Turnitin Originality Report

Processed on: 24-Jun-2021 10:44 WIB
ID: 1611391640
Word Count: 22742
Submitted: 1

How to Modify LAMMPS By Iwan Sahputra

Similarity Index	Similarity by Source	
5%	Internet Sources:	3%
	Publications:	0%
	Student Papers:	15%

5% match (student papers from 12-May-2020)
Submitted to Cranfield University on 2020-05-12

chemengineering Article How to Modify LAMMPS: From the Prospective of a Particle Method Researcher Andrea Albano 1,* , Eve le Guillou 2, Antoine Danzé 2, Irene Moulitsas 2 , Iwan H. Sahputra 1,3, Amin Rahmat 1, Carlos Alberto Duque-Daza 1,4, Xiaocheng Shang 5 , Khai Ching Ng 6, Mostapha Ariane 7 and Alessio Alexiadis 1,* ?????????? ??????? Citation: Albano, A.; le Guillou, E.; Danzé, A.; Moulitsas, I.; Sahputra, I.H.; Rahmat, A.; Duque-Daza, C.A.; Shang, X.; Ching Ng, K.; Ariane, M.; et al. How to Modify LAMMPS: From the Prospective of a Particle Method Researcher. ChemEngineering 2021, 5, 30. https://doi.org/10.3390/ chemengineering5020030 Academic Editors: Mark P. Heitz and Andrew S. Paluch Received: 11 January 2021 Accepted: 26 May 2021 Published: 13 June 2021 Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affil- iations. Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https:// creativecommons.org/licenses/by/ 4.0/). 1 2 3 4 5 6 7 * School of Chemical Engineering, University of Birmingham, Birmingham B15 2TT, UK; halimits@yahoo.com (I.H.S.); A.Rahmat@bham.ac.uk (A.R.); C.A.Duque-Daza@bham.ac.uk (C.A.D.-D.) Centre for Computational Engineering Sciences, Cranfield University, Bedford MK43 0AL, UK; Eve.M.Le-Guillou@cranfield.ac.uk (E.l.G.); A.Danze@cranfield.ac.uk (A.D.); i.moulitsas@cranfield.ac.uk (I.M.) Industrial Engineering Department, Petra Christian University, Surabaya 60236, Indonesia Department of Mechanical and Mechatronic Engineering, Universidad Nacional de Colombia, Bogotá 111321, Colombia School of Mathematics, University of Birmingham, Edgbaston, Birmingham B15 2TT, UK; X.Shang.1@bham.ac.uk Department of Mechanical, Materials and Manufacturing Engineering, University of Nottingham Malaysia, Jalan Broga, Semenyih 43500, Malaysia; KhaiChing.Ng@nottingham.edu.my Department of Materials and Engineering, Sayens-University of Burgundy, 21000 Dijon, France; Mostapha.Ariane@u-bourgogne.fr Correspondence: axa1220@student.bham.ac.uk or aaalbano@gmail.com (A.A.); a.alexiadis@bham.ac.uk (A.A.) Abstract: LAMMPS is a powerful simulator originally developed for molecular dynamics that, today, also accounts for other particle-based algorithms such as DEM, SPH, or Peridynamics. The versatility of this software is further enhanced by the fact that it is open-source and modifiable by users. This property suits particularly well Discrete Multiphysics and hybrid models that combine multiple particle methods in the same simulation. Modifying LAMMPS can be challenging for researchers with little coding experience. The available material explaining how to modify LAMMPS is either too basic or too advanced for the average researcher. In this work, we provide several examples, with increasing level of complexity, suitable for researchers and practitioners in physics and engineering, who are familiar with coding without been experts. For each feature, step by step instructions for implementing them in LAMMPS are shown to allow researchers to easily follow the procedure and compile a new version of the code. The aim is to fill a gap in the literature with particular reference to the scientific community that uses particle methods for (discrete) multiphysics. Keywords: LAMMPS; particle method; discrete multiphysics 1. Introduction LAMMPS, acronym for Large-scale Atomic/Molecular Massively Parallel Simulator, was originally written in F77 by Steve Plimpton [1] in 1993 with the goal of having a large-scale parallel classical Molecular Dynamic (MD) code. The project was a Cooperative Research and Development Agreement (CRADA) between two DOE labs (Sandia and LLNL) and three companies (Cray, Bristol Myers Squibb, and Dupont). Since the initial release LAMMPS has been improved and expanded by many researchers who imple- mented many mesh-free computational methods such as Perydynamics, Smoothed particle hydrodynamics (SPH), Discrete Elemet Method (DEM) and many more [2–11]. Such a large number of computational methods within the same simulator allows researchers to easily combine them for the simulation of complex phenomena. In particular, our research group has used during the years LAMMPS in a variety of settings that go from ChemEngineering 2021, 5, 30. https://doi.org/10.3390/chemengineering5020030 https://www.mdpi.com/journal/chemengineering classic Molecular Dynamics [12–16], to Discrete Multiphysics simulations of cardiovascular flows [17–20], Modelling drug adsorption in human organs [21–24], Cavitation [25–27], multiphase flow containing cells or capsules [28–31], solidification/dissolution [32–34], material properties [35,36] and even epidemiology [37] and coupling particles methods with Artificial Intelligence [38–40]. An example of a Discrete Multiphysics simulation run with the basic LAMMPS's code is shown in Appendix A. Thanks to its modular design open source nature and its large community, LAMMPS has been conceived to be modified and expanded by adding new features. In fact, about 95% of its source code is add-on file [41]. However, this can be a tough challenge for researcher with no to little knowledge of coding. The LAMMPS user manual [41] describes the internal structure and algorithms of the code with the intent of helping researcher to expand LAMMPS. However, due to the lack of examples of implementation and validation, the document can be hard to read for user who are not programmers. In fact, the available material is either very basic [41] or requires advanced programming skills [42,43]. The aim of this work is to provide several step-by-step examples with increasing level of complexity that can fill the gap in the middle to help and encourage researchers to use LAMMPS for discrete multiphysics and expand it with new adds on to the code that could fit their needs. In fact, most of the available material focuses on Molecular Dynamics (MD) and implicitly assumes that the reader 's background is in MD rather than other particle methods such as SPH or DEM. On the contrary, this paper is dedicated to the particle community and highlights how LAMMPS can be used and modified for methods other than MD. This goal fits particularly well with the scope of this Special Issue on "Discrete Multiphysics: Modelling Complex Systems with Particle Methods" In particular, it relates to some of the topics of the Special Issue such by exploring the potential of LAMMPS for coupling particle methods, and by sharing some "tricks of the trade" on how to modify its code that cannot be found anywhere else in the literature. In Section 2 LAMMPS structure and hierarchy are explained introducing the concept of style. Following the LAMMPS authors advice, to avoid writing a new style from scratch, Sections 3–6 new styles are developed using existing style are as reference. Finally, in Section 7, all the steps to write a class from scratch are shown. 2. LAMMPS Structure After initial releases in F77 and F90, LAMMPS is now written in C++, an object oriented language that allows any programmer to exploit the class programming paradigm. The declaration of a class, including the signature of the instance variables and functions (or methods), which can be accessed and used by creating an instance of that class. The data and functions within a class are called members of the class. The definition (or implementation) of a member function can be given inside or outside the class definition. A class has private, public, and protected sections which contain the corresponding class members. • The private members, defined before the keyword public, cannot be accessed from outside the class. They can only be accessed by class or “friend” functions, which are declared as having access to class members, without themselves being members. All the class members are private by default. • The public members can be accessed from outside the class anywhere within the scope of the class object. • The protected members are similar to private members but they can be accessed by derived classes or child classes while private members cannot. 2.1. Inheritance An important concepts in object-oriented programming is that of inheritance. Inher- itance allows to define a class in terms of another class and the new class inherits the members of the existing class. This existing class is called the base (or parent) class, and the new class is referred to as a subclass, or child class, or derived class. The idea of inheritance implements the “is a” relationship. For example, Mammal IS-A Animal, Dog IS-A Mammal hence Dog IS-A Animal as well. The inheritance relationship between the parent and the derived classes is declared in the derived class with the following syntax: Listing 1: C++ syntax for classes inheritance 1 class name_child_class: access_specifier name_parent_class 2 { /*...*/ }; The type of inheritance is specified by the access-specifier, one of public, protected, or private. If the access-specifier is not used, then it is private by default, but public inheritance is commonly used: public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class. A base class’s private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class. 2.2. Virtual Function The signature of a function f must be declared with a virtual keyword in a base class C to allow its definition (implementation), or redefinition, in a derived class D. Then, when a derived class D object is used as an element of the base class C, and f is called, the derived class’s implementation of the function is executed. There is nothing wrong with putting the virtual in front of functions inside of the derived classes, but it is not required, unless it is known for sure that the class will not have any children who would need to override the functions of the base class. A class that declares or inherits a virtual function is called a polymorphic class. 2.3. LAMMPS Inheritance and Class Syntax A schematic representation of the LAMMPS inheritance tree is shown in Figure 1: LAMMPS is the top-level class for the entire code, then all the core classes, highlighted in blue, inherit all the constructors, destructors, assignment operator members, friends and private members declared and defined in LAMMPS. The core classes perform LAMMPS fundamental actions. For instance, the Atom class collects and stores all the per-atom, or per-particle, data while Neighbor class builds the neighbor lists [41]. The style classes, highlighted in reds, inherit all the constructors, destructors, assign- ment operator members, friends and private members declared and defined in LAMMPS and in the corresponding core class. The style classes are also virtual parents class of many child classes that implement the interface defined by the parent class. For example, the fix style has around 100 child classes. Each style is composed of a pair of files: • namestyle.h The header of the style, where the class style is defined and all the objects, methods and constructors are declared. • namestyle.cpp Where all the objects, methods and constructors declared in the class of style are defined. When a new style is written both namestyle.h and namestyle.cpp files need to be created. Each “family” style has its own set of methods, declared in the header and defined in the cpp file, in order to define the scope of the style. For example, the pair style are classes that set the formula(s) LAMMPS uses to compute pairwise interactions while bond style set the formula(s) to compute bond interactions between pairs of atoms [41]. Each pair style has some recurrent functions such as compute, allocate and coeff. Although the final scope of those functions can differ for different styles, they all share a similar role within the classes. ChemEngineering 2021, 1, 1 4 of 64 LAMMPS Force Timer Memory Figure 1. Class hierarchy within LAMMPS source code. An example of a pair style, sph/taitwater, header in LAMMPS is shown in Listing 2. Listing 2: Header file of sph/taitwater pair style (pair_sph_taitwater.h) 1 class PairSPHTaitwater: public Pair{// class definition, accessibility and Inheritance 2 public: // access specifier: public 3 // public methods 4 PairSPHTaitwater(class LAMMPS *); // Constructors 5 virtual ~PairSPHTaitwater(); // Destructors 6 virtual void compute(int, int); 7 void settings(int, char **); 8 void coeff(int, char **); 9 virtual double init_one(int, int); 10 virtual double single(int, int, int, int, double, double, double, double &); 11 12 protected: // access specifier: protected 13 double *rho0, *soundspeed, *B; 14 double **cut,**viscosity; 15 int first; 16 // protected methods void allocate(); }; All the class members are defined in the cpp file. Taking sph/taitwater pair style as reference, each method declared in Listing 2 will be defined and commented in the next sections. Although this can be style-specific, the aim is to give an overview of how the methods are defined in the cpp in LAMMPS. Albeit different style has different methods, the understanding gained can be transferred into others style, as shown in Sections 3 and 6. 2.3.1. Constructor Any class usually include a member function called constructors. The constructor is mechanically invoked when an object of the class is created. This allows the class to initialise members or allocate storage. Unlike the other member of the class, the constructor name must match the name of the class and it does not have a return type. Listing 3: Constructor definition in sph/taitwater pair style (pair_sph_taitwater.cpp) 1 PairSPHTaitwater::PairSPHTaitwater(LAMMPS *lmp) : Pair(lmp) 2 { 3 restartinfo = 0; 4 first = 1; 5 } 2.3.2. Destructor The role of destructors is to de-allocate the allocated dynamic memory, see Section 2.3.8, being mechanically invoked just before the end of the class lifetime. Similarly to construc- tors, destructors does not have a return type and have the same name as the class name with a tilde (~) prefix. Listing 4: Destructors definition in sph/taitwater pair style (pair_sph_taitwater.cpp) 1 PairSPHTaitwater::~PairSPHTaitwater() { 2 if (allocated) { /// check if the pair style uses allocate, see Section 2.8 3 /// cleanup the memory used by allocate, see Section 2.8 4 memory->destroy(setflag); 5 memory->destroy(cutsq); 6 memory->destroy(cut); 7 memory->destroy(rho0); 8 memory->destroy(soundspeed); 9 memory->destroy(B); 10 memory->destroy(viscosity); 11 } 12 } 2.3.3. compute compute is virtual member of the pair style and is one of the most relevant functions in a number of classes in LAMMPS. For instance, in pair style classes is used to compute pairwise interaction of the specific pair style. This can seen in the commented Listing 5, where the force applied on a pair of neighboring particles is derived using the Tait equa- tion, lines 131–151. In compute all the local parameters needed to compute the pairwise interaction are declared and defined within the method. Listing 5: compute definition in sph/taitwater pair style (pair_sph_taitwater.cpp) 1 2 3 4 5 6 7 8 9 void PairSPHTaitwater::compute(int eflag, int vflag) { /// start variables and pointer declaration int i, j, ii, jj, inum, jnum, itype, jtype; double xtmp, ytmp, ztmp, delx, dely, delz, fpair; int *ilist, *jlist, *numneigh, **firstneigh; double vxtmp, vytmp, vztmp, imass, jmass, fi, fj, fvisc, h, ih, ihsq; double rsq, tmp, wfd, delVdotDelR, mu, deltaE; // end 11 12 if (eflag || vflag) 13 ev_setup(eflag, vflag); 14 else 15 evflag = vflag_fdotr = 0; 16 17 /// others variables and pointers declaration and initialisation 18 double **v = atom->vest; // pass the value of the pointer that points to a pointers 19 // pointing to the first element of velocity vector of the particles 20 double **x = atom->x; // pass the value of the pointer that points to a pointers 21 // pointing to the first element of position vector of the particles 22 double **f = atom->f; // pass the value of the pointer that points to a pointers 23 // pointing to the first element of force vector of the particles 24 double *rho = atom->rho; // pass the value of the pointer that points 25 // to the density vector of the particles 26 double *mass = atom->mass; // pass the value of the pointer that points 27 // to the mass vector of the particles 28 double *de = atom->de; // pass the value of the pointer that points 29 // to the change of internal energy of the particles 30 double *drho = atom->drho; // pass the value of the pointer that points 31 // to the change of density of the particles 32 int *type = atom->type; // pass the value of the pointer that points to the type of the particles 33 int nlocal = atom->nlocal; // pass the value of the numbers of owned and ghost atoms on this proc 34 int newton_pair = force->newton_pair; // pass the value of the Newton’s 3rd law settings 35 /// end 36 37 38 // check consistency of pair coefficients 39 40 if (first) { 41 for (i = 1; i <= atom->ntypes; i++) { 42 for (j = 1; i <= atom->ntypes; j++) { 43 if (cutsq[i][j] > 1.e-32) { 44 if (!setflag[i][i] || !setflag[j][j]) { 45 if (comm->me == 0) { 46 printf(47 "SPH particle types %d and %d interact with cutoff=%g, 48 but not all of their single particle properties are set.\n", 49 i, j, sqrt(cutsq[i][j])); 50 } } } } } 51 first = 0; 52 } 53 54 55 inum = list->inum; // pass the value of number of I atoms neighbors are stored for 56 ilist = list->ilist; // pass the value of the pointer pointing to the local indices of I atoms 57 numneigh = list->numneigh; // pass the address of a pointer pointing to the number of J neighbors 58 // for each I atom 59 firstneigh = list->firstneigh; // pass the value of a pointer that points to pointer 60 // pointing to 1st J int value of each I atom 61 62 63 64 65 for (ii = 0; ii < inum; ii++) { // loop for each i particles stored in inum 66 i = ilist[ii]; // pass the index of the i particle 67 xtmp = x[i][0]; // pass the x position of the i particle 68 ytmp = x[i][1]; // pass the y position of the i particle 69 ztmp = x[i][2]; // pass the z position of the i particle 70 vxtmp = v[i][0]; // pass the x velocity of the i particle 71 vytmp = v[i][1]; // pass the y velocity of the i particle 72 vztmp = v[i][2]; // pass the z velocity of the i particle 73 itype = type[i]; // pass the type of the i particle 74 jlist = firstneigh[i]; // pass the 1st J int value of each I atom 75 jnum = numneigh[i]; //pass number of J neighbors for each I atom imass = mass[itype]; // pass the mass of the i particle 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 // compute force of atom i with Tait EOS tmp = rho[i] / rho0[itype]; fi = tmp * tmp * tmp; fi = B[itype] * (fi * fi * tmp - 1.0) / (rho[i] * rho[i]); // end for (jj = 0; jj < jnum; jj++) { //

loop over neighbours list of particle i j = jlist[jj]; // pass the index of the j particle j &= NEIGHMASK; delx = xtmp - x[j][0]; // x distance between particles i and j dely = ytmp - x[j][1]; // y distance between particles i and j delz = ztmp - x[j][2]; // z distance between particles i and j rsq = delx * delx + dely * dely + delz * delz; // squared distance between particles i and j 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 jtype = type[j]; // pass the type of the i particle jmass = mass[jtype]; // pass the mass of the j particle if (rsq < cutsq[itype][jtype]) { // check if i and j are neighbor h = cut[itype][jtype]; // pass the smoothing length ih = 1.0 / h; // calculate the inverse, divisions are computationally expensive ihsq = ih * ih ; // squared inverse wfd = h - sqrt(rsq); if (domain->dimension == 3) { // Lucy Kernel, 3d wfd = -25.066903536973515383e0 * wfd * wfd * ihsq * ihsq * ihsq * ih; } else { // Lucy Kernel, 2d wfd = -19.098593171027440292e0 * wfd * wfd * ihsq * ihsq * ihsq; } // compute force of atom j with Tait EOS tmp = rho[j] / rho0[jtype]; fj = tmp * tmp * tmp; fj = B[jtype] * (fj * fj * tmp - 1.0) / (rho[j] * rho[j]); // end // dot product of velocity delta and distance vector delVdotDelR = delx * (vxtmp - v[j][0]) + dely * (vytmp - v[j][1]) + delz * (vztmp - v[j][2]); // artificial viscosity (Monaghan 1992) if (delVdotDelR < 0.) { mu = h * delVdotDelR / (rsq + 0.01 * h * h); fvisc = -viscosity[itype][jtype] * (soundspeed[itype] + soundspeed[jtype]) * mu / (rho[i] + rho[j]); } else { fvisc = 0.; } fpair = -imass * jmass * (fi + fj + fvisc) * wfd; // total pair force deltaE = -0.5 * fpair * delVdotDelR; // internal energy increment // change in force in each direction for particle i f[i][0] += delx * fpair; f[i][1] += dely * fpair; f[i][2] += delz * fpair; //change in density for particle i drho[i] += jmass * delVdotDelR * wfd; // change in internal energy for particle i de[i] += deltaE; if (newton_pair || j < nlocal) { // change in force in each direction for particle j 147 f[j][0] -= delx * fpair; 148 f[j][1] -= dely * fpair; 149 f[j][2] -= delz * fpair; 150 151 de[j] += deltaE; // change in internal energy for particle j 152 153 drho[j] += imass * delVdotDelR * wfd; // change in density for particle j 154 } 155 156 if (evflag) 157 ev_tally(i, j, nlocal, newton_pair, 0.0, 0.0, fpair, delx, dely, delz); 158 } 159 } 160 } 161 162 if (vflag_fdotr) virial_fdotr_compute(); 163 } 2.3.4. settings settings is a public void function that reads the input script checking that all the arguments of the pair style are declared. If arguments are present, settings stores them so they can be used by compute. Examples for no arguments pair style and arguments pair style input script with the corresponding settings are listed below: • No arguments pair style: sph/taitwater As described in the SPH for LAMMPS manual [6], the command line to invoke the sph/taitwater pair style is shown in Listing 6. Listing 6: Command line to invoke sph/taitwater pair style 1 pair_style sph/taitwater In this pair style there is just a string defining the pair style, sph/taitwater, with no arguments. For this reason in settings, Listing 7, when the if statement is true (number of arguments other than zero) an error is produced. Listing 7: setting definition in sph/taitwater pair style (pair_sph_taitwater.cpp) 1 void PairSPHTaitwater::settings(int narg, char **arg) { 2 if (narg != 0) /// check the number of arguments 3 error->all(FLERR,"Illegal number of setting arguments for pair_style sph/ taitwater"); 4 } • Arguments pair syle: sph/rhosum As described in the SPH for LAMMPS manual [6], the command line to invoke the sph/rhosum pair style is shown in Listing 8. Listing 8: Command lines to invoke sph/rhosum pair style 1 pair_style sph/rhosum Nstep In this pair style there is a string defining the pair style, sph/rhosum, plus one argu- ment, Nstep. For this reason in settings, Listing 9, when the if statement is true (number of arguments other than one) an error is produced. When the if statement is false settings assigns the value of Nstep in the variable nstep, line 5, by using the inumeric function defined in the force class. Listing 9: setting definition in sph/rhosum pair style (pair_sph_rhosum.cpp) 1 2 3 4 void PairSPHRhoSum::settings(int narg, char **arg) { if (narg != 1) /// check the number of arguments error->all(FLERR, "Illegal number of setting arguments for pair_style sph/rhosum"); } nstep = force->inumeric(FLERR,arg[0]); // store the variable in the position 0 (Nstep) into nstep 2.3.5. coeff Similar to setting, coeff is a public void function that reads and set the coefficients used in by compute of the pair style. For each i j pair is possible to set different coefficients. The coefficients are passed in the input file with the command line pair coeff, see Listing 10. As before, examples for different pair coeff input script and the corresponding coeff are listed below: • sph/taitwater As described in the SPH for LAMMPS manual [6], the command line to invoke sph/taitwater pair coeff is shown in Listing 10. Listing 10: Command line to invoke sph/taitwater pair coeff 1 pair_coeff I J rho_0 c_0 alpha h In total there are six arguments. Thus, in coeff, Listing 11, when if statement is true (number of arguments other than six) an error is produced. When the if statement is false coeff assigns the type of particles I and J plus the value of rho_0, c_0, alpha and h in from the string to the variables by using the numeric function defined in force class. At last, within the double for loop from line 19 to 32, the variables are assigned for each particles. Listing 11: coeff definition in sph/taitwater pair style (pair_sph_taitwater.cpp) 1 void PairSPHTaitwater::coeff(int narg, char **arg) { 2 if (narg != 6) /// check the number of arguments 3 error->all(FLERR, 4 "Incorrect args for pair_style sph/taitwater coefficients"); 5 if (!allocated) /// check if allocate has been called 6 allocate(); /// call allocate, see section 2.8 7 8 int ilo, ihi, jlo, jhi; 9 force->bounds(arg[0], atom->ntypes, ilo, ihi); 10 force->bounds(arg[1], atom->ntypes, jlo, jhi); 11 12 /// store the variables in the position 2--5 13 double rho0_one = force->numeric(FLERR,arg[2]); 14 double soundspeed_one = force->numeric(FLERR,arg[3]); 15 double viscosity_one = force->numeric(FLERR,arg[4]); 16 double cut_one = force->numeric(FLERR,arg[5]); 17 /// B_one is a constant used in tait EOS inside compute, see section 2.3 18 double B_one = soundspeed_one * soundspeed_one * rho0_one / 7.0; 19 20 /// assign the coefficient to the corresponding particle (i) 21 /// and to the pair of particles (i,j) 22 int count = 0; 23 for (int i = ilo; i <= ihi; i++) { 24 rho0[i] = rho0_one; 25 soundspeed[i] = soundspeed_one; 26 B[i] = B_one; 27 for (int j = MAX(jlo,i); j <= jhi; j++) { 28 viscosity[i][j] = viscosity_one; 29 cut[i][j] = cut_one; 30 31 setflag[i][j] = 1; 32 33 count++; 34 } 35 } 36 if (count == 0) /// check if the arguments have been assigned 37 error->all(FLERR,"Incorrect args for pair coefficients"); 38 } • sph/rhosum As described in the SPH for LAMMPS manual [6], the syntax to invoke the command is shown in Listing 12. Listing 12: Command lines to invoke sph/rhosum pair style 1 pair_coeff I J h In this case there are three arguments. Thus, in the coeff, Listing 13, when the if statement is true (number of arguments other than six) an error is produced. When the error is not produced function assigns the type of particles I and J plus the value of h in the string to the variable cut_one, line 11, by using bounds and numeric function defined in force class. At last, within the double for loop from line 14 to 20, the variables are assigned for each particles. Listing 13: coeff definition in sph/rhosum pair style (pair_sph_rhosum.cpp) 1 void PairSPHRhoSum::coeff(int narg, char **arg) { 2 if (narg != 3) /// check the number of arguments 3 error->all(FLERR,"Incorrect number of args for sph/rhosum coefficients"); 4 if (!allocated) /// check if allocate has been called 5 allocate(); /// call allocate, see section 2.8 6 7 int ilo, ihi, jlo, jhi; 8 force->bounds(arg[0], atom->ntypes, ilo, ihi); 9 force->bounds(arg[1], atom->ntypes, jlo, jhi); 10 11 double cut_one = force->numeric(FLERR,arg[2]); 12 13 /// assign the coefficient to the pair of particles (i,j) 14 int count = 0; 15 for (int i = ilo; i <= ihi; i++) { 16 for (int j = MAX(jlo,i); j <= jhi; j++) { 17 cut[i][j] = cut_one; 18 setflag[i][j] = 1; 19 count++; 20 } 21 } 22 23 if (count == 0) /// check if the arguments have been assigned 24 error->all(FLERR,"Incorrect args for pair coefficients"); 25 } 2.3.6. init_one init_one check if all the pair coefficients for a given i j pair have been assigned. If they were assigned the methods ensure the symmetry of the matrix. Listing 14: init_one definition in sph/taitwater pair style (pair_sph_taitwater.cpp) 1 double PairSPHTaitwater::init_one(int i, int j) { 2 /// check if the coefficient of the pair of particles (i,j) were assigned 3 if (setflag[i][j] == 0) { 4 error->all(FLERR,"Not all pair sph/taitwater coeffs are set"); 5 } 6 /// ensure the matrix symmetry 7 cut[j][i] = cut[i][j]; 8 viscosity[j][i] = viscosity[i][j]; 9 10 return cut[i][j]; 11 } 2.3.7. single In single the force and energy of a single pairwise interaction, or single bond or angle (in case of bond or angle style), between two atoms is evalutated. The method is specifically invoked by the command line compute pair/local (or compute bond/local) to calculate properties of individual pair, or bond, interactions [41]. Listing 15: single definition in sph/taitwater pair style (pair_sph_taitwater.cpp) 1 double PairSPHTaitwater::single(int i, int j, int itype, int jtype, 2 double rsq, double factor_coul, double factor_lj, double &fforce) { 3 fforce = 0.0; 4 5 return 0.0; 6 } 2.3.8. allocate allocate is a protected void function that allocates dynamic memory. The dynamic memory allocation is used when the amount of memory needed depends on user input. As explained before, at the end of the lifetime of the class, the destructors will de-allocate the memory the memory used by allocate. Listing 16: allocate definition in sph/taitwater pair style (pair_sph_taitwater.cpp) 1 void PairSPHTaitwater::allocate() { 2 allocated = 1; /// confirm that allocated has been called 3 int n = atom->ntypes; /// assigm the value of the number of types 4 5 memory->create(setflag, n + 1, n + 1, "pair:setflag"); 6 for (int i = 1; i <= n; i++) 7 for (int j = i; j <= n; j++) 8 setflag[i][j] = 0; 9 10 /// allocate the memory for the arguments of the pair style 11 memory->create(cutsq, n + 1, n + 1, "pair:cutsq"); 12 memory->create(rho0, n + 1, "pair:rho0"); 13 memory->create(soundspeed, n + 1, "pair:soundspeed"); 14 memory->create(B, n + 1, "pair:B"); 15 memory->create(cut, n + 1, n + 1, "pair:cut"); 16 memory->create(viscosity, n + 1, n + 1, "pair:viscosity"); 17 } 3. Kelvin–Voigt Bond Style We can use what we learned in the previous section to generate a new dissipative bond potential that can be used to model viscoelastic materials. The Kelvin–Voigt model [44] is used to model viscoelastic material as a purely viscous damper and purely elastic spring connected in parallel as shown in Figure 2. Since the two components of the model are arranged in parallel, the strain in each component is identical:

εtot = εspring = εdamper. (1) On the other hand, the total stress σtot will be split into σspring and σdamper to have εspring = εdamper. Thus we have σtot = σspring + σdamper. (2) Combining Equations (1) and (2) with the constitutive relation for both the spring and the dumper, σspring = kε and σdamper = bε˙ , is possible to write that σ = kε(t) + b dt = kε(t) + bε˙, dε(t) (3) where k is the elastic modulus and b is the coefficient of viscosity. Equation (3) relates stress to strain and strain rate for a Kelvin–Voigt material [44]. where k is the elastic modulus and b is the coefficient of viscosity. Equation (3) relates stress to strain and strain rate for a Kelvin-Voigt material [44]. Figure 2. Schematic representation of Kelvin–Voigt model [44]. Figure 2. Schematic representation of Kelvin-Voigt model [44]. Similarly to bond test to write a new pair style called bond kv we take the bond Similharalrymtoonibcopnadir tsetystletaoswrerfietreenacen.eTwhepnaeiwrsptyailresctyalleleidsdbeocnladrekdvanwdeintiatkiaelistehde ibnobnodnd_kv.h harmonic panaidr bstoynled_aksvr.ecfpepresnacvee.dThinetnheew/spraci/rMstOylLeEisCdUeLcEladreidreacntodryinaintidaliitssedhiienrabrocnhdy_iksvs.hhown in and bond_Fkivg.ucrpep3s.aved in the /src/MOLECULE directory. LAMMPS Force Bond Figure 3. Class hierarchy of the new bond style. 3.1. Validation The bond kv pair style has been validated by Sahputra et al. [45] in their Discrete Multiphysics model for encapsulate particles with a soft outer shell. 3.2. bond_kv.cpp All the functions will be the same as in the reference bond harmonic. However, in our new bond kv, we need to substitute the "BondHarmonic" text by a new "BondKv" text, as can be seen in Listings 17 and 18. Form now on, when we show a side-by-side comparison between the reference and the modified file, we highlight in yellow the modified lines and in red the deleted lines. Listing 17: Original script (bond_harmonic.cpp) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 #include "math.h" #include "stdlib.h" #include "bond_harmonic.h" #include "atom.h" #include "neighbor.h" #include "domain.h" #include "comm.h" #include "force.h" #include "memory.h" #include "error.h" using namespace LAMMPS_NS; BondHarmonic::BondHarmonic(LAMMPS *lmp) : Bond(lmp) {} BondHarmonic::~BondHarmonic() { ... } void BondHarmonic::compute(int eflag, int vflag) { ... } void BondHarmonic::allocate() 21 22 { ... } void BondHarmonic::coeff(int narg, char **arg) 23 { ... } 24 double BondHarmonic::equilibrium_distance(int i) 25 { ... } 26 void BondHarmonic::write_restart(FILE *fp) 27 { ... } 28 void BondHarmonic::read_restart(FILE *fp) 29 { ... } 30 void BondHarmonic::write_data(FILE *fp) 31 { ... } 32 double BondHarmonic::single(int type, double rsq, 33 int i, int j, double &fforce) 34 { ... } Listing 18: Modified script (bond_kv.cpp) 1 #include "math.h" 2 #include "stdlib.h" 3 #include "bond_kv.h" 4 #include "atom.h" 5 #include "neighbor.h" 6 #include "domain.h" 7 #include "comm.h" 8 #include "force.h" 9 #include "memory.h" 10 #include "error.h" 11 12 using namespace LAMMPS_NS; 13 14 BondKv::BondKv(LAMMPS *lmp) : Bond(lmp) 15 {} 16 BondKv::~BondKv() 17 { ... } 18 void BondKv::compute(int eflag, int vflag) 19 { ... } 20 void BondKv::allocate() 21 { ... } 22 void BondKv::coeff(int narg, char **arg) 23 { ... } 24 double BondKv::equilibrium_distance(int i) 25 { ... } 26 void BondKv::write_restart(FILE *fp) 27 { ... } 28 void BondKv::read_restart(FILE *fp) 29 { ... } 30 void BondKv::write_data(FILE *fp) 31 { ... } 32 double BondKv::single(int type, double rsq, 33 int i, int j, double &fforce) 34 { ... } Compared to the bond harmonic we are introducing a new parameter, b, from the input file. For this reason we need to modify destructor, compute, allocate, coeff, write_restart and read_restart. Following the order of function initialisation, see Listing 18, the destructor is modified as shown in Listing 20. Listing 19: Original destructor (bond_harmonic.cpp) 1 BondHarmonic::~BondHarmonic() 2 { 3 if (allocated) { 4 memory->destroy(setflag); 5 memory->destroy(k); 6 memory->destroy(r0); 7 } 8 } Listing 20: Modified destructor (bond_kv.cpp) 1 BondKv::~ BondKv() 2 { 3 if (allocated) { 4 memory->destroy(setflag); 5 memory->destroy(k); 6 memory->destroy(r0); 7 memory->destroy(b); /* dashpot/damper costant */ 8 } 9 } The next function to modify is compute. The strain rate, ε˙, can also be seen as the speed of deformation. To use it within the new pair style we need to declared and initialised the velocities of each particles, see Listing 22. Listing 21: Original compute (bond_harmonic.cpp) 1 void BondTest::compute(int eflag, int vflag) 2 { 3 int i1,i2,n,type; 4 double delx,dely,delz,ebond,fbond; 5 double rsq,r,dr,rk; 6 7 ebond = 0.0; 8 if (eflag || vflag) ev_setup(eflag,vflag); 9 else evflag = 0; 10 11 double **x = atom->x; 12 double **f = atom->f; 13 } 14 } Listing 22: Modified compute (bond_kv.cpp) 1 void BondTest::compute(int eflag, int vflag) 2 { 3 int i1,i2,n,type; 4 double delx,dely,delz,ebond,fbond; 5 double rsq,r,dr,rk 6 7 /* declaration of new variables */ 8 double delv_x, delv_y, delv_z; 9 double dir_vx1, dir_vy1, dir_vz1, dir_vx2, dir_vy2, 10 dir_vz2, dir_vx1, dir_vx1; 11 double rsq_x, rsq_y, rsq_z; 12 /* end declaration of new variables */ 13 14 15 16 ebond = 0.0; 17 if (eflag || vflag) ev_setup(eflag,vflag); 18 else evflag = 0; 19 20 double **x = atom->x; 21 double **f = atom->f; 22 double **v = atom->v; /* delcaration and inizalitaizon 23 of a new pointer*/ 24 } 25 } Moreover, inside the loop for (n = 0; n < nbondlist; n++) of the original compute, we need to add a new set of lines between the lines to calculate the spring force and the lines to calculate force and energy increment. Those lines calculate velocities and directions to compute the dashpot forces, see Listing 23. Now is possible to write the new expression of the force applied to pair of atoms. Listing 23: Modified compute (bond_kv.cpp) 1 /* dashpot velocities and directions */ 2 dev_x = v[i1][0] - v[i2][0]; 3 dev_y = v[i1][1] - v[i2][1]; 4 dev_z = v[i1][2] - v[i2][2]; 5 rsq_vx = dev_x * dev_x; 6 rsq_vy = dev_y * dev_y; 7 rsq_vz = dev_z * dev_z; 8 velx = sqrt(rsq_vx); 9 vely = sqrt(rsq_vy); 10 velz = sqrt(rsq_vz); 11 12 if (v[i1][0] >= 0.0) dir_vx1 = 1; 13 else dir_vx1 = -1; 14 15 if (v[i1][1] >= 0.0) dir_vy1 = 1; 16 else dir_vy1 = -1; 17 18 if (v[i1][2] >= 0.0) dir_vz1 = 1; 19 else dir_vz1 = -1; 20 21 if (v[i2][0] >= 0.0) dir_vx2 = 1; 22 else dir_vx2 = -1; 23 24 if (v[i2][1] >= 0.0) dir_vy2 = 1; 25 else dir_vy2 = -1; 26 27 if (v[i2][2] >= 0.0) dir_vz2 = 1; 28 else dir_vz2 = -1; Listing 24: Original compute (bond_harmonic.cpp) 1 if (newton_bond || i1 < nlocal) { 2 f[i1][0] += delx*fbond; 3 f[i1][1] += dely*fbond; 4 f[i1][2] += delz*fbond; 5 } 6 7 if (newton_bond || i2 < nlocal) { 8 f[i2][0] -= delx*fbond; 9 f[i2][1] -= dely*fbond; 10 f[i2][2] -= delz*fbond; 11 } 12 13 if (evflag) ev_tally(i1,i2,nlocal, 14 newton_bond,ebond,fbond,delx,dely,delz); 15 } 16 } Listing 25: Modified compute (bond_kv.cpp) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 /// eq 3 implementation for each force component } if (newton_bond || i1 < nlocal) { f[i1][0] += (delx*fbond) - (dir_vx1*b[type]*velx); f[i1][1] += (dely*fbond) - (dir_vy1*b[type]*vely); f[i1][2] += (delz*fbond) - (dir_vy1*b[type]*velz); } if (newton_bond || i2 < nlocal) { f[i2][0] -= (delx*fbond) - (dir_vx2*b[type]*velx); f[i2][1] -= (dely*fbond) - (dir_vy2*b[type]*vely); f[i2][2] -= (delz*fbond) - (dir_vz2*b[type]*velz); } if (evflag) ev_tally(i1,i2,nlocal, newton_bond,ebond,fbond,delx,dely,delz); 17 } With the introduction of a new parameter in the pair style we need to make a new dynamic memory allocation by modifying allocate. Listing 26: Original allocate (bond_harmonic.cpp) 1 void BondHarmonic::allocate() 2 { 3 allocated = 1; 4 int n = atom->nbondtypes; 5 6 memory->create(k,n+1,"bond:k"); 7 memory->create(r0,n+1,"bond:r0"); 8 9 memory->create(setflag,n+1,"bond:setflag"); 10 for (int i = 1; i <= n; i++) setflag[i] = 0; 11 } Listing 27: Modified allocate (bond_kv.cpp) 1 void BondKv::allocate() 2 { 3 allocated = 1; 4 int n = atom->nbondtypes; 5 6 memory->create(k,n+1,"bond:k"); 7 memory->create(r0,n+1,"bond:r0"); 8 memory->create(b,n+1,"bond:b"); // new line to 9 // dynamically allocate b 10 11 memory->create(setflag,n+1,"bond:setflag"); 12 for (int i = 1; i <= n; i++) setflag[i] = 0; 13 } The viscosity of the damper, b, is given by the user in the input file. For this reason, we also need to modify coeff. Listing 28: Original coeff (bond_harmonic.cpp) 1 void BondHarmonic::coeff(int narg, char **arg) 2 { 3 if (narg != 3) error->all(FLERR,"Incorrect args for 4 bond coefficients"); 5 if (!allocated) allocate(); 6 7 int ilo,ihi; 8 force->bounds(arg[0],atom->nbondtypes,ilo,ihi); 9 10 double k_one = force->numeric(FLERR,arg[1]); 11 double r0_one = force->numeric(FLERR,arg[2]); 12 13 int count = 0; 14 for (int i = ilo; i <= ihi; i++) { 15 k[i] = k_one; 16 r0[i] = r0_one; 17 setflag[i] = 1; 18 count++; 19 } 20 21 if (count == 0) error->all(FLERR,"Incorrect args for 22 bond coefficients"); 23 } Listing 29: Modified coeff (bond_kv.cpp) 1 void BondKv::coeff(int narg, char **arg) 2 { 3 if (narg != 4) error->all(FLERR,"Incorrect args for 4 bond coefficients"); 5 if (!allocated) allocate(); 6 7 int ilo,ihi; 8 force->bounds(arg[0],atom->nbondtypes,ilo,ihi); 9 10 double k_one = force->numeric(FLERR,arg[1]); 11 double r0_one = force->numeric(FLERR,arg[2]); 12 double b_one = force->numeric(FLERR,arg[3]); 13 // to allocate in b_one the 3rd argument of bond_coeff 14 int count = 0; 15 for (int i = ilo; i <= ihi; i++) { 16 k[i] = k_one; 17 r0[i] = r0_one; 18 b[i] = b_one; 19 // to allocate the value stored in b_one used in compute 20 setflag[i] = 1; 21 count++; 22 } 23 24 if (count == 0) error->all(FLERR,"Incorrect args for 25 bond coefficients"); 26 } This pair style also has the write_restart and read_restart functions that have to be modified. They basically, write and read geometry file that can be used as a support file in the input file. Listing 30: Original write_restart and read_restart (bond_harmonic.cpp) 1 void BondHarmonic::write_restart(FILE *fp) 2 { 3 fwrite(&k[1],sizeof(double),atom->nbondtypes,fp); 4 fwrite(&r0[1],sizeof(double),atom->nbondtypes,fp); 5 } 6 /*------------*/ 7 void BondHarmonic::read_restart(FILE *fp) 8 { 9 allocate(); 10 11 if (comm->me == 0) { 12 fread(&k[1],sizeof(double),atom->nbondtypes,fp); 13 fread(&r0[1],sizeof(double),atom->nbondtypes,fp); 14 } 15 MPI_Bcast(&k[1],atom->nbondtypes,MPI_DOUBLE,0,world); 16 MPI_Bcast(&r0[1],atom->nbondtypes,MPI_DOUBLE,0,world); 17 18 for (int i = 1; i <= atom->nbondtypes; i++) 19 setflag[i] = 1; 20 } Listing 31: Modified write_restart and read_restart (bond_kv.cpp) 1 void BondKv::write_restart(FILE *fp) 2 { 3 fwrite(&k[1],sizeof(double),atom->nbondtypes,fp); 4 fwrite(&r0[1],sizeof(double),atom->nbondtypes,fp); 5 fwrite(&b[1],sizeof(double),atom->nbondtypes,fp); 6 } 7 /*------------*/ 8 void BondKv::read_restart(FILE *fp) { allocate(); 12 if (comm->me == 0) { 13 fread(&k[1],sizeof(double),atom->nbondtypes,fp); 14 fread(&r0[1],sizeof(double),atom->nbondtypes,fp); 15 fread(&b[1],sizeof(double),atom-

>nbondtypes,fp); 16 } 17 MPI_Bcast(&k[1],atom->nbondtypes,MPI_DOUBLE,0,world); 18 MPI_Bcast(&r0[1],atom->nbondtypes,MPI_DOUBLE,0,world); 19 MPI_Bcast(&n[1],atom->nbondtypes,MPI_DOUBLE,0,world); 20 21 22 for (int i = 1; i <= atom->nbondtypes; i++) 23 setflag[i] = 1; 24 } 3.3. bond_kv.h In the header of the new pair style we need to substitute the "BondHarmonic" text by a new "BondKv" text as well as declare a new protected member in the class, the pointer to b. Listing 32: Original header (bond_harmonic.h) 1 #ifdef BOND_CLASS 2 3 BondStyle(harmonic,BondHarmonic) 4 5 #else 6 7 #ifndef LMP_BOND_HARMONIC_H 8 #define LMP_BOND_HARMONIC_H 9 10 #include "stdio.h" 11 #include "bond.h" 12 13 namespace LAMMPS_NS { 14 15 class BondHarmonic : public Bond { 16 public: 17 BondHarmonic(class LAMMPS *); 18 virtual ~BondHarmonic(); 19 virtual void compute(int, int); 20 void coeff(int, char **); 21 double equilibrium_distance(int); 22 void write_restart(FILE *); 23 void read_restart(FILE *); 24 void write_data(FILE *); 25 double single(int, double, int, int, double &); 26 27 protected: 28 double *k,*r0; 29 30 void allocate(); 31 }; 32 } 33 #endif 34 #endif Listing 33: Modified header (bond_kv.h) 1 #ifdef BOND_CLASS 2 3 BondStyle(kv,BondKv) 4 5 #else 6 7 #ifndef LMP_BOND_KV_H #define LMP_BOND_KV_H #include "stdio.h" 11 #include "bond.h" 12 13 namespace LAMMPS_NS { 14 15 class BondKv : public Bond { 16 public: 17 BondKv(class LAMMPS *); 18 virtual ~BondKv(); 19 virtual void compute(int, int); 20 void coeff(int, char **); 21 double equilibrium_distance(int); 22 void write_restart(FILE *); 23 void read_restart(FILE *); 24 void write_data(FILE *); 25 double single(int, double, int, int, double &); 26 27 protected: 28 double *k,*r0, *b; // new pointer 29 30 void allocate(); 31 }; 32 } 33 #endif 34 #endif 3.4. Invoking kv Pair Style Now the new pair style is completed. To run LAMMPS with the new style we need to compile it and then invoke it by writing the command lines in shown in Listing 34 in the input file. Listing 34: Command lines to invoke the kv pair style 1 bond_style kv 2 bond_coeff K r0 b 4. Noble–Abel Stiffened-Gas Pair Style In the SPH framework is possible to determine all the particles properties by solving the particle form of the continuity equation [6,26] ddpti = ∑ mjvij∇jWij; (4) j the momentum equation [6,26] mi ddvti = ∑ mimj Pρii Pρii + Πij ∇jWij; + (5) j ( ) and the energy conservation equation [6,26] mimj (κi + κj)(Ti − Tj) mi dt dei = 1 2 ∑j mimj(Pi Pi ρi ρi + Πij : vij∇jWij − ∑ ρiρj + ri2j rij · ∇jWij. (6)) j However, to be able to solve this set of equations an Equation of State (EOS) linking the pressure P and the density ρ is needed [46]. In the user-SPH package of LAMMPS one EOS is used for the liquid (Tait's EOS) and one for gas phase (ideal gas EOS). In this section we will implement a new EOS for the liquid phase. Note that with similar steps is also possible to implement a new gas EOS. ChemEngineering 2021, 1, 1 20 of 61 ChemEngineering 2021, 5, 30 we will implement a new EOS for the liquid phase. Note that with similar steps is also possible to imLpeleMméetanytear naenwd gSaausrEeOl[S4.7] combined the "Noble–Abel" and the "Stiffened-Gas" EOS Le pMréotpaoyseirn&g aSanuerwelE[4O7S] ccoamllebdinNedobthlee–"ANboebllSet-iAffebnele"da-Gndasth(NeA"SStGiff)e,nseudit-aGbales"foErOmSultiphase proposinflgowa.neTwheEeOxSprceasllseiodnNoofbtlhee-AEbOelSSdtioffeesnnedo-tGchasan(NgeAwSGit)h, stuhietapbhleasfoercmonuslitdipehreadse. For each flow. Thpehaexsepsr,etshsieopnroefsstuhreeEaOnSd dteomespneorattcuhraenagree wcailtchutlhaetepdhaassefucnocntsioidneroefdd.eFnosriteyaacnhd specific phases, itnhteerpnraelsseunreergayn,de.tge.m,perature are calculated as function of density and specific internal energy, e.g., P(ρ, e) =P((γρ,−e)1=) ((eγ− q1)) −1γ−P∞b, − γP∞, (e − q) 1ρ − b ρ) and temperature-wise ( ) (and temperature wise T(ρ, e) =T(eρC−,veq) = e C1ρ−v−q b−)(PC∞ρv ,− b) PC∞v 1 − , ((7) (8) (7) (8) where Pw,ρh,eer,eaPn,dρq, ea,rae,nrdesqpaerceti,vreelsyp,ethcteivperleys,stuhree,ptrheessduernes,itthy,etdheensspietyc,ifithceinstpeernciafilceinnetregryn,al energy, and the ahneadttbhoenhdeoatf bthoencdoorrfetshpeocnodrirnegsppohnadsein.gγ,pPh∞a,seq., aγn,dP∞b ,aqre,acnodnsbtaanrte ccooenffistcainenttcsotehfafitcients that defines tdheefitnheesrmthoedtyhnearmmoicdpyrnoapmeritciepsroopftehrteieflsuoidf.the fluid. For this nFeowrtphiasirnsetwylep,aciarlsletydlesp,cha/lnleadsgslpiqhu/inda,swgeliqtaukied,awsae rteafkeereanscae rtehfeesrepnhc/etathitewsaptehr/taitwater pair stylpeadirecsltayrleeddaencdlarineditiaanlidseidnitiaplaisier_dspinhp_taaiirt_wspathe_r.thaiatwndatpear.ihr_asnpdh_ptaaiitrw_saptehr_.ctapiptwfialetesr.cpp files in the diirnectthoerydi/resrcct/orUyS/EsRr-cS/PUHS.EARl-lStPhHefi.Alelsl rtehgeafirdleinsgresgpahr/dninagsgslpiqhu/indamsgulsiqtubiedsmavuesdt ibne saved in the /src/thUeS/EsRrc-S/PUHSEdRir-eScPtoHryd.irectory and its hierarchy is shown in Figure 4.. LAMMPS Force Pair Figure 4. Class hierarchy of the new bond style. 4.1. Validation The sph/nasgliquid pair style has validated by Albano and Alexiadis [26] to study the Rayleigh collapse of an empty cavity. 4.2. pair_sph_nasgliquid.cpp All the functions will be the same as in the reference sph/taitwater. However, in our new sph/nasgliquid, we need to substitute the "PairSPHTaitwater" text in "PairSPHNas- gliquid", as can be seen in Listings 35 and 36. Listing 35: Original script (pair_sph_taitwater.cpp) 1 #include 2 #include 3 #include "pair_sph_taitwater.h" 4 #include "atom.h" 5 #include "force.h" 6 #include "comm.h" 7 #include "neigh_list.h" 8 # include "memory.h" 9 #include "error.h" 10 #include "domain.h" 11 12 using namespace LAMMPS_NS; 13 14 PairSPHTaitwater::PairSPHTaitwater(LAMMPS *lmp) : 15 Pair(lmp) 16 {...} 17 PairSPHTaitwater::~PairSPHTaitwater() 18 {...} void PairSPHTaitwater::compute(int eflag, int vflag) {...} void PairSPHTaitwater::allocate() 22 {...} 23 void PairSPHTaitwater::settings(int narg, char **/*arg*/) 24 {...} 25 void PairSPHTaitwater::coeff(int narg, char **arg) 26 {...} 27 double PairSPHTaitwater::init_one(int i, int j) 28 {...} Listing 36: Modified script (pair_sph_nasgliquid.cpp) 1 #include 2 #include 3 #include "pair_sph_nasgliquid.h" 4 #include "atom.h" 5 #include "force.h" 6 #include "comm.h" 7 # include "neigh_list.h" 8 #include "memory.h" 9 #include "error.h" 10 #include "domain.h" 11 12 using namespace LAMMPS_NS; 13 14 PairSPHNasgliquid:: PairSPHNasgliquid(LAMMPS *lmp) : 15 Pair(lmp) 16 {...} 17 PairSPHNasgliquid::~ PairSPHNasgliquid() 18 {...} 19 void PairSPHNasgliquid::compute(int eflag, int vflag) 20 {...} 21 void PairSPHNasgliquid::allocate() 22 {...} 23 void PairSPHNasgliquid::settings(int narg, char **/*arg*/) 24 {...} 25 void PairSPHNasgliquid::coeff(int narg, char **arg) 26 {...} 27 double PairSPHNasgliquid::init_one(int i, int j) 28 {...} For the sph/nasgliquid we need to pass a total of 12 arguments from the input file, while they were only six for sph/taitwater. For this reason we need to modify destructor, compute, allocate, settings and coeff. Following the order of function initialisation, see Listing 36, the destructor is modified as shown in Listing 38. Listing 37: Original destructor (pair_sph_taitwater.cpp) 1 PairSPHTaitwater::~PairSPHTaitwater() { 2 if (allocated) { 3 memory->destroy(setflag); 4 memory->destroy(cutsq); 5 memory->destroy(cut); 6 memory->destroy(rho0); 7 memory->destroy(soundspeed); 8 memory->destroy(B); 9 memory->destroy(viscosity); 10 } 11 } Listing 38: Modified destructor (pair_sph_nasgliquid.cpp) 1 PairSPHNasgliquid::~PairSPHNasgliquid() { 2 if (allocated) { 3 memory->destroy(setflag); 4 memory->destroy(cutsq); 5 memory->destroy(cut); memory->destroy(soundspeed); memory->destroy(B); memory->destroy(CP); 9 memory->destroy(CV); 10 memory->destroy(gamma); 11 memory->destroy(P00); 12 memory->destroy(b); 13 memory->destroy(q); 14 memory->destroy(q1); 15 memory->destroy(viscosity); 16 } 17 } In the NASG EOS the pressure is function of both density, ρ, and internal energy, e. For this reason, we need to declare more pointers and variables in compute compared to the reference pair style, see line 6 and 20 in Listing 40. Listing 39: Original compute (pair_sph_taitwater.cpp) 1 void PairSPHTaitwater::compute(int eflag, int vflag) { 2 int i, j, ii, jj, inum, jnum, itype, jtype; 3 double xtmp, ytmp, ztmp, delx, dely, delz, fpair; 4 5 int *ilist, *jlist, *numneigh, **firstneigh; 6 double vxtmp, vytmp, vztmp, imass, jmass, 7 fi, fj, fvisc, h, ih, ihsq; 8 double rsq, tmp, wfd, delVdotDelR, mu, deltaE; 9 10 if (eflag || vflag) 11 ev_setup(eflag, vflag); 12 else 13 evflag = vflag_fdotr = 0; 14 15 double **v = atom->vest; 16 double **x = atom->x; 17 double **f = atom->f; 18 double *rho = atom->rho; 19 double *mass = atom->mass; 20 double *de = atom->de; 21 double *drho = atom->drho; 22 int *type = atom->type; 23 int nlocal = atom->nlocal; 24 int newton_pair = force->newton_pair; Listing 40: Modified compute (pair_sph_nasgliquid.cpp) 1 void PairSPHNasgliquid::compute(int eflag, int vflag) { 2 int i, j, ii, jj, inum, jnum, itype, jtype; 3 double xtmp, ytmp, ztmp, delx, dely, delz, fpair; 4 5 int *ilist, *jlist, *numneigh, **firstneigh; 6 double vxtmp, vytmp, vztmp, imass, jmass, 7 fi, fj, fvisc, h, ih, ihsq, iirho, ijrho; 8 double rsq, tmp, wfd, delVdotDelR, mu, deltaE; 9 10 if (eflag || vflag) 11 ev_setup(eflag, vflag); 12 else 13 evflag = vflag_fdotr = 0; 14 15 double **v = atom->vest; 16 double **x = atom->x; 17 double **f = atom->f; 18 double *rho = atom->rho; 19 double *mass = atom->mass; 20 double *de = atom->de; 21 double *e = atom->e; 22 double *drho = atom->drho; 23 int *type = atom->type; int nlocal = atom->nlocal; int newton_pair = force->newton_pair; Another modification for compute regards the expression of the force applied to the i-th, see Listing 42, and j-th, see Listing 44, particle. Listing 41: Original compute (pair_sph_taitwater.cpp) 1 // compute pressure of atom i with Tait EOS 2 tmp = rho[i]/rho0[itype]; 3 fi = tmp * tmp * tmp; 4 fi = B[itype] * (fi * fi * tmp - 1.0)/ (rho[i] * rho[i]); Listing 42: Modified compute (pair_sph_nasgliquid.cpp) 1 // compute pressure of atom i with NASG EOS 2 tmp = e[i] / imass; 3 iirho= 1.0/rho[i]; 4 iirho= iirho - b[itype]; 5 fi = ((tmp - q[itype]) * B[itype] / iirho); 6 fi = fi - gamma[itype] * P00[itype]; 7 fi = fi / (rho[i] * rho[i]); Listing 43: Original compute (pair_sph_taitwater.cpp) 1 // compute pressure of atom j with Tait EOS 2 tmp = rho[j] / rho0[jtype]; 3 fj = tmp * tmp * tmp; 4 fj = B[jtype] * (fj * fj * tmp - 1.0) / (rho[j] * rho[j]); Listing 44: Modified compute (pair_sph_nasgliquid.cpp) 1 // compute pressure of atom j with NASG EOS 2 tmp = e[j] / jmass; 3 ijrho= 1/rho[j]; 4 ijrho= ijrho - b[jtype]; 5 fj = ((tmp - q[jtype])* B[jtype]/ijrho); 6 fj = fj - gamma[jtype]*P00[jtype]; 7 fj = fj / (rho[j] * rho[j]); With the introduction of a new parameter in the pair style we need

to make a new dynamic memory allocation by modifying allocate. Listing 45: Original allocate (pair_sph_taitwater.pp) 1 void PairSPHTaitwater::allocate() { 2 allocated = 1; 3 int n = atom->ntypes; 4 5 memory->create(setflag, n + 1, n + 1, "pair:setflag"); 6 for (int i = 1; i <= n; i++) 7 for (int j = i; j <= n; j++) 8 setflag[i][j] = 0; 9 10 memory->create(cutsq, n + 1, n + 1, "pair:cutsq"); 11 memory->create(rho0, n + 1, "pair:rho0"); 12 memory->create(soundspeed, n + 1, "pair:soundspeed"); 13 memory->create(B, n + 1, "pair:B"); 14 memory->create(cut, n + 1, n + 1, "pair:cut"); 15 memory->create(viscosity,n + 1,n + 1,"pair:viscosity"); 16 } Listing 46: Modified allocate (pair_sph_nasgliquid.cpp) 1 void PairSPHNasgliquid::allocate() { 2 allocated = 1; 3 int n = atom->ntypes; 4 5 memory->create(setflag, n + 1, n + 1, "pair:setflag"); for (int i = 1; i <= n; i++) for (int j = i; j <= n; j++) setflag[i][j] = 0; 9 10 memory->create(cutsq, n + 1, n + 1, "pair:cutsq"); 11 memory->create(soundspeed, n + 1, "pair:soundspeed"); 12 memory->create(B, n + 1, "pair:B"); 13 memory->create(CP, n + 1, "pair:CP"); 14 memory->create(CV, n + 1, "pair:CV"); 15 memory->create(gamma, n + 1, "pair:gamma"); 16 memory->create(P00, n + 1, "pair:P00"); 17 memory->create(b, n + 1, "pair:b"); 18 memory->create(q, n + 1, "pair:q"); 19 memory->create(q1, n + 1, "pair:q1"); 20 memory->create(cut, n + 1, n + 1, "pair:cut"); 21 memory->create(viscosity,n + 1,n + 1,"pair:viscosity"); 22 } The 12 arguments used in the pair style are passed by the used in the input file. For this reason, we also have to modify coeff. Listing 47: Original coeff (pair_sph_taitwater.cpp) 1 void PairSPHTaitwater::coeff(int narg, char **arg) { 2 if (narg != 6) 3 error->all(FLERR, 4 "Incorrect args for pair_style sph/taitwater 5 coefficients"); 6 if (!allocated) 7 allocate(); 8 int ilo, ihi, jlo, jhi; 9 force->bounds(FLERR,arg[0], atom->ntypes, ilo, ihi); 10 force->bounds(FLERR,arg[1], atom->ntypes, jlo, jhi); 11 double rho0_one = force->numeric(FLERR,arg[2]); 12 double soundspeed_one = force->numeric(FLERR,arg[3]); 13 double viscosity_one = force->numeric(FLERR,arg[4]); 14 double cut_one = force->numeric(FLERR,arg[5]); 15 double B_one=soundspeed_one*soundspeed_one*rho0_one/7.0; 16 int count = 0; 17 for (int i = ilo; i <= ihi; i++) { 18 rho0[i] = rho0_one; 19 soundspeed[i] = soundspeed_one; 20 B[i] = B_one; 21 for (int j = MAX(jlo,i); j <= jhi; j++) { 22 viscosity[i][j] = viscosity_one; 23 cut[i][j] = cut_one; 24 setflag[i][j] = 1; 25 count++; } } Listing 48: Modified coeff (pair_sph_nasgliquid.cpp) 1 void PairSPHNasgliquid::coeff(int narg, char **arg) { 2 if (narg != 12) 3 error->all(FLERR, 4 "Incorrect args for pair_style sph/nasgliquid 5 coefficients"); 6 if (!allocated) 7 allocate(); 8 int ilo, ihi, jlo, jhi; 9 force->bounds(FLERR,arg[0], atom->ntypes, ilo, ihi); 10 force->bounds(FLERR,arg[1], atom->ntypes, jlo, jhi); 11 double soundspeed_one = force->numeric(FLERR,arg[2]); 12 double viscosity_one = force->numeric(FLERR,arg[3]); 13 double cut_one = force->numeric(FLERR,arg[4]); 14 double CP_one = force->numeric(FLERR,arg[5]); 15 double CV_one = force->numeric(FLERR,arg[6]); 16 double gamma_one = force->numeric(FLERR,arg[7]); 17 double P00_one = force->numeric(FLERR,arg[8]); 18 double b_one = force->numeric(FLERR,arg[9]); double q_one = force->numeric(FLERR,arg[10]); double q1_one = force->numeric(FLERR,arg[11]); double B_one = (gamma_one - 1); 22 int count = 0; 23 for (int i = ilo; i <= ihi; i++) { 24 soundspeed[i] = soundspeed_one; 25 B[i] = B_one; 26 CP[i] = CP_one; 27 CV[i] = CV_one; 28 gamma[i] = gamma_one; 29 P00[i] = P00_one; 30 b[i] = b_one; 31 q[i] = q_one; 32 q1[i] = q1_one; 33 for (int j = MAX(jlo,i); j <= jhi; j++) { 34 viscosity[i][j] = viscosity_one; 35 cut[i][j] = cut_one; 36 setflag[i][j] = 1; 37 count++; } } 4.3. pair_sph_nasgliquid.h In the header of the new pair style we need to substitute the "PairSPHTaitwater " text in "PairSPHNasgliquid" as well as declare new protected members in the class, the pointers to the new arguments. Listing 49: Original header (pair_sph_taitwater.h) 1 #ifdef PAIR_CLASS 2 3 PairStyle(sph/taitwater,PairSPHTaitwater) 4 5 #else 6 7 #ifndef LMP_PAIR_TAITWATER_H 8 #define LMP_PAIR_TAITWATER_H 9 10 #include "pair.h" 11 12 namespace LAMMPS_NS { 13 14 class PairSPHTaitwater : public Pair { 15 public: 16 PairSPHTaitwater(class LAMMPS *); 17 virtual ~PairSPHTaitwater(); 18 virtual void compute(int, int); 19 void settings(int, char **); 20 void coeff(int, char **); 21 virtual double init_one(int, int); 22 23 protected: 24 double *rho0, *soundspeed, *B; 25 double **cut,**viscosity; 26 int first; 27 void allocate(); 28 }; 29 } 30 #endif 31 #endif Listing 50: Modified header (pair_sph_nasgliquid.h) 1 #ifdef PAIR_CLASS 2 3 PairStyle(sph/nasgliquid,PairSPHNasgliquid) 4 5 #else 6 #ifndef LMP_PAIR_NASGLIQUID_H #define LMP_PAIR_NASGLIQUID_H 10 #include "pair.h" 11 12 namespace LAMMPS_NS { 13 14 class PairSPHNasgliquid : public Pair { 15 public: 16 PairSPHNasgliquid(class LAMMPS *); 17 virtual ~PairSPHNasgliquid(); 18 virtual void compute(int, int); 19 void settings(int, char **); 20 void coeff(int, char **); 21 virtual double init_one(int, int); 22 23 protected: 24 double *soundspeed, *B, *CP, *CV, *gamma, *P00, 25 *b, *q, *q1; 26 double **cut,**viscosity; 27 int first; 28 void allocate(); 29 }; 30 } 31 #endif 32 #endif 4.4. Invoking Sph/Nasgliquid Pair Style Now the new pair style is completed. To run LAMMPS with the new style we need to compile it and then invoke it by writing the command lines shown in Listing 51 in the input file. Listing 51: Command lines to invoke the NASG pair style for liquid 1 pair_style sph/nasgliquid 2 pair_coeff I J c_0 alpha h Cv Cp gamma P00 b q q' 5. Multiphase (Liquid–Gas) Heat Exchange Pair Style In LAMMPS thermal conductivity between SPH particles is enabled using the sph/heat- conduction pair style inside the user-SPH package. However, the pair style is designed only for mono phase fluid where the thermal conductivities is constant (κi = κ). When more than one phase is present, the heat conduction at the interface can be implemented by using [6,26] mi ddeti = ∑ mimj (κi + κj)(Ti − Tj) ρi ρj ri2j rij · ∇jWij. (9) j In the new pair style, called sph/heatgasliquid, one phase is assumed to be liquid with an initial temperature of Tl,0 and the other is assumed to be and ideal gas. Each time-step the temperature of the fluid is updated as [26]. Tl = Tl,0 + El − El,0 , (10) Cp,l where Tl,0 is the reference temperature, E0 the internal energy in [J], El internal energy [J] at the current time step and Cp,l is heat capacity of the fluid in [J K−1]. The temperature of the gas is updated following the ideal EOS [26]. Tg = M M (γ − 1)eg R , (11) ChemEngineering 2021, 1, 1 27 of 61 where Tl,0 is the reference temperature, E0 the internal energy in [J], El internal energy [J] at the current time step and Cp,l is heat capacity of the fluid in [J K-1]. The temperature of the gas is updated following the ideal EOS [26] where MM is the molarTmg=assM[Mkg(γkm−Ro1)l−eg1,], eg is the specific internal en(e1r1g)y in [J kg−1], wthheehreethaMtecMcahpaiosciitctheyeromaftiotohlaaernrmdefaResrsise[nkthgceekisdmteaoatll-e1gs]a,Esegcl,o0isnistshtaaenrstbpiinetrc[aiJfirKcy-,i1nbktuemtrnoilaf-l1t]he.neGeEergnqyeuriaanltli[yJotknhgeo-1fc]h,Soγtiacitsee (EOS) used γ is the heat capacity ratio and R is the ideal gas constant in [J K−1 kmol−1]. Generally of theforerftehreenpcehsatsaeteissEful,0nicstaiorbnitorafrby,obthutdifenthseiteyquanatdioinnotefrsntaatlee(nEOerSg)yusoefdthfoerrtehfeeprehnacsee state will be is fundcteitoenrmofibnoetdh bdyentshiteyEaOndSi.nternal energy of the reference state will be determined by the EOS. In the sph/heatgasliquid pair style is important to check if the i-th and j-th particles IanrethleiqsupihdohreatgasplihqausiedtpoaairpsptylyleeiisthimerpEorqtuanattitoonc(h1e0c)koirftEhqeui-atthioannd(1j1-)th.Tphairsti‟clpehsase check" is are liqexupidlaoirnegdasinphSaescetitoona5p.p2lyceoimthpeurtEeqfuuanticotnio(n10i)somroEdqiufiaetdio.n (11). This "phase check‟ is explaineFdoinrtSheectieonne5r.g2ycobmaplautnecefutnhcetionnewismpoadirifisetdy.le needs Tl,0, El,0, Cp,l and κl for the liquid Fpohratsheeaenndergygfboarlatnhceegtahsepnhewaspe.aiMrsotryeleovneere,dfsoTrlt,0h,eEpl,0h,aCspe,lcahnedckκ,l ftohretphaerltiiqculeidtypes of each phasepahnadseκsgmfourstthbeegasspp echiafiseed..MAolrletohviesr,infortmheaptihoansseicshpecakss,ethdebpyartthiceleustyepreisnotfheeaicnhthe input file. phases musTthbeersepfeecriefinedce.Aplalitrhisstiynlfeorimsastpiohn/shiesaptacsosneddubcytitohne.usIetriisndtheeclinartehde iannpdutifinlieti.alised in the Tphaeirr_esfpehre_nhceeaptcaoinrdstuycletioisns.cpphp/hpeaaitrc_osnpdhu_hcteioatnc.oIntdisudcteicolnar.cepdpanfidlesiniintiathlieseddiriencttohrey /src/USER-pSPaiHr_.sSpAPhlHl_h.thAeaeltlcfitolhenesdfurieclegtisaorrnde.gcinpagprdpsipanihgr/_sshppehha/_tghhaeesaaltticgqoaunsidldiuqmcutiiudosntm.cbpuepstsfiablveeessdianvitnehdethdineirt/ehcsetroc/r/ysUr/cSs/Er USEPSREH-RSP-H directory directaonrdy.its hierarchy is shown in Figure 5. LAMMPS Force Pair Figure 5. Class hierarchy of the new pair style. 5.1. Validation The sph/heatgasliquid pair style has validated by Albano and Alexiadis [26] to study the role of the heat diffusion in for a gas filled Rayleigh collapse in water. 5.2. pair_sph_heatgasliquid.cpp All the functions will be the same as in the reference sph/heatconduction. However, in our new sph/heatgasliquid, we need to substitute the "PairSPHHeatConduction" text in "PairSPHHeatgasliquid", as can be seen in Listings 52 and 53. Listing 52: Original script (pair_sph_heatconduction.cpp) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 #include "math.h" #include "stdlib.h" #include "pair_sph_heatconduction.h" #include "atom.h" #include "force.h" #include "comm.h" #include "memory.h" #include "error.h" #include "neigh_list.h" #include "domain.h" using namespace LAMMPS_NS; PairSPHHeatConduction::PairSPHHeatConduction(LAMMPS *lmp) : Pair(lmp) { ... } PairSPHHeatConduction::~PairSPHHeatConduction() { ... } void PairSPHHeatConduction::compute(int eflag, int vflag) { ... } void PairSPHHeatConduction::allocate() { ... } void PairSPHHeatConduction::settings(int narg, char **arg) { ... } void PairSPHHeatConduction::coeff(int narg, char **arg) { ... } double PairSPHHeatConduction::init_one(int i, int j) { ... } double PairSPHHeatConduction::single(int i, int j, int itype, int jtype, double rsq, double factor_coul, double factor_lj, double &fforce) 32 { ... } Listing 53: Modified script (pair_sph_heatgasliquid.cpp) 1 #include 2 #include 3 #include "pair_sph_heatgasliquid.h" 4 #include "atom.h" 5 #include "force.h" 6 #include "comm.h" 7 #include "memory.h" 8 #include "error.h" 9 #include "neigh_list.h" 10 #include "domain.h" 11 12 using namespace LAMMPS_NS; 13 14 PairSPHHeatgasliquid::PairSPHHeatgasliquid(LAMMPS *lmp) 15 :

Pair(lmp) 16 { ... } 17 PairSPHHeatgasliquid::~PairSPHHeatgasliquid() 18 { ... } 19 void PairSPHHeatgasliquid::compute(int eflag, int vflag) 20 { ... } 21 void PairSPHHeatgasliquid::allocate() 22 { ... } 23 void PairSPHHeatgasliquid::settings(int narg, char **arg) 24 { ... } 25 void PairSPHHeatgasliquid::coeff(int narg, char **arg) 26 { ... 27 double PairSPHHeatgasliquid::init_one(int i, int j) 28 { ... } 29 double PairSPHHeatgasliquid::single(int i, int j, 30 int itype, int jtype, double rsq, double factor_coul, 31 double factor_lj, double &fforce) 32 { ... } For the sph/heatgasliquid we need to pass a total of nine arguments from the input file, while they were only seven for sph/heatconduction. For this reason we need to modify destructor, compute, allocate, settings and coeff. Following the order of function initialisation, see Listing 53, the destructor is modified by removing the heat diffusion coefficient, line 6 in Listing 54. Listing 54: Original destructor (pair_sph_heatconduction.cpp) 1 PairSPHHeatConduction::~PairSPHHeatConduction() { 2 if (allocated) { 3 memory->destroy(setflag); 4 memory->destroy(cutsq); 5 memory->destroy(cut); 6 memory->destroy(alpha); 7 } 8 } Listing 55: Modified destructor (pair_sph_heatgasliquid.cpp) 1 PairSPHHeatgasliquid::~PairSPHHeatgasliquid() { 2 if (allocated) { 3 memory->destroy(setflag); 4 memory->destroy(cutsq); 5 memory->destroy(cut); 6 } 7 } To compute Equation (6) we need to declare more variables in compute compared to the reference pair style, see line 4 in Listing 57. Listing 56: Original compute (pair_sph_heatconduction.cpp) 1 void PairSPHHeatConduction::compute(int eflag, int vflag){ 2 int i, j, ii, jj, inum, jnum, itype, jtype; 3 double xtmp, ytmp, ztmp, delx, dely, delz; Listing 57: Modified compute (pair_sph_heatgasliquid.cpp) 1 void PairSPHHeatgasliquid::compute(int eflag, int vflag){ 2 int i, j, ii, jj, inum, jnum, itype, jtype; 3 double xtmp, ytmp, ztmp, delx, dely, delz; 4 double Ti, Tj, ki, kj; /// new parameters Another important modification is to add the phase check inside compute. The phase check has to be implemented for both the i-th particle and the j-th particle inside the loop over neighbours, for (ii = 0; ii < inum; ii++) in the reference pair style. The phase check for the i-th particle starts after the assignment of imass, line 3 of Listing 58. Listing 58: Modified compute (pair_sph_heatgasliquid.cpp) 1 imass = mass[itype]; 2 3 if (itype == liquidtype) 4 { 5 Ti= e[i] - el0; 6 Ti= Ti/CPl; 7 Ti= T0l + Ti; 8 ki=kl; 9 } 10 else { 11 Ti=0.40*e[i]*18; 12 Ti= Ti/imass; 13 Ti= Ti/8314.33; 14 ki=kg; 15 } Similarly, for the j-th the phase check start at line 3 of Listing 59. Listing 59: Modified compute (pair_sph_heatgasliquid.cpp) 1 jmass = mass[jtype]; 2 3 if (jtype == liquidtype) 4 { 5 Tj= e[j] - el0; 6 Tj= Tj/CPl; 7 Tj= T0l + Tj; 8 kj=kl; 9 } 10 else { 11 Tj=0.40*e[j]*18; 12 Tj= Tj/jmass; 13 Tj= Tj/8314.33; 14 kj=kg; 15 } 16 The last change in compute is to implement the change in internal energy as shown in Equation (9). Listing 60: Original compute (pair_sph_heatconduction.cpp) 1 D = alpha[itype][jtype]; // diffusion coefficient 2 3 deltaE = 2.0 * imass * jmass / (imass+jmass); 4 deltaE *= (rho[i] + rho[j]) / (rho[i] * rho[j]); 5 deltaE *= D * (e[i] - e[j]) * wfd; 6 7 de[i] += deltaE; 8 if (newton_pair || j < nlocal) { 9 de[j] -= deltaE; 10 } Listing 61: Modified compute (pair_sph_heatgasliquid.cpp) 1 deltaE = imass * jmass / (rho[i] * rho[j]); /// 2 deltaE *= (ki + kj) * (Ti - Tj) * wfd; /// 3 /// implementation of eq 3.4 4 de[i] += deltaE; 5 if (newton_pair || j < nlocal) { 6 de[j] -= deltaE; 7 } With the introduction of new arguments in the pair style we need to make a new dynamic memory allocation by modifying allocate. Listing 62: Original allocate (pair_sph_heatconduction.cpp) 1 void PairSPHHeatConduction::allocate() { 2 allocated = 1; 3 int n = atom->ntypes; 4 5 memory->create(setflag, n + 1, n + 1, "pair:setflag"); 6 for (int i = 1; i <= n; i++) 7 for (int j = i; j <= n; j++) 8 setflag[i][j] = 0; 9 10 memory->create(cutsq, n + 1, n + 1, "pair:cutsq"); 11 memory->create(cut, n + 1, n + 1, "pair:cut"); 12 memory->create(alpha, n + 1, n + 1, "pair:alpha"); 13 } Listing 63: Modified allocate (pair_sph_heatgasliquid.cpp) 1 void PairSPHHeatgasliquid::allocate() { 2 allocated = 1; 3 int n = atom->ntypes; 4 5 memory->create(setflag, n + 1, n + 1, "pair:setflag"); 6 for (int i = 1; i <= n; i++) 7 for (int j = i; j <= n; j++) 8 setflag[i][j] = 0; 9 10 memory->create(cutsq, n + 1, n + 1, "pair:cutsq"); 11 memory->create(cut, n + 1, n + 1, "pair:cut"); 12 } The nine arguments used in the pair style are passed by the user in the input file. For this reason, we also have to modify coeff. Listing 64: Original coeff (pair_sph_heatconduction.cpp) 1 void PairSPHHeatConduction::coeff(int narg, char **arg) { 2 if (narg != 4) 3 error->all(FLERR,"Incorrect number of args for 4 pair_style sph/heatconduction coefficients"); if (!allocated) allocate(); 8 9 10 11 int ilo, ihi, jlo, jhi; force->bounds(arg[0], atom->ntypes, ilo, ihi); force->bounds(arg[1], atom->ntypes, jlo, jhi); 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 double alpha_one = force->numeric(FLERR,arg[2]); double cut_one = force->numeric(FLERR,arg[3]); int count = 0; for (int i = ilo; i <= ihi; i++) { for (int j = MAX(jlo,i); j <= jhi; j++) { //printf("setting cut[%d][%d] = %f\n", i, j, cut_one); } cut[i][j] = cut_one; alpha[i][j] = alpha_one; setflag[i][j] = 1; count++; } } if (count == 0) error->all(FLERR,"Incorrect args for pair coefficients"); Listing 65: Modified coeff (pair_sph_heatgasliquid.cpp) 1 void PairSPHHeatgasliquid::coeff(int narg, char **arg) { 2 if (narg != 9) 3 error->all(FLERR,"Incorrect number of args for 4 pair_style sph/heatgasliquid coefficients"); 5 if (!allocated) 6 allocate(); 7 8 int ilo, ihi, jlo, jhi; 9 force->bounds(FLERR,arg[0], atom->ntypes, ilo, ihi); 10 force->bounds(FLERR,arg[1], atom->ntypes, jlo, jhi); 11 12 el0 = force->numeric(FLERR,arg[2]); 13 kl = force->numeric(FLERR,arg[3]); 14 kg = force->numeric(FLERR,arg[4]); 15 T0l = force->numeric(FLERR,arg[5]); 16 double cut_one = force->numeric(FLERR,arg[6]); 17 CPl = force->numeric(FLERR,arg[7]); 18 liquidtype = force->numeric(FLERR,arg[8]); 19 20 int count = 0; 21 for (int i = ilo; i <= ihi; i++) { 22 for (int j = MAX(jlo,i); j <= jhi; j++) { 23 //printf("setting cut[%d][%d] = %f\n", i, j, cut_one); 24 cut[i][j] = cut_one; 25 setflag[i][j] = 1; 26 count++; 27 } 28 } 29 if (count == 0) 30 error->all(FLERR,"Incorrect args for pair 31 coefficients"); 32 } 5.3. pair_sph_heatgasliquid.h In the header of the new pair style we need to substitute the "PairSPHHeatConduction" text in "PairSPHHeatgasliquid" and declare new protected members in the class. Listing 66: Original header (pair_sph_heatconduction.h) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 #ifdef PAIR_CLASS PairStyle(sph/heatconduction,PairSPHHeatConduction) #else #ifndef LMP_PAIR_SPH_HEATCONDUCTION_H #define LMP_PAIR_SPH_HEATCONDUCTION_H #include "pair.h" namespace LAMMPS_NS { class PairSPHHeatConduction : public Pair { public: PairSPHHeatConduction(class LAMMPS *); virtual ~PairSPHHeatConduction(); virtual void compute(int, int); void settings(int, char **); void coeff(int, char **); virtual double init_one(int, int); virtual double single(int, int, int, int, double, double, double, double &); protected: double **cut, **alpha; void allocate(); }; } #endif #endif Listing 67: Modified header (pair_sph_heatgasliquid.h) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 #ifdef PAIR_CLASS PairStyle(sph/heatgasliquid,PairSPHHeatgasliquid) #else #ifndef LMP_PAIR_SPH_HEATGASLIQUID_H #define LMP_PAIR_SPH_HEATGASLIQUID_H #include "pair.h" namespace LAMMPS_NS { class PairSPHHeatgasliquid : public Pair { public: PairSPHHeatgasliquid(class LAMMPS *); virtual ~PairSPHHeatgasliquid(); virtual void compute(int, int); void settings(int, char **); void coeff(int, char **); virtual double init_one(int, int); virtual double single(int, int, int, int, double, double, double, double &); protected: int liquidtype; double el0, kg, kl, T0l, CPl; double **cut; void allocate(); }; } #endif #endif 5.4. Invoking Sph/Heatgasliquid Pair Style Now the new pair style is completed. To run LAMMPS with the new style we need to compile it and then invoke it by writing the command lines shown in Listing 68 in the input file. Listing 68: Command lines to invoke the sph/heatgasliquid pair style 1 pair_style sph/heatgasliquid 2 pair_coeff i j el0 kl kg Tl0 h Cpl liquidtype ChemEngineering 2021, 1, 1 6. Full Stationary Fix Style 34 of 61 In LAMMPS a fix style is any operation that is applied to the system, usually to a 6. FullgSrtoautiponoafrpyaFritxicSletys,leduring time stepping or minimisation used to alter some property of InthLeAsMytMemPS[a41f]i.xTshtyelreeisaraenyhuonpedrraetdiosnotfhfiatxiessadpepfilinededtointhLeAsyMstMemP,Suasunadllnyetwoaones can be groupaodfdpaerdti.cUless,udaullryinfigxteimsaerseteupspeidngfoorrtmimineiministeagtiroantiuosne,dftoorcaeltecrosnosmtreaipnrtosp,ebrotyunofdtahrey conditions sytema[4n1d].dTiahgerneoasrteichs.undreds of fixes defined in LAMMPS and new ones can be added. Usually fixeIns athree uusseedr-sfoprhtpimacekiangteegraLtiAonM,fMorPcSe cthoenrsetriasinthtse, sboocuanldleadrymceosnod/istitoantisonanardy fix used to diagnossetticbso.undary condition. With meso/stationary is possible to fix position and velocity for Inatgherouusepr-osfpphapratcikclaegse, winaLlAlsMasMePxSatmheprelei,sbthuet sinotcearlnleadl meneesorg/ystaatniodnadreynfisxituyswediltlobe updated. set bouInndsaormyceocnadsietiso,n. Wisiuthsemfueslot/osthaativoenaaryfuilslyposstsaitbiolentaorfiyxcpoonsditiitoionnasndthvaetlomcaitiynftoarins constant a grouaplsoof pthaertiecnleesr,gwyaallnsdasthexeadmepnlsei,tbyu.tFinotretrhniasl enneewrgfiyxa,ncdadlleendsimtyewsoil/lfbuellustpadtaiotenda.ry, we take In somaes caasreesf,eriteniscuestehfeulmtoeshoa/vsetaatfiounllyarsytafitixondaercylacroenddaitniodnsintihtiaatlmiseadintainfisx_comnesstaon_tstationary.h also thaenednfierxg_ymaensdo_tshteatdioennasirtyy..cpFporfitlhesisinnetwhefidxi,reccatlolerdy m/sersco//UfuSlElsRta-StiPoHna.rAy,llwtheetafiklees regarding as a remfereesnoc/efuthllestmateisoon/asrtyatmionuasrtybfiexsadveecdlarinedthaend/sinrcit/iaUliSsEedR-inSPfiHx_dmireescot_osrtyatiaonndariyt.shhierarchy is and fixs_hmowesno_isntaFtiiognuarrey.6c.pp files in the directory /src/USER-SPH. All the files regarding meso/fullstationary must be saved in the /src/USER-SPH directory. LAMMPS Modify Fix Figure 6. Class hierarchy of the new fix style. 6.1. Validation The meso/fullstationary has been used in the validation of the new viscosity class to set the boundary condition of a constant asymmetric heated walls, see Section 7.2. 6.2. fix_meso_fullstationary.cpp All the functions will be the same as in the reference meso/stationary. However, in our new fullstationary, we need to substitute the "FixMesoStationary" text in "FixMeso- FullStationary", as can be seen in Listings 69 and 70. Listing 69: Original script (fix_meso_stationary.cpp)

1 #include 2 #include 3 #include 4 #include 5 #include "fix_meso_stationary.h" 6 #include "atom.h" 7 #include "comm.h" 8 #include "force.h" 9 #include "neighbor.h" 10 #include "neigh_list.h" 11 #include "neigh_request.h" 12 #include "update.h" 13 #include "integrate.h" 14 #include "respa.h" 15 #include "memory.h" #include "error.h" #include "pair.h" 19 using namespace LAMMPS_NS; 20 using namespace FixConst; 21 22 FixMesoStationary:: FixMesoStationary(LAMMPS *lmp, 23 int narg, char **arg) : Fix(lmp, narg, arg) 24 {...} 25 int FixMesoStationary::setmask() 26 {...} 27 void FixMesoStationary::init() 28 {...} 29 void FixMesoStationary::initial_integrate(int /*vflag*/) 30 {...} 31 void FixMesoStationary::final_integrate() 32 {...} 33 void FixMesoStationary::reset_dt() 34 {...} Listing 70: Modified script (fix_meso_fullstationary.cpp) 1 #include 2 #include 3 #include 4 #include 5 #include "fix_meso_fullstationary.h" 6 #include "atom.h" 7 #include "comm.h" 8 #include "force.h" 9 #include "neighbor.h" 10 #include "neigh_list.h" 11 #include "neigh_request.h" 12 #include "update.h" 13 #include "integrate.h" 14 #include "respa.h" 15 #include "memory.h" 16 #include "error.h" 17 #include "pair.h" 18 19 using namespace LAMMPS_NS; 20 using namespace FixConst; 21 22 FixMesoFullStationary::FixMesoFullStationary(LAMMPS *lmp, 23 int narg, char **arg) : Fix(lmp, narg, arg) 24 {...} 25 int FixMesoFullStationary::setmask() 26 {...} 27 void FixMesoFullStationary::init() 28 {...} 29 void FixMesoFullStationary::initial_integrate 30 (int /*vflag*/) 31 {...} 32 void FixMesoFullStationary::final_integrate() 33 {...} 34 void FixMesoFullStationary::reset_dt() 35 {...} For the meso/fullstationary we need to modify two function: initial_integrate, see Listing 72 line 16 and 17, and final_integrate, see Listing 74 line 14 and 15. Listing 71: Original initial_integrate (fix_meso_stationary.cpp) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 void FixMesoStationary::initial_integrate(int vflag) { double *rho = atom->rho; double *drho = atom->drho; double *e = atom->e; double *de = atom->de; int *type = atom->type; int *mask = atom->mask; int nlocal = atom->nlocal; int i; if (igroup == atom->firstgroup) nlocal = atom->nfirst; for (i = 0; i < nlocal; i++) { if (mask[i] & groupbit) { e[i] += dtf * de[i]; // with this line is possible to update internal energy rho[i] += dtf * drho[i]; // ... and density every half-step }}} Listing 72: Modified initial_integrate (fix_meso_fullstationary.cpp) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 void FixMesoFullStationary::initial_integrate(int vflag) { double *rho = atom->rho; double *drho = atom->drho; double *e = atom->e; double *de = atom->de; int *mask = atom->mask; int nlocal = atom->nlocal; int i; if (igroup == atom->firstgroup) nlocal = atom->nfirst; for (i = 0; i < nlocal; i++) { if (mask[i] & groupbit) { e[i] +=0; // with this line internal energy rho[i] += 0; // ... and density are constant }}} Listing 73: Original final_integrate (fix_meso_stationary.cpp) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 void FixMesoStationary::final_integrate() { double *e = atom->e; double *de = atom->de; double *rho = atom->rho; double *drho = atom->drho; int *type = atom->type; int *mask = atom->mask; double *mass = atom->mass; int nlocal = atom->nlocal; if (igroup == atom->firstgroup) nlocal = atom->nfirst; for (int i = 0; i < nlocal; i++) { if (mask[i] & groupbit) { e[i] += dtf * de[i]; rho[i] += dtf * drho[i]; }}} Listing 74: Modified final_integrate (fix_meso_fullstationary.cpp) 1 vvoid FixMesoFullStationary::final_integrate() { 2 3 double *e = atom->e; 4 double *de = atom->de; 5 double *rho = atom->rho; 6 double *drho = atom->drho; 7 int *mask = atom->mask; 8 int nlocal = atom->nlocal; 9 if (igroup == atom->firstgroup) 10 nlocal = atom->nfirst; 11 12 for (int i = 0; i < nlocal; i++) { 13 if (mask[i] & groupbit) { 14 e[i] += 0; // with this line internal energy 15 rho[i] += 0; //... and density are constant 16 }}} 6.3. fix_mes_fullstationary.h In the header of the new fix we need to substitute the "FixMesoStationary" text in "FixMesoFullStationary". Listing 75: Original header (pair_sph_heatconduction.h) 1 #ifdef FIX_CLASS 2 3 FixStyle(meso/stationary,FixMesoStationary) 4 5 #else 6 7 #ifndef LMP_FIX_MESO_STATIONARY_H 8 #define LMP_FIX_MESO_STATIONARY_H 9 10 #include "fix.h" 11 12 namespace LAMMPS_NS { 13 14 class FixMesoStationary : public Fix { 15 public: 16 FixMesoStationary(class LAMMPS *, int, char **); 17 int setmask(); 18 virtual void init(); 19 virtual void initial_integrate(int); 20 virtual void final_integrate(); 21 void reset_dt(); 22 23 private: 24 class NeighList *list; 25 protected: 26 double dtv,dtf; 27 double *step_respa; 28 int mass_require; 29 30 class Pair *pair; 31 }; 32 } 33 #endif 34 #endif Listing 76: Modified header (pair_sph_heatgasliquid.h) 1 #ifdef FIX_CLASS 2 3 FixStyle(meso/fullstationary,FixMesoFullStationary) #else 6 7 #ifndef LMP_FIX_MESO_FULLSTATIONARY_H 8 #define LMP_FIX_MESO_FULLSTATIONARY_H 9 10 #include "fix.h" 11 12 namespace LAMMPS_NS { 13 14 class FixMesoFullStationary: public Fix { 15 public: 16 FixMesoFullStationary(class LAMMPS *, int, char **); 17 int setmask(); 18 virtual void init(); 19 virtual void initial_integrate(int); 20 virtual void final_integrate(); 21 void reset_dt(); 22 23 private: 24 class NeighList *list; 25 protected: 26 double dtv,dtf; 27 double *step_respa; 28 int mass_require; 29 30 class Pair *pair; 31 }; 32 } 33 #endif 34 #endif 6.4. Invoking Meso/Fullstationary Fix Now the new fix is completed. To run LAMMPS with the new style we need to compile it and then invoke it by writing the command lines shown in Listing 77 in the input file. Listing 77: Command lines to invoke the new pair style 1 fix ID group-ID meso/fullstationary 7. Viscosity Class Viscosity in the SPH method has been addressed with different solutions [46]. Shock waves, for example, have been a challenge to model due to the arise of numerical oscilla- tions around the shocked region. Monaghan solved this problem with the introduction of the so-called Monaghan artificial viscosity [48]. Artificial viscosity is still used nowadays for energy dissipation and to prevent unphysical penetration for particles approaching each other [25,49]. The SPH package of LAMMPS uses the following artificial viscosity expression [6], within the sph/idealgas and sph/taitwater pair style. Πij = −αh ρi + ρj r2ij + εh2 ci + cj vij · rij , (12) where α is the dimensionless dissipation factor, ci and cj are the speed of sound of particle i and j. The dissipation factor, α, can be linked with the real viscosity in term of [6] α=8 μ , (13) chρ where c is the speed of sound, ρ the density, μ the dynamic viscosity and h the smooth- ing length. The artificial viscosity approach performs well at a high Reynolds number but better solutions are available for laminar flow: Morris et al. [50] approximated and implemented the viscosity momentum term for SPH. The same solution can be found in the sph/taitwa- ter/morris pair style with the expression [6]. solutions are available for laminar flow: Morris et al. [50] approximated and implemented the viscosity momentum term for SPH. The same solution can be found in the sph/taitwa- ter/morris pair style with the expression [6] Σ mimj(μi + μj)vij 1 ∂Wij ρiρj (rij ∂ri) , (14) j mimj(μi + μj)vij 1 ∂Wij where μ is the real dynamic viscosity. Σ j ρiρj (rij ∂ri) , (14) In LAMMPS both the dissipation factor and the dynamic viscosity are treated as a constant between a wphaeirrIenoμLfiAspMthaeMrrtPeiacSllbedoystnhwatmhhiecednviisstschiopseaiyttiyo.innftaecrtoarcatnwdtihtehdinyntahmeicsvmisocoositthyianregtrleeantegdthas. aIn this section we want to cmonastkanet tbheteweveinsacopasiirtoyf paarptieclresawtohemnthperyoinpteerratcytwiinthsintethaedsmoofoathipnagilernpgtrho.Ipnethrtisy only existing within a paseicrtiosntywleew.aMntotoremoavkeetrh,efivivsceostietymaperaatotmurperodpeerptyeinnsdteeadnotfvaipsaciropsriotpyermtyoondlyels are added. For this exameaxdidpsetlidne.g,Fnworoitthhriinesfeaexpraaeminrpscletey,nlfieo.lrMeefoeirsreeonuvcseeerfi,dlfie;viaseutnesmeedpw;earacnteluawrsecsdl,aesVps,einsdceonstivtyis,icsoissitiymmpoldeemlsaerinneted in LAMMPS from scraLAtcMhM.PS from scratch and its hierarchy is shown in Figure 7. ViscoFourParameterExp ViscoSutherlandViscosityLaw Viscosity ViscoPowerLawGas ViscosityArrhenius ViscosityConstant Figure 7. Class hieraFrcighuyre o7.f Cthlaess nhieewrarcchlyaosfsthe new class. 7.1. Temperature Dependant Viscosity 7.1. Temperature dependant viscosity In literature multiples empirical models that correlate viscosity with temperature In literature maruelatvipaillaebsle e[5m1–p53i]r.iIcnathlemneowdveislcsostihtyactlacssofirvreedlaiftfeerent viscosity mwodietlhs htaevmebpeeenrature are available [51–53im].pIlenmtehnteedn:ew viscosity class five different viscosity models have been implemented: 1. Andrade's equation [54] 1. Andrade's equation [54] μ = Aexp (BT + CT + DT2), (15) where μ is the viscosity in [Kg m−1 s−1], T is the static temperature in Kelvin, A, B, C and D are fluiμd-d=epeAndeexnptdimen+sioCnaTlco+effiDciTen2ts a,vailable in literature. B (15) 2. Arrhenius viscosity by Ram(anT[55,56]) where μ is the viscosity in [Kg m-1 s-1], T μis=thC1eexspt(aCt2i/cTt)e,mperature in Kelvin(,16A), B, C and D are fluid-dewpheernedμeisnthteddiymnaemnicsivoisncoaslitycoine[fKfigcmie−n1tss−1a],vTaiisltahbelteeminperlaittuerreaitnuKreelv.in, C1 2. Arrhenius viscosiatyndbCy2 aCre.Vflu.Rida-dmepaennd[e5nt5d,5im6e]nsional coefficients available in literature. 3. Sutherland's viscosity [57,58] for gas phase Sutherland's law can be expressed as: 4. μ = C1T3/2 T + C2 , (17) where μ is the viscosity in [Kg m−1 s−1], T is the static temperature in Kelvin, C1 and C2 are dimensional coefficients. Power-Law viscosity law [57] for gas phase A power-law viscosity law with two coefficients has the form : 5. μ = BTn, (18) where μ is the viscosity in [Kg m−1 s−1], T is the static temperature in Kelvin, and B is a dimensional coefficient. Constant viscosity With constant viscosity both dissipation factor and dynamic viscosity will be constant during the simulation. When the artificial viscosity is used the dissipation factor of Equation (12) is defined as the arithmetic mean of the dissipation factors of i-th particle and j-th particle. αij = − μi 4 h(ciρi + μj cjρj) , (19) where αij is the dissipation factor of the particles pair i and j. 7.2. Validation In order to validate the new Viscosity class, we will study the effect of asymmetrically heating walls in a channel flow, and more specifically the effect on the velocity field of the fluid. The data obtained with our model will be compared with the analytical solution obtained by Sameen and Govindarajan [59]. The water flows between two walls in the x-direction with periodic conditions. The walls are set at different temperatures Tcold and Thot , see Figure 8. Both water and walls are modelled as fluid

following the tait EOS. The physical properties of the walls are set constant throughout the simulation using the full stationary conditions described in Section 6. Figure 8. Geometry of the simulation. To match the condition used by Sameen and Govindarajan we set the cold wall temperature to Tcold = 295 K and the temperature dependence of the dynamic viscosity described by the Arrhenius model, Equation (16), with C1 = 0.000183 [Ns m−2] and C2 = 1879.9 K [59]. To describe the asymmetric heating Sameen and Govindarajan introduced the parameter m, defined as: m = μcold (20) μre f where μref = μhot is the viscosity at the hot wall in the case of asymmetric heating and μcold is the viscosity at the cold wall. By combining (16) and (20), with the given Tcold, is possible to express the temperature difference of the walls ΔT as function of m. Figure 9 shows the viscosity trend for different values of m and the corresponding ΔT. Sometimes, in particle methods, instantaneous data can be noisy (scattered) as can be seen from the blue circles of both Figures 9 and 10. 3 2.5 2 7r 7 ef [-] Instantaneous data 1.5 Trend curve Analytical solution 1 0.5 0 -1 -0.5 y 0 0.5 1 h [-] (a) m = 1 ↔ ΔT = 0 K 3 2.5 2 7r 7 ef [-] Instantaneous data 1.5 Trend curve Analytical solution 1 0.5 0 -1 -0.5 y 0 0.5 1 h [-] (b) m = 1.65 ↔ ΔT = 25 K 3 2.5 2 7r 7 ef [-] Instantaneous data 1.5 Trend curve Analytical solution 1 0.5 0 -1 -0.5 y 0 0.5 1 h [-] (c) m = 2.5 ↔ ΔT = 50 K Figure 9. Dimensionless viscosity profile for different m = μcold/μre f . Blue circles are the instantaneous data in the x direction, the orange curve is the trend curve extrapolated from the instantaneous data, yellow circles are obtained form the analytical solution from Sameen and Govindarajan [59]. In all the cases considered, the model is in good agreement with the work of Sameen and Govindarajan. Figure 10 shows the dimensionless velocity trend for different values of m. Again, the model is in good agreement with the analytical solution of Sameen and Govindarajan always laying within the velocity scattered points. In both our model and in the analytical solution the maximum of the velocity shifts to the right as m increases. We can conclude that our model is in good agreement with the literature, showing the typical viscosity and velocity profiles for asymmetric heating confirming the correct functionality of the new viscosity class. 7.3. New Abstract Class: Viscosity To implement the new viscosity model a new abstract class has been created, called Viscosity. The class has no attribute, and one virtual method: compute_visc, that is used to compute the viscosity using one of the Equations (15)–(18). As usual, the Viscosity class is divided in two files, see Listings 78 and 79. As it is an abstract class, it cannot be instantiated. It is used as a base, a mold, to implement the viscosity models. All implemented viscosity classes, such as the ones implementing the Arrhenius viscosity or the Sutherland viscosity, will inherit from this class. 1.2 1 0.8 Vmax [-] Instantaneous data 0.6 Trend curve V Analytical solution 0.4 0.2 0 -1 -0.5 y 0 0.5 1 h [-] (a) m = 1 ↔ ΔT = 0 K 1.2 1 0.8 Vmax [-] Instantaneous data 0.6 Trend curve V Analytical solution 0.4 0.2 0 -1 -0.5 y 0 0.5 1 h [-] (b) m = 1.65 ↔ ΔT = 25 K 1.2 1 0.8 Vmax [-] Instantaneous data 0.6 Trend curve V Analytical solution 0.4 0.2 0 -1 -0.5 y 0 0.5 1 h [-] (c) m = 2.5 ↔ ΔT = 50 K Figure 10. Dimensionless velocity profile for different m = μcold/μre f . Blue circles are the instantaneous data in the x direction, the orange curve is the trend curve extrapolated from the instantaneous data, yellow circles are obtained form the analytical solution from Sameen and Govindarajan [59]. Listing 78: viscosity.cpp 1 #include "viscosity.h" 2 3 using namespace LAMMPS_NS; 4 5 Viscosity::Viscosity() {}; Listing 79: viscosity.h 1 #ifndef LAMMPS_VISCOSITY_H 2 #define LAMMPS_VISCOSITY_H 3 4 namespace LAMMPS_NS { 5 class Viscosity { 6 /** 7 * Abstract base class for the viscosity attribute. 8 * All viscosity types should inherit from this class. 9 */ 10 public: 11 Viscosity(); 12 /** 13 * Virtual function. 14 * Returns the viscosity, given the temperature. 15 */ 16 virtual double compute_visc(double temperature) = 0; 17 }; 18 } 19 #endif //LAMMPS_VISCOSITY_H This type of base class is called an interface, though as the code is written in C++, there is no actual difference in the implementation. The difference is only in concepts. This structure allows for a very simple procedure to add a new viscosity type to LAMMPS, as one doesn't have to go through all of the code everytime a new viscosity type is implemented. All that is required is to implement a new viscosity class inheriting from the Viscosity abstract class and modify the add_viscosity function. The details of the changes required for those two actions are detailed later in this section. Another structure one might think of to implement the viscosity abstract base class would be a template. Indeed, templates are more efficient than inherited classes as inherited classes create additional virtual calls when calling the class's methods. However, the choice of which viscosity should be called is made at runtime, and not at compile time, which means the abstract base class would be a better fit. When runtime polymorphism is needed, the structure preferred is an abstract base class. The abstract class is not the most efficient implementation, but it allows for simplicity of use, which is important considering most of LAMMPS users are not programmers. In this work, we have chosen to sacrifice a bit of efficiency to gain ease of use. 7.4. Implementing a New Viscosity Class In this section the steps to implement one of Equations (15)–(18) are shown, using the four parameter exponential viscosity as an example. A new class is created that inherits from the Viscosity abstract class. The new class have as much attributes as the viscosity type has parameters. In this example that means four, as shown in the header in Listing 80. Listing 80: viscosity_four_parameter_exp.h 1 #ifndef LAMMPS_VISCOSITY_FOURPARAMETEREXP_H 2 #define LAMMPS_VISCOSITY_FOURPARAMETEREXP_H 3 4 #include "math.h" 5 #include "viscosity.h" 6 namespace LAMMPS_NS{ 7 8 class ViscosityFourParameterExp : public Viscosity{ 9 /** 10 * Implementation of the four parameter exponential viscosity. 11 * This viscosity has four attributes. 12 */ 13 private: 14 double A; 15 double B; 16 double C; 17 double D; 18 public: 19 ViscosityFourParameterExp(double A, double B, double C, double D); 20 21 double compute_visc(double temperature) override final; 22 23 }; 24 }; 25 #endif //LAMMPS_VISCOSITY_FOURPARAMETEREXP_H The constructor therefore should take as arguments the four parameters of the An- drade's equation and initialise the class's attributes with those values. The last step is to implement the compute_visc method so it returns the value of the viscosity at the given temperature. The implementation of both those functions is shown in Listing 81. Listing 81: viscosity_four_parameter_exp.cpp 1 #include "viscosity_four_parameter_exp.h" 2 3 using namespace LAMMPS_NS; 4 5 ViscosityFourParameterExp::ViscosityFourParameterExp(double A, double B, double C, double D) { 6 this->A = A; 7 this->B = B; 8 this->C = C; 9 this->D = D; 10 } 11 12 double ViscosityFourParameterExp::compute_visc(double temperature) { 13 return A*exp(B/temperature +C*temperature + D *temperature*temperature); 14 } Similar steps have to be taken to implement the classes corresponding to the other viscosity models, see the Supplementary material. 7.5. Processing the Viscosity in the Atom Class In the header of the Atom class we need to include the new viscosity class and declare a new set of public members. Listing 82: Original header (atom.h) 1 #include "pointers.h" 2 #include 3 #include Listing 83: Modified header (atom.h) 1 #include "pointers.h" 2 #include "viscosity.h" 3 #include 4 #include We add two new attributes in the USER-SPH section of the Atom attribute lists: viscosity, a pointer to a Viscosity object and viscosities, a pointer to an array containing the values of dynamic viscosities for all atoms at the current time step. Listing 84: Original header (atom.h) 1 2 3 1 2 3 4 5 6 // USER-SPH package double *rho,*drho,*e,*de,*cv; double **vest; Listing 85: Modified header (atom.h) // USER-SPH package double *rho,*drho,*e,*de,*cv; double **vest; Viscosity *viscosity; double *viscosities; We want to be able to choose which type of viscosity is being used in the simulation from the input file, using a new command called viscosity. Let's discuss the implemen- tation of this feature. First we need to define the viscosity command. This is done by modifying the execute_command method of the Input class. We then define a new func- tion called add_viscosity, whose declaration is shown in Listing 86 and definition in Listing 87. This function will have to be modified each time one wants to create a new viscosity class. In add_viscosity, the element arg[0] is the string representing the type of viscosity. For each viscosity class, the method performs the following procedure: • It checks which type of viscosity is asked to be created using the function strcmp on arg[0] (for Andrade's viscosity it corresponds to line 3 of Listing 87) • It checks if the number of arguments is coherent with the number of parameter of the viscosity type (line 4–5) • • It scans the coefficients of that viscosity type (line 6–10) It creates the appropriate viscosity and initializes the Viscosity attribute (line 11). This process should be followed for any new implementation. Listing 86: Modified header (atom.h) 1 void add_viscosity(int narg, char **arg); Listing 87: New add_viscosity function (atom.cpp) 1 void Atom::add_viscosity(int narg, char **arg) { 2 if (narg < 1) error->all(FLERR, "Too few arguments for creation of viscosity"); 3 if (!strcmp(arg[0], "FourParameterExp")) { 4 if (narg != 5) 5 error->all(FLERR, "Wrong number of arguments for creation of four 6 parameter exponential viscosity"); 7 double A, B, C, D; 8 sscanf(arg[1], "%lg", &A); 9 sscanf(arg[2], "%lg", &B); 10 sscanf(arg[3], "%lg", &C); 11 sscanf(arg[4], "%lg", &D); 12 this->viscosity = new ViscosityFourParameterExp(A, B, C, D); 13 std::cout <<"Viscosity created" <all(FLERR, "Wrong number of arguments for creation of Sutherland viscosity"); 18 double A, B; 19 sscanf(arg[1], "%lg", &A); 20 sscanf(arg[2], "%lg", &B); 21 this->viscosity = new SutherlandViscosityLaw(A, B); 22 std::cout <<"Viscosity created" <all(FLERR, "Wrong number of arguments for creation of power law gas viscosity"); 27 double B; 28 sscanf(arg[1], "%lg", &B); 29 this->viscosity = new PowerLawGas(B); 30 std::cout <<"Viscosity created" <all(FLERR, "Wrong number of arguments for creation of Arrhenius viscosity"); 35 double A; 36 double B; 37 sscanf(arg[1], "%lg", &A); 38 sscanf(arg[2], "%lg", &B); 39 this->viscosity = new ViscosityArrhenius(A, B); 40 std::cout <<"Viscosity created" <all(FLERR, "Wrong number of arguments for creation 45 of Constant viscosity"); 46 double A; 47 sscanf(arg[1], "%lg", &A); 48 this->viscosity = new ViscosityConstant(A); std::cout <<"Viscosity created" < 2 #include 3 #include "viscosity_four_parameter_exp.h" 4 #include "viscosity_sutherland_law.h" 5 #include "viscosity_power_law_gas.h" 6 #include "viscosity_arrhenius.h" 7 #include "viscosity_constant.h" The viscosity attribute is initialised to NULL in the constructor, see Listing 89. Listing 89: Inside Atom::Atom(LAMMPS *lmp) : Pointers(lmp) (atom.cpp) 1 viscosity = NULL; In the destructor of the Atom class, we add a line to delete the viscosity attribute, see Listing 90. Listing 90: Inside Atom:: Atom() (atom.cpp) 1 memory->destroy(viscosity); The extract function is modified to process the viscosity attribute, see Listing 91. Listing 91: Modified extract function (atom.cpp) 1 if (strcmp(name, "viscosity") == 0) return (void *) viscosity; 7.6. Using compute_Visc in SPH Pair Styles: Tait Water Implementation The dynamic viscosity is used to compute the artificial viscosity force, that is used in the compute function of the following SPH pair style: sph/idealgas, sph/lj, sph/tait- water and sph/taitwater/morris. In this section the steps to implement compute_visc in sph/taitwater are shown, the others required a similar procedure. The first function to modify is the destructor, as we don't have to allocate the viscosity parameter anymore. Listing 92: Original file (pair_sph_taitwater.cpp) 1 PairSPHTaitwater::~PairSPHTaitwater() { 2 if (allocated) { 3 memory->destroy(setflag); 4 memory->destroy(cutsq); 5 memory->destroy(cut); 6 memory->destroy(rho0); 7 memory->destroy(soundspeed); 8 memory->destroy(B); 9 memory->destroy(viscosity); 10 } 11 } Listing 93: Modified file (pair_sph_taitwater.cpp) 1 PairSPHTaitwater::~PairSPHTaitwater() { if (allocated) { memory->destroy(setflag); 4 memory->destroy(cutsq); 5 memory->destroy(cut); 6 memory->destroy(rho0); 7 memory->destroy(soundspeed); 8 memory->destroy(B); 9 } 10 } For the same reason as the destructor we need to modify allocate. Listing 94: Original file (pair_sph_taitwater.cpp) 1 void PairSPHTaitwater::allocate() { 2 allocated = 1; 3 int n = atom->ntypes; 4 memory->create(setflag, n + 1, n + 1, "pair:setflag"); 5 for (int i = 1; i <= n; i++) 6 for (int j = i; j <= n; j++) 7 setflag[i][j] = 0; 8 memory->create(cutsq, n + 1, n + 1, "pair:cutsq"); 9 memory->create(rho0, n + 1, "pair:rho0"); 10 memory->create(soundspeed, n + 1, "pair:soundspeed"); 11 memory->create(B, n + 1, "pair:B"); 12 memory->create(cut, n + 1, n + 1, "pair:cut"); 13 memory->create(viscosity, n + 1,n + 1,"pair:viscosity"); 14 } Listing 95: Modified file (pair_sph_taitwater.cpp) 1 void PairSPHTaitwater::allocate() { 2 allocated = 1; 3 int n = atom->ntypes; 4 memory->create(setflag, n + 1, n + 1, "pair:setflag"); 5 for (int i = 1; i <= n; i++) 6 for (int j = i; j <= n; j++) 7 setflag[i][j] = 0; 8 memory->create(cutsq, n + 1, n + 1, "pair:cutsq"); 9 memory->create(rho0, n + 1, "pair:rho0"); 10 memory->create(soundspeed, n + 1, "pair:soundspeed"); 11 memory->create(B, n + 1, "pair:B"); 12 memory->create(cut, n + 1, n + 1, "pair:cut"); 13 } Inside the compute function of the sph/taitwater pair style we need to declare a new set of variables. Where e is the energy and cv the heat capacity, now needed to calculate the temperature and thus the viscosity. Listing 96: Original file (pair_sph_taitwater.cpp) 1 int *type = atom->type; 2 int nlocal = atom->nlocal; 3 int newton_pair = force->newton_pair; 1 2 [linebackgroundcolor={\listyellow{4,5,6,7}}, label=820, caption={\small Modified file (pair\textunderscore sph\textunderscore taitwater.cpp)}\label{32}] % Start your code-block 3 4 5 6 7 8 9 10 int *type = atom->type; int nlocal = atom->nlocal; int newton_pair = force->newton_pair; double *e = atom->e; double *cv = atom->cv; Viscosity* viscosity = atom->viscosity; double* viscosities = atom->viscosities; The next modification is inside the loop over the j-th atom when the force induced by the artificial viscosity is calculated inside the pair's compute function. The dynamic viscosities μi and μj are calculated for each atoms, using the formula implemented in the compute_visc method. The temperature for the i-th atom is obtained using Ti = ei/cvi. It is important to note that using such expression for the energy balance prevents the reference state of the internal energy to be set at 0. The constant viscosity matrix element is replaced by the formula defined in Equation (19), see Listings 97 and 98. Listing 97: Original file (pair_sph_taitwater.cpp) 1 // artificial viscosity (Monaghan 1992) 2 if (delVdotDelR < 0.) { 3 mu = h * delVdotDelR / (rsq + 0.01 * h * h); 4 fvisc = -viscosity[itype][jtype] * (soundspeed[itype] 5 + soundspeed[jtype]) * mu / (rho[i] + rho[j]); 6 } else { 7 fvisc = 0.; 8 } Listing 98: Modified file (pair_sph_taitwater.cpp) 1 viscosities[i] = viscosity->compute_visc(e[i]/cv[i]); 2 viscosities[j] = viscosity->compute_visc(e[j]/cv[j]); 3 // artificial viscosity (Monaghan 1992) 4 if (delVdotDelR < 0.) { 5 mu = h * delVdotDelR / (rsq + 0.01 * h * h); 6 fvisc =-4/h*(viscosities[i]/(soundspeed[itype]*rho[i]) 7 +viscosities[j]/(soundspeed[jtype]*rho[j])) 8 *(soundspeed[itype]+ soundspeed[jtype]) 9 * mu / (rho[i] + rho[j]); 10 } else { 11 fvisc = 0.; 12 } Viscosity is now a per atom property, this means that we don't have to pass its value then the pair style is invoked. For this reason we need to delete the viscosity related lines inside coeff. Listing 99: Original coeff (pair_sph_taitwater.cpp) 1 void PairSPHTaitwater::coeff(int narg, char **arg) { 2 if (narg != 6) 3 error->all(FLERR, 4 "Incorrect args for pair_style sph/taitwater 5 coefficients"); 6 if (!allocated) 7 allocate(); 8 9 int ilo, ihi, jlo, jhi; 10 force->bounds(FLERR,arg[0], atom->ntypes, ilo, ihi); 11 force->bounds(FLERR,arg[1], atom->ntypes, jlo, jhi); 12 13 double rho0_one = force->numeric(FLERR,arg[2]); 14 double soundspeed_one = force->numeric(FLERR,arg[3]); 15 double viscosity_one = force->numeric(FLERR,arg[4]); 16 double cut_one = force->numeric(FLERR,arg[5]); 17 double B_one = soundspeed_one*soundspeed_one*rho0_one/7; 18 19 int count = 0; 20 for (int i = ilo; i <= ihi; i++) { 21 rho0[i] = rho0_one; 22 soundspeed[i] = soundspeed_one; 23 B[i] = B_one; 24 for (int j = MAX(jlo,i); j <= jhi; j++) { 25 viscosity[i][j] = viscosity_one; 26 cut[i][j] = cut_one; 27 setflag[i][j] = 1; 28 count++; 29 } } if (count == 0) 32 error->all(FLERR,"Incorrect args for pair 33 coefficients"); 34 } Listing 100: Modified coeff (pair_sph_taitwater.cpp) 1 void PairSPHTaitwater::coeff(int narg, char **arg) { 2 if (narg != 5) 3 error->all(FLERR, 4 "Incorrect args for pair_style sph/taitwater 5 coefficients"); 6 if (!allocated) 7 allocate(); 8 9 int ilo, ihi, jlo, jhi; 10 force->bounds(FLERR,arg[0], atom->ntypes, ilo, ihi); 11 force->bounds(FLERR,arg[1], atom->ntypes, jlo, jhi); 12 13 double rho0_one = force->numeric(FLERR,arg[2]); 14 double soundspeed_one = force->numeric(FLERR,arg[3]); 15 double cut_one = force->numeric(FLERR,arg[4]); 16 double B_one = soundspeed_one*soundspeed_one*rho0_one/7; 17 18 int count = 0; 19 for (int i = ilo; i <= ihi; i++) { 20 rho0[i] = rho0_one; 21 soundspeed[i] = soundspeed_one; 22 B[i] = B_one; 23 for (int j = MAX(jlo,i); j <= jhi; j++) { 24 cut[i][j] = cut_one; 25 setflag[i][j] = 1; 26 count++; 27 } 28 } 29 if (count == 0) 30 error->all(FLERR,"Incorrect args for pair 31 coefficients"); 32 } The last modification is in init_one. Again, we delete lines related to the former viscosity attribute. Listing 101: Original file (pair_sph_taitwater.cpp) 1 double PairSPHTaitwater::init_one(int i, int j) { 2 if (setflag[i][j] == 0) { 3 error->all(FLERR,"All pair sph/taitwater coeffs 4 are set"); 5 } 6 cut[j][i] = cut[i][j]; 7 viscosity[j][i] = viscosity[i][j]; 8 return cut[i][j]; 9 } Listing 102: Modified file (pair_sph_taitwater.cpp) 1 double PairSPHTaitwater::init_one(int i, int j) { 2 if (setflag[i][j] == 0) { 3 error->all(FLERR,"All pair sph/taitwater coeffs 4 are set"); 5 } 6 cut[j][i] = cut[i][j]; 7 return cut[i][j]; 8 } 7.7. Running the New Software with Mpirun At this stage, the software is designed to only run in serial. Changes need to be made to make it run with Message Passing Interface (MPI). This will allow the software to run in parallel: some computations being independent from each other, they can be performed at the same time. Instead of using one processor for a long time, we will use multiple processors for a shorter period. The simulation will therefore take more computing resources but will take a lot shorter to compute. The original SPH module can already be run with MPI however as we have modified the code that is no longer true. We need to make additional changes to the software. All those changes are located in the Atom Vec Meso class of the SPH module. In LAMMPS, the different MPI processes have to communicate with each other as the computations they perform are not completely independent from each other. They need data from other processes in order to perform their own calculations. They communicate with each other using a buffer that will contain all the necessary data. The buffer is simply an array that we will fill with the data. The different methods for packing and unpacking this buffer are defined in the Atom Vec Meso class. We need to add a new data to transmit: the calculated viscosity. The first thing to do is to increase the size of the buffers in their initialisation so they can accept the viscosity value, an example is shown in Listings 103 and 104. Listing 103: Original constructor (atom_vec_meso.cpp) 1 AtomVecMeso::AtomVecMeso(LAMMPS *lmp) : AtomVec(lmp) 2 { 3 molecular = 0; 4 mass_type = 1; 5 forceclearflag = 1; 6 7 // we communicate not only x forward but also vest .. 8 comm_x_only = 0; . 9 // we also communicate de and drho in reverse direction 10 comm_f_only = 0; 11 // 3 + rho + e + vest[3], that means we may 12 // only communicate 5 in hybrid 13 size_forward = 8; 14 size_reverse = 5; // 3 + drho + de 15 size_border = 12; // 6 + rho + e + vest[3] + cv 16 size_velocity = 3; 17 size_data_atom = 8; 18 size_data_vel = 4; 19 xcol_data = 6; 20 21 atom->e_flag = 1; 22 atom->rho_flag = 1; 23 atom->cv_flag = 1; 24 atom->vest_flag = 1; 25 } Listing 104: Modified constructor (atom_vec_meso.cpp) 1 AtomVecMeso::AtomVecMeso(LAMMPS *lmp) : AtomVec(lmp) 2 { 3 molecular = 0; 4 mass_type = 1; 5 forceclearflag = 1; 6 7 // we communicate not only x forward but also vest ... 8 comm_x_only = 0; 9 // we also communicate de and drho in reverse direction 10 comm_f_only = 0; 11 // 3 + rho + e + vest[3] + viscosities, that means we may 12 // only communicate 6 in hybrid 13 size_forward = 9; 14 size_reverse = 5; // 3 + drho + de 15 // 6 + rho + e + vest[3] + cv + viscosities 16 size_border = 13; size_velocity = 3; size_data_atom = 8; size_data_vel = 4; 20 xcol_data = 6; 21 22 atom->e_flag = 1; 23 atom->rho_flag = 1; 24 atom->cv_flag = 1; 25 atom->vest_flag = 1; 26 } Then, we added the relevant elements of the attribute viscosities to the buffer in all the methods handling buffers, an example is shown in Listings 105 and 106. Listing 105: Original pack_vec_hybrid (atom_vec_meso.cpp) 1 int AtomVecMeso::pack_comm_hybrid(int n, int *list, 2 double *buf) { 3 //printf("in AtomVecMeso::pack_comm_hybrid\n"); 4 int i, j, m; 5 6 m = 0; 7 for (i = 0; i < n; i++) { 8 j = list[i]; 9 buf[m++] = rho[j]; 10 buf[m++] = e[j]; 11 buf[m++] = vest[j][0]; 12

buf[m++] = vest[j][1]; 13 buf[m++] = vest[j][2]; 14 } 15 return m; 16 } Listing 106: Modified pack_vec_hybrid (atom_vec_meso.cpp) 1 int AtomVecMeso::pack_comm_hybrid(int n, int *list, 2 double *buf) { 3 //printf("in AtomVecMeso::pack_comm_hybrid\n"); 4 int i, j, m; 5 6 m = 0; 7 for (i = 0; i < n; i++) { 8 j = list[i]; 9 buf[m++] = rho[j]; 10 buf[m++] = e[j]; 11 buf[m++] = vest[j][0]; 12 buf[m++] = vest[j][1]; 13 buf[m++] = vest[j][2]; 14 buf[m++] = viscosities[j]; 15 } 16 return m; 17 } After making those changes for all the methods in the class, the software can be run using mpirun. 7.8. Invoking, Selecting and Computing a Viscosity Object To compute the new viscosity a new argument was added to the compute command: viscosities. This allows the user to use the compute command to output the dynamic viscosity to the dump file. This can be done by the following command: 1 compute viscosities_peratom all meso/viscosities/atom The implementation of this feature is simple, as it is very similar to other compute argument implementation. All that needs to be done is to modify another compute's implementation, such as compute_meso_rho_atom so it processes the variable viscosities instead of rho. The viscosity used in the simulation can be invoked in the input file, using the following command: 1 viscosity [type of viscosity] [parameters of the viscosity] The type of viscosity can be chosen from the following list: • FourParameterExp: the four parameter exponential viscosity law. • SutherlandViscosityLaw: the Sutherland viscosity law. • PowerLawGas: the power viscosity law for gases. • Arrhenius: the Arrhenius viscosity law. • Constant: a constant viscosity. For example, to invoke the four parameter exponential viscosity, we can write in the input file: 1 viscosity FourParameterExp C1 C2 C3 C4 As stated earlier, this list can easily be extended by the user by modifying the add_viscosity function defined earlier. 8. Conclusions Particle methods are very versatile and can be applied in a variety of applications, ranging from modelling of molecules to the simulation of galaxies. Their power is even amplified when they are coupled together within a discrete multiphysics framework. This versatility matches well with LAMMPS, which is a particle simulator, whose open-source code can be extended with new functionalities. However, modifying LAMMPS can be challenging for researchers with little coding experience and the available support material on how to modify LAMMPS is either too basic or too advanced for the average researcher. Moreover, most of the available material focuses on MD; while the aim of this paper is to support researchers that use other particle methods such as SPH or DEM. In this work, we present several examples, explained step-by-step and with increasing level of complexity. We begin with simple cases and concluding with more complex ones: Section 3 shows the implementation of the Kelvin–Voigt bond style used to model encap- sulate particles with a soft outer shell and validated validated by simulating spherical homogeneous linear elastic and viscoelastic particles [45]; Section 7 show how to imple- ment a new per-atom temperature dependant viscosity property and is validated finding the same viscosity and velocity trend shown by Sameen and Govindarajan [59] in their analytical solution for a channel flow in a asymmetrical heating walls. The work perfectly fits in the "Discrete Multiphysics: Modelling Complex Systems with Particle Methods" special issue by sharing some in dept know-how and "trick and trades" developed by our group in years of use of LAMMPS. In fact, the aim is to support, in several ways, researchers that use computational particle methods. Often researchers tend to write their own code. The advantage of this approach is that the code is well understood by the researcher and, therefore, easily extendible. However, this sometimes implies reinventing the wheel and countless hours of debugging. Familiarity with a code like LAMMPS, which has an active community of practice and is periodically enriched with new features would be beneficial to this type of researchers allowing them to save considerable time. In the long term, there is another advantage. Modules written for in-house code are hardly sharable. At the moment, the largest portion of the LAMMPS community is dedicated to MD. While this article was under review, for instance, a new book dedicated to modifying LAMMPS came out [60]. However, it focuses only on MD and it does not mention other discrete methods like SPH or DEM. Instead, the aim of this paper is to make LAMMPS more accessible for the Discrete Multiphysics community facilitating sharing reusable code among practitioners in this field. Supplementary Materials: The codes used in this work are freely available under the GNU General Public License v3 and can be downloaded from the University of Birmingham repository (http: //edata.bham.ac.uk/560/). Author Contributions: Conceptualization, A.A. (Andrea Albano) and A.A. (Alessio Alexiadis); methodology, A.A. (Andrea Albano); validation, A.A. (Andrea Albano), E.I.G., A.D., I.H.S. and A.R.; writing—original draft preparation, A.A. (Andrea Albano), E.I.G., A.D., I.H.S.; writing—review and editing, A.A. (Andrea Albano), A.A. (Alessio Alexiadis), C.A.D.-D., X.S., K.C.N. and M.A.; supervision, A.A. (Alessio Alexiadis), A.R. and I.M.; funding acquisition, A.A. (Alessio Alexiadis). All authors have read and agreed to the published version of the manuscript. Funding: This work was supported by the US office of Naval Research Global (ONRG) under NICOP Grant N62909-17-1-2051. Acknowledgments: The authors would like to thank Prof Albano (University of Pisa) for his advice and comments. The computations described in this paper were performed using the University of Birmingham's BlueBEAR HPC service, which provides a High Performance Computing service to the University's research community. See http://www.birmingham.ac.uk/bear for more details. Conflicts of Interest: The authors declare no conflict of interest. Abbreviations The following abbreviations are used in this manuscript: MS DMP SPH LAMMPS EOS LSM Molecular Dynamics Discrete MultiPhysics Smoothed Particle Hydrodynamics Large-scale Atomic/Molecular Massively Parallel Simulator Equation Of State Lattice Spring Model Appendix A. An Example of Discrete Multiphysics Simulation in LAMMPS In this section we present a simple case of DMP simulation with LAMMPS. It is an explanatory example deliberately simple for illustrative purposes. It involves only a small number of particles. Sensitivity analysis of the results with the model resolution or other numerical parameters are beyond the scope of this example and not carried out. The geometry is a 2D tube with an elastic membrane at one end (Figure A1). The tube contains a liquid simulated with the SPH model, Tait EOS and Morris viscosity. The wall is simulated with stationary particles and the membrane with the LSM using Hookean springs. In Figure A1, the liquid particles are red, the wall particles blue and the membrane particles yellow. During the simulation, the fluid is subjected to a force in the x-direction that pushed the particles against the membrane. Because the membrane is elastic, it stretches inflating the right end of the tube like a balloon. The resolution of the membrane is ten times higher than the fluid. This ensures that, as the membrane stretches, fluid particles do not 'leak' in the gaps formed between two consecutive membrane particles. The Lennard Jones potential, truncated to consider only the repulsive part, is used to avoid compenetration between solid and liquid particles. A weaker Lennard Jones potential is used as 'artificial pressure' to avoid excessive compression of the fluid particles. The initial data file (data.initial) for the geometry was create according to LAMMPS' rules for formatting the Data File [41] and is shared as additional material. In Data File, the fluid particles are called type 1, the wall particles type 2 and the membrane particles type 3. Here we focus on the input file (membrane.lmp), which is also shared in its entirety as additional material. We do not discuss LAMMPS syntax (the reader can refer to LAMMPS User 's Guide for this [41]), but only on specific parts of the input file that concern the DMP implementation. Figure A1. The inflating balloon simulation. The first section of the input file determines the dimensionality of the problem (2D), the boundary conditions (periodic), the units used (SI), the type of potential used in the simulation (atom_style) and the input file that contains the initial position of all the particles 1 dimension 2 2 boundary p p p 3 units si 4 atom_style hybrid meso bond angle 5 read_data data.initial The crucial line for DMP simulations is the hybrid keyword of the atom_style, which allows for combining different particle models. The keyword meso refers to the SPH model and bond, in the case under consideration, to the LSM. The angle keyword corresponds to angular springs, but, as it will be clear later, it is not used in this simulation. The following section contains several variables that are going to be used later on. In particular, the initial particle distance is dL and their mass m. The resolution of the membrane is Nt times higher than the fluid. The initial distance between membrane particle is therefore db=dL/Nt and their mass mM=m/Nt. 1 variable 2 variable 3 variable 4 variable 5 variable 6 variable 7 variable 8 variable 9 variable 10 variable 11 variable 12 variable 13 variable variable variable variable dL m Nt dB mM h h2 c mu rho kA kB skin epsL epsS sgmL equal 0.000111111 equal 1.23457e-05 equal 10 equal ${dL}/${Nt} equal ${m}/${Nt} equal 1.5*${dL} equal ${dL}/${Nt} equal 0.1 equal 1.0e-3 equal 1000 equal 1.e-8 equal 100 equal 0.3*${h} equal 1.e-12 equal 1.e-10 equal ${dL} 17 variable 18 variable 19 variable sgmS equal 0.5*${sgmL}/${Nt} fmax equal 0.00005 ft equal ramp(0.,${fmax}) The section below identifies particles type 1 as a group called fluid, particles type 2 as a group called wall and particles type 3 as a group called membrane. The mass of type 3 particles is assigned (the mass of type 1, 2 was assigned in the data.initial file). The density of all particle is also assigned based on the value rho defined previously. 1 group fluid type 1 2 group wall type 2 3 group membrane type 3 4 mass 3 ${mM} 5 set group all meso/rho ${rho} The next section defines the pair potentials for non-bonded particles. In this simulation, we use different styles together (keyword hybrid/overlay). The sph/taitwater/morris pair style, which is used for all pair interactions except 2-2 (i.e., wall particles with them- selves); and the Lennard Jones potential lj/cut, which, as explained above, is used both as 'artificial pressure' and to avoid compenetration of solid and fluid particles. 1 pair_style hybrid/overlay sph/taitwater/morris lj/cut ${sgmL} 2 pair_coeff 1 * sph/taitwater/morris ${rho} ${c} ${mu} ${h} 3 pair_coeff 2 3 sph/taitwater/morris ${rho} ${c} ${mu} ${h2} 4 pair_coeff 3 3 sph/taitwater/morris ${rho} ${c} ${mu} ${h2} 5 6 pair_coeff 1 * lj/cut ${epsL} ${sgmL} 7 pair_coeff 2 * lj/cut ${epsL} ${sgmL} 8 pair_coeff 1 3 lj/cut ${epsS} ${sgmL} 9 pair_coeff 3 3 lj/cut ${epsS} ${sgmS} After the non-bonded potentials, the script assigns the harmonic potential, with Hook constant kB and equilibrium distance dB, to the bonded particles (i.e., the membrane). All pairs of bonded particles are assigned in the data.initial file. 1 bond_style harmonic 2 bond_coeff 1 ${kB} ${dB} 3 angle_style none The next section assigns several parameters that determine how the Newton equation of motion is solved numerically. The force fmax is added to all fluid particle in the x- direction, and an artificial viscosity is added for stability reasons. 1 fix 2 fluid addforce ${fmax} 0.0 0.0 2 fix 5 fluid meso 3 fix 6 membrane meso 4 fix 8 wall meso/stationary 5 fix 9 all viscous 0.01 The last commands determine the value and the number of timesteps used in the simulation plus a variety of computations for output and other purposes that are not discussed here (the reader can refer to the User ’s Guide). 1 compute rho_peratom all meso/rho/atom 2 compute rho_ave all reduce ave c_rho_peratom 3 compute vmax fluid reduce max vx 4 thermo 10000 5 thermo_style custom step c_rho_ave c_vmax 6 thermo_modify norm no 7 neighbor ${skin} bin 8 dump dump_id all custom 10000 dump.lammpstrj id type x y z vx vy timestep 1.e-6 run 2500000 Appendix B. How to Compile LAMMPS LAMMPS is build as a library and executable [41] either by using GNU make [61] or a build environment with CMake [62]. In this appendix LAMMPS will be compiled only using make and it is compiled in BlueBEAR. For more details of the compiling process in LAMMPS refer to the user manual [41]. To compile LAMMPS in your own directory you can follow those steps 1. Download the file from here. Select the code you want, click the “Download Now” button, and your browser should download a gzipped tar file. Save the file in your directory on BlueBEAR 2. Unpack the file with the following command line command prompt: Listing A1: Command to open the tar file on BlueBEAR 1 tar -xvf lammps-stable.tar.gz 3. Before compiling is important to set up the environment, with BlueBEAR Listing A2: Commands to set the environment for compile LAMMPS on BlueBEAR 1 module purge 2 3 module load bluebear 4 5 module load Eigen/3.3.4-foss-2019a 4. Enter in the /src directory in your new LAMMPS directory. The src directory contains the C++ source and header files for LAMMPS. It also contains a top-level Makefile and a MAKE sub-directory with low-level Makefile.* files for many systems and machines. 5. Type the following command to compile a serial version of LAMMPS: Listing A3: Command to compile LAMMPS on BlueBEAR 1 make serial or a multi-threaded (parallel) version of LAMMPS: Listing A4: Command to compile LAMMPS on BlueBEAR 1 make mpi If you get no errors and an executable file lmp_mpi is produced. 6. Depending on the features you need, you will have to install same packages in your compiled LAMMPS. Is possible to check which packages is installed in your compiled LAMMPS by typing Listing A5: Command to check the list of installed packages (you must be inside the /src directory) 1 make ps It is possible to install the packages you need with the command line Listing A6: Command to install a specific package 1 make yes-NAMEPACK or un-install them with Listing A7: Command to un-install a specific package 1 make no-NAMEPACK More make commands are explained in LAMMPS user manual [41]. After the installa- tion of the desired packages you need to compile it again (step 5).

References 1. Plimpton, S. Fast Parallel Algorithms for Short-Range Molecular Dynamics; Technical Report; Sandia National Labs.: Albuquerque, NM, USA, 1993. 2. Plimpton, S.; Pollock, R.; Stevens, M. Particle-Mesh Ewald and rRESPA for Parallel Molecular Dynamics Simulations. In Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN, USA, 14–17 March 1997. 3. Auhl, R.; Everaers, R.; Grest, G.S.; Kremer, K.; Plimpton, S.J. Equilibration of long chain polymer melts in computer simulations. J. Chem. Phys. 2003, 119, 12718–12728. 4. Parks, M.L.; Lehoucq, R.B.; Plimpton, S.J.; Silling, S.A. Implementing peridynamics within a molecular dynamics code. Comput. Phys. Commun. 2008, 179, 777–783. 5. Petersen, M.K.; Lechman, J.B.; Plimpton, S.J.; Grest, G.S.; Veld, P.J.; Schunk, P. Mesoscale hydrodynamics via stochastic rotation dynamics: Comparison with Lennard-Jones fluid. J. Chem. Phys. 2010, 132, 174106. 6. Ganzenmüller, G.C.; Steinhauser, M.O.; Van Liedekerke, P.; Leuven, K.U. The implementation of Smooth Particle Hydrodynamics in LAMMPS. Paul Van Liedekerke Kathol. Univ. Leuven 2011, 1, 1–26. 7. Jaramillo-Botero, A.; Su, J.; Qi, A.; Goddard III, W.A. Large-scale, long-term nonadiabatic electron molecular dynamics for describing material properties and phenomena in extreme environments. J. Comput. Chem. 2011, 32, 497–512. 8. Coleman, S.; Spearot, D.; Capolungo, L. Virtual diffraction analysis of Ni [0 1 0] symmetric tilt grain boundaries. Model. Simul. Mater. Sci. Eng. 2013, 21, 055020. 9. Singraber, A.; Behler, J.; Dellago, C. Library-based LAMMPS implementation of high-dimensional neural network potentials. J. Chem. Theory Comput. 2019, 15, 1827–1840. 10. Ng, K.; Alexiadis, A.; Chen, H.; Sheu, T. A coupled Smoothed Particle Hydrodynamics-Volume Compensated Particle Method (SPH-VCPM) for Fluid Structure Interaction (FSI) modelling. Ocean Eng. 2020, 218, 107923. 11. Daraio, D.; Villoria, J.; Ingram, A.; Alexiadis, A.; Stitt, E.H.; Munnoch, A.L.; Marigo, M. Using Discrete Element method (DEM) simulations to reveal the differences in the γ-Al2O3 to α-Al2O3 mechanically induced phase transformation between a planetary ball mill and an attritor mill. Miner. Eng. 2020, 155, 106374. 12. Qiao, G.; Lasfargues, M.; Alexiadis, A.; Ding, Y. Simulation and experimental study of the specific heat capacity of molten salt based nanofluids. Appl. Therm. Eng. 2017, 111, 1517–1522. 13. Qiao, G.; Alexiadis, A.; Ding, Y. Simulation study of anomalous thermal properties of molten nitrate salt. Powder Technol. 2017, 314, 660–664. 14. Anagnostopoulos, A.; Navarro, H.; Alexiadis, A.; Ding, Y. Wettability of NaNO3 and KNO3 on MgO and Carbon Surfaces— Understanding the Substrate and the Length Scale Effects. J. Phys. Chem. C 2020, 124, 8140–8152. 15. Sahputra, I.H.; Alexiadis, A.; Adams, M.J. Effects of Moisture on the Mechanical Properties of Microcrystalline Cellulose and the Mobility of the Water Molecules as Studied by the Hybrid Molecular Mechanics–Molecular Dynamics Simulation Method. J. Polym. Sci. Part B Polym. Phys. 2019, 57, 454–464. 16. Sahputra, I.H.; Alexiadis, A.; Adams, M.J. Temperature dependence of the Young’s modulus of polymers calculated using a hybrid molecular mechanics–molecular dynamics method. J. Phys. Condens. Matter 2018, 30, 355901. 17. Mohammed, A.M.; Ariane, M.; Alexiadis, A. Using Discrete Multiphysics Modelling to Assess the Effect of Calcification on Hemodynamic and Mechanical Deformation of Aortic Valve. ChemEngineering 2020, 4, 48. 18. Ariane, M.; Vigolo, D.; Brill, A.; Nash, F.; Barigou, M.; Alexiadis, A. Using Discrete Multi-Physics for studying the dynamics of emboli in flexible venous valves. Comput. Fluids 2018, 166, 57–63. 19. Ariane, M.; Wen, W.; Vigolo, D.; Brill, A.; Nash, F.; Barigou, M.; Alexiadis, A. Modelling and simulation of flow and agglomeration in deep veins valves using discrete multi physics. Comput. Biol. Med. 2017, 89, 96–103. 20. Ariane, M.; Allouche, M.H.; Bussone, M.; Giacosa, F.; Bernard, F.; Barigou, M.; Alexiadis, A. Discrete multi-physics: A mesh-free model of blood flow in flexible biological valve including solid aggregate formation. PLoS ONE 2017, 12, e0174795. 21. Schütt, M.; Stamatopoulos, K.; Simmons, M.; Batchelor, H.; Alexiadis, A. Modelling and simulation of the hydrodynamics and mixing profiles in the human proximal colon using Discrete Multiphysics. Comput. Biol. Med. 2020, 121, 103819. 22. Alexiadis, A.; Stamatopoulos, K.; Wen, W.; Batchelor, H.; Bakalis, S.; Barigou, M.; Simmons, M. Using discrete multi-physics for detailed exploration of hydrodynamics in an in vitro colon system. Comput. Biol. Med. 2017, 81, 188–198. 23. Ariane, M.; Kassinos, S.; Velaga, S.; Alexiadis, A. Discrete multi-physics simulations of diffusive and convective mass transfer in boundary layers containing motile cilia in lungs. Comput. Biol. Med. 2018, 95, 34–42. 24. Ariane, M.; Sommerfeld, M.; Alexiadis, A. Wall collision and drug-carrier detachment in dry powder inhalers: Using DEM to devise a sub-scale model for CFD calculations. Powder Technol. 2018, 334, 65–75. 25. Albano, A.; Alexiadis, A. Interaction of Shock Waves with Discrete Gas Inhomogeneities: A Smoothed Particle Hydrodynamics Approach. Appl. Sci. 2019, 9, 5435. 26. Albano, A.; Alexiadis, A. A smoothed particle hydrodynamics study of the collapse for a cylindrical cavity. PLoS ONE 2020, 15, e0239830. 27. Albano, A.; Alexiadis, A. Non-Symmetrical Collapse of an Empty Cylindrical Cavity Studied with Smoothed Particle Hydrody-namics. Appl. Sci. 2021, 11, 3500. 28. Alexiadis, A. The discrete multi-hybrid system for the simulation of solid-liquid flows. PLoS ONE 2015, 10, e0124678. 29. Alexiadis, A. A new framework for modelling the dynamics and the breakage of capsules, vesicles and cells in fluid flow. Procedia IUTAM 2015, 16, 80–88. 30. Alexiadis, A. A smoothed particle hydrodynamics and coarse-grained molecular dynamics hybrid technique for modelling elastic particles and breakable capsules under various flow conditions. Int. J. Numer. Methods Eng. 2014, 100, 713–719. 31. Rahmat, A.; Barigou, M.; Alexiadis, A. Deformation and rupture of compound cells under shear: A discrete multiphysics study. Phys. Fluids 2019, 31, 051903. 32. Alexiadis, A.; Ghraybeh, S.; Qiao, G. Natural convection and solidification of phase-change materials in circular pipes: A SPH approach. Comput. Mater. Sci. 2018, 150, 475–483. 33. Rahmat, A.; Barigou, M.; Alexiadis, A. Numerical simulation of dissolution of solid particles in fluid flow using the SPH method. Int. J. Numer. Methods Heat Fluid Flow 2019, 30, 290–307. 34. Rahmat, A.; Meng, J.; Emerson, D.; Wu, C.Y.; Barigou, M.; Alexiadis, A. A practical approach for extracting mechanical properties of microcapsules using a hybrid numerical model. Microfluid. Nanofluidics 2021, 25, 1–17. 35. Ruiz-Riancho, I.N.; Alexiadis, A.; Zhang, Z.; Hernandez, A.G. A Discrete Multi-Physics Model to Simulate Fluid Structure Interaction and Breakage of Capsules Filled with Liquid under Coaxial Load. Processes 2021, 9, 354. 36. Sanfilippo, D.; Ghiassi, B.; Alexiadis, A.; Hernandez, A.G. Combined Peridynamics and Discrete Multiphysics to Study the Effects of Air Voids and Freeze-Thaw on the Mechanical Properties of Asphalt. Materials 2021, 14, 1579. 37. Alexiadis, A.; Albano, A.; Rahmat, A.; Yildiz, M.; Kefal, A.; Ozbulut, M.; Bakirci, N.; Garzón-Alvarado, D.; Duque-Daza, C.; Eslava-Schmalbach, J. Simulation of pandemics in real cities: Enhanced and accurate digital laboratories. Proc. R. Soc. A 2021, 477, 20200653. 38. Alexiadis, A. Deep Multiphysics and Particle–Neuron Duality: A Computational Framework Coupling (Discrete) Multiphysics and Deep Learning. Appl. Sci. 2019, 9, 5369. 39. Alexiadis, A. Deep multiphysics: Coupling discrete multiphysics with machine learning to attain self-learning in-silico models replicating human physiology. Artif. Intell. Med. 2019, 98, 27–34. 40. Alexiadis, A.; Simmons, M.; Stamatopoulos, K.; Batchelor, H.; Moulitsas, I. The duality between particle methods and artificial neural networks. Sci. Rep. 2020, 10, 1–7. 41. Sandia Corporation LAMMPS Users Manual. 2003. Available online: https://lammps.sandia.gov/doc/Developer.pdf (accessed on 11 Jauaury 2021). 42. Plimpton. LAMMPS Developer Guide. Available online: https://lammps.sandia.gov/doc/Developer.pdf (accessed on 11 Jauaury 2021). 43. Plimpton, S.J. Modifying & Extending LAMMPS; Technical Report; Sandia National Lab. (SNL-NM): Albuquerque, NM, USA, 2014. 44. Flügge, W. Viscoelasticity; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2013. 45. Sahputra, I.H.; Alexiadis, A.; Adams, M.J. A Coarse Grained Model for Viscoelastic Solids in Discrete Multiphysics Simulations. ChemEngineering 2020, 4, 30. 46. Liu, M.; Liu, G. Smoothed particle hydrodynamics (SPH): An overview and recent developments. Arch. Comput. Methods Eng. 2010, 17, 25–76. 47. Le Métayer, O.; Saurel, R. The Noble-Abel stiffened-gas equation of state. Phys. Fluids 2016, 28, 046102. 48. Monaghan, J.J.; Gingold, R.A. Shock simulation by the particle method SPH. J. Comput. Phys. 1983, 52, 374–389. 49. Lattanzio, J.; Monaghan, J.; Pongracic, H.; Schwarz, M. Controlling penetration. SIAM J. Sci. Stat. Comput. 1986, 7, 591–598. 50. Morris, J.P.; Fox, P.J.; Zhu, Y. Modeling low Reynolds number incompressible flows using SPH. J. Comput. Phys. 1997, 136, 214–226. 51. Cornelissen, J.; Waterman, H. The viscosity temperature relationship of liquids. Chem. Eng. Sci. 1955, 4, 238–246. 52. Seeton, C.J. Viscosity–temperature correlation for liquids. Tribol. Lett. 2006, 22, 67–78. 53. Stanciu, I. A new viscosity-temperature relationship for vegetable oil. J. Pet. Technol. Altern. Fuels 2012, 3, 19–23. 54. Gutmann, F.; Simmons, L. The temperature dependence of the viscosity of liquids. J. Appl. Phys. 1952, 23, 977–978. 55. De Guzman, J. Relation between fluidity and heat of fusion. Anales Soc. Espan. Fis. Quim 1913, 11, 353–362. 56. Raman, C. A theory of the viscosity of liquids. Nature 1923, 111, 532–533. 57. Chapman, S.; Cowling, T.G.; Burnett, D. The Mathematical Theory of Non-Uniform Gases: An Account of the Kinetic Theory of Viscosity, Thermal Conduction and Diffusion in Gases; Cambridge University Press: Cambridge, UK, 1990. 58. Rathakrishnan, E. Theoretical Aerodynamics; John Wiley & Sons: Hoboken, NJ, USA, 2013. 59. Sameen, A.; Govindarajan, R. The effect of wall heating on instability of channel flow. J. Fluid Mech. 2007, 577, 417–442. 60. Mubin, S.; Li, J. Extending and Modifying LAMMPS; Packt Publishing Ltd.: Birmingham, UK, 2021. 61. Project, G. Make-GNU Project-Free Software Fondation. Available online: https://www.gnu.org/software/make/ (accessed on 14 October 2020). 62. Martin, K.; Hoffman, B. Mastering CMake: A Cross-Platform Build System; Kitware: Clifton Park, NY, USA, 2010. ChemEngineering 2021, 5, 30 2 of 57 ChemEngineering 2021, 5, 30 3 of 57 ChemEngineering 2021, 5, 30 4 of 57 ChemEngineering 2021, 5, 30 5 of 57 ChemEngineering 2021, 5, 30 6 of 57 ChemEngineering 2021, 5, 30 7 of 57 ChemEngineering 2021, 5, 30 8 of 57 ChemEngineering 2021, 5, 30 9 of 57 ChemEngineering 2021, 5, 30 10 of 57 ChemEngineering 2021, 5, 30 11 of 57 ChemEngineering 2021, 5, 30 12 of 57 ChemEngineering 2021, 5, 30 13 of 57 ChemEngineering 2021, 5, 30 14 of 57 ChemEngineering 2021, 5, 30 15 of 57 ChemEngineering 2021, 5, 30 16 of 57 ChemEngineering 2021, 5, 30 17 of 57 ChemEngineering 2021, 5, 30 18 of 57 ChemEngineering 2021, 5, 30 19 of 57 20 of 57 ChemEngineering 2021, 5, 30 21 of 57 ChemEngineering 2021, 5, 30 22 of 57 ChemEngineering 2021, 5, 30 23 of 57 ChemEngineering 2021, 5, 30 24 of 57 ChemEngineering 2021, 5, 30 25 of 57 ChemEngineering 2021, 5, 30 26 of 57 ChemEngineering 2021, 5, 30 27 of 57 ChemEngineering 2021, 5, 30 28 of 57 ChemEngineering 2021, 5, 30 29 of 57 ChemEngineering 2021, 5, 30 30 of 57 ChemEngineering 2021, 5, 30 31 of 57 ChemEngineering 2021, 5, 30 32 of 57 ChemEngineering 2021, 5, 30 33 of 57 ChemEngineering 2021, 5, 30 34 of 57 ChemEngineering 2021, 5, 30 35 of 57 ChemEngineering 2021, 5, 30 36 of 57 ChemEngineering 2021, 5, 30 37 of 57 ChemEngineering 2021, 5, 30 38 of 57 ChemEngineering 2021, 5, 30 39 of 57 ChemEngineering 2021, 5, 30 40 of 57 ChemEngineering 2021, 5, 30 41 of 57 ChemEngineering 2021, 5, 30 42 of 57 ChemEngineering 2021, 5, 30 43 of 57 ChemEngineering 2021, 5, 30 44 of 57 ChemEngineering 2021, 5, 30 45 of 57 ChemEngineering 2021, 5, 30 46 of 57 ChemEngineering 2021, 5, 30 47 of 57 ChemEngineering 2021, 5, 30 48 of 57 ChemEngineering 2021, 5, 30 49 of 57 ChemEngineering 2021, 5, 30 50 of 57 ChemEngineering 2021, 5, 30 51 of 57 ChemEngineering 2021, 5, 30 52 of 57 ChemEngineering 2021, 5, 30 53 of 57 ChemEngineering 2021, 5, 30 54 of 57 ChemEngineering 2021, 5, 30 55 of 57 ChemEngineering 2021, 5, 30 56 of 57 ChemEngineering 2021, 5, 30 57 of 57 17 18 10 76 77 5 6 20 15 16 9 10 11 8 9 10 19 20 21 6 7 8 24 25 6 7 8 19 20 21 7 8 9 29 30 31 5 6 7 31 32 33 16 17 18 4 5 49 2 3 30 31 17 18 19 14 15 16 9 10