Molecular Dynamics Simulation of Methane Diffusion Through Shale Rock

Stanley Urbanek

Completed for the Certificate in Scientific Computation
University of Texas at Austin
SDS 379R

Supervised by Dr. Michael Marder

December 14, 2015

Dr. Michael Marder 12/14/2015

Abstract

A molecular dynamics simulation is designed and coded in LAMMPS for the purpose of modeling the diffusion of gaseous methane through nanoscopic pores in shale rock, as part of a larger migration to the surface of the Earth. Thermodynamic and flow profile quantities of a shale gas are tracked over the course of simulation. Comparison of theoretical and simulated volumetric flow rates is made. This study is of direct relevance to hydrofracturing, which has come to the foreground of the United States recently as a source of energy, as a commodity, and as an environmental concern.

Contents

1 Introduction

This paper serves as the final research project for the Certificate of Scientific Computation, at the University of Texas at Austin (UT). It was begun as SDS 379R in January 2015, and submitted December 2015. The purpose of this paper, is to detail how this project was set up and currently runs, with enough detail so as to be reproducible.

1.1 Background

The landscape for natural gas production is drastically changing. In 2000, shale gas was a negligible portion of all natural gas production in the United States. By 2035, the Energy Information Administration (EIA) predicts that it will comprise half of all US natural gas production. The US has gone from being one of the largest importers of gas, to being self-sufficient, in less than a decade.[8] Shale gas is clearly of current relevance to the US energy sector and economy. Accompanying its rise in popularity, is uncertainty in the physical details behind its extraction. A current topic of research, is the paths by which methane takes as it escapes the shale, from small scale diffusion to large scale migration.

Shale is a common sedimentary rock, characterized by small particles and organic carbon material, and is finely layered. This makes shale compact and impermeable. Gas (largely methane) is formed by the cooking of this organic material, and is trapped by the shale. Shale is so impermeable, that gas creeps along only a few millimeters every few thousand years.[8] Fractures, made both naturally and through hydrofracturing, make the shale permeable; through these fractures, methane can diffuse. These fractures (hereby, fractures/pores of diameter 1 - 100s nm are called nanopores) are typically determined by the diameter of the sand grains - fractions of a millimeter - that are used to prop it open.

1.2 Goal and Model

Methane is trapped under shale rock, within a given range of temperatures and pressures. Hydrofracturing creates nanopores through which it can diffuse. The goal of this project, is to design a molecular dynamics simulation of methane diffusing through nanopores of varying diameters. The aim is to

have a system of flexible code, allowing for various conditions and restraints to be placed on the diffusive dynamics.

Methane is created in a molecular dynamics simulator. This project uses LAMMPS as its classical molecular dynamics code. LAMMPS was chosen because it is distributed as open source code under the GPL, by a reputable source - Sandia National Laboratories, and it is well documented.[1,2] The pores are modeled by carbon nanotubes (CNT). CNTs were chosen because they are a stable body of the same order diameter as these pores, can have their diameters adjusted, and have similar atomic composition as shale rock, which contains organic carbon material. Methane is initialized within the carbon nanotube, thermostatted and pressurized, and set to evolve with time.

Time permitting, the dynamics of methane diffusion will be studied. Namely, the velocity profile of the methane will be monitored on its way to equilibrium, as a function of density, pore size, distance from the center of the tube, and so on. If larger systems are created, beyond the computational scope of dual-processor laptops, the simulations can be run on the Texas Advanced Computing Center (TACC) computer clusters at UT.

2 Theory

The Knudsen number, as it relates to the modeling regimes, and Poiseuille's Law are examined.

2.1 Knudsen number

In general, simulation of flow through nanopores employs either a continuum or molecular approach. The molecular approach determines individual particle dynamics, based on Boltzmann distributions, while the continuum approach determines larger scale fluid dynamics. The Knudsen number is an indication of a method's applicability to a given system. The dimensionless parameter is defined as

$$Kn = \frac{\lambda}{\Lambda} \tag{1}$$

where λ is the mean free path, and Λ the macroscopic length scale of a physical system. λ is calculated with

$$\lambda = \frac{16\mu}{5\rho\sqrt{2\pi RT}} \tag{2}$$

where μ is the the coefficient of dynamic viscosity, ρ is the density, R is the specific gas constant, and T the temperature. As Kn increases beyond 10^{-3}, the validity of the standard continuum approach diminishes. No-slip conditions diminish as Kn moves beyond 10^{-1}. Above $Kn = 10^{1}$ lies the free molecular modeling regime.[11]

2.2 Poiseuille's Law

Poisseuille's Law is applicable in the continuum mechanics regime, under assumptions of laminar flow through a long, symmetric cylinder. It determines the volumetric flow rate Φ.

$$\Phi = \frac{dV}{dt} = v\pi r^2 = \frac{\pi r^4 \Delta P}{8\nu L} \tag{3}$$

Here, V is the volume, v is the mean velocity along the tube, r is the radius of the tube, L is its length, ΔP is the pressure difference across the tube, and ν is the dynamic fluid viscosity.[14] From these, the volumetric flow rate can be determined both theoretically and experimentally.

3 LAMMPS

3.1 The Molecular Dynamics Simulator

LAMMPS, an acronym for Large-scale Atomic/Molecular Massively Parallel Simulator, is a molecular dynamics simulator. LAMMPS operates by integrating Newton′s equations of motion for atomic and molecular systems, calculating a variety of atomic potentials. Its code has a wide range of potentials, allowing for the modeling from individual atoms, up to large molecular systems, such as solid-state materials, soft matter, and coarse-grained or mesoscopic systems. LAMMPS can be run on any single-processor laptop, or parallel computer, that compiles C++ and supports the MPI message-passing library. LAMMPS is free to download and modify as desired. It is possible to modify and extend its capabilities, by creating new potentials, force fields, and atom types. LAMMPS is robustly documented, and a maintained mailing user list exists.[1]

LAMMPS calculation time can become quicker and more efficient, by being run on the Texas Advanced Computing Center (TACC) computer clusters. TACC's main operating system, which resides at UT, is linux. Preemptive familiarity with Linux was advised, so the main system computing this project is a dual processor, Dell Inspiron laptop, running Linux Mint 17.1.

For the visualization tool, MPlayer was chosen to compress and process the LAMMPS dumped images. MPlayer, an open source media player, is installed through apt-get, via the online Linux repository.

3.2 Compiling LAMMPS (with Optional Packages)

While LAMMPS has a core set of files that allow it to run as is, there are some optional packages that can be compiled in for greater simulation depth. These optional packages contain, for example, additional molecular interatomic potentials. Two packages used in this project, that must be separately compiled into LAMMPS, are the MEAM and ReaxFF potentials. All compilations are done within the /src directory. The list of installed optional packages are listed with the command

> make package-status

within the /src directory. If the desired package is not part of the executable, it is included in the compilation with

> make yes-meam.

LAMMPS is compiled with

> make serial.

The executable is made accessible in the terminal, within all directories, by

> sudo cp lmp_serial /usr/local/bin

which copies the LAMMPS executable, lmp_serial, to the /usr/local/bin directory. To remove undesired packages, the command

>make no-meam

is used, and LAMMPS is then recompiled.

3.3 The Potentials

The LAMMPS code includes optional potentials packages. Two packages that are used in this project, are the reactive force field (ReaxFF) and modified embedded-atom method (MEAM) potentials.

3.3.1 ReaxFF

ReaxFF is designed for molecular dynamics simulations of large scale reactive chemical systems.[5] Because it was designed and coded specifically for hydrocarbons (including methane), the ReaxFF potential is used as the the potential for the molecular simulation of methane. ReaxFF determines the potential energy from chemical bonding by distance-dependent bond-order functions.[1] It also includes Coulomb and van der Waals potentials for non-bonded atomic interactions. ReaxFF breaks down the system energy into components, via

$$E_{system} = E_{bond}+E_{over}+E_{under}+E_{val}+E_{pen}+E_{tors}+E_{conj}+E_{vdWaals}+E_{coulomb} \tag{4}$$

Constants in use are experimentally determined.[5,6]

3.3.2 MEAM

The MEAM potential is an extension of the embedded-atom method, which is based on density-functional theory, to include forces that have angular dependence. MEAM parameters are specific to the energy and lattice constants of graphite. Because a CNT is a curled graphene sheet, MEAM is used as the potential for the CNT. The formulation of the total energy is governed by

$$E = \sum_i \{F_i(\rho_i) + \frac{1}{2}\sum_{i \neq j} \Phi_{ij}(r_{ij})\} \tag{5}$$

where F is the embedding energy of an atom into a given background electron density, and Φ is a pair interaction between 2 atoms.[7]

3.4 Input Script Format

LAMMPS is fed an input script, conventionally named 'in.scriptName'. LAMMPS follows somewhat typical syntax and parsing conventions, such as commenting lines out (using a preceding #) and continuing a command onto the next line (using a succeeding &), for example. Variables can be defined and used within. Commands begin with any of a given keyword, and are followed by any number arguments and specifiers. An input script is broken down into four main sections: Initialization, atom definition, settings, and running a simulation.

The initialization section defines parameters to be used by the system. Many of these parameters necessarily need to be determined before the molecules are created, or read-in from a file. Some of the necessary parameters are units used, dimension and boundaries of the system, how the processors will be mapped, and what attributes will be associated with the atoms.

The atom definition section introduces atoms into the simulation system. There are 2 main ways to do so. Firstly, atomic attributes (mass, charge, etc.) and initial positions can be read in via a data file. This has the benefit of being explicitly clear in the atomic initialization, but can be tedious for large systems. An additional program in another language (like python) may be necessary, to facilitate the writing of the data file. Secondly, atoms can be introduced via a series of built-in commands, by defining a lattice to place the atoms or molecules on, within a given region. This has the benefit of elegance, written in much fewer lines than the former method, but needs greater syntactic precision.

The settings section determines details about the simulation, such as force field coefficients, simulation parameters, and output options. Pairwise, bond, and angle coefficients can be specified. Details of the atomic neighbors list can be modified, along with the timesteps and time integrator. Any of a wide range of operations, called fixes, can be applied to the system during timestepping, such as thermostatting, updating atomic positions/velocities, or boundary impositions. Additionally, the types of computations and thermodynamic outputs can be determined. This section can be repeated, or overwritten, as many times as desired.

The final section runs the simulation that has been created. This can also be repeated as many times as desired, with modifications as needed.

4 Carbon Nanotube

A carbon nanotube is a 3-dimensional (3D) object, comprised entirely of carbon atoms. The carbon atoms are located at the points of a 2-dimensional (2D) honeycomb lattice, that has been rolled into 3D. This section details the 2D honeycomb lattice, and how it is rolled into 3D. It explains the python script written to automate the determination of these carbon locations. The section ends by walking through the coding of the CNT in LAMMPS.

4.1 Honeycomb Lattice

A lattice is a configuration of points, where a given pattern is repeated indefinitely. For some lattices, it is possible to rotate or translate the lattice, such that a new version is indistinguishable from the original. Two examples of lattices, are Bravais lattices, and lattices with bases.

Bravais lattices are the simplest type of lattice. In a Bravais lattice, a point's list of neighbor points, and relative orientations, are identical to all the others in the lattice. These points are determined by linear combinations of primitive vectors. In 2D, these points are determined by

$$\vec{R} = n_1\vec{a}_1 + n_2\vec{a}_2 \tag{6}$$

where n_i are integers, and the primitive vectors are linearly independent. Primitive vectors need not be unique. One Bravais lattice, of relevance to the honeycomb lattice, is the triangular lattice. Its symmetries lie with x- and y-plane reflections, and 60 degree rotations.[3]

A second type of lattice, of higher complexity but more commonly found in nature, is lattices with bases. These lattices are structured over Bravais lattices, and each point has been replaced with a collection of points. The replacement vectors need not be unique, nor linearly independent.

The honeycomb lattice is a lattice with bases. It is founded on a triangular Bravais lattice, using the primitive vectors

$$V_1 = (1, 0) \tag{7}$$

$$V_2 = \frac{1}{2}(1, \sqrt{3}) \tag{8}$$

Each point is replaced by 2, using the linearly dependent vectors:

$$V_3 = (0, \frac{1}{2\sqrt{3}}) \tag{9}$$

$$V_4 = (0, -\frac{1}{2\sqrt{3}}) \tag{10}$$

Because these vectors are normalized to 1, they can all be multiplied by a constant scale factor to make them consistent with atomic physics scales, all while preserving their original lattice features.[3]

4.2 Mapping 2 Dimensions to 3

A CNT is a 2D honeycomb lattice rolled into 3. To determine the CNT atomic locations, this project orients the 2D honeycomb lattice to lie in the xy plane. When rolled, the x axis becomes the axis of symmetry of the nanotube, and the y axis is curled so that it lies in the yz plane. This means that whatever length and width the honeycomb lattice is given, translates to the length and radius of the nanotube. Specifically, the length X determines the length of the tube, and Y determines the radius.

The mapping used is

$$(x_i, y_i) \rightarrow (x, r * sin(\frac{y_i}{r}), r * (1 - cos(\frac{y_i}{r}))) \tag{11}$$

where $r = \frac{Y}{2\pi}$, and all subscripted values are individual 2D atomic locations.

4.3 Python Script

The foundational idea of the python script (Appendix A.1 - vectors.py) is to be a program that, given any 2 basis vectors, propagates forth those vectors within given bounds. It rolls the atomic locations into 3D, using mapping 8, and prints the atomic locations to a text file. This text file is used by LAMMPS to construct the carbon nanotube. Because the script uses any arbitrarily scaled basis vectors, it is easy to adjust the length and width of the carbon nanotube.

The script is broken into different functions that sequentially complete each step of creation for a 3D CNT. The main function passes the 2D primitive vectors, honeycomb bounds, vector scaling factor, and visualization details to a function (function honeycombLattice) that determines the lattice locations for a triangular lattice for the given information, and replaces them with the replacement points, in function getHoneycombYCoordinates. Then the function conv2to3dim converts these 2D points into 3D.

As a preliminary visual confirmation that the program is working properly, python′s turtle graphics were used. Turtle graphics is a library that is included in the standard python installation. There is an additional function, rotate3dim, that rotates the final 3D CNT atomic locations. The rotation matrix

$$R = \begin{pmatrix} sin(\theta)cos(\phi) & sin(\theta)sin(\phi) & 0 \\ -sin(\theta)sin(\phi) & sin(\theta)cos(\phi) & 0 \\ 0 & 0 & cos(\theta) \end{pmatrix} \tag{12}$$

yields the new atomic rotated locations, which are then displayed in the turtle graphics window.

4.4 Creating the CNT in LAMMPS

The CNT is modeled in 3 dimensions, so the command used is

> dimension 3.

Units must also be specified. All unit choices use the physical constants from the National Institute of Standards and Technology[4], except for lj style. The choice of units simply sets internal conversion factors within LAMMPS. This means that any simulation will be exactly replicated, including particle trajectories and thermodynamic outputs, for any choice.[1] Doing so requires all input parameters to be converted. Because of this simplicity to convert to any units, the unitless choice was made

> units lj

where all constants are set to 1. This allows greater flexibility in LAMMPS for later stages of simulation. Hereby, when units are mentioned, like nanometers for CNT locations, it means conversion pending.

To set the boundary style for the simulation box, the boundary command is used to specify each direction′s style

> boundary p s s

The first argument, p, means that, in the x direction (along axis of CNT), the boundaries are periodic. Particles are allowed to interact across the boundary, and they are free to exit one end of the box, being reintroduced on the other end. This was chosen because, as the CNT lengthens, the computational time dramatically increases. By shortening the box and treating it this way, computation time is held reasonable, but the steady state flow of methane through it still possible. The second and third arguments, s and s, mean that in the y and z directions respectively, the boundary is non-periodic and shrink wrapped. Atoms will be traced no matter how far they stray from their original positions. This was chosen for 2 reasons. Firstly, the CNT should not break apart, and secondly the methane should be completely contained within the CNT.

Next, LAMMPS must be told which attributes to assign to the atoms. By default, LAMMPS stores data for coordinates, velocities, atom IDs and types.[1] The choice of atom style says what additional information is tracked. For example, point-like uncharged molecules require fewer characteristics than charged ones, and than rigid bodies. For this project, the charge style was chosen

> atom_style charge.

This was chosen preemptively, as methane has a small dipole moment.[7]

Now that the system has been initialized as desired, it is time to read in the atoms. The atomic locations for the CNT, of arbitrary length and width, are given by the python script. This can be put in proper LAMMPS data file format, and read in via

> read_data in.meamCNT

where in.meamCNT is, by LAMMPS convention, the name of the data file. (The first couple of lines in.meamCNT is in Appendix B) The data file has a header and a body section, but is rather flexible in its order. The header contains a description of the file, and is ignored by LAMMPS. The body has zero or more sections, where each section either begins with a keyword, and is followed by arguments specific to that keyword on the following lines, or has an inline command with the keyword succeeding the parameters. The first 2 lines establish how many atoms, and how many different types, are being read in, via

>> 3500 atoms

>> 1 atom types

Next, the simulation box boundaries are specified. They are established in each direction by

>> -171 171 xlo xhi

>> -50 50 ylo yhi

>> -50 50 zlo zhi

with the boundaries made to envelop the whole CNT. Next, to begin the designation of carbon atoms, the atomic mass is assigned by

>> Masses

>> 1 12.0110

This section begins with the keyword 'Masses', and says that all type 1 atoms have mass 12.0110 (amu). This can be extended to include arbitrarily many atom types. Finally, the carbon locations are read in from the python script, with

>> Atoms

>> 1 1 0.0 0.0 0.70916978246 0.0297230203843

>> 2 1 0.0 2.45951214675 0.70916978246 0.0297230203843

>> . . .

The keyword 'Atoms' begins the placement. The syntax for each line, is (atom id, atom type, charge, Xlocation, Ylocation, Zlocation). An atom_type of 1 will assign all atom_type quantities to this atom; here, it will assign a mass of 12.0110 to these carbon atoms. All charge values are set to zero, because the coefficient files hold the proper values.

To facilitate manipulation of the CNT, it is desirable to assign all the constituent carbon atoms to a group. This is done with the command

> group CNT type 1

where group initializes the designation, CNT is the group ID, and all type 1 atoms (here, C) are added to it. This allows the group ID to be used in conjunction with any later fixes or computations, instead of having to refer to all 3500 atoms individually.

Next comes the designation of potentials. To designate the pairwise potential meam,

> pair_style meam

is used. This simply sets the formulas used by LAMMPS to compute the pairwise interactions. To specify the pairwise coefficients,

> pair_coeff * * library.meam C NULL C

is used. The first 2 arguments specify which pairs the coefficients are describing. '* *' signifies that they are for all combinations of atom types. 'library2.meam' specifies the potential file that the coefficients are read from. This is included with the meam package. The first 'C' specifies which entries to pull from 'library2.meam', those for carbon, 'NULL' is a placeholder for an additional potential file (that is not used), and the final argument designates the coefficients to atom type 1. If 2 C's existed after 'NULL', those coefficients would be assigned to atom types 1 and 2.

LAMMPS tracks the movement and forces on atoms by using a neighbor list. The force cutoff, beyond where the force between 2 atoms is ignored, determines which atoms are to be included in a neighbor list. With so many atoms in a tight lattice area, it is of interest to extend the neighbor list, to include further reaching atoms. This is done with the command

> neighbor 0.3 bin

> neigh_modify delay 10

where neighbor specifies that this change will be made, 0.3 distance units are added to the force cutoff, and bin style selects which algorithm is used to

build the neighbor list (bin is done by binning). Because of the importance of neighbor lists to LAMMPS, the neighbor calculations can be further modified with the neigh_modify command, where the 'delay' option determines how frequently neighbor lists are rebuilt. Here, 'delay 10' signifies that new neighbor lists are not calculated until at least 10 timesteps have passed since the previous build. These values are based off the LAMMPS recommended values for similar systems.

A fix must be applied to the system, such that meaningful dynamics occur. The following fix is applied

>fix 1 all nve

This fix, called 1, recalculates the position and velocity for all atoms, through constant nve integration. It determines the atomic paths, consistent with the microcanonical ensemble, by isolating the system from changes in the number of atoms (n), the volume (v), and the energy (e). This ensures that meaningful dynamics happen to the atoms of the CNT. It is advised in the documentation, however, that if a stationary container is sought, such that there is atomic presence but no movement, it is best to remove this fix.[1]

To help extract meaningful information from the simulation, thermodynamic information can be printed to the screen at given time intervals. The following command is used

> thermo 10

to print out thermodynamic information, such as energy, temperature, and pressure, every 10 timesteps. The timestep must also be specified, using the command

> timestep 1e-3

where the timestep is set as dt = 1e-3 time units. This was taken as an example from others.

To facilitate the determination of whether the simulation is running as expected, image snapshots of the system are taken. The dump command dumps PNG, JPG, or PPM image files of the atomic configurations every given time steps. This single command

>dump 3 all movie 5 RefiningCNT.mpg element element &

> axes yes 0.8 0.02 view 93 -180 zoom 11 adiam 1.5 size 1028 1028

is continued onto the second line with &. The first argument '3' specifies the dump number, 'all' specifies that all atomic images will be dumped, and 'movie' means that the dumped images are combined and compressed into a movie file. The name of the movie is 'RefiningCNT.mpg'. Some of the following parameters ('element') are placeholders. Additional visual

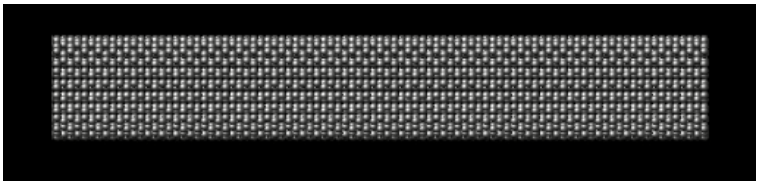

Figure 1: A lengthwise view of a CNT simulated in LAMMPS.

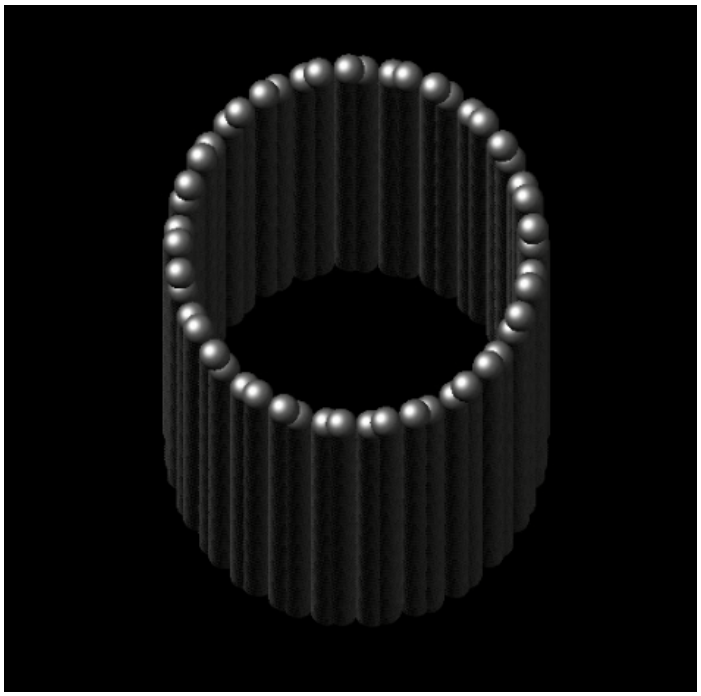

Figure 2: An angled view of a CNT simulated in LAMMPS.

parameters are 'view 93 -180', which rotates the simulation box by 93 degrees and -180 degrees in orthogonal directions, 'zoom 11', which zooms in on the system by 11 times, 'adiam 1.5', which sets the atomic diameters to 1.5 units (for visual purposes only), and 'size 1028 1028', which sets the overall number of pixels stored.

The final step is to run the simulation, using the command

> run 5000

This makes the system run for 5000 timesteps. Multiple run statements can be used in the same input script.

5 Methane

Methane is the gaseous hydrocarbon that is to populate the inside of the CNT. This section details the determination of the atomic locations of methane, by use of object-oriented programming in python, and its creation in LAMMPS.

5.1 Python Script

In the spirit of the python script used to write the CNT atomic locations, a script was similarly written to determine the atomic locations of methane. Object-oriented programming was implemented (mainly as a demonstration of the author's ability), and can be found in Appendix A.2 (methaneClass.py).

The main function invokes an instance of the class Methane. The constructor of Methane creates class variables for the 3D coordinates of the origin of the CNT, and a current iterator location. The main function runs through a given number of points spaced throughout a rectangular approximation of the CNT. Each point is checked for validity (its proximity to the CNT if outside the CNT, or too close to the wall, the point in contention is invalid) by the method self.isValidMethaneLocation(arg). If the point is valid, the current iterator location is updated by method self.updateCurrent(arg), and the carbon and hydrogen atomic locations are calculated around the point, using the method self.retunsMethaneAtomicLocations(). The atomic locations are then written to a text file, which is used by LAMMPS.

5.2 Creating Methane in LAMMPS

For the same reasons of flexibility, the dimensionless lj units are chosen. Per section 3.4, the atom_style charge is chosen. As methane is gaseous, and not self-contained in any direction, the boundaries are chosen to be periodic in all 3 directions. To read in the methane atoms, the following command is used

> read_data data.makeItMethane

where data.makeItMethane is the equivalent of data.meamCNT. It similarly reads in the number of atoms (1100), the types (3, and 1 is redundant),

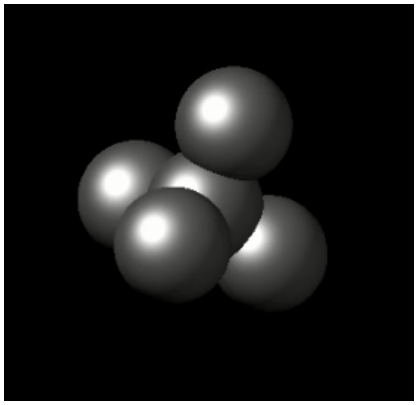

Figure 3: A single methane atom modeled in LAMMPS.

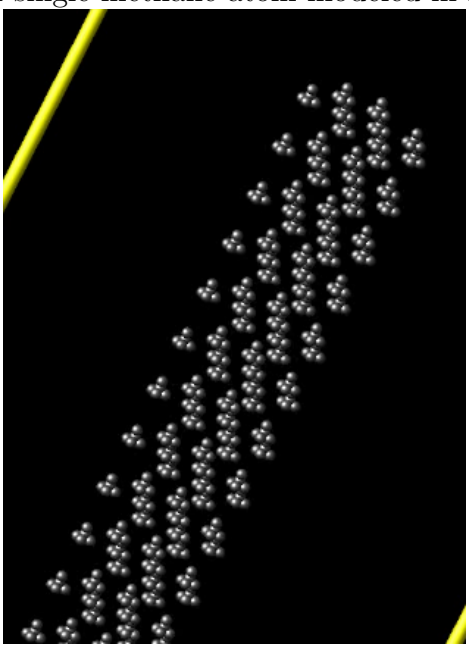

Figure 4: A regular arrangement of methane atoms, as how it might be shaped by a CNT.

the simulation boundaries, and the masses.

The reax potential, and pairwise coefficients, are specified through the following 2 commands:

> pair_style reax/c lmp_control

> pair_coeff * * ffield.reax.cho C H C

where pair_style initializes the reax/c potential, and the /c marks the difference between the older, Fortran based Reax, and the newer stand-alone C code version. The control file lmp_control defines package-included values of control variables. The pair_coeff command determines the coefficients between all atoms as specified in ffield.reax.cho, which is also included in this package. It respectively assigns the carbon, hydrogen, and carbon coefficients to atom types 1, 2, and 3. The neighbor and neigh_modify commands are similar to those used for the CNT.

There are 3 fixes associated with the reax potential, and they are as follows:

> fix 1 all nve

> fix 2 all qeq/reax 1 0.0 10.0 1e-6 param.qeq

> fix 3 all temp/berendsen 340.0 340.0 20.0

The first 'nve' fix is the constant nve integration. Fix 2, 'qeq/reax', performs charge equilibration for the system. It minimizes the electrostatic energy of methane by adjusting the atom's partial charge according to its interactions with neighbors.[1] The following numbers, and param.qeq, are all coefficient values. The 'temp/berendsen' fix is a Berendsen thermostat, keeping the low and high temperature at 340.0 temperature units , with '20' time units specifying the temperature relaxation rate.

The timestep, thermo, dump, and run commands are similar to those used for the CNT.

6 Driving methane through CNTs

Now that the CNT and methane are separately scripted, it is necessary to combine them. This requires the splicing of (hopefully) non-mutually-exclusive LAMMPS commands. The methane must be made to populate entirely within the confines of the CNT, and not be allowed to escape laterally. Pressure must be added to initiate the flow of methane.

There was some foresight during the separate creation of the 2 scripts. Hence, units lj and atom_style charge are already in agreement. The data files from both are combined into one, and all the atomic locations are specified (Appendix B.1 data.realDataCombo). To facilitate operating directly on the separate types, 2 groups are defined:

> group CNT type 1

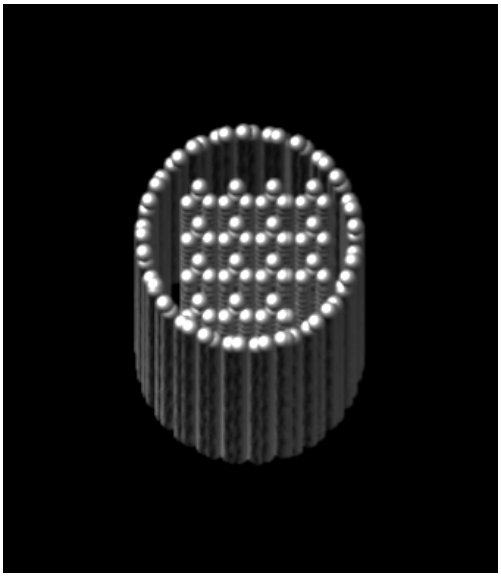

Figure 5: The initial population of the CNT with methane.

> group methane type 2 3

where all type 1 atoms belong to the CNT, and type 2 and 3 belong to methane.

Multiple interatomic potentials can be specified in one script. However, there is an issue with the combination with the reax and potentials. When run, LAMMPS returns, 'ERROR: Incorrect args for pair coefficients', but thef commands are entered syntactically correct. According to a thread in the LAMMPS mailing list, it is inconsistent to mix and match certain potentials, such as Reax and MEAM. Rather, one such potential should be used for the entire system.[10] So, because ReaxFF is designed for hydrocarbon systems, including methane and graphene, it is chosen to represent both systems, with the modification to the type specification argument

> pair_coeff * * ffield.reax.cho C H C .

Now, atom types 1, 2, and 3 are C, H, and C respectively.

To introduce pressure to the system, the command to add force is used:

> fix 4 methane addforce 10.0 0.0 0.0

where a force is applied to methane only, in the x direction with 10.0 units of strength, and none in the y and z directions. This directs the force down the axis of symmetry of the CNT. To calculate the average particle

velocity along the CNT, the following command is used:

> fix 5 methane ave/spatial 10 5 50 x 0 171 vx density/mass file ve2.profile

This computes the average velocity in the x direction and the mass density of the methane group every 50 timesteps. It saves this data to file 've2.profile'.

This script is 'in.CNTandMethane', in Appendix A.5.

7 Data and Analysis

The simulation was run for 2000 time steps. The thermodynamic outputs and spatial-averaged flow profile were recorded with varying frequencies.

7.1 Thermodynamic Outputs

```
# Data log of methane passing through CNT
      Step        Temp      E_pair           E_mol      TotEng        Press
       0             0   -149.83496             0   -149.83496   0.48257086
      10   0.16329194   -149.90716             0   -149.66227   0.48260786
      20   0.55130593   -150.02227             0   -149.19548   0.48304069
      30    1.0971671   -150.10233             0    -148.4569   0.48340489
      40    1.7183544   -150.05087             0   -147.47384   0.48359935
      50    2.3590219    -149.8178             0   -146.27996   0.48332848
      . . . . .
    1950    2269.6587   -125.32022             0    3278.5067    7.3076626
    1960    2296.4061   -124.99352             0    3318.9468    7.3882241
    1970    2323.3867   -124.81452             0    3359.5888    7.4665786
    1980    2350.7425   -124.91962             0    3400.5094    7.5514794
    1990     2378.464    -125.1077             0    3441.8955    7.6358442
    2000    2406.1382   -125.07858             0     3483.428    7.7155241
```

The temperature, pair wise energy, total energy, and pressure are tracked in 10 timestep increments. All values recorded are in lj units.

7.2 Flow profile

```
# Spatial-averaged data for fix 5 and group methane
# Timestep Number-of-bins
```

```
# Bin Coord Ncount vx density/mass
50 1
  1 85.5 1650 2.5661 0.00309596
100 1
  1 85.5 1650 5.11447 0.00309596
150 1
  1 85.5 1650 6.57532 0.00309596
200 1
  1 85.5 1650 8.40586 0.00309596
250 1
  1 85.5 1650 10.0452 0.00309596
. . . . .
1800 1
  1 85.5 1650 113.199 0.00309596
1850 1
  1 85.5 1650 117.192 0.00309596
1900 1
  1 85.5 1650 121.181 0.00309596
1950 1
  1 85.5 1650 125.162 0.00309596
2000 1
  1 85.5 1650 129.164 0.00309596
```

The time steps are in 50 step increments. The average velocity in the x direction, vx, is along the flow axis of the nanopore. The mass density and number of atoms $Ncount$ stay constant, as expected. All values are in lj units.

7.3 Constants

The following constants and values are used for all calculations:

r	0.85 nm
ν^{12}	11.2 μPa s
Λ	17.2 nm
T	300 K
R	8.314 $\frac{J}{molK}$
σ^{12}	0.36652 nm

7.4 Analysis

Looking at the thermodynamic data, and the video dumped by the simulation, the system appears to be working as designed. As expected from the addition of an external force, the pairwise energy, total energy, and pressure all increase. The only odd performance is by temperature, which increases drastically over time, when it is expected to stay constant due to the Berendsen thermostat. This may be attributed to the relaxation time given to the thermostat fix ($T_{time\ damp} = 100.0\ units$). Indeed $T_{temperature} = 300K$ translates to $T_{temperature} = 52200\ units$, and the temperature has not yet equilibrated.

Physically, nanopores in shale rock range from 2-50 nm, although they are typically under 10nm. Shale gas is typically in the range T = 300-360K and P = 5-28 MPa, with a molecular free path between 0.3-2.2 nm.[13] This puts a typical $Kn = 10^{-2}$ to 10^{0}. Theoretical calculations for this CNT system, using Eqns (1) and (2), yield $Kn = 0.75$. This means that the continuum assumption may break down in this regime, for both typical systems and this CNT.

The theoretical prediction of volumetric flow rate for this system, using Eq. 3, yields $\Phi = 2.6 * 10^{-25} \frac{m^3}{s}$. Using the average velocity along the tube, the volumetric flow rate is $\Phi = 1.4 * 10^{-27} \frac{m^3}{s}$. These 2 orders of magnitude may point at deviations from the continuum regime. However, it is more likely that it represents non-equilibrium flow.

8 Conclusion

A flexible code is written, simulating the gaseous diffusion of methane through shale rock. The methane passes through the CNT, and returns across the periodic boundary successfully. Data is collected on the thermodynamic and flow profile. A comparison of volumetric flow rates from theoretical calculations and experimental determination is made.

For future work, the compatibility of Reax and MEAM as hybrid potentials in a single script should be investigated further. The long term behaviour of both the CNT and methane should be studied, to see how their respective potentials perform. The flow profiles of additional systems should be studied. Simulations should be run on TACC, such that longer run time may be achieved efficiently.

References

[1] "LAMMPS Molecular Dynamics Simulator." LAMMPS Molecular Dynamics Simulator. Sandia National Laboratories. Web. 2 Dec. 2015. <http://lammps.sandia.gov/>

[2] "The GNU Operating System and the Free Software Movement." The GNU Operating System and the Free Software Movement. Free Software Foundation. Web. 2 Dec. 2015. <http://www.gnu.org/>

[3] M. P. Marder, Condensed Matter Physics, (John Wiley & Sons, new York, 2000)

[4] "NIST Physical Measurement Laboratory Homepage." NIST Physical Measurement Laboratory Homepage. Web. 2 Dec. 2015. <www.physics.nist.gov>

[5] K. Chenoweth, A.C.T. van Duin and W.A. Goddard, "A ReaxFF Reactive Force Field for Molecular Dynamics Simulations of Hydrocarbon Oxidation," J. Phys. Chem. A 112, 1040-1053, (2008).

[6] Duin, Adri C. T. Van, Siddharth Dasgupta, Francois Lorant, and William A. Goddard. "ReaxFF: A Reactive Force Field for Hydrocarbons." J. Phys. Chem. A The Journal of Physical Chemistry A: 9396-409.

[7] Baskes, M. I. "Modified Embedded-atom Potentials for Cubic Materials and Impurities." Phys. Rev. B Physical Review B: 2727-742.

[8] Stephenson, Michael H. "Shale Gas and Fracking: The Science behind the Controversy. Elsevier, 2015.

[9] Oyanagi, Chikako, and Kiyoshi Yagi. "Highly Accurate Potential-energy and Dipole Moment Surfaces for Vibrational State Calculations of Methane." The Journal of Chemical Physics, 124 (2006).

[10] Kohlmeyer, Axel. [lammps-users] mailing list. <http://sourceforge.net/p/lammps/mailman/message/34596731/>; <http://lammps.sandia.gov/threads/msg54226.html>;

[11] S. Roy and R. Raju. "Modeling gas flow through microchannels and nanopores." The Journal of Applied Physics. Vol 93, Num 8. 15 April 2003.

[12] D. Friend, *et al*. "Thermodynamic Properties of Methane." J. Phys. Chem. Ref. Data. Vol 18, No 2. 1989.

[13] X. Zhang. "Lattice Boltzmann Simulation of shale gas transport in organic nano-pores." Scientific Reports. 4, 4843. 2 May 2014.

[14] Kirby, B.J. (2010). "Micro- and Nanoscale Fluid Mechanics: Transport in Microfluidic Devices." Cambridge University Press. ISBN 978-0-521-11903-0.

A Code

A.1 Python - vectors.py

```
#########################################################################
# File:  vectors.py
#
# Decription: Creates lattice using linear combination of 2 basis
#          vectors, and returns 2D and 3D atomic locations
#
# Author: Stanley Urbanek
#
# Created: 4/20/15
#
# Last updated: 5/28/15
#########################################################################

#################
## IMPORT LIST ##
#################

import math
from math import sin
from math import cos
from math import pi
import turtle
from numpy import *

#####################################################################
## FUNCTION: Draws a square boundary
##
##   [args]: ttl   - turtle object;
##        (xi, yi) - initial x,y coords of ttl;
##        (dx, dy) - boundary side lengths
##
#####################################################################
```

```
def drawBound(ttl, xi, yi, dx, dy):
ttl.penup()
ttl.goto(xi, yi)
ttl.pendown()
ttl.goto(xi + dx, yi)
ttl.goto(xi + dx, yi + dy)
ttl.goto(xi, yi + dy)
ttl.goto(xi, yi)
ttl.penup()

##########################################################################
## FUNCTION: Creates triangular lattice
##        Draws all possible combinations of atomic locations within bound
##        box, by linearly combining two, 2-D vectors
##
##   [args]: ttl     - turtle object;
##        (xi, yi)   - initial x,y coords of ttl;
##        (Xb, Yb  ) - x,y bounds of simulation box;
##        (v1x, v1y) - x,y components of vector 1;
##        (v2x, v2y) - x,y components of vector 2;
##        doesRotate - boolean, for "There is a rotation";
##        theta, phi - rotation angles about Y, Z axes, respectively
##        scale   - scaling factor for problem; facilitates visualizations only
##########################################################################

def triangularLattice( ttl, xi, yi, Xb, Yb, v1x, v1y, v2x, v2y, doesRotate, theta,

# Initialization
sgn = 1
nx = 0
ny = 0
P0x = xi #Should this be set to 0?
P0y = yi
pnewx = xi
pnewy = yi

# Within boundary conditions
```

```
while ny * v2y < Yb:
     while nx * v1x < Xb:

          ########## 2 Dim ####################

          # Get new point, (pnewx, pnewy)
          pnewx = P0x + nx * v1x
          pnewy = P0y + nx * v1y

          # DRAW 2 dim point
          ttl.goto(pnewx, pnewy)
          ttl.pendown()
          ttl.dot(1)
          ttl.penup()

          # FUTURE FIX
          print( str(pnewx) + ' ' + str(pnewy))

          ############ 3 Dim ####################

          # Calculate 3 dim point
          xf, yf, zf = conv2to3dim(pnewx, pnewy, Xb, Yb, doesRotate, theta, phi, sc

          # DRAW 3 dim point
          # CHANGE: if you want ONLY the nth (1st, 3rd) number drawn
          # Start:

          # If not rotated, only draw 1st point
          if (not doesRotate):
              if nx == 0:
                  ttl.goto(-100 + yf, zf)
                  ttl.dot(1)
                  ttl.penup()

          # Otherwise, draw all points
          else:
              ttl.goto(-100 + yf, zf)
              ttl.dot(1)
```

```
            ttl.penup()

        # End indent
        nx += 1

    # Alter variables
    print('\n')
    nx = 0
    P0x = P0x + sgn * v2x
    P0y = P0y + v2y
    sgn = sgn * -1
    ny = ny + 1

return

######################################################################
## FUNCTION: Creates honeycomb lattice;
##        Draws all combinations of atomic locations within bound
##        box, by linearly combining two, 2-D vectors
##
##   [args]: ttl     - turtle object;
##        (xi, yi)   - initial x,y coords of ttl;
##        (Xb, Yb  ) - x,y bounds of simulation box;
##        (v1x, v1y) - x,y components of vector 1;
##        (v2x, v2y) - x,y components of vector 2;
##        doesRotate - boolean, for "There is a rotation";
##        theta, phi - rotation angles about Y, Z axis, respectively
##        scale   - scaling factor for problem; facilitates visualizations only
##        trace   - boolean, for "Trace atomic pairs"; facilitates visualizations
##        infile  - file for saving all atomic locations in
######################################################################

def honeycombLattice(ttl, xi, yi, Xb, Yb, v1x, v1y, v2x, v2y, doesRotate, theta, p

# Initialization
sgn = 1
nx = 0
ny = 0
```

```
P0x = xi #Should this be set to 0?
P0y = yi
pnewx = xi
pnewy = yi

atomCounter = 0

# Within boundary conditions
while ny * v2y <= Yb: #F<=
     while nx * v1x < Xb:

          ########## 2 Dim ####################

          # TRIANGLE TO HONEYCOMB

          # TRI code \\:
          # Get new point, (pnewx, pnewy)
          pnewx = P0x + nx * v1x
          pnewy = P0y + nx * v1y
          #pnewy = P0y + ny * v1y

          # Send these TRI code coords, to HONEY fcn
          pnewy1, pnewy2 = getHoneycombYCoordinates ( ttl, pnewy, scale )

          # Print both new coordinates

          # Coord set 1
          # POINT OF CONTENTION: Boundary Conditions


          if (Yb - (ny * v2y) >= v2y/2): # >=v2y
              ttl.goto(pnewx, pnewy1)
              ttl.pendown()
              ttl.dot(1)
              ttl.penup()

              # FUTURE FIX (1/2)
              print( str(pnewx) + ' ' + str(pnewy1))
```

```
# Coord set 2
# POINT OF CONTENTION: Boundary Conditions
# Conditional created, to not include lowermost, extra-boundarial points

if ny != 0:
    ttl.goto(pnewx, pnewy2)
    ttl.pendown()
    ttl.dot(1)
    ttl.penup()

    # FUTURE FIX (2/2)
    print( str(pnewx) + ' ' + str(pnewy2))


############ 3 Dim ######################


# Calculate 3 dim point
# Should only calculate them, if not at boundary points

# For the pnewy1, upper lattice point in the duo
# if within normal boundary, confirms to do upper point
if (Yb - (ny * v2y) >= v2y/2): # >=v2y
    xf1, yf1, zf1, atomCounter = conv2to3dim(pnewx, pnewy1, Xb, Yb, doesR


# For the pnewy2, lower lattice point in the duo
# if within normal boundary, confirms to do lower point
if ny != 0:
    xf2, yf2, zf2, atomCounter = conv2to3dim(pnewx, pnewy2, Xb, Yb, doesR


#xf1, yf1, zf1 = conv2to3dim(pnewx, pnewy1, Xb, Yb, doesRotate, theta, ph
#xf2, yf2, zf2 = conv2to3dim(pnewx, pnewy2, Xb, Yb, doesRotate, theta, ph

# DRAW 3 dim point
# CHANGE: if you want ONLY the nth (1st, 3rd) number drawn
```

```
# Start:

# If at upper boundary region, keep it simple, and only draw that pt
if (Yb - (ny * v2y) >= v2y/2) and (ny == 0): # >=v2y
    ttl.goto(-100 + yf1, zf1)
    ttl.dot(1)
    ttl.penup()

# Now, if at lower boundary region
elif (Yb - (ny * v2y) < v2y/2) and (ny != 0):
    ttl.goto(-100 + yf2, zf2)
    ttl.dot(1)
    ttl.penup()

# If not rotated, only draw 1st point
elif (not doesRotate):
    if nx == 0:
        ttl.goto(-100 + yf1, zf1)
        ttl.dot(1)
        if trace:
            ttl.pendown()
        else:
            ttl.penup()
        ttl.goto(-100 + yf2, zf2)
        ttl.dot(1)
        ttl.penup()

# Otherwise, draw all points
else:
    ttl.goto(-100 + yf1, zf1)
    ttl.dot(1)
    if trace:
        ttl.pendown()
    else:
        ttl.penup()
    ttl.goto(-100 + yf2, zf2)
    ttl.dot(1)
    ttl.penup()
```

```
        # End indent
        nx += 1

    # Alter variables
    print('\n')
    nx = 0
    P0x = P0x + sgn * v2x
    P0y = P0y + v2y
    sgn = sgn * -1
    ny = ny + 1

return


##############################################################################
## FUNCTION: Converts 2-Dim coordinate set into 3-Dim using
##        (x,y) => (x,y,z)
##              =  (x, r*sin(th), r*(1-cos(th)))
##                  where r = L_y/ 2*pi, th = y_i / r
##
## OPTIONAL: Rotate 3-dim coords by theta, phi
##        Controlled by boolean 'doesRotate'
##
##  NOTE: For visualizations, when rolled into 3D, the 2D object is seen
##        from -x looking towards +x; i.e. it is the LHS y-side that is
##        seen, from edge on, in 3D
##
##   [args]: (x_i, y_i) - initial x,y coordinates
##        (Xb, Yb)   - boundary lengths
##        doesRotate - boolean, for "There is a rotation"
##        theta   - rotation angle about Y-axis
##        phi     - rotation angle about Z-axis
##        scale   - scaling factor, for visualizations
##        infile  - file for saving atomic locations in
##        atomCounter- integer, counts number of atoms assigned a location
##############################################################################
```

```
def conv2to3dim(x_i, y_i, Xb, Yb, doesRotate, theta, phi, scale, infile, atomCount

# Initialization
r = ( Yb ) / ( 2 * math.pi ) # Radius; should work for cases when y_i != 1
#print(">>>>>RADIUS: " + str(r))

# Update
atomCounter = atomCounter + 1

# Converting a single point
x_f = x_i
y_f = r * math.sin( y_i / r)
z_f = r * (1 - math.cos ( y_i / r) )

# If we do NOT have rotation, doesRotate = False
if (not doesRotate):
     # Print coords, and return them

     print(str(x_f) + ' ' + str(y_f) + ' ' + str(z_f))
     infile.write(str(atomCounter) +' ' +  str(1)+ ' ' +str(0.0) + ' ' + str(x_f)
     infile.write('\n')
     return x_f, y_f, z_f, atomCounter

# Otherwise, we DO have rotation
# Send final coordinates to rotate3dim
else:
     x_f, y_f, z_f = rotate3dim ( x_f, y_f, z_f, theta, phi )

     # FUTURE FIX
     print(str(x_f) + ' ' + str(y_f) + ' ' + str(z_f))
     infile.write(str(atomCounter) +' ' +  str(1) + ' ' +str(0.0) + ' ' + str(x_f)
     infile.write('\n')
     return float(x_f), float(y_f), float(z_f), atomCounter


#########################################################################
## FUNCTION: Rotates 3-dim coordinate set (x,y,z)  by theta, phi
```

```
##
##   [args]: (x_f, y_f, z_f) - finalized, 3-dim coordinates; unrotated
##        theta        - rotation angle about Y axis
##        phi          - rotation angle about Z axis
##
#############################################################################

def rotate3dim ( x_f, y_f, z_f, theta, phi ):
#Rotation matrix for R_y * R_z
rotate = matrix( [ [cos(theta)*cos(phi), cos(theta) * sin(phi), - sin(theta)],[-si
rotated=rotate*matrix([[x_f],[y_f],[z_f]])

# Returns rotated 3-dim x,y,z coordinates
return float(rotated[0]), float(rotated[1]),float(rotated[2])


#############################################################################
## FUNCTION: Gets the 2 replacement, y coordinates, for the honeycomb lattice;
##        Uses points from the triangle lattice, to determine honeycomb
##        point locations (y components only)
##
##   [args]: ttl   - turtle object
##        yi - initial y position, for triangle lattice
##        scale - scaling factor, for visualization facilitation
##
#############################################################################

def getHoneycombYCoordinates ( ttl, yi , scale) :

# Initialization ~ triangular lattice replacement vectors for honeycomb
# Note: No Vx components exist
Vy1 = 1.0 / (2.0*math.sqrt(3)) * scale
Vy2 = -1.0 / (2.0*math.sqrt(3)) * scale

# Calculate the 2 new locations
yf1 = yi + Vy1
yf2 = yi - Vy1
```

```
# Return the 2 new y-positions
return yf1, yf2

######################################################################
## MAIN FUNCTION
##
##
######################################################################

def main():

# Turtle initialization
turtle.setup(1000,1000,0,0)
turtle.screensize(100,100)
turtle.speed(0)
ttl = turtle.Turtle()

# Scale initialization
d_carbonBondLength = 1.42 #Angstroms
scale = math.sqrt(3) * d_carbonBondLength  # Ref. notebook 5/26
                                            #25 good for visuals


# Basis vector designation
v1x = 1*scale #S
v1y = 0 * scale #S
v2x = 0.5 * scale #S
v2y = math.sqrt(3)/2 * scale #S

# Boundary setting ~ (basis vector) * (scale) * (number of atoms spaced)
Xb = 1 * scale * 70  # S              # Yields len = 172.16585 A
Yb = math.sqrt(3)/2 * scale * 25 # S  # Yields dia = 16.950001 A ~ 1.7 nm
                                        # dia = Yb / pi

doesRotate = False
trace = False
theta = pi/4
phi = pi/4
```

```
# Because I want to write all atomic locations to a text file
infile = open("honeycombAtomicPlacement.txt","w")


drawBound(ttl, 0, 0, 1*Xb,1*Yb)

infile.close()
ttl.penup()
ttl.goto(-150,-150)
turtle.done()

main()
```

A.2 Python - methaneClass.py

```
 # To determine atomic locations of methane, using classes

import math

class Methane:
    def __init__(self, Xorigin, Yorigin, Zorigin):
        self.xOrigin = Xorigin
        self.yOrigin = Yorigin
        self.zOrigin = Zorigin
        self.currentLocation = [Xorigin, Yorigin, Zorigin]

    # Update place keeping of atomic location
    def updateCurrent(self, xCurrent, yCurrent, zCurrent):
        self.currentLocation = [xCurrent, yCurrent, zCurrent]

    # Check y and z validity, because x is automatically propogated
    # Only y-z plane needs to be within radius of CNT
    #    If valid, updates updateCurrent piece
    def isValidMethaneLocation(self, yInQuestion,zInQuestion, R_cnt):
        if math.hypot(self.yOrigin-yInQuestion, self.zOrigin - zInQuestion) < R_cr
```

```
            self.updateCurrent(self.currentLocation[0], yInQuestion, zInQuestion)
            return True
        else:
            return False

    # Methane template
    # Returns a list of new methane locations
    def returnsMethaneAtomicLocations(self):
        y = [[0,0,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0]]
        # This is molecular template (ref 10)
        x = [[0,0,0],[ 0.86500,  -0.02200,   0.60400],[-0.91100 , -0.03400 ,  0.58
        # Go through each spot and update template

        for row in range(0,5):
            for col in range(0,3):
               y[row][col] = self.currentLocation[col] + x[row][col]
               #print(y[row][col])

        self.List =y
        #return y

    def __str__(self):
        for i in range(len(y)):
            print(self.List)

def main():
    print('success')

    #instance of methane
    methane = Methane(2.2,0,16.15/2.0)


    #######
    # Necessary info
    r_atom = 1.1
    R_cnt = 16.95/2.0

    # Because I want to write all atomic locations to a text file
```

```
    infile = open("methaneAtomicPlacement.txt","w")

    # go through all possible locations
    ix = r_atom*2
    jy = 0
    kz = 0

    s = 3501
    while ix < 171:
        for jy in range(0,10):
            for kz in range(0,10):
                # Check if latest iteration is valid
                if methane.isValidMethaneLocation(-16.15/2.0 + jy*3, kz * 3, R_cnt
                    # If valid, write outputs to file
                    methane.returnsMethaneAtomicLocations()
                    for p in range(5):
                        if p == 0: # Then it is carbon (type 3)
                            infile.write(str(s) +' ' + str(3) + ' ' +str(0.0)+' '
                        else:      # else, it is hydrogen (type 2)
                            infile.write(str(s) +' '+ str(2) + ' ' +str(0.0)+' '+
                        s += 1
                    print(methane.List)



        ix += r_atom*7
        methane.updateCurrent(ix, jy, kz)

    # Necessary file closure
    infile.close()
main()
```

A.3 LAMMPS - in.meamCNT

Carbon Nanotube code

```
# MEAM potential for CNT system

dimension 3
units     lj
boundary    p p p

atom_style    charge


print '==> TEST: Marker 1'

read_data    data.meamCNT
          # Reads in: num atoms, num atom types, sim box size, atomic locations

print '==> TEST: Marker 2 set'

group CNT type 1

pair_style    meam
pair_coeff    * * library2.meam C NULL C
neighbor    0.3 bin
neigh_modify    delay 10


#fix     1 all nve
thermo     10
timestep    1e-3                #.001



dump     3 all movie 5 RefiningCNT.mpg element element &
     axes yes 0.8 0.02 view 93 -180 zoom 11 adiam 1.5 size 1028 1028

run     5000
```

A.4 LAMMPS - Methane

```
# REAX potential for methane

units     lj

boundary     p p p

atom_style    charge
read_data     data.makeItMethane

pair_style    reax/c lmp_control
pair_coeff    * * ffield.reax.cho C H C


neighbor    2 bin
neigh_modify    every 10 delay 0 check no

fix        1 all nve
fix          2 all qeq/reax 1 0.0 10.0 1e-6 param.qeq
fix          3 all temp/berendsen 340.0 340.0 20.0


timestep    1e-3#0.25
thermo     10

dump     3 all movie 5 methaneMOVES.mpg element element &
     axes yes 0.8 0.02 view 25 25 zoom 5 adiam 1.5 size 1028 1028

run     3000
```

A.5 LAMMPS - Combined Script

```
# Combining methane and CNT scripts

dimension    3
units     lj

boundary    p p p
atom_style    charge

read_data    data.realDataCombo
    # 1 = C (CNT); 2 = H (Methane); 3 = C (Methane)

print 'success 1'

group      CNT type 1
group     methane type 2 3

print 'success 2'

pair_style  reax/c lmp_control
print ' 2.2 '
pair_coeff  * * ffield.reax.cho C H C

print 'success 3'

neighbor     2 bin
neigh_modify     every 10 delay 0 check no

fix     1 methane nve
fix     2 methane qeq/reax 1 0.0 10.0 1e-6 param.qeq  # From methane
fix     3 methane temp/berendsen 500.0 500.0 100.0 # From methane
fix      4 methane addforce 100.0 0.0 0.0
fix      5 methane ave/spatial 10 5 50 x 0 171 vx density/mass norm all file ve2.p

thermo     10  # From CNT
timestep    1e-3
```

```
dump     3 all movie 5 realestIllest.mpg element element &
     axes yes 0.8 0.02 view 93 -120 zoom 8 adiam 1.5 size 3000 3000
# 93 -145

run      5000
```

B Data Files

B.1 data.realDataCombo

Beginning of combined data file for input atomic locations

```
# Methane and CNT read-ins

5150 atoms
3 atom types

0 171 xlo xhi
-50 50 ylo yhi
-50 50 zlo zhi

Masses

1 12.0107
2 1.0080
3 12.0107

Atoms

1 1 0.0 0.0 0.70916978246 0.0297230203843
2 1 0.0 2.45951214675 0.70916978246 0.0297230203843
3 1 0.0 4.9190242935 0.70916978246 0.0297230203843
4 1 0.0 7.37853644024 0.70916978246 0.0297230203843
5 1 0.0 9.83804858699 0.70916978246 0.0297230203843
. . .
```

B.2 library.meam modification

```
'C'      'dia'   4.   6         12.0111
4.38     4.25 2.800   2.00      5.00  3.567   7.37 1.18
1.0      3.2          1.44      -4.48      1.00 3
```