# The New Taper Chain Code

Adam Sokolow

February 17, 2014

$\beta$Version: 1-31-2014

## 1 Introduction

A complete rewrite of the taper chain code to allow for many more capabilities. A restructure was clearly needed after many updates and additional capabilities were added to the original code. This code was written in a way so that it can have added functionality added in a much more clean and simple fashion (at least that's what I hope). This is the beta version, so hopefully we'll track down any bugs or narrow down what options can and cannot be used together.

## 2 Conventions

Let me just briefly summarize some conventions being used throughout the code:

1. Grains are numbered from left to right starting at 1 and going to $N$.
2. The coordinate system is also left to right, so grain $N$ is at a larger location than grain 1.
3. Left is a synonym to the negative side of a grain.
4. Right is a synonym to the positive side of a grain.
5. Units - there are no unit dependencies in the code. **The user must use consistent units**.

***A lot of old commands are gone. A major change is that all chains are now specified by patterns (see commands section).***

## 3 Compilation

Right now there are **9 files** needed to compile the code.

1. TaperChainMain.cpp
2. Simulation.cpp
3. Simulation.h
4. Grain.cpp
5. Grain.h
6. OneDimDoubleVector.cpp
7. OneDimDoubleVector.h
8. methods.cpp
9. methods.h

If you're using Code::Blocks or a Microsoft Suite where you can build projects and add files, then you should be able to handle compilation on your own. Essentially all you have to do is add all the files to the project and typically these programs can figure it all out. Using g++ at the terminal you could just type

```
> g++ TaperChainMain.cpp Simulation.cpp Grain.cpp OneDimDoubleVector.cpp methods.cpp
```

and that should work.

## 4 Basic Usage

After you have compiled you have an executable file. I run this executable from the command line on a Windows machine. From the start menu, run the command "cmd.exe" and this will open a command window. Navigate to your working directory (where ever the executable is) using DOS commands and you should be able to run it. Double clicking will run the program too but often it closes afterward and you cannot see what errors you might have run into.

***Unlike previous versions, you are not forced into calling the parameters file "parameters.txt". Whatever you type at the command line after the executable is assumed to be the file name.***

Running the program without a parameters file name specified will print all the commands (just their names) and all the old command names no longer used.

The following is an example parameters.txt file that will run the traditional taper chain problem with twenty grains. There will be a whole lot more examples to come...

```
 ChainPattern: [a#20]
DEFAULTGRAIN:sphere#1|rho#1|poisson#0.3|youngs#1
subunit: a
loadstep: number # 1 | solver : velocity verlet | start # 0 | step # 0.1 |  stop # 30 |
rightwall: load step # 1 | subunit : a
leftwall: load step # 1 | subunit : a
initial condition: loadstep #1 | variable : velocity | addvalue # 0.1 | grains: 1
end
```

***Parameter files now have to end with the keyword "end".***

Running the above program will make a monodisperse chain, with walls, and give the first grain an initial velocity of 0.1. In this case the only output file is the ReadMe file. The readme file tells you everything the parser figured out that you wanted! This is the file that you will need to check to make sure that the radii, material properties, initial conditions, etc were interpreted properly.

***All of the commands are **space insensitive** but they have to be followed by a colon ':'*** So the following parameter file results in the exact same execution as the one above:

```

 C h a i n Pat     ter  n : [             a          #          20 ]
DEFAU LTG RAIN:sphere#1|rho#1|poisson# 0.3|yo u ngs#1
sub un it: a

loa ds tep: number # 1 | solver : velocity verlet | start # 0 | step # 0.1 |  stop # 30 |

righ twall: load step # 1 | subunit : a
leftwall: load step # 1 | subunit : a
initialcondition:loadstep#1|variable:velocity|addvalue#0.1|grains:1



end
```

This means you should feel free to add in white space where needed. However, **do not split a command onto multiple lines**.

## 5 Commands

This is an overview of each command implemented. Old commands have been replaced with new ones so that the user has to bring things up to date.

***Each command is assigned an internal priority so that they are guaranteed to be executed in the correct order.*** This makes it easier for the user to not have to worry about the order something appears.

New Commands, here is a list and description.

## 5.1 General Commands

### 5.1.1 c

The **c** command is a comment line, the parser will just skip over this line..
Usage example of the command **c**

```
c: The following is not executed by the program
```

This is good if you want to turn off or change something one at a time, like commenting out initial conditions.

```
c: initial condition: loadstep #1 | variable : velocity | addvalue # 0.1 | grains: 1
```

### 5.1.2 silent mode

The **silent mode** command if used will not echo the readme file to the command line. If used you also wont see what time it is when the simulation is running. This command takes no input and can appear anywhere in the code.
Usage example of the command **silent mode**

```
silent mode:
```

### 5.1.3 end

The **end** command is the final line of the program. Either use just the words "end" or you can follow it with a colon. It does not matter for this command.
Usage example of the command **end**

```
end:
```

## 5.2 Specifying Output

### 5.2.1 filename

The **filename** command sets the base name of an output file to begin with the base file name. The default is the name "baseName".
Usage example of the command **filename**

```
filename: baseFileNameString
```

### 5.2.2 output files

The **output files** command turns on the output files that correspond to the letters in parentheses: (A)ll of the outputs that follow, interface (O)verlaps, (I)nterface force, (F)orce on grains, (X)relative location, a(B)solute positions, (V)elocity, (K)inetic energy grain, (P)otential energy interfaces. Note that some quantities refer to the grains and others to the interfaces between. The following example requests for the relative location, the absolute location, the velocities, kinetic energies, and forces of the grains. It also asks

for the interface values of the interface forces as well as the overlap.
Usage example of the command **output files**

    output files: XBVKFIO

or to specify all the files use:

    output files: A

The default is that no files are output. Since writing to the harddrive is a slow process, the program will run faster with fewer outputs. So if you know what you want, it might be a good idea to limit the outputs.

### 5.2.3 output precision

The **output precision** command sets the decimal precision of the output in the files. Larger number will get you more decimal places, but larger files which are not always needed.
Usage example of the command **output precision**

    output precision: 6

### 5.2.4 output interval

The **output interval** command this command sets the output interval, that is the time between output prints to the file. This command replaces "time spits".
Usage example of the command **output interval**

    output interval: 0.015

This can influence the time step used in the solvers. If you request output at intervals smaller than the time steps, the time step is adjusted to be smaller. Similarly if your output interval is not some multiple of the time step, the solver will adjust to a smaller time step to match the output time, then return to the normal step you requested.

### 5.2.5 output file extension

The **output file extension** command this command sets the extension to use for output files. Default is .dat, but could be .txt or .whatever.
Usage example of the command **output file extension**

    output file extension: .dat

### 5.2.6 output grains

The **output grains** command this command specifies which grains have output. If it is absent, the default is that all grains and interfaces are specified. It can understand commas and dashes like a printer would for specifying pages in a document. If you specify a number larger than the number of grains it will just ignore your request.
Usage example of the command **output grains**

    output grains: 1-4,7-9,15

The example above will print the output for grains 1 through 4, 7 through 9 and 15. Also will output the interface data for any interface that has that grain, so the interface between 14 and 15, and 15 and 16 will be output. Print info for grains 1 through 20.

    output grains: 1-20

## 5.3 Specifying the Chain

Typically only the relative location of the chain's grains are considered. ***The initial condition for all chains is that the grains are in contact but not under compression.*** The absolute position of the chain is defined relative to the first grain. The left edge of the first grain is at the origin. The next grain is placed so that it is in contact according to the shape parameters. The walls are defined by their edge (not centroid). This edge can be shifted by an offset in the wall command.

***Most of the default commands are no longer here.*** For example, "D", "rho", "rlarge", "n", "q", and "w" are all gone. The way to specify a chain is now through the **chain pattern only**.

### 5.3.1 chain pattern

The **chain pattern** command uses a sort of algebra like syntax to specify repeats of the chain pattern through subunits defined later. The following is a simple example that makes a chain of ten default grains grains. The "DEFAULTGRAIN" needs to be defined since all grains fill in unknown/unspecified entries off of this subunit definition. .

Usage example of the command **chain pattern**

```
chain pattern: [DEFAULTGRAIN#10]
```

This command produces a pattern of "ababab":

```
chain pattern: [ababab]
```

which can also be done through the following:

```
chain pattern: [a | b^3]
```

This command produces a pattern of "aabaabaab":

```
chain pattern: [a#2|b^3]
```

This command produces a pattern of "aaabaaa":

```
chain pattern: [a#3][b][a#3]
```

So you might want a monodisperse chain but with an impurity at 100th grain of a 200 grain chain:

```
chain pattern: [DEFAULTGRAIN#99][b][DEFAULTGRAIN#100]
```

Or you might want a 100 repeat diatom chain (200 grains):

```
chain pattern: [a|b^100]
```

or a diatom chain with an impurity replacing an 'a':

```
chain pattern: [a|b^50][c][b|a^50][b]
```

Each distinct pattern should be included in the square brackets. Within each square bracket the caret symbol tells you how many repeats of the pattern. The pattern is defined by vertical bar separations and the pound symbol to specify repetitions of a subunit.

### 5.3.2 default grain

The **default grain** command defines the default subunit to be used. There are many options to set here, including material properties like density, Young's Modulus and Poisson's Ratio, and items that specify the shape.

Usage example of the command **default grain**

```
default grain: sphere#1|rho#2|poisson#0.3|youngs#3
```

This is a special case of the command for one reason. It is a special case since we used "sphere" where the number that follows is the radius. Additional material specification is through: "Rho" specifies the density, "poisson" the Poisson's ratio, and "youngs" the Young's modulus. ***The default grain command has to appear somewhere in the code.*** ***Use consistent units throughout.***

### 5.3.3 subunit

The **subunit** command defines a subunit to be used. There are many options to set here, including material properties like density, Young's Modulus and Poisson's Ratio, and items that specify the shape. **This is partly a case sensitive command**.
Usage example of the command **subunit**

```
subunit: CaseSensitiveName|sphere#1|rho#2|poisson#0.3|youngs#193
```

Additional material specification is through: "Rho" specifies the density, "poisson" the Poisson's ratio, and "youngs" the Young's modulus. ***Any parameter that you leave off from the subunit definition will be inherited from the default grain.*** Here is another example. Now that we have a default grain, this subunit will inherit everything from the default grain but then overwrite with any new specifications. So here, for a new subunit called "a", the density is reduced to 1.

```
subunit: a|rho#1
```

There are more general options for the shape. Instead of using sphere you can use the following:

```
subunit: a|radiusminus#1|widthparam#2|radiusplus#2|alphaminus#2|alphaplus#3|mparam#1.4
```

This specifies all the parameters that were in Sonny's thesis to specify a shape and the calculate the Hertz law between them. If the left and right radii and alphas are the same you can use the **radius** and **alpha** commands instead:

```
subunit: a|radius#1|widthparam#2|alpha#3|mparam#1.4
```

*****Note that the volume command and the mass command override the actual shape's mass or volume. ***** In this way you could specify the force law in between grains by specifying shapes, and then set the grain masses or volumes equal through the other command. So this command defines a grain called "b" and makes its volume 3 in the units you are using, no matter what the default grain's shape is that it normally would inherit.

```
subunit: b|volume#3
```

Similarly the following would use the default grain subunit, then revise its mass to be 3 - no matter what density or volume it would have had!

```
subunit: b|mass#3
```

### 5.3.4 tapering ratio

The **tapering ratio** command sets an overlying tapering ratio. Starting with the left grain, the next grain over has its radius reduced by the ratio to the power of the grain number. This applies after any chain pattern you might have specified and will then recalculate the volumes and masses. If for a particular subunit you specified the mass, the tapering will change the radius but not the mass! So in this way the tapering ratio will only affect the prefactor calculated..
Usage example of the command **tapering ratio**

```
tapering ratio: 0.9
```

So that if the first grain had a radius 1, the second 2, and the third 3, they would be updated to: $1 \times 0.9^0$, $2 \times (0.9)^1$, and $3 \times (0.9)^2$. In a monodisperse chain with tapering applied this is equivalent to the old "q" command.

The use of a negative starts with the right grain and moves left instead.

```
tapering ratio: -0.9
```

### 5.3.5 leftwall

The **leftwall** command turns on the left wall. The wall inherits material properties and shape properties of subunit a. The simplest usage is just to turn it on and let it inherit properties from a previously defined subunit as in the following example.
Usage example of the command **leftwall**

    leftwall: loadstep # 2 | subunit : a

An alternate usage is to give the wall an offset. This will displace the wall so that an overlap is not detected until the relative location of the grain is equal to the offset or greater. Or if the offset is positive (or negative for a right wall) there will be an overlap that is applied to the grain. The following command offsets the wall to the left and thus widens the gap between the grain to 1.

    leftwall: loadstep # 2 | subunit : a | offset # -1

You can also override the radius of the subunit the wall inherits from to set to be flat. To do this use the radius command as follows:

    leftwall: loadstep # 2 | subunit : a | radius # inf

I think internally right now a subunit prefactor calculation uses the limit of R goes to infinity if the submitted radius is -1. Not entirely sure if this works with the subunit command.

### 5.3.6 rightwall

The **rightwall** command turns on the right wall. Same syntax as above for the left wall..
Usage example of the command **rightwall**

    rightwall: loadstep # 2 | subunit : a | radius # inf | offset # 1

***The default is that there are no walls on in a load step. So you have to redefine the walls at each load step.***

## 5.4 Loadsteps, Integrators, Solvers, Initial Conditions, Boundary Conditions

New to this version is the notion of a "load step". This is an interval of time during which boundary conditions, initial conditions, solvers, walls, damping, and body accelerations can be turned on or off. The default is that a loadstep has none of these conditions, and you need to add them for each load step you want them to be in.

### 5.4.1 loadstep

The **loadstep** command defines a new load step. It has to have a number of things, including the number you want to refer to it by, the name of the solver you want to use, the starting time, the time step, and the finishing time. The following command sets up the first load step, makes it use a Velocity Verlet solver, starting at time = 0 , taking time steps of 0.01, and finishing at t = 30.
Usage example of the command **loadstep**

    loadstep: number # 1 | solver : velocity verlet| start# 0 | step # 0.01| stop # 30

The current solvers that you have to choose from:

1. Velocity Verlet
2. Runge Kutta Four
3. Monte Carlo Equilibrium
4. Equilibrium Body Only

I plan to add at least one Adaptive Time Step method (Runge Kutta Fehlberg), and a Predictor Corrector (probably a RK4 predictor, Adams-Bashforth corrector).

The Velocity Verlet and the Runge Kutta Forth Order solvers are pretty standard as far as solving ordinary differential equations.

The "equilibrium body only" solver is a way to solve for a body force/acceleration. It will balance the body forces with the correct interface forces and calculate the overlaps. Right now you need to use a wall (either left or right depending on the sign of the body force) for this to work.

The Monte Carlo Equilibrium solver uses a Monte Carlo Method to iteratively solve for equilibrium. This can be quite finicky! Since we are trying to solve a set of N, non-linear equations it can be quite difficult to find the solution, and since these equations are stiff, it is often it is necessary to apply the boundary conditions slowly. Because it is so difficult to use, you have access to some of its parameters.

### 5.4.2 Monte Carlo params

The **Monte Carlo params** command controls the parameters for a Monte Carlo solver. The overall idea is that a bunch of random chain states are created and ones that reduce the energy or forces are accepted and others that do not reduce the energy are accepted with some probability..
Usage example of the command **Monte Carlo params**

Monte Carlo params: monte carlo params: load step #1 | max iterations # 2000 |max cutbacks # 10 | shift# 0.25 | accept ratio # 0.1 | scale # 0.1 | tolerance # 0.0001 | mc avgs # 5000
For the Monte Carlo params command to work you need to specify all of the following:

1. Load Step #
2. Max Iterations #
3. Max Cutbacks #
4. Shift #
5. Accept Ratio #
6. Scale #
7. Tolerance #
8. MC Avgs #

I'll start with how a "test" state is created and work my way up in the algorithm. First the current state of the chain at some time is used to calculate all the forces on the grains. These forces are used to direct the random search. Most of the time you want to follow the forces, but at times to avoid oscillations you do not. A normally distributed random number, call it $0 \leq r \leq 1$ is created for each grain in the chain. A new test relative location of the grain is calculated by:

$$x_i^{n+1} = x_i + \text{scale } 2\,(r - \text{shift})\,\frac{f_i}{m_i} \tag{1}$$

Or the grain is not moved if it is not "movable" (see freeze and unfreeze grain below). This procedure is done for each grain and then the forces are calculated again. The sum of the forces on a grain is stored in the state variable and if the average of the absolute values of the grain forces (not the interface forces!) is less than in the previous step the new state is accepted and put into a collection to use later. If the average force is larger then another random variable $p$ is picked. If $p \leq$ Accept Ratio it is kept even though its force is larger, and if it is larger than the accept ratio it is thrown out. This procedure is repeated MC Avgs times, so in the best case there will be MC Avgs (equal to 5000 in the above example) that the program will average over. The average solution is taken as the next configuration. The procedure is repeated using this next configuration as the current configuration until the magnitude of the maximum force drops below the tolerance (ignoring unmovable or frozen grains). If the procedure fails to converge, that is it reaches the Maximum Iterations before the error (average magnitude of the maximum force) drops below the tolerance, the scale of the probe is reduced. The problem can fail and the probe can be reduced at maximum the Max Cutbacks. If the problem does not converge within the Max Cutbacks it will exit with failure (welcome to iterative solvers that have trouble converging).

I highly recommend looking at the driving function command for use in equilibrium solvers. Time doesn't mean the same thing as it does in a dynamic problem but the boundary conditions (say moving two grains) can be applied slowly using a linear function.

### 5.4.3 freeze grain

The **freeze grain** command does exactly what it sounds like, it makes a grain unmovable during a load step. This is one way to implement walls if you wanted. Or it is a way to apply a pre-stress to the system through an equilibrium solver followed by a dynamic load step where you unfreeze the grain and let the chain relax..
Usage example of the command **freeze grain**

```
freeze grain: loadstep # 1 | grains : 1,3-5
```

As you can see it uses the standard grain specification syntax, i.e., "grains : " followed by numbers separated by commas and dashes.

### 5.4.4 unfreeze grain

The **unfreeze grain** command makes a grain movable during a load step..
Usage example of the command **unfreeze grain**

```
unfreeze grain: loadstep # 2 |grains: 1,3-5
```

***The default is that during a load step all grains are movable!*** So you might wonder why you need to unfreeze a grain. The only reason is so that you could perhaps freeze all the grains using one command, and then lift the constraint on one or two grains. Maybe this is superfluous but it seems reasonable to want to have.

### 5.4.5 body acceleration

The **body acceleration** command adds a body acceleration to all grains, e.g. gravity..
Usage example of the command **body acceleration**

```
body acceleration: loadstep # 1 |const # 9.8
```

If you do not have any walls (or a frozen grain) the grains will just fall off to infinity! Also the grains at the initial time step are taken to be in contact but not compressed, so if you want to use gravity and need them at equilibrium you should read the load step command for adding a body force/acceleration (above).

### 5.4.6 velocity based damping

The **velocity based damping** command adds a damping force proportional to the velocity of the grain. In this current implementation the constant is the same for all grains. In the future, it could easily be changed to allow for grain/interface specific loss..
Usage example of the command **velocity based damping**

velocity based damping: loadstep # 1 |const # 0.3

### 5.4.7 unload force ratio

The **unload force ratio** command adds a reduction in the unloading force as compared to the loading force. This command replaces what used to be called "restitution", however I believe there is a change in the convention of the number as well.
Usage example of the command **unload force ratio**

unload force ratio: loadstep # 1 |const # 0.9

***Restitution used to be used as a small number, e.g., 0.1. Unload force ratio is now a number close to but less than 1.***

$$F_{\text{unload}} = \text{Unload Force Ratio} \times F_{i,j} \tag{2}$$

### 5.4.8 initial condition

The **initial condition** command sets up the initial condition that will apply at the beginning of a load step.
Usage example of the command **initial condition**

initial condition: loadstep # 1 |grains: 1| variable : velocity | add value # 0.1

The grains portion of this command can accept the usual syntax (commas and dashes). The add value is added to the variable you specified at the beginning of the load step. Variable can either be velocity or displacement.

initial condition: loadstep # 1 |grains: 1| variable : displacement | add value # 0.1

### 5.4.9 driving function

The **driving function** command sets up a function that directly changes a grain as the simulation runs. You have a few options here so I'll spread this one out.
Usage example of the command **driving function**

driving function: load step # 1 | add or set : add | grains : 1 | variable : displacement | function type : cosine | params : .25,4,0

All driving functions have to be defined for a particular load step. They need to be specified whether it adds to the current value, or whether it overrides the current state variable (add or set). Grains is again the standard comma and dashes to which it applies. ***The driving function can override frozen grains it **should** be used with freeze grain! Otherwise the velocity values (if you apply a displacement) or displacement (if you apply a velocity) are just erroneous since they are not based on the integrator but rather the function.*** Variable can either be velocity or displacement. In the future this might be extended to say a force, but for now it is only the two. The function types have a number of options.

1. cosine
2. sine
3. poly

These are easily extended to other functions, but these seemed sufficient at first. Each function needs its parameters specified through the "params" command. Let params be a vector $p_i$, where the above example defines three entries: $p = \{0.25, 4, 0\}$. For a cosine function:

$$f(t) = p_1 \cos(p_2\, t - p_3) \tag{3}$$

So in the above equation we specified: $f(t) = 0.25\cos(4\,t - 0)$. The sine function is very similar:

$$f(t) = p_1 \sin(p_2\, t - p_3) \tag{4}$$

The poly function is a polynomial specification. More parameters specify a larger order polynomial.

$$f(t) = \sum_{i=1} p_i\, t^{i-1} \tag{5}$$

So if we changed the command to read "poly" instead of cosine in the above example, we would have specified the function $f(t) = 0.25 + 4\,t + 0\,t^2$. Through multiple loadstep commands and driving functions you could define a piecewise function. This is sort of clumsy but that is all I can support at this time. ***If you use more than one set commands in the driving function on the same variable, you will end up with a random choice as to which wins!*** You can use multiple driving functions with add and sets appropriately chosen to build up complicated functions. I'll try to include an example in the example section.

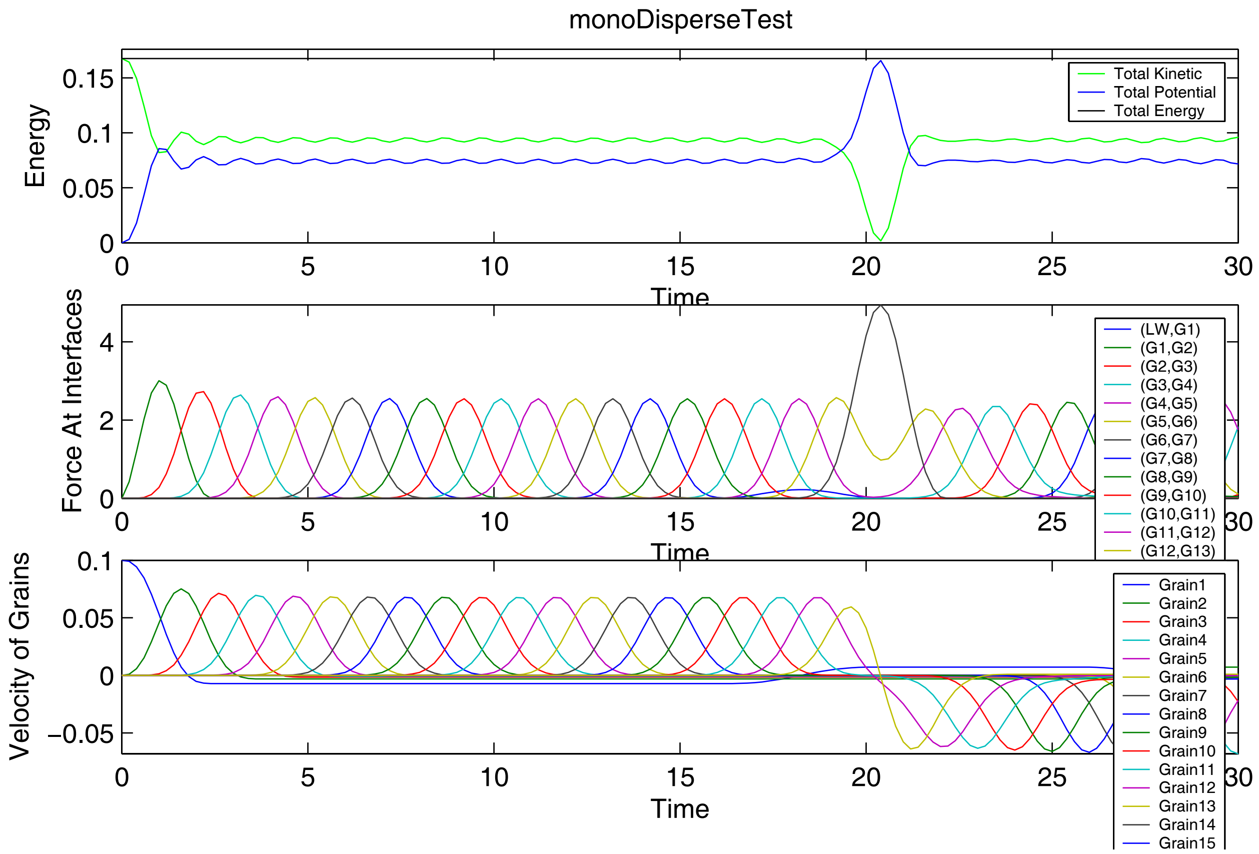
## 6 Examples

### 6.1 Monodisperse

Here is a simple example that sets up a monodisperse chain, and applies the initial condition of velocity to the first grain. This is the input file:

```
c: comment line, this is ignored
c: not a bad idea to have some info here for your own notes
c: we are outputting all files, extension .dat, with 7 digit precision and interval 0.2 ↩
    time units
outputfiles: a
output file extension: .dat
outputinterval: 0.2
outputprecision: 7
c: output all grains
outp ut grai ns : 1−20
fileName: monoDisperseTest
c: some random numbers (not sure these correspond to real units)
DEFAULTGRAIN:sphere#1|rho#8|poisson#0.3|youngs#200|
c: set up the monodisperse chain
 ChainPattern: [a#20]
subunit: a
c: Use a load step that uses the Velocity Verlet
loadstep: number # 1 | solver : velocity verlet | start # 0 | step # 0.01 |  stop # 30 |
c: put in two flat walls
rightwall: load step # 1 | subunit : a | radius : inf
leftwall: load step # 1 | subunit : a  | radius : inf
c: give the first grain a push!
initial condition: loadstep #1 | variable : velocity | addvalue # 0.1 | grains: 1
end
```

And this is the result:



## 6.2 Monodisperse fewer outputs

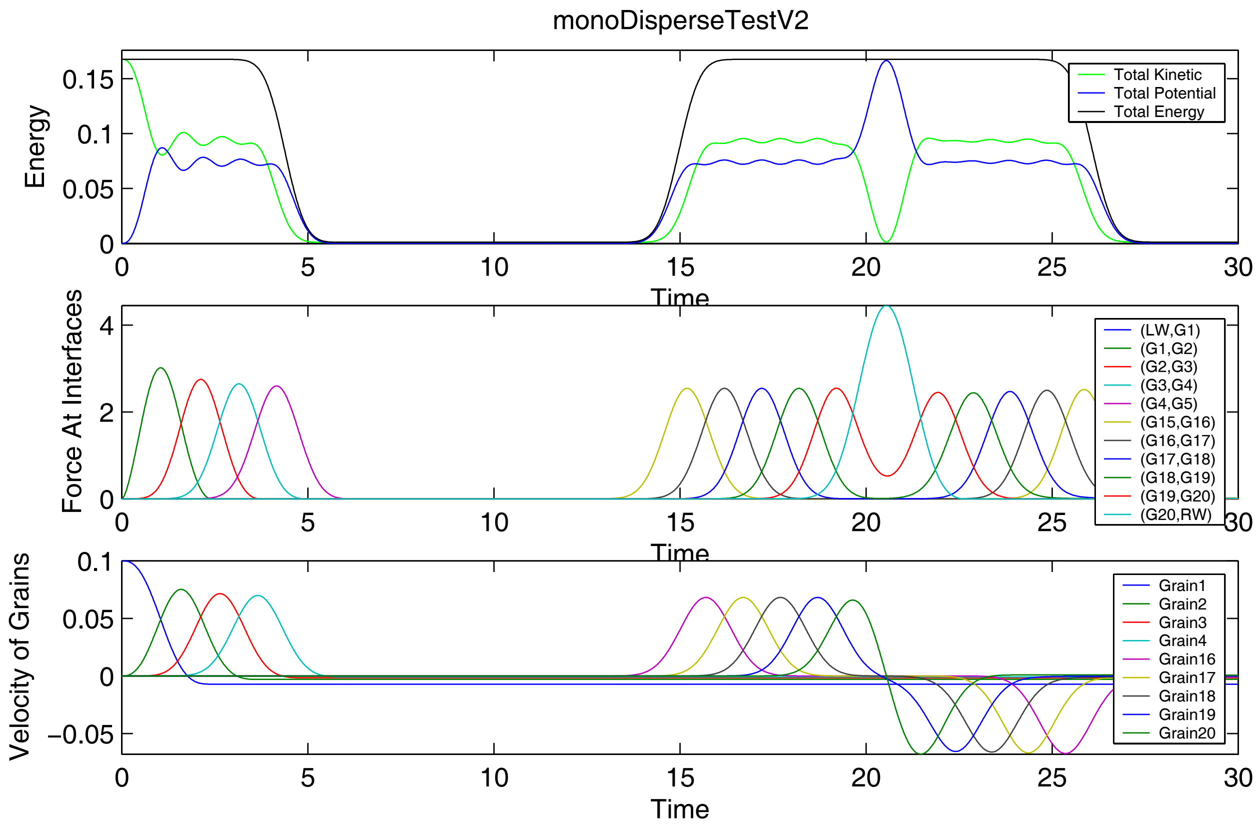
This is a similar simulation, except the the output interval is increased, the left wall is commented out (and so is not used) and the right wall is no longer flat, and the grain output is restricted to grains 1-4 and 16-20. This is the input file:

```
c: comment line, this is ignored
c: not a bad idea to have some info here for your own notes
c: we are outputting all files, extension .dat, with 7 digit precision and interval 0.2 ↩
   time units
outputfiles: a
output file extension: .dat
outputinterval: 0.02
outputprecision: 7
c: output all grains
outp ut grai ns : 1−4,16−20
fileName: monoDisperseTestV2
c: some random numbers (not sure these correspond to real units)
DEFAULTGRAIN:sphere#1|rho#8|poisson#0.3|youngs#200|
c: set up the monodisperse chain
 ChainPattern: [a#20]
subunit: a
c: tapering ratio : 0.9
c: Use a load step that uses the Velocity Verlet
```

```
loadstep: number # 1 | solver : velocity verlet | start # 0 | step # 0.01 |  stop # 30 |
c: put in two flat walls
rightwall: load step # 1 | subunit : a
c:leftwall: load step # 1 | subunit : a  | radius : inf
c: give the first grain a push!
initial condition: loadstep #1 | variable : velocity | addvalue # 0.1 | grains: 1

end
```

These are the results. As you will see, the energy files are only the energies for the grains and interfaces you requested. So if you are interested in the total energy of the system, right now you have to specify output for all grains. If this is too debilitating, it could be fixed in the future.



## 6.3 Taper Chain

Test a simple taper chain...

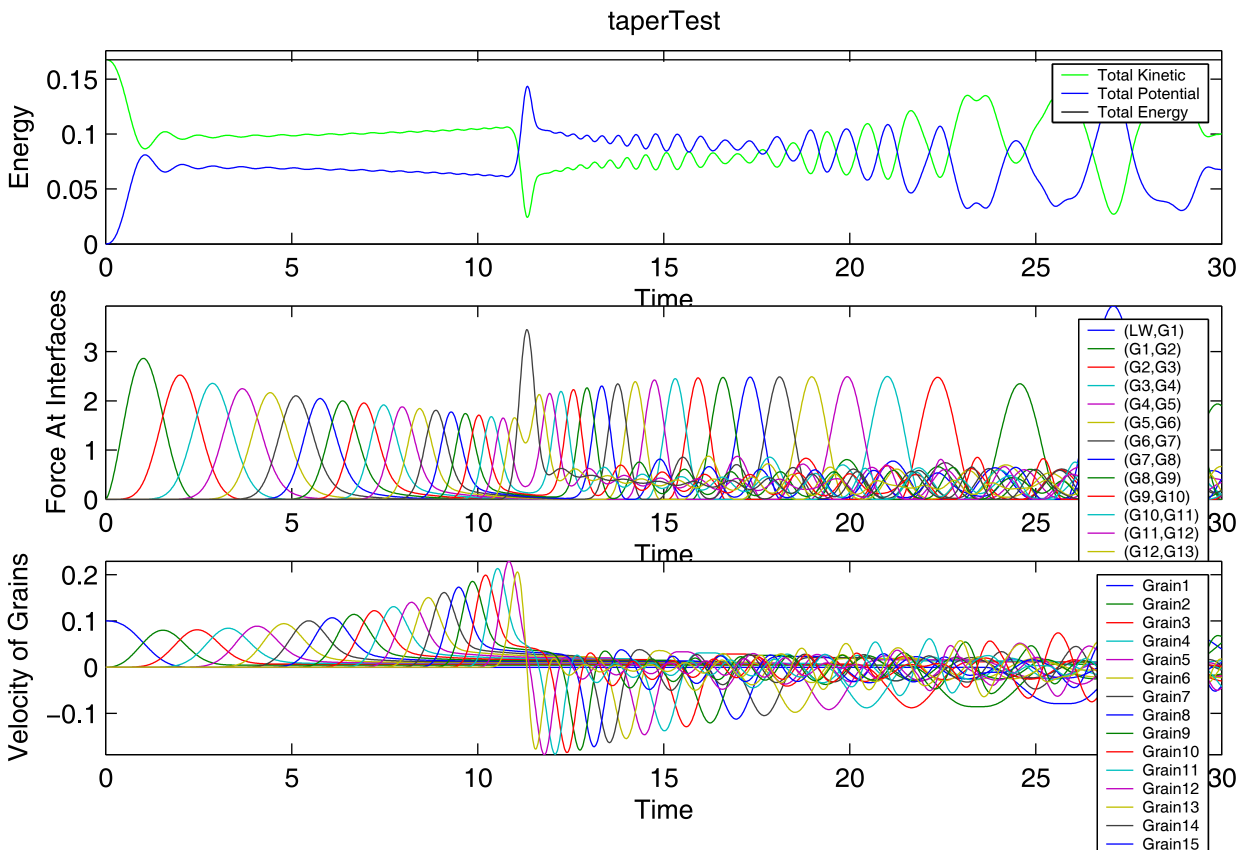
```
c: Taper Chain

outputfiles: v p k i
output file extension: .dat
outputinterval: 0.02
outputprecision: 5
fileName: taperTest
c: some random numbers (not sure these correspond to real units)
DEFAULTGRAIN:sphere#1|rho#8|poisson#0.3|youngs#200|
c: set up the chain
```

```
 ChainPattern: [a#20]
subunit: a
tapering ratio : 0.95
c: Use a load step that uses the Velocity Verlet
loadstep: number # 1 | solver : velocity verlet | start # 0 | step # 0.001 |  stop # 30 |
c: put in two flat walls
rightwall: load step # 1 | subunit : a | radius : inf
leftwall: load step # 1 | subunit : a  | radius : inf
c: give the first grain a push!
initial condition: loadstep #1 | variable : velocity | addvalue # 0.1 | grains: 1

end
```

These are the results.



## 6.4 Diatom Chain

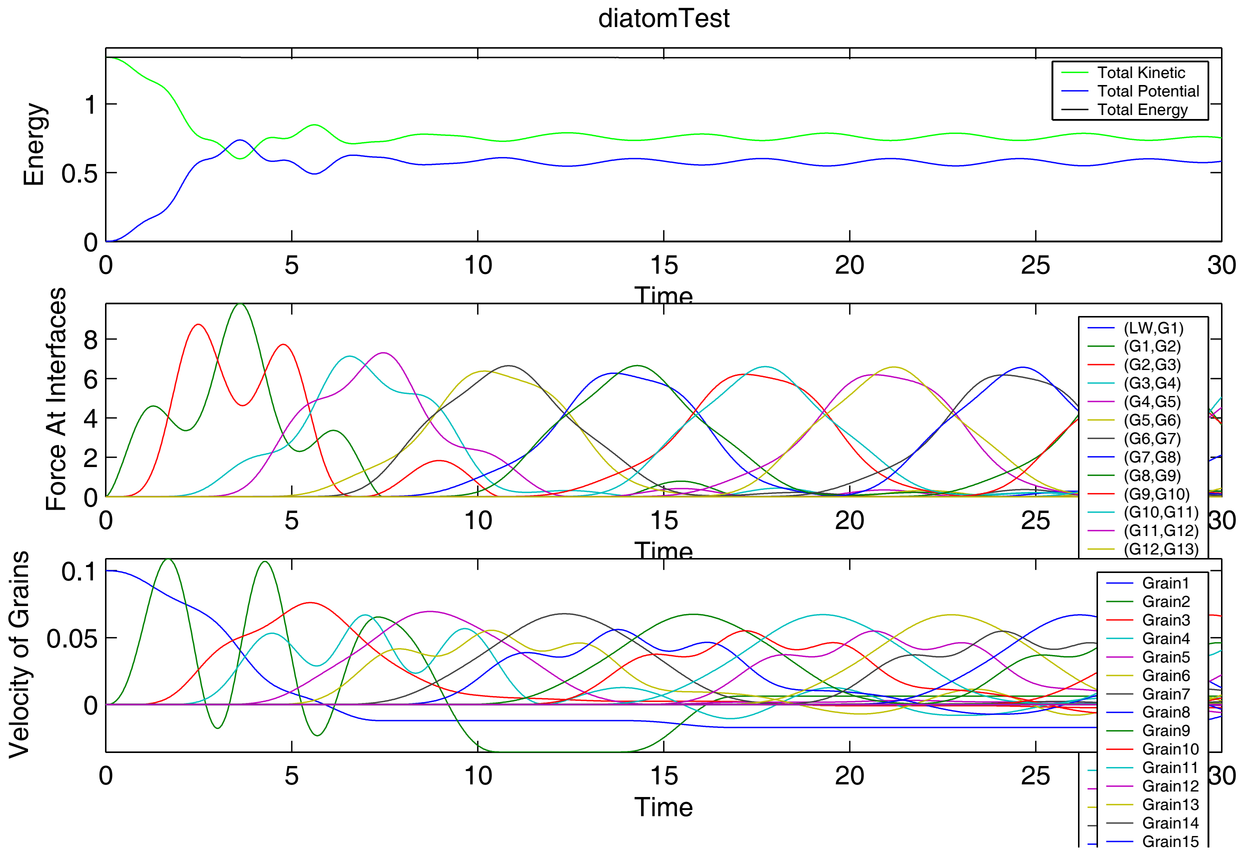
Test a simple diatom chain using Runge Kutta solver.

```
c: Diatom Chain

outputfiles: v p k i
output file extension: .dat
outputinterval: 0.02
outputprecision: 5
fileName: diatomTest
c: some random numbers (not sure these correspond to real units)
```

```
DEFAULTGRAIN:sphere#1|rho#8|poisson#0.3|youngs#200|
c: set up the chain
 ChainPattern: [a|b^20]
subunit: a | radius#2
subunit: b | radius#1
c: Use a load step that uses the Runge Kutta
loadstep: number # 1 | solver : runge kutta four | start # 0 | step # 0.001 |  stop # 30 ↩
    |
c: put in two flat walls
rightwall: load step # 1 | subunit : a | radius : inf
leftwall: load step # 1 | subunit : a  | radius : inf
c: give the first grain a push!
initial condition: loadstep #1 | variable : velocity | addvalue # 0.1 | grains: 1
end
```

These are the results.



## 6.5 Pattern Chain with Damping

Test a simple diatom chain using Runge Kutta solver and a patterned chain. Small damping added as well.

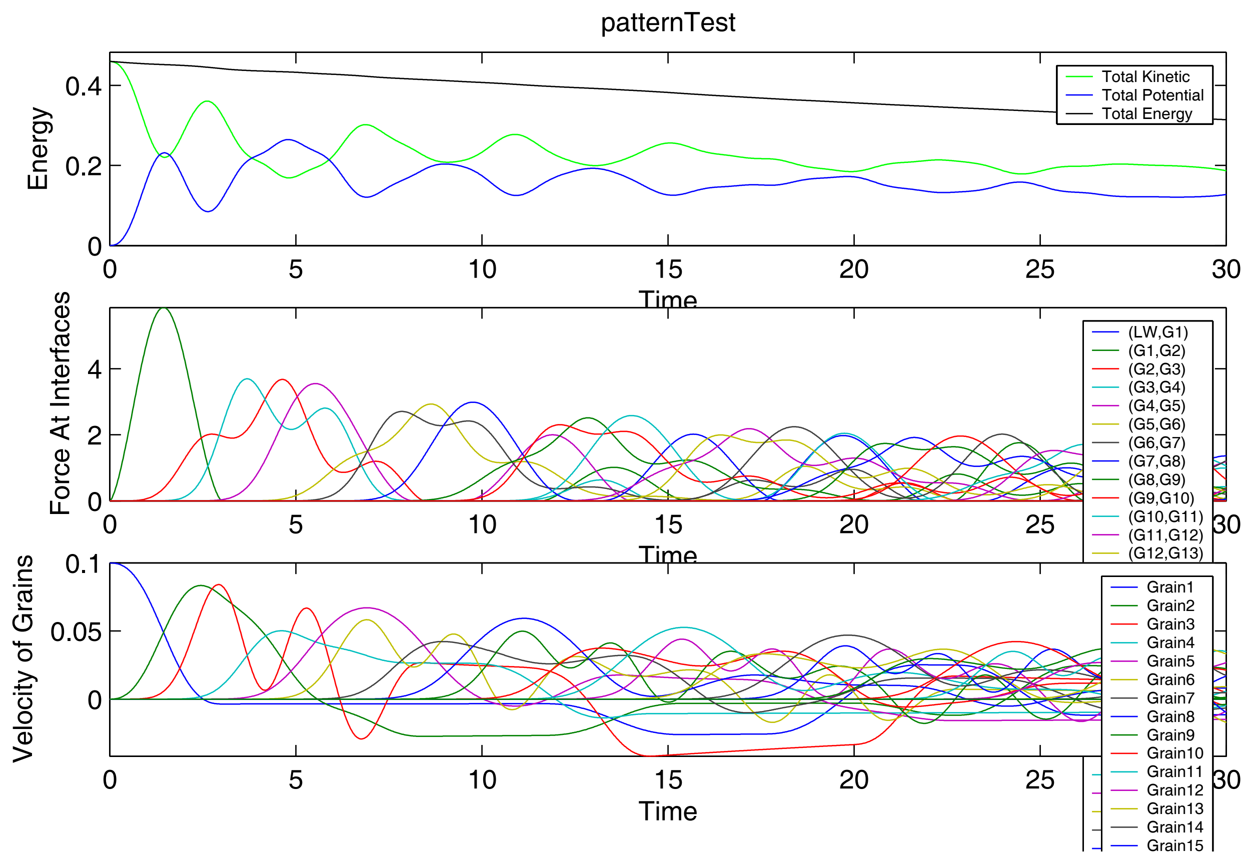
```
c: Pattern Chain

outputfiles: v p k i
output file extension: .dat
outputinterval: 0.02
outputprecision: 5
```

```
fileName: patternTest
c: some random numbers (not sure these correspond to real units)
DEFAULTGRAIN:sphere#1|rho#8|poisson#0.3|youngs#200|
c: set up the chain
 ChainPattern: [a#2|b^10]
subunit: a | radius#1.4 |
subunit: b | radius#1 | rho#4 | youngs #100
c: Use a load step that uses the Runge Kutta
loadstep: number # 1 | solver : runge kutta four | start # 0 | step # 0.001 |  stop # 30 ↩
   |
c: put in two flat walls
rightwall: load step # 1 | subunit : a | radius : inf
leftwall: load step # 1 | subunit : a  | radius : inf
c: give the first grain a push!
initial condition: loadstep #1 | variable : velocity | addvalue # 0.1 | grains: 1
velocity based damping : loadstep #1 | const # 0.7

end
```

These are the results.



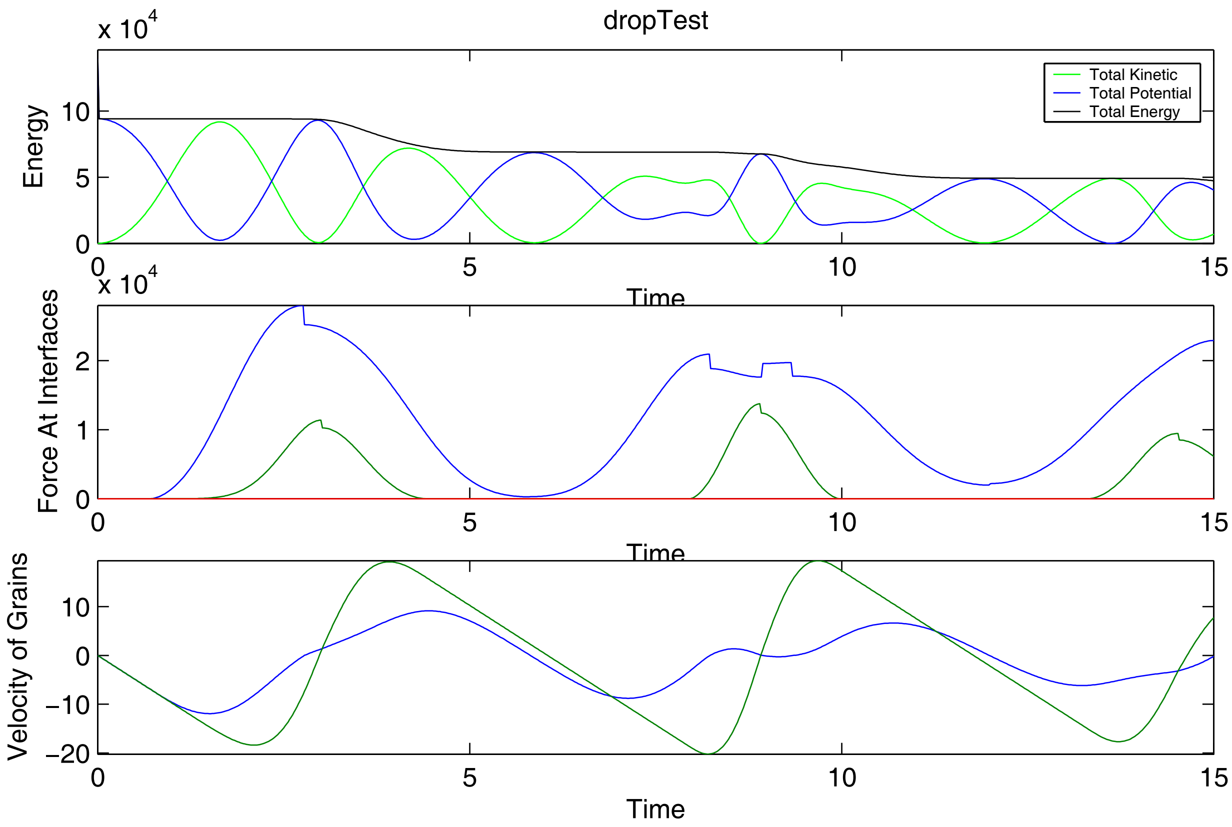
## 6.6 Body Acceleration With Restitution Unload Force Reduction

Classic problem , small grain on top of larger grain, drop from height. Runge Kutta solver and a patterned chain. Small damping added as well.

```
c: Drop Test
```

```
outputfiles: v p k i
output file extension: .dat
outputinterval: 0.02
outputprecision: 5
fileName: dropTest
c: some random numbers (not sure these correspond to real units)
DEFAULTGRAIN:sphere#1|rho#8|poisson#0.3|youngs#200|
c: set up the chain
 ChainPattern: [a|b]
subunit: a | radius#3
subunit: b | radius#2|rho#7
c: Use a load step that uses the Runge Kutta
loadstep: number # 1 | solver : runge kutta four | start # 0 | step # 0.001 |  stop # 15 ↩
    |
c: put in flat wall
leftwall: load step # 1 | subunit : a  | radius : inf
c: displace both grains to the right - could also add a negative wall offset.
initial condition: loadstep #1 | variable : displacement | addvalue # 2.5 | grains: 1-2
unload force ratio : loadstep # 1 | const # .9
c: add gravity
bodyacceleration : load step # 1 | const # -9.8
end
```

These are the results.



***Right now potential energy is a bit funny. It only accounts for the grain interface energy. So when you have body forces, you have to use the aboslute positions of the grains and do some post processing on your own.***

## 6.7 Taper Chain, at Equilibrium under Body Acceleration, then hit at the top

Use two different solvers in three steps.

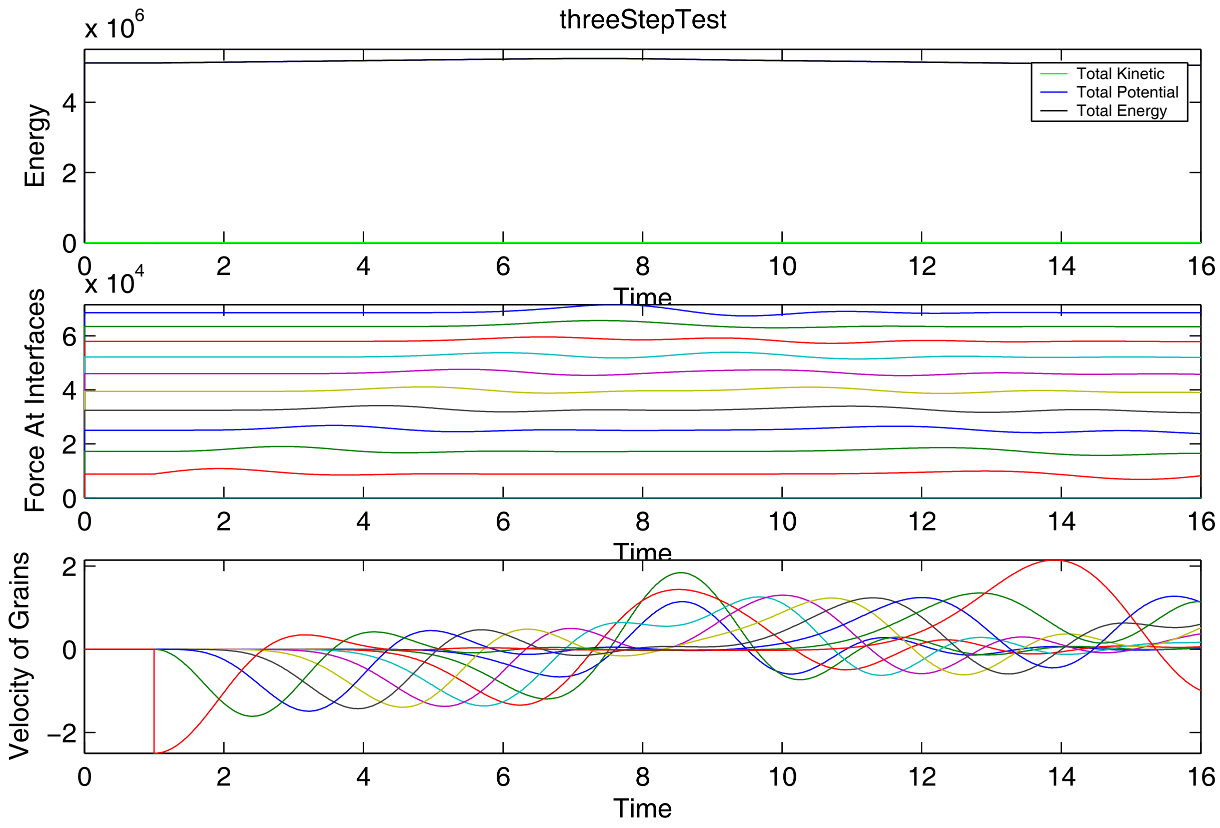
```
c: Equilibrium and Collision
outputfiles: v p k i
output file extension: .dat
outputinterval: 0.02
outputprecision: 5
fileName: threeStepTest
DEFAULTGRAIN:sphere#1|rho#8|poisson#0.3|youngs#200|
c: set up the chain
 ChainPattern: [a#10]
subunit: a | radius#3
tapering ratio: −0.98
c: Use a load step that solves equilibrium
loadstep: number # 1 | solver : equilibrium body only | start # 0 | step # 0.001 |  stop ↩
   # 0.001 |
c: put in flat wall
leftwall: load step # 1 | subunit : a  | radius : inf
c: displace both grains to the right − could also add a negative wall offset.
c: add gravity
bodyacceleration : load step # 1 | const # −9.8

c: Then use a load step that solves dynamics, verify nobody moves
loadstep: number # 2 | solver : velocity verlet | start # 0.001 | step # 0.001 |  stop # ↩
   1 |
c: put in flat wall
leftwall: load step # 2 | subunit : a  | radius : inf
c: displace both grains to the right − could also add a negative wall offset.
c: add gravity
bodyacceleration : load step # 2 | const # −9.8

c: Then use a load step that solves dynamics
loadstep: number # 3 | solver : velocity verlet | start # 1 | step # 0.001 |  stop # 16 |
c: put in flat wall
leftwall: load step # 3 | subunit : a  | radius : inf
c: displace both grains to the right − could also add a negative wall offset.
c: add gravity
bodyacceleration : load step # 3 | const # −9.8
initial condition: loadstep #3 | variable : velocity | addvalue # −2.5 | grains: 10

end
```

These are the results.



## 6.8 Driving Function, Freeze grain

A complicated example:

```
c: Equilibrium and Driving
outputfiles: v p k i x
output file extension: .dat
outputinterval: 0.02
outputprecision: 5
fileName: complicatedTest
DEFAULTGRAIN:sphere#1|rho#8|poisson#0.3|youngs#200|
c: set up the chain
 ChainPattern: [a#20]
subunit: a | radius#3
c: Use a load step that solves equilibrium
loadstep: number # 1 | solver : monte carlo equilibrium | start # .1 | step # .4 |  stop ↩
    # 1 |
freezegrain: load step #1 | grains : 1,7,20
monte carlo params: load step #1 | max iterations # 750 |max cutbacks # 10 | shift # 0.5 ↩
    | accept ratio # 0.3 | scale # 3 | tolerance # 1 | mc avgs # 2500
c: displace end grains inward
driving function: load step # 1 | add or set : set | grains : 1 | variable : displacement↩
     | function type : poly | params : 0,0.5
driving function: load step # 1 | add or set : set | grains : 20 | variable : ↩
    displacement | function type : poly | params : 0,−0.5
```

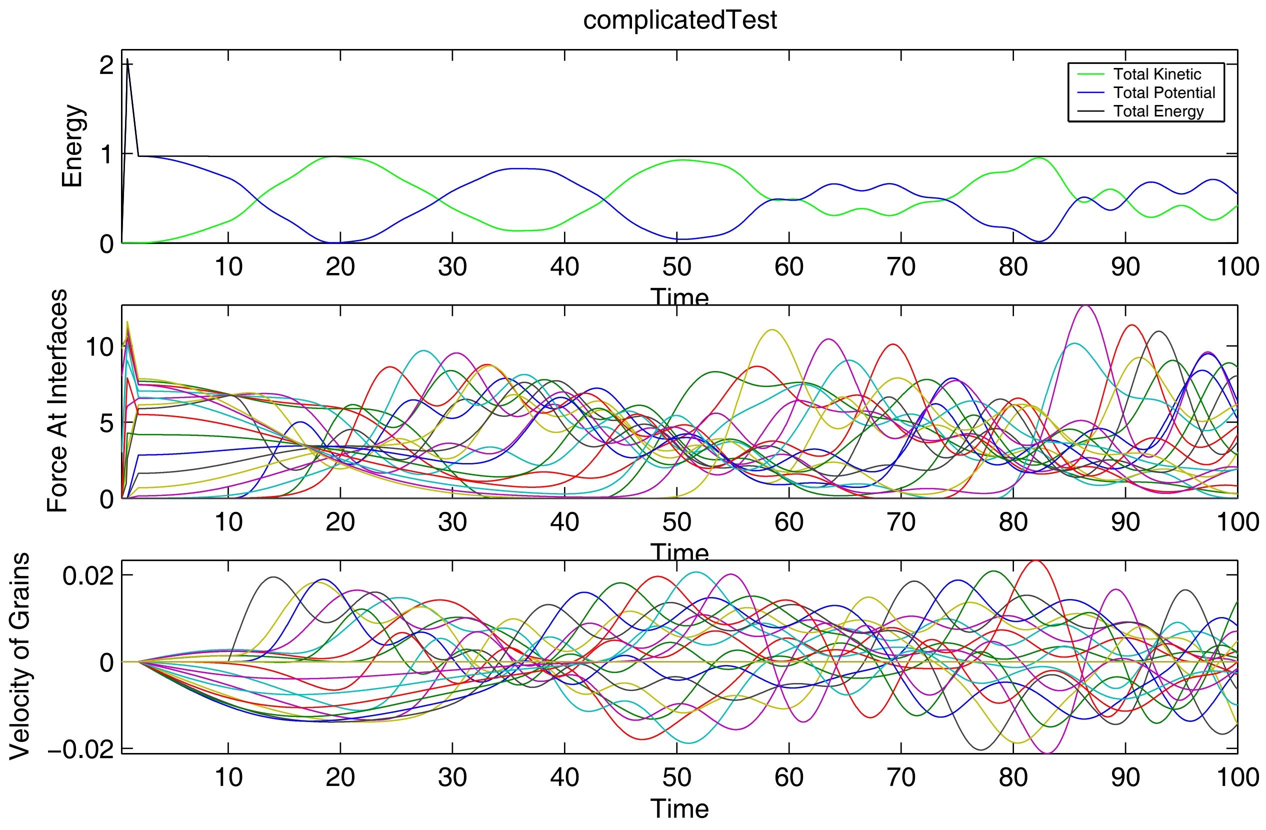
```
c: Use a load step that solves equilibrium
loadstep: number # 2 | solver : monte carlo equilibrium | start # 1.5 | step # .5 |  stop↩
     # 2 |
freezegrain: load step #2 | grains : 1,7,20
monte carlo params: load step #2 | max iterations # 2000 |max cutbacks # 10 | shift # ↩
    0.25 | accept ratio # 0.4 | scale # 6 | tolerance # .5 | mc avgs # 5000
c: displace end grains inward
driving function: load step # 2 | add or set : set | grains : 1 | variable : displacement↩
     | function type : poly | params : 0.5
driving function: load step # 2 | add or set : set | grains : 20 | variable : ↩
    displacement | function type : poly | params : −0.5


c: use VV to verify no one moves after equi solve:
loadstep: number # 3 | solver : velocity verlet | start # 2.001 | step # 0.001 |  stop # ↩
    10 |
freezegrain: load step #3 | grains : 1,7,20
velocity based damping : load step # 3 | const # 2


c: now release grain 7
loadstep: number # 4 | solver : velocity verlet | start # 10.001 | step # 0.001 |  stop #↩
     100 |
freezegrain: load step #4 | grains : 1,20

end
```

These are the results. The drop in energy at early time is after tightening the convergence tolerances.



This is a bit ridiculous, but the first grain is driven with a saw tooth wave (only 5 terms used) and the remaining chain is a tapered chain. ***Even though there is no right wall used, there is an overlap output for it. This gets rather large because there is no force to prevent it from growing.***

## 6.9 Anoter Driving Function Example

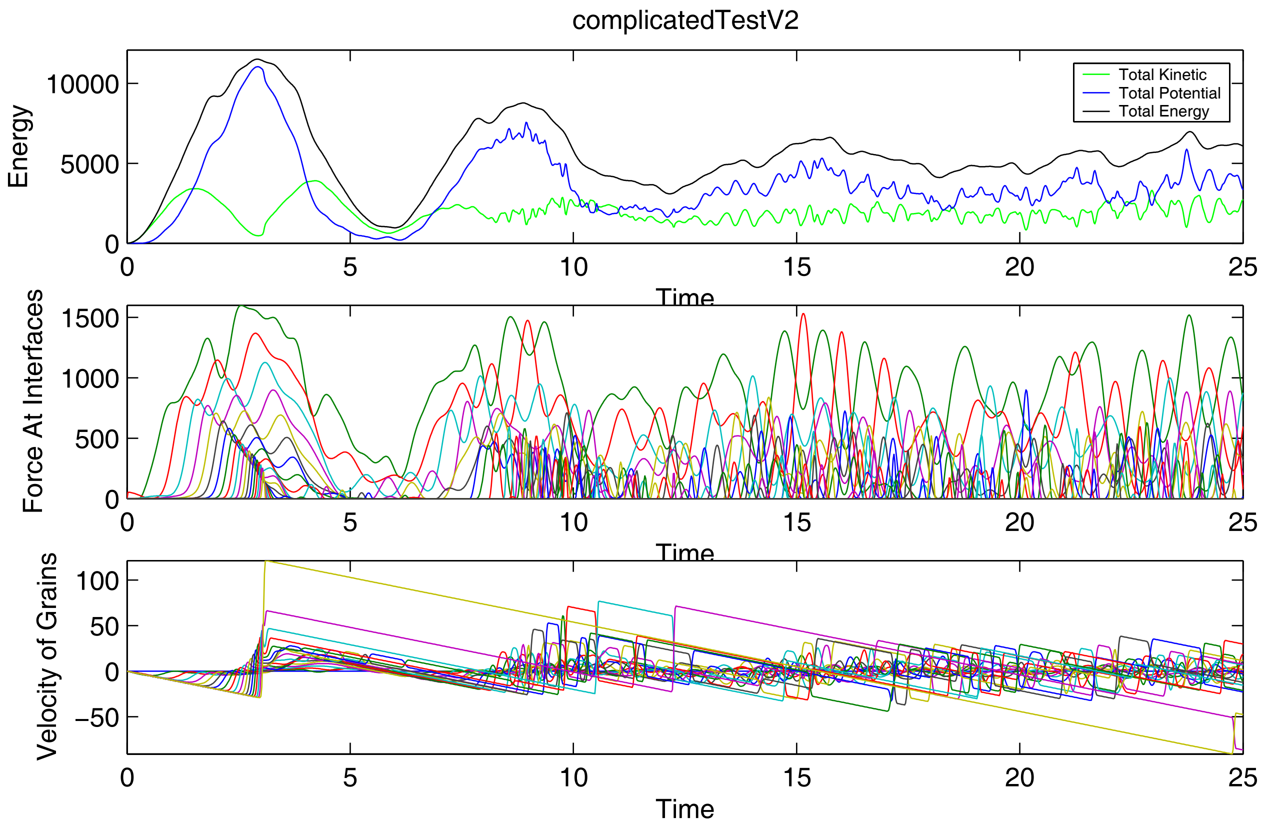
A complicated example part 2:

```
c: Equilibrium and Driving
outputfiles: v p k i x o
output file extension: .dat
outputinterval: 0.02
outputprecision: 5
fileName: complicatedTestV2
DEFAULTGRAIN:sphere#1|rho#8|poisson#0.3|youngs#200|
c: set up the chain
 ChainPattern: [a#20]
subunit: a | radius#1
tapering ratio : 0.9

c: now move grain 1 around with saw tooth wave  function
loadstep: number # 1 | solver : runge kutta four | start # 0 | step # 0.001 |  stop # 25 ↩
    |
c: saw tooth, first couple of terms
freeze grain : loadstep # 1 | grains: 1
driving function: load step # 1 | add or set : set | grains : 1 | variable : displacement↩
     | function type : poly | params : 0.5
driving function: load step # 1 | add or set : add | grains : 1 | variable : displacement↩
     | function type : sine | params : −.318,3.1415,0
driving function: load step # 1 | add or set : add | grains : 1 | variable : displacement↩
     | function type : sine | params : −.159,6.28,0
driving function: load step # 1 | add or set : add | grains : 1 | variable : displacement↩
     | function type : sine | params : −.106,9.4248,0
driving function: load step # 1 | add or set : add | grains : 1 | variable : displacement↩
     | function type : sine | params : −.079,12.566,0
initial condition: load step # 1 | variable : displacement | add value : 0.5 | grains :2
body acceleration : loadstep # 1 | const # −9.8
leftwall: load step # 1 | subunit : a

end
```

These are the results. The drop in energy at early time is after tightening the convergence tolerances.


complicatedTestV2
Total Kinetic
Total Potential
Total Energy
Energy
Force At Interfaces
Velocity of Grains
Time