Predictive Modeling of Viscosity: A Novel Approach Using Data Science Techniques

Background:

Viscosity (η) measures the fluid's resistance to flow[i]. It is a very fundamental property of materials as it influences almost every physical property such as softening and melting[ii] etc. Our group used the potential energy approach to find the viscosity at low stresses[iii].
Very small stresses give rise to very small strain rates as well. As can be seen in **Figure 13** the plot for the logarithmic viscosity scales linearly with σ^2 at very low stresses and diverges at large stresses[35]. Low strain rates are very hard to measure in an MD simulation. That's because the change in the simulation box is very minute and hard to quantify. However, after shearing as shown in schematic **Figure 14** every atom breaks bonds and forms other bonds and/or maintains previous bonds. The total number of events occurring at the atomic level is the sum of bond breaking and bond forming events. Since the overall strain has to be a total sum of those individual events, the aim is to be able to relate the total events to the strain rate/time as shown in schematic **Figure 15**.

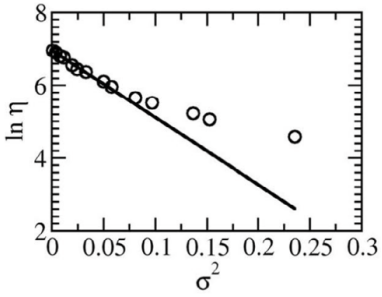

Figure 13. Plot of the logarithmic viscosity as a function of the square of stress.

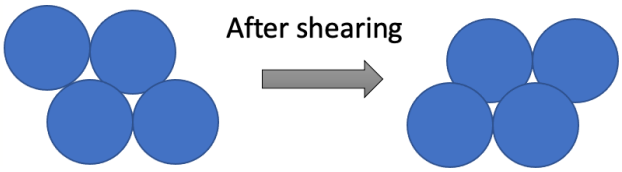

Figure 14. Schematic showing atoms breaking and forming bonds after shearing.

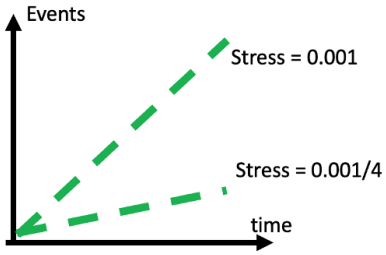

Figure 15. Schematic plot showing number of events scaling linearly with time at two different stresses.

Code description:

To achieve this, I wrote the **Python code** below. Broadly it performs two tasks:

1. It would generate the species dependent radial distribution function for the system at every frame so that the total number of nearest neighbors for every atom at every frame dumped can be discerned.
2. It would track every atom throughout the duration of the simulation at every frame and calculates the new total number of neighbors as the atom moves to a new location and compares it to the initial neighbors i.e. at frame 0. It then calculates the total new events for the entire system.

I plot these two tasks as shown in **Figure 16** and **Figure 17.** Once I gather all this data and calibrate the system, I test it at different dumps. I can then scale the events with the strain rate and use that to calculate the viscosity at very small strain rates that would otherwise cannot be quantified accurately.

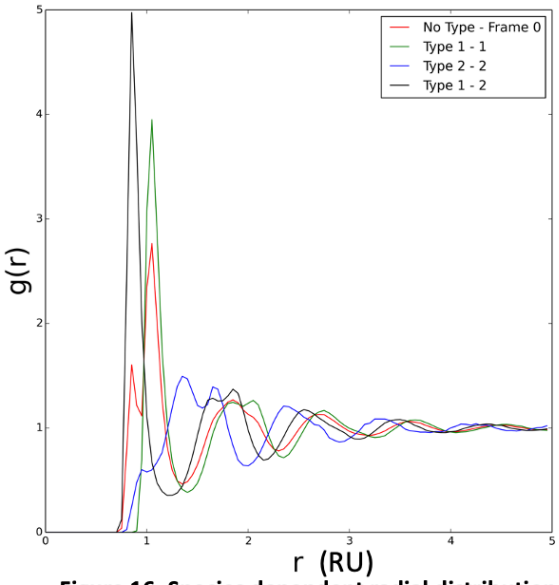

Figure 16. Species dependent radial distribution functions at initial frame generated by code.

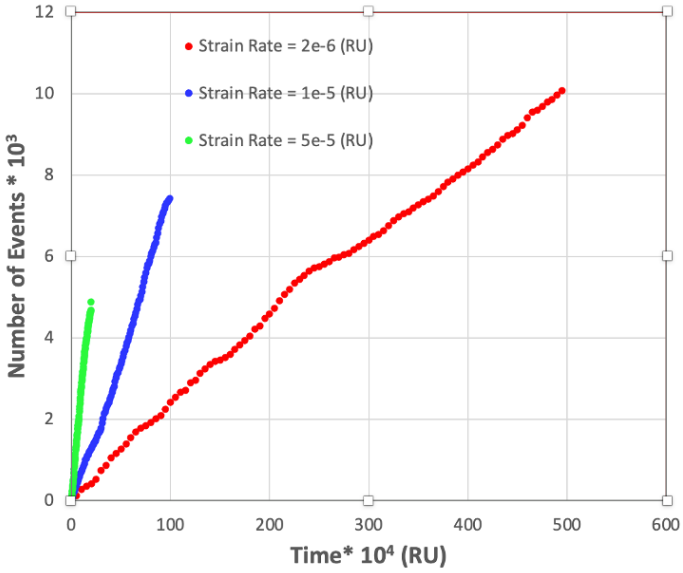

Figure 17. Plot shows the number of events as a function of time at three different strain rates.

---

[i] Gresham RM. Viscosity: A fluid's resistance to flow. Tribology & lubrication technology. 2008 Nov 1;64(11):55.
[ii] Dinsdale AT, Quested PN. The viscosity of aluminium and its alloys--A review of data and models. Journal of materials science. 2004 Dec;39(24):7221-8.
[iii] Zhang Y, Huang L, Shi Y. Molecular dynamics study on the viscosity of glass-forming systems near and below the glass transition temperature. Journal of the American Ceramic Society. 2021 Dec;104(12):6227-41.

```
#!/usr/bin/env python3

import os
import sys
import re
import time
import collections
import pathlib
import numpy as np
from scipy.spatial import KDTree
from scipy.signal import find_peaks
from scipy.signal import savgol_filter
import matplotlib.pyplot as plt
import argparse

Timestep = collections.namedtuple('Timestep', 'at atom_id atom_type atom_pos bounds
no_of_atoms progress')

def read_timesteps(filename):
    '''Read a LAMMPS input file

    This function reads in the MD simulation and yields each
    timestep as a Timestep() named tuple.
    '''
    size = os.path.getsize(filename)
    f = open(filename)
    no_of_atoms = None
    while True:
        line = f.readline()
        if not line:
            break
        line = line.rstrip()
        if line == 'ITEM: TIMESTEP':
            ts_name = int(f.readline().rstrip())
        elif line == 'ITEM: NUMBER OF ATOMS':
            no_of_atoms = int(f.readline().rstrip())
            atom_id = np.empty(no_of_atoms, dtype=int)
            atom_type = np.empty(no_of_atoms, dtype=int)
            atom_pos = np.empty((no_of_atoms, 3), dtype=float)
        elif line == 'ITEM: BOX BOUNDS pp pp pp':
            x0, x1 = map(float, f.readline().split())
            y0, y1 = map(float, f.readline().split())
            z0, z1 = map(float, f.readline().split())
            bounds = np.array([[x0, x1], [y0, y1], [z0, z1]])
        elif line == 'ITEM: ATOMS id type x y z vx vy vz fx fy fz':
            for i in range(no_of_atoms):
                line = f.readline()
                parts = line.split()
```

```
                atom_id[i] = int(parts[0])
                atom_type[i] = int(parts[1])
                atom_pos[i] = float(parts[2]), float(parts[3]), float(parts[4])
            yield Timestep(ts_name, atom_id, atom_type, atom_pos, bounds,
no_of_atoms, f.tell()/size)
        else:
            print('WARNING!!! SKIPPING UNKNOWN LAMPPS INPUT', line)

def find_r11_or_f12(r, rdf):
    '''Find the first valley next to the largest peak given the rdf'''
    smooth_rdf = savgol_filter(rdf, len(rdf)//40*2+1, 4) # smooth to remove noise
    argmax = smooth_rdf.argmax() # find the largest peak value
    valleys, props = find_peaks(-smooth_rdf) # find all valleys
    idx = valleys[valleys > argmax][0] # find index of the first valley after the
        largest peak
    return r[idx]

def find_r22(r, rdf):
    '''Find the position halfway the first two peaks in the rdf'''
    smooth_rdf = savgol_filter(rdf, len(rdf)//40*2+1, 4) # smooth to remove noise
    peaks,_info = find_peaks(smooth_rdf) # find all peaks
    peaks = peaks[smooth_rdf[peaks] > 0.1] # remove small peaks (filter introduces
        small bump at start)
    first, second = peaks[:2] # take the first two peaks
    return (r[first] + r[second]) / 2 # average between them

def rdf_for_timestep(atom_type, atom_pos, bounds, cutoff, bin_edges):
    '''Calculate the rdf for atom types 1 and 2

    atom_type:  array of int where 1 is type 1 and 2 is type 2
    atom_pos:   array of (natoms,3) with the 3-dimensional positions
    bounds:     array of (3, 2) with for each dimension (x, y, z) the low and high
value
    cutoff:     maximum atom-atom distance for rdf calculation (maximum is half the
box size)
    bin_edges:  array of the edges of the radius discretization for the rdf
    '''
    assert cutoff < bounds.ptp(1).min() / 2
    boxsize = bounds.ptp(1)
    if (bounds[:,0] != 0).any():
        # memory optimization: don't remove left border if its 0
        atom_pos = atom_pos - bounds[:,0]
    # quickly find pairs
    tree = KDTree(atom_pos%boxsize, boxsize=boxsize)
    pairs = tree.query_pairs(cutoff, output_type='ndarray')
    # find distances
    pos = atom_pos[pairs] # (npairs, 2, 3) array of position for each pair
    delta = abs(pos[:,0,:] - pos[:,1,:]) # 0 and 1 refer to sides of the pairs,
```

```
        abs() to make periodic bounds easier
    m0 = delta[:,0] > boxsize[0]/2; delta[m0,0] = boxsize[0] - delta[m0,0] #fix
periodic boundaries (x)
    m1 = delta[:,1] > boxsize[1]/2; delta[m1,1] = boxsize[1] - delta[m1,1] #fix
periodic boundaries (y)
    m2 = delta[:,2] > boxsize[2]/2; delta[m2,2] = boxsize[2] - delta[m2,2] #fix
periodic boundaries (z)
    r = np.linalg.norm(delta, axis=1) # r is the distance for each pair
    # global rdf
    counts, _edges = np.histogram(r, bins=bin_edges)
    # typed rdf
    pair_types = atom_type[pairs]
    p11 = (pair_types[:,0] == 1) & (pair_types[:,1] == 1)
    p12 = (pair_types[:,0] == 1) & (pair_types[:,1] == 2)
    p22 = (pair_types[:,0] == 2) & (pair_types[:,1] == 2)
    counts11, _edges = np.histogram(r[p11], bins=bin_edges)
    counts12, _edges = np.histogram(r[p12], bins=bin_edges)
    counts22, _edges = np.histogram(r[p22], bins=bin_edges)
    # normalize
    dr = bin_edges[1] - bin_edges[0]
    r = bin_edges[:-1]
    V = np.product(boxsize)
    n = len(atom_pos)
    n1 = (atom_type == 1).mean()
    n2 = (atom_type == 2).mean()
    rho = len(atom_pos) / V
    rdf, rdf11, rdf12, rdf22 = np.zeros((4, len(counts)))
    rdf[1:] = counts[1:] / (4 * np.pi * r**2 * dr * n * n / V)[1:]
    rdf *= 2 # we're double counting because of pairs
    rdf11[1:] = 2 * counts11[1:] / (4 * np.pi * r**2 * dr * n * n / V)[1:] / (n1 *
n1)
    rdf12[1:] = 2 * counts12[1:] / (4 * np.pi * r**2 * dr * n * n / V)[1:] / (n1 *
n2)
    rdf22[1:] = 2 * counts22[1:] / (4 * np.pi * r**2 * dr * n * n / V)[1:] / (n2 *
n2)
    return rdf, rdf11, rdf12, rdf22

class AtomsByType:
    def __init__(self, ts, tid):
        '''AtomsByType is class that selects atoms from a Timestep() given a type

        and builds a KDTree() with periodic boundaries given the boxsize

        ts:     Timestep() tuple
        tid:    type id (either 1 or 2)
        '''
        self.tid = tid
        mask = ts.atom_type == tid
```

```
        self.pos = ts.atom_pos[mask]
        self.id = ts.atom_id[mask]
        self.tree = KDTree((self.pos - ts.bounds[:,0])%ts.bounds.ptp(1),
boxsize=ts.bounds.ptp(1))

class FindNeighbours:
    def __init__(self, a, b, r):
        '''FindNeighbours is a class that performs the distance query
        and stores the results.

        a: object of AtomsByType() (eg all atoms of type 1)
        b: object of AtomsByType() (eg all atoms of type 2)
        r: the cutoff ratius to consider two atoms in a bond
        '''
        pairs = a.tree.query_ball_tree(b.tree, r)
        # BEGIN slow part
        self.bonds = set()
        for idx_a, nn_b in enumerate(pairs):
            if a is b: nn_b = list(set(nn_b) - {idx_a})
            if not nn_b: continue
            pos_a = a.pos[idx_a]
            poss_b = b.pos[nn_b]
            id_a = a.id[idx_a]
            ids_b = b.id[nn_b]
            for id_b in ids_b:
                if a is b:
                    if id_a > id_b:
                        self.bonds.add((id_b, id_a))
                    else:
                        self.bonds.add((id_a, id_b))
                else:
                    self.bonds.add((id_a, id_b))
        # END slow part

def main():
    main_start_time = time.time()
    args = parse_args()
    verbose = not args.quiet
    filename = str(getattr(args, 'input-file'))
    rdf_cutoff = args.cutoff
    rdf_nbins = args.nbins

    r11 = args.r11
    r12 = args.r12
    r22 = args.r22

    nn11prev = None
    nn12prev = None
```

```
    nn22prev = None

    ### This part deals with the RDF calculation and estimating r11/r12/r22
    # (RDF) First, we open the file to write to and calculate discretization bins
    if args.rdf.max >= 0:
        if args.rdf_file is None and args.rdf_plot is None:
            print('Warning! not saving rdf results')
            print('Pass --rdf-file or --rdf-plot if you want to save the rdf
calculation')
        bin_edges = np.linspace(0, rdf_cutoff, rdf_nbins+1)
        rdf_r = bin_edges[:-1]
        if args.rdf_file is not None:
            rdf_file = open(args.rdf_file, 'w')
            print('frame', 'timestep', 'subset', 'rxx', *rdf_r, file=rdf_file,
flush=True, sep=',')
            if verbose:
                print('wrote rdf header to file', args.rdf_file)
    rdf_means = [[], [], [], []]
    rxx_means = [[], [], []]
    # (RDF) Then, we loop over the selected timesteps
    for frameno, ts in enumerate(read_timesteps(filename)):
        if frameno > args.rdf.max:
            break
        if not args.rdf(frameno):
            continue
        last_time = time.time()
        rdf, rdf11, rdf12, rdf22 = rdf_for_timestep(
                    ts.atom_type, ts.atom_pos, ts.bounds, rdf_cutoff, bin_edges)
        new_time = time.time()
        if verbose:
            print(f'frame {frameno} / filepos {ts.progress*100:.1f}%: calculating
rdf (took {new_time-last_time:.2f}s)')
        # record rdf's for averaging
        rdf_means[0].append(rdf)
        rdf_means[1].append(rdf11)
        rdf_means[2].append(rdf12)
        rdf_means[3].append(rdf22)
        # record r11/r12/r22 for plotting and averaging
        rxx_means[0].append(find_r11_or_f12(rdf_r, rdf11))
        rxx_means[1].append(find_r11_or_f12(rdf_r, rdf12))
        rxx_means[2].append(find_r22(rdf_r, rdf22))
        if args.rdf_file is not None:
            print(frameno, ts.at, 'all', '-', *rdf, file=rdf_file, flush=True,
sep=',')
            print(frameno, ts.at, '11', rxx_means[0][-1], *rdf11, file=rdf_file,
flush=True, sep=',')
            print(frameno, ts.at, '12', rxx_means[1][-1], *rdf12, file=rdf_file,
flush=True, sep=',')
```

```
            print(frameno, ts.at, '22', rxx_means[2][-1], *rdf22, file=rdf_file,
flush=True, sep=',')
    # (RDF) Finally, we estimate the r11/r12/r22 and write the datafile / plotfile
    if args.rdf.max >= 0:
        rdf_means = np.array(rdf_means)
        if verbose:
            print(f'calculating mean rdf')
        rdfmean =   rdf_means[0].mean(0)
        rdf11mean = rdf_means[1].mean(0)
        rdf12mean = rdf_means[2].mean(0)
        rdf22mean = rdf_means[3].mean(0)
        r11estim = np.mean(rxx_means[0])
        r12estim = np.mean(rxx_means[1])
        r22estim = np.mean(rxx_means[2])
        if args.r11 == 'estimate':
            r11 = r11estim
        if args.r12 == 'estimate':
            r12 = r22estim
        if args.r22 == 'estimate':
            r22 = r22estim
        if verbose:
            print(f'estimate for r11: {r11estim:.4f}')
            print(f'estimate for r12: {r12estim:.4f}')
            print(f'estimate for r22: {r22estim:.4f}')
        if args.rdf_file is not None:
            print('avg', 'avg', 'all', '-', *rdfmean, file=rdf_file, flush=True,
sep=',')
            print('avg', 'avg', '11', r11estim, *rdf11mean, file=rdf_file,
flush=True, sep=',')
            print('avg', 'avg', '12', r12estim, *rdf12mean, file=rdf_file,
flush=True, sep=',')
            print('avg', 'avg', '22', r22estim, *rdf22mean, file=rdf_file,
flush=True, sep=',')
            if verbose:
                print('closing rdf file', args.rdf_file)
            rdf_file.close()
        if args.rdf_plot is not None:
            plt.plot(rdf_r, rdfmean, label='all')
            line, = plt.plot(rdf_r, rdf11mean, label='11')
            plt.axvline(r11estim, color=line.get_color(), alpha=0.5)
            line, = plt.plot(rdf_r, rdf12mean, label='12')
            plt.axvline(r12estim, color=line.get_color(), alpha=0.5)
            line, = plt.plot(rdf_r, rdf22mean, label='22')
            plt.axvline(r22estim, color=line.get_color(), alpha=0.5)
            plt.title(f'RDF plot (averaged over {rdf_means.shape[1]} frames)')
            plt.legend()
            plt.xlabel('r')
            plt.ylabel('RDF(r)')
```

```
        plt.savefig(args.rdf_plot)
        plt.clf()

if r11 == 'estimate' or r12 == 'estimate' or r22 == 'estimate':
    print('either provide --r11/--r12/--r22 or provie rdf frames to estimate')
    exit()
r11 = float(r11)
r12 = float(r12)
r22 = float(r22)

### This part deals with the events
# (EVENTS) First, we create our output csv file
if args.events.max >= 0:
    file_events = open(args.events_file, 'w')
    print('frame', 'timestep', 'n11_added', 'n11_removed', 'n11_events',
            'n12_added', 'n12_removed', 'n12_events', 'n22_added',
            'n22_removed', 'n22_events', 'total_added', 'total_removed',
            'total_events', flush=True, sep=',', file=file_events)
    if verbose:
        print('wrote events header to file', args.events_file)
# (EVENTS) Then, we loop over the selected timesteps
for frameno, ts in enumerate(read_timesteps(filename)):
    if frameno > args.events.max:
        break
    if not args.events(frameno):
        continue
    # separate atom types
    type1 = AtomsByType(ts, 1)
    type2 = AtomsByType(ts, 2)
    # find neighbours
    nn11 = FindNeighbours(type1, type1, r11)
    nn12 = FindNeighbours(type1, type2, r12)
    nn22 = FindNeighbours(type2, type2, r22)
    # then we compare the previous bond list with the new bond list
    if nn11prev is not None:
        # record different event types
        n11_added = len(nn11.bonds - nn11prev.bonds)
        n11_removed = len(nn11prev.bonds - nn11.bonds)
        n11_events = n11_added + n11_removed
        n12_added = len(nn12.bonds - nn12prev.bonds)
        n12_removed = len(nn12prev.bonds - nn12.bonds)
        n12_events = n12_added + n12_removed
        n22_added = len(nn22.bonds - nn22prev.bonds)
        n22_removed = len(nn22prev.bonds - nn22.bonds)
        n22_events = n22_added + n22_removed
        new_time = time.time()
        if verbose:
            print(f'frame {frameno} / filepos {ts.progress*100:.1f}%: writing
```

```
events (took {new_time-last_time:.2f}s)')
            # and write to the file
            print(frameno, ts.at,
                    n11_added, n11_removed, n11_events,
                    n12_added, n12_removed, n12_events,
                    n22_added, n22_removed, n22_events,
                    n11_added+n12_added+n22_added,
                    n11_removed+n12_removed+n22_removed,
                    n11_events+n12_events+n22_events,
                    flush=True,
                    file=file_events,
                    sep=','
                    )
        else:
            if verbose:
                print(f'frame {frameno}: recorded first bond list')
        nn11prev, nn12prev, nn22prev = nn11, nn12, nn22
        last_time = time.time()
    # (EVENTS) Finally, close the file

    if args.events.max >= 0:
        file_events.close()

    ### This part deals with writing ovito output files
    # (OVITO) First, we open the file for writing
    if args.ovito.max >= 0:
        f = open(args.ovito_file, 'w')
    # (OVITO) Then, we loop over the selected timesteps
    for frameno, ts in enumerate(read_timesteps(filename)):
        if frameno > args.ovito.max:
            break
        if not args.ovito(frameno):
            continue
        if verbose:
            print(f'frame {frameno} / filepos {ts.progress*100:.1f}%: appending
trajectory to {args.ovito_file}')
        # Again, we separate the atom types
        type1 = AtomsByType(ts, 1)
        type2 = AtomsByType(ts, 2)
        # and calculate the bond lists
        # it's double work maybe, but we are only interested in
        # performing this step for a small amount of timeframes so
        # the performance impact is neglible
        nn11 = FindNeighbours(type1, type1, r11)
        nn12 = FindNeighbours(type1, type2, r12)
        nn22 = FindNeighbours(type2, type2, r22)
        # start writing out the LAMMPS Trajectory file
        # we record all timesteps in the same file as ovito is then able
```

```
        # to combine it with the orignal dump so you can use the time controls
        # to step through
        # we start with writing some headers
        print('ITEM: TIMESTEP', file=f)
        print(ts.at, file=f)
        print('ITEM: NUMBER OF ENTRIES', file=f)
        print(len(nn11.bonds) + len(nn12.bonds) + len(nn22.bonds), file=f)
        print('ITEM: BOX BOUNDS pp pp pp', file=f)
        print(f'{ts.bounds[0,0]:.6f} {ts.bounds[0,1]:.6f}', file=f)
        print(f'{ts.bounds[1,0]:.6f} {ts.bounds[1,1]:.6f}', file=f)
        print(f'{ts.bounds[2,0]:.6f} {ts.bounds[2,1]:.6f}', file=f)
        # and then finally the bonds list:
        print('ITEM: ENTRIES index id1 id2', file=f)
        i = 1
        for a, b in nn11.bonds:
            print(f'{i} {a} {b}', file=f)
            i = i + 1
        for a, b in nn12.bonds:
            print(f'{i} {a} {b}', file=f)
            i = i + 1
        for a, b in nn22.bonds:
            print(f'{i} {a} {b}', file=f)
            i = i + 1
    # (OVITO) and at last we close the file
    if args.ovito.max >= 0:
        f.close()

    # That's it. Print total runtime:
    main_end_time = time.time()
    if verbose:
        print(f'finished. took {main_end_time-main_start_time:.2f}s')

##############################################################################
#                        COMMAND LINE USAGE BELOW                            #
##############################################################################

def frame_usage():
    # This is shown during --help
    return '''frame specification (for FRAMES):
    one of:
        start:stop:step
        list,of,frames
        none
        all

    for example:
        ::5         select every 5th frame
        1:10:3      select frame 1, 4 and 7
```

```
        0,100,200   select frame 0, 100 and 200
        none        do not perform this action
        all         perform action for each timeframen
        \n'''

def example_usage():
    # This is shown during --help
    return r'''some examples:

    estimate r11, rt12 and r22 from first 10 frames
    then calculate events for all frames:

        python3 script_v1.py dump.shear \
                --rdf 0:10 --events all

    same, but calculate events by skipping odd frames

        python3 script_v1.py dump.shear \
                --rdf 0:10 --events ::2 --events-file period2.csv

    same, but explicitly list r11, r12 and r22 and limit
    to first 200 timeframes

        python3 script_v1.py dump.shear \
                --r11 1.42 --r12 1.24 --r22 1.192 \
                --events :200:2 --events-file period2.csv

    output bonds list for timeframes 0, 5 and 10 for usage in ovito

        python3 script_v1.py dump.shear \
                --r11 1.42 --r12 1.24 --r22 1.192 \
                --ovito 0:15:5

        # load LAMMPS output file in ovito
        ovito dump.shear
        # in ovito, select Load Trajectory
        # then in the window select the file
        # ignore index, select id1 and id2 as
        # Particle Identifier'''.replace('script_v1.py', sys.argv[0])

def frame_range(s):
    '''Parse a frame specification

    and return a function that evaluates to true for
    the selected frames
    frame_range('::5') == lambda x: x%5 == 0
    frame_range('1,2,3') == lambda x: x in [1,2,3]
    '''
```

```
    if not s or s == 'none':
        f = lambda x: False
        f.max = -1
        return f
    elif s == 'all':
        f = lambda x: True
        f.max = float('inf')
        return f
    elif ':' in s:
        if s.count(':') == 1: s = s + ':'
        assert s.count(':') <= 2
        start, stop, step = s.split(':')
        start = 0 if not start else int(start)
        stop = float('inf') if not stop else int(stop)
        step = 1 if not step else int(step)
        f = lambda x, start=start, stop=stop, step=step: \
                start <= x < stop and (x-start) % step == 0
        f.max = stop - 1
        return f
    else:
        parts = s.split(',')
        selected = set(int(part) for part in parts)
        f = lambda x, selected=selected: \
                x in selected
        f.max = max(selected)
        return f

def test_frame_range():
    '''pytest tests for range specification'''
    range10 = list(range(10))
    frame_range('::5')
    def lfilter(f):
        return list(filter(f, range10))
    assert lfilter(frame_range('::5')) == [0, 5]
    assert lfilter(frame_range('1::3')) == [1, 4, 7]
    assert lfilter(frame_range('1,2,100')) == [1, 2]

def parse_args():
    '''This function parses the command line arguments'''
    class CustomFormatter(argparse.ArgumentDefaultsHelpFormatter,
                          argparse.RawDescriptionHelpFormatter):
        pass
    parser = argparse.ArgumentParser(
            description='Process LAMMPS output',
            formatter_class=CustomFormatter,
            epilog=frame_usage() + example_usage()
            )
    parser.add_argument('--rdf', metavar='FRAMES', type=frame_range,
```

```
            default='none', help='average rdf calculation over these frames')
    parser.add_argument('--rdf-file', metavar='FILE', type=pathlib.Path,
            default=None, help='write all rdf numerical data to this file')
    parser.add_argument('--rdf-plot', metavar='FILE', type=pathlib.Path,
            default=None, help='plot averaged rdf data to this file')
    parser.add_argument('--cutoff', metavar='DIST.', type=float,
            default=4, help='cutoff for rdf calculation')
    parser.add_argument('--nbins', metavar='NUM', type=int,
            default=200, help='number of bins for rdf calculation')
    parser.add_argument('--events', metavar='FRAMES', type=frame_range,
            default='none', help='write events for these frames')
    parser.add_argument('--ovito', metavar='FRAMES', type=frame_range,
            default='none', help='write out an ovito file for each frame')
    parser.add_argument('--events-file', metavar='FILE', type=pathlib.Path,
            default='events.csv', help='filename for events')
    parser.add_argument('--ovito-file', metavar='FILE', type=pathlib.Path,
            default='bond_topology_trajectory.txt', help='pattern for ovito
trajectory output')
    parser.add_argument('-q', '--quiet',
            action='store_true', help='don\'t log progress updates'
            )
    parser.add_argument('input-file', type=pathlib.Path,
            help='input filename'
            )
    parser.add_argument('--r11', metavar='DIST.', help='r11 cutoff distance',
type=str, default='estimate')
    parser.add_argument('--r12', metavar='DIST.', help='r12 cutoff distance',
type=str, default='estimate')
    parser.add_argument('--r22', metavar='DIST.', help='r22 cutoff distance',
type=str, default='estimate')

    args = parser.parse_args()
    return args

if __name__ == '__main__':
    main()
```