PULSEDYN (v1.1) – User Manual

-Rahul Kashyap

(2018)

The software contains the following potentials

Toda:

$$V_{i,i+1} = \left(\frac{k1}{k2}\right)\exp\left(k2(x_i - x_{i+1})\right) + k1(x_i - x_{i+1}) - k1/k2$$

FPUT cubic + quartic:

$$V_{i,i+1} = \mathrm{k1}\left((x_i - x_{i+1})\right)^2 + \mathrm{k2}\left((x_i - x_{i+1})\right)^3 + \mathrm{k3}\left((x_i - x_{i+1})\right)^4$$

Morse:

$$V_{i,i+1} = k1(\exp\left(k2(x_i - x_{i+1})\right) - 1)^2$$

Lennard-Jones

$$V_{i,i+1} = k2\left[\left(\frac{k1}{k1 + x_i - x_{i-1}}\right)^{12} - 2\left(\frac{k1}{k1 + x_i - x_{i-1}}\right)^{6} + 1\right]$$

The code has the following options to be set in a parameter file:

1. Integration methods available: Gear 5th order predictor corrector and velocity Verlet algorithm.
2. Can adjust values of k1 and k2
3. Can set initial conditions of specific values and types, or load from file
4. Can set the various simulation parameters available in the integrators i.e. time step, total number of recorded time steps etc.

The file should be named "`parameters.txt`" and it should be present in the same folder as the binary file. You don't need to compile the files/program again. If you're using Windows, just double click the binary file and make sure the parameter file is in the same folder. Alternatively, you can download and install Code::Blocks software with the necessary compilers and compile and run it yourself. The second option is very useful if you need to change the code itself. If you're working on Linux, you can double click on the object file the same way and the code will run. However, you won't see the console window showing you the progress of the simulation. To see the console output, open a terminal window, navigate to the object file directory and use the command `./PULSEDYN` to get the simulation started. If changes are made to the code, make sure the code is recompiled before running it. The default commands require GCC version > 5.2. There is a makefile provided to compiler the code if the user wishes to use PULSEDYN independently of Code::Blocks.

On Linux, just navigate inside the terminal to where the makefile is and type make in the terminal. It will compile the code. The makefile must be placed in the uppermost level of the PULSEDYN directory tree. The object files are placed in the `/obj` folder and the executable is placed in the `/bin` folder. On Linux, the make file can be used to create the executable and can be run directly. On Windows, the make utility must be installed. The makefile has been written in a Linux style to maintain consistency with scientific computing tradition which favors Linux over Windows. So we recommend using Cygwin on Windows. The executable generated by Cygwin can be run inside the Cygwin terminal by navigating to the folder with the PULSEDYN executable and using the command `./PULSEDYN`, but double clicking will not work by default. In order for double clicking to work, place the three `.dll` files provided in the `/dll` folder next to the executable and the executable can then be run as a standalone executable. Alternatively, the makefile can be rewritten in a Windows style for it to work natively on Windows.

The following are the commands for the parameter file (<u>copy paste if you can, all commands are case sensitive</u>)

1. Set the potential and the potential parameters in the following manner
   ```
   model: modelname k1 k2 k3
   ```

   The models available are "`fput`", "`toda`", "`morse`" and "`lennardjones`". The next example shows the fput model with k1 = 0.1 and k2 = 1.0, k3 = 2.0.

   ```
   Ex:
   model: fput 0.1 1.0 2.0
   ```

   You can change the value of the parameters by changing the numbers next to them. Always leave a space to the right of the semicolon. Otherwise the code does not run properly.

2. Set the integration method
   If you want to use Gear 5th order algorithm use
   ```
   method: gear5
   ```

   If you want to use velocity Verlet give the command
   ```
   method: velocityverlet
   ```

   Always use a space after the semicolon otherwise the code will run with errors.

3. Specify the chain length and the simulation parameters as follows. In the example below, the system size is 100, time step is 0.00001, recorded steps are 100000 and the data is recorded/sampled every 10000 time iterations.

   ```
   systemsize: 100
   timestep: 0.00001
   recsteps: 100000
   printint: 1000
   ```

Here, the command “printint” sets the sampling interval. In the case shown above, the data is recorded after 10000 iterations. The command “recsteps” determines how many time points are recorded. In the above example, data at 100000 time points are recorded for each particle.

Along with the time step, “printint” and “recsteps” determine the total system time that the simulation runs to. In the above case, each iteration is 0.00001 in the time units you have chosen. If data is recorded every 1000 iterations, then every time the data is recorded corresponds to an increment in system time of $\Delta t = 0.01$. Recording data for 100000 time points pushes the final system time to $t = 10000*0.01$ or $t = 1000$ in system time units at the end of the simulation.

Again, use a space after the semicolon. The default for system size is 100. The default time step is 0.01. The default for total recorded data points is 100. The default sampling rate is automatically chosen to be `1/timestep`. However, if you specify a time step greater than 1.0 the sampling interval becomes less than 1. So the code resets it to 10. Setting the sampling interval separately is the best way to have control over this parameter.

Setting large time steps in conjunction with bad values of parameters i.e. if the units have not been rescaled to reduce order of magnitude differences, can cause numerical instability. So make sure that the time step chosen is appropriate for the problem. For instance, studying a high frequency object like a localized nonlinear excitation would probably require smaller time steps and smaller sampling intervals as opposed to the FPUT recurrence problem where the energy is initially fed into the lowest mode of the system.

4. Specify initial conditions as follows:

   a) You can specify which particle, which kind of perturbation (initial velocity or initial position) and the value of the perturbation in the following way

   ```
   init: particleNo. Typeofperturbation value
   ```

   Ex:

   ```
   init: 40 pos 0.1
   init: 60 vel -0.1
   ```

   The first command assigns a position perturbation to particle 40 of value 0.1 and the second one assigns velocity perturbation of value -0.1 to particle 60. You can add as many perturbations as you wish. Make sure that the chain is long enough.

   Typically, a solitary wave (SW) is seeded at a site by giving a particle a velocity perturbation. A localized nonlinear excitation (LNE) can be seeded by assigning equal and opposite position perturbations to adjacent particles.

   Two SWs can be seeded in the following way.

```
Ex:

init: 40 vel 0.1
init: 60 vel -0.1
```

LNE can be seeded as follows.

```
Ex:

init: 40 pos 0.1
init: 41 pos -0.1
```

b) One can add uniformly distributed random perturbations at specific sites using the following template

```
init: particleNo type random lowervalue uppervalue

Ex:

init: 20 vel random 0.2 0.5
```

The above example adds a velocity perturbation to particle 20 between the range 0.2 to 0.5.

c) One can load values of position, velocity and acceleration from file in the following way.

```
init: file filename

Ex:

init: file init.txt
```

The above example loads the values of position, velocity and acceleration from a restart file from a previous run.

The file must have the following format. The values must be numeric in the file. There should be three columns. The first one for position, the second one for velocity and the third one for acceleration. If you leave out a column, the code will behave unexpectedly and crash. Even if you don't want to set all the positions or velocities, leave a 0.0. **Don't leave any value as a blank space. Be very careful to not leave trailing blank lines at the end of files or beginning of input files. There is no provision to have comments in the initial condition files.**

It should also be mentioned here that in the Lennard-Jones potential, k1 sets the bond length explicitly. Therefore, if adjacent particles have too much energy, they might get too close and try to crossover. Particles crossing over in the chain in not supported by PULSEDYN. This would cause the energies to blow up and the simulation would become unstable thereby giving unphysical results. Therefore, bond lengths and energies in the Lennard-Jones potential must be set carefully. For the Lennard-Jones potential, at all times $|x_{i+1} - x_i| < k1$, since here k1 sets the bond length between two adjacent particles. Even in general, it is recommended that for best numerical accuracy, rescaling the system to units in a way that reduces differences in orders of magnitude should be done before setting up simulations. For instance, a value of k1 = 1 and initial velocity = 100 at a particle would likely give you more inaccuracy than a system of units in which k1 = 0.5 and initial velocity becomes 0.7.

5. Boundary conditions can be set as

```
boundary: side type
```

The boundary sides are “left and “right” and the types are “fixed”, “open” and periodic. By default, the boundaries are fixed. Also, if you pick one of the boundaries to be “periodic”, the other boundary is automatically set to be periodic. You can mix and match the “open” and “closed” boundaries. When using open boundaries, the system can show drifting.

```
Ex:

boundary: left fixed
```

The above example sets a fixed boundary at the left end.

6. The default mass of each particle is set to 1.0. Mass impurities can be added to the chain using the following command.

```
mass: particleNo. Value
```

```
Ex:

mass: 20 0.2
```

The above example sets the mass of the 20th particle to 0.2. There are some things to be careful about. Make sure that the chain is long enough else the code might crash. For instance, don’t set the mass of particle 20 when your system size is defined to be 15. If you

end up setting negative mass or zero mass, the code will automatically reassign the mass to its default value i.e. 1.0. It is still not recommended that you set bad values for mass.

7. The system also admits forces and dissipation. Forces can be added to each particle specifically or to the end entire chain in the following manner. By default, no force is set.

   ```
   force: particleNo. forceType A t1 t2 t3 ramp
   ```

   In the above command, the force is set at a specified particle using a specific type of force. The force types available are "sine" and "cosine". Further,
   a) t1 – start time of the force.
   b) t2 – time period of the force
   c) t3 – time when forcing ends
   d) ramp – frequency ramp such that $f(t) = f_o + ramp * time$
   e) A – amplitude of the force

   Ex:

   ```
   force: 20 sine 0.5 0.0 0.7 100 0.2
   ```

   The above example sets the force on particle 20 to be a "sine" force of amplitude 0.5 starting at t = 0.0 and ending at t = 100 with a period of 0.7. The force ramp is 0.2 i.e. after every time unit, the frequency is increased by 0.2 units.

   The force can also be set for all particles at a time using the value "all" for particle No.

   Ex:

   ```
   force: all sine 0.5 0.0 0.7 100 0.2
   ```

   The same force as in the previous example is now set for all particles rather than just particle 20. The times t1, t2 and t3 are all in system time. So make sure you do the conversions (similar to the ones in part 3 of this manual) before entering those numbers in.

8. Dissipation can be added as a velocity dependent dissipation term of the form γv in the following manner. Default value of dissipation is 0.

   ```
   dissipation: particleNo. gammaValue
   ```

   Ex:

   ```
   dissipation: 20 0.5
   ```

   The above example sets the value of gamma at 0.5 for particle 20. Dissipation also recognizes the "all" keyword.

   Ex:

   ```
   dissipation: all 0.5
   ```

The above example sets the value of dissipation to 0.5 for all particles.

9. The code outputs the following data as numerical arrays. The column is the particle number and the row is the time.
    1. `ke.dat` – This file contains the kinetic energy of all the particles in the chain as a function of time.
    2. `pe.dat` - This file contains the potential energy of all the springs in the chain as a function of time.
    3. `position.dat` - This file contains the displacement about equilibrium of all the particles in the chain as a function of time.
    4. `velocity.dat` - This file contains the velocity of all the particles in the chain as a function of time.
    5. `acceleration.dat` - This file contains the acceleration of all the particles in the chain as a function of time.

    In addition to the dynamical variables above, the code also outputs the following data

    6. `mass.dat` – This file contains the masses of each particle. In this case they have all been uniformly set to 1.
    7. `totalEnergy.dat` – This file contains the total energy of the chain as a function of time.
    8. `restart.dat` – This stores all the values of position, velocity and acceleration at the end of the simulation in the same format as the initial conditions file format.

Final thoughts and suggestions:

1. **Always enter the system, method and system size first.**

2. **Whenever you set the values at a particular particle, make sure your chain size has been set before and can accommodate particle of that index. For instance, don't set any values for particle 50 when your system size is 20.**

3. When reading in values from a file, make sure that the number of entries for the position, velocity and acceleration in the file is the same as the system size you have entered in the parameter file.

   Even better, don't enter the system size command at all if you're reading from a file. The code will assume that the size of the system is the number of elements in the columns of position, velocity and acceleration each.

4. If you are driving the system or adding dissipation, choose the gear5 algorithm. Velocity-Verlet is an algorithm that deals only with Hamiltonian systems. If you add forces or dissipation, the algorithm would blow up quite spectacularly. So, if you select velocity-Verlet and still give a

forcing or dissipation command, the code simply ignores the forces and simulates with the other parameters you have set.

5. All commands are case sensitive and the semicolon matters. Leave a space between the different keywords.

6. Make sure you set k1, k2 and k3 appropriately. If you set a value that corresponds to unstable spots in the potential, the code will blow up.

7. Make sure your time step is small enough that you are recovering physically meaningful results.

8. If there are older files bearing the same name as those in the output of the program they will be deleted. Use caution! This is true especially if you are resuming simulations from a previous simulation. Even though the code will accept restart.dat from a previous run, it will overwrite that file with a new restart.dat. So if you want to read data from a restart file, rename it to init.txt or some other name so your initial conditions file doesn't get overwritten.

9. The program also outputs a `log.txt` file. In this file, it shows the progress of code as it steps through each line of the parameter file and it should help in troubleshooting your parameter file. It also prints the running time.

10. You can leave blank lines in the parameter file if you wish. You can also comment out lines in your parameter file by adding a # symbol followed by a space at the beginning of the line you want commented out. Each comment line needs a # following by a space to be recognized as a comment.

Notes: