```
//Rewriting of the Taperchain programs by Adam Sokolow
//(velocity verlet algorithm taken from original code written by Jan Pfannes)
//20/06/15: MP added the ability to put mass impurities in, and also added driving ability, and resuming functionality.


//Large particle is particle N-1, and small is 0

/*************************************************
READ THIS PLEASE:

This code has been altered so that it will run from a file...
any parameter that you would like to change should be done so
through this built in file interface... otherwise this program
may produce undesired results.. or no results at all.

Also if you have little to no programming experience, keep in mind
that every little thing does matter, and if something gets accidentally
removed, the code may still compile and run, and the results you could get
could be completely believable, showing no obvious sign of error...

Look... but be very careful. (and save a backup)

*************************************************/

#include <cmath>
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string>
#include <sstream>
#include <iomanip>
//#include <malloc.h>
#include <stdio.h>
#include <time.h>

//**********************CONSTANTS/VARIABLES BETWEEN RUNS

 const double PI = 4 * atan(1.0);
 int nptles=20;                    // total number of particles
 int restart=0;                    // resume variable. If set to 1, then simulation will start where it previously left off. Set it in the parameter file.

 int DEFAULTPRECISION = 10;
 bool leftWall=true;
 bool rightWall=true;

 double rho = 7.82; // density of steel grains
//We need two density values now since we are putting a mass impurity in the chain.
 double rhoImpure = 1.00; // density of rubber grains (impurities)
//Parameter for the host chain grain and impurity grain interactions- calculated for steel and rubber
 double D1 = 19.0010245142;
//Parameter for the host chain grains (calculated for stainless steel)
 double D2 = 0.0070490285;
//Parameter for the impurity grain interactions (calculated for rubber-rubber):
 double D3 = 38.48;
 int ImpGr = 19;  //This is the position of the impurity grain- we want it at the center (position 20) of 39 grain chain, but indexing starts at 0.
 double ImpRF = 1.0;  //This is the factor by which the radius is larger of the mass impurity.
 int NumImp = 0;  //This is the number of impurities in the chain. Impurity grains will be ImpGr + 1 --> ImpGr + 1 + NumImp (mp 06/15/15)

 double rlarge = 0.5;           // (radius of large ptle (mm))
 double q = 0;      // (tapering factor (%), q=0: monodisp.)

//changed below in code...

```

```
 double xn = 2.5;                    // (exponent in potential)
 double dt = 0.00001;                // (timestepwidth (musec))
 unsigned long long int nsteps = 500000;//12500000;// (# steps integration loop)

 double timeSpits = .5;  //in the output files, how frequent do we want output...
 double ForceSpits = timeSpits;
 int realspit = int(timeSpits/dt);
 int realFspit = int(ForceSpits/dt);

 int window=0;
 int windowleft=0;
 int windowright=0;

 double AMP, PER;
 double konstForce=0.0; //constant force applied to the first grain.
 double konstForceLast=0.0; //constant force applied to the last grain.
 double konstForceSym=0.0;
 double randForceVal=0.0;
 double randForceValLast=0.0; //random force applied to the last grain.
 double randForceValFL=0.0; //random force applied symmetrically to first and last grains.
 int cursign=-1;
 int cursignSym=-1;
 int cursignR=-1;
 int asymLeft = 0;   //Controls the program flow--- if asymLeft = 0 it means no force applied to left edge of chain-- if == 1 then left asymmetric force.
 int asymRight = 0;
 int symLeftRight = 0; // If symLeftRight == 1, it means that a symmetric driving force is being applied to both ends of the chain.
 int konstLength = 0;  //length of driving intervals (in micro-s) for constant applied force. First is left edge of chain, second one is right edge of chain-- if doing asymmetric driving.
 int konstLengthLast = 0;
 int konstLengthSym = 0; //length of driving intervals (in micro-s) for constant applied force for the symmetrically driven chain.


//************* Function at end of code called  addForcesomethignsomething... upgrades this
 double SmallInitialVelocity = 0.0;       // (initial v /small/ ptle (mm/musec))
 double LargeInitialVelocity = 0.00;//-1*0.149*0.001*1;//-.01      // initial v /large/ ptle (mm/musec))
//************

 double epsilon = 1; // ((1 - restitution factor) all ptles)
 int EP = 100; //change this variable... do not change the one above...

//**********Compression - set the initial overlap between the wall and the large particle
//from this a force balance is made... and all the particles are at rest (except minor vibrations
//due to inprecision in the decimals cause some crazy very small energy noise)
 double InitialOverlapLargeandWall=0.00;
 double loadingForce=0;
//******************

//************dont alter these...
 double ChainLength;
 double ForceInChain;
//************************
 int count;
 double Force=0.0;
 double ForceLast=0.0;
 double ForceFirstLast=0.0;
 double storeForce=Force;
 double storeForceLast=ForceLast;
 double storeForceFL=ForceFirstLast;
//Given any single particle... it has the following:
struct particle
```

```
{
    double relativeLocation;
    double currentVelocity;
    double currentAccel;
    double radius;
    double currentKE;
    double mass;
    double absolutePosition;
};

//now we have an array,(the chain) of the above particles
particle* Chain;

double* deltas;//the overlaps
double* smalla;           //the small-a in the calcs...
double* overbefore;  //stuff for the algorithm
double pot;
double* energy loss;
double otime = 0.0;
double* forceBefore;

//these variables keep track of the max velocities... going in the direction of the initial impulse...
//(so ripples going back that are of larger velocity are ignored...)... this is easily alterable in the code
//the time variables correspond to what time that large velocity occurred.
double maxOverlapLargeParticle=0,maxOverlapSmallParticle=0, maxOLargeTime = 0, maxOSmallTime = 0;
double maxVelocityLargeParticle=0,maxVelocitySmallParticle=0, maxVLargeTime = 0, maxVSmallTime = 0;

std::ofstream KEouts, ETot, Vouts, XRelOuts, ForceAll, AppliedForce, AppliedForceLast, AppliedForceSym;  //file streams


void initializeAllParticles(); //set up the chain
void makeReadmeFile();         //make a readme file
void ChainRelPositions();      //set up more stuff
void CompressChain(double &force, double &length); //apply a compression

void velocityVerletStep();
void computeAccelerations();

double addForceDueToWave(double ttime);     //new addition... cerca 8/30/03
double addForceDueToWaveLast(double ttime);     //new addition... MP 20/06/15
double addForceDueToWaveFirstLast(double ttime);        //new addition... MP 20/06/15

void spitKEs(double ttime);  //spit to the file the KEs of the particles at a given time
void spitVs(double ttime);   //similar but velocities
void spitXs(double ttime);
void spitFapp(double ttime,double force);
void spitFappLast(double ttime,double forceLast);
void spitFappSym(double ttime,double forceSym);
double absolute(double val);
double sign(double val);

void loadFromFile();
void Resume(); //For resuming program from where it left off. MP 11/24/14.
void makeResumeFile();

//more of these output files can be added... but if its not needed, why output and slow the program down?

double recordTotE;

double TimeMin=0.0;
double TimeMax=nsteps*dt;   //We will set this after reading in parameters just to make sure everything is set up correctly. We specify where we want to start in
```

```
                                //the parameter file and the number of time steps, and then the program determines what the maximum time is from this. For now just leave
                                //TimeMin at 0.
std::string fileName=" ";
bool FF=true, KK=true, VV=true, XX=true, TE=true, FA=false, FAL=false, FAS=false;

int topGrain=nptles;
int bottomGrain=1;

struct deltafunc
{
    int grain;
    double time;
    double addedVelocity;
        int howmany;   //MP-- How many times do you want to hit this thing??
    int interval;  //MP-- Interval between hits (in micro-s).
};

deltafunc* deltaVel;
deltafunc* deltaVelcopy;  //MP-- Used for writing to Readme File.
int deltafunclength=0;
int deltafunclengthcopy=0;  //MP-- Used for writing to Readme File.

struct drivingforces
{
    char type;
    double amplitude,frequency,phase,duration;
};

drivingforces* sourceforce;
drivingforces* sourceforceLast;
drivingforces* sourceforceFL;
int sourceforcelength =0;
int sourceforcelengthLast =0;
int sourceforcelengthFL =0;

int control;
double deltaW;
double deltaS;


//=============================================================================================================\\
//                                                                                                  MAIN       \\
PROGRAM                                                                                                          \\
//=============================================================================================================\\

int main()
{
    srand((unsigned)(time(0)));

    std::string FileName;
    std::ostringstream *buffer;

    loadFromFile();

    TimeMax=nsteps*dt+TimeMin;   //Now we are setting the maximum time. If the program is not meant to be restarted, then TimeMin will be set to zero in loadFromFile();.

    ForceSpits = timeSpits;
    realspit = int(timeSpits/dt);
    realFspit = int(ForceSpits/dt);

    {
        std::cout<<"October 13, 2008 Version 1"<<std::endl;
        std::cout<<"number of grains: "<<nptles<<std::endl;
```

```
246             std::cout<<"nsteps: "<<std::setprecision(8)<<nsteps<<std::endl;
247             std::cout<<"dt: "<<dt<<std::endl;
248             std::cout<<"Exponential: "<<xn<<std::endl;
249             std::cout<<"q: "<<q<<std::endl;
250             std::cout<<"epsilon: "<<epsilon<<std::endl;
251 
252             if(KK)
253             {
254                 buffer = new std::ostringstream();
255                 (*buffer) << "KE"<<fileName<<".dat";
256                 FileName = buffer->str();
257 
258                 if(restart == 1){ KEouts.open(FileName.c str(), std::ios::app); }
259                 else{ KEouts.open(FileName.c str()); }
260                 KEouts.precision(DEFAULTPRECISION);
261              }
262 
263              if(TE)
264              {
265                  buffer = new std::ostringstream();
266                  (*buffer)<<"ETot"<<fileName<<".dat";
267                  FileName = buffer->str();
268 
269                  if(restart == 1){ ETot.open(FileName.c str(), std::ios::app); }
270                  else{ ETot.open(FileName.c str()); }
271                  ETot.precision(DEFAULTPRECISION);
272              }
273 
274              if(VV)
275              {
276                  buffer = new std::ostringstream();
277                  (*buffer) << "VEL"<<fileName<< ".dat";
278                  FileName = buffer->str();
279 
280                  if(restart == 1){ Vouts.open(FileName.c str(), std::ios::app); }
281                  else{ Vouts.open(FileName.c str()); }
282                  Vouts.precision(DEFAULTPRECISION);
283              }
284 
285              if(XX)
286              {
287                  buffer = new std::ostringstream();
288                  (*buffer) << "XRel"<<fileName<< ".dat";
289                  FileName = buffer->str();
290 
291                  if(restart == 1){ XRelOuts.open(FileName.c str(), std::ios::app); }
292                  else{ XRelOuts.open(FileName.c str()); }
293                  XRelOuts.precision(DEFAULTPRECISION);
294              }
295 
296              if(FF)
297              {
298                  buffer = new std::ostringstream();
299                  (*buffer) << "Force"<<fileName<< ".dat";
300                  FileName = buffer->str();
301 
302                  if(restart == 1){ ForceAll.open(FileName.c str(), std::ios::app); }
303                  else{ ForceAll.open(FileName.c str()); }
304                  ForceAll.precision(DEFAULTPRECISION);
305              }
306 
307              if(FA)
308              {
309                  buffer = new std::ostringstream();
310                  (*buffer) << "DrivingForceLeft"<<fileName<< ".dat";
311                  FileName = buffer->str();
312 
313                  if(restart == 1){ AppliedForce.open(FileName.c str(), std::ios::app); }
314                  else{ AppliedForce.open(FileName.c_str()); }
```

```
315             AppliedForce.precision(DEFAULTPRECISION);
316         }
317 
318         if(FAL)  //right edge grain asymmetric driving
319         {
320             buffer = new std::ostringstream();
321             (*buffer) << "DrivingForceRight"<<fileName<< ".dat";
322             FileName = buffer->str();
323 
324             if(restart == 1){ AppliedForceLast.open(FileName.c str(),
                std::ios::app); }
325             else{ AppliedForceLast.open(FileName.c str()); }
326             AppliedForceLast.precision(DEFAULTPRECISION);
327         }
328 
329         if(FAS)  //symmetric driving
330         {
331             buffer = new std::ostringstream();
332             (*buffer) << "DrivingForceSymmetric"<<fileName<< ".dat";
333             FileName = buffer->str();
334 
335             if(restart == 1){ AppliedForceSym.open(FileName.c str(), std::ios::app); }
336             else{ AppliedForceSym.open(FileName.c str()); }
337             AppliedForceSym.precision(DEFAULTPRECISION);
338         }
339 
340 
341         initializeAllParticles();
342 
343         ChainRelPositions();
344 
345         makeReadmeFile();
346 
347         computeAccelerations();
348 
349         if(asymLeft==1)  //asymmetric driving on the left edge grain:
350         {
351             Chain[nptles-1].currentAccel-=addForceDueToWave(otime);
352             //std::cout<<Chain[nptles-1].currentAccel<<std::endl;
353         }
354 
355 
356         if(asymRight==1) //asymmetric driving on the right edge grain:
357         {
358             Chain[0].currentAccel+=addForceDueToWaveLast(otime);
359             //std::cout<<Chain[0].currentAccel<<std::endl;
360         }
361 
362 
363         if(symLeftRight==1) //symmetric driving on the both edge grains:
364         {
365             Chain[0].currentAccel+=addForceDueToWaveFirstLast(otime)/Chain[0].mass;
366 
                Chain[nptles-1].currentAccel-=addForceDueToWaveFirstLast(otime)/Chain[npt
                les-1].mass;
367             //std::cout<<Chain[0].currentAccel<<std::endl;
368         }
369 
370 
371 //Now that everything has been initialized as it normally would, it is time to be
    resetting the necessary quantities for resuming where we left off.
372 //MP- 11/24/14. We will call on the "Resume" function, and in the resume function,
    we read in velocities and relative locations. Then we recompute the current grain
373 //accelerations. After this point, we are in a position to perform the simulation:
374 
375         for(unsigned long long int lcv = 0; lcv<nsteps; ++ lcv)
376         {
377             count = lcv;
378             otime = lcv * dt + TimeMin;   //MP-- added this for the restart mode.
```

```
379
380             if(restart==1&&lcv==0){
381                Resume();   //This will reset all of the velocities and relative positions of the grains. After this, we use these positions and velocities to compute their
382                            //Current accelerations. Then we can go through the Velocity Verlet steps.
383                computeAccelerations();
384             }
385             else{
386                velocityVerletStep();
387             }
388
389             for(int lcount=0;lcount<deltafunclength;++lcount)   //Adding multiple delta function velocity perturbations, if specified in the parameter file by user
390             {
391                 control = deltaVel[lcount].howmany;
392                 if(control == 1){
393                    if(deltaVel[lcount].time-otime<dt)
394                    {
395                        Chain[nptles-deltaVel[lcount].grain].currentVelocity+=deltaVel[lcount].addedVelocity ;
396                        deltaVel[lcount].addedVelocity=0;
397                    }
398                 }
399                 if(control == -1){
400                    deltaS = deltaVel[lcount].time;  //Time to start the "delta" perturbation-- MP
401                    deltaW = deltaS + deltaVel[lcount].interval; //Time to end the "delta" perturbation-- MP
402                    if((otime-deltaS<=dt)|((deltaS<otime)&&(otime<=deltaW))){
403                        Chain[nptles-deltaVel[lcount].grain].currentVelocity=deltaVel[lcount].addedVelocity ;
404                        //std::cout<<"applying delta to grain"<<deltaVel[lcount].grain<<std::endl;
405                    }
406                }
407             }
408
409             if(lcv%realspit==0&&otime>=TimeMin&&otime<= TimeMax)
410             {
411                 if(KK||TE)
412                     spitKEs(otime);
413                 if(VV)
414                     spitVs(otime);
415                 if(FF||XX||FA||FAL||FAS)
416                     spitXs(otime);
417             }
418
419             if((-1*Chain[0].currentVelocity)>maxVelocitySmallParticle) //Added 6/7/03
420             {
421                 maxVelocitySmallParticle=(Chain[0].currentVelocity*-1);
422                 maxVSmallTime = otime;
423             } //
424             if((-1*Chain[nptles-1].currentVelocity)>maxVelocityLargeParticle) //
425             {
426                 maxVelocityLargeParticle=(Chain[nptles-1].currentVelocity*-1);
427                 maxVLargeTime = otime;
428             }  //
429
430             if((overbefore[0])>maxOverlapSmallParticle)    //Added 6/7/03
431             {
432                 maxOverlapSmallParticle=(overbefore[0]);
433                 maxOSmallTime = otime-dt;
```

```
434                 } //
435
436                 if(otime<100)
437                     if((overbefore[nptles-1])>maxOverlapLargeParticle)       //
438                     {
439                         maxOverlapLargeParticle=(overbefore[nptles-1]);
440                         maxOLargeTime = otime-dt;
441                     }  //
442             }
443
444             if(VV)
445                 Vouts.close();
446
447             if(KK)
448                 KEouts.close();
449
450             if(TE)
451                 ETot.close();
452
453             if(XX)
454                 XRelOuts.close();
455
456             if(FF)
457                 ForceAll.close();
458
459             if(FA)
460                 AppliedForce.close();
461
462             if(FAL)
463                 AppliedForceLast.close();
464
465             if(FAS)
466                 AppliedForceSym.close();
467
468
469             makeResumeFile();
470
471         }
472
473         return 0;
474     }
475
476     //All of the functions used in the main part of the program:
477
478     void initializeAllParticles()
479     {
480         Chain=(particle *) malloc(nptles*sizeof(particle));
481
482         forceBefore = (double *)malloc((nptles+1)*sizeof(double));
483         deltas= (double*)malloc((nptles+1)*sizeof(double));///the overlaps
484         smalla= (double*)malloc((nptles+1)*sizeof(double));           //the small-a in
        the calcs...
485         overbefore = (double *)malloc((nptles+1)*sizeof(double));  //stuff for the
        algorithm
486         energy loss = (double*)malloc((nptles+1)*sizeof(double));
487
488         Chain[nptles-1].radius = rlarge;
489
490         maxVelocityLargeParticle=0;
491         maxVelocitySmallParticle=0;
492         maxVLargeTime = 0;
493         maxVSmallTime = 0;
494
495         maxOverlapLargeParticle=0;
496         maxOverlapSmallParticle=0;
497         maxOLargeTime = 0;
498         maxOSmallTime = 0;
499
500         for(int lcv = nptles-1;lcv>=0;lcv--)
```

```
501         {
502             Chain[lcv].relativeLocation=0;
503             Chain[lcv].currentVelocity=0;
504             Chain[lcv].currentAccel=0;
505             Chain[lcv].currentKE=0;
506 
507             if(lcv!=nptles-1)
508                 Chain[lcv].radius = ((100.0-q)/100.0)*Chain[lcv+1].radius;
509 
510             Chain[lcv].mass =                                                    ↵
                Chain[lcv].radius*Chain[lcv].radius*Chain[lcv].radius*(4.0/3.0)*PI*rho;
511 
512             deltas[lcv]=0.0;
513             smalla[lcv]=0.0;
514             overbefore[lcv]=0.0;
515             energy loss[lcv]=0.0;
516             forceBefore[lcv] = 0.0;
517             Chain[lcv].absolutePosition = 0.0;
518         }
519 
520     //Now we determine the index (index = grain number-1 since indexing starts at 0) of  ↵
        the first impurity grain:
521         if(nptles % 2 == 0)  //chain has an even number of grains: then the number of    ↵
            impurities will also be an even number.
522         {
523             ImpGr = int(nptles/2 - NumImp/2);
524         }
525         else
526         {
527             ImpGr = int((nptles-1)/2) - int((NumImp-1)/2);
528         }
529 
530     //Now I want to set the radius and mass of the impurity grains:
531         if(NumImp>0)
532         {
533             for (int i3 = 0; i3 < NumImp; i3++)//
534             {
535                 Chain[ImpGr+i3].radius = ImpRF*rlarge;
536                 Chain[ImpGr+i3].mass =                                           ↵
                    Chain[ImpGr+i3].radius*Chain[ImpGr+i3].radius*Chain[ImpGr+i3].radius*(4.0/ ↵
                    3.0)*PI*rhoImpure;
537             }
538         }
539 
540         Chain[nptles-1].currentVelocity = LargeInitialVelocity;
541         Chain[0].currentVelocity = SmallInitialVelocity;
542 
543         deltas[nptles]=0.0;
544         smalla[nptles]=0.0;
545         overbefore[nptles]=0.0;
546         energy loss[nptles]=0.0;
547 
548         forceBefore[nptles]=0.0;
549     }
550 
551     void makeReadmeFile()
552     {
553         std::string FileName;
554         std::ostringstream *buffer;
555         std::ofstream out file;
556 
557         buffer = new std::ostringstream();
558 
559         (*buffer) << "ReadMe"<<fileName<< ".dat";
560         FileName = buffer->str();
561 
562         if(restart == 1){
563             out file.open(FileName.c str(), std::ios::app);     //MP -- if we are in  ↵
                resume mode then we want to append to the files, not write over them.
```

```
564         }
565         else{
566             out_file.open(FileName.c_str());
567         }
568         out_file.precision(DEFAULTPRECISION);
569 
570 
571         if(restart == 1)
572         {out_file<<"Resume mode. Resuming simulation at "<<TimeMin<<"                    micro-s"<<std::endl;}     //MP 11/24/14-- Added in for the resume mode. Output when we are picking up from.
573 
574         out_file<<"Number of Particles: "<<nptles<<std::endl;
575         out_file<<"potential exponential: "<<xn<<std::endl;
576         out_file<<"rho: "<<rho<<" (mg/mm^3)"<<std::endl;
577             if(NumImp>0){
578             out_file<<"rhoImpure: "<<rhoImpure<<" (mg/mm^3)"<<std::endl;
579         }
580             if(NumImp>0){
581             out_file<<"D1, host-impurity interaction: "<<D1<<" (mm^2/kN)"<<std::endl;
582         }
583         out_file<<"D2, host-host interaction: "<<D2<<" (mm^2/kN)"<<std::endl;
584         if(NumImp>1){
585             out_file<<"D3, impurity-impurity interaction: "<<D3<<" (mm^2/kN)"<<std::endl;
586         }
587         out_file<<"Tapering percent: "<<q<<std::endl;
588         out_file<<"Restitution: "<<1-epsilon<<std::endl;
589         out_file<<"Epsilon (1-w): "<<epsilon<<std::endl;
590         out_file<<"Radius of Large Particle: "<<rlarge<<" (mm)"<<std::endl;
591 
592         out_file<<"Radii of all Particles: (mm)"<<std::endl;      //Recall that everything that is done in the program is swapped with the "real chain". i.e. left-->right
593         for(int lcv=0; lcv<nptles;++lcv)
594             out_file<<Chain[nptles-1-lcv].radius<<'\t';
595         out_file<<std::endl;
596 
597         out_file<<"Masses of all Particles: (mg)"<<std::endl;
598         for(int lcv2=0; lcv2<nptles;++lcv2)
599             out_file<<Chain[nptles-1-lcv2].mass<<'\t';
600         out_file<<std::endl;
601 
602         out_file<<"Total force prefactor (a) of all Particles: "<<std::endl;
603         for(int lcv3=0; lcv3<nptles+1;++lcv3)
604             out_file<<smalla[nptles-lcv3]<<'\t';
605         out_file<<std::endl;
606 
607         out_file<<"dt: "<<dt<<"musec"<<std::endl;
608         out_file<<"number of those steps: "<<nsteps<<std::endl;
609         out_file<<"Length of run: "<<dt*nsteps<<" (musec)"<<std::endl;
610 
611         out_file<<"Initial Velocity of Large Particle: "<<-LargeInitialVelocity<<" (mm/musec)"<<std::endl;
612         out_file<<"Initial Velocity of Small Particle: "<<-SmallInitialVelocity<<" (mm/musec)"<<std::endl;
613             if(rightWall)
614                 out_file<<"Right Wall"<<std::endl;
615             else
616                 out_file<<"No Right Wall"<<std::endl;
617             if(leftWall)
618                 out_file<<"Left Wall"<<std::endl;
619             else
620                 out_file<<"No Left Wall"<<std::endl;
621 
622         if(deltafunclengthcopy>0){
623             double dVel = 0.0;
624             double dTime = 0.0;
625             int dGrain = 0;
626             int dNum = 0;
```

```
627             int dInt = 0;
628             for(int lcv4=0; lcv4<deltafunclengthcopy; lcv4++){
629                 dVel = -deltaVelcopy[lcv4].addedVelocity;       //Negative because
                    everything is negated when read into the program.
630                 dGrain = deltaVelcopy[lcv4].grain;
631                 dTime = double(deltaVelcopy[lcv4].time);
632                 dNum = deltaVelcopy[lcv4].howmany;
633                 dInt = deltaVelcopy[lcv4].interval;
634                 if(dNum == -1){
635                     out file<<"Constant velocity perturbation applied to grain "<<
                        dGrain <<" of magnitude "<< dVel <<" (mm/musec), beginning at
                        t="<<dTime<<" musec, for a window of "<< dInt              <<"
                        musec."<<std::endl;
636                 }
637                 else{
638                     if(dNum == 1){
639                         out file<<"Delta velocity perturbation applied to grain "<<
                            dGrain <<" of magnitude "<< dVel <<" (mm/musec) at t="<< dTime
                            <<" musecs."<<std::endl;
640                     }
641                     else{
642                         out file<< dNum <<" delta velocity perturbations applied to
                            grain "<< dGrain <<" of magnitude "<< dVel <<" (mm/musec). First
                            perturbation at t="<< dTime
643                         <<" musecs, and the remainder at "<< dInt <<" musec
                            intervals."<<std::endl;
644                     }
645                 }
646             }
647         }
648
649
650         if(loadingForce>0)
651             out file<<"Initial Loading of Chain "<<loadingForce<<" (kN)"<<std::endl;
652
653
654         if(symLeftRight == 1)
655         {
656             if(konstForceSym > 0.0)
657             {
658                 out file<<"Constant force applied symmetrically to edge grains
                    "<<konstForceSym<<" (kN), window: "<<konstLengthSym<<"
                    (micro-s)."<<std::endl;
659             }
660             for(int c=0;c<sourceforcelengthFL;++c)
661             {
662                 switch(sourceforceFL[c].type)
663                 {
664                     case 's':
665                     {
666                         out file<<"Sine wave applied symmetrically to edge grains,
                            amplitude: "<<sourceforceFL[c].amplitude<<" (kN), angular
                            frequency: "
667                             <<sourceforceFL[c].frequency<<" (rad/micro-s), phase:
                            "<<sourceforceFL[c].phase<<" (rad), window: "
668                             <<sourceforceFL[c].duration<<" musecs."<<std::endl;
669                         break;
670                     }
671                     case 'c':
672                     {
673                         out file<<"Cosine wave applied symmetrically to edge grains,
                            amplitude: "<<sourceforceFL[c].amplitude<<" (kN), angular
                            frequency: "
674                             <<sourceforceFL[c].frequency<<" (rad/micro-s), phase:
                            "<<sourceforceFL[c].phase<<" (rad), window: "
675                             <<sourceforceFL[c].duration<<" musecs."<<std::endl;
676                         break;
677                     }
678                     case 't':
```

```
679                 {
680                     out file<<"Triangle wave applied symmetrically to edge grains,
                        amplitude: "<<sourceforceFL[c].amplitude<<" (kN), angular
                        frequency: "
681                         <<sourceforceFL[c].frequency<<" (rad/micro-s), phase:
                            "<<sourceforceFL[c].phase<<" (rad), window: "
682                         <<sourceforceFL[c].duration<<" musecs."<<std::endl;
683                     break;
684                 }
685                 case 'w':
686                 {
687                     out file<<"Saw wave applied symmetrically to edge grains,
                        amplitude: "<<sourceforceFL[c].amplitude<<" (kN), angular
                        frequency: "
688                         <<sourceforceFL[c].frequency<<" (rad/micro-s), phase:
                            "<<sourceforceFL[c].phase<<" (rad), window: "
689                         <<sourceforceFL[c].duration<<" musecs."<<std::endl;
690                     break;
691                 }
692                 case 'q':
693                 {
694                     out file<<"Square wave applied symmetrically to edge grains,
                        amplitude: "<<sourceforceFL[c].amplitude<<" (kN), angular
                        frequency: "
695                         <<sourceforceFL[c].frequency<<" (rad/micro-s), phase:
                            "<<sourceforceFL[c].phase<<" (rad), window: "
696                         <<sourceforceFL[c].duration<<" musecs."<<std::endl;
697                     break;
698                 }
699                 case 'r':
700                 {
701                     out file<<"Random force applied symmetrically to edge grains,
                        maximum value: "<<sourceforceFL[c].amplitude<<" (kN), minimum
                        value: "
702                         <<sourceforceFL[c].frequency<<" (kN), frequency:
                            "<<sourceforceFL[c].phase<<" (rad/micro-s), window: "
703                         <<sourceforceFL[c].duration<<" musecs."<<std::endl;
704                     break;
705                 }
706             }
707         }
708     }
709
710
711     if(asymLeft == 1)
712     {
713         if(konstForce > 0.0)
714         {
715             out file<<"Constant force applied to left edge only "<<konstForce<<"
                (kN), window: "<<konstLength<<" (micro-s)"<<std::endl;
716         }
717         for(int c=0;c<sourceforcelength;++c)
718         {
719             switch(sourceforce[c].type)
720             {
721                 case 's':
722                 {
723                     out file<<"Sine wave applied to left edge only, amplitude:
                        "<<sourceforce[c].amplitude<<" (kN), angular frequency: "
724                         <<sourceforce[c].frequency<<" (rad/micro-s)"<<"phase:
                            "<<sourceforce[c].phase<<" (rad), window: "
725                         <<sourceforce[c].duration<<" musecs."<<std::endl;
726                     break;
727                 }
728                 case 'c':
729                 {
730                     out file<<"Cosine wave applied to left edge only, amplitude:
                        "<<sourceforce[c].amplitude<<" (kN), angular frequency: "
731                         <<sourceforce[c].frequency<<" (rad/micro-s)"<<"phase:
```

```
                        "<<sourceforce[c].phase<<" (rad), window: "
                        <<sourceforce[c].duration<<" musecs."<<std::endl;
                    break;
                }
                case 't':
                {
                    out file<<"Triangle wave applied to left edge only, amplitude:
                    "<<sourceforce[c].amplitude<<" (kN), angular frequency: "
                        <<sourceforce[c].frequency<<" (rad/micro-s)"<<"phase:
                        "<<sourceforce[c].phase<<" (rad), window: "
                        <<sourceforce[c].duration<<" musecs."<<std::endl;
                    break;
                }
                case 'w':
                {
                    out file<<"Saw wave applied to left edge only, amplitude:
                    "<<sourceforce[c].amplitude<<" (kN), angular frequency: "
                        <<sourceforce[c].frequency<<" (rad/micro-s)"<<"phase:
                        "<<sourceforce[c].phase<<" (rad), window: "
                        <<sourceforce[c].duration<<" musecs."<<std::endl;
                    break;
                }
                case 'q':
                {
                    out file<<"Square wave applied to left edge only, amplitude:
                    "<<sourceforce[c].amplitude<<" (kN), angular frequency: "
                        <<sourceforce[c].frequency<<" (rad/micro-s), "<<"phase:
                        "<<sourceforce[c].phase<<" (rad), window: "
                        <<sourceforce[c].duration<<" musecs."<<std::endl;
                    break;
                }
                case 'r':
                {
                    out file<<"Random force applied to left edge only, maximum
                    value: "<<sourceforce[c].amplitude<<" (kN), minimum value: "
                        <<sourceforce[c].frequency<<" (kN), "<<" frequency:
                        "<<sourceforce[c].phase<<" (rad/micro-s), window: "
                        <<sourceforce[c].duration<<" musecs."<<std::endl;
                    break;
                }
            }
        }
    }


    if(asymRight == 1)
    {
        if(konstForceLast > 0.0)
        {
            out file<<"Constant force applied to right edge only
            "<<konstForceLast<<" (kN), window: "<<konstLengthLast<<"
            (micro-s)"<<std::endl;
        }
        for(int c=0;c<sourceforcelengthLast;++c)
        {
            switch(sourceforceLast[c].type)
            {
                case 's':
                {
                    out file<<"Sine wave applied to right edge only, amplitude:
                    "<<sourceforceLast[c].amplitude<<" (kN), angular frequency: "
                        <<sourceforceLast[c].frequency<<" (rad/micro-s)"<<"phase:
                        "<<sourceforceLast[c].phase<<" (rad), window: "
                        <<sourceforceLast[c].duration<<" musecs."<<std::endl;
                    break;
                }
                case 'c':
                {
                    out_file<<"Cosine wave applied to right edge only, amplitude:
```

```
                    "<<sourceforceLast[c].amplitude<<" (kN), angular frequency: "
788                     <<sourceforceLast[c].frequency<<" (rad/micro-s)"<<"phase:      ↵
                        "<<sourceforceLast[c].phase<<" (rad), window: "
789                     <<sourceforceLast[c].duration<<" musecs."<<std::endl;
790                 break;
791             }
792             case 't':
793             {
794                 out file<<"Triangle wave applied to right edge only, amplitude:    ↵
                    "<<sourceforceLast[c].amplitude<<" (kN), angular frequency: "
795                     <<sourceforceLast[c].frequency<<" (rad/micro-s)"<<"phase:      ↵
                        "<<sourceforceLast[c].phase<<" (rad), window: "
796                     <<sourceforceLast[c].duration<<" musecs."<<std::endl;
797                 break;
798             }
799             case 'w':
800             {
801                 out file<<"Saw wave applied to right edge only, amplitude:         ↵
                    "<<sourceforceLast[c].amplitude<<" (kN), angular frequency: "
802                     <<sourceforceLast[c].frequency<<" (rad/micro-s)"<<"phase:      ↵
                        "<<sourceforceLast[c].phase<<" (rad), window: "
803                     <<sourceforceLast[c].duration<<" musecs."<<std::endl;
804                 break;
805             }
806             case 'q':
807             {
808                 out file<<"Square wave applied to right edge only, amplitude:     ↵
                    "<<sourceforceLast[c].amplitude<<" (kN), angular frequency: "
809                     <<sourceforceLast[c].frequency<<" (rad/micro-s), "<<"phase:    ↵
                        "<<sourceforceLast[c].phase<<" (rad), window: "
810                     <<sourceforceLast[c].duration<<" musecs."<<std::endl;
811                 break;
812             }
813             case 'r':
814             {
815                 out file<<"Random force applied to right edge only, maximum       ↵
                    value: "<<sourceforceLast[c].amplitude<<" (kN), minimum value: "
816                     <<sourceforceLast[c].frequency<<" (kN), "<<" frequency:       ↵
                        "<<sourceforceLast[c].phase<<" (rad/micro-s), window: "
817                     <<sourceforceLast[c].duration<<" musecs."<<std::endl;
818                 break;
819             }
820         }
821     }
822 }
823
824
825 out file<<"Files use these conventions:"<<std::endl;
826 out file<<"Time in mu-sec"<<std::endl;
827 out file<<"Force in kN"<<std::endl;
828 out file<<"Energy in J"<<std::endl;
829 out file<<"Distances in mm"<<std::endl;
830 out file<<"Velocities in mm/musec"<<std::endl;
831 out file<<"Density in mg/mm^3"<<std::endl<<std::endl;
832
833
834 out file.close();
835 delete buffer;
836 }
837
838 void ChainRelPositions()
839 {
840
841     double force = 0;
842     double length = 0;
843
844     if(loadingForce>0)  //if(InitialOverlapLargeandWall!=0) old line, replaced 10/13/08
845     {
846         //Had to change the edge grain- wall parameters. Recall that there is one     ↵
```

```
            more interface than number of grains. Need to also set this for
847         //the grains next to the impurity. Updated June 15/15 MP-- automated.
            Wall/edge grain parameter being set to homogeneous chain.
848         smalla[0] = (1 / (xn * D2)) * (sqrt(Chain[0].radius));
849         smalla[nptles] =  (1 / (xn * D2)) * (sqrt(Chain[nptles-1].radius));//in
            compresschain i set these to 0 if there are no walls
850
851         for (int i = 1; i < nptles; i++)
852             smalla[i] = (1 / (xn * D2)) *
                (sqrt((Chain[i].radius*Chain[i-1].radius)/(Chain[i].radius+Chain[i-1].radi
                us)));
853
854          //Here we will reset the appropriate D values (host-impurity is D1,
             impurity-impurity is D3):
855
856         if(NumImp>0)
857         {
858             smalla[ImpGr] = (1 / (xn * D1)) *
                (sqrt((Chain[ImpGr].radius*Chain[ImpGr-1].radius)/(Chain[ImpGr].radius+Cha
                in[ImpGr-1].radius))); //both set to host-impurity e.g. steel-rubber
                interaction
859             smalla[ImpGr+NumImp] = (1 / (xn * D1)) *
                (sqrt((Chain[ImpGr+NumImp].radius*Chain[ImpGr+NumImp-1].radius)/(Chain[Imp
                Gr+NumImp].radius+Chain[ImpGr+NumImp-1].radius)));
860         }
861
862         if(NumImp>1)
863         {
864             for (int i4 = 1; i4 < NumImp; i4++) //Now reset all impurity-impurity
                (D3) interactions.
865             {
866                 smalla[ImpGr+i4] = (1 / (xn * D3)) *
                    (sqrt((Chain[ImpGr+i4].radius*Chain[ImpGr+i4-1].radius)/(Chain[ImpGr+i
                    4].radius+Chain[ImpGr+i4-1].radius)));
867             }
868         }
869
870         CompressChain(force, length);
871
872     }
873     else
874     {
875         for(int lcv = 0; lcv<nptles; ++lcv)
876         {
877             length = length + 2 * Chain[lcv].radius;
878             Chain[lcv].absolutePosition = length - Chain[lcv].radius;
879         }
880         ChainLength = length;
881
882         if(rightWall)
883             smalla[0] = (1.0 / (xn * D2)) * (sqrt(Chain[0].radius));
884         else
885             smalla[0] = 0;
886         if(leftWall)
887             smalla[nptles] =  (1.0 / (xn* D2)) * (sqrt(Chain[nptles-1].radius));
888         else
889             smalla[nptles] = 0;
890
891         for (int i = 1; i < nptles; i++)
892             smalla[i] = (1.0 / (xn * D2)) *
                (sqrt((Chain[i].radius*Chain[i-1].radius)/(Chain[i].radius+Chain[i-1].radi
                us)));
893
894         //We will reset everything here too since it appears this is done twice!!!
            Homogeneous walls (described by D2). Host-impurity = D1, impurity-impurity =
            D3.
895         if(NumImp>0)
896         {
897             smalla[ImpGr] = (1 / (xn * D1)) *
```

```
            (sqrt((Chain[ImpGr].radius*Chain[ImpGr-1].radius)/(Chain[ImpGr].radius+Cha
            in[ImpGr-1].radius))); //both set to host-impurity e.g. steel-rubber
            interaction
898         smalla[ImpGr+NumImp] = (1 / (xn * D1)) *
            (sqrt((Chain[ImpGr+NumImp].radius*Chain[ImpGr+NumImp-1].radius)/(Chain[Imp
            Gr+NumImp].radius+Chain[ImpGr+NumImp-1].radius)));
899         }
900
901         if(NumImp>1)
902         {
903             for (int i4 = 1; i4 < NumImp; i4++) //Now reset all impurity-impurity
            (D3) interactions.
904             {
905                 smalla[ImpGr+i4] = (1 / (xn * D3)) *
                (sqrt((Chain[ImpGr+i4].radius*Chain[ImpGr+i4-1].radius)/(Chain[ImpGr+i
                4].radius+Chain[ImpGr+i4-1].radius)));
906             }
907         }
908
909     }
910
911     ForceInChain = force;
912     std::cout<<"Force is: "<<force<<"kN"<<std::endl;
913     std::cout<<"Length is: "<<length<<"mm"<<std::endl;
914
915 }
916
917 void CompressChain(double &force, double &length)
918 {
919     std::cout<<"loading chain..."<<std::endl;
920     InitialOverlapLargeandWall = pow((loadingForce/smalla[nptles]*1/xn),(1/(xn-1)));
921
922     deltas[nptles]=InitialOverlapLargeandWall;//set up initial overlap for beginning
        of recursion/loop
923
924     force = xn * smalla[nptles] *pow(deltas[nptles],(xn-1.0));
925
926     for(int lcv = nptles-1; lcv>=0;--lcv)
927     {
928         deltas[lcv] = pow((force/smalla[lcv]*1/xn),(1/(xn-1)));
929     }
930
931     if(!leftWall)
932     {
933         deltas[nptles]=0;
934         smalla[nptles]=0;
935     } //added these ifs in b/c if there is no wall but the chain is precompressed
936
937     if(!rightWall)          //these overlaps must be 0, ACS 10/13/08
938     {
939         deltas[0]=0;
940         smalla[0]=0;
941     }
942
943     for(int lcv2 = 0; lcv2<nptles; ++lcv2)
944     {
945         length = length + 2*Chain[lcv2].radius - deltas[lcv2];
946         Chain[lcv2].absolutePosition = length - Chain[lcv2].radius;
947     }
948
949     length = length - deltas[nptles];
950
951     ChainLength = length;
952
953     for(int lcv3=0; lcv3<=nptles; ++lcv3)
954         overbefore[lcv3] = deltas[lcv3];
955
956 }
957
```

```
958
959  void velocityVerletStep()
960  {
961      for (int j = 0; j < nptles; j++)
962      {
963
964          Chain[j].relativeLocation += Chain[j].currentVelocity * dt + 0.5 *
         Chain[j].currentAccel * dt*dt;
965          Chain[j].currentVelocity += 0.5 * Chain[j].currentAccel * dt;
966      }
967
968      computeAccelerations();
969
970      //Now we have to control the program flow and apply forces at edges appropriately.
971
972      if(symLeftRight == 1)
973      {
974          if(konstForceSym > 0.0)    //If there is a nonzero constant force being
         applied symmetrically to the chain edges
975          {
976              window = otime - konstLengthSym;
977              if(window > 0)  //means we are no longer in the window of applying the
             force symmetrically to the chain edges
978              {
979                  konstForceSym = 0.0;
980              }
981              else   //means we are applying a constant force since we are inside the
             window.
982              {
983                  Chain[0].currentAccel+=addForceDueToWaveFirstLast(otime)/Chain[0].mass;
984
                     Chain[nptles-1].currentAccel-=addForceDueToWaveFirstLast(otime)/Chain[
                     nptles-1].mass;
985              }
986          }
987          else      //there may be another type of driving to apply, which is not
         constant force over a window. We also might modify this so we can specify
         driving time
988          {         //for a non-constant force. Will be done as above-- trivial. MP
         06/20/15
989              Chain[0].currentAccel+=addForceDueToWaveFirstLast(otime)/Chain[0].mass;
990
                 Chain[nptles-1].currentAccel-=addForceDueToWaveFirstLast(otime)/Chain[nptl
                 es-1].mass;
991          }
992      }
993      else
994      {
995          if(asymLeft == 1)
996          {
997              if(konstForce > 0.0)    //If there is a nonzero constant force being
             applied asymmetrically to the left edge of the chain.
998              {
999                  windowleft = otime - konstLength;
1000                 if(windowleft > 0)  //means we are no longer in the window of
                 applying the force
1001                 {
1002                     konstForce = 0.0;
1003                 }
1004                 else   //means we are applying a constant force to the left edge
                 since we are inside the window.
1005                 {
1006                     Chain[nptles-1].currentAccel-=addForceDueToWave(otime);
1007                 }
1008             }
1009             else      //there may be another type of driving to apply, which is not
             constant force over a window. We also might modify this so we can
             specify driving time
1010             {         //for a non-constant force. Will be done as above-- trivial.
```

```
            MP 06/20/15
1011            Chain[nptles-1].currentAccel-=addForceDueToWave(otime);
1012            }
1013        }
1014
1015        if(asymRight == 1)    //applying an asymmetric driving force to the right edge of the chain. This can be done alone or in combination with asymmetric left.
1016        {
1017            if(konstForceLast > 0.0)    //If there is a nonzero constant force being applied asymmetrically to the right edge of the chain.
1018            {
1019                windowright = otime - konstLengthLast;
1020                if(windowright > 0)  //means we are no longer in the window of applying the force
1021                {
1022                    konstForceLast = 0.0;
1023                }
1024                else   //means we are applying a constant force to the right edge since we are inside the window.
1025                {
1026                    Chain[0].currentAccel+=addForceDueToWaveLast(otime);
1027                }
1028            }
1029            else      //there may be another type of driving to apply, which is not constant force over a window. We also might modify this so we can specify driving time
1030            {         //for a non-constant force. Will be done as above-- trivial.
                MP 06/20/15
1031                Chain[0].currentAccel+=addForceDueToWaveLast(otime);
1032            }
1033        }
1034    }
1035
1036    for (int j2 = 0; j2 < nptles; j2++)
1037        Chain[j2].currentVelocity += 0.5 * Chain[j2].currentAccel * dt;
1038
1039  }
1040
1041
1042  void computeAccelerations()
1043  {
1044
1045      double over,overnm1,forceBetw,forceFactor,moved way,energy comp,energy deco,forceSmall,forceLarge;
1046
1047      for (int i = 0; i < nptles; i++) // zeroing all acc in every call
1048          Chain[i].currentAccel = 0.0;
1049
1050      pot = 0.0;
1051
1052      /******* potential/force between neighboring ptles *************/
1053      for (int i2 = 0; i2 < nptles-1; i2++)//
1054      {
1055          if ((-1*Chain[i2].relativeLocation + Chain[i2+1].relativeLocation - deltas[i2+1])<=0) //swap del    had -delta[i2]>
1056          {//added in deltas 5/27                    // only when overlap
1057              over = Chain[i2].relativeLocation + deltas[i2+1] - Chain[i2+1].relativeLocation;  //added in deltas 5/27 swap del
1058              overnm1 = pow(over, (xn - 1.0));
1059              pot += over * overnm1 * smalla[i2+1];
1060              forceBetw = smalla[i2+1] * xn * overnm1;
1061
1062              if (overbefore[i2+1] < over)            // when compressing
1063                  forceFactor = 1.0;
1064              else forceFactor = epsilon;         // when decompressing
1065
1066              forceBetw *= forceFactor;
```

```
1067
1068            Chain[i2].currentAccel -= forceBetw;           // sign(-): towards
                smaller x                swapped
1069            Chain[i2+1].currentAccel += forceBetw;         // sign(+): towards
                larger x      swapped
1070
1071            forceBefore[i2+1] = forceBetw;
1072
1073            /****** calculate energy loss between ptles ********/
1074            if (overbefore[i2+1] < over)
1075            {                                                  // compressing
1076                    moved way = over - overbefore[i2+1];
1077                    energy comp = forceBetw * moved way;
1078                energy loss[i2+1] += energy comp;   // whenever loading
1079            }
1080            if (overbefore[i2+1] > over)
1081            {                        // deco.; (don't mind '='-case)
1082                moved way = overbefore[i2+1] - over;
1083                energy deco = forceBetw * moved way;
1084                    energy loss[i2+1] -= energy deco;   // whenever unloading
1085            }
1086
1087          /*************************************************/
1088
1089            overbefore[i2+1] = over;          // update for next timestep
1090        }
1091        else
1092        {
1093            overbefore[i2+1] = 0.0;//deltas[i2+1];//switched from 0.0 to
                deltas[i2+1] 7/16/03
1094            //pot += deltas[i2+1]   * pow(deltas[i2+1], (xn - 1.0)) *
                smalla[i2+1];      //added 7/16/03
1095        }           // reset when no overlap
1096    }
1097
1098    /** pot./force between fixed wall (small, x=0) <-> small ptle **/   //THIS IS
        REALLY LARGE PARTICLE?
1099    if ((-1*Chain[0].relativeLocation + deltas[0])>=0 )
1100    {    //swap deltas
1101        over = -1*Chain[0].relativeLocation + deltas[0];        //swap deltas
1102        overnm1 = pow(over, (xn - 1.0));
1103        pot += over * overnm1 * smalla[0];
1104        forceSmall = smalla[0] * xn * overnm1;
1105
1106        if (overbefore[0] < over)
1107            forceFactor = 1.0;
1108        else forceFactor = epsilon;
1109
1110        forceSmall *= forceFactor;
1111
1112        if(rightWall)
1113            Chain[0].currentAccel += forceSmall;    //**********************swap -+
1114
1115        forceBefore[0] = forceSmall;
1116
1117        /****** calculate energy loss at wall (small) ********/
1118
1119        if (overbefore[0] < over)
1120        {                        // compressing
1121            moved way = over - overbefore[0];
1122            energy comp = forceSmall * moved way;
1123            energy loss[0] += energy comp;
1124        }
1125
1126        if (overbefore[0] > over)
1127        {
1128            moved way = overbefore[0] - over;
1129            energy deco = forceSmall * moved way;
1130            energy_loss[0] -= energy_deco;
```

```
1131            }
1132
1133            /*****************************************************/
1134
1135            overbefore[0] = over;
1136        }
1137        else
1138        {
1139            overbefore[0] = 0.0;//deltas[0];//0.0; previously just 0.0  7/16
1140            //pot += deltas[0]  * pow(deltas[0], (xn - 1.0)) * smalla[0]; //added 7/16
1141        }
1142
1143        /*** pot./force between fixed wall (large) <-> large ptle ******/
1144        if ((1*Chain[nptles-1].relativeLocation + deltas[nptles] )>=0)
1145        {                               //Added deltas 5/27 ACS  //swap delta
1146            over = 1*Chain[nptles-1].relativeLocation + deltas[nptles]; //Added 5/27
            ACS    //swap delta
1147            overnm1 = pow(over, (xn - 1.0));
1148            pot += over * overnm1 * smalla[nptles];
1149            forceLarge = smalla[nptles] * xn * overnm1;
1150
1151            if (overbefore[nptles] < over)
1152                forceFactor = 1.0;
1153            else forceFactor = epsilon;
1154
1155            forceLarge *= forceFactor;
1156
1157            forceBefore[nptles] = forceLarge;
1158
1159            if(leftWall)
1160                Chain[nptles-1].currentAccel -= forceLarge; //**************swap +-
1161            //else
1162                //Chain[nptles-1].currentAccel -= 0;
1163
1164            /****** calculate energy loss at wall (large) ********/
1165            if (overbefore[nptles] < over)
1166            {                               // compressing
1167                moved way = over - overbefore[nptles];
1168                energy comp = forceLarge * moved way;
1169                energy loss[nptles] += energy comp;
1170            }
1171            if (overbefore[nptles] > over)
1172            {
1173                moved way = overbefore[nptles] - over;
1174                energy deco = forceLarge * moved way;
1175                energy loss[nptles] -= energy deco;
1176            }
1177            /*****************************************************/
1178
1179            overbefore[nptles] = over;
1180        }
1181        else
1182        {
1183            overbefore[nptles] = 0.0;//deltas[nptles];//0.0; changed by ACS 7/16
1184            //pot += deltas[nptles] * pow(deltas[nptles], (xn - 1.0)) * smalla[nptles];
            //added 7/16
1185        }
1186
1187        /***** real dim of acc: division by mass **********/
1188        for (int i3 = 0; i3 < nptles; i3++)
1189            Chain[i3].currentAccel = Chain[i3].currentAccel / Chain[i3].mass;
1190
1191    }
1192
1193
1194    void spitKEs(double ttime)
1195    {
1196        double totKE=0;
1197        double tV;
```

```
1198        if(KK)
1199            KEouts<<(ttime);
1200        if(TE)
1201            ETot<<(ttime);
1202
1203
1204        for(int lcv = 0; lcv<nptles;++lcv)
1205        {
1206            tV = Chain[lcv].currentVelocity;
1207            totKE = totKE +  (tV*tV*0.5*Chain[lcv].mass);
1208        }
1209
1210
1211        if(KK)
1212            for(int lcv = bottomGrain; lcv<=topGrain;++lcv)
1213            {
1214                tV = Chain[nptles-lcv].currentVelocity;
1215                KEouts<<'\t'<<(tV*tV*0.5*Chain[nptles-lcv].mass);
1216            }
1217
1218        if(KK)
1219            KEouts<<std::endl;
1220
1221
1222        recordTotE = pot+totKE;
1223
1224        if(TE)
1225            ETot<<'\t'<<pot<<'\t'<<totKE<<'\t'<<(pot+totKE)<<std::endl;
1226
1227    }
1228
1229
1230    void spitVs(double ttime)
1231    {
1232
1233        double tV;
1234        if(VV)
1235            Vouts<<(ttime);
1236
1237        for(int lcv = bottomGrain; lcv<=topGrain;++lcv)
1238        {
1239            tV = -Chain[nptles-lcv].currentVelocity;
1240
1241            if(VV)
1242                Vouts<<'\t'<<(tV);
1243        }
1244
1245        if(VV)
1246            Vouts<<std::endl;
1247
1248    }
1249
1250
1251    void spitXs(double ttime)
1252    {
1253        double tX;
1254
1255        if(FA)
1256            spitFapp(ttime,storeForce);
1257        if(FAL)
1258            spitFappLast(ttime,storeForceLast);
1259        //ForceFile<<(ttime);
1260        if(FAS)
1261            spitFappSym(ttime,storeForceFL);////***
1262        if(XX)
1263            XRelOuts<<(ttime);
1264        if(FF)
1265            ForceAll<<(ttime);
1266
```

```
1267        if(FF&&leftWall&&bottomGrain==1)
1268            ForceAll<<'\t'<<forceBefore[nptles];
1269        else if(FF&&bottomGrain!=1)
1270            ForceAll<<'\t'<<forceBefore[nptles-bottomGrain+1];
1271
1272        for(int lcv = bottomGrain; lcv<topGrain;++lcv)
1273        {
1274            tX = Chain[nptles-lcv].relativeLocation;
1275            if(FF)              //if(bottomGrain!=1||leftWall==true)
1276                ForceAll<<'\t'<<forceBefore[nptles-lcv];//(smalla[lcv] * xn *
                    pow(overbefore[lcv], (xn - 1.0))* epsilon);
1277
1278            if(XX)
1279                XRelOuts<<'\t'<<-(tX);
1280        }
1281
1282    //right wall corresponds to grain 0
1283
1284        if(FF&&rightWall)
1285            ForceAll<<'\t'<<forceBefore[nptles-topGrain];//(smalla[nptles] * xn *
                pow(overbefore[nptles], (xn - 1.0))* epsilon);
1286
1287        if(XX)
1288            XRelOuts<<'\t'<<-Chain[nptles-topGrain].relativeLocation;
1289
1290        if(FF)
1291            ForceAll<<std::endl;
1292
1293        if(XX)
1294            XRelOuts<<std::endl;
1295
1296    }
1297
1298
1299    void spitFapp(double ttime,double force) //Force applied to left grain (mp 06/15/15)
1300    {
1301        AppliedForce<<(ttime)<<'\t'<<force<<std::endl;
1302    }
1303
1304
1305    void spitFappLast(double ttime,double forceLast) //Force applied to right grain (mp
    06/15/15)
1306    {
1307        AppliedForceLast<<(ttime)<<'\t'<<forceLast<<std::endl;
1308    }
1309
1310
1311    void spitFappSym(double ttime,double forceSym) //Force applied symmetrically to
    left/right edge grains.
1312    {
1313        AppliedForceSym<<(ttime)<<'\t'<<forceSym<<std::endl;
1314    }
1315
1316
1317
1318    //This function adds an acceleration to the large particle as you determine you want
    to do it..
1319    //it is given a force you provide, and simply divides out the mass of the particle...
1320    //that could easily be changed...
1321
1322    double addForceDueToWave(double ttime)
1323    {
1324        double Fduration;
1325
1326        Force = konstForce;
1327
1328        for(int c=0;c<sourceforcelength;++c)
1329        {
1330            Fduration = sourceforce[c].duration;
```

```
1331
1332            if(ttime<=Fduration){
1333                                                                                                    ↵
                    //std::cout<<sourceforce[c].type<<'\t'<<sourceforce[c].amplitude<<'\t'<<so ↵
                    urceforce[c].frequency<<'\t'<<sourceforce[c].phase<<'\t'<<std::endl;
1334                switch(sourceforce[c].type)
1335                {
1336                    case 's':
1337                    {
1338                                                                                                    ↵
                        Force+=sourceforce[c].amplitude*sin(sourceforce[c].frequency*ttime ↵
                         -sourceforce[c].phase);
1339                        break;
1340                    }
1341                    case 'c':
1342                    {
1343                                                                                                    ↵
                        Force+=sourceforce[c].amplitude*cos(sourceforce[c].frequency*ttime ↵
                         -sourceforce[c].phase);
1344                        break;
1345                    }
1346                    case 't':
1347                    {
1348                                                                                                    ↵
                        Force+=sourceforce[c].amplitude*(absolute(4.0*((ttime-sourceforce[ ↵
                        c].phase)*sourceforce[c].frequency/(2.0*PI)
1349                                                                                                    ↵
                                -floor(sourceforce[c].frequency/(2.0*PI)*(ttime-sourceforc ↵
                                e[c].phase)+0.5)))-1.0);
1350                        break;
1351                    }
1352                    case 'w':
1353                    {
1354                                                                                                    ↵
                        Force+=sourceforce[c].amplitude*2.0*((ttime-sourceforce[c].phase)* ↵
                        sourceforce[c].frequency/(2.0*PI)
1355                                                                                                    ↵
                                -floor(sourceforce[c].frequency/(2.0*PI)*(ttime-sourceforc ↵
                                e[c].phase)+0.5));
1356                        break;
1357                    }
1358                    case 'q':
1359                    {
1360                                                                                                    ↵
                        Force+=sourceforce[c].amplitude*sign(sin(sourceforce[c].frequency* ↵
                        ttime-sourceforce[c].phase));
1361                        break;
1362                    }
1363                    case 'r':
1364                    {
1365                        if(cursign==sign(sin(sourceforce[c].phase/2*ttime-PI))) //We       ↵
                        want to generate a random number between -1 and +1;
1366                        {                               //Previously this was set to          ↵
                        rand()/(RAND MAX + 1) which actually gives a number between -1     ↵
                        and 0.
1367                            randForceVal =                                                      ↵
                            double(2.0*rand()/(double)(RAND MAX+1.0)-1.0)*(sourceforce[c]. ↵
                            amplitude-sourceforce[c].frequency)/2.0
1368                                                                                                ↵
                                    +(sourceforce[c].amplitude+sourceforce[c].frequency)/2 ↵
                                    .0;
1369                            cursign=cursign*-1;
1370                        }
1371                        Force+=randForceVal;
1372                        break;
1373                    }
1374                    default :
1375                        std::cout<<"unknown force"<<std::endl;break;
1376                }
```

```
1377            }
1378        }
1379        /*if(randomForce)
1380        {
1381        Force += double(rand()%10)/10
1382        }*/
1383            //Force = 0;//+sin(ttime*PI/100);
1384            //initial force 0...
1385        //if(ttime<358)                     //while time is under 50... apply the following force
1386        //Force = (AMP/1000)*pow((sin(pow(sin(ttime/10000),2)*(ttime*(PI*(PER/100))))),2);
1387        //Force = 0.5*AMP;
1388        //else
1389        //Force = 0;    //otherwise...its 0...
1390        //Force = 0.05*cos(ttime*4)+0.3*sin(ttime/10)+5*sin(ttime/30);
1391        //Force = 4 + sin(ttime) + 2*cos(ttime*PI) + 3*sin(ttime/2);
1392        //This function could easily be applied cotinuously... by removing the if statement
1393        //it could be switched up any way you want...basically you have the freedom to define
1394        //a piece wise function, and they can be anything you want, ... so its kinda nice...
1395
1396        //just be careful not to spit too much energy into the system, otherwise you'll
1397        //get some results that may not be accurate due to over compression of the spheres...
1398
1399        storeForce= Force;
1400        //if((int(ttime/dt) % int(ForceSpits/dt))==0)
1401        /*if((count)%realFspit == 0)
1402        {
1403            spitFapp(ttime,(Force));
1404            //Force = double(rand()%10)/10;
1405        }*/
1406
1407        return Force/Chain[nptles-1].mass;
1408    }
1409
1410    double addForceDueToWaveLast(double ttime)   //Same as structure above, but for the last grain in the chain.
1411    {
1412        double Fduration;
1413
1414        ForceLast = konstForceLast;
1415
1416        for(int cl=0;cl<sourceforcelengthLast;++cl)
1417        {
1418            Fduration = sourceforceLast[cl].duration;
1419
1420            if(ttime<=Fduration){
1421
1422                switch(sourceforceLast[cl].type)
1423                {
1424                    case 's':
1425                    {
1426                        ForceLast+=sourceforceLast[cl].amplitude*sin(sourceforceLast[cl].frequency*ttime -sourceforceLast[cl].phase);
1427                        break;
1428                    }
1429                    case 'c':
1430                    {
1431                        ForceLast+=sourceforceLast[cl].amplitude*cos(sourceforceLast[cl].frequency*ttime -sourceforceLast[cl].phase);
1432                        break;
1433                    }
1434                    case 't':
1435                    {
1436
```

```
                    ForceLast+=sourceforceLast[cl].amplitude*(absolute(4.0*((ttime-sou
                    rceforceLast[cl].phase)*sourceforceLast[cl].frequency/(2.0*PI)
1437                        -floor(sourceforceLast[cl].frequency/
                    (2.0*PI)*(ttime-sourceforceLast[cl].phase)+0.5)))-1.0);
1438                    break;
1439                }
1440                case 'w':
1441                {
1442
                    ForceLast+=sourceforceLast[cl].amplitude*2.0*((ttime-sourceforceLa
                    st[cl].phase)*sourceforceLast[cl].frequency/(2.0*PI)
1443
                            -floor(sourceforceLast[cl].frequency/(2.0*PI)*(ttime-sourc
                            eforceLast[cl].phase)+0.5));
1444                    break;
1445                }
1446                case 'q':
1447                {
1448
                    ForceLast+=sourceforceLast[cl].amplitude*sign(sin(sourceforceLast[
                    cl].frequency*ttime-sourceforceLast[cl].phase));
1449                    break;
1450                }
1451                case 'r':
1452                {
1453                    if(cursignR==sign(sin(sourceforceLast[cl].phase/2*ttime-PI)))
1454                    {
1455                        randForceValLast =
                        double(2.0*rand()/(double)(RAND_MAX+1.0)-1.0)*(sourceforceLast
                        [cl].amplitude-sourceforceLast[cl].frequency)/2.0
1456
                                    +(sourceforceLast[cl].amplitude+sourceforceLast[cl
                                    ].frequency)/2.0;
1457                        cursignR=cursignR*-1;
1458                    }
1459                    ForceLast+=randForceValLast;
1460                    break;
1461                }
1462                default :
1463                    std::cout<<"unknown force"<<std::endl;break;
1464            }
1465        }
1466    }
1467
1468    storeForceLast= ForceLast;
1469    return ForceLast/Chain[0].mass;
1470 }
1471
1472
1473 double addForceDueToWaveFirstLast(double ttime)   //Same as structure above, but for
    the last grain in the chain.
1474 {
1475     double Fduration;
1476
1477     ForceFirstLast = konstForceSym;
1478
1479     for(int cfl=0;cfl<sourceforcelengthFL;++cfl)
1480     {
1481         Fduration = sourceforceFL[cfl].duration;
1482
1483         if(ttime<=Fduration){
1484             switch(sourceforceFL[cfl].type)
1485             {
1486                 case 's':
1487                 {
1488
                    ForceFirstLast+=sourceforceFL[cfl].amplitude*sin(sourceforceFL[cfl
                    ].frequency*ttime -sourceforceFL[cfl].phase);
1489                    break;
```

```
1490                 }
1491                 case 'c':
1492                 {
1493                                                                                 ↵
                         ForceFirstLast+=sourceforceFL[cfl].amplitude*cos(sourceforceFL[cfl ↵
                         ].frequency*ttime -sourceforceFL[cfl].phase);
1494                     break;
1495                 }
1496                 case 't':
1497                 {
1498                                                                                 ↵
                         ForceFirstLast+=sourceforceFL[cfl].amplitude*(absolute(4.0*((ttime ↵
                         -sourceforceFL[cfl].phase)*sourceforceFL[cfl].frequency/(2.0*PI)
1499                                                                                 ↵
                                 -floor(sourceforceFL[cfl].frequency/(2.0*PI)*(ttime-source ↵
                                 forceFL[cfl].phase)+0.5)))-1.0);
1500                     break;
1501                 }
1502                 case 'w':
1503                 {
1504                                                                                 ↵
                         ForceFirstLast+=sourceforceFL[cfl].amplitude*2.0*((ttime-sourcefor ↵
                         ceFL[cfl].phase)*sourceforceFL[cfl].frequency/(2.0*PI)
1505                                                                                 ↵
                                 -floor(sourceforceFL[cfl].frequency/(2.0*PI)*(ttime-source ↵
                                 forceFL[cfl].phase)+0.5));
1506                     break;
1507                 }
1508                 case 'q':
1509                 {
1510                                                                                 ↵
                         ForceFirstLast+=sourceforceFL[cfl].amplitude*sign(sin(sourceforceF ↵
                         L[cfl].frequency*ttime-sourceforceFL[cfl].phase));
1511                     break;
1512                 }
1513                 case 'r':
1514                 {
1515                     if(cursignSym==sign(sin(sourceforceFL[cfl].phase/2*ttime-PI)))
1516                     {
1517                         randForceValFL =                                        ↵
                         double(2.0*rand()/(double)(RAND_MAX+1.0)-1.0)*(sourceforceFL[c ↵
                         fl].amplitude-sourceforceFL[cfl].frequency)/2.0
1518                                                                                 ↵
                                     +(sourceforceFL[cfl].amplitude+sourceforceFL[cfl]. ↵
                                     frequency)/2.0;
1519                         cursignSym=cursignSym*-1;
1520                     }
1521                         ForceFirstLast+=randForceValFL;
1522                     break;
1523                 }
1524                 default :
1525                     std::cout<<"unknown force"<<std::endl;break;
1526             }
1527         }
1528     }
1529
1530     storeForceFL= ForceFirstLast;
1531     return ForceFirstLast;  ///Check this line
1532 }
1533
1534
1535 double absolute(double val)
1536 {
1537     if(val<0)
1538         return val*-1;
1539     else
1540         return val;
1541 }
1542
```

```
1543
1544  double sign(double val)
1545  {
1546      if(val<0)
1547          return -1;
1548      else
1549          return 1;
1550  }
1551
1552
1553  //Adding a new function in here called resume. We will open a "Resume.txt" file, and it will have a list of relative positions, followed by a blank line, followed by a
1554  //list of velocities. We read these in and place them appropriately in the data structures.----MP 11/24/14.
1555
1556  void Resume()
1557  {
1558      std::ifstream in file2("resume.txt");  //Open resume.txt for processing.
1559      char data[50];                        //We will read the data in as a set of characters, then convert the string into a numeric format below.
1560      double DataRead;
1561      int Rcount = 0;
1562
1563      while(!in file2.eof()&&!in file2.fail()){
1564          Rcount = Rcount + 1;
1565          for (int i = bottomGrain; i <= topGrain; i++){
1566              if ( i != nptles) {
1567                  in file2.getline(data, 50, '\t');  //If it is not the last particle, then the data is separated by a tab.
1568              }
1569              else {in file2.getline(data, 50); }  //If it is the last particle, then it will not be tab-delimited. We will have an end of line character, which is default.
1570              DataRead = strtod(data,NULL);        //Now we convert the data into a double. Then we will place it appropriately in the arrays. If Rcount = 1, then we need to
1571              if(Rcount == 1){                      //fill in the relative position array, and if Rcount = 2, then we shall fill in the current velocities.
1572                  Chain[nptles-i].relativeLocation=-DataRead;
1573              }   //Recall, we are filling the array backwards, and we also have to negate everything.
1574              if(Rcount == 2){
1575                  Chain[nptles-i].currentVelocity=-DataRead;
1576              }
1577          }
1578      }
1579      in file2.close();
1580  }
1581
1582
1583  void loadFromFile()
1584  {
1585      bool newTimeRange=false;
1586      bool Specified=false;
1587      bool newGrains=false;
1588      bool grainsSpecified=false;
1589      std::string param;
1590      char cc;
1591      char ccl;
1592      char ccfl;
1593      double parval;
1594      double Fdur;
1595
1596      std::ifstream in file("parameters.txt");
1597
1598      in file>>param;
1599
1600      while(!in file.eof()&&!in file.fail())
1601      {
```

```
1602            for(unsigned int i =0;i<param.length();++i)
1603                param[i]=tolower(param[i]);
1604
1605                                                                                                                    ↵
            if(param!="files:"&&param!="filename:"&&param!="addforce:"&&param!="addforcela ↵
            st:"&&param!="addforcefirstlast:")
1606                in file>>parval;
1607
1608            if(param=="n:")
1609            {
1610                nptles=int(parval);
1611                newGrains=true;
1612            }
1613            if(param=="dt:")
1614            {
1615                dt=parval;
1616                newTimeRange = true;
1617            }
1618            //if(param=="firstimpure:")       //Added June 15 2015 MP-- grain number of    ↵
            the first impurity--- Changed so the program has the capability to determine   ↵
            the position on it own..
1619            //{
1620            //  ImpGR=int(parval-1);
1621            //}
1622            if(param=="numimpure:")         //Added June 15 2015 MP-- number of mass      ↵
            impurities
1623            {
1624                NumImp=int(parval);
1625            }
1626            if(param=="rhoimp:")         //Added June 15 2015 MP-- density of mass        ↵
            impurities
1627            {
1628                rhoImpure=parval;
1629            }
1630            if(param=="rhofac:")         //Added June 15 2015 MP-- radius factor of mass  ↵
            impurities (R imp = rhoFac*rlarge)
1631            {
1632                ImpRF=parval;
1633            }
1634            if(param=="nsteps:")
1635            {
1636                nsteps= (unsigned long long int) (parval); // ( ) added by YT. 06202009
1637                newTimeRange = true;
1638            }
1639            if(param=="restart:")
1640            {
1641                restart = (unsigned int) (parval);  //If restart = 1, then we will go     ↵
                into a special mode to restart from where we left off.
1642            }
1643            if(param=="q:")
1644                q=parval;
1645            if(param=="w:")
1646                epsilon=1-parval;//std::cout<<"wfound"<<parval<<std::endl;
1647            if(param=="rho:")
1648                rho=parval;
1649            if(param=="precision:")
1650                DEFAULTPRECISION=int(parval);
1651            if(param =="exponential:")
1652                xn = parval;
1653            if(param =="preload:")
1654                loadingForce = parval;//kN
1655
1656
1657            if(param=="d:")
1658                D2=parval;     //D value for host chain grain-grain interaction (and      ↵
                currently grain-wall interaction-- this can be changed).
1659            if(param=="dhi:")  //D value for host chain and impurity grain interaction
1660                D1=parval;
1661            if(param=="dii:")  //D value for impurity-impurity interaction
```

```
1662            D3=parval;
1663        if(param=="wall:")
1664        {
1665            if(int(parval)==0)
1666            {
1667                leftWall=rightWall=false;
1668            }
1669            if(int(parval)==11)
1670            {
1671                leftWall=rightWall=true;
1672            }
1673            if(int(parval)==10)
1674            {
1675                leftWall=true;
1676                rightWall = false;
1677            }
1678            if(int(parval)==1)
1679            {
1680                leftWall=false;
1681                rightWall = true;
1682            }
1683        //  std::cout<<"Wallfound"<<parval<<std::endl;
1684        }
1685
1686        if(param=="rlarge:")
1687            rlarge = parval;
1688        if(param=="timespits:")
1689            timeSpits = parval;
1690        if(param=="smallinitv:")
1691            SmallInitialVelocity = -1*parval;
1692        if(param=="largeinitv:")
1693            LargeInitialVelocity = -1*parval;
1694
1695        if(param=="deltav:")
1696        {
1697            deltafunc* temp;
1698            temp=(deltafunc*)malloc((deltafunclength+1)*sizeof(deltafunc));
1699
1700            for(int c=0;c<deltafunclength;++c)
1701            {
1702                temp[c].addedVelocity = deltaVel[c].addedVelocity;
1703                temp[c].grain = deltaVel[c].grain;
1704                temp[c].time = deltaVel[c].time;
1705                temp[c].howmany = deltaVel[c].howmany;     //MP-- added in so we can
                    specify a window for delta velocity driving
1706                temp[c].interval = deltaVel[c].interval;
1707            }
1708
1709            temp[deltafunclength].grain = int(parval);
1710            in file>>parval;
1711            temp[deltafunclength].time = (parval);
1712            in file>>parval;
1713            temp[deltafunclength].addedVelocity = -(parval);
1714            in file>>parval;
1715                        temp[deltafunclength].howmany = int(parval);
1716            in file>>parval;
1717                        temp[deltafunclength].interval = int(parval);
1718
1719            deltafunclength++;
1720
1721            deltaVel = temp;
1722            temp = NULL;
1723            deltaVelcopy = deltaVel;   //MP-- make a copy now since later we modify
                deltaVel-- we need deltaVelcopy and deltafunclengthcopy for the readme
                file.
1724                        deltafunclengthcopy = deltafunclength;
1725        }
1726
1727        if(param=="timemin:")
```

```
1728            {   if(restart == 1)
1729                { TimeMin = parval;}
1730                else {TimeMin = 0.0;}    //MP-- At this point, if we specify restart as
                    == 1, then the minimum time will be set to what we choose. Otherwise, we
                    are not
1731            }                                   //restarting from a previous simulation,
                and the minimum time will be set to 0. Note we are using TimeMin as the
                variable to control
1732                                                //the restarting of the program. If TimeMin
                                                    is zero, we will start as per usual, but if
                                                    TimeMin is not zero, we need to put the
1733                                                //program in the final state it was in, and
                                                    go from here.
1734            if(param=="timemax:")
1735            {   TimeMax = parval;
1736                Specified = true;
1737            }
1738
1739            if(param == "files:")
1740            {
1741                FF=false;
1742                TE=false;
1743                KK=false;
1744                XX=false;
1745                VV=false;
1746                FA=false;
1747                FAL=false;
1748                FAS=false;
1749
1750
1751                if(!in file.eof()&&!in file.fail())
1752                    in file>>param;
1753
1754                for(unsigned int i =0;i<param.length();++i)
1755                {
1756                    param[i]=tolower(param[i]);
1757
1758                    if(param[i]=='f')
1759                        FF=true;
1760                    if(param[i]=='t')
1761                        TE=true;
1762                    if(param[i]=='k')
1763                        KK=true;
1764                    if(param[i]=='x')
1765                        XX=true;
1766                    if(param[i]=='v')
1767                        VV=true;
1768                    if(param[i]=='a')
1769                        FA=true;
1770                    if(param[i]=='l')
1771                        FAL=true;
1772                    if(param[i]=='s')
1773                        FAS=true;
1774                }
1775            }
1776
1777            if(param == "addforce:")    //MP-- This is for perturbing the first grain in
                the chain (i.e. the left-most grain) with a specified force.
1778            {
1779                asymLeft = 1;           //MP-- Set the flag to 1 (meaning true) for
                    asymmetric driving on the left end of the chain.
1780                cc ='d';//default
1781                if(!in file.eof()&&!in file.fail())
1782                    in file>>cc;
1783
1784                //std::cout<<cc<<std::endl;
1785
1786                cc=tolower(cc);
1787                if(cc=='k')
```

```
1788                {
1789                    if(!in file.eof()&&!in file.fail())
1790                    {
1791                        parval =0;
1792                        in file>>parval;
1793                        konstForce +=parval;
1794                        in file>>parval;
1795                        konstLength = parval;   //MP-- Now we specify the length in terms of micro-s. We will shut off the constant force after this interval.
1796                    }
1797                }
1798                else
1799                {
1800                    drivingforces* temp;
1801                    temp=(drivingforces*)malloc((sourceforcelength+1)*sizeof(drivingforces));
1802
1803                    for(int c=0;c<sourceforcelength;++c)
1804                    {
1805                        temp[c].type = sourceforce[c].type;
1806                        temp[c].amplitude = sourceforce[c].amplitude;
1807                        temp[c].frequency = sourceforce[c].frequency ;
1808                        temp[c].phase = sourceforce[c].phase;
1809                        temp[c].duration = sourceforce[c].duration;
1810                    }
1811
1812                    temp[sourceforcelength].type = cc;
1813                    in file>>parval;
1814                    temp[sourceforcelength].amplitude = (parval);
1815                    in file>>parval;
1816                    temp[sourceforcelength].frequency = (parval);
1817                    in file>>parval;
1818                    temp[sourceforcelength].phase = (parval);
1819                    in file>>parval;
1820                    temp[sourceforcelength].duration = (parval);
1821                    sourceforcelength++;
1822
1823                    sourceforce = temp;
1824                    temp = NULL;
1825                }
1826            }
1827
1828            if(param == "addforcelast:")   // This is for perturbing the last grain in the chain with a specified force (i.e. the right-most grain)
1829            {
1830                asymRight = 1;
1831                ccl ='d';//default
1832                if(!in file.eof()&&!in file.fail())
1833                    in file>>ccl;
1834
1835                //std::cout<<ccl<<std::endl;
1836
1837                ccl=tolower(ccl);
1838                if(ccl=='k')
1839                {
1840                    if(!in file.eof()&&!in file.fail())
1841                    {
1842                        parval =0;
1843                        in file>>parval;
1844                        konstForceLast +=parval;
1845                        in file>>parval;
1846                        konstLengthLast = parval;  //Same as above for konstLength.
1847                    }
1848                }
1849                else
1850                {
1851                    drivingforces* tempLast;
```

```
1852                                                                                                    ↵
                        tempLast=(drivingforces*)malloc((sourceforcelengthLast+1)*sizeof(d ↵
                        rivingforces));
1853
1854                    for(int cl=0;cl<sourceforcelengthLast;++cl)
1855                    {
1856                        tempLast[cl].type = sourceforceLast[cl].type;
1857                        tempLast[cl].amplitude = sourceforceLast[cl].amplitude;
1858                        tempLast[cl].frequency = sourceforceLast[cl].frequency ;
1859                        tempLast[cl].phase = sourceforceLast[cl].phase;
1860                        tempLast[cl].duration = sourceforceLast[cl].duration;
1861                    }
1862
1863                    tempLast[sourceforcelengthLast].type = ccl;
1864                    in file>>parval;
1865                    tempLast[sourceforcelengthLast].amplitude = (parval);
1866                    in file>>parval;
1867                    tempLast[sourceforcelengthLast].frequency = (parval);
1868                    in file>>parval;
1869                    tempLast[sourceforcelengthLast].phase = (parval);
1870                    in file>>parval;
1871                    tempLast[sourceforcelengthLast].duration = (parval);
1872                    sourceforcelengthLast++;
1873
1874                    sourceforceLast = tempLast;
1875                    tempLast = NULL;
1876                }
1877            }
1878
1879
1880            if(param == "addforcefirstlast:")   //This is for both ends of the chain to    ↵
                be symmetrically perturbed by a specified force.
1881            {
1882                symLeftRight = 1;
1883                ccfl ='d';//default
1884                if(!in file.eof()&&!in file.fail())
1885                    in file>>ccfl;
1886
1887                //std::cout<<cc<<std::endl;
1888
1889                ccfl=tolower(ccfl);
1890                if(ccfl=='k')
1891                {
1892                    if(!in file.eof()&&!in file.fail())
1893                    {
1894                        parval =0;
1895                        in file>>parval;
1896                        konstForceSym +=parval;
1897                        in file>>parval;
1898                        konstLengthSym = parval;   //length of interval (in micro-s) of    ↵
                            driving the chain symmetrically with constant force.
1899                    }
1900                }
1901                else
1902                {
1903                    drivingforces* tempFL;
1904                                                                                                    ↵
                        tempFL=(drivingforces*)malloc((sourceforcelengthFL+1)*sizeof(drivingfo ↵
                        rces));
1905
1906                    for(int cfl=0;cfl<sourceforcelengthFL;++cfl)
1907                    {
1908                        tempFL[cfl].type = sourceforceFL[cfl].type;
1909                        tempFL[cfl].amplitude = sourceforceFL[cfl].amplitude;
1910                        tempFL[cfl].frequency = sourceforceFL[cfl].frequency ;
1911                        tempFL[cfl].phase = sourceforceFL[cfl].phase;
1912                        tempFL[cfl].duration = sourceforceFL[cfl].duration;
1913                    }
1914
```

```
                tempFL[sourceforcelengthFL].type = ccfl;
                in file>>parval;
                tempFL[sourceforcelengthFL].amplitude = (parval);
                in file>>parval;
                tempFL[sourceforcelengthFL].frequency = (parval);
                in file>>parval;
                tempFL[sourceforcelengthFL].phase = (parval);
                in file>>parval;
                tempFL[sourceforcelengthFL].duration = (parval);
                sourceforcelengthFL++;

                sourceforceFL = tempFL;
                tempFL = NULL;
            }
        }


        if(param == "grains:")
        {
            bottomGrain= (int(parval));
            in file>>parval;
            topGrain= (int(parval));
            grainsSpecified=true;
        }

        if(param == "filename:")
        {
            in file>>param;
            fileName = " " + param;
        }

        in file>>param;
    }

    in file.close();


    ////MP----This is where I am modifying the //deltaVel structure a bit: Need to run
    a couple of loops:

        if(deltafunclength != 0)
        {
            int dvmax = deltafunclength;
            std::cout << dvmax <<std::endl;
            int dcount = 0;
            int c2count = 0;
        int c3count = 0;
            int dvmax2 = 0;
            int dindex = 0;
                int ccount = 0;
            deltafunclength = 0;
            for(int c2=0;c2<dvmax;++c2)
            {
            c3count = deltaVel[c2].howmany;
            if(c3count == -1){
                c3count = 1;
            }
                c2count = c2count + c3count;
            }

            deltafunc* temp;
            temp=(deltafunc*)malloc((c2count)*sizeof(deltafunc));

            for(int c=0;c<dvmax;++c){
                dvmax2 = deltaVel[c].howmany;
            if(dvmax2 != -1){
                    if(c != 0){
                    ccount = deltaVel[c-1].howmany;
                    if(ccount == -1){
```

```
                        ccount = 1;        //Can't add deltaVel[c-1].howmany since this quantity is -1 to specify a window.
                    }
                            dcount = dcount + ccount;
                        }
                        for(int d=0;d<dvmax2;++d){
                            dindex = d + dcount;
                            temp[dindex].addedVelocity = deltaVel[c].addedVelocity;
                        temp[dindex].grain = deltaVel[c].grain;
                    temp[dindex].time = deltaVel[c].time+d*deltaVel[c].interval;        //Have to adjust the time--- this is where running the loop becomes useful!
                        temp[dindex].howmany = 1;  //Added this line in oct.11/14 MP.  Now if the d loop runs 1 time, temp will have dvmax2 entries. so in the next run of the c loop we need
                            temp[dindex].interval = 0;
                    deltafunclength++;        //to take this into account. This is why we have this additional variable dcount. I wanted to generalize it to more than just 2 edge
                }                  //grains hence the loop.
                }
            else{
                if(c != 0){
                    ccount = deltaVel[c-1].howmany;
                    if(ccount == -1){
                        ccount = 1;        //Can't add deltaVel[c-1].howmany since this quantity is -1 to specify a window.
                    }
                            dcount = dcount + ccount;
                        }
                        dindex = dcount;
                    temp[dindex].addedVelocity = deltaVel[c].addedVelocity;
                temp[dindex].grain = deltaVel[c].grain;
                temp[dindex].time = deltaVel[c].time;        //Have to adjust the time--- this is where running the loop becomes useful!
                    temp[dindex].howmany = -1;
                temp[dindex].interval = deltaVel[c].interval;
                        deltafunclength++;
                    }
        }

        deltaVel = temp;      //Rewrite over the existing deltaVel structure, replacing it with the appropriately restructured temp, and then as in the loops above, reset temp to nothing.
        temp = NULL;

                for(int i=0; i<deltafunclength; i++){
            std::cout<<deltaVel[i].grain<<'\t'<<deltaVel[i].time<<'\t'<<deltaVel[i].addedVelocity<<'\t'<<deltaVel[i].howmany<<'\t'<<deltaVel[i].interval<<std::endl;
        }
    }


    if(newTimeRange&&!Specified)
        TimeMax = TimeMin + nsteps*dt;
    if(newGrains&&!grainsSpecified)
        topGrain = nptles;


    if(asymLeft==1){        //MP--added June 2015-- if we want to apply a non-constant force for the entire duration of the simulation, we can specify the duration as -1.
        for(int c1=0;c1<sourceforcelength;++c1)
        {
            Fdur = sourceforce[c1].duration;
            if(Fdur==-1){
                sourceforce[c1].duration = TimeMax;
```

```
2036                }
2037            }
2038        }
2039
2040        if(asymRight==1){
2041            for(int c2=0;c2<sourceforcelengthLast;++c2)
2042            {
2043                Fdur = sourceforceLast[c2].duration;
2044                if(Fdur==-1){
2045                    sourceforceLast[c2].duration = TimeMax;
2046                }
2047            }
2048        }
2049
2050        if(symLeftRight==1){
2051            for(int c3=0;c3<sourceforcelengthFL;++c3)
2052            {
2053                Fdur = sourceforceFL[c3].duration;
2054                if(Fdur==-1){
2055                    sourceforceFL[c3].duration = TimeMax;
2056                }
2057            }
2058        }
2059    }
2060
2061
2062    void makeResumeFile()
2063    {
2064        double tXr,tVr;
2065        std::string ResumeName;
2066        std::ostringstream *buffer;
2067        std::ofstream out file2;
2068
2069        buffer = new std::ostringstream();
2070
2071        (*buffer) << "resume.txt";
2072        ResumeName = buffer->str();
2073
2074
2075        if(restart==1){ out file2.open(ResumeName.c str(), std::ios::app);}
2076        else{ out file2.open(ResumeName.c str()); }
2077
2078        out file2.precision(DEFAULTPRECISION);
2079
2080        //Output relative locations followed by blank line, followed by grain velocities. This file will be used if one decides to restart the program later.
2081        if(restart==1){ out file2<<std::endl; }
2082        for(int lcv = bottomGrain; lcv<topGrain;++lcv)
2083        {
2084            tXr = Chain[nptles-lcv].relativeLocation;
2085            out file2<<-(tXr)<<'\t';
2086        }
2087        out file2<<std::endl<<std::endl;
2088
2089        for(int lcv = bottomGrain; lcv<=topGrain;++lcv)
2090        {
2091            tVr = -Chain[nptles-lcv].currentVelocity;
2092            out file2<<(tVr)<<'\t';
2093        }
2094        out file2<<std::endl;
2095
2096        out file2.close();
2097        delete buffer;
2098    }
2099
```