizza.py's ">" prompt is different from Python's ">>>" prompt to indicate the extra functionality is available, but you can type any Python command you would type at the Python prompt.

Python is a powerful scripting and programming language, similar in scope and universality to Perl. This little Pizza.py manual cannot attempt to teach you how to use Python, its syntax, or its rich set of powerful built-in commands. If you only use the extra tools provided by Pizza.py, you can think of Pizza.py as an application with a self-contained set of commands. However, if you learn more about Python, you will be able to write more powerful Pizza.py scripts, access and manipulate data stored inside Pizza.py tools, or even add your own commands and tools, which need not have anything to do with LAMMPS or ChemCell.

You can learn about Python at www.python.org. My most-used Python book is Essential Python by Dave Beazley which assumes some programming experience but covers both the basics of Python and its many powerful libraries in a well-written, concise manner.

3.2 Pizza.py command line arguments

When running Pizza.py, several command-line options can be added as switches, e.g.

```
pizza.py switch args switch args ...
```

-s	silent (else print start-up help)
-t log dump raster	load only these tools
-x raster rasmol	load all tools except these
-f file arg1 arg2	run script file with args
-c "vec = range(100)"	run Python command
-q	quit (else interactive)

Switches can appear in any order and be used multiple times. The -f scripts and -c commands are executed in the order they appear. Script files are Python files which can contain Python or Pizza.py tool commands. Pizza.py looks for script files in 3 places: your current working directory, the pizza/scripts directory, and any extra directories you list in the src/DEFAULTS.py file. This means you can add your own scripts to pizza/scripts or to directories of your choosing.

Note that the arguments of the -f switch file (arg1,arg2,etc) cannot begin with a single "-" or they will be interpreted as arguments to Pizza.py. They can however begin with a double "--".

Also note that the argument of the -c switch will typically need to be enclosed in quotes to avoid being interpreted by the shell. This also allows multiple Python commands to be separated by semi-colons, e.g.

```
-c "a = range(100); print a"
```

3.3 Pizza.py extensions to the Python interpreter

As mentioned above, the standard Python syntax is extended a bit at the Pizza.py ">" interactive prompt. These options were inspired by the LazyPython.py code of Nathan Gray, which taught me how to extend the Python interpreter. These are the short-cuts:

?	print help message
??	one-line for each tool and script
? raster	list tool commands or script syntax
?? energy.py	full documentation of tool or script
!ls -l	shell command
@cd ..	cd to a new directory
@log tmp.log	log all commands typed so far to file
@run block.py arg1 arg2	run script file with args
@time d = dump("*.dump")	time a command

Shell commands begun with a "!" can include the redirection operators "". The shell command "!cd" will not change directories permanently; use the "@cd" short-cut instead. Any short-cut command starting with "@" can be abbreviated with one or more letters. E.g. "@r" is the same as "@run". The @log command requires that the Python readline library be available on your system.

Each of the above short-cuts can be performed by native Python commands; they are just not as simple to type. Here is how several of the short-cuts can be written in Python, which is what you need to do in a script, since the above short-cuts only work at the Pizza.py interactive prompt:

Short-cut	Native Python
!ls -l	sys.command("ls -l")
@cd ..	os.chdir("..")
@run myfile.py	execfile("myfile.py")
CTRL-D	sys.exit()

3.4 Using Pizza.py tools

The tools that Pizza.py adds to Python are each implemented as a single Python class (e.g. dump, log, raster), so the first step in using a tool is to create an instance of the class (an object). Each class defines a set of methods (functions) that operate on the objects you create and their associated data. Each method, including the constructor, takes zero or more arguments, which may be previously created objects. In practical terms, this means that you type commands like this:

```
d = dump("dump.*")
p = pdb("my.pdb",d)
p.many()
dnew = dump("dump.all")
```

The first 2 commands create dump and pdb objects named "d" and "p" respectively. The "d" and "p" are Python variable names; you could use any names you wish: "dump12" or "Dump_mine" or whatever. The 3rd line invokes the "many" method within the pdb class for the pdb object "p". This method writes out a series of PDB files using the snapshots in "d" which was passed to "p" when it was created. The final command creates a new dump object "dnew" from another dump file. You can create and manage as many objects (of the same or different classes) simultaneously as you wish. If the last line assigned the object to "d", the original dump object with the same name would be deleted by Python.

Various Pizza.py tools create temporary files as they operate. These are all named tmp.*. Pizza.py does not clean up all of these files, since they are sometimes useful to look at for debugging or other purposes.

Python syntax allows for powerful combinations of tools to be invoked in one or a few commands. For example

```
lg = log("log.*")
m = matlab()
plotview(lg,m)
```

could be abbreviated as

```
plotview(log("log.*"),matlab())
```

With the -c command line switch, this one-liner could be specified when Pizza.py is launched. This example also illustrates that created objects (like the plotview object) do not need to be assigned to variables if they will not be accessed in subsequent commands.

3.5 Running Pizza.py and Python scripts

A file containing Python and/or Pizza.py commands can be executed as a script and arguments can be passed to it (if desired). The script can be run in several different ways:

(1) From the Pizza.py command line

```
% pizza -f script.sample file.test 10 ...
```

(2) From the Pizza.py interactive prompt

```
> @run script.sample file.test 10 ...
```

(3) From the Python command line

```
% python -i script.sample file.test 10 ...
```

(4) From a shell prompt with #!/usr/local/bin/python -i as 1st line of script

```
% script.sample arg1 arg2 ...
```

(5) From the Python interactive prompt

```
>>> argv = 0,"file.test","10",...
>>> execfile("script.sample")
```

(6) As a nested script from within another Python or Pizza.py script file

```
argv = 0,"file.test","10",...
execfile("script.sample")
```

The Pizza.py interpreter short-cut commands described in the next section cannot be used in a script file.

There are 2 additional issues to address in your script files.

(A) First, if the script uses Pizza.py commands and you want to run it from Python itself (methods 3,4,5,6), then your script should import the necessary Pizza.py tools directly. E.g. if your script uses the log and matlab tools, you would put these lines at the top:

```
from log import log
from matlab import matlab
```

This is OK to do even if the script will be run by Pizza.py since it doesn't matter that Pizza.py already imported the tools. Note that if you do this, you can then give your script file and the Python tool *.py files it uses to someone who doesn't have Pizza.py and they can run your script with their Python.

(B) Second, if your script takes arguments and you want the same script to run identically for all 6 methods, then you need to include this line at the beginning of the script:

```
if not globals().has_key("argv"): argv = sys.argv
```

This will enable the arguments to be accessed in the script as argv**1** for the 1st argument, argv**2** for the 2nd, etc.

This works because in methods 3,4 Python stores the script arguments in sys.argv and the script name in sys.argv**0**. The above line of Python code copies sys.argv to argv. When Pizza.py runs the script (methods 1,2) it loads the arguments directly into the "argv" variable. Methods 5,6 load the arguments into argv explicitly before executing the script via execfile(). In this case argv**0** is a dummy argument to conform with the Python convention for sys.argv.

Also note in methods 5,6 that all arguments such as "10" must be strings even if they are numeric values, since this is the way they are passed to the script in methods 1,2,3,4.

As an example of the flexibility enabled by combining scripts, arguments, and command-line options in Pizza.py, consider the 3-line example of the previous sub-section. We modify the script as follows and save it as logview.py:

```
files = ' '.join(argv1:)   # create one string from list of filenames
lg = log(files)
m = matlab()
plotview(lg,m)
```

If an alias is defined in your shell start-up file, such as

```
alias logview ~/pizza/src/pizza.py -f logview.py
```

then you can type the following one-liner at the shell prompt to invoke Pizza.py on the logview.py script with a list of files you specfiy.

```
% logview log.1 log.2 ...
```

A set of plots and a control GUI will appear on your screen.

3.6 Error messages

If you mistype a Pizza.py or Python command or pass an invalid argument to a tool method, an error message will be printed by Python. Usually these will be self-explanatory. Sometimes they will point to a line of code inside a tool which Python was unable to execute successfully. This could be because you passed the wrong arguments to the tool, the data the tool is operating on is invalid, or because there's a bug in the tool. In the latter case, please figure out as much as you can about the bug and email a description and the necessary files to reproduce the bug in the simplest possible way to sjplimp@sandia.gov.

4. Tools within Pizza.py

Th previous section describes how Pizza.py tools are used in Pizza.py.

Help on the syntax for invoking a tool and using its methods and settings can be accessed interactively within Pizza.py itself by typing "? tool" or "?? tool". Typing "??" gives a one-line description of each tool.

These are the different categories of Pizza.py tools:

LAMMPS in/out files	chain, data, dump, log, patch, bdump, ldump, tdump
ChemCell in/out files	cdata, olog, dump
SPPARKS in/out files	olog, dump
SPARTA in/out files	sdata, olog, dump
Visualization	gl, rasmol, raster, svg, vmd
File conversion	cfg, ensight, pdbfile, vtk, xyz
GUI wrappers	animate, image, plotview, vcr
Plotting	gnu, matlab
Miscellaneous	histo, mdump, pair, vec

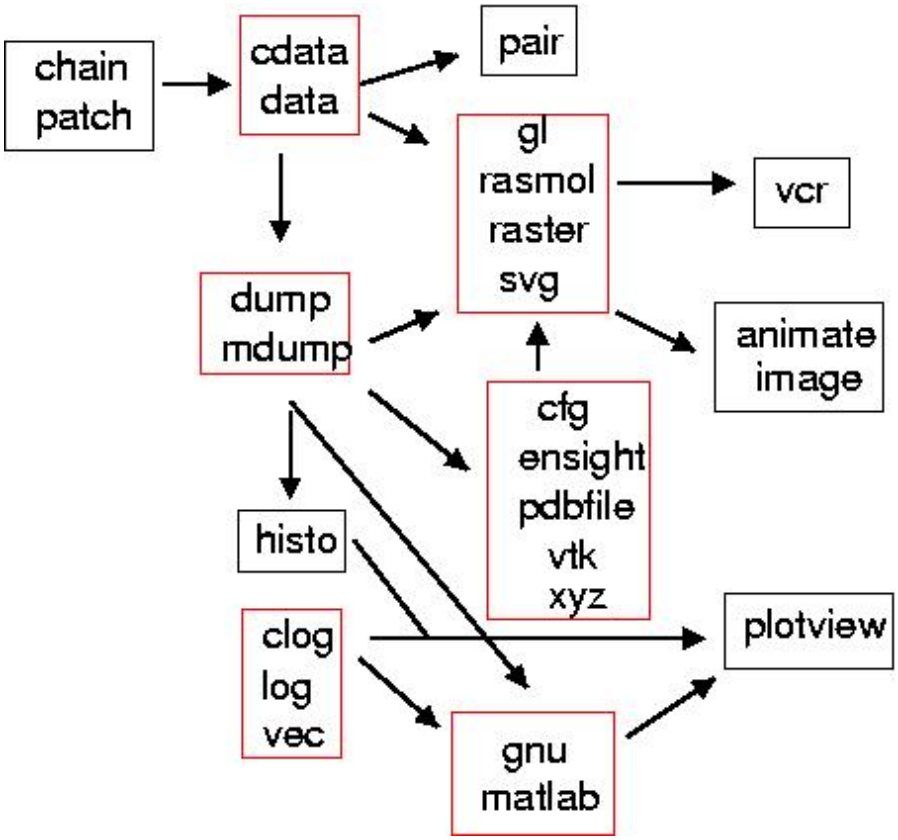
Within the plotting and viz categories, individual tools share many common methods, so the tools can often be used interchangeably. For example, the same script can produce an animation using either Raster3d or SVG to generate the movie frames, by simply changing the line that creates the visualizer object, or by passing the object into the script as an argument.

This is the complete list of tools in Pizza.py; the link is to each tool's documentation page.

animate.py	Animate a series of image files
bdump.py	Read LAMMPS dump local files with bond info
cdata.py	Read, write, manipulate Chemcell data files
chain.py	Create bead-spring chains for LAMMPS input
cfg.py	Convert LAMMPS snapshots to CFG format
data.py	Read, write, manipulate LAMMPS data files
dump.py	Read, write, manipulate dump files and particle attributes
ensight.py	Convert LAMMPS snapshots to Ensight format
gl.py	3d interactive visualization via OpenGL
gnu.py	Create plots via GnuPlot plotting program
histo.py	Particle density histogram from a dump
image.py	View and manipulate images
ldump.py	Read LAMMPS dump files with line info
log.py	Read LAMMPS log files and extract thermodynamic data
matlab.py	Create plots via MatLab numerical analysis program
mdump.py	Read, write, manipulate mesh dump files
olog.py	Read other log files (ChemCell, SPPARKS, SPARTA) and extract time-series data
pair.py	Compute LAMMPS pairwise energies
patch.py	Create patchy Lennard-Jones particles for LAMMPS input

pdbfile.py	Read, write PDB files in combo with LAMMPS snapshots
plotview.py	Plot multiple vectors from a data set
rasmol.py	3d visualization via RasMol program
raster.py	3d visualization via Raster3d program
sdata.py	Read, write, manipulate SPARTA surface files
svg.py	3d visualization via SVG files
tdump.py	Read LAMMPS dump files with triangle info
vcr.py	VCR-style GUI for 3d interactive OpenGL visualization
vec.py	Create numeric vectors from columns in file or list of vecs
vtk.py	Convert LAMMPS snapshots to VTK format
vmd.py	Wrapper on VMD visualization package
xyz.py	Convert LAMMPS snapshots to XYZ format

This diagram represents the different ways tools can be interconnected by Pizza.py. Tools within the same red box are tools that are (roughly) interchangeable.



5. Example scripts

The Pizza.py distribution includes 2 sets of example scripts. A listing of included scripts is given on the Pizza.py WWW site.

(A) The examples directory has a README file and a variety of scripts. For each tool in Pizza.py there is a test_tool.py script that invokes the tool on a simple data set (from the files sub-directory) and exercises various tool methods. These scripts are meant to illustrate how the tool is used.

Assuming pizza.py is in your path, each of the test scripts can be run from within the examples directory by typing one of these lines (from the shell or from within Pizza.py):

```
% pizza.py -f test_animate.py
> @run test_animate.py
```

The remaining scripts in the examples directory illustrate how to do various tasks in Pizza.py. They are not meant to be generically useful nor are they well documented. Rather they are illustrations of how to do some specific task quickly and straightforwardly. Most of these scripts are not meant to be run by other users, since their data files are not included.

(B) The scripts directory contains several scripts you may find useful either to use directly or to modify to create a new script for your purposes.

The top of each script file describes its purpose and syntax. That information can be be accessed from within Pizza.py by typing "??" or "? name.py" or "?? name.py".

As explained in this section, any file in the scripts directory can be run from the directory where your data lives, by typing a line appropriate to the script's syntax, e.g.

```
% pizza.py -f movie.py svg 60 135 dump.protein      from the shell
> @run movie.py svg 60 135 dump.protein            from Pizza.py
```

6. Extending Pizza.py

Pizza.py can easily be extended in several ways:

1. fix a bug
2. make a tool method faster or cleaner
3. add a new method or setting to a tool
4. add a new tool
5. add a generically useful script to the scripts dir
6. add a script that does something interesting to the examples dir
7. send a picture or movie you made with Pizza.py for the WWW page.

You might be able to do (2) because you're better with Python than we are! Note that generally, we've opted for simplicity versus speed in writing the tools, unless the operation is very costly. An example of something I'd like to speed up is the reading of large dump files in dump.py.

Some of the ideas we've had for (3), but haven't gotten around to, are listed at the top of the src/*.py files as ToDo items.

If you think your addition will be useful to other Pizza.py users, email it to sjplimp@sandia.gov and it can be added to the distribution with an attribution to you, both in the source code and on the Pizza.py WWW site.

Here are ideas to consider when creating new Pizza.py tools or scripts:

(1) For tools, your *.py file should contain a Python class with the same name as the *.py file since it will be imported into Pizza.py with a statement like

```
from dump import dump
```

(2) Your scripts can use methods from Pizza.py classes to make data analysis easier. E.g. scripts can be written that use the dump tool to read dump files, then use the iterator calls and vecs() method from dump.py to loop over snapshots and extract lists of particles for further computation. See the scripts and examples directories for examples of this.

(3) To flag an error in your script or tool and exit back to the Pizza.py prompt, use a line like:

```
raise StandardError,"error message"
```

(4) Document your tool be defining the "oneline" and "docstr" variables at the top of the file. This is what will be printed when you type "? dump", for example.

(5) If you intend your tool to interact with other Pizza.py tools, you should follow the Pizza.py philosophy of having objects be passed as arguments to the tool methods. If you're creating a tool that is similar to an existing tool, but a different flavor (e.g. wrapping another plotting package in addition to MatLab or GnuPlot), then try to make the interface similar to the others so all the related tools can be used interchangeably.

(6) From the Pizza.py perspective, the difference between a script and a tool is as follows. A script typically does a specific operation (with or without arguments). E.g. process a dump file and compute a quantity. A tool version of the same operation would allow it to store internal state and persist, or to be packaged in a way that other tools or scripts can create instances of it and call its methods. From the Python perspective the code for the 2 cases may

not look very different. The tool version might just be some Python variables and methods stuck inside a Python class, which is what a Pizza.py tool basically is.

(7) The various Pizza.py tools are mostly related to the "LAMMPS" molecular dynamics or ChemCell cell simulator packages. But of course you can write Pizza.py tools that have nothing to do with LAMMPS or ChemCell. If you think they will still be of general interest to Pizza.py users, you can send them to us to include in the Pizza.py distribution. Or you can keep the top-level pizza.py file, throw away the LAMMPS and ChemCell tool files, build your own toolkit for whatever application you wish, and use or even distribute it yourself. That's the open-source philosophy.

animate tool

Purpose:

Animate a series of image files.

Description:

The animate tool displays a GUI to view and animate a series of image files.

The animate constructor creates the GUI. The animation can be controlled by the GUI widgets or by invoking the tool methods: play(), stop(), next(), etc. The frame slider can be dragged to view a desired frame.

Image files can be in any format (PNG, GIF, BMP, etc) recognized by the Python Image Library (PIL) installed in your Python. Various Pizza.py visualization tools (raster, rasmol, etc) create such image files. If a particular image format fails to load, your PIL installation was linked without support for that format. Rebuild PIL, and follow its install instructions.

Usage:

```
a = animate("image*.png")      create GUI to animate set of image files
a = animate("image*.png",1)    2nd arg = sort filenames, 0 = no sort, def = 1

Actions (same as GUI widgets):

a.first()                      go to first frame
a.prev()                       go to previous frame
a.back()                       play backwards from current frame to start
a.stop()                       stop on current frame
a.play()                       play from current frame to end
a.next()                       go to next frame
a.last()                       go to last frame

a.frame(31)                    set frame slider
a.delay(0.4)                   set delay slider
```

Related tools:

gl, raster, rasmol, svg, vcr

Prerequisites:

Python Tkinter and PIL packages.

bdump tool

Purpose:

Read dump files with bond info.

Description:

The bdump tool reads one or more LAMMPS dump files, and stores their contents as a series of snapshots with 2d arrays of atom attributes. It is assumed that each entry contains info for a bond in a LAMMPS simulation as is typically written by the dump local command in LAMMPS. Other tools use bdump objects to extract bond info for visualization, like the dump tool via its extra() method.

The constructor method is passed a string containing one or more dump filenames. They can be listed in any order since snapshots are sorted by timestep after they are read and duplicate snapshots (with the same time stamp) are deleted. If a 2nd argument is specified, the files are not immediately read, but snapshots can be read one-at-a-time by the next() method.

The map() method assigns names to columns of attributes. The id,type,atom1,atom2 names must be assigned in order for bond info to be extracted.

The viz() method is called by Pizza.py tools that visualize snapshots of atoms (e.g. gl, raster, svg tools).

Normally, LAMMPS creates the dump files read in by this tool. If you want to create them yourself, the format of LAMMPS dump local files is simple. Each snapshot is formatted as follows:

```
ITEM: TIMESTEP
100
ITEM: NUMBER OF ENTRIES
32
ITEM: ENTRIES
1 1 5 10
2 1 11 45
3 2 6 8
...
N -3 23 456
```

There are N lines following "ITEM: ENTRIES" where N is the number of entries. Entries do not have to be listed in any particular order. There can be a different number of entries in each snapshot. Each line must contain the bond ID, type, and the 2 atom IDs of the atoms in the bond, as specified by the map() command.

Usage:

```
b = bdump("dump.one")             read in one or more dump files
b = bdump("dump.1 dump.2.gz")     can be gzipped
b = bdump("dump.*")               wildcard expands to multiple files
b = bdump("dump.*",0)             two args = store filenames, but don't read

  incomplete and duplicate snapshots are deleted
  no column name assignment is performed

time = b.next()                   read next snapshot from dump files
```

```
  used with 2-argument constructor to allow reading snapshots one-at-a-time
  snapshot will be skipped only if another snapshot has same time stamp
  return time stamp of snapshot read
  return -1 if no snapshots left or last snapshot is incomplete
  no column name assignment is performed

b.map(1,"id",3,"x")              assign names to atom columns (1-N)

  must assign id,type,atom1,atom2

time,box,atoms,bonds,tris,lines = b.viz(index)   return list of viz objects

  viz() returns line info for specified timestep index
    can also call as viz(time,1) and will find index of preceding snapshot
    time = timestep value
    box = NULL
    atoms = NULL
    bonds = id,type,atom1,atom2 for each line as 2d array
    tris = NULL
    lines = NULL
```

Related tools:

dump, gl, raster, svg

Prerequisites:

Numeric or NumPy Python packages. Gunzip command (if you want to read gzipped files).

cdata tool

Purpose:

Read, create, manipulate ChemCell data files.

Description:

The cdata tool reads and writes ChemCell data files which contain particle and surface information. It enables the creation of geometric models of cells for input to ChemCell.

The cdata constructor reads in the specified ChemCell data file. With no argument, an empty cdata object is created which can have objects added to it later, and then be written out.

A cdata object or file contains "objects" of different types. Each object has a unique user-assigned ID. Objects can be of several types: "group", "triangles", "region", "facets", or "lines". A group is a list of particles, all of the same type. Triangles define a surface and include a list of vertices, triangle definitions, and adjacent edge connections. A region is a geometric object that defines a surface, such as a sphere. Facets are a CUBIT meshing format that defines a set of vertices and triangles; it is converted into a triangles entry. Lines are a collection of line segments.

The box(), sphere(), shell(), cyl(), and cap() methods create a new "region" object. The q() method sets triangulations factors that are used when the region is converted into a triangulated surface for output or visualization.

The line() and ibox() methods create a "lines" object.

The surf() method creates a "triangles" object from a region, using the q() parameters. The surftri() method creats a new "triangles" object from a list of triangle indices belonging to another surface. The surfselect() method applies an if test to vertices and triangles in another surface to choose a subset of them to form a new "triangles" object. The bins() method sets the number of bins in x,y when surfaces are binned for the purpose of creating particles inside/outside the surface.

The part(), part2d(), partarray(), and partring() methods create a "group" of particles inside, outside, or on regions or triangulated surfaces. Particles are 3d by default, or 2d if created on a surface. The partsurf() method can be used to change the name of the surface 2d particles are on, since that attribute is written to a file when the particles are output. The creation of particles uses a random # generator whose initial seed can be set via the seed() method.

The random() method can be used to pick a random point on the surface of a "region" or "triangles" object. The project() method maps particles to the surface of a "region" or "triangulate" object.

The center(), trans(), rotate(), and scale() methods are used to perform a geometric transformation on a "group" of particles or a "triangles" object. The union() method creates a new object of "union" type from a list of objects. The join() method does the same thing except all objects in the list must be of the same type and the new object is also of that type. A join() can only be done for "group", "triangles", or "line" objects. The delete(), rename(), and copy() methods manipulate the IDs of previously defined objects.

Be default all objects are selected when created. The select() and unselect() methods can be used to select a subset of existing objects. The write() and append() methods write out selected objects to a file.

The iterator() and viz() methods are called by Pizza.py tools that visualize snapshots of atoms (e.g. gl, raster, svg tools). Only selected objects are returned to the caller. A cdata file can be visualzed similarly to a snapshots from a dump file. In the case of a cdata file, there is only a single snapshot with index 0.

Usage:

```
c = cdata()                          create a datafile object
c = cdata("mem.surf")                read in one or more ChemCell data files
c = cdata("mem.part.gz mem.surf")    can be gzipped
c = cdata("mem.*")                   wildcard expands to multiple files
c.read("mem.surf")                   read in one or more data files

  read() has same argument options as constructor
  files contain the following kinds of entries, each of which becomes an object
    particles, triangles, region, facets
    particles is a list of particles -> becomes a group
    triangles is 3 lists of vertices, triangles, connections -> becomes a surf
    region is a ChemCell command defining a region -> becomes a region
    facets is a CUBIT format of vertices and triangles -> becomes a surf
  each object is assigned an ID = name in file
  ID can be any number or string, must be unique

c.box(ID,xlo,ylo,zlo,xhi,yhi,zhi)    create a box region
c.sphere(ID,x,y,z,r)                 create a sphere region
c.shell(ID,x,y,z,r,rinner)           create a shell region
c.cyl(ID,'x',c1,c2,r,lo,hi)          create a axis-aligned cylinder region
c.cap(ID,'x',c1,c2,r,lo,hi)          create a axis-aligned capped-cylinder region
c.q(ID,q1,q2,...)                    set region triangulation quality factors

  box() can create an axis-aligned plane, line, or point if lo=hi
  cyl() can create an axis-aligned circle if lo=hi
  for cyl() and cap(): 'x' c1,c2 = y,z; 'y' c1,c2 = x,z; 'z' c,c2 = x,y
  q's are size factors for region triangulation
    for box, q1,q2,q3 = # of divisions per xyz of box
    for sphere or shell, q1 = # of divisions per face edge of embedded cube
    for cyl or cap, q1 = # of divisions per face edge of end cap, must be even
                    q2 = # of divisions along length of cylinder

c.line(ID,x1,y1,z1,x2,y2,z2)         create a line object with one line
c.lbox(ID,xlo,ylo,zlo,xhi,yhi,zhi)   create a line object with 12 box lines

c.surf(ID,id-region)                 create a triangulated surf from a region
c.surftri(ID,id-surf,t1,t2,...)      create a tri surf from list of id-surf tris
c.surfselect(ID,id-surf,test)        create a tri surf from test on id-surf tris
c.bins(ID,nx,ny)                     set binning parameters for a surf

  triangulation of a shell is just done for the outer sphere
  for surftri(), one or more tri indices (1-N) must be listed
  for surfselect(), test is string like "$x <2.0 and $y > 0.0"
  bins are used when particles are created inside/outside a surf

c.part(ID,n,id_in)                   create N particles inside object id_in
c.part(ID,n,id_in,id_out)            particles are also outside object id_out
c.part2d(ID,n,id_on)                 create 2d particles on object id_on
c.partarray(ID,nx,nz,nz,x,y,z,dx,dy,dz)   create 3d grid of particles
c.partring(ID,n,x,y,z,r,'x')                create ring of particles
c.partsurf(ID,id_on)                 change surf of existing 2d particle group
c.seed(43284)                        set random # seed (def = 12345)

  generate particle positions randomly (unless otherwise noted)
  for part(), id_in and id_out must be IDs of a surf, region, or union object
```

```
    inside a union object means inside any of the lower-level objects
    outside a union object means outside all of the lower-level objects
  for part2d(), id_on must be ID of a surf, region, or union object
  for part2d(), particles will be written as 2d assigned to surf id_on
  for partring(), ring axis is in 'x','y', or 'z' direction
  partsurf() changes surf id_on for an existing 2d particle group

x,n = c.random(ID)                 pick a random pt on surf of object ID
c.project(ID,ID2,dx,dy,dz,eps,fg)  project particles in ID to surf of obj ID2

  random() returns pt = [x,y,z] and normal vec n [nx,ny,nz]
  for random(), ID can be surf or region obj
  project() remaps particle coords in group ID
    moves each particle along dir until they are within eps of surface
    if no fg arg, dir = (dx,dy,dz)
    if fg arg, dir = line from particle coord to (dx,dy,dz)
    ID2 can be surf or region obj
    particles are converted to 2d assigned to surf ID2

c.center(ID,x,y,z)                 set center point of object
c.trans(ID,dx,dy,dz)               translate an object
c.rotate(ID,'x',1,1,0,'z',-1,1,0)  rotate an object
c.scale(ID,sx,sy,sz)               scale an object

  objects must be surface or particle group, regions cannot be changed
  for center(), default is middle of bounding box (set when obj is created)
  for rotate(), set any 2 axes, must be orthogonal, 3rd is inferred
    object is rotated so that it's current xyz axes point along new ones
  rotation and scaling occur relative to center point

c.union(ID,id1,id2,...)            create a new union object from id1,id2,etc
c.join(ID,id1,id2,...)             create a new object by joining id1,id2,etc
c.delete(id1,id2,...)              delete one or more objects
c.rename(ID,IDnew)                 rename an object
c.copy(ID,IDnew)                   create a new object as copy of old object

  for union, all lower-level objects must be of surface, region, or union style
  for join, all joined objects must be of same style: group, surf, line
    new object is the same style

c.select(id1,id2,...)              select one or more objects
c.select()                         select all objects
c.unselect(id1,id2,...)            unselect one or more objects
c.unselect()                       unselect all objects

  selection applies to write() and viz()

c.write("file")                    write all selected objs to ChemCell file
c.write("file",id1,id2,...)        write only listed & selected objects to file
c.append("file")                   append all selected objs to ChemCell file
c.append("file",id1,id2,...)       append only listed & selected objects

  union objects are skipped, not written to file

index,time,flag = c.iterator(0/1)          loop over single snapshot
time,box,atoms,bonds,tris,lines = c.viz(index)   return list of viz objects

  iterator() and viz() are compatible with equivalent dump calls
  iterator() called with arg = 0 first time, with arg = 1 on subsequent calls
    index = timestep index within dump object (only 0 for data file)
    time = timestep value (only 0 for data file)
    flag = -1 when iteration is done, 1 otherwise
```

```
viz() returns info for selected objs for specified timestep index (must be 0)
  time = 0
  box = [xlo,ylo,zlo,xhi,yhi,zhi]
  atoms = id,type,x,y,z for each atom as 2d array
    NULL if atoms do not exist
  bonds = NULL
  tris = id,type,x1,y1,z1,x2,y2,z2,x3,y3,z3,nx,ny,nz for each tri as 2d array
    regions are triangulated according to q() settings by viz()
    NULL if surfaces do not exist
  lines = id,type,x1,y1,z1,x2,y2,z2 for each line as 2d array
    NULL if lines do not exist
  types are assigned to each object of same style in ascending order
```

Related tools:

olog, data, gl, raster, svg

Prerequisites: none

cfg tool

Purpose:

Convert LAMMPS snapshots to AtomEye CFG format.

Description:

The cfg tool converts atom snapshots in a LAMMPS dump or data file to the CFG format used by the AtomEye visualization tool.

The cfg constructor takes an object that stores atom snapshots (dump, data) as its first argument. The atom snapshots must have "id", "type", "x", "y", and "z" defined; see the map() methods of those tools.

The one(), many(), and single() methods convert specific snapshots to the CFG format and write them out. Optionally, a file prefix for the CFG output files can also be specified. A ".cfg" suffix will be appended to all output files.

If your atom snapshots are not sorted by atom ID (e.g. because they were written out by a parallel LAMMPS run), then you may want to sort them before converting them to CFG files with this tool. This can be done by "d = dump("tmp.dump"); d.sort()". This is because AtomEye does not use atom IDs directly but infers an ID by the order of atoms as they appear in the CFG file.

Usage:

```
c = cfg(d)              d = object containing atom coords (dump, data)

c.one()                 write all snapshots to tmp.cfg
c.one("new")            write all snapshots to new.cfg
c.many()                write snapshots to tmp0000.cfg, tmp0001.cfg, etc
c.many("new")           write snapshots to new0000.cfg, new0001.cfg, etc
c.single(N)             write snapshot for timestep N to tmp.cfg
c.single(N,"file")      write snapshot for timestep N to file.cfg
```

Related tools:

data, dump, ensight, vtk, xyz

Prerequisites: none

chain tool

Purpose:

Create bead-spring chains for LAMMPS input.

Description:

The chain tool creates random, overlapping bead-spring (FENE) chains and writes them out as a LAMMPS data file. They need to be simulated with a soft potential in LAMMPS to un-overlap them before they form a proper melt.

The chain constructor uses the total number of monomers and the Lennard-Jones reduced density to create a simulation box of the appropriate size. Optionally, the box shape can also be specified.

The build() method creates N chains, each with M monomers. It can be invoked multiple times to create sets of chains with different properties. The starting point of each chain is chosen randomly, as is the position of subsequent monomers. The seed value sets the random number generator used for coordinate generation.

The mtype, btype, blen, and dmin settings affect how the chain and its monomers are created. Dmin is the minimum distance allowed between a new monomer and the monomer two before it, so it determines the stiffness of the chain. Each monomer is assigned a molecule ID as it is created, in accord with the id setting.

Once N total monomers have been created, the ensemble of chains is written to a LAMMPS data file via the write() method.

Usage:

```
c = chain(N,rho)            setup box with N monomers at reduced density rho
c = chain(N,rho,1,1,2)      x,y,z = aspect ratio of box (def = 1,1,1)

c.seed = 48379              set random # seed (def = 12345)
c.mtype = 2                 set type of monomers (def = 1)
c.btype = 1                 set type of bonds (def = 1)
c.blen = 0.97               set length of bonds (def = 0.97)
c.dmin = 1.02               set min dist from i-1 to i+1 site (def = 1.02)

c.id = "chain"              set molecule ID to chain # (default)
c.id = "end1"               set molecule ID to count from one end of chain
c.id = "end2"               set molecule ID to count from either end of chain

c.build(100,10)             create 100 chains, each of length 10

  can be invoked multiple times interleaved with different settings
  must fill box with total of N monomers

c.write("data.file")        write out all built chains to LAMMPS data file
```

Related tools:

data, patch

Prerequisites: none

data tool

Purpose:

Read, write, manipulate LAMMPS data files.

Description:

The data tool reads and writes LAMMPS data files. It also allows their content to be accessed or modified.

The data constructor reads in the specified LAMMPS data file. With no argument, an empty data object is created which can have fields added to it later, and then be written out.

The map() method assigns names to different atom attributes by their column number (1-N). The get() method extracts columns of information from the specified section of the data file.

The title, headers, and sections variables can be set directly. The header values correspond to one-line definition that appear at the top of the data file. The box size values should be set to a Python tuple, e.g.

```
d.headers["xlo xhi"] = (-30, 30)
```

The section value should be a list of text lines, each of which includes a newline at the end.

The delete() method deletes a header or entire section of the data file. The replace() method allows one column of a section to be replaced with a vector of new values. The newxyz() methods replaces the xyz coords of the "Atoms" section of the data file with xyz values from the Nth snapshot of a dump object containing snapshots.

The iterator() and viz() methods are called by Pizza.py tools that visualize snapshots of atoms (e.g. raster, svg tools). A data file can be visualzed similarly to snapshots from a dump file. In the case of a data file, there is only a single snapshot with index 0.

The write() method outputs a LAMMPS data file.

Usage:

```
d = data("data.poly")            read a LAMMPS data file, can be gzipped
d = data()                       create an empty data file

d.map(1,"id",3,"x")              assign names to atom columns (1-N)

coeffs = d.get("Pair Coeffs")    extract info from data file section
q = d.get("Atoms",4)

  1 arg = all columns returned as 2d array of floats
  2 args = Nth column returned as vector of floats

d.reorder("Atoms",1,3,2,4,5)     reorder columns (1-N) in a data file section

  1,3,2,4,5 = new order of previous columns, can delete columns this way

d.title = "My LAMMPS data file"  set title of the data file
d.headers["atoms"] = 1500        set a header value
```

```
d.sections["Bonds"] = lines      set a section to list of lines (with newlines)
d.delete("bonds")                delete a keyword or section of data file
d.delete("Bonds")
d.replace("Atoms",5,vec)         replace Nth column of section with vector
d.newxyz(dmp,1000)               replace xyz in Atoms with xyz of snapshot N

  newxyz assumes id,x,y,z are defined in both data and dump files
    also replaces ix,iy,iz if they are defined

index,time,flag = d.iterator(0/1)          loop over single data file snapshot
time,box,atoms,bonds,tris,lines = d.viz(index)   return list of viz objects

  iterator() and viz() are compatible with equivalent dump calls
  iterator() called with arg = 0 first time, with arg = 1 on subsequent calls
    index = timestep index within dump object (only 0 for data file)
    time = timestep value (only 0 for data file)
    flag = -1 when iteration is done, 1 otherwise
  viz() returns info for specified timestep index (must be 0)
    time = 0
    box = [xlo,ylo,zlo,xhi,yhi,zhi]
    atoms = id,type,x,y,z for each atom as 2d array
    bonds = id,type,x1,y1,z1,x2,y2,z2,t1,t2 for each bond as 2d array
      NULL if bonds do not exist
    tris = NULL
    lines = NULL

d.write("data.new")              write a LAMMPS data file
```

Related tools:

gl, raster, svg

Prerequisites: none

dump tool

Purpose:

Read, write, manipulate dump files and particle attributes.

Description:

The dump tool reads one or more LAMMPS dump files, stores their contents as a series of snapshots with 2d arrays of atom attributes, and allows the values to be accessed and manipulated. Other tools use dump objects to convert LAMMPS files to other formats or visualize the atoms.

The constructor method is passed a string containing one or more dump filenames. They can be listed in any order since snapshots are sorted by timestep after they are read and duplicate snapshots (with the same time stamp) are deleted. If a 2nd argument is specified, the files are not immediately read, but snapshots can be read one-at-a-time by the next() method.

The map() method assigns names to columns of atom attributes. The tselect() methods select one or more snapshots by their time stamp. The delete() method deletes unselected timesteps so their memory is freed up. This can be useful to do when reading snapshots one-at-a-time for huge data sets. The aselect() methods selects atoms within selected snapshots. The write() and scatter() methods write selected snapshots and atoms to one or more files.

The scale(), unscale(), wrap(), unwrap(), and owrap() methods change the coordinates of all atoms with respect to the simulation box size. The sort() method sorts atoms within snapshots by their ID or another specified column.

The set() method enables new or existing columns to be set to new values. The minmax() method sets a column to an integer value between the min and max values in another column; it can be used to create a color map. The clone() method copies that column values at one timestep to other timesteps on a per-atom basis.

The time(), atom(), and vecs() methods return time or atom data as vectors of values.

The iterator() and viz() methods are called by Pizza.py tools that visualize snapshots of atoms (e.g. gl, raster, svg tools). You can also use the iterator() method in your scripts to loop over selected snapshots. The atype setting determines what atom column is returned as the atom "type" by the viz() method. The bonds() method can be used to create a static list of bonds that are returned by the viz() method.

Normally, LAMMPS creates the dump files read in by this tool. If you want to create them yourself, the format of LAMMPS dump files is simple. Each snapshot is formatted as follows:

```
ITEM: TIMESTEP
100
ITEM: NUMBER OF ATOMS
32
ITEM: BOX BOUNDS
0 3.35919
0 3.35919
0 7.50
ITEM: ATOMS
1 1 0 0 0
2 1 0.25 0.25 0
3 1 0.25 0 0.25
```

```
...
N 3 0.7 0.5 0.6
```

The box bounds are listed as xlo xhi on the 1st line, ylo yhi on the next line, zlo zhi on the last. There are N lines following "ITEM: ATOMS" where N is the number of atoms. As the dump tool commands indicate, atoms do not have to be listed in any particular order. There can be a different number of atoms in each snapshot. The values on each atom line are "ID type x y z" by default, but other quantities can be listed in any desired order. The map() command can be used to specify the ordering if it is not the default.

Usage:

```
d = dump("dump.one")              read in one or more dump files
d = dump("dump.1 dump.2.gz")      can be gzipped
d = dump("dump.*")                wildcard expands to multiple files
d = dump("dump.*",0)              two args = store filenames, but don't read

  incomplete and duplicate snapshots are deleted
  atoms will be unscaled if stored in files as scaled
  self-describing column names assigned

time = d.next()                   read next snapshot from dump files

  used with 2-argument constructor to allow reading snapshots one-at-a-time
  snapshot will be skipped only if another snapshot has same time stamp
  return time stamp of snapshot read
  return -1 if no snapshots left or last snapshot is incomplete
  no column name assignment or unscaling is performed

d.map(1,"id",3,"x")               assign names to columns (1-N)

  not needed if dump file is self-describing

d.tselect.all()                   select all timesteps
d.tselect.one(N)                  select only timestep N
d.tselect.none()                  deselect all timesteps
d.tselect.skip(M)                 select every Mth step
d.tselect.test("$t >= 100 and $t <10000")      select matching timesteps
d.delete()                        delete non-selected timesteps

  selecting a timestep also selects all atoms in the timestep
  skip() and test() only select from currently selected timesteps
  test() uses a Python Boolean expression with $t for timestep value
    Python comparison syntax: == != <>  <= >= and or

d.aselect.all()                             select all atoms in all steps
d.aselect.all(N)                            select all atoms in one step
d.aselect.test("$id > 100 and $type == 2")  select match atoms in all steps
d.aselect.test("$id > 100 and $type == 2",N)  select matching atoms in one step

  all() with no args selects atoms from currently selected timesteps
  test() with one arg selects atoms from currently selected timesteps
  test() sub-selects from currently selected atoms
  test() uses a Python Boolean expression with $ for atom attributes
    Python comparison syntax: == != <>  <= >= and or
    $name must end with a space

d.write("file")                    write selected steps/atoms to dump file
d.write("file",head,app)           write selected steps/atoms to dump file
d.scatter("tmp")                   write selected steps/atoms to multiple files
```

```
  write() can be specified with 2 additional flags
    head = 0/1 for no/yes snapshot header, app = 0/1 for write vs append
  scatter() files are given timestep suffix: e.g. tmp.0, tmp.100, etc

d.scale()                          scale x,y,z to 0-1 for all timesteps
d.scale(100)                       scale atom coords for timestep N
d.unscale()                        unscale x,y,z to box size to all timesteps
d.unscale(1000)                    unscale atom coords for timestep N
d.wrap()                           wrap x,y,z into periodic box via ix,iy,iz
d.unwrap()                         unwrap x,y,z out of box via ix,iy,iz
d.owrap("other")                   wrap x,y,z to same image as another atom
d.sort()                           sort atoms by atom ID in all selected steps
d.sort("x")                        sort atoms by column value in all steps
d.sort(1000)                       sort atoms in timestep N

  scale(), unscale(), wrap(), unwrap(), owrap() operate on all steps and atoms
  wrap(), unwrap(), owrap() require ix,iy,iz be defined
  owrap() requires a column be defined which contains an atom ID
    name of that column is the argument to owrap()
    x,y,z for each atom is wrapped to same image as the associated atom ID
    useful for wrapping all molecule's atoms the same so it is contiguous

m1,m2 = d.minmax("type")               find min/max values for a column
d.set("$ke = $vx * $vx + $vy * $vy")   set a column to a computed value
d.setv("type",vector)                  set a column to a vector of values
d.spread("ke",N,"color")               2nd col = N ints spread over 1st col
d.clone(1000,"color")                  clone timestep N values to other steps

  minmax() operates on selected timesteps and atoms
  set() operates on selected timesteps and atoms
    left hand side column is created if necessary
    left-hand side column is unset or unchanged for non-selected atoms
    equation is in Python syntax
    use $ for column names, $name must end with a space
  setv() operates on selected timesteps and atoms
    if column label does not exist, column is created
    values in vector are assigned sequentially to atoms, so may want to sort()
    length of vector must match # of selected atoms
  spread() operates on selected timesteps and atoms
    min and max are found for 1st specified column across all selected atoms
    atom's value is linear mapping (1-N) between min and max
    that is stored in 2nd column (created if needed)
    useful for creating a color map
  clone() operates on selected timesteps and atoms
    values at every timestep are set to value at timestep N for that atom ID
    useful for propagating a color map

t = d.time()                       return vector of selected timestep values
fx,fy,... = d.atom(100,"fx","fy",...)   return vector(s) for atom ID N
fx,fy,... = d.vecs(1000,"fx","fy",...)  return vector(s) for timestep N

  atom() returns vectors with one value for each selected timestep
  vecs() returns vectors with one value for each selected atom in the timestep

index,time,flag = d.iterator(0/1)          loop over dump snapshots
time,box,atoms,bonds,tris,lines = d.viz(index)   return list of viz objects
d.atype = "color"                          set column returned as "type" by viz
d.extra(obj)                               extract bond/tri/line info from obj

  iterator() loops over selected timesteps
  iterator() called with arg = 0 first time, with arg = 1 on subsequent calls
    index = index within dump object (0 to # of snapshots)
    time = timestep value
```

```
  flag = -1 when iteration is done, 1 otherwise
viz() returns info for selected atoms for specified timestep index
  can also call as viz(time,1) and will find index of preceding snapshot
  time = timestep value
  box = \[xlo,ylo,zlo,xhi,yhi,zhi\]
  atoms = id,type,x,y,z for each atom as 2d array
  bonds = id,type,x1,y1,z1,x2,y2,z2,t1,t2 for each bond as 2d array
    if extra() used to define bonds, else NULL
  tris = id,type,x1,y1,z1,x2,y2,z2,x3,y3,z3,nx,ny,nz for each tri as 2d array
    if extra() used to define tris, else NULL
  lines = id,type,x1,y1,z1,x2,y2,z2 for each line as 2d array
    if extra() used to define lines, else NULL
atype is column name viz() will return as atom type (def = "type")
extra() extracts bonds/tris/lines from obj each time viz() is called
  obj can be data object for bonds, cdata object for tris and lines,
    bdump object for bonds, tdump object for tris, ldump object for lines.
    mdump object for tris
```

Related tools:

data, gl, mdump, tdump, raster, svg

Prerequisites:

Numeric or NumPy Python packages. Gunzip command (if you want to read gzipped files).

ensight tool

Purpose:

Convert LAMMPS snapshots or meshes to Ensight format.

Description:

The ensight tool converts atom snapshots in a LAMMPS dump or data file or mesh data from a mesh file to the format used by the Ensight visualization package. See the dump or mdump tools for info about the format of these files and what data they contain.

The ensight constructor takes an object that stores atom or mesh snapshots (dump, data) as its first argument. The atom snapshots must have "id", "type", "x", "y", and "z" defined; see the map() methods of those tools.

The one(), many(), and single() methods convert specific snapshots to Ensight format and write them out. These methods take a file prefix as an optional first argument; the prefix "tmp" will be used if not specified. These methods all create a prefix.case file which is used by Ensight to define the format and list all associated files. One or more prefix*.xyz files are also produced which contain atom coordinates sorted by atom type. If additional pairs of arguments are specified, attributes from the shapshot data (atoms or elements) can be written into Ensight variable files, and used by Ensight as display attributes for the atoms or mesh elements.

Usage:

```
e = ensight(d)        d = object with atoms or elements (dump,data,mdump)
e.change = 1          set to 1 if element nodal xyz change with time (def = 0)
e.maxtype = 10        max particle type, set if query to data will be bad

e.one()
e.one("new")
e.one("cns","Centro","eng","Energy")
e.one("new","cns","Centro","eng","Energy")
                      write all snapshots as an Ensight data set
                      Ensight header file = tmp.case (no 1st arg) or new.case
                      Ensight coord file = tmp.xyz or new.xyz
                      additional pairs of args create auxiliary files:
                        tmp.cns, tmp.eng or new.cns, new.eng
                      cns,eng = column name in dump file and file name suffix
                      Centro,Energy = Ensight name for the variable

e.increment()         same args as one(), but process dump out-of-core

e.many()              same args as one(), but create multiple Ensight files
                      tmp0000.xyz, tmp0001.xyz, etc
                      new0000.cns, new0001.cns, etc
                      new0000.eng, new0001.eng, etc

e.single(N)           same args as one() prepended by N, but write a single snap
```

Related tools:

cfg, data, dump, mdump, vtk, xyz

Prerequisites: none

gl tool

Purpose:

3d interactive visualization via OpenGL.

Description:

The gl tool does visualizes LAMMPS or ChemCell snapshots or data files and mesh files via OpenGL graphics calls. The interface to OpenGL is provided to Python by the open-source PyOpenGL toolkit. To do interactive visualization, consider using this tool in conjunction with the vcr tool.

The gl constructor takes a data object containing atom or mesh snapshots as an argument (dump, data, cdata, mdump).

The show() method displays a single image of the selected atoms or triangles of snapshot N and saves it as image.png. While an image is displayed, the view can be changed directly by using the mouse buttons in the OpenGl window to translate (left), rotate (middle), or zoom (right) the image. However, these mouse changes are not permanent, unless the corresponding trans(), rotate(), and zoom() methods are invoked. The all() method loops thru all selected snapshots, displaying each one in turn. The resulting image files are saved to image0000.png, image0001.png, etc. The prefix "image" can be changed via the file setting.

The bg(), size(), rotate(), trans(), zoom(), box(), label(), and nolabel() methods control various aspects of the images produced. Without the trans() and zoom() settings, the GL image should roughly fill the window and be centered.

Additional movie effects can be produced using the pan() and select() methods. The pan() method specifies an initial and final viewpoint that is applied to the images produced by the all() method. For intermediate images, the view parameters will be interpolated between their initial and final values. The pan() method can thus be used to rotate a single image or fly-by the simulation as it animates. The select() method performs additional atom selection for each image of the all() sequence. An image-dependent %g variable can be used in the select() string. The select() method can thus be used to slice thru the data set.

The acol(), arad(), bcol(), brad(), tcol(), and tfill() methods change attributes of the atoms, bonds, and triangles displayed. Each atom or bond returned from the data object has an integer "type" from 1 to N. The type is used to index into a list of RGB colors or radii for atoms and bond thickness. The adef(), bdef(), and tdef() methods setup default mappings of types to colors/radii. The other methods enable specific assignments to be made. The interpolation option (e.g. many types assigned to a few colors), enables a smooth rainbow of colors to be assigned to a range of types. Note that the dump tool allows any vector to be returned as an atom "type" via its atype setting. Thus displayed colors can be keyed to atom velocity or other properties.

Colors are specified with a string, e.g. "red". There are 140 pre-defined colors from this WWW page which can be examined by importing the "colors" variable from "vizinfo". New colors can be defined by assigning a nickname of your choice to an RGB triplet, as described below.

The visual quality of atom spheres can be set via the q() method. The axis() method toggles whether xyz axes are displayed in the OpenGL window. Red is the +x axis, green is +y, and blue is +z. The ortho() method toggles between an orthographic and perspective display of the snapshot. The clip() method can be used to narrow the amount of data that is visualized. The reload() method is needed if you change the selection attributes of the underlying data assigned to the gl tool, such as via the dump tool's methods. These changes will not be visible in

the OpenGL window until the data is reloaded.

Usage:

```
g = gl(d)                    create OpenGL display for data in d
g = gl(d,N)                  window of size NxN (def = 512x512)
g = gl(d,N,M)                window of size NxM

  d = atom snapshot object (dump, data)

g.bg("black")                set background color (def = "black")
g.rotate(60,135)             view from z theta and azimuthal phi (def = 60,30)
g.shift(x,y)                 translate by x,y pixels in view window (def = 0,0)
g.zoom(0.5)                  scale image by factor (def = 1)
g.box(0/1/2)                 0/1/2 = none/variable/fixed box
g.box(0/1/2,"green")         set box color
g.box(0/1/2,"red",4)         set box edge thickness
g.file = "image"             file prefix for created images (def = "image")

g.show(N)                    show image of snapshot at timestep N

g.all()                      make images of all selected snapshots
g.all(P)                     images of all, start file label at P
g.all(N,M,P)                 make M images of snapshot N, start label at P

g.pan(60,135,1.0,40,135,1.5)    pan during all() operation
g.pan()                         no pan during all() (default)

  args = z theta, azimuthal phi, zoom factor at beginning and end
  values at each step are interpolated between beginning and end values

g.select = "$x > %g*3.0"     string to pass to d.aselect.test() during all()
g.select = ""                no extra aselect (default)

  %g varies from 0.0 to 1.0 from beginning to end of all()

g.acol(2,"green")                  set atom colors by atom type (1-N)
g.acol([2,4],["red","blue"])       1st arg = one type or list of types
g.acol(0,"blue")                   2nd arg = one color or list of colors
g.acol(range(20),["red","blue"])   if list lengths unequal, interpolate
g.acol(range(10),"loop")           assign colors in loop, randomly ordered

  if 1st arg is 0, set all types to 2nd arg
  if list of types has a 0 (e.g. range(10)), +1 is added to each value
  interpolate means colors blend smoothly from one value to the next

g.arad([1,2],[0.5,0.3])            set atom radii, same rules as acol()

g.bcol()                           set bond color, same args as acol()
g.brad()                           set bond thickness, same args as arad()

g.tcol()                           set triangle color, same args as acol()
g.tfill()                          set triangle fill, 0 fill, 1 line, 2 both

g.lcol()                           set line color, same args as acol()
g.lrad()                           set line thickness, same args as arad()

g.adef()                           set atom/bond/tri/line properties to default
g.bdef()                           default = "loop" for colors, 0.45 for radii
g.tdef()                           default = 0.25 for bond/line thickness
```

```
g.ldef()                          default = 0 fill

  by default 100 types are assigned
  if atom/bond/tri/line has type > # defined properties, is an error

from vizinfo import colors        access color list
print colors                      list defined color names and RGB values
colors["nickname"] = [R,G,B]      set new RGB values from 0 to 255

  140 pre-defined colors: red, green, blue, purple, yellow, black, white, etc

Settings specific to gl tool:

g.q(10)                     set quality of image (def = 5)
g.axis(0/1)                 turn xyz axes off/on
g.ortho(0/1)                perspective (0) vs orthographic (1) view
g.clip('xlo',0.25)          clip in xyz from lo/hi at box fraction (0-1)
g.reload()                  force all data to be reloaded
g.cache = 0/1               turn off/on GL cache lists (def = on)
theta,phi,x,y,scale,up = g.gview()   grab all current view parameters
g.sview(theta,phi,x,y,scale,up)      set all view parameters

  data reload is necessary if dump selection is used to change the data
  cache lists usually improve graphics performance
  gview returns values to use in other commands:
    theta,phi are args to rotate()
    x,y are args to shift()
    scale is arg to zoom()
    up is a 3-vector arg to sview()
```

Related tools:

dump, rasmol, raster, svg, vcr, vmd

Prerequisites:

Python PyOpenGL and PIL packages.

gnu tool

Purpose:

Create plots via GnuPlot plotting program.

Description:

The gnu tool is a wrapper on the GnuPlot plotting package. GnuPlot is open source software and runs on any platform.

The gnu constructor launches GnuPlot as a process which the gnu tool sends commands to. The GnuPlot process can be killed via the stop() method, though this is typically unnecessary.

The plot() method plots a single vector against a linear index or pairs of vectors against each other. The pairs of vectors are written to files and read-in by GnuPlot's plot command.

The mplot() method creates a series of plots and saves them each to a numbered file. Each file is a plot of an increasing portion of the vector(s). This can be used to make an animation of a plot.

The enter() method can be used to interact with GnuPlot directly. Each subsequent line you type is a GnuPlot command, until you type "quit" or "exit" to return to Pizza.py. Single GnuPlot commands can be issued as string arguments to the gnu tool.

The export() method writes numeric data as columns to text files, so that GnuPlot can read them via its "plot" command.

Mutliple windows can be displayed, plotted to, and manipulated using the select() and hide() methods. The save() method writes the currently selected plot to a PostScript file.

The remaining methods (aspect, title, xrange, etc) set attributes of the currently selected plot. The erase() method resets all attributes to their default values.

Usage:

```
g = gnu()                       start up GnuPlot
g.stop()                        shut down GnuPlot process

g.plot(a)                       plot vector A against linear index
g.plot(a,b)                     plot B against A
g.plot(a,b,c,d,...)             plot B against A, D against C, etc
g.mplot(M,N,S,"file",a,b,...)   multiple plots saved to file0000.eps, etc

  each plot argument can be a tuple, list, or Numeric/NumPy vector
  mplot loops over range(M,N,S) and create one plot per iteration
    last args are same as list of vectors for plot(), e.g. 1, 2, 4 vectors
    each plot is made from a portion of the vectors, depending on loop index i
      Ith plot is of b[0:i] vs a[0:i], etc
    series of plots saved as file0000.eps, file0001.eps, etc
    if use xrange(),yrange() then plot axes will be same for all plots

g("plot 'file.dat' using 2:3 with lines")      execute string in GnuPlot
```

```
g.enter()                               enter GnuPlot shell
gnuplot> plot sin(x) with lines         type commands directly to GnuPlot
gnuplot> exit, quit                     exit GnuPlot shell

g.export("data",range(100),a,...)       create file with columns of numbers

  all vectors must be of equal length
  could plot from file with GnuPlot command: plot 'data' using 1:2 with lines

g.select(N)                     figure N becomes the current plot

  subsequent commands apply to this plot

g.hide(N)                       delete window for figure N
g.save("file")                  save current plot as file.eps

Set attributes for current plot:

g.erase()                       reset all attributes to default values
g.aspect(1.3)                   aspect ratio
g.xtitle("Time")                x axis text
g.ytitle("Energy")              y axis text
g.title("My Plot")              title text
g.title("title","x","y")        title, x axis, y axis text
g.xrange(xmin,xmax)             x axis range
g.xrange()                      default x axis range
g.yrange(ymin,ymax)             y axis range
g.yrange()                      default y axis range
g.xlog()                        toggle x axis between linear and log
g.ylog()                        toggle y axis between linear and log
g.label(x,y,"text")             place label at x,y coords
g.curve(N,'r')                  set color of curve N

  colors: 'k' = black, 'r' = red, 'g' = green, 'b' = blue
          'm' = magenta, 'c' = cyan, 'y' = yellow
```

Related tools:

matlab, plotview

Prerequisites:

GnuPlot plotting package.

histo tool

Purpose:

Particle density histogram from a dump.

Description:

The histo tool creates spatial histograms of particle snapshots in a dump file.

The histo constructor takes an object that stores atom snapshots (dump, data) as its argument.

The compute() method creates a histogram in a specific dimension at a desired resolution, averaged across all selected snapshots and atoms in the dump. The returned vectors can be plotted; x is the distance along the chosen dimension, y is the histogram counts.

Usage:

```
h = histo(d)                          d = dump/cdump object

x,y = h.compute('x',N,lo,hi)          compute histogram in dim with N bins

  lo/hi are optional, if not used histo will be over entire box
```

Related tools:

dump

Prerequisites: none

image tool

Purpose:

View and manipulate images.

Description:

The image tool can be used to display image files or convert them to other formats via the ImageMagick tools (or alternate tools if specified in the DEFAULTS.py file).

The image constructor creates a GUI to view a set of image files as a palette of thumbnail-size images. Each thumbnail can be clicked on to view the full-size image. Clicking on the full-size image removes it. The view() method does the same operation for a new set of files.

The convert() method invokes the ImageMagick "convert" command to convert an image file to a different format. If both arguments have a wildcard character, one conversion is done for each file in the 1st argument to create a file in the 2nd argument, e.g. "convert image0012.svg new0012.png". If either argument has no wildcard, one "convert" command is issued using both arguments. This form can be used to create a movie file, e.g. "convert *.png ligand.mpg".

The montage() method invokes the ImageMagick "montage" command to combine 2 image files to create a 3rd. If all 3 arguments have a wildcard character, one montage is created for each file in the 1st argument, paired with one file in the 2nd argument, e.g. "montage image0012.svg plot0012.eps combine0012.png". For this to work, the 1st arguments must each expand to the same number of files. If any of the 3 arguments does not have a wildcard, one "montage" command is issued using all 3 arguements.

Image files can be in any format (PNG, GIF, JPG, etc) recognized by the Python Image Library (PIL) installed in your Python or by ImageMagick. Various Pizza.py visualization tools (raster, deja, rasmol, etc) create such image files. If a particular image format fails to load, your PIL installation was linked without support for that format. Rebuild PIL, and follow the install instructions included in its top directory.

Usage:

```
i = image("my1.gif my2.gif")    display thumbnails of matching images
i = image("*.png *.gif")        wildcards allowed
i = image("*.png *.gif",0)      2nd arg = sort filenames, 0 = no sort, def = 1
i = image("")                   blank string matches all image suffixes
i = image()                     no display window opened if no arg

  image suffixes for blank string = *.png, *.bmp, *.gif, *.tiff, *.tif
  click on a thumbnail to view it full-size
  click on thumbnail again to remove full-sized version

i.view("*.png *.gif")           display thumbnails of matching images

  view arg is same as constructor arg

i.convert("image*.svg","new*.png")                      each SVG file to PNG
i.convert("image*.svg","new*.jpg","-quality 50")        3rd arg is switch
i.convert("image*.png","movie.mpg")                     all PNGs to MPG movie
i.convert("image*.png","movie.mpg","-resize 128x128")   3rd arg is switch
i.montage("","image*.png","plot*.png","two*.png")       image + plot = two
```

```
i.montage("-geometry 512x512","i*.png","new.png")       1st arg is switch

  convert with 2 wildcard args loops over 1st set of files to make 2nd set
  convert with not all wildcard args will issue single convert command
  montage with all wildcard args loops over 1st set of files,
    combines with one file from other sets, to make last set of files
  montage with not all wildcard args will issue single montage command
```

Related tools:

raster, rasmol, animate

Prerequisites:

Python Tkinter, Pmw, and PIL packages. ImageMagick convert and montage commands or equivalent.

ldump tool

Purpose:

Read dump files with line segment info.

Description:

The ldump tool reads one or more LAMMPS dump files, and stores their contents as a series of snapshots with 2d arrays of atom attributes. It is assumed that each atom contains line segment info from a 2d LAMMPS simulation using atom_style line. Other tools use ldump objects to extract line segment info for visualization, like the dump tool via its extra() method.

The constructor method is passed a string containing one or more dump filenames. They can be listed in any order since snapshots are sorted by timestep after they are read and duplicate snapshots (with the same time stamp) are deleted. If a 2nd argument is specified, the files are not immediately read, but snapshots can be read one-at-a-time by the next() method.

The map() method assigns names to columns of atom attributes. The id,type,end1x,end1y,end2x,end2y names must be assigned in order for line segment info to be extracted.

The viz() method is called by Pizza.py tools that visualize snapshots of atoms (e.g. gl, raster, svg tools).

Normally, LAMMPS creates the dump files read in by this tool. If you want to create them yourself, the format of LAMMPS dump files is simple. Each snapshot is formatted as follows:

```
ITEM: TIMESTEP
100
ITEM: NUMBER OF ATOMS
32
ITEM: BOX BOUNDS
0 3.35919
0 3.35919
0 7.50
ITEM: ATOMS
1 1 0 0 0
2 1 0.25 0.25 0
3 1 0.25 0 0.25
...
N 3 0.7 0.5 0.6
```

The box bounds are listed as xlo xhi on the 1st line, ylo yhi on the next line, zlo zhi on the last. There are N lines following "ITEM: ATOMS" where N is the number of atoms. Atoms do not have to be listed in any particular order. There can be a different number of atoms in each snapshot. Each line must contain the atom ID, type, and the end points of the associated line segment, as specified by the map() command.

Usage:

```
l = ldump("dump.one")              read in one or more dump files
l = ldump("dump.1 dump.2.gz")      can be gzipped
l = ldump("dump.*")                wildcard expands to multiple files
l = ldump("dump.*",0)              two args = store filenames, but don't read
```

```
  incomplete and duplicate snapshots are deleted
  no column name assignment is performed

time = l.next()                   read next snapshot from dump files

  used with 2-argument constructor to allow reading snapshots one-at-a-time
  snapshot will be skipped only if another snapshot has same time stamp
  return time stamp of snapshot read
  return -1 if no snapshots left or last snapshot is incomplete
  no column name assignment is performed

l.map(1,"id",3,"x")               assign names to atom columns (1-N)

  must assign id,type,end1x,end1y,end2x,end2y

time,box,atoms,bonds,tris,lines = l.viz(index)   return list of viz objects

  viz() returns line info for specified timestep index
    can also call as viz(time,1) and will find index of preceding snapshot
    time = timestep value
    box = \[xlo,ylo,zlo,xhi,yhi,zhi\]
    atoms = NULL
    bonds = NULL
    tris = NULL
    lines = id,type,x1,y1,z1,x2,y2,z2 for each line as 2d array
      id,type are from associated atom

l.owrap(...)                      wrap lines to same image as their atoms

  owrap() is called by dump tool's owrap()
  useful for wrapping all molecule's atoms/lines the same so it is contiguous
```

Related tools:

dump, gl, raster, svg

Prerequisites:

Numeric or NumPy Python packages. Gunzip command (if you want to read gzipped files).

log tool

Purpose:

Read LAMMPS log files and extract thermodynamic data.

Description:

Read one or more LAMMPS log files and combine their thermodynamic data into long, named vectors (versus time). The vectors can be used in Python for further processing and plotting, or they can be output to a file.

The log constructor reads one or more log files. If 2 arguments are specified, a single file is specified, and it can be read incrementally (e.g. as it is created) by the next() method.

The nvec, nlen, and names values give the # of vectors, their length, and names. The get() method returns one of more vectors as a Python list. The write() method outputs the numeric vectors to a file.

Usage:

```
l = log("file1")                    read in one or more log files
l = log("log1 log2.gz")             can be gzipped
l = log("file*")                    wildcard expands to multiple files
l = log("log.lammps",0)             two args = store filename, but don't read

  incomplete and duplicate thermo entries are deleted

time = l.next()                     read new thermo info from file

  used with 2-argument constructor to allow reading thermo incrementally
  return time stamp of last thermo read
  return -1 if no new thermo since last read

nvec = l.nvec                       # of vectors of thermo info
nlen = l.nlen                       length of each vectors
names = l.names                     list of vector names
t,pe,... = l.get("Time","KE",...)   return one or more vectors of values
l.write("file.txt")                 write all vectors to a file
l.write("file.txt","Time","PE",...) write listed vectors to a file

  get and write allow abbreviated (uniquely) vector names
```

Related tools:

olog, plotview, gnu, matlab

Prerequisites: none

matlab tool

Purpose:

Create plots via MatLab numerical analysis program.

Description:

The matlab tool is a wrapper on the MatLab numerical analysis package, primarily designed to use its plotting capabilities. MatLab is commercial software available on a variety of platforms.

The matlab constructor launches MatLab as a process which the matlab tool sends commands to. The MatLab process can be killed via the stop() method, though this is typically unnecessary.

The plot() method plots a single vector against a linear index or pairs of vectors against each other. The pairs of vectors are written to files and read-in by MatLab. The Nth curve in a plot is stored in the MatLab variable pizzaN, so the 2 vectors can be accessed within MatLab as pizzaN(:,1) and pizzaN(:,2).

The mplot() method creates a series of plots and saves them each to a numbered file. Each file is a plot of an increasing portion of the vector(s). This can be used to make an animation of a plot.

The enter() method can be used to interact with MatLab directly. Each subsequent line you type is a MatLab command, until you type "quit" or "exit" to return to Pizza.py. Single MatLab commands can be issued as string arguments to the gnu tool.

The export() method writes numeric data as columns to text files, so that MatLab can read them via its "importdata" command.

Mutliple windows can be displayed, plotted to, and manipulated using the select() and hide() methods. The save() method writes the currently selected plot to a PostScript file.

The remaining methods (aspect, title, xrange, etc) set attributes of the currently selected plot. The erase() method resets all attributes to their default values.

Usage:

```
m = matlab()                    start up MatLab
m.stop()                        shut down MatLab process

m.plot(a)                       plot vector A against linear index
m.plot(a,b)                     plot B against A
m.plot(a,b,c,d,...)             plot B against A, D against C, etc
m.mplot(M,N,S,"file",a,b,...)   multiple plots saved to file0000.eps, etc

  each plot argument can be a tuple, list, or Numeric/NumPy vector
  mplot loops over range(M,N,S) and create one plot per iteration
    last args are same as list of vectors for plot(), e.g. 1, 2, 4 vectors
    each plot is made from a portion of the vectors, depending on loop index i
      Ith plot is of b[0:i] vs a[0:i], etc
    series of plots saved as file0000.eps, file0001.eps, etc
    if use xrange(),yrange() then plot axes will be same for all plots

m("c = a + b")                  execute string in MatLab
```

```
m.enter()                       enter MatLab shell
matlab> c = a + b               type commands directly to MatLab
matlab> exit, quit              exit MatLab shell

m.export("data",range(100),a,...)       create file with columns of numbers

  all vectors must be of equal length
  could plot from file with MatLab commands:
    cols = importdata('data')
    plot(cols(:,1),cols(:,2))

m.select(N)                     figure N becomes the current plot

  subsequent commands apply to this plot

m.hide(N)                       delete window for figure N
m.save("file")                  save current plot as file.eps

Set attributes for current plot:

m.erase()                       reset all attributes to default values
m.aspect(1.3)                   aspect ratio
m.xtitle("Time")                x axis text
m.ytitle("Energy")              y axis text
m.title("My Plot")              title text
m.title("title","x","y")        title, x axis, y axis text
m.xrange(xmin,xmax)             x axis range
m.xrange()                      default x axis range
m.yrange(ymin,ymax)             y axis range
m.yrange()                      default y axis range
m.xlog()                        toggle x axis between linear and log
m.ylog()                        toggle y axis between linear and log
m.label(x,y,"text")             place label at x,y coords
m.curve(N,'r')                  set color of curve N
m.curve(N,'g','--')             set color and line style of curve N
m.curve(N,'b','-','v')          set color, line style, symbol of curve N

  colors:  'k' = black, 'r' = red, 'g' = green, 'b' = blue
           'm' = magenta, 'c' = cyan, 'y' = yellow
  styles:  '-' = solid, '--' = dashed, ':' = dotted, '-.' = dash-dot
  symbols: '+' = plus, 'o' = circle, '*' = asterik, 'x' = X,
           's' = square, 'd' = diamond, '^' = up triangle,
           'v' = down triangle, '>' = right triangle,
           '
```

Related tools:

gnu, plotview

Prerequisites:

The MatLab numerical analysis package.

mdump tool

Purpose:

Read, write, manipulate mesh dump files.

Description:

The mdump tool reads one or more mesh dump files, stores their contents as a series of snapshots with node and element values, and allows the values to be accessed and manipulated. Other tools use mdump objects to convert mesh files to other formats or visualize the mesh.

The constructor method is passed a string containing one or more mesh dump filenames. They can be listed in any order since snapshots are sorted by timestep after they are read and duplicate snapshots (with the same time stamp) are deleted. If a 2nd argument is specified, the files are not immediately read, but snapshots can be read one-at-a-time by the next() method.

The map() method assigns names to columns of element values. The tselect() methods select one or more snapshots by their time stamp. The delete() method deletes unselected timesteps so their memory is freed up. This can be useful to do when reading snapshots one-at-a-time for huge data sets. The eselect() methods selects elements within selected snapshots.

The time() and vecs() methods return time or element values as vectors of values.

The iterator() and viz() methods are called by Pizza.py tools that visualize triangles (e.g. gl, raster, svg tools). You can also use the iterator() method in your scripts to loop over selected snapshots. The etype setting determines what element value is returned as the element "type" by the viz() method. The mviz() method is called by the ensight tool to extract per-node and per-element data to include in Ensight files it outputs, so that it can visualized in the Ensight package.

The format of mesh dump files read in by this tool is simple. There are 4 kinds of entries that can appear within each timestep, in any order. Note that each entry is formatted with its own timestep stamp, but entries with duplicate stamps are combined.

The 4 kinds of entries are a list of node coordinates, a list of element topologies (trianglular, tetrahedral, square, and cubic elements are currently supported), a list of nodal values, and a list of element values.

Note that a mesh file does not need to include all 4 entries for every timestep. For example, if the file only defines a mesh, but no nodal or element values, then the nodal and element value entries need never appear. If the mesh is static, then the nodal coordinates and element topologies only need to be defined once (e.g. for timestep 0). If the mesh moves, but its topology is static, then the element topology can be defined once (timestep 0), but new nodal coordinates could be defined every timestamp.

A single mesh file can contain multiple snapshots, listed one after the other.

The format of a nodal coordinates entry is as follows:

```
ITEM: TIMESTEP
0
ITEM: NUMBER OF NODES
121
```

```
ITEM: BOX BOUNDS
0 10
0 10
0 0
ITEM: NODES
1 1 0 0 0
2 1 0 1 0
3 1 0 2 0
...
N 3 7.5 3 2.5
```

The box bounds are listed as xlo xhi on the 1st line, ylo yhi on the next line, zlo zhi on the last. There are N lines following "ITEM: NODES" where N is the number of nodes. The nodes do not need to be listed in any particular order. There can be a different number of nodes in each snapshot. The values on each node line are "ID type x y z".

The format of an element topology entry is in one of the following styles (triangles, tetrahedra, squares, cubes):

```
ITEM: TIMESTEP
0
ITEM: NUMBER OF TRIANGLES
200
ITEM: TRIANGLES
1 1 1 13 12
2 1 1 2 13
3 1 2 14 13
...
N 3 259 340 432
```

```
ITEM: TIMESTEP
0
ITEM: NUMBER OF TETS
200
ITEM: TETS
1 1 1 13 12 27
2 1 1 2 13 35
3 1 2 14 13 103
...
N 3 259 340 432 1005
```

```
ITEM: TIMESTEP
0
ITEM: NUMBER OF SQUARES
200
ITEM: SQUARES
1 1 1 13 12 4
2 1 1 2 13 6
3 1 2 14 13 11
...
N 3 259 340 432 500
```

```
ITEM: TIMESTEP
0
ITEM: NUMBER OF CUBES
200
ITEM: CUBES
1 1 1 13 12 5 3 6 10 20
2 1 1 2 13 10 3 4 5 6
3 1 2 14 13 15 10 18 19
...
N 3 259 340 432 200 456 918 1002 988
```

There are N lines following "ITEM: TRIANGLES" or "ITEM: TETS" or "ITEM: SQUARES" or "ITEM: CUBES" where N is the number of elements. The elements do not need to be listed in any particular order. There can be a different number of elements in each snapshot. The values on each element line are "ID type node1 node2 ... nodeN", where N depends on the type of element. For triangles, N = 3, and the order should give an outward normal via the right-hand rule. For tets, N = 4, and the order should give a right-hand rule for the first 3 nodes that points towards the 4th node. For squares, N = 4, and the nodes should be listed in counter-clockwise order around the square. For cubes, N = 8, and the first 4 nodes are the lower face (ordered same as a square), and the last 4 nodes are directly above the lower 4 as the upper face.

The format of a nodal values entry is as follows:

```
ITEM: TIMESTEP
0
ITEM: NUMBER OF NODE VALUES
200
ITEM: NODE VALUES
1 1
2 1
3 9
...
N 32
```

There are N lines following "ITEM: NODE VALUES" where N is the number of nodes. The nodes do not need to be listed in any particular order. There can be a different number of nodes in each snapshot, but it should be consistent with the current nodal coordinates entry. The values on each node line are "ID value1 value2 ..." where there are as many columns as desired. Each line should have the same number of values.

The format of an element values entry is as follows:

```
ITEM: TIMESTEP
0
ITEM: NUMBER OF ELEMENT VALUES
200
ITEM: ELEMENT VALUES
1 1
2 1
3 9
...
N 32
```

There are N lines following "ITEM: ELEMENT VALUES" where N is the number of elements. The elements do not need to be listed in any particular order. There can be a different number of elements in each snapshot, but it should be consistent with the current element topology entry. The values on each element line are "ID value1 value2 ..." where there are as many columns as desired. Each line should have the same number of values. The map() command can be used to assign names to each of these columns of values.

Usage:

```
m = mdump("mesh.one")             read in one or more mesh dump files
m = mdump("mesh.1 mesh.2.gz")     can be gzipped
m = mdump("mesh.*")               wildcard expands to multiple files
m = mdump("mesh.*",0)             two args = store filenames, but don't read

  incomplete and duplicate snapshots are deleted

time = m.next()                   read next snapshot from dump files
```

```
  used with 2-argument constructor to allow reading snapshots one-at-a-time
  snapshot will be skipped only if another snapshot has same time stamp
  return time stamp of snapshot read
  return -1 if no snapshots left or last snapshot is incomplete
  no column name assignment or unscaling is performed

m.map(2,"temperature")            assign names to element value columns (1-N)

m.tselect.all()                   select all timesteps
m.tselect.one(N)                  select only timestep N
m.tselect.none()                  deselect all timesteps
m.tselect.skip(M)                 select every Mth step
m.tselect.test("$t >= 100 and $t <10000")      select matching timesteps
m.delete()                        delete non-selected timesteps

  selecting a timestep also selects all elements in the timestep
  skip() and test() only select from currently selected timesteps
  test() uses a Python Boolean expression with $t for timestep value
    Python comparison syntax: == != <>  <= >= and or

m.eselect.all()                               select all elems in all steps
m.eselect.all(N)                              select all elems in one step
m.eselect.test("$id > 100 and $type == 2")    select match elems in all steps
m.eselect.test("$id > 100 and $type == 2",N)  select matching elems in one step

  all() with no args selects atoms from currently selected timesteps
  test() with one arg selects atoms from currently selected timesteps
  test() sub-selects from currently selected elements
  test() uses a Python Boolean expression with $ for atom attributes
    Python comparison syntax: == != <>  <= >= and or
    $name must end with a space

t = m.time()                       return vector of selected timestep values
fx,fy,... = m.vecs(1000,"fx","fy",...)  return vector(s) for timestep N

  vecs() returns vectors with one value for each selected elem in the timestep

index,time,flag = m.iterator(0/1)          loop over mesh dump snapshots
time,box,atoms,bonds,tris,lines = m.viz(index)  return list of viz objects
nodes,elements,nvalues,evalues = m.mviz(index)  return list of mesh viz objects
m.etype = "color"                          set column returned as "type" by viz

  iterator() loops over selected timesteps
  iterator() called with arg = 0 first time, with arg = 1 on subsequent calls
    index = index within dump object (0 to # of snapshots)
    time = timestep value
    flag = -1 when iteration is done, 1 otherwise
  viz() returns info for selected elements for specified timestep index
    can also call as viz(time,1) and will find index of preceding snapshot
    time = timestep value
    box = \[xlo,ylo,zlo,xhi,yhi,zhi\]
    atoms = NULL
    bonds = NULL
    tris = id,type,x1,y1,z1,x2,y2,z2,x3,y3,z3,nx,ny,nz for each tri as 2d array
      each element is decomposed into tris
    lines = NULL
  mviz() returns info for all elements for specified timestep index
    can also call as mviz(time,1) and will find index of preceding snapshot
    time = timestep value
    box = \[xlo,ylo,zlo,xhi,yhi,zhi\]
    nodes = list of nodes = id,type,x,y,z
    elements = list of elements = id,type,node1,node2,...
```

```
    nvalues = list of node values = id,type,value1,value2,...
    evalues = list of element values = id,type,value1,value2,...
  etype is column name viz() will return as element type (def = "" = elem type)
```

Related tools:

dump, gl, raster, svg

Prerequisites:

Numeric or NumPy Python packages. Gunzip command (if you want to read gzipped files).

olog tool

Purpose:

Read other log files (ChemCell, SPPARKS, SPARTA) and extract time-series data.

Description:

Read one or more log files from other programs (besides LAMMPS) and combine their species statistical data into long, named vectors (versus time). The vectors can be used in Python for further processing and plotting, or they can be output to a file.

This tool will work from log files from the Sandia ChemCell, SPPARKS, and SPARTA packages. For LAMMPS log files, see the log tool.

The only difference between the log files of those programs, so far as this tool is concerned is the line that delimits the start of the sections that contain time-series data. This can be set via the second argument in the constructor, as documented below. For ChemCell and SPARTA, the default string "Step" can be used. For SPPARKS, the string "Time" will work. Note that SPARTA can change the text in the line that starts a section of time-series data. In this case a start string specific to the log file must be used.

The log constructor reads one or more log files. The names of these one or more files are listed in the 1st argument. The 2nd argument is a string with text that begins the line that delimits the start of sections that contain time-series data, as explained above. If a 3rd argument is specified, then a single file should be specified as the 1st argument, and it is assumed to contain data for multiple runs, which are averaged.

The nvec, nlen, and names values give the # of vectors, their length, and names. The get() method returns one of more vectors as a Python list. The write() method outputs the numeric vectors to a file.

Usage:

```
o = olog("file1")                    read in one or more log files
o = olog("log1 log2.gz")             can be gzipped
o = olog("file*")                    wildcard expands to multiple files
o = olog("log.spparks","Time")       2nd arg = start string for time section
o = olog("log.cell","",0)            3rd arg = average all runs

  incomplete and duplicate thermo entries are deleted
  if specify 2nd arg, it delimits a time section
  no 2nd arg or empty string, use default = "Step"
  if specify any 3rd arg, average all runs, assume all start at time 0

nvec = o.nvec                        # of vectors of thermo info
nlen = o.nlen                        length of each vectors
names = o.names                      list of vector names
a,b,... = o.get("A","B",...)         return one or more vectors of values
o.write("file.txt")                  write all vectors to a file
o.write("file.txt","A","B",...)      write listed vectors to a file

  get and write allow abbreviated (uniquely) vector names
```

Related tools:

plotview, gnu, log, matlab

Prerequisites: none

pair tool

Purpose:

Compute LAMMPS pairwise energies.

Description:

The pair tool computes a pairwise energy between 2 particles using a LAMMPS molecular dynamics force field. Thus it can be used in an analysis script to compute energies between groups of atoms from a LAMMPS snapshot file.

The pair constructor specifies the force field style. Only some of the LAMMPS pair styles are currently included in this tool, but new styles can easily be added. Code from the LAMMPS pair*.cpp file needs to be re-coded in Python to make this work.

The coeff() method reads the pairwise coefficients for the force field from a data file object (see the data tool). The init() method does pre-computations for the force field parameters needed by the single() method which does a pairwise computation between two atoms of type = itype,jtype separated by a squared distance rsq. The arguments for init() and single() can be different for a particular force field style. When you write the init() and single() methods for a new style, you can define what arguments are needed.

Usage:

```
p = pair("lj/charmm/coul/charmm")   create pair object for specific pair style

  available styles: lj/cut, lj/cut/coul/cut, lj/charmm/coul/charmm

p.coeff(d)                          extract pairwise coeffs from data object
p.init(cut1,cut2,...)               setup based on coeffs and cutoffs

  init args are specific to pair style:
    lj/cut = cutlj
    lj/cut/coul/cut = cutlj,cut_coul (cut_coul optional)
    lj/charmm/coul/charmm = cutlj_inner,cutlj,cutcoul_inner,cut_coul
      (last 2 optional)

e_vdwl,e_coul = p.single(rsq,itype,jtype,q1,q2,...)   compute LJ/Coul energy

  pairwise energy between 2 atoms at distance rsq with their attributes
  args are specific to pair style:
    lj/cut = rsq,itype,jtype
    lj/cut/coul/cut = rsq,itype,jtype,q1,q2
    lj/charmm/coul/charmm = rsq,itype,jtype,q1,q2
```

Related tools:

data

Prerequisites: none

patch tool

Purpose:

Create patchy or rigid particles for LAMMPS input.

Description:

The patch tool creates large multi-atom particles and writes them out as a LAMMPS data file. They need to be simulated with a soft potential in LAMMPS to un-overlap them before they form a proper ensemble.

The individual particles consist of a collection of Lennard-Jones atoms of various types. By defining force field coefficients appropriately, specific atoms can be made attractive or repulsive, so that "patches" of atoms on the particle surface are reactive. The Pizza.py WWW site has example images and movies of simulations using such particles. A paper by Sharon Glotzer's group at U Michigan describing a variety of patchy particle models was the motivation for this tool.

The patch constructor takes a volume fraction as an argument to determine how densely to fill the simulation box. Optionally, the box shape can also be specified.

The build() method creates N particles, each of speficied style and with specified atom types. Several styles are available and new ones can easily be added to patch.py. You will need to look in patch.py for the details of what each style represents. For example, "hex2" uses a C60 bucky ball as a template and creates hexagonal 6-atom patches (atoms of a different type) on either side of the ball.

The build() method can be invoked multiple times to create collections of particles. The position and orientation of each particle is chosen randomly. The seed value sets the random number generator used for coordinate generation.

The ensemble of chains is written to a LAMMPS data file via the write() method.

Usage:

```
p = patch(vfrac)            setup box with a specified volume fraction
p = patch(vfrac,1,1,2)      x,y,z = aspect ratio of box (def = 1,1,1)

p.seed = 48379              set random # seed (def = 12345)
p.randomized = 0            1 = choose next mol randomly (def), 0 = as generated
p.dim = 2                   set dimension of created box (def = 3)
p.blen = 0.97               set length of tether bonds (def = 0.97)
p.dmin = 1.02               set min r from i-1 to i+1 tether site (def = 1.02)
p.lattice = [Nx,Ny,Nz]      generate Nx by Ny by Nz lattice of particles
p.displace = [Dx,Dy,Dz]     displace particles randomly by +/- Dx,Dy,Dz
p.style = "sphere"          atom-style of data file, molecular or sphere
p.extra = "Molecules"       add extra Molecules section to data file
p.extratype = 1             add extra atom types when write data file

  randomized means choose molecules in random order when creating output
  if lattice is set, Nx*Ny*Nz must equal N for build (Nz = 1 for 2d)
  lattice = [0,0,0] = generate N particles randomly = default
  displace = [0,0,0] = default
  displacement applied when writing molecule to data file
  style = molecular by default
  style is auto-set to line,tri,box by corresponding keywords
```

```
  extratype = 0 by default

p.build(100,"hex2",1,2,3)  create 100 "hex2" particles with params 1,2,3

  can be invoked multiple times
  keywords:
    c60hex2: diam,1,2,3 = C-60 with 2 hex patches and ctr part, types 1,2,3
    hex2: diam,1,2 = one large particle with 2 7-mer hex patches, types 1,2
    hex4: diam,1,2 = one large particle with 4 7-mer hex patches, types 1,2
    ring: diam,N,1,2 = one large part with equatorial ring of N, types 1,2
    ball: diam,m1,m2,1,2,3 = large ball with m12-len tethers, types 1,2,3
    tri5: 1,2 = 3-layer 5-size hollow tri, types 1,2
    rod: N,m1,m2,1,2,3 = N-length rod with m12-len tethers, types 1,2,3
    tri: N,m1,m2,m3,1,2,3,4 = N-size tri with m123-len tethers, types 1-4
    trid2d: N,r,1 = 3d equilateral tri, N beads r apart, type 1, no bonds
    hex: m1,m2,m3,m4,m5,m6,1,2,3,4,5,6,7 = 7-atom hex with m-len tethers, t 1-7
    dimer: r,1 = two particles r apart, type 1, no bond
    star2d: N,r,1 = 2d star of length N (odd), beads r apart, type 1, no bonds
    box2d: N,M,r,1 = 2d NxM hollow box, beads r apart, type 1, no bonds
    pgon2d: Nlo,Nhi,m = 2d hollow polygons with random N beads from Nlo to Nhi
    sphere3d: Nlo,Nhi,m = 3d hollow spheres with random N beads/cube-edge
                          from Nlo to Nhi
    tritet: A,m = 4-tri tet with edge length A, tri type m
    tribox: Alo,Ahi,Blo,Bhi,Clo,Chi,m = 12-tri box with side lengths A,B,C & m
    linebox: Alo,Ahi,Blo,Bhi,m = 4-line 2d rectangle with random side lengths
                                 from Alo to Ahi and Blo to Bhi, line type m
                                 built of line particles
    linetri: Alo,Ahi,Blo,Bhi,m = 3-line 2d triangle with random base
                                 from Alo to Ahi and height Blo to Bhi, type m
                                 built of triangle particles
    bodypgon: Nlo,Nhi,m = 2d polygons with random N particles from Nlo to Nhi
                                 built of body particles

p.write("data.patch")      write out system to LAMMPS data file
```

Related tools:

chain, data

Prerequisites: none

(Glotzer) Zhang and Glotzer, NanoLetters, 4, 1407-1413 (2004).

pdbfile tool

Purpose:

Read, write PDB files in combo with LAMMPS snapshots.

Description:

The pdbfile tool reads in PDB (Protein Data Bank) files of protein coordinates and uses them in conjunction with LAMMPS snapshots in various ways. The PDB format is commonly used by various vizualization and analysis programs.

The pdbfile constructor takes a string argument listing one or more PDB filenames and a data argument with LAMMPS atom information (data or dump object). The atom snapshots must have "id", "type", "x", "y", and "z" defined; see the map() methods of those tools.

Both arguments to the constructor are optional, as described below. If a single PDB file is given as a constructor argument, and it is 4 letters long, and it does not exist on your system, then it is treated as a PDB identifier and the matching PDB file is downloaded from a PDB repository on the WWW to your machine.

The one(), many(), and single() methods write out PDB files in various manners, depending on the 1 or 2 arguments used in the constructor. If only a string was specified (no data object), the specified PDB files are written out as-is. Thus the one() method concatenates the files together. If only a data object was specified (no string of PDB files), then PDB files in a generic format (Lennard-Jones atoms) are created. If both arguments are specified, then only a single PDB file can be listed. It is treated as a template file, and the atom coordinates in the data object replace the atom coordinates in the template PDB file to create a series of new PDB files. This replacement is only done for selected atoms (in the dump object) and for atom IDs that appear in the PDB file.

The iterator() method is called by the rasmol tool to create a series of PDB files for visualization purposes.

Usage:

```
p = pdbfile("3CRO")          create pdb object from PDB file or WWW
p = pdbfile("pep1 pep2")     read in multiple PDB files
p = pdbfile("pep*")          can use wildcards
p = pdbfile(d)               read in snapshot data with no PDB file
p = pdbfile("3CRO",d)        read in single PDB file with snapshot data

  string arg contains one or more PDB files
    don't need .pdb suffix except wildcard must expand to file.pdb
    if only one 4-char file specified and it is not found,
      it will be downloaded from http://www.rcsb.org as 3CRO.pdb
  d arg is object with atom coordinates (dump, data)

p.one()                      write all output as one big PDB file to tmp.pdb
p.one("mine")                write to mine.pdb
p.many()                     write one PDB file per snapshot: tmp0000.pdb, ...
p.many("mine")               write as mine0000.pdb, mine0001.pdb, ...
p.single(N)                  write timestamp N as tmp.pdb
p.single(N,"new")            write as new.pdb

  how new PDB files are created depends on constructor inputs:
    if no d: one new PDB file for each file in string arg (just a copy)
    if only d specified: one new PDB file per snapshot in generic format
```

```
    if one file in str arg and d: one new PDB file per snapshot
      using input PDB file as template
    multiple input PDB files with a d is not allowed

index,time,flag = p.iterator(0)
index,time,flag = p.iterator(1)

  iterator = loop over number of PDB files
    call first time with arg = 0, thereafter with arg = 1
    N = length = # of snapshots or # of input PDB files
    index = index of snapshot or input PDB file (0 to N-1)
    time = timestep value (time stamp for snapshot, index for multiple PDB)
    flag = -1 when iteration is done, 1 otherwise
  typically call p.single(time) in iterated loop to write out one PDB file
```

Related tools:

data, dump, rasmol

Prerequisites: none

plotview tool

Purpose:

Plot multiple vectors from a data set.

Description:

The plotview tool displays a GUI for showing a series of plots of vectors stored in another Pizza.py tool. Individual plots can be displayed or hidden or saved to a file.

The plotview constructor creates the GUI and takes a data object (log, vec) and a plot object (gnu, matlab) as arguments.

The vectors in the data object are converted to individual plots. The 2nd thru Nth vectors are each plotted against the 1st vector to create N-1 plots. One or more plots can be displayed simultaneously (right buttons in GUI), but only one is selected at a time (left buttons in GUI). The yes(), no(), and select() methods perform the same function as the GUI buttons.

The currently selected plot can be modified (title, range, etc) by the methods of the plot object.

The file() and save() methods (or corresponding GUI widgets) save the currently selected plot to a PostScript file.

Usage:

```
p = plotview(d,pl)      create GUI for viewing plots

  d = Pizza.py object that contains vectors (log, vec)
  pl = Pizza.py plotting object (gnu, matlab)

p.select(2)             select one plot as current (1-N)
p.yes(3)                toggle one plot's visibility
p.no(3)

  only one plot is selected at a time
  multiple plots can be visible at same time
  select is same as clicking on left-side radio-button
  yes/no is same as clicking on right-side checkbox

p.x = "Time"            which vector is X vector (1st vec by default)
p.file("pressure")      filename prefix for saving a plot
p.save()                save currently selected plot to file.eps
```

Related tools:

log, gnu, matlab

Prerequisites:

Python Tkinter package.

rasmol tool

Purpose:

3d visualization via RasMol program.

Description:

The rasmol tool is a wrapper on the RasMol visualization program. RasMol is open source software and runs on many platforms. The link above is for the Open Rasmol WWW site, not the Protein Explorer WWW site. Protein Explorer is a derivative of RasMol and runs primarily on Windows machines within a browser. This Pizza.py tool wraps the original RasMol program, not Protein Explorer.

The rasmol constructor takes a pdbfile object as its argument which produces PDB files that RasMol reads in. The pdbfile object can produce PDB files from a LAMMPS dump or data file, as well as in other ways.

The show() method runs RasMol on the atoms of snapshot N (converted to a PDB file) and displays the resulting image stored as image.gif. Either a default RasMol script or one you specify is used to format the RasMol image. The all() method loops thru all selected snapshots and runs RasMol on each one. The resulting image files are saved to image0000.gif, image0001.gif, etc. The prefix "image" can be changed via the file setting.

A RasMol script can be created by running RasMol itself (outside of Pizza.py), typing commands or choosing menu options to format the display as desired, then typing "write script filename". Alternatively the run() method will do this for you. It runs RasMol on snapshot N and lets you interact with RasMol directly either via typing or mouse operations in the RasMol window. When you type "quit" or "exit" the script file will be saved (do not exit via the Rasmol menu).

Usage:

```
r = rasmol(p)           create RasMol wrapper for pdb object p

r.file = "image"        file prefix for created images (def = "image")

r.show(N)               show snapshot at timestep N with default script
r.show(N,"my.rasmol")   use file as RasMol script

r.all()                 make images of all selected snapshots with def script
r.all("my.rasmol")      use file as RasMol script

r.run(N)                run RasMol interactivly on snapshot N
r.run(N,"new.rasmol")                adjust via mouse or RasMol commands
r.run(N,"new.rasmol","old.rasmol")   type quit to save RasMol script file

  if 2 args, 2nd arg is new script file, else save to "tmp.rasmol"
  if 3 args, 3rd arg is initial script file, else use default script
```

Related tools:

dump, gl, pdbfile, raster, svg

Prerequisites:

The RasMol program.

raster tool

Purpose:

3d visualization via Raster3d program.

Description:

The raster tool is a wrapper on the Raster3d visualization program. Raster3d is open source software and runs on many platforms.

The raster constructor takes a data object containing atom or mesh snapshots as an argument (dump, data, cdata, mdump).

The show() method runs Raster3d on the selected atoms or triangles of snapshot N and displays the resulting image stored as image.png. The all() method loops thru all selected snapshots and runs Raster3d on each one. The resulting image files are saved to image0000.png, image0001.png, etc. The prefix "image" can be changed via the file setting.

The bg(), size(), rotate(), trans(), zoom(), box(), label(), and nolabel() methods control various aspects of the images produced. Without the trans() and zoom() settings, the Raster3d image should roughly fill the window and be centered.

Additional movie effects can be produced using the pan() and select() methods. The pan() method specifies an initial and final viewpoint that is applied to the images produced by the all() method. For intermediate images, the view parameters will be interpolated between their initial and final values. The pan() method can thus be used to rotate a single image or fly-by the simulation as it animates. The select() method performs additional atom selection for each image of the all() sequence. An image-dependent %g variable can be used in the select() string. The select() method can thus be used to slice thru the data set.

The acol(), arad(), bcol(), brad(), and tcol() methods change attributes of the atoms, bonds, and triangles displayed. Each atom or bond returned from the data object has an integer "type" from 1 to N. The type is used to index into a list of RGB colors or radii for atoms and bond thickness. The adef(), bdef(), and tdef() methods setup default mappings of types to colors/radii. The other methods enable specific assignments to be made. The interpolation option (e.g. many types assigned to a few colors), enables a smooth rainbow of colors to be assigned to a range of types. Note that the dump tool allows any vector to be returned as an atom "type" via its atype setting. Thus displayed colors can be keyed to atom velocity or other properties.

Colors are specified with a string, e.g. "red". There are 140 pre-defined colors from this WWW page which can be examined by importing the "colors" variable from "vizinfo". New colors can be defined by assigning a nickname of your choice to an RGB triplet, as described below.

Usage:

```
r = raster(d)               create Raster3d wrapper for data in d

  d = atom snapshot object (dump, data)

r.bg("black")               set background color (def = "black")
r.size(N)                   set image size to NxN
r.size(N,M)                 set image size to NxM
```

```
r.rotate(60,135)            view from z theta and azimuthal phi (def = 60,30)
r.shift(x,y)                translate by x,y pixels in view window (def = 0,0)
r.zoom(0.5)                 scale image by factor (def = 1)
r.box(0/1/2)                0/1/2 = none/variable/fixed box
r.box(0/1/2,"green")        set box color
r.box(0/1/2,"red",4)        set box edge thickness
r.file = "image"            file prefix for created images (def = "image")

r.show(N)                   show image of snapshot at timestep N

r.all()                     make images of all selected snapshots
r.all(P)                    images of all, start file label at P
r.all(N,M,P)                make M images of snapshot N, start label at P

r.pan(60,135,1.0,40,135,1.5)    pan during all() operation
r.pan()                         no pan during all() (default)

  args = z theta, azimuthal phi, zoom factor at beginning and end
  values at each step are interpolated between beginning and end values

r.select = "$x > %g*3.0"    string to pass to d.aselect.test() during all()
r.select = ""               no extra aselect (default)

  %g varies from 0.0 to 1.0 from beginning to end of all()

r.label(x,y,"h",size,"red","This is a label")    add label to each image
r.nolabel()                                      delete all labels

  x,y coords = -0.5 to 0.5, "h" or "t" for Helvetica or Times font
  size = fontsize (e.g. 10), "red" = color of text

r.acol(2,"green")                   set atom colors by atom type (1-N)
r.acol([2,4],["red","blue"])        1st arg = one type or list of types
r.acol(0,"blue")                    2nd arg = one color or list of colors
r.acol(range(20),["red","blue"])    if list lengths unequal, interpolate
r.acol(range(10),"loop")            assign colors in loop, randomly ordered

  if 1st arg is 0, set all types to 2nd arg
  if list of types has a 0 (e.g. range(10)), +1 is added to each value
  interpolate means colors blend smoothly from one value to the next

r.arad([1,2],[0.5,0.3])             set atom radii, same rules as acol()

r.bcol()                            set bond color, same args as acol()
r.brad()                            set bond thickness, same args as arad()

r.tcol()                            set triangle color, same args as acol()
r.tfill()                           set triangle fill, 0 fill, 1 line, 2 both

r.lcol()                            set line color, same args as acol()
r.lrad()                            set line thickness, same args as arad()

r.adef()                            set atom/bond/tri/line properties to default
r.bdef()                            default = "loop" for colors, 0.45 for radii
r.tdef()                            default = 0.25 for bond/line thickness
r.ldef()                            default = 0 fill

  by default 100 types are assigned
  if atom/bond/tri/line has type > # defined properties, is an error

from vizinfo import colors          access color list
```

```
print colors                        list defined color names and RGB values
colors["nickname"] = [R,G,B]        set new RGB values from 0 to 255

  140 pre-defined colors: red, green, blue, purple, yellow, black, white, etc
```

Related tools:

dump, gl, rasmol, svg

Prerequisites:

The Raster3d render and label3d programs.

sdata tool

Purpose:

Read, create, manipulate SPARTA surf files.

Description:

The cdata tool reads and writes SPARTA data files which contain surface element information. It enables the creation of geometric surface objects for input to SPARTA.

The sdata constructor reads in the specified SPARTA surface file. With no argument, an empty sdata object is created which can have objects added to it later, and then be written out.

An sdata object contains one or more "objects". Each object has a unique user-assigned ID. 2d objects are made of points and line segments. 3d objects are made of points and triangles. When objects are written to a SPARTA surface file, the list of points and lines or triangles are all that is written. The IDs are not written, though a SPARTA command assigns its own ID to the surface file.

The circle(), rect(), tri(), and spikycircle() methods create 2d objects, namely a circle, rectangle, triangle, and spiky circle. The latter is intially a perfect circle with each of its points displaced to random radial positions from Rmin to Rmax. The seed variable can be be set to seed the random number generator in a reproducible manner if desired.

The sphere(), box(), and spikysphere() methods create 3d objects, namely a sphere, rectangular box, and spiky sphere. The latter is intially a perfect sphere with each of its points displaced to random radial positions from Rmin to Rmax. The seed variable can be be set to seed the random number generator in a reproducible manner if desired.

The surf2d() and surf3d() methods create arbitrary 2d or 3d surface objects from lists of points and lists of lines or triangles. Note that to be compatible with SPARTA, objects should be closed and "watertight". This means that in 2d every point has exactly 2 lines connected to it. In 3d, every edge bewteen 2 points is part of exactly two triangles.

The center(), trans(), rotate(), and scale() methods are used to perform a geometric transformation on the points of an object. The union() method creates a new object with a new ID from a list of objects. The delete(), rename(), and copy() methods manipulate the IDs of previously defined objects.

Be default all objects are selected when created. The select() and unselect() methods can be used to select a subset of existing objects. The write() and append() methods write out selected objects to a file.

The iterator() and viz() methods are called by Pizza.py tools that visualize objects and particles (e.g. gl, raster, svg tools). Only selected objects are returned to the caller. A sdata file can be visualzed similarly to a snapshots from a dump file. In the case of an sdata file, there is only a single snapshot with index 0.

The grid() method can be used to define a 2d or 3d regular grid of lines that will be included in the output when the viz() method is called by other Pizza.py tools. This can correspond to the grid used by SPARTA to track particle motion.

Usage:

```
s = sdata()                          create a surf data object
s = sdata(ID,"mem.surf")             read in one or more SPARTA surf files
s = sdata(ID,"mem.part.gz mem.surf")  can be gzipped
s = sdata(ID,"mem.*")                wildcard expands to multiple files
s.read(ID,"mem.surf")                read in one or more data files

  all surf data in files becomes one surf with ID
  surf files contain the following kinds of entries in SPARTA format
    points and lines (for 2d), points and triangles (for 3d)
  read() has same argument options as constructor

s.seed = 48379                       set random # seed (def = 12345)
s.circle(ID,x,y,r,n)                 create a 2d circle with N lines and ID
s.rect(ID,x1,y1,x2,y2,nx,ny)         create a 2d rect, 2 corner pts, Nx,Ny segs
s.tri(ID,x1,y1,x2,y2,x3,y3,n1,n2,n3)  create a 2d tri, 3 pts, N1,N2,N3 segs
s.sphere(ID,x,y,z,r,n)               create a 3d sphere with NxN sqs per face
s.box(ID,x1,y1,z1,x2,y2,z2,nx,ny,nz)  3d box, 2 corner pts, Nx,Ny,Nz per face
s.spikycircle(ID,x,y,rmin,rmax,n)  2d circle, N lines, Rmin <= rad <= Rmax
s.spikysphere(ID,x,y,z,rmin,rmax,n)  3d sphere, NxN sqs, Rmin <= rad <= Rmax

  tri should be ordered so that (0,0,1) x (pt2-pt1) = outward normal
  spikycircle is same as circle, with each of N pts at random Rmin <rad  <Rmax
  spikysphere is same as sphere, with each surf pt at random Rmin  <rad  <Rmax

s.surf2d(ID,plist,llist)             create a custom 2d surf
s.surf3d(ID,plist,tlist)             create a custom 3d surf

  each point in plist is (x,y) for 2d or (x,y,z) for 3d
  each line in llist is (i,j) for C-style indices into plist
  each triangle in tlist is (i,j,k) for C-style indices into plist

s.center(ID,x,y,z)                   set center point of surf
s.trans(ID,dx,dy,dz)                 translate surf and its center point
s.rotate(ID,theta,Rx,Ry,Rz)          rotate surf by theta around R vector
s.scale(ID,sx,sy,sz)                 scale a surf around center point
s.invert(ID)                         invert normal direction of surf

  default center for created surfs is the x,y,z or geometric center
  default center for read-in surf is center of bounding box of all points
  rotation and scaling of surf are relative to its center point

s.join(ID,id1,id2,...)               combine id1,id2,etc into new surf with ID
s.delete(id1,id2,...)                delete one or more surfs
s.rename(ID,IDnew)                   rename a surf
s.copy(ID,IDnew)                     create a new surf as copy of old surf

  join does not delete id1,id2,etc
  center for joined surf becomes center of bounding box of all points

s.select(id1,id2,...)                select one or more surfs
s.select()                           select all surfs
s.unselect(id1,id2,...)              unselect one or more surfs
s.unselect()                         unselect all surfs

  selection applies to write() and viz()
  surfs are selected by default when read or created

s.write("file")                      write all selected surfs to SPARTA file
s.write("file",id1,id2,...)          write only listed & selected surfs to file

s.grid(xlo,xhi,ylo,yhi,ny,ny)        bounding box and Nx by Ny grid
```

```
s.grid(xlo,xhi,ylo,yhi,zlo,zhi,ny,ny,nz)   ditto for 3d
s.gridfile(xlo,xhi,ylo,yhi,file)   bbox and SPARTA-formatted parent grid file
s.gridfile(xlo,xhi,ylo,yhi,zlo,zhi,file)   ditto for 3d

  grid command superpose a grid, for viz only
  also changes bounding box to xyzlo to xyzhi

index,time,flag = s.iterator(0/1)          loop over single snapshot
time,box,atoms,bonds,tris,lines = s.viz(index)   return list of viz objects

  iterator() and viz() are compatible with equivalent dump calls
  iterator() called with arg = 0 first time, with arg = 1 on subsequent calls
    index = timestep index within dump object (only 0 for data file)
    time = timestep value (only 0 for data file)
    flag = -1 when iteration is done, 1 otherwise
  viz() returns info for selected objs for specified timestep index (must be 0)
    time = 0
    box = [xlo,ylo,zlo,xhi,yhi,zhi] = bounding box
    atoms = NULL
    bonds = NULL
    tris = id,type,x1,y1,z1,x2,y2,z2,x3,y3,z3,nx,ny,nz for each tri as 2d array
      NULL if triangles do not exist
    lines = id,type,x1,y1,z1,x2,y2,z2 for each line as 2d array
      NULL if lines do not exist
    types are assigned to each surf in ascending order
```

Related tools: none

Prerequisites: none

svg tool

Purpose:

3d visualization via SVG files.

Description:

The svg tool creates Scalable Vector Graphics (SVG) files from atom or mesh snapshots. The SVG format can be displayed or animated by various tools and is a popular format for viewing solid state lattices.

The svg constructor takes a data object containing atom or mesh snapshots as an argument (dump, data, cdata, mdump).

The show() method creates an image.svg file for snapshot N. The all() method loops thru all selected snapshots and creates the image files image0000.svg, image0001.svg, etc. The prefix "image" can be changed via the file setting.

The bg(), size(), rotate(), trans(), zoom(), box(), label(), and nolabel() methods control various aspects of the images produced. Without the trans() and zoom() settings, the Raster3d image should roughly fill the window and be centered.

Additional movie effects can be produced using the pan() and select() methods. The pan() method specifies an initial and final viewpoint that is applied to the images produced by the all() method. For intermediate images, the view parameters will be interpolated between their initial and final values. The pan() method can thus be used to rotate a single image or fly-by the simulation as it animates. The select() method performs additional atom selection for each image of the all() sequence. An image-dependent %g variable can be used in the select() string. The select() method can thus be used to slice thru the data set.

The acol(), arad(), bcol(), brad(), and tcol() methods change attributes of the atoms, bonds, and triangles displayed. Each atom or bond returned from the data object has an integer "type" from 1 to N. The type is used to index into a list of RGB colors or radii for atoms and bond thickness. The adef(), bdef(), and tdef() methods setup default mappings of types to colors/radii. The other methods enable specific assignments to be made. The interpolation option (e.g. many types assigned to a few colors), enables a smooth rainbow of colors to be assigned to a range of types. Note that the dump tool allows any vector to be returned as an atom "type" via its atype setting. Thus displayed colors can be keyed to atom velocity or other properties.

Colors are specified with a string, e.g. "red". There are 140 pre-defined colors from this WWW page which can be examined by importing the "colors" variable from "vizinfo". New colors can be defined by assigning a nickname of your choice to an RGB triplet, as described below.

Usage:

```
s = svg(d)                  create SVG object for data in d

  d = atom snapshot object (dump, data)

s.bg("black")               set background color (def = "black")
s.size(N)                   set image size to NxN
s.size(N,M)                 set image size to NxM
s.rotate(60,135)            view from z theta and azimuthal phi (def = 60,30)
```

```
s.shift(x,y)                translate by x,y pixels in view window (def = 0,0)
s.zoom(0.5)                 scale image by factor (def = 1)
s.box(0/1/2)                0/1/2 = none/variable/fixed box
s.box(0/1/2,"green")        set box color
s.box(0/1/2,"red",4)        set box edge thickness
s.file = "image"            file prefix for created images (def = "image")

s.show(N)                   show image of snapshot at timestep N

s.all()                     make images of all selected snapshots
s.all(P)                    images of all, start file label at P
s.all(N,M,P)                make M images of snapshot N, start label at P

s.pan(60,135,1.0,40,135,1.5)    pan during all() operation
s.pan()                         no pan during all() (default)

  args = z theta, azimuthal phi, zoom factor at beginning and end
  values at each step are interpolated between beginning and end values

s.select = "$x > %g*3.0"    string to pass to d.aselect.test() during all()
s.select = ""               no extra aselect (default)

  %g varies from 0.0 to 1.0 from beginning to end of all()

s.label(x,y,"h",size,"red","This is a label")    add label to each image
s.nolabel()                                      delete all labels

  x,y coords = -0.5 to 0.5, "h" or "t" for Helvetica or Times font
  size = fontsize (e.g. 10), "red" = color of text

s.acol(2,"green")                   set atom colors by atom type (1-N)
s.acol([2,4],["red","blue"])        1st arg = one type or list of types
s.acol(0,"blue")                    2nd arg = one color or list of colors
s.acol(range(20),["red","blue"])    if list lengths unequal, interpolate
s.acol(range(10),"loop")            assign colors in loop, randomly ordered

  if 1st arg is 0, set all types to 2nd arg
  if list of types has a 0 (e.g. range(10)), +1 is added to each value
  interpolate means colors blend smoothly from one value to the next

s.arad([1,2],[0.5,0.3])             set atom radii, same rules as acol()

s.bcol()                            set bond color, same args as acol()
s.brad()                            set bond thickness, same args as arad()

s.tcol()                            set triangle color, same args as acol()
s.tfill()                           set triangle fill, 0 fill, 1 line, 2 both

s.lcol()                            set line color, same args as acol()
s.lrad()                            set line thickness, same args as arad()

s.adef()                            set atom/bond/tri/line properties to default
s.bdef()                            default = "loop" for colors, 0.45 for radii
s.tdef()                            default = 0.25 for bond/line thickness
s.ldef()                            default = 0 fill

  by default 100 types are assigned
  if atom/bond/tri/line has type > # defined properties, is an error

from vizinfo import colors          access color list
print colors                        list defined color names and RGB values
```

```
colors["nickname"] = [R,G,B]       set new RGB values from 0 to 255

  140 pre-defined colors: red, green, blue, purple, yellow, black, white, etc

Settings specific to svg tool:

s.thick = 2.0               pixel thickness of black atom border
```

Related tools:

dump, gl, raster, rasmol

Prerequisites:

Display program for viewing *.svg image files.

tdump tool

Purpose:

Read dump files with triangle info.

Description:

The tdump tool reads one or more LAMMPS dump files, and stores their contents as a series of snapshots with 2d arrays of atom attributes. It is assumed that each atom contains triangle segment info from a LAMMPS simulation using atom_style tri. Other tools use tdump objects to extract triangle info for visualization, like the dump tool via its extra() method.

The constructor method is passed a string containing one or more dump filenames. They can be listed in any order since snapshots are sorted by timestep after they are read and duplicate snapshots (with the same time stamp) are deleted. If a 2nd argument is specified, the files are not immediately read, but snapshots can be read one-at-a-time by the next() method.

The map() method assigns names to columns of atom attributes. The id,type, corner1x,corner1y,corner1z, corner2x,corner2y,corner2z, corner3x,corner3y,corner3z names must be assigned in order for line segment info to be extracted.

The viz() method is called by Pizza.py tools that visualize snapshots of atoms (e.g. gl, raster, svg tools).

Normally, LAMMPS creates the dump files read in by this tool. If you want to create them yourself, the format of LAMMPS dump files is simple. Each snapshot is formatted as follows:

```
ITEM: TIMESTEP
100
ITEM: NUMBER OF ATOMS
32
ITEM: BOX BOUNDS
0 3.35919
0 3.35919
0 7.50
ITEM: ATOMS
1 1 0 0 0
2 1 0.25 0.25 0
3 1 0.25 0 0.25
...
N 3 0.7 0.5 0.6
```

The box bounds are listed as xlo xhi on the 1st line, ylo yhi on the next line, zlo zhi on the last. There are N lines following "ITEM: ATOMS" where N is the number of atoms. Atoms do not have to be listed in any particular order. There can be a different number of atoms in each snapshot. Each line must contain the atom ID, type, and the end points of the associated line segment, as specified by the map() command.

Usage:

```
t = tdump("dump.one")              read in one or more dump files
t = tdump("dump.1 dump.2.gz")      can be gzipped
t = tdump("dump.*")                wildcard expands to multiple files
t = tdump("dump.*",0)              two args = store filenames, but don't read
```

```
  incomplete and duplicate snapshots are deleted
  no column name assignment is performed

time = t.next()                   read next snapshot from dump files

  used with 2-argument constructor to allow reading snapshots one-at-a-time
  snapshot will be skipped only if another snapshot has same time stamp
  return time stamp of snapshot read
  return -1 if no snapshots left or last snapshot is incomplete
  no column name assignment is performed

t.map(1,"id",3,"x")               assign names to atom columns (1-N)

  must assign id,type,corner1x,corner1y,corner1z,corner2x,corner2y,corner2z,corner3x,corner3y,corner

time,box,atoms,bonds,tris,lines = t.viz(index)   return list of viz objects

  viz() returns line info for specified timestep index
    can also call as viz(time,1) and will find index of preceding snapshot
    time = timestep value
    box = \[xlo,ylo,zlo,xhi,yhi,zhi\]
    atoms = NULL
    bonds = NULL
    tris = id,type,x1,y1,z1,x2,y2,z2,x3,y3,z3 for each tri as 2d array
      id,type are from associated atom
    lines = NULL

t.owrap(...)                      wrap tris to same image as their atoms

  owrap() is called by dump tool's owrap()
  useful for wrapping all molecule's atoms/tris the same so it is contiguous
```

Related tools:

dump, gl, raster, svg

Prerequisites:

Numeric or NumPy Python packages. Gunzip command (if you want to read gzipped files).

vcr tool

Purpose:

VCR-style GUI for 3d interactive OpenGL visualization.

Description:

The vcr tool displays a GUI to do 3d interactive visualization of LAMMPS snapshots or data files. It is a wrapper on the gl tool which draws the individual images in OpenGL windows

The vcr constructor creates the GUI. Note that multiple OpenGL windows can be run by the same GUI so that multiple views of the same data set can be manipulated simultaneously (or views of different data sets so long as they have the same # of snapshots).

The view can be controlled by the GUI widgets or by invoking the tool methods: play(), stop(), axis(), etc. The frame slider can be dragged to view a desired frame. The mouse can also be used in the OpenGL window to translage, rotate, or zoom the scene. The clipping sliders or methods can be used to narrow the view of displayed data, though their interactivity can be slow for scenes with lots of data.

The reload() method is needed if you change the selection attributes of the underlying data assigned to the gl tool, such as via the dump tool's methods. These changes will not be visible in the OpenGL windows until the data is reloaded.

The save() method will save the current OpenGL window contents to a PNG file. If multiple OpenGL windows are being used, multiple files will be created. The save-all checkbox or method will store one file per snapshot if the Play or Back buttons are used to start an animation.

Usage:

```
v = vcr(gl1,gl2,...)       start vcr GUI with one or more gl windows
v.add(gl)                  add a gl window to vcr GUI

Actions (same as GUI widgets):

v.first()                  go to first frame
v.prev()                   go to previous frame
v.back()                   play backwards from current frame to start
v.stop()                   stop on current frame
v.play()                   play from current frame to end
v.next()                   go to next frame
v.last()                   go to last frame

v.frame(31)                set frame slider
v.delay(0.4)               set delay slider
v.q(5)                     set quality slider

v.xaxis()                  view scene from x axis
v.yaxis()                  view scene from y axis
v.zaxis()                  view scene from z axis
v.box()                    toggle bounding box
v.axis()                   toggle display of xyz axes
v.norm()                   recenter and resize the view
v.ortho()                  toggle ortho/perspective button
```

```
v.reload()                reload all frames from gl viewer data files

v.clipxlo(0.2)            clip scene at x lo fraction of box
v.clipxhi(1.0)            clip at x hi
v.clipylo(0.2)            clip in y
v.clipyhi(1.0)
v.clipzlo(0.2)            clip in z
v.clipzhi(1.0)

v.save()                  save current scene to file.png
v.file("image")           set filename
v.saveall()               toggle save-all checkbox
```

Related tools:

animate, gl

Prerequisites:

Python Tkinter package.

vec tool

Purpose:

Create numeric vectors from columns in file or list of vecs.

Description:

The vec tool creates numeric vectors that can be extracted, written to files, or imported into other tools like plotview for plotting.

The vec constructor takes either a file or list argument. For a file argument, columns of numeric data are read from the file. Blank lines or lines that do not start with a numeric character (0123456789.-) are skipped. Each column must have the same number of values. For a list argument, each element of the list is assumed to be a vector of values (i.e. the list is a list of lists), as shown in the example below.

The columns of data may be accessed by number (1-N) or by name ("col1" thru "colN").

The get() and write() methods extract the numeric vectors of data.

Usage:

```
v = vec("file1")                    read in numeric vectors from a file
v = vec(array)                      array = list of numeric vectors

  skip blank lines and lines that start with non-numeric characters
  example array with 2 vecs = [[1,2,3,4,5], [10,20,30,40,50]]
  assigns names = "col1", "col2", etc

nvec = v.nvec                       # of vectors
nlen = v.nlen                       lengths of vectors
names = v.names                     list of vector names
x,y,... = l.get(1,"col2",...)       return one or more vectors of values
l.write("file.txt")                 write all vectors to a file
l.write("file.txt","col1",7,...)    write listed vectors to a file

  get and write allow abbreviated (uniquely) vector names or digits (1-Nvec)
```

Related tools:

data, dump

Prerequisites: none

vmd tool

Purpose:

Control VMD from python.

Description:

The vmd tool is a simple wrapper on the VMD visualization package, a popular tool for visualizing snapshots from molecular dynamics simulations.

The vmd constructor invokes VMD in its own window. Data can be passed to it for visualization via the new(), data(), replace(), and append() methods. The other methods permit more specialized interactions with VMD through its command syntax.

Usage:

```
v = vmd()                       start up VMD
v.stop()                        shut down VMD instance
v.clear()                       delete all visualizations

v.rep(style)                    set default representation style. One of
                                (Lines|VDW|Licorice|DynamicBonds|Points|CPK)
v.new(file[,type])              load new file (default file type 'lammpstrj')
v.data(file[,atomstyle])        load new data file (default atom style 'full')
v.replace(file[,type])          replace current frames with new file
v.append(file[,type])           append file to current frame(s)
v.set(snap,x,y,z,(True|False)) set coordinates from a pizza.py snapshot to new or current frame

v.frame(frame)                  set current frame
v.flush()                       flush pending input to VMD and update GUI
v.read(file)                    read Tcl script file (e.g. saved state)

v.enter()                       enter interactive shell
v.debug([True|False])           display generated VMD script commands?
```

Related tools:

gl

Prerequisites:

Python pexpect package.

vtk tool

Purpose:

Convert LAMMPS snapshots to VTK format.

Description:

The vtk tool converts atom snapshots in a LAMMPS dump or data file to the VTK format used by various visualization packages.

The vtk constructor takes an object that stores atom snapshots (dump, data) as its first argument. The atom snapshots must have "id", "type", "x", "y", and "z" defined; see the map() methods of those tools.

The one(), many(), and single() methods convert specific snapshots to the VTK format and write them out. Optionally, a file prefix for the XYZ output files can also be specified. A ".xyz" suffix will be appended to all output files.

Usage:

```
v = vtk(d)              d = object containing atom coords (dump, data)

v.one()                 write all snapshots to tmp.vtk
v.one("new")            write all snapshots to new.vtk
v.many()                write snapshots to tmp0000.vtk, tmp0001.vtk, etc
v.many("new")           write snapshots to new0000.vtk, new0001.vtk, etc
v.single(N)             write snapshot for timestep N to tmp.vtk
v.single(N,"file")      write snapshot for timestep N to file.vtk

  surfaces in snapshot will be written to SURF1.vtk, SURF2.vtk, etc
    where each surface (triangle type) is in a different file
```

Related tools:

cfg, data, dump, ensight, xyz

Prerequisites: none

xyz tool

Purpose:

Convert LAMMPS snapshots to XYZ format.

Description:

The xyz tool converts atom snapshots in a LAMMPS dump or data file to the XYZ format used by various visualization packages.

The xyz constructor takes an object that stores atom snapshots (dump, data) as its first argument. The atom snapshots must have "id", "type", "x", "y", and "z" defined; see the map() methods of those tools.

The one(), many(), and single() methods convert specific snapshots to the XYZ format and write them out. Optionally, a file prefix for the XYZ output files can also be specified. A ".xyz" suffix will be appended to all output files.

Usage:

```
x = xyz(d)              d = object containing atom coords (dump, data)

x.one()                 write all snapshots to tmp.xyz
x.one("new")            write all snapshots to new.xyz
x.many()                write snapshots to tmp0000.xyz, tmp0001.xyz, etc
x.many("new")           write snapshots to new0000.xyz, new0001.xyz, etc
x.single(N)             write snapshot for timestep N to tmp.xyz
x.single(N,"file")      write snapshot for timestep N to file.xyz
```

Related tools:

cfg, data, dump, ensight, vtk

Prerequisites: none