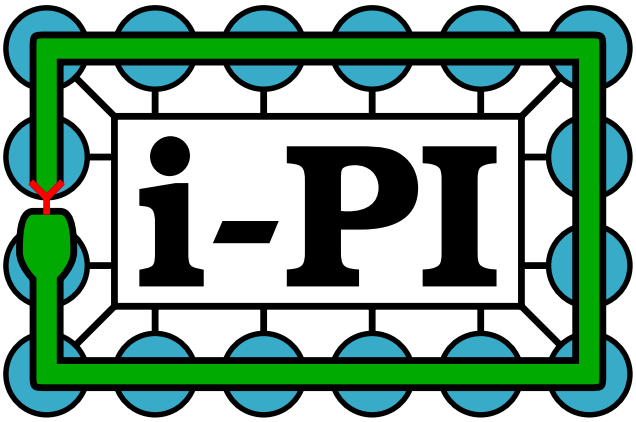

A Python interface for Path Integral Molecular Dynamics

v. 1.0

Contents

1 About i-PI **1**
- 1.1 Manual structure . . . 1
- 1.2 Path Integral Molecular Dynamics . . . 2
- 1.3 Implementation . . . 3
  - 1.3.1 Automated evaluation (depend objects) . . . 3
  - 1.3.2 Communication protocol . . . 3
  - 1.3.3 Internal units . . . 4
- 1.4 Core features . . . 4
- 1.5 Licence and credits . . . 5
  - 1.5.1 Contributors . . . 5
- 1.6 On-line resources . . . 5
  - 1.6.1 Python resources . . . 5
  - 1.6.2 Client code resources . . . 5
  - 1.6.3 i-PI resources . . . 6

2 Getting started **7**
- 2.1 Installing i-PI . . . 7
  - 2.1.1 Requirements . . . 7
    - 2.1.1.1 Using the setup.py module . . . 7
  - 2.1.2 i-PI download . . . 8
  - 2.1.3 Installing NumPy . . . 8
  - 2.1.4 PyFFTW . . . 9
- 2.2 Installing clients . . . 10
  - 2.2.1 Patching CP2K . . . 10
  - 2.2.2 Patching Quantum-Espresso . . . 10
  - 2.2.3 Patching LAMMPS . . . 11
  - 2.2.4 Writing a patch . . . 11
- 2.3 Running i-PI . . . 11
  - 2.3.1 Running the i-PI server . . . 11
  - 2.3.2 Running the client code . . . 12
    - 2.3.2.1 Built-in, example client . . . 12
    - 2.3.2.2 CP2K . . . 12
    - 2.3.2.3 Quantum-Espresso . . . 13
    - 2.3.2.4 LAMMPS . . . 13
  - 2.3.3 Running on a HPC system . . . 14
- 2.4 Testing the install . . . 15

3 User guide **17**
3.1 Input files . . . 17
3.1.1 Input file format and structure . . . 17
3.1.1.1 Overriding default units . . . 18
3.1.2 Initialization section . . . 19
3.1.2.1 Configuration files . . . 19
3.1.2.2 Initialization from checkpoint files . . . 20
3.2 Output files . . . 20
3.2.1 Properties . . . 20
3.2.2 Trajectory files . . . 25
3.2.3 Checkpoint files . . . 26
3.2.3.1 Soft exit and RESTART . . . 26
3.3 Distributed execution . . . 27
3.3.1 Communication protocol . . . 27
3.3.2 Parallelization . . . 28
3.3.3 Sockets . . . 28
3.3.4 Running i-PI over the network . . . 29
3.3.4.1 Understanding the network layout . . . 29
3.3.4.2 ssh tunnelling . . . 30

4 Input reference **32**
4.1 SIMULATION . . . 32
4.2 INITIALIZER . . . 34
4.3 INITFILE . . . 37
4.4 INITPOSITIONS . . . 37
4.5 INITMOMENTA . . . 37
4.6 INITVELOCITIES . . . 38
4.7 INITLABELS . . . 38
4.8 INITMASSES . . . 38
4.9 INITCELL . . . 39
4.10 INITTHERMO . . . 39
4.11 ENSEMBLE . . . 39
4.12 FORCES . . . 40
4.13 SOCKET . . . 41
4.14 CELL . . . 42
4.15 BEADS . . . 42
4.16 ATOMS . . . 43
4.17 NORMALMODES . . . 44
4.18 BAROSTAT . . . 44
4.19 THERMOSTATS . . . 45
4.20 PRNG . . . 46
4.21 OUTPUTS . . . 47
4.22 CHECKPOINT . . . 49
4.23 PROPERTIES . . . 49
4.24 TRAJECTORY . . . 50

5 A simple tutorial **51**
5.1 Part 1 - *NVT* Equilibration run . . . 51
5.1.1 Client code . . . 51
5.1.2 Creating the xml input file . . . 52
5.1.2.1 Initializing the configurations . . . 52
5.1.2.2 Creating the server socket . . . 55
5.1.2.3 Generating the correct ensemble . . . 55
5.1.2.4 Customizing the output . . . 56
5.1.3 Running the simulation . . . 60
5.1.4 Output data . . . 60
5.2 Part 2 - *NPT* simulation . . . 61
5.2.1 Modifying the RESTART file . . . 61
5.2.2 Initialization from RESTART . . . 62
5.2.3 Running the simulation . . . 62
5.3 Part 3 - A fully converged simulation . . . 63

6 Troubleshooting **64**
6.1 Input errors . . . 64
6.2 Initialization errors . . . 65
6.3 Output errors . . . 67
6.4 Socket errors . . . 68
6.5 Mathematical errors . . . 68

About i-PI

i-PI is a Path Integral Molecular Dynamics (PIMD) interface written in Python, designed to be used together with an *ab initio* evaluation of the interactions between the atoms. The main goal is to decouple the problem of evolving the ionic positions to sample the appropriate thermodynamic ensemble and the problem of computing the inter-atomic forces.

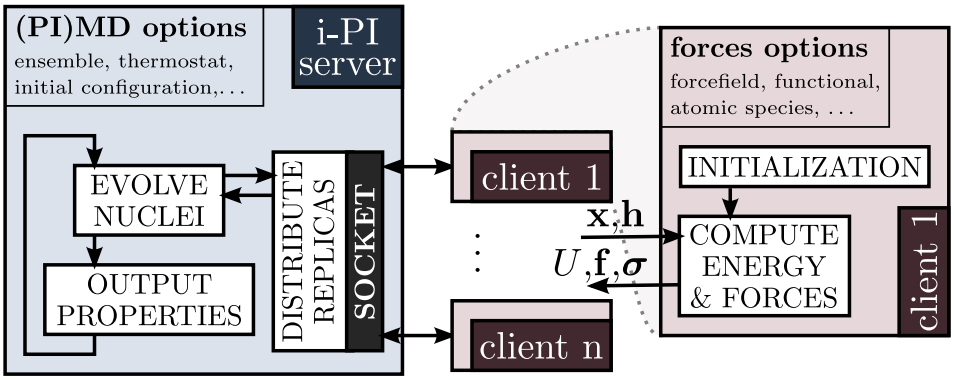

Figure 1.1: Schematic representation of the functioning of i-PI.

The implementation is based on a client-server paradigm, where i-PI acts as the server and deals with the propagation of the nuclear dynamics, whereas the calculation of the potential energy, forces and the potential energy part of the pressure virial is delegated to one or more instances of an external code, acting as clients. Since the main focus is on performing *ab initio* PIMD – where the cost of the force evaluation is overwhelming relative to the ionic dynamics – clarity has been privileged over speed. Still, the implementation of i-PI is efficient enough that it can be used with empirical forcefields to perform simple benchmarks and preparatory simulations.

1.1 Manual structure

This manual will be structured as follows:

- In chapter 1 we briefly discuss the basis for PIMD and some of the specialized techniques used in i-PI.

- In chapter 2 we will discuss how to install and run the code, and test that it is working.
- In chapter 3 we explain in more detail the form of the input and output files and how the communication between the client and server codes is done.
- In chapter 4 a full list of the major classes used in the code is given, along with the appropriate tag names and a brief description of all the fields that can be specified in the xml input file.
- In chapter 5 we give a simple step-by-step walkthrough of an example i-PI simulation.
- In chapter 6 we list some of the more commonly encountered problems, and their solutions.

1.2 Path Integral Molecular Dynamics

Molecular dynamics (MD) is a technique used to study the properties of a system of interacting particles by applying Newton's equations of motion to produce trajectories which can be used to efficiently explore the phase space. This can be used to calculate many equilibrium and dynamical properties and to study systems from isolated gas molecules to condensed phase bulk materials.

However, while this technique has been very successful, in most MD implementations the assumption is made that the nuclei behave as classical particles, which for light nuclei such as hydrogen is often a very poor approximation as the effect of zero-point energy (ZPE) and quantum tunnelling can be large. For example, even at room temperature the vibrational frequency of an OH stretch in water is over 15 times larger than the available thermal energy, and so this motion will be highly quantized. The current state-of-the-art method to include nuclear quantum effects (NQE) in the calculation of static properties of condensed phase systems is path integral molecular dynamics (PIMD).

PIMD generates the quantum-mechanical ensemble of a system of interacting particles by using MD in an extended phase space. This is derived from the path integral formalism [1], which relates the statistics of a collection of quantum particles to those of a set of classical ring polymers, a ring polymer being a number of replicas of a particle coupled by harmonic springs. This so-called classical isomorphism is exact in the limit as the number of replicas goes to infinity, but in practice is converged numerically with only a finite number.

This then allows quantum phase space averages to be calculated from classical trajectories, with only about an order of magnitude more computing time than would be required for standard MD. Also, since PIMD is simply classical MD in an extended phase space, many of the techniques developed to improve the scope and efficiency of MD simulations can be applied straightforwardly to the equivalent PIMD calculations [2, 3]. Finally, several techniques designed specifically for PIMD simulations are now available to increase the rate of convergence with respect to the number of replicas used [4–9], further reducing the computational overhead of the method. All of these facts mean that it is now feasible to do PIMD simulations with thousands of molecules, or even to use *ab initio* electronic structure calculations to propagate the dynamics for small systems.

Furthermore, the framework used to run PIMD simulations can be adapted to generate approximate quantum dynamical information [10–13], and so can also be used to calculate correlation functions. While real-time quantum coherences cannot be captured, the inclusion of quantum statistical information and the rapid decoherence observed in condensed phase systems mean that in many cases very accurate results can be obtained from such approximate treatments of quantum dynamics [14].

1.3 Implementation

1.3.1 Automated evaluation (depend objects)

i-PI uses a caching mechanism with automatic value updating to make the code used to propagate the dynamics as simple and clear as possible. Every physical quantity that is referenced in the code is created using a "depend" object class, which is given the parameters on which it depends and a function used to calculate its value.

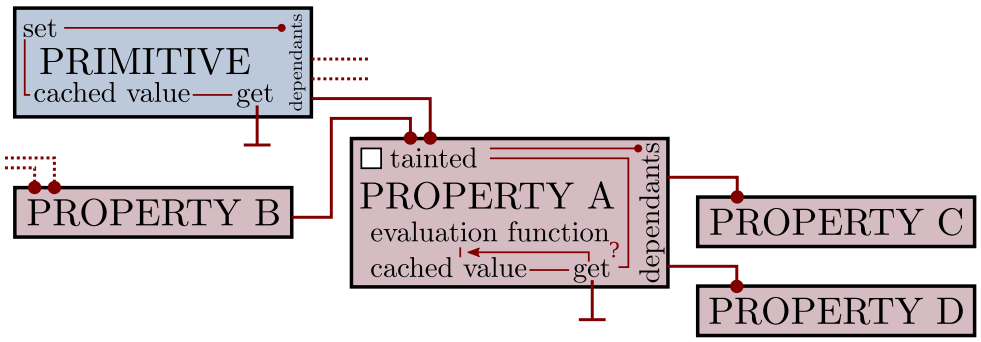

Figure 1.2: Schematic overview of the functioning of the *depend* class used as the base for properties and physical quantities in i-PI. A few "primitive" quantities – such as atomic positions or momenta – can be modified directly. For most properties, one defines a function that can compute based on the value of other properties. Whenever one property is modified, all the quantities that depend on it are marked as tainted, so that – when the value of one of the properties is used, the function can be invoked and the updated value obtained. If a quantity is not marked as tainted, the cached value is returned instead.

"Depend" objects can be called to get the physical quantity they represent. However, they have further functionality. Firstly, once the value of a "depend" object has been calculated, its value is cached, so further references to that quantity will not need to evaluate the function that calculates it. Furthermore, the code keeps track of when any of the dependencies of the variable are updated, and makes sure that the quantity is automatically recomputed when it is needed.

This choice makes implementation slightly more complex when the physical observables are first introduced as variables, as one has to take care of stating their dependencies as well as the function that computes them. However, the advantage is that when the physical quantities are used, in the integrator of the dynamics or in the evaluation of physical properties, one does not need to take care of book-keeping and the code can be clean, transparent and readable.

1.3.2 Communication protocol

Since i-PI is designed to be used with a wide range of codes and platforms, it has to rely on a simple and robust method for communicating between the server and

client. Even though other choices are possible, and it should be relatively simple to implement other means of communication, the preferred approach relies on sockets as the underlying infrastructure. Both Internet and Unix domain sockets can be used: the latter allow for fast communication on a single node, whereas the former make it possible to realise a distributed computing paradigm, with clients running on different nodes or even on different HPC facilities. In order to facilitate implementation of the socket communication in client codes, a simple set of C wrappers to the standard libraries socket implementation is provided as part of the i-PI distribution, that can be used in any programming language that can be linked with C code.

As far as the communication protocol is concerned, the guiding principle has been keeping it to the lowest common denominator, and avoiding any feature that may be code-specific. Only a minimal amount of information is transferred between the client and the server; the position of the atoms and cell parameters in one direction, and the forces, virial and potential in the other.

For more details about sockets and communication, see 3.3.

1.3.3 Internal units

All the units used internally by i-PI are atomic units, as given below. By default, both input and output data are given in atomic units, but in most cases the default units can be overridden if one wishes so. For details on how to do this, see 3.1.1.1 and 3.2.1.

Unit	Name	S.I. Value
Length	Bohr radius	5.2917721e-11 m
Time	N.A.	2.4188843e-17 s
Mass	Electron mass	9.1093819e-31 kg
Temperature	Hartree	315774.66 K
Energy	Hartree	4.3597438e-18 J
Pressure	N.A.	2.9421912e13 Pa

1.4 Core features

The functionality of i-PI includes:

- Thermostats for constant temperature ensembles, including:
  - Local and global stochastic thermostats [15, 16], with optional optimized sampling of the ring polymer normal mode coordinates [2].
  - Optimal sampling generalized Langevin equation (GLE) thermostats [17].
  - Path integral + GLE (PI+GLE) thermostats [5] for accelerating the convergence of the potential energy with respect to the number of replicas, as well as the more recent PIGLET method [18], which also accelerates the convergence of the kinetic energy.
- Barostats for constant pressure ensembles [3, 19].
- Ring polymer contraction [4].
- Scaled path finite difference energy and heat capacity estimators [20].

- Displaced path momentum distribution estimator [21].
- Dynamical property calculation modes:
  - Ring polymer molecular dynamics [12].
  - Partially-adiabatic centroid molecular dynamics [22, 23].

1.5 Licence and credits

Most of this code is distributed under the GPL licence. For more details see www.gnu.org/licences/gpl.html. So that they can easily be incorporated in other codes, the files in the directory "drivers" are all held under the MIT licence. For more details see https://fedoraproject.org/wiki/Licensing:MIT.

If you use this code in any future publications, please cite this using [cpc paper citation].

1.5.1 Contributors

i-PI was originally written by M. Ceriotti and J. More at Oxford University, together with D. Manolopoulos. T. Spura and M. Rossi contributed to the initial development of the code, R. Distasio and B. Santra for helped creating patches for Quantum Espresso.

1.6 On-line resources

1.6.1 Python resources

For help with Python programming, see www.python.org. For information about the NumPy mathematical library, see www.numpy.org, and for worked examples of its capabilities see www.scipy.org/Tentative_NumPy_Tutorial. Finally, see http://hgomersall.github.io/pyFFTW/ for documentation on the Python FFTW library that is currently implemented with i-PI.

1.6.2 Client code resources

There are currently client patches available for Quantum Espresso and CP2K for all currently maintained versions. It should also be possible to adapt these patches to other versions of the codes with minor modifications. For more information about Quantum Espresso and CP2K, go to www.quantum-espresso.org and www.cp2k.org respectively.

There is also a patch for the latest version of the LAMMPS empirical potential MD code. More information on this code can be found at http://lammps.sandia.gov/index.html.

There are several Fortran and C libraries that most client codes will probably need to run, such as FFTW, BLAS and LAPACK. These can be found at www.fftw.org, www.netlib.org/blas and www.netlib.org/lapack respectively.

These codes do not come as part of the i-PI package, and must be downloaded separately. See chapter 2.2 for more details of how to do this.

1.6.3 i-PI resources

For more information about i-PI and to download the source code go to http://imx-cosmo.github.io/gle4md/, where one can also obtain colored-noise parameters to run Path Integral with Generalized Langevin Equation thermostat (PI+GLE/PIGLET) calculations.

Getting started

2.1 Installing i-PI

2.1.1 Requirements

i-PI is Python code, and as such strictly speaking does not need to be compiled and installed. The `i-pi` file in the root directory of the distribution is the main (executable) script, and can be run as long as the system has installed:

- Python version 2.4 or greater
- The Python numerical library NumPy

See 2.3.1 for more details on how to launch i-PI.

Additionally, most client codes will have their own requirements. Many of them, including the test client codes given in the "drivers" directory, will need a suitable Fortran compiler. A C compiler is required for the sockets.c wrapper to the sockets standard library. Most electronic structure codes will also need to be linked with some mathematical libraries, such as BLAS, FFTW and LAPACK. Installation instructions for these codes should be provided as part of the code distribution and on the appropriate website, as given in 1.6.2. Patching for use with i-PI should not introduce further dependencies.

2.1.1.1 Using the setup.py module

While the `i-pi` file can be used to run any i-PI simulation, it is often more convenient to install the package to the system's Python modules path, so that it is accessible by all users and can be run without specifying the path to the Python script.

For this purpose we have included a module in the root directory of the i-PI distribution, `setup.py`, which handles creating a package with the executable and all the modules which are necessary for it to run. The first step is to build the distribution using:

```
> python setup.py build
```

Note that this requires the distutils package that comes with the python-dev package.

This creates a "build" directory containing only the files that are used to run an i-PI simulation, which can then be used to create the executable. This can be done in two ways, depending on whether or not the user has root access. If the user does have root

access, then the following command will add the relevant source files to the standard Python library directory:

```
> python setup.py install
```

This will install the package in the /usr/lib/py_vers directory, where py_vers is the version of Python that is being used. This requires administrator priviledges.

Otherwise, one can install i-PI in a local Python path. If such path does not exist yet, one must create directories for the package to go into, using:

```
> mkdir ~/bin
> mkdir ~/lib/py_vers
> mkdir ~/lib/py_vers/site-packages
```

Next, you must tell Python where to find this library, by appending to the Linux environment variable PYTHONPATH, using:

```
> export PYTHONPATH=$PYTHONPATH:~/lib/py_vers/site-packages/
```

Finally, the code can be installed using:

```
> python setup.py install --prefix=~
```

Either way, it will now be possible to run the code automatically, using

```
> i-pi input_file.xml
```

2.1.2 i-PI download

A tar file can be downloaded from the website http://imx-cosmo.github.io/gle4md/. To install this you need to input the following command:

```
> tar xf [ipi_tar_file.tar]
```

You can also obtain a local clone of the git repository (listed on http://imx-cosmo.github.io/gle4md/) using:

```
> git clone [repository name]
```

The i-PI executable will run immediately, without needing to be installed or compiled, but we include a setup.py module in the main directory so it can be installed to the Python tree if so desired.

2.1.3 Installing NumPy

NumPy is the standard Python mathematics library, and is used for most of the array manipulation and linear algebra in i-PI. It should be installed alongside most standard Python environments on HPC facilities. Otherwise, it is generally relatively straightforward to install it.

In any case you must first obtain the NumPy code, which can be downloaded as a tar file from http://www.numpy.org. If the version of NumPy being installed is given by “np_vers”, this can be extracted using:

```
> tar xzf np_vers.tar.gz
```

Before installing this code it first needs to be configured correctly. Note that this requires the distutils package that comes with the python-dev package. Assuming that the required software is installed, the NumPy package is built using:

```
> python setup.py build
```

The next step is to install NumPy. By default the download is to the directory /usr/local. If you have root access, and so can write to /usr, then all that needs to be done to finish the install is:

```
> python setup.py install
```

If you do not have root access, then the next step depends on which version of Python is beind used. With versions 2.6 or later there is a simple command to automatically download into the directory $HOME/local:

```
> python setup.py install --user
```

With Python 2.4/2.5 the process is a little more involved. First you must explicitly install the package in the directory of choice, "np_dir" say, with the following command:

```
> python setup.py install --prefix=np_dir
```

Next, you must tell Python where to find this library, by appending to the Linux environment variable PYTHONPATH. If you are using Python version "py_vers", then the NumPy libraries will have been installed in the directory "np_dir/lib/py_vers/site-packages", or a close analogue of this. In the above case the following command will allow the Python interpreter to find the NumPy libraries:

```
> export PYTHONPATH=$PYTHONPATH:np_dir/lib/py_vers/site-packages
```

Now Python scripts can import the NumPy libraries using:

```
import numpy
```

2.1.4 PyFFTW

Some of the steps in the dynamics algorithm involve a change of variables from the bead coordinates to the normal modes of the ring polymers. Currently, this transformation is, at least by default, computed using a fast-Fourier transform (FFT) library within the NumPy distribution. This however is not the only distribution that could be used, and indeed faster stand-alone versions exist. The gold-standard FFT library is the FFTW library, which is a set of C libraries that have been heavily optimized for a wide range of applications. There have been a number of Python wrappers built around the FFTW library, one of which is currently interfaced with i-PI. This code can be found at https://github.com/hgomersall/pyFFTW, and has documentation at http://hgomersall.github.io/pyFFTW/.

This code has the following dependencies:

- Python version 2.7 or greater
- Numpy version 1.6 or greater
- FFTW version 3.2 or greater

This can be installed in the same way as NumPy, except using the code distribution above, or using various installation packages as per the instructions on the above documentation. Note that no other options need to be specified in the input file; i-PI will check to see if this library is available, and if it is it will be used by default. Otherwise the slower NumPy version will be used.

2.2 Installing clients

2.2.1 Patching CP2K

You can download the source code for CP2K at www.cp2k.org/. This can either be downloaded as a tar file, which can be extracted in the same way as the Python and NumPy libraries above, or via a svn client.

As an example of the latter, CP2K version 2.4 can be downloaded to a directory "cp2k-2.4" using the command:

```
> svn checkout svn://svn.code.sf.net/p/cp2k/code/branches/cp2k-2_4-branch cp2k-2.4
```

Before CP2K can run with i-PI, the code must be adapted to use the socket interface. For this purpose i-PI is distributed with patch files in the directory "patches". When applied to a clean distribution, these will adjust the source code to make it compatible with i-PI.

For example, if we take CP2K version 2.4.0, and assuming that you are currently in the top level directory of the CP2K distribution, the patch can be applied to the source code using:

```
> patch -p1 < ipidir/i-pi/patches/cp2k-2.4.0_ipi.patch
```

where ipidir is the directory containing the i-PI source code.

After this, continue the compilation as per the instructions at www.cp2k.org/ to complete the install. There are currently patches available for CP2K versions 2.4, 2.3, 2.2 and 2.1.

2.2.2 Patching Quantum-Espresso

You can download the source code for Quantum Espresso at www.quantum-espresso.org/ as a tar file. The tar file can be extracted in the same way as the Python and NumPy libraries above. As for CP2K above, we include patch files to adapt the Quantum Espresso source code so that it will work with i-PI. Taking version 5.0 as an example, and again assuming you are in the top level directory of the Quantum Espresso distribution, this patch can be applied to the source code using:

```
> patch -p1 < ipidir/i-pi/patches/espresso-5.0_ipi.patch
```

After this, continue the compilation as per the instructions at www.quantum-espresso.org/ to complete the install. There are currently patches available for Quantum Espresso versions 5.0, 4.3, 4.2 and 4.1.3.

2.2.3 Patching LAMMPS

You can download the source code for LAMMPS at http://lammps.sandia.gov/index.html. This can either be downloaded as a tar file, which can be extracted in the same way as the Python and NumPy libraries above, via a svn client, which can be run in the same way as for CP2K, or from a git repository, which can be downloaded in the same way as for i-PI.

As for CP2K above, we include patch files to adapt the LAMMPS source code so that it will work with i-PI. Currently we only provide a patch file for a recent version of LAMMPS, which can be applied to the source code using:

```
> patch -p1 < ipidir/i-pi/patches/lammps-26Aug13_ipi.patch
```

After this, continue the compilation as per the instructions at http://lammps.sandia.gov/index.html to complete the install. Note that the optional libraries CLASS2, KSPACE, MANYBODY and MOLECULE will all need to be included to run the examples provided in the "examples" directory.

2.2.4 Writing a patch

If you have edited a client code, and wish to make a patch available for the new version, then this can be done very simply. If your edited code is in a directory "new", and a clean distribution is held in a directory "old", then a patch "changes.patch" can be created using:

```
> diff -rupN old/ new/ > changes.patch
```

2.3 Running i-PI

i-PI functions based on a client-server protocol, where the evolution of the nuclear dynamics is performed by the i-PI server, whereas the energy and forces evaluation is delegated to one or more instances of an external program, that acts as a client. This design principle has several advantages, in particular the possibility of performing PIMD based on the forces produced by one's favourite electronic structure/empirical force field code. However, it also makes running a simulation slightly more complicated, since the two components must be set up and started independently.

2.3.1 Running the i-PI server

i-PI simulations are run using the i-pi Python script found in the "i-pi" directory. This script takes an xml-formatted file as input, and automatically starts a simulation as specified by the data held in it. If the input file is called "input_file.xml", then i-PI is run using:

```
> python i-pi input_file.xml
```

This reads in the input data, initializes all the internally used objects, and then creates the server socket. The code will then wait until at least one client code has connected to the server before running any dynamics. Note that until this has happened the code is essentially idle, the only action that it performs is to periodically poll for incoming connections.

2.3.2 Running the client code

2.3.2.1 Built-in, example client

While i-PI is designed with *ab initio* electronic structure calculations in mind, it also includes a Fortran empirical potential client code to do simple calculations and to run the examples.

The source code for this is included in the directory "drivers", and can be compiled into an executable "driver.x" using the UNIX utility make.

This code currently has four empirical potentials hardcoded into it, a Lennard-Jones potential, the Silvera-Goldman potential [24], a 1D harmonic oscillator potential, and the ideal gas (i.e. no potential interaction).

How the code is run is based on what command line arguments are passed to it. The command line syntax is:

```
> driver.x [-u] -h hostname -p port -m [gas|lj|sg|harm] -o parameters [-v]
```

The flags do the following:

-u: Optional parameter. If specified, the client will connect to a unix domain socket. If not, it will connect to an internet socket.

-h: Is followed in the command line argument list by the hostname of the server.

-p: Is followed in the command line argument list by the port number of the server.

-m: Is followed in the command line argument list by a string specifying the type of potential to be used. "gas" gives no potential, "lj" gives a Lennard-Jones potential, "sg" gives a Silvera-Goldman potential and "harm" gives a 1D harmonic oscillator potential.

-o: Is followed in the command line argument list by a string of comma separated values needed to initialize the potential parameters. "gas" requires no parameters, "harm" requires a spring constant, "sg" requires a cut-off radius and "lj" requires the length and energy scales and a cut-off radius to be specified. All of these must be given in atomic units.

-v: Optional parameter. If given, the client will print out more information each time step.

This code should be fairly simple to extend to other pair-wise interaction potentials, and examples of its use can be seen in the "examples" directory, as explained in 2.4.

2.3.2.2 CP2K

To use CP2K as the client code using an internet domain socket on the host address "host_address" and on the port number "port" the following lines must be added to its input file:

```
&GLOBAL
   ...
   RUN_TYPE DRIVER
   ...
&END GLOBAL
```

```
&MOTION
   ...
   &DRIVER
      HOST host_address
      PORT port
   &END DRIVER
   ...
&END MOTION
```

If instead a unix domain socket is required then the following modification is necessary:

```
&MOTION
   ...
   &DRIVER
      HOST host_address
      PORT port
      UNIX
   &END DRIVER
   ...
&END MOTION
```

The rest of the input file should be the same as for a standard CP2K calculation, as explained at www.cp2k.org/.

2.3.2.3 Quantum-Espresso

To use Quantum-Espresso as the client code using an internet domain socket on the host address "host_address" and on the port number "port" the following lines must be added to its input file:

```
&CONTROL
   ...
   calculation='driver'
   srvaddress='host_address:port'
   ...
/
```

If instead a unix domain socket is required then the following modification is necessary:

```
&CONTROL
   ...
   calculation='driver'
   srvaddress='UNIX:host_address:port'
   ...
/
```

The rest of the input file should be the same as for a standard Quantum Espresso calculation, as explained at www.quantum-espresso.org/.

2.3.2.4 LAMMPS

To use LAMMPS as the client code using an internet domain socket on the host address "host_address" and on the port number "port" the following lines must be added to its input file:

```
fix 1  all driver host_address port
```

If instead a unix domain socket is required then the following modification is necessary:

```
fix 1  all driver host_address port unix
```

The rest of the input file should be the same as for a standard LAMMPS calculation, as explained at http://lammps.sandia.gov/index.html.

2.3.3 Running on a HPC system

Running i-PI on a high-performance computing (HPC) system can be a bit more challenging than running it locally using UNIX-domain sockets or using the *localhost* network interface. The main problem is related to the fact that different HPC systems adopt a variety of solutions to have the different nodes communicate with each other and with the login nodes, and to queue and manage computational jobs.

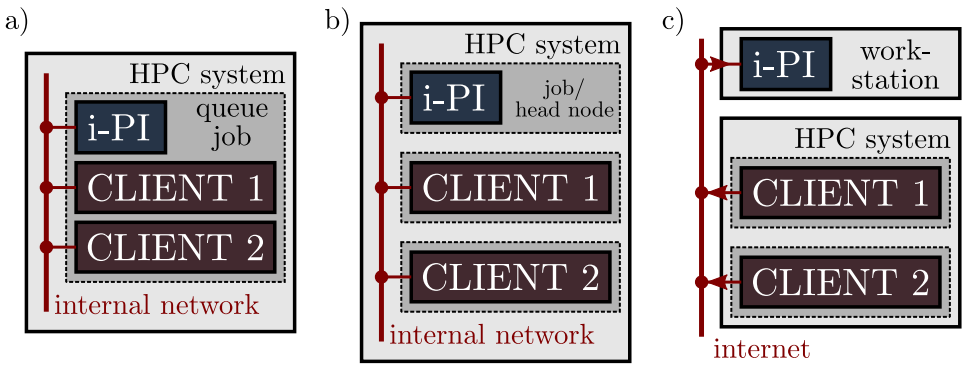

Figure 2.1: Different approaches to run i-PI and a number of instances of the forces code on a HPC system: a) running i-PI and the clients in a single job; b) running i-PI and the clients on the same system, but using different jobs, or running i-PI interactively on the login node; c) running i-PI on a local workstation, communicating with the clients (that can run on one or multiple HPC systems) over the internet.

Figure 2.1 represents schematically three different approaches to run i-PI on a HPC system:

1. running both i-PI and multiple instances of the client as a single job on the HPC system. The job submission script must launch i-PI first, as a serial background job, then wait a few seconds for it to load and create a socket

   ```
   > python i-pi input_file.xml &> log & wait 10
   ```

   Then, one should launch with mpirun or any system-specific mechanism one or more independent instances of the client code. Note that not all queing systems allow launching several mpirun instances from a single job.

2. running i-PI and the clients on the HPC system, but in separate jobs. Since i-PI consumes very little resources, one should ideally launch it interactively on a login node

   ```
   > nohup python i-pi input_file.xml < /dev/null &> log &
   ```

or alternative on a queue with a very long wall-clock time. Then, multiple instances of the client can be run as independent jobs: as they start, they will connect to the server which will take care of adding them dynamically to the list of active clients, dispatching force calculations to them, and removing them from the list when their wall-clock time expires. This is perhaps the model that applies more easily to different HPC systems; however it requires having permission to run on the head node, or having access to a long wall-clock time queue that ensures that i-PI is always active.

3. running i-PI on a simple workstation, and performing communication over the internet with the clients that run on one or more HPC systems. This model exploits in full the distributed-computing model that underlies the philosophy of i-PI and is very robust – as the server can be always on, and the output of the simulation is generated locally. However, this is also the most complicated to set up, as the local workstation must accept in-coming connections from the internet – which is not always possible when behind a firewall – and the compute nodes of the HPC centre must have an outgoing connection to the internet, which often requires ssh tunnelling through a login node (see section 3.3 for more details).

2.4 Testing the install

Several test cases are distributed with the code to ensure that your distribution is working correctly. There are also simple tests to see if the client codes are working correctly.

All the input files are contained in the directory "examples", which is subdivided into the following directories:

tutorial: Contains the input files needed to run the tutorial in 5.

lj: This gives a simple classical Lennard-Jones simulation of Ne. The state points are given by (N, ρ, T) = (864, 0.35, 1.62), (N, ρ, T) = (864, 0.75, 1.069) and (N, ρ, T) = (864, 0.88, 1.095) in reduced Lennard-Jones units, so that the results can be compared to those in [25].

ph2: This simulates para-hydrogen using the isotropic Silvera-Goldman pair potential [24]. There are three directories, "RPMD", "nvt" and "Tuckerman". "RPMD" and "nvt" have tests which can be compared to the results of [26], and "Tuckerman" has tests which can be compared to the results of [3].

qespresso: This has two simple examples to test to see if the Quantum-Espresso client is functioning correctly. There is one simple 4-atom lithium test, and a test using a single water molecule.

harmonic: This has a simple example of a 1D harmonic oscillator. This demonstrates the displaced path integral momentum distribution estimator as given in [21]. As the momentum distribution is known analytically for this simple system, this provides an indication of how well the method is working.

lammps: This has a simple implementation of the q-TIP4P-F empirical water model of [27] using the classical molecular dynamics code LAMMPS. It demonstrates both

the convergence of the PIGLET method [18], as well as the use of ring-polymer contraction methods [4].

This also contains one example using LAMMPS to calculate the interactions between carbon atoms in graphene. This uses the optimized Tersoff parameters for carbon given in [28, 29].

cp2k: Contains the tests for the CP2K client code. Holds input files to run the high-pressure water calculations presented in [cpc publication citation].

User guide

3.1 Input files

3.1.1 Input file format and structure

In order to give the clearest layout, xml formatting was chosen as the basis for the main input file. An xml file consists of a set of hierarchically nested tags. There are three parts to an xml tag. Each tag is identified by a tag name, which specifies the class or variable that is being initialized. Between the opening and closing tags there may be some data, which may or may not contain other tags. This is used to specify the contents of a class object, or the value of a variable. Finally tags can have attributes, which are used for metadata, i.e. data used to specify how the tag should be interpreted. As an example, a 'mode' attribute can be used to select between different thermostatting algorithms, specifying how the options of the thermostat class should be interpreted.

A xml tag has the following syntax:

```
<tag_name attribute_name='attribute_data'>tag_data</tag_name>
```

The syntax for the different types of tag data is given below:

Data type	Syntax
Boolean	<tag>True</tag> or <tag>False</tag>
Float	<tag>11.111</tag> or <tag>1.1111e+1</tag>
Integer	<tag>12345</tag>
String	<tag>string_data</tag>
Tuple	<tag> (int1, int2, ...)</tag>
Array	<tag> [entry1, entry2, ...] </tag>
Dictionary	<tag>{name1: data1, name2: data2, ...}</tag>

Note that arrays are always given as one-dimensional lists. In cases where a multi-dimensional array must be entered, one can use the 'shape' attribute, that determines how the list will be reshaped into a multi-dimensional array. For example, the bead positions are represented in the code as an array of shape (number of beads, 3*number of atoms). If we take a system with 20 atoms and 8 beads, then this can be represented in the xml input file as:

```
<beads nbeads='8' natoms='20'>
   <q shape='(8,60)'>[ q11x, q11y, q11z, q12x, q12y, ... ]</q>
   ...
</beads>
```

If 'shape' is not specified, a 1D array will be assumed.

The code uses the hierarchical nature of the xml format to help read the data; if a particular object is held within a parent object in the code, then the tag for that object will be within the appropriate parent tags. This is used to make the structure of the simulation clear.

For example, the system that is being studied is partly defined by the thermodynamic ensemble that should be sampled, which in turn may be partly defined by the pressure, and so on. To make this dependence clear in the code the global simulation object which holds all the data contains an ensemble object, which contains a pressure variable.

Therefore the input file is specified by having a "simulation" tag, containing an "ensemble" tag, which itself contains a "pressure" tag, which will contain a float value corresponding to the external pressure. In this manner, the simulation class structure can be constructed iteratively.

As a specific example, suppose we want to generate a *NPT* ensemble at an external pressure of 1e-7 atomic pressure units. This would be specified by the following input file:

```
<simulation>
   <ensemble mode='npt'>
      <pressure> 1e-7 </pressure>
      ...
   </ensemble>
   ...
</simulation>
```

To help detect any user error the recognized tag names, data types and acceptable options are all specified in the code in a specialized input class for each class of object. A full list of all the available tags and a brief description of their function is given in chapter 4.

3.1.1.1 Overriding default units

Many of the input parameters, such as the pressure in the above example, can be specified in more than one unit. Indeed, often the atomic unit is inconvenient to use, and we would prefer something else. Let us take the above example, but instead take an external pressure of 3 MPa. Instead of converting this to the atomic unit of pressure, it is possible to use pascals directly using:

```
<simulation>
   <ensemble mode='npt'>
      <pressure units='pascal'> 3e6 </pressure>
      ...
   </ensemble>
   ...
</simulation>
```

The code can also understand S.I. prefixes, so this can be simplified further using:

```
<simulation>
   <ensemble mode='npt'>
      <pressure units='megapascal'> 3 </pressure>
      ...
   </ensemble>
   ...
</simulation>
```

A full list of which units are defined for which dimensions can be found in the units.py module.

3.1.2 Initialization section

The input file can contain a "initialize" tag, which contains a number of fields that determine the starting values of the various quantities that define the state of the simulation – atomic positions, cell parameters, velocities, ... These fields (positions, velocities, cell, masses, labels, file) specify how the values should be obtained: either from a manually-input list or from an external file.

3.1.2.1 Configuration files

Instead of initializing the atom positions manually, the starting configuration can be specified through a separate data file. The name of the configuration file is specified within one of the possible fields of an "initialize" tag. The file format is specified with the "mode" attribute. The currently accepted file formats are:

- pdb
- xyz
- chk

the last of which will be described in the next section.

Depending on the field name, the values read from the external file will be used to initialize one component of the simulation or another (e.g. the positions or the velocities). The file tag can be used as a shortcut to initialize the atom positions, labels, masses and possibly the cell parameters at the same time. For instance,

```
<initialize nbeads="8">
   <file mode="pdb"> init.pdb </file>
</initialize>
```

is equivalent to

```
<initialize nbeads="8">
   <positions mode="pdb"> init.pdb </positions>
   <labels mode="pdb"> init.pdb </labels>
   <masses mode="pdb"> init.pdb </masses>
   <cell mode="pdb"> init.pdb </cell>
</initialize>
```

In practice, the using the file tag will only read the information that can be inferred from the given file type, so for an 'xyz' file, the cell parameters will not be initialized.

3.1.2.2 Initialization from checkpoint files

i-PI gives the option to output the entire state of the simulation at a particular timestep as an xml input file, called a checkpoint file (see 3.2.3 for details). As well as being a valid input for i-PI, a checkpoint can also be used inside an "initialize" tag to specify the configuration of the system, discarding other parameters of the simulation such as the current time step or the chosen ensemble. Input from a checkpoint is selected by using "chk" as the value of the "mode" attribute. As for the configuration file, a checkpoint file can be used to initialize either one or many variables depending on which tag name is used.

3.2 Output files

i-PI uses a very flexible mechanism to specify how and how often atomic configurations and physical properties should be output. Within the "output" tag of the xml input file the user can specify multiple tags, each one of which will correspond to a particular output file. Each file is managed separately by the code, so what is output to a particular file and how often can be adjusted for different files independently.

For example, some of the possible output properties require more than one force evaluation per time step to calculate, and so can considerably increase the computational cost of a simulation unless they are computed once every several time steps. On the other hand, for properties such as the conserved energy quantity it is easy, and often useful, to output them every time step as they are simple to compute and do not take long to output to file.

There are three types of output file that can be specified; property files for system level properties, trajectory files for atom/bead level properties, and checkpoint files which save the state of the system and so can be used to restart the simulation from a particular point. For a brief overview of the format of each of these types of files, and some of their more common uses, see 5.1. To give a more in depth explanation of each of these files, they will now be considered in turn.

3.2.1 Properties

This is the output file for all the system and simulation level properties, such as the total energy and the time elapsed. It is designed to track a small number of important properties throughout a simulation run, and as such has been formatted to be used as input for plotting programs such as gnuplot.

The file starts with a header, which describes the properties being written in the different columns and their output units. This is followed by the actual data. Each line corresponds to one instant of the simulation. The file is fixed formatted, with two blank characters at the start of each row, then the data in the same order as the header row. By default, each column is 16 characters wide and every float is written in exponential format with 8 digits after the decimal point.

For example, if we had asked for the current time step, the total simulation time in picoseconds, and the potential energy in electronvolt, then the properties output file would look something like:

```
# column   1     --> step : The current simulation time step.
# column   2     --> time{picosecond} : The elapsed simulation time.
# column   3     --> potential{electronvolt} : The physical system potential energy.
    0.00000000e+00     0.00000000e+00    -1.32860475e+04
```

```
    1.00000000e+00     1.00000000e-03    -1.32865789e+04
...
```

The properties that are output are determined by the "properties" tag in the xml input file. The format of this tag is:

```
<properties stride='' filename='' flush='' shape=''>
   [ prop1name{units}(arg1; ... ), prop2name{...}(...), ...  ]
</properties>
```

e.g.

```
<properties stride='100' filename='output'>
   [ step, atom_x{angstrom}(index=2;bead=0) ]
</properties>
```

The attributes have the following meanings:

stride The number of steps between each output to file

filename The name of the output file

flush The number of output lines between buffer flushes

shape The number of properties in the list.

The tag data is an array of strings, each of which contains three different parts:

- The property name, which describes which type of property is to be output. This is a mandatory part of the string.
- The units that the property will be output in. These are specified between curly brackets. If this is not specified, then the property will be output in atomic units. Note that some properties can only be output in atomic units.
- The arguments to be passed to the function. These are specified between standard brackets, with each argument separated by a semi-colon. These may or may not be mandatory depending on the property, as some arguments have well defined default values. The arguments can be specified by either of two different syntaxes, (name1=arg1; ...) or (arg1; ...).

  The first syntax uses keyword arguments. The above example would set the variable with the name "name1" the value "arg1". The second syntax uses positional arguments. This syntax relies on the arguments being specified in the correct order, as defined in the relevant function in the property.py module, since the user has not specified which variable to assign the value to.

  The two syntaxes may be mixed, but positional arguments must be specified first otherwise undefined behaviour will result. If no arguments are specified, then the defaults as defined in the properties.py module will be used.

The different available properties are:

atom_f: The force (x,y,z) acting on a particle given its index. Takes arguments index and bead (both zero based). If bead is not specified, refers to the centroid.

dimension: force; size: 3;

atom_p: The momentum (x,y,z) of a particle given its index. Takes arguments index and bead (both zero based). If bead is not specified, refers to the centroid.

dimension: momentum; size: 3;

atom_v: The velocity (x,y,z) of a particle given its index. Takes arguments index and bead (both zero based). If bead is not specified, refers to the centroid.

dimension: velocity; size: 3;

atom_x: The position (x,y,z) of a particle given its index. Takes arguments index and bead (both zero based). If bead is not specified, refers to the centroid.

dimension: length; size: 3;

cell_abcABC: The lengths of the cell vectors and the angles between them in degrees as a list of the form [a, b, c, A, B, C], where A is the angle between the sides of length b and c in degrees, and B and C are defined similarly. Since the output mixes different units, a, b and c can only be output in bohr.

size: 6;

cell_h: The simulation cell as a matrix. Returns the 6 non-zero components in the form [xx, yy, zz, xy, xz, yz].

dimension: length; size: 6;

conserved: The value of the conserved energy quantity per bead.

dimension: energy;

density: The mass density of the physical system.

dimension: density;

displacedpath: This is the estimator for the end-to-end distribution, that can be used to calculate the particle momentum distribution as described in in L. Lin, J. A. Morrone, R. Car and M. Parrinello, 105, 110602 (2010), Phys. Rev. Lett. Takes arguments 'ux', 'uy' and 'uz', which are the components of the path opening vector. Also takes an argument 'atom', which can be either an atom label or index (zero based) to specify which species to find the end-to-end distribution estimator for. If not specified, all atoms are used. Note that one atom is computed at a time, and that each path opening operation costs as much as a PIMD step. Returns the average over the selected atoms of the estimator of exp(-U(u)) for each frame.

isotope_scfep: Returns the (many) terms needed to compute the scaled-coordinates free energy perturbation scaled mass KE estimator (M. Ceriotti, T. Markland, J. Chem. Phys. 138, 014112 (2013)). Takes two arguments, 'alpha' and 'atom', which give the scaled mass parameter and the atom of interest respectively, and default to '1.0' and ''. The 'atom' argument can either be the label of a particular kind of atom, or an index (zero based) of a specific atom. This property computes, for each atom in the selection, an estimator for the kinetic energy it would have had if it had the mass scaled by alpha. The 7 numbers output are the average over the selected atoms of the log of the weights <h>, the average of the squares <h**2>, the average of the un-weighted scaled-coordinates kinetic energies <T_CV> and of the squares <T_CV**2>, the log sum of the weights LW=ln(sum(e**(-h))), the sum of the re-weighted kinetic energies, stored as a log modulus and sign, LTW=ln(abs(sum(T_CV e**(-h)))) STW=sign(sum(T_CV e**(-h))). In practice, the best estimate of the estimator can be computed as [sum_i exp(LTW_i)*STW_i]/[sum_i exp(LW_i)]. The other terms can be used to compute diagnostics for the statistical accuracy of the re-weighting process. Note that evaluating this estimator costs as much as a PIMD step for each atom in the list. The elements that are output have different units, so the output can be only in atomic units.

size: 7;

isotope_tdfep: Returns the (many) terms needed to compute the thermodynamic free energy perturbation scaled mass KE estimator (M. Ceriotti, T. Markland, J. Chem. Phys. 138, 014112 (2013)). Takes two arguments, 'alpha' and 'atom', which give the scaled mass parameter and the atom of interest respectively, and default to '1.0' and ''. The 'atom' argument can either be the label of a particular kind of atom, or an index (zero based) of a specific atom. This property computes, for each atom in the selection, an estimator for the kinetic energy it would have had if it had the mass scaled by alpha. The 7 numbers output are the average over the selected atoms of the log of the weights <h>, the average of the squares <h**2>, the average of the un-weighted scaled-coordinates kinetic energies <T_CV> and of the squares <T_CV**2>, the log sum of the weights LW=ln(sum(e**(-h))), the sum of the re-weighted kinetic energies, stored as a log modulus and sign, LTW=ln(abs(sum(T_CV e**(-h)))) STW=sign(sum(T_CV e**(-h))). In practice, the best estimate of the estimator can be computed as [sum_i exp(LTW_i)*STW_i]/[sum_i exp(LW_i)]. The other terms can be used to compute diagnostics for the statistical accuracy of the re-weighting process. Evaluating this estimator is inexpensive, but typically the statistical accuracy is worse than with the scaled coordinates estimator. The elements that are output have different units, so the output can be only in atomic units.

size: 7;

kinetic_cv: The centroid-virial quantum kinetic energy of the physical system. Takes an argument 'atom', which can be either an atom label or index (zero based) to specify which species to find the kinetic energy of. If not specified, all atoms are used.

dimension: energy;

kinetic_ij: The centroid-virial off-diagonal quantum kinetic energy tensor of the physical system. This computes the cross terms between atoms i and atom j, whose average is <p_i*p_j/(2*sqrt(m_i*m_j))>. Returns the 6 independent components in the form [xx, yy, zz, xy, xz, yz]. Takes arguments 'i' and 'j', which give the indices of the two desired atoms.

dimension: energy; size: 6;

kinetic_md: The kinetic energy of the (extended) classical system. Takes an argument 'atom', which can be either an atom label or index (zero based) to specify which species to find the kinetic energy of. If not specified, all atoms are used.

dimension: energy;

kinetic_tens: The centroid-virial quantum kinetic energy tensor of the physical system. Returns the 6 independent components in the form [xx, yy, zz, xy, xz, yz]. Takes an argument 'atom', which can be either an atom label or index (zero based) to specify which species to find the kinetic tensor components of. If not specified, all atoms are used.

dimension: energy; size: 6;

kstress_cv: The quantum estimator for the kinetic stress tensor of the physical system. Returns the 6 independent components in the form [xx, yy, zz, xy, xz, yz].

dimension: pressure; size: 6;

kstress_md: The kinetic stress tensor of the (extended) classical system. Returns the 6 independent components in the form [xx, yy, zz, xy, xz, yz].

dimension: pressure; size: 6;

potential: The physical system potential energy.

dimension: energy;

pressure_cv: The quantum estimator for pressure of the physical system.

dimension: pressure;

pressure_md: The pressure of the (extended) classical system.

dimension: pressure;

r_gyration: The average radius of gyration of the selected ring polymers. Takes an argument 'atom', which can be either an atom label or index (zero based) to specify which species to find the radius of gyration of. If not specified, all atoms are used and averaged.

dimension: length;

scaledcoords: Returns the estimators that are required to evaluate the scaled-coordinates estimators for total energy and heat capacity, as described in T. M. Yamamoto, J. Chem. Phys., 104101, 123 (2005). Returns eps_v and eps_v', as defined in that paper. As the two estimators have a different dimensions, this can only be output in atomic units. Takes one argument, 'fd_delta', which gives the value of the finite difference parameter used - which defaults to -1e-05. If the value of 'fd_delta' is negative, then its magnitude will be reduced automatically by the code if the finite difference error becomes too large.

size: 2;

spring: The total spring potential energy between the beads of all the ring polymers in the system.

dimension: energy;

step: The current simulation time step.

dimension: number;

stress_cv: The total quantum estimator for the stress tensor of the physical system. Returns the 6 independent components in the form [xx, yy, zz, xy, xz, yz].

dimension: pressure; size: 6;

stress_md: The total stress tensor of the (extended) classical system. Returns the 6 independent components in the form [xx, yy, zz, xy, xz, yz].

dimension: pressure; size: 6;

temperature: The current temperature, as obtained from the MD kinetic energy of the (extended) ring polymer. Takes a single, optional argument 'atom', which can be either an atom label or an index (zero-based) to specify which species or individual atom to output the temperature of. If not specified, all atoms are used and averaged.

dimension: temperature;

time: The elapsed simulation time.

dimension: time;

virial_cv: The quantum estimator for the virial stress tensor of the physical system. Returns the 6 independent components in the form [xx, yy, zz, xy, xz, yz].

dimension: pressure; size: 6;

virial_md: The virial tensor of the (extended) classical system. Returns the 6 independent components in the form [xx, yy, zz, xy, xz, yz].

dimension: pressure; size: 6;

volume: The volume of the cell box.

dimension: volume;

3.2.2 Trajectory files

These are the output files for atomic or bead level properties, such as the bead positions. In contrast to properties files, they output data for all atomic degrees of freedom, in a format that can be read by visualization packages such as VMD.

Multiple trajectory files can be specified, each described by a separate "trajectory" tag within the "output" section of the input file. The allowable file formats for the trajectory output files are the same as for the configuration input files, given in 3.1.2.1.

These tags have the format:

```
<trajectory stride='' filename='' format='' cell_units='' flush='' bead=''>
   traj_name{units}(arg1;...)
</trajectory>
```

This is very similar to the "properties" tag, except that it has the additional tags "format" and "cell_units", and only one *traj_name* quantity can be specified per file. 'format' specifies the format of the output file, and 'cell_units' specifies the units in which the cell dimensions are output. Depending on the quantity being output, the trajectory may consist of just one file per time step (e.g. the position of the centroid) or of several files, one for each bead, whose name will be automatically determined by appending the bead index to the specified "filename" attribute (e.g. the beads position). In the latter case it is also possible to output the quantity computed for a single bead by specifying its (zero-based) index in the "bead" attribute.

The quantities that can be output in trajectory files are:

extras: The additional data returned by the client code, printed verbatim. Will print out one file per bead, unless the bead attribute is set by the user.

f_centroid: The force acting on the centroid.

dimension: force;

forces: The force trajectories. Will print out one file per bead, unless the bead attribute is set by the user.

dimension: force;

kinetic_cv: The centroid virial quantum kinetic energy estimator for each atom, resolved into Cartesian components [xx, yy, zz]

dimension: energy;

kinetic_od: The off diagonal elements of the centroid virial quantum kinetic energy tensor [xy, xz, yz]

dimension: energy;

momenta: The momentum trajectories. Will print out one file per bead, unless the bead attribute is set by the user.

dimension: momentum;

p_centroid: The centroid momentum.

dimension: momentum;

positions: The atomic coordinate trajectories. Will print out one file per bead, unless the bead attribute is set by the user.

dimension: length;

r_gyration: The radius of gyration of the ring polymer, for each atom and resolved into Cartesian components [xx, yy, zz]

dimension: length;

v_centroid: The centroid velocity.

dimension: velocity;

velocities: The velocity trajectories. Will print out one file per bead, unless the bead attribute is set by the user.

dimension: velocity;

x_centroid: The centroid coordinates.

dimension: length;

3.2.3 Checkpoint files

As well as the above output files, the state of the system at a particular time step can also be saved to file. These checkpoint files can later be used as input files, with all the information required to restore the state of the system to the point at which the file was created.

This is specified by the "checkpoint" tag which has the syntax:

```
<checkpoint stride='' filename='' overwrite=''>
   step
</checkpoint>
```

Again, this is similar to the "trajectory" and "properties" tags, but instead of having a value which specifies what to output, the value simply gives a number to identify the current checkpoint file. There is also one additional attribute, "overwrite", which specifies whether each new checkpoint file overwrites the old one, or whether all checkpoint files are kept. If they are kept, they will be written not to the file "filename", but instead an index based on the value of "step" will be appended to it to distinguish between different files.

If the 'step' parameter is not specified, the following syntax can also be used:

```
<checkpoint stride='' filename='' overwrite=''/>
```

3.2.3.1 Soft exit and RESTART

As well as outputting checkpoint files during a simulation run, i-PI also creates a checkpoint automatically at the end of the simulation, with file name "RESTART". In the same way as the checkpoint files discussed above, it contains the full state of the simulation. It can be used to seamlessly restart the simulation if the user decides that a longer run is needed to gather sufficient statistics, or if i-PI is terminated before the desired number of steps have been completed.

i-PI will try to generate a RESTART file when it terminates, either because *total_time* has elapsed, or because it received a (soft) kill signal by the operating system. A soft exit can also be forced by creating an empty file named "EXIT" in the directory in which i-PI is running.

An important point to note is that since each time step is split into several parts, it is only at the end of each step that all the variables are consistent with each other in such a way that the simulation can be restarted from them without changing the dynamics. Thus if a soft exit call is made during a step, then the restart file that is created must correspond to the state of the system *before* that step began. To this end, the state of the system is saved at the start of every step.

3.3 Distributed execution

3.3.1 Communication protocol

i-PI is based on a clear-cut separation between the evolution of the nuclear coordinates and the evaluation of energy and forces, which is delegated to an external program. The two parts are kept as independent as possible, to minimize the client-side implementation burden, and to make sure that the server will be compatible with any empirical or *ab initio* code that can compute inter-atomic forces for a given configuration.

Once a communication channel has been established between the client and the server (see 3.3.3), the two parties exchange minimal information: i-PI sends the atomic positions and the cell parameters to the client, which computes energy, forces and virial and returns them to the server.

The exchange of information is regulated by a simple communication protocol. The server polls the status of the client, and when the client signals that is ready to compute forces i-PI sends the atomic positions to it. When the client responds to the status query by signalling that the force evaluation is finished, i-PI will prepare to receive the results of the calculation. If at any stage the client does not respond to a query, the server will wait and try again until a prescribed timeout period has elapsed, then consider the client to be stuck, disconnect from it and reassign its force evaluation task to another active instance. The server assumes that 4-byte integers, 8-byte floats and 1-byte characters are used. The typical communication flow is as follows:

1. a header string "**STATUS**" is sent by the server to the client that has connected to it;

2. a header string is then returned, giving the status of the client code. Recognized messages are:

   "NEEDINIT": if the client code needs any initialising data, it can be sent here. The server code will then send a header string "INIT", followed by an integer corresponding to the bead index, another integer giving the number of bits in the initialization string, and finally the initialization string itself.

   "READY": sent if the client code is ready to calculate the forces. The server socket will then send a string "POSDATA", then nine floats for the cell vector matrix, then another nine floats for the inverse matrix. The server socket will then send one integer giving the number of atoms, then the position data as 3 floats for each atom giving the 3 cartesian components of its position.

   "HAVEDATA": is sent if the client has finished computing the potential and forces. The server socket then sends a string "GETFORCE", and the client socket returns "FORCEREADY". The potential is then returned as a float, the number of atoms as an integer, then the force data as 3 floats per atom in the same way as the positions, and the virial as 9 floats in the same way as the cell vector matrix. Finally, the client may return an arbitrary string containing additional data that have been obtained by the electronic structure calculation (atomic charges, dipole moment, ...). The client first returns an integer specifying the number of characters, and then the string, which will be output verbatim if this "extra" information is requested in the output section (see 3.2.2).

3. The server socket waits until the force data for each replica of the system has been calculated and returned, then the MD can be propagated for one more time step, and new force requests will be dispatched.

3.3.2 Parallelization

As mentioned before, one of the primary advantages of using this type of data transfer is that it allows multiple clients to connect to an i-PI server, so that different replicas of the system can be assigned to different client codes and their forces computed in parallel. In the case of *ab initio* force evaluation, this is a trivial level of parallelism, since the cost of the force calculation is overwhelming relative to the overhead involved in exchanging coordinates and forces. Note that even if the parallelization over the replicas is trivial, often one does not obtain perfect scaling, due to the fact that some of the atomic configurations might require more steps to reach self-consistency, and the wall-clock time per step is determined by the slowest replica.

i-PI maintains a list of active clients, and distributes the forces evaluations among those available. This means that, if desired, one can run an n-bead calculation using only $m < n$ clients, as the server takes care of sending multiple replicas to each client per MD step. To avoid having clients idling for a substantial amount of time, m should be a divisor of n. The main advantage of this approach, compared to one that rigidly assigns one instance of the client to each bead, is that if each client is run as an independent job in a queue (see 2.3.3), i-PI can start performing PIMD as soon as a single job has started, and can carry on advancing the simulation even if one of the clients becomes unresponsive.

Especially for *ab initio* calculations, there is an advantage in running with $m = n$. i-PI will always try to send the coordinates for one path integral replica to the client that computed it at the previous step: this reduces the change in the particle positions between force evaluations, so that the charge density/wavefunction from the previous step is a better starting guess and self-consistency can be achieved faster. Also, receiving coordinates that represent a continuous trajectory makes it possible to use extrapolation strategies that might be available in the client code.

Obviously, most electronic-structure client codes provide a further level of parallelisation, based on OpenMP and/or MPI. This is fully compatible with i-PI, as it does not matter how the client does the calculation since only the forces, potential and virial are sent to the server, and the communication is typically performed by the master process of the client.

3.3.3 Sockets

The communication between the i-PI server and the client code that evaluates forces is implemented through sockets. A socket is a data transfer device that is designed for internet communication, so it supports both multiple client connections to the same server and two-way communication. This makes sockets ideal for use in i-PI, where each calculation may require multiple instances of the client code. A socket interface can actually function in two different modes.

UNIX-domain sockets are a mechanism for local, inter-process communication. They are fast, and best suited when one wants to run i-PI with empirical potentials, and the latency of the communication with the client becomes a significant overhead for the calculation. UNIX-domain sockets create a special file in the local file system, that serves as a rendezvous point between server and clients, and are uniquely identified by

the name of the file itself, that can be specified in the "address" tag of "socket" in the xml input file and in the input of the client.

Unfortunately, UNIX sockets do not allow one to run i-PI and the clients on different computers, which limits greatly their utility when one needs to run massively parallel calculations. In these cases – typically when performing *ab initio* simulations – the force calculation becomes the bottleneck, so there is no need for fast communication with the server, and one can use internet sockets, that instead are specifically designed for communication over a network.

Internet sockets are described by an address and a port number. The address of the host is given as the IP address, or as a hostname that is resolved to an IP address by a domain name server, and is specified by the "address" variable of a socket object. The port number is an integer between 1 and 65535 used to distinguish between all the different sockets open on a particular host. As many of the lower numbers are protected for use in important system processes or internet communication, it is generally advisable to only use numbers in the range 1025-65535 for simulations.

The socket object has two more parameters. The option "latency" specifies how often i-PI polls the list of active clients to dispatch positions and collect results: setting it to a small value makes the program more responsive, which is appropriate when the evaluation of the forces is very fast. In *ab initio* simulations, it is best to set it to a larger value (of the order of 0.01 seconds), as higher latency will have no noticeable impact on performance, but will reduce the cost of having i-PI run in the background to basically zero.

Normally, i-PI can detect when one of the clients dies or disconnects, and can remove it from the active list and dispatch its force calculation to another instance. If however one of the client hangs without closing the communication channel, i-PI has no way of determining that something is going wrong, and will just wait forever. One can specify a parameter "timeout", that corresponds to the maximum time – in seconds – that i-PI should wait before deciding that one of the clients has become unresponsive and should be discarded.

3.3.4 Running i-PI over the network

3.3.4.1 Understanding the network layout

Running i-PI in any non-local configuration requires a basic understanding of the layout of the network one is dealing with. Each workstation, or node of a HPC system, may expose more than one network interface, some of which can be connected to the outside internet, and some of which may be only part of a local network. A list of the network interfaces available on a given host can be obtained for instance with the command

```
> /sbin/ip addr
```

which will return a list of interfaces of the form

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast qlen 1000
    link/ether 00:25:b3:e7:a0:44 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.254/16 brd 192.168.255.255 scope global eth0
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast qlen 1000
    link/ether 00:25:b3:e7:a0:46 brd ff:ff:ff:ff:ff:ff
    inet 129.67.106.153/22 brd 129.67.107.255 scope global eth1
```

Each item corresponds to a network interface, identified by a number and a name (lo, eth0, eth1, ...). Most of the interfaces will have an associated IP address – the four numbers separated by dots that are listed after "inet", e.g. 192.168.1.254 for the eth0 interface in the example above.

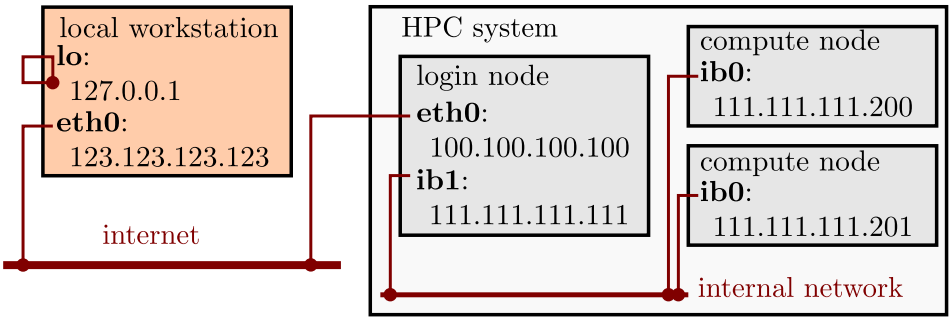

Figure 3.1: A schematic representation of the network layout one typically finds when running i-PI and the clients on a HPC system and/or on a local workstation.

Figure 3.1 represents schematically a typical network layout for a HPC system and a local workstation. When running i-PI locally on a workstation, one can use the loopback interface (that can be referred to as "localhost" in the "address" field of both i-PI and the client) for communication. When running both i-PI and the clients on a HPC cluster, one should work out which of the the interfaces that are available on the node where the i-PI server runs are accessible from the compute nodes. This requires some trial and error, and possibly setting the "address" field dynamically from the job that launches i-PI. For instance, if one was running i-PI on the login node, and the clients on different compute nodes, as in Figure 2.1b, then on the HPC system described in Figure 3.1 one should set the address to that of the *ib1* interface – 111.111.111.111 in the example above. If instead i-PI was launched in a job script, then the submission script would have to check for the IP address associated with the *ib0* interface on the node the job has been dispatched to, and set that address (e.g. 111.111.111.200) in the inputs of both i-PI and the clients that will be launched in the same (or separate) jobs.

Running i-PI on a separate workstation (Figure 2.1c) gives maximum flexibility, but is also trickier as one has to reach the internet from the compute nodes, that are typically not directly connected to it. We discuss this more advanced setup in the next paragraph.

3.3.4.2 ssh tunnelling

If i-PI is to be run in a distributed computing mode, then one should make sure that the workstation on which the server will run is accessible from the outside internet on the range of ports that one wants to use for i-PI. There are ways to circumvent a firewall, but we will not discuss them here, as the whole point of i-PI is that it can be run on a low-profile PC whose security does not need to be critical. Typically arrangements can be made to open up a range of ports for incoming connections.

A more substantial problem – as it depends on the physical layout of the network rather than on software settings of the firewall – is how to access the workstation from the compute nodes, which in most cases do not have a network interface directly connected to the outside internet.

The problem can be solved by creating a ssh tunnel, i.e. an instance of the ssh

secure shell that will connect the compute node to the login node, and then forward all traffic that is directed to a designated port on the compute node to the remote location that is running i-PI, passing through the outbound network interface of the login node.

In the example above, if i-PI is running on a local workstation, one should run:

```
> ssh -f -N -L local_port:workstation:remote_port -2 login_node
```

from the job script that launches the client. For instance, with the network layout of Figure 3.1, and if the i-PI server is listening on port 12345 of the *eth0* interface, the tunnel should be created as:

```
> ssh -f -N -L 54321:123.123.123.123:12345 -2 111.111.111.111
```

The client should then be configured to connect to *localhost* on port 54321. The connection with i-PI will be established through the tunnel, and the data exchange can begin.

Note that, in order to be able to include the above commands in a script, the login node and the compute nodes should be configured to allow password-less login within the HPC system. This can be achieved easily, and does not entail significant security risks, since it only allows one to connect from one node to another within the local network. To do so, you should log onto the HPC system, and create a pair of ssh keys (if this has not been done already, in which case an id_rsa.pub file should be present in the user's ~/.ssh/ directory) by issuing the command

```
> ssh-keygen -t rsa
```

The program will then prompt for a passphrase twice. Since we wish to have use this in a job script where we will not be able to enter a password, just hit enter twice.

This should now have created two files in the directory ~/.ssh, id_rsa and id_rsa.pub. These should be readable only by you, so use the following code to set up the correct file permissions:

```
> chmod 600 ~/.ssh/id_rsa ~/.ssh/id_rsa.pub
```

Finally, copy the contents of the file id_rsa.pub and append them to the file authorized_keys in the directory ~/.ssh of the user on the login node, which is typically shared among all the nodes of a cluster and therefore allows password-less login from all of the compute nodes.

Input reference

This chapter gives a complete list of the tags that can be specified in the xml input file, along with the hierarchy of objects. Note that every xml input file must start with the root tag "simulation". See the accompanying "help.xml" file in the "doc/help_files" directory to see the recommended input file structure.

Each section of this chapter describes one of the major input classes used in the i-PI initialization. These sections will start with some normal text which describes the function of this class in detail. After this the attributes of the tag will be listed, enclosed within a frame for clarity. Finally, the fields contained within the tag will be listed in alphabetical order, possibly followed by their attributes. **Attribute** names will be bold and **field** names both bold and underlined. *Other supplementary information* will be in small font and italicized.

4.1 SIMULATION

This is the top level class that deals with the running of the simulation, including holding the simulation specific properties such as the time step and outputting the data.

verbosity: The level of output on stdout.
default: 'low'; data type: string; options: 'quiet', 'low', 'medium', 'high', 'debug';

total_time: The maximum wall clock time (in seconds).
default: 0 ; data type: float;

step: The current simulation time step.
default: 0 ; data type: integer;

initialize: Specifies the number of beads, and how the system should be initialized.

nbeads: The number of beads. Will override any provision from inside the initializer. A ring polymer contraction scheme is used to scale down the number of beads if required. If instead the number of beads is scaled up, higher normal modes will be initialized to zero.
data type: integer;

beads: Describes the bead configurations in a path integral simulation.

> **natoms**: The number of atoms.
>
> *default: 0 ; data type: integer;*
>
> **nbeads**: The number of beads.
>
> *default: 0 ; data type: integer;*

total_steps: The total number of steps that will be done. If 'step' is equal to or greater than 'total_steps', then the simulation will finish.

default: 1000 ; data type: integer;

normal_modes: Deals with the normal mode transformations, including the adjustment of bead masses to give the desired ring polymer normal mode frequencies if appropriate. Takes as arguments frequencies, of which different numbers must be specified and which are used to scale the normal mode frequencies in different ways depending on which 'mode' is specified.

dimension: frequency; default: [ ] ; data type: float;

> **units**: The units the input data is given in.
>
> *default: ''; data type: string;*
>
> **shape**: The shape of the array.
>
> *default: (0,) ; data type: tuple;*
>
> **mode**: Specifies the technique to be used to calculate the dynamical masses. 'rpmd' simply assigns the bead masses the physical mass. 'manual' sets all the normal mode frequencies except the centroid normal mode manually. 'pa-cmd' takes an argument giving the frequency to set all the non-centroid normal modes to. 'wmax-cmd' is similar to 'pa-cmd', except instead of taking one argument it takes two ([wmax,wtarget]). The lowest-lying normal mode will be set to wtarget for a free particle, and all the normal modes will coincide at frequency wmax.
>
> *default: 'rpmd'; data type: string; options: 'pa-cmd', 'wmax-cmd', 'manual', 'rpmd';*
>
> **transform**: Specifies whether to calculate the normal mode transform using a fast Fourier transform or a matrix multiplication. For small numbers of beads the matrix multiplication may be faster.
>
> *default: 'fft'; data type: string; options: 'fft', 'matrix';*

cell: Deals with the cell parameters. Takes as array which can be used to initialize the cell vector matrix.

dimension: length; default: [0. 0. 0. 0. 0. 0. 0. 0. 0.]; data type: float;

> **units**: The units the input data is given in.
>
> *default: ''; data type: string;*
>
> **shape**: The shape of the array.
>
> *default: (3, 3) ; data type: tuple;*

forces: Deals with creating all the necessary forcefield objects.

output: This class defines how properties, trajectories and checkpoints should be output during the simulation. May contain zero, one or many instances of properties, trajectory or checkpoint tags, each giving instructions on how one output file should be created and managed.

> **prefix**: A string that will be prepended to each output file name. The file name is given by 'prefix'.'filename' + format_specifier. The format specifier may also include a number if multiple similar files are output.
>
> *default: ''; data type: string;*

prng: Deals with the pseudo-random number generator.

ensemble: Holds all the information that is ensemble specific, such as the temperature and the external pressure, and the thermostats and barostats that control it.

> **mode**: The ensemble that will be sampled during the simulation. 'replay' means that a simulation is restarted from a previous simulation.
>
> *data type: string; options: 'nve', 'nvt', 'npt', 'replay';*

4.2 INITIALIZER

Specifies the number of beads, and how the system should be initialized.

> **nbeads**: The number of beads. Will override any provision from inside the initializer. A ring polymer contraction scheme is used to scale down the number of beads if required. If instead the number of beads is scaled up, higher normal modes will be initialized to zero.
>
> *data type: integer;*

cell: Initializes the configuration of the cell. Will take a 'units' attribute of dimension 'length'

> *data type: string;*
>
> **mode**: This decides whether the system box is created from a cell parameter matrix, or from the side lengths and angles between them. If 'mode' is 'manual', then 'cell' takes a 9-elements vector containing the cell matrix (row-major). If 'mode' is 'abcABC', then 'cell' takes an array of 6 floats, the first three being the length of the sides of the system parallelopiped, and the last three being the angles (in degrees) between those sides. Angle A corresponds to the angle between sides b and c, and so on for B and C. If mode is 'abc', then this is the same as for 'abcABC', but the cell is assumed to be orthorhombic. 'pdb' and 'chk' read the cell from a PDB or a checkpoint file, respectively.
>
> *default: 'manual'; data type: string; options: 'manual', 'pdb', 'chk', 'abc', 'abcABC';*

labels: Initializes atomic labels

data type: string;

index: The index of the atom for which the value will be set. If a negative value is specified, then all atoms are assumed.

default: -1 ; data type: integer;

bead: The index of the bead for which the value will be set. If a negative value is specified, then all beads are assumed.

default: -1 ; data type: integer;

mode: The input data format. 'xyz' and 'pdb' stand for xyz and pdb input files respectively. 'chk' stands for initialization from a checkpoint file. 'manual' means that the value to initialize from is giving explicitly as a vector.

default: 'chk'; data type: string; options: 'manual', 'xyz', 'pdb', 'chk';

file: Initializes everything possible for the given mode. Will take a 'units' attribute of dimension 'length'. The unit conversion will only be applied to the positions and cell parameters.

data type: string;

mode: The input data format. 'xyz' and 'pdb' stand for xyz and pdb input files respectively. 'chk' stands for initialization from a checkpoint file.

default: 'chk'; data type: string; options: 'xyz', 'pdb', 'chk';

positions: Initializes atomic positions. Will take a 'units' attribute of dimension 'length'

data type: string;

index: The index of the atom for which the value will be set. If a negative value is specified, then all atoms are assumed.

default: -1 ; data type: integer;

bead: The index of the bead for which the value will be set. If a negative value is specified, then all beads are assumed.

default: -1 ; data type: integer;

mode: The input data format. 'xyz' and 'pdb' stand for xyz and pdb input files respectively. 'chk' stands for initialization from a checkpoint file. 'manual' means that the value to initialize from is giving explicitly as a vector.

default: 'chk'; data type: string; options: 'manual', 'xyz', 'pdb', 'chk';

momenta: Initializes atomic momenta. Will take a 'units' attribute of dimension 'momentum'

data type: string;

index: The index of the atom for which the value will be set. If a negative value is specified, then all atoms are assumed.

default: -1 ; data type: integer;

bead: The index of the bead for which the value will be set. If a negative value is specified, then all beads are assumed.

default: -1 ; data type: integer;

mode: The input data format. 'xyz' and 'pdb' stand for xyz and pdb input files respectively. 'chk' stands for initialization from a checkpoint file. 'manual' means that the value to initialize from is giving explicitly as a vector. 'thermal' means that the data is to be generated from a Maxwell-Boltzmann distribution at the given temperature.

default: 'chk'; data type: string; options: 'manual', 'xyz', 'pdb', 'chk', 'thermal';

velocities: Initializes atomic velocities. Will take a 'units' attribute of dimension 'velocity'

data type: string;

index: The index of the atom for which the value will be set. If a negative value is specified, then all atoms are assumed.

default: -1 ; data type: integer;

bead: The index of the bead for which the value will be set. If a negative value is specified, then all beads are assumed.

default: -1 ; data type: integer;

mode: The input data format. 'xyz' and 'pdb' stand for xyz and pdb input files respectively. 'chk' stands for initialization from a checkpoint file. 'manual' means that the value to initialize from is giving explicitly as a vector. 'thermal' means that the data is to be generated from a Maxwell-Boltzmann distribution at the given temperature.

default: 'chk'; data type: string; options: 'manual', 'xyz', 'pdb', 'chk', 'thermal';

masses: Initializes atomic masses. Will take a 'units' attribute of dimension 'mass'

data type: string;

index: The index of the atom for which the value will be set. If a negative value is specified, then all atoms are assumed.

default: -1 ; data type: integer;

bead: The index of the bead for which the value will be set. If a negative value is specified, then all beads are assumed.

default: -1 ; data type: integer;

mode: The input data format. 'xyz' and 'pdb' stand for xyz and pdb input files respectively. 'chk' stands for initialization from a checkpoint file. 'manual' means that the value to initialize from is giving explicitly as a vector.

default: 'chk'; data type: string; options: 'manual', 'xyz', 'pdb', 'chk';

gle: Initializes the additional momenta in a GLE thermostat.

data type: string;

mode: 'chk' stands for initialization from a checkpoint file. 'manual' means that the value to initialize from is giving explicitly as a vector.

default: 'manual'; data type: string; options: 'chk', 'manual';

4.3 INITFILE

This is the class to initialize from file.

data type: string;

mode: The input data format. 'xyz' and 'pdb' stand for xyz and pdb input files respectively. 'chk' stands for initialization from a checkpoint file.

default: 'chk'; data type: string; options: 'xyz', 'pdb', 'chk';

4.4 INITPOSITIONS

This is the class to initialize positions.

data type: string;

index: The index of the atom for which the value will be set. If a negative value is specified, then all atoms are assumed.

default: -1 ; data type: integer;

bead: The index of the bead for which the value will be set. If a negative value is specified, then all beads are assumed.

default: -1 ; data type: integer;

mode: The input data format. 'xyz' and 'pdb' stand for xyz and pdb input files respectively. 'chk' stands for initialization from a checkpoint file. 'manual' means that the value to initialize from is giving explicitly as a vector.

default: 'chk'; data type: string; options: 'manual', 'xyz', 'pdb', 'chk';

4.5 INITMOMENTA

This is the class to initialize momenta.

data type: string;

index: The index of the atom for which the value will be set. If a negative value is specified, then all atoms are assumed.

default: -1 ; data type: integer;

bead: The index of the bead for which the value will be set. If a negative value is specified, then all beads are assumed.

default: -1 ; data type: integer;

mode: The input data format. 'xyz' and 'pdb' stand for xyz and pdb input files respectively. 'chk' stands for initialization from a checkpoint file. 'manual' means that the value to initialize from is giving explicitly as a vector. 'thermal' means that the data is to be generated from a Maxwell-Boltzmann distribution at the given temperature.

default: 'chk'; data type: string; options: 'manual', 'xyz', 'pdb', 'chk', 'thermal';

4.6 INITVELOCITIES

This is the class to initialize velocities.

data type: string;

index: The index of the atom for which the value will be set. If a negative value is specified, then all atoms are assumed.

default: -1 ; data type: integer;

bead: The index of the bead for which the value will be set. If a negative value is specified, then all beads are assumed.

default: -1 ; data type: integer;

mode: The input data format. 'xyz' and 'pdb' stand for xyz and pdb input files respectively. 'chk' stands for initialization from a checkpoint file. 'manual' means that the value to initialize from is giving explicitly as a vector. 'thermal' means that the data is to be generated from a Maxwell-Boltzmann distribution at the given temperature.

default: 'chk'; data type: string; options: 'manual', 'xyz', 'pdb', 'chk', 'thermal';

4.7 INITLABELS

This is the class to initialize atomic labels.

data type: string;

index: The index of the atom for which the value will be set. If a negative value is specified, then all atoms are assumed.

default: -1 ; data type: integer;

bead: The index of the bead for which the value will be set. If a negative value is specified, then all beads are assumed.

default: -1 ; data type: integer;

mode: The input data format. 'xyz' and 'pdb' stand for xyz and pdb input files respectively. 'chk' stands for initialization from a checkpoint file. 'manual' means that the value to initialize from is giving explicitly as a vector.

default: 'chk'; data type: string; options: 'manual', 'xyz', 'pdb', 'chk';

4.8 INITMASSES

This is the class to initialize atomic masses.

data type: string;

index: The index of the atom for which the value will be set. If a negative value is specified, then all atoms are assumed.

default: -1 ; data type: integer;

bead: The index of the bead for which the value will be set. If a negative value is specified, then all beads are assumed.

default: -1 ; data type: integer;

mode: The input data format. 'xyz' and 'pdb' stand for xyz and pdb input files respectively. 'chk' stands for initialization from a checkpoint file. 'manual' means that the value to initialize from is giving explicitly as a vector.

default: 'chk'; data type: string; options: 'manual', 'xyz', 'pdb', 'chk';

4.9 INITCELL

This is the class to initialize cell.

data type: string;

mode: This decides whether the system box is created from a cell parameter matrix, or from the side lengths and angles between them. If 'mode' is 'manual', then 'cell' takes a 9-elements vector containing the cell matrix (row-major). If 'mode' is 'abcABC', then 'cell' takes an array of 6 floats, the first three being the length of the sides of the system parallelopiped, and the last three being the angles (in degrees) between those sides. Angle A corresponds to the angle between sides b and c, and so on for B and C. If mode is 'abc', then this is the same as for 'abcABC', but the cell is assumed to be orthorhombic. 'pdb' and 'chk' read the cell from a PDB or a checkpoint file, respectively.

default: 'manual'; data type: string; options: 'manual', 'pdb', 'chk', 'abc', 'abcABC';

4.10 INITTHERMO

This is the class to initialize the thermostat (ethermo and fictitious momenta).

data type: string;

mode: 'chk' stands for initialization from a checkpoint file. 'manual' means that the value to initialize from is giving explicitly as a vector.

default: 'manual'; data type: string; options: 'chk', 'manual';

4.11 ENSEMBLE

Holds all the information that is ensemble specific, such as the temperature and the external pressure, and the thermostats and barostats that control it.

mode: The ensemble that will be sampled during the simulation. 'replay' means that a simulation is restarted from a previous simulation.

data type: string; options: 'nve', 'nvt', 'npt', 'replay';

barostat: Simulates an external pressure bath.

mode: The type of barostat. Currently, only a 'isotropic' barostat is implemented, that combines ideas from the Bussi-Zykova-Parrinello barostat for classical MD with ideas from the Martyna-Hughes-Tuckerman centroid barostat for PIMD; see Ceriotti, More, Manolopoulos, Comp. Phys. Comm. 2013 for implementation details.

default: 'dummy'; data type: string; options: 'dummy', 'isotropic';

thermostat: The thermostat for the atoms, keeps the atom velocity distribution at the correct temperature.

> **mode**: The style of thermostatting. 'langevin' specifies a white noise langevin equation to be attached to the cartesian representation of the momenta. 'svr' attaches a velocity rescaling thermostat to the cartesian representation of the momenta. Both 'pile_l' and 'pile_g' attaches a white noise langevin thermostat to the normal mode representation, with 'pile_l' attaching a local langevin thermostat to the centroid mode and 'pile_g' instead attaching a global velocity rescaling thermostat. 'gle' attaches a coloured noise langevin thermostat to the cartesian representation of the momenta, 'nm_gle' attaches a coloured noise langevin thermostat to the normal mode representation of the momenta and a langevin thermostat to the centroid and 'nm_gle_g' attaches a gle thermostat to the normal modes and a svr thermostat to the centroid.
>
> *data type: string; options: '', 'langevin', 'svr', 'pile_l', 'pile_g', 'gle', 'nm_gle', 'nm_gle_g';*

timestep: The time step.

dimension: time; default: 1.0 ; data type: float;

> **units**: The units the input data is given in.
>
> *default: ''; data type: string;*

pressure: The external pressure.

dimension: pressure; default: 1.0 ; data type: float;

> **units**: The units the input data is given in.
>
> *default: ''; data type: string;*

replay_file: This describes the location to read a trajectory file from.

default: ''; data type: string;

> **mode**: The input data format. 'xyz' and 'pdb' stand for xyz and pdb input files respectively. 'chk' stands for initialization from a checkpoint file.
>
> *default: ''; data type: string; options: 'xyz', 'pdb', 'chk';*

fixcom: This describes whether the centre of mass of the particles is fixed.

default: True ; data type: boolean;

temperature: The temperature of the system.

dimension: temperature; default: 1.0 ; data type: float;

> **units**: The units the input data is given in.
>
> *default: ''; data type: string;*

4.12 FORCES

Deals with creating all the necessary forcefield objects.

socket: Deals with the assigning of force calculation jobs to different driver codes, and collecting the data, using a socket for the data communication.

> **weight**: A scaling factor for this forcefield, to be applied before adding the force calculated by this forcefield to the total force.
>
> *default: 1.0 ; data type: float;*
>
> **mode**: Specifies whether the driver interface will listen onto a internet socket [inet] or onto a unix socket [unix].
>
> *default: 'inet'; data type: string; options: 'unix', 'inet';*
>
> **pbc**: Applies periodic boundary conditions to the atoms coordinates before passing them on to the driver code.
>
> *default: True ; data type: boolean;*
>
> **nbeads**: If the forcefield is to be evaluated on a contracted ring polymer, this gives the number of beads that are used. If not specified, the forcefield will be evaluated on the full ring polymer.
>
> *default: 0 ; data type: integer;*

4.13 SOCKET

Deals with the assigning of force calculation jobs to different driver codes, and collecting the data, using a socket for the data communication.

> **weight**: A scaling factor for this forcefield, to be applied before adding the force calculated by this forcefield to the total force.
>
> *default: 1.0 ; data type: float;*
>
> **mode**: Specifies whether the driver interface will listen onto a internet socket [inet] or onto a unix socket [unix].
>
> *default: 'inet'; data type: string; options: 'unix', 'inet';*
>
> **pbc**: Applies periodic boundary conditions to the atoms coordinates before passing them on to the driver code.
>
> *default: True ; data type: boolean;*
>
> **nbeads**: If the forcefield is to be evaluated on a contracted ring polymer, this gives the number of beads that are used. If not specified, the forcefield will be evaluated on the full ring polymer.
>
> *default: 0 ; data type: integer;*

latency: This gives the number of seconds between each check for new clients.

default: 0.001 ; data type: float;

slots: This gives the number of client codes that can queue at any one time.

default: 4 ; data type: integer;

port: This gives the port number that defines the socket.

default: 65535 ; data type: integer;

timeout: This gives the number of seconds before assuming a calculation has died. If 0 there is no timeout.

default: 0.0 ; data type: float;

address: This gives the server address that the socket will run on.

default: 'localhost'; data type: string;

4.14 CELL

Deals with the cell parameters. Takes as array which can be used to initialize the cell vector matrix.

dimension: length; data type: float;

units: The units the input data is given in.

default: ''; data type: string;

shape: The shape of the array.

default: (0,) ; data type: tuple;

4.15 BEADS

Describes the bead configurations in a path integral simulation.

natoms: The number of atoms.

default: 0 ; data type: integer;

nbeads: The number of beads.

default: 0 ; data type: integer;

q: The positions of the beads. In an array of size [nbeads, 3*natoms].

dimension: length; default: [ ] ; data type: float;

units: The units the input data is given in.

default: ''; data type: string;

shape: The shape of the array.

default: (0,) ; data type: tuple;

p: The momenta of the beads. In an array of size [nbeads, 3*natoms].

dimension: momentum; default: [ ] ; data type: float;

units: The units the input data is given in.

default: ''; data type: string;

shape: The shape of the array.

default: (0,) ; data type: tuple;

m: The masses of the atoms, in the format [m1, m2, ...].
dimension: mass; default: [ ] ; data type: float;

units: The units the input data is given in.
default: ''; data type: string;

shape: The shape of the array.
default: (0,) ; data type: tuple;

names: The names of the atoms, in the format [name1, name2, ...].
default: [ ] ; data type: string;

shape: The shape of the array.
default: (0,) ; data type: tuple;

4.16 ATOMS

Deals with a single replica of the system or classical simulations.

q: The positions of the atoms, in the format [x1, y1, z1, x2, ...].
dimension: length; default: [ ] ; data type: float;

units: The units the input data is given in.
default: ''; data type: string;

shape: The shape of the array.
default: (0,) ; data type: tuple;

p: The momenta of the atoms, in the format [px1, py1, pz1, px2, ...].
dimension: momentum; default: [ ] ; data type: float;

units: The units the input data is given in.
default: ''; data type: string;

shape: The shape of the array.
default: (0,) ; data type: tuple;

natoms: The number of atoms.
default: 0 ; data type: integer;

m: The masses of the atoms, in the format [m1, m2, ...].
dimension: mass; default: [ ] ; data type: float;

units: The units the input data is given in.
default: ''; data type: string;

shape: The shape of the array.
default: (0,) ; data type: tuple;

names: The names of the atoms, in the format [name1, name2, ...].
default: [ ] ; data type: string;

shape: The shape of the array.
default: (0,) ; data type: tuple;

4.17 NORMALMODES

Deals with the normal mode transformations, including the adjustment of bead masses to give the desired ring polymer normal mode frequencies if appropriate. Takes as arguments frequencies, of which different numbers must be specified and which are used to scale the normal mode frequencies in different ways depending on which 'mode' is specified.

dimension: frequency; data type: float;

units: The units the input data is given in.

default: ''; data type: string;

shape: The shape of the array.

default: (0,) ; data type: tuple;

mode: Specifies the technique to be used to calculate the dynamical masses. 'rpmd' simply assigns the bead masses the physical mass. 'manual' sets all the normal mode frequencies except the centroid normal mode manually. 'pa-cmd' takes an argument giving the frequency to set all the non-centroid normal modes to. 'wmax-cmd' is similar to 'pa-cmd', except instead of taking one argument it takes two ([wmax,wtarget]). The lowest-lying normal mode will be set to wtarget for a free particle, and all the normal modes will coincide at frequency wmax.

default: 'rpmd'; data type: string; options: 'pa-cmd', 'wmax-cmd', 'manual', 'rpmd';

transform: Specifies whether to calculate the normal mode transform using a fast Fourier transform or a matrix multiplication. For small numbers of beads the matrix multiplication may be faster.

default: 'fft'; data type: string; options: 'fft', 'matrix';

4.18 BAROSTAT

Simulates an external pressure bath.

mode: The type of barostat. Currently, only a 'isotropic' barostat is implemented, that combines ideas from the Bussi-Zykova-Parrinello barostat for classical MD with ideas from the Martyna-Hughes-Tuckerman centroid barostat for PIMD; see Ceriotti, More, Manolopoulos, Comp. Phys. Comm. 2013 for implementation details.

default: 'dummy'; data type: string; options: 'dummy', 'isotropic';

tau: The time constant associated with the dynamics of the piston.

dimension: time; default: 1.0 ; data type: float;

units: The units the input data is given in.

default: ''; data type: string;

thermostat: The thermostat for the cell. Keeps the cell velocity distribution at the correct temperature. Note that the 'pile_l', 'pile_g', 'nm_gle' and 'nm_gle_g' options will not work for this thermostat.

> **mode**: The style of thermostatting. 'langevin' specifies a white noise langevin equation to be attached to the cartesian representation of the momenta. 'svr' attaches a velocity rescaling thermostat to the cartesian representation of the momenta. Both 'pile_l' and 'pile_g' attaches a white noise langevin thermostat to the normal mode representation, with 'pile_l' attaching a local langevin thermostat to the centroid mode and 'pile_g' instead attaching a global velocity rescaling thermostat. 'gle' attaches a coloured noise langevin thermostat to the cartesian representation of the momenta, 'nm_gle' attaches a coloured noise langevin thermostat to the normal mode representation of the momenta and a langevin thermostat to the centroid and 'nm_gle_g' attaches a gle thermostat to the normal modes and a svr thermostat to the centroid.
>
> *data type: string; options: '', 'langevin', 'svr', 'pile_l', 'pile_g', 'gle', 'nm_gle', 'nm_gle_g';*

p: Momentum (or momenta) of the piston.

> *dimension: momentum; default: [ ] ; data type: float;*
>
> **units**: The units the input data is given in.
>
> *default: ''; data type: string;*
>
> **shape**: The shape of the array.
>
> *default: (0,) ; data type: tuple;*

4.19 THERMOSTATS

Simulates an external heat bath to keep the velocity distribution at the correct temperature.

> **mode**: The style of thermostatting. 'langevin' specifies a white noise langevin equation to be attached to the cartesian representation of the momenta. 'svr' attaches a velocity rescaling thermostat to the cartesian representation of the momenta. Both 'pile_l' and 'pile_g' attaches a white noise langevin thermostat to the normal mode representation, with 'pile_l' attaching a local langevin thermostat to the centroid mode and 'pile_g' instead attaching a global velocity rescaling thermostat. 'gle' attaches a coloured noise langevin thermostat to the cartesian representation of the momenta, 'nm_gle' attaches a coloured noise langevin thermostat to the normal mode representation of the momenta and a langevin thermostat to the centroid and 'nm_gle_g' attaches a gle thermostat to the normal modes and a svr thermostat to the centroid.
>
> *data type: string; options: '', 'langevin', 'svr', 'pile_l', 'pile_g', 'gle', 'nm_gle', 'nm_gle_g';*

A: The friction matrix for GLE thermostats.

> *dimension: frequency; default: [ ] ; data type: float;*
>
> **units**: The units the input data is given in.
>
> *default: ''; data type: string;*
>
> **shape**: The shape of the array.
>
> *default: (0,) ; data type: tuple;*

tau: The friction coefficient for white noise thermostats.

dimension: time; default: 0.0 ; data type: float;

units: The units the input data is given in.

default: ''; data type: string;

C: The covariance matrix for GLE thermostats.

dimension: temperature; default: [ ] ; data type: float;

units: The units the input data is given in.

default: ''; data type: string;

shape: The shape of the array.

default: (0,) ; data type: tuple;

ethermo: The initial value of the thermostat energy. Used when the simulation is restarted to guarantee continuity of the conserved quantity.

dimension: energy; default: 0.0 ; data type: float;

units: The units the input data is given in.

default: ''; data type: string;

s: Input values for the additional momenta in GLE.

dimension: ms-momentum; default: [ ] ; data type: float;

units: The units the input data is given in.

default: ''; data type: string;

shape: The shape of the array.

default: (0,) ; data type: tuple;

pile_scale: Scaling for the PILE damping relative to the critical damping.

default: 1.0 ; data type: float;

4.20 PRNG

Deals with the pseudo-random number generator.

has_gauss: Determines whether there is a stored gaussian number or not. A value of 0 means there is none stored.

default: 0 ; data type: integer;

state: Gives the state vector for the random number generator. Avoid directly modifying this unless you are very familiar with the inner workings of the algorithm used.

default: [ ] ; data type: integer;

shape: The shape of the array.

default: (0,) ; data type: tuple;

seed: This is the seed number used to generate the initial state of the random number generator.

default: 123456 ; data type: integer;

set_pos: Gives the position in the state array that the random number generator is reading from.

default: 0 ; data type: integer;

gauss: The stored Gaussian number.

default: 0.0 ; data type: float;

4.21 OUTPUTS

This class defines how properties, trajectories and checkpoints should be output during the simulation. May contain zero, one or many instances of properties, trajectory or checkpoint tags, each giving instructions on how one output file should be created and managed.

prefix: A string that will be prepended to each output file name. The file name is given by 'prefix'.'filename' + format_specifier. The format specifier may also include a number if multiple similar files are output.

default: 'i-pi'; data type: string;

checkpoint: Each of the checkpoint tags specify how to create a checkpoint file, which can be used to restart a simulation.

data type: integer;

stride: The number of steps between successive writes.

default: 1 ; data type: integer;

overwrite: This specifies whether or not each consecutive checkpoint file will overwrite the old one.

default: True ; data type: boolean;

filename: A string to specify the name of the file that is output. The file name is given by 'prefix'.'filename' + format_specifier. The format specifier may also include a number if multiple similar files are output.

default: 'restart'; data type: string;

trajectory: Each of the trajectory tags specify how to create a trajectory file, containing a list of per-atom coordinate properties.

data type: string;

format: The output file format.

default: 'xyz'; data type: string; options: 'xyz', 'pdb';

filename: A string to specify the name of the file that is output. The file name is given by 'prefix'.'filename' + format_specifier. The format specifier may also include a number if multiple similar files are output.

default: 'traj'; data type: string;

bead: Print out only the specified bead. A negative value means print all.

default: -1 ; data type: integer;

stride: The number of steps between successive writes.

default: 1 ; data type: integer;

flush: How often should streams be flushed. 1 means each time, zero means never.

default: 1 ; data type: integer;

cell_units: The units for the cell dimensions.

default: ''; data type: string;

properties: Each of the properties tags specify how to create a file in which one or more properties are written, one line per frame.

data type: string;

stride: The number of steps between successive writes.

default: 1 ; data type: integer;

shape: The shape of the array.

default: (0,) ; data type: tuple;

flush: How often should streams be flushed. 1 means each time, zero means never.

default: 1 ; data type: integer;

filename: A string to specify the name of the file that is output. The file name is given by 'prefix'.'filename' + format_specifier. The format specifier may also include a number if multiple similar files are output.

default: 'out'; data type: string;

4.22 CHECKPOINT

This class defines how a checkpoint file should be output. Optionally, between the checkpoint tags, you can specify one integer giving the current step of the simulation. By default this integer will be zero.

data type: integer;

stride: The number of steps between successive writes.

default: 1 ; data type: integer;

overwrite: This specifies whether or not each consecutive checkpoint file will overwrite the old one.

default: True ; data type: boolean;

filename: A string to specify the name of the file that is output. The file name is given by 'prefix'.'filename' + format_specifier. The format specifier may also include a number if multiple similar files are output.

default: 'restart'; data type: string;

4.23 PROPERTIES

This class deals with the output of properties to one file. Between each property tag there should be an array of strings, each of which specifies one property to be output.

data type: string;

stride: The number of steps between successive writes.

default: 1 ; data type: integer;

shape: The shape of the array.

default: (0,) ; data type: tuple;

flush: How often should streams be flushed. 1 means each time, zero means never.

default: 1 ; data type: integer;

filename: A string to specify the name of the file that is output. The file name is given by 'prefix'.'filename' + format_specifier. The format specifier may also include a number if multiple similar files are output.

default: 'out'; data type: string;

4.24 TRAJECTORY

This class defines how one trajectory file should be output. Between each trajectory tag one string should be given, which specifies what data is to be output.

data type: string;

format: The output file format.

default: 'xyz'; data type: string; options: 'xyz', 'pdb';

filename: A string to specify the name of the file that is output. The file name is given by 'prefix'.'filename' + format_specifier. The format specifier may also include a number if multiple similar files are output.

default: 'traj'; data type: string;

bead: Print out only the specified bead. A negative value means print all.

default: -1 ; data type: integer;

stride: The number of steps between successive writes.

default: 1 ; data type: integer;

flush: How often should streams be flushed. 1 means each time, zero means never.

default: 1 ; data type: integer;

cell_units: The units for the cell dimensions.

default: ''; data type: string;

A simple tutorial

Here we give a simple step-by-step guide through an example simulation, exploring some of the more generally useful options that i-PI offers and making no assumptions of previous experience of this code or other MD codes. Excerpts from the relevant input files are reproduced here, for explanation purposes, but to get the most out of this tutorial the user is strongly encouraged to work through it themselves. For this purpose, the input files have been included with the i-PI distribution, in the "test/tutorial" directory.

The chosen problem is that of a small NPT simulation of para-hydrogen, using the Silvera-Goldman potential [24]. We will take (N,P,T) = (108, 0, 25 K).

5.1 Part 1 - NVT Equilibration run

5.1.1 Client code

Let us now consider the problem of how to use i-PI to run a NPT simulation of para-hydrogen. The first thing that is required is a client code that is capable of calculating the potential interactions of para-hydrogen molecules. Fortunately, one of the client codes distributed with i-PI has an appropriate empirical potential already hard-coded into it, and so all that is required is to create the "driver.x" file in the "drivers" directory, using the UNIX utility make.

This client code can be used for several different problems (see 2.3.2.1), some of which are explored in the "examples" directory, but for the current problem we will use the Silvera-Goldman potential with a cut-off radius of 15 a_0. This is run using the following command:

```
> ./driver.x -m sg -h localhost -o 15 -p 31415
```

The option "-m" is followed by the empirical potential required, in this case we use "sg" for Silvera-Goldman, "-h localhost" sets up the client hostname as "localhost", "-o 15" sets the cut-off to 15 a_0, and "-p 31415" sets the port number to 31415.

Note that usually this step will require setting up appropriate client code input files, possibly for an *ab initio* electronic structure code, and so is generally a more involved process. Refer to 2.3.2, and the documentation of the appropriate client code, for more details on how to do this step.

5.1.2 Creating the xml input file

Now that the client code is ready, an appropriate xml input file needs to be created from which the host server and the simulation data can be initialized. Here, we will go step by step through the creation of a minimal input file for a simple NVT equilibration run. Note that the working final version is held within the "tutorial-1" directory for reference.

Firstly, when reading the input file the i-PI xml functions look for a "simulation" tag as a sign to start reading data. For those familiar with xml jargon, we have defined "simulation" as the root tag, so all the input data read in must start and end with a "simulation" tag, as show below:

```
<simulation>
   Input data here...
</simulation>
```

xml syntax requires a set of hierarchially nested tags, each of which contain data and/or more tags. Also, i-PI itself requires certain tags to be present, and keeps track of which tags are supposed to be where. More information about which tags are available can be found in 4, more information on xml syntax can be found in 3.1.1, and possible errors which can occur if the input file is not well formed can be found in 6.

For the sake of this first tutorial however, we will simply discuss the those tags which are needed for a single NVT equilibration run. The most important tags are "initialize", "ensemble", "total_steps" and "forces". These correspond to the tag to initialize the atom configurations, the tag to define the appropriate ensemble, the tag to set the length of the simulation and the tag to specify the client code respectively. We will also discuss "output", which is used to define what output data is generated by the code.

At this point then, the input file looks like:

```
<simulation>
   <total_steps>
      ...
   </total_steps>
   <initialize>
      ...
   </initialize>
   <forces>
      ...
   </forces>
   <ensemble>
      ...
   </ensemble>
   <output>
      ...
   </output>
</simulation>
```

5.1.2.1 Initializing the configurations

Now let us consider each of these tags in turn. Firstly, "initialize". As the name suggests, this initializes the state of the system, so this is where we will specify the atom positions and the cell parameters. Firstly, this takes an attribute which specifies the number of replicas of the system, called "nbeads". An attribute is a particular type of xml syntax designed to specify a single bit of data, and has the following syntax:

```
<initialize nbeads='4'>
   ...
</initialize>
```

Note that an attribute forms part of the opening tag, and that the value being assigned to it is held within quotation marks. In this case, we have set the number of replicas, or beads, to 4.

Next, we must specify the atomic configuration. Rather than initialize the atom positions manually, we will instead use a separate configuration file for this purpose. Here we will discuss two of the input formats that are compatible with i-PI, xyz files and pdb files.

Note that, for the sake of this tutorial, we have included valid xyz and pdb input files in the "tutorial-1" directory called "our_ref.xyz" and "our_ref.pdb" respectively.

The xyz format is the simplest input format for a configuration file that i-PI accepts, and has the following syntax:

```
natoms
# COMMENT LINE: PUT TITLE OF FILE HERE
atom1   x1  y1  z1
atom2   x2  y2  z2
...
```

where "natoms" is replaced by an integer giving the total number of atoms, in this case 108, atom1 is a label for atom 1, in this case H2 (since we are simulating para-hydrogen), and (x1, y1, z1) are the x, y and z components of atom 1 respectively.

Note that we are treating the para-hydrogen molecules isotropically here, i.e. as spherical psuedo-atoms. For the current system this is a good approximation, since at the state point under consideration every molecule is in its rotational ground state. For further details on this potential, and a demonstration of its application to quantum dynamics, see [24] and [26].

Other than its simplicity, the main advantage of this type of file is that it is free-formatted, and so there is no set precision to which each value must be written. This greatly simplifies both reading and writing these files.

The other file format that we can use is the pdb format. This has the following structure:

```
TITLE insert title here...
CRYST1        a        b        c      A      B      C P 1          1
ATOM      1   n1   1     1          x1      y1      z1  0.00  0.00            0
ATOM      2   n2   1     1          x2      y2      z2  0.00  0.00            0
...
```

where a, b and c are the cell vector lengths, A, B and C are the angles between them, n1 and n2 are the labels for atoms 1 and 2, and (x1, y1, z1) and (x2, y2, z2) give the position vectors of atoms 1 and 2.

Note that this is fixed-formatted, so the number of spaces matters. Essentially, the above format needs to be copied verbatim, using the same column widths and all the same keywords. For an exact specification of the file format (of which only a subset is implemented with i-PI) see http://deposit.rcsb.org/adit/docs/pdb_atom_format.html.

Here we will show how to specify the xml input file in both of these cases, assuming that the user has already created the configuration file themselves. Note that these file formats can be read by visualization programs such as VMD, and so it is generally

advised when making your own input files to use such software to make sure that the configuration is as expected.

To use a configuration file the "file" tag in "initialize" should be used. This will take an input file with a given name and use it to initialize all relevant data. Both of these formats have the atom positions and labels, so this will initialize the positions, labels and masses of all the particles in the system, with the masses being implicitly set based on the atom label. The pdb configuration file will also be used to set the cell parameters.

Let us take these two file types in turn, and form the appropriate input sections. First, the xyz file. There are two attributes which are relevant to the "file" tag for our current problem, "mode" and "units". "mode" is used to describe what kind of data is being used to initialize from, and so in this case will be "xyz". "units" specifies which units the file is given in, and so in this case is given by "angstrom", which are the standard units of both xyz and pdb files. Note that if no units are specified then atomic units are assumed. For more information on the i-PI unit conversion libraries, and the available units, see 3.1.1.1.

This then gives:

```
<initialize nbeads='4'>
   <file mode='xyz' units='angstrom'> our_ref.xyz </file>
   ...
</initialize>
```

In this case, since the cell parameters are not specified in the configuration file we must specify them separately. To initialize just the cell parameters, we use the tag "cell". These could in theory be set using a separate file, but here we will initialize them manually. Taking a cubic cell with cell parameter 17.847 angstroms, we can specify this using the "cell" tag in three different ways:

```
<cell mode='manual' units='angstrom'>
   [17.847, 0, 0, 0, 17.847, 0, 0, 0, 17.847]
</cell>

<cell mode='abcABC' units='angstrom'>
   [17.847, 17.847, 17.847, 90, 90, 90]
</cell>

<cell mode='abc' units='angstrom'>
   [17.847, 17.847, 17.847]
</cell>
```

Note the use of the different "mode" attributes, "manual", "abcABC" and "abc". The first creates the cell vector matrix manually, the second takes the length of the three unit vectors and the angles between them in degrees, and the last assumes an orthorhombic cell and so only takes the length of the three unit vectors as arguments. We will take the last version for brevity, giving as our final "initialize" section:

```
<initialize nbeads='4'>
   <file mode='xyz' units='angstrom'> our_ref.xyz </file>
   <cell mode='abc' units='angstrom'>
      [17.847, 17.847, 17.847]
   </cell>
   ...
</initialize>
```

The pdb file is specified in a similar way, except that no "cell" tag needs be specified and the "mode" tag should be set to "pdb":

```
<initialize nbeads='4'>
   <file mode='pdb' units='angstrom'> our_ref.pdb </file>
   ...
</initialize>
```

As well as initializing all the atom positions, this section can also be used to set the atom velocities. Rather than setting these manually, it is usually simpler to sample these randomly from a Maxwell-Boltzmann distribution. This can be done using the "velocities" tag by setting the "mode" attribute to "thermal". This then takes an argument specifying the temperature to initialize the velocities to. With this, the final "initialize" section is:

```
<initialize nbeads='4'>
   <file mode='pdb' units='angstrom'> our_ref.pdb </file>
   <velocities mode='thermal' units='kelvin'> 25 </velocities>
</initialize>
```

5.1.2.2 Creating the server socket

Next let us consider the "forces" section, which deals with communication with the client codes. Since in this example we do not use ring-polymer contraction, we only need to specify a single "socket" tag:

```
<forces>
   <socket>
      ...
   </socket>
</forces>
```

A socket is specified with three parameters; the port number, the hostname and whether it is a unix or an internet socket. These are specified by the "port" and "address" tags and the "mode" attribute respectively. To match up with the client socket specified above, we will take an internet socket on the hostname localhost and use port number 31415.

This gives the final "forces" section:

```
<forces>
   <socket mode="inet">
      <address> localhost </address>
      <port> 31415 </port>
   </socket>
</forces>
```

5.1.2.3 Generating the correct ensemble

The next section that we will need is "ensemble", which determines how the dynamics integrator will be initialized. Since we wish to do a NVT simulation, we set the "mode" attribute to "nvt" (note that we use lower case, and that the tags are case sensitive), and must specify the temperature using the appropriate tag:

```
<ensemble mode='nvt'>
   ...
   <temperature units='kelvin'> 25 </temperature>
</ensemble>
```

This defines the ensemble that will be sampled. We also must decide which integration algorithm to use, and how large the time step should be. In general, the time step should be made as large as possible without there being a drift in the conserved quantity. Usually we would take a few short runs with different time steps to try and optimize this, but for the sake of this tutorial we will use a safe value of 1 femtosecond, giving:

```
<ensemble mode='nvt'>
   ...
   <timestep units='femtosecond'> 1 </timestep>
   <temperature units='kelvin'> 25 </temperature>
</ensemble>
```

Finally, while the microcanonical part of the integrator is initialized automatically, there are several different options for the constant temperature sampling algorithm, specified by "thermostat". For simplicity we will use (the global version of) the path-integral Langevin equation (PILE) algorithm [2], which is specifically designed for path integral simulations. This is specified by the "mode" tag "pile_g". This integrator also has to be initialized with a time scale parameter, "tau", which determines how strong the thermostat is, which we will set to 25 femtoseconds. Putting all of this together, we get:

```
<ensemble mode='nvt'>
   <thermostat mode='pile_g'>
      <tau units='femtosecond'> 25 </tau>
   </thermostat>
   <timestep units='femtosecond'> 1 </timestep>
   <temperature units='kelvin'> 25 </temperature>
</ensemble>
```

Now that we have decided on the time step, we will decide the total number of steps to run the simulation for. Equilibrating the system is likely to take around 5 picoseconds, so we will take 5000 time steps, using:

```
<total_steps> 5000 </total_steps>
```

5.1.2.4 Customizing the output

So far, we have only considered how to set up the simulation, and not the data we wish to obtain from it. However, there are a wide variety of properties of interest that i-PI can calculate and a large number of different output options, so to avoid confusion let us go through them one at a time.

Firstly, the amount of data sent to standard output can be adjusted with the "verbosity" attribute of "simulation":

```
<simulation verbosity='high'>
   ...
</simulation>
```

By default the verbosity is set to "low", which only outputs important warning messages and information, and some statistical information every 1000 time steps. Here we will set it to "high", which will tell i-PI to output the following data every time step:

```
 # Average timings at MD step S. t/step: TOTAL [p: P q: Q t: T]
 # MD diagnostics: V: POTENTIAL Kcv: KINETIC Ecns: CONSERVED
 @SOCKET: Assigning [ X ] request id ID to client with last-id LID ( CID/ CTOT : )
```

where the output values have been replaced with the following:

S: This gives the current time step.

TOTAL: This gives the amount of time the current time step took.

P: This gives how long the momentum propagation step took.

Q: This gives how long the free-ring polymer propagation step took.

T: This gives how long the thermostat integration step took.

POTENTIAL: This gives the current potential energy of the system.

KINETIC: This gives the current kinetic energy of the system.

CONSERVED: This gives the current conserved quantity.

X: This says whether or not i-PI found a match for the calculation of replica ID or not. If one of the connected client codes calculated the forces for this replica on the last time step, then X will be "match", and i-PI will automatically assign this replica to the same client as before. This should happen with all the replicas if CTOT is the same as the number of beads.

ID: The index of the replica currently being assigned to a client code.

LID: The index of the replica which the client code last did a force calculation of.

CID: The index of the client code in the list of all connected client codes.

CTOT: The total number of connected client codes.

What output gets written to file is specified by the "output" tag. There are three types of files; properties files, trajectory files and checkpoint files, which are specified with "properties", "trajectory" and "checkpoint" tags respectively. For an in-depth discussion on these three types of output files see 3.2, but for now let us just explain the rationale behind each of these output file types in turn.

checkpoint files: These give a snapshot of the state of the simulation. If used as an input file for a new i-PI simulation, this simulation will start from the point where the checkpoint file was created in the old simulation.

trajectory files: These are used to print out properties relevant to all the atoms, such as the velocities or forces, for each degree of freedom. These can be useful for calculating correlation functions or radial distribution functions, but possibly their most useful feature is that visualization programs such as VMD can read them, and then use this data to show a movie of how the simulation is progressing.

properties files: These are usually used to print out system level properties, such as the timestep, temperature, or kinetic energy. Essentially these are used to keep track of a small number of important properties, either to visualize the progress of the simulation using plotting programs such as gnuplot, or to be used to get ensemble averages.

Now that we know what each input file is used for, let us take an example of an output section and show how the xml input section works. The default output, i.e. what would be output if nothing was set by the user, would be generated with the following "output" section:

```
<output prefix='i-pi'>
   <properties filename='md' stride='10'>
      [time, step, conserved, temperature, potential, kinetic_cv]
   </properties>
   <trajectory filename='pos' stride='100' format='xyz'>
      positions
   </trajectory>
   <checkpoint filename='checkpoint' stride='1000' overwrite='True'/>
</output>
```

This creates 6 files: "i-pi.md", "i-pi.pos_0.xyz", "i-pi.pos_1.xyz", "i-pi.pos_2.xyz", "i-pi.pos_3.xyz" and "i-pi.checkpoint". "i-pi.md" is the properties file, "i-pi.pos_x.xyz" are the position trajectory files, and "i-pi.checkpoint" is the checkpoint file.

The filenames are created using the syntax "prefix"."filename"[_(file specifier)][.(file format)], where the file specifier is added to separate similar files. For example, in the above case the different position trajectories for each bead are given a file specifier corresponding to the appropriate bead index.

The "stride" attributes set how often data is output to each file; so in the above case the properties are written out every 10 time steps, the trajectories every 100, and the checkpoints every 1000. The "format" attribute sets the format of the trajectory files, and the "overwrite" attribute sets whether each checkpoint file overwrites the previous one or not.

There are several options we can use to customize the output data. Firstly, the "prefix" attribute should be set to something which can be used to distinguish the files from different simulation runs. In this case we can simply set it to "tut1":

```
<output prefix='tut1'>
   ...
</output>
```

As for the input parameters, the units the output data is given in can be set by the user. Unlike the input parameters however, this is done by specifying an appropriate unit in curly braces after the name of the property or trajectory of interest, as shown below:

```
<output prefix='tut1'>
   <properties filename='md' stride='10'>
      [step, time{picosecond}, conserved{kelvin},
       temperature{kelvin}, potential{kelvin}, kinetic_cv{kelvin}]
   </properties>
   <trajectory filename='pos' stride='100' format='xyz'>
      positions{angstrom}
   </trajectory>
   <checkpoint filename='checkpoint' stride='1000' overwrite='True'/>
</output>
```

Next, let us adjust some of the attributes. Let us suppose that we wish to output the properties every time step, to check for conserved quantity jumps, and to output the trajectory in pdb format. To do this we would set the "stride" and "format" tags, as shown below:

```
<output prefix='tut1'>
   <properties filename='md' stride='1'>
      [step, time{picosecond}, conserved{kelvin},
       temperature{kelvin}, potential{kelvin}, kinetic_cv{kelvin}]
   </properties>
   <trajectory filename='pos' stride='100' format='pdb' cell_units='angstrom'>
      positions{angstrom}
   </trajectory>
   <checkpoint filename='checkpoint' stride='1000' overwrite='True'/>
</output>
```

Note that we have added a "cell_units" attribute to the "trajectory" tag, so that the cell parameters are consistent with the position output.

Finally, let us suppose that we wished to output another output property to a different file to the others. One example of when this might be necessary is if there were an output property which was more expensive to calculate than the others, and so it would be impractical to output it every time step. With i-PI this is easy to do, all that is required is to add another "properties" tag with a different filename.

For demonstration purposes, we will choose to print out the forces acting on one tagged bead, since this requires an argument to be passed to the function that calculates it. The i-PI syntax for doing this is to have the arguments to be passed to the function between standard braces, separated by semi-colons.

To print out the forces acting on one bead we need the "atom_f" property, which takes two arguments, "atom" and "bead", giving the index of the atom and bead tagged respectively. The appropriate syntax is then given below:

```
<properties>
   [atom_f(atom=0;bead=0)]
</properties>
```

This will print out the force vector acting on bead 0 of atom 0. i-PI also accepts positional arguments (i.e. arguments not specified by a name, but just by their position in the list of arguments), and so this could also be written as:

```
<properties>
   [atom_f(0;0)]
</properties>
```

Finally, putting all this together, and adjusting some of the parameters of the new file, we get:

```
<output prefix='tut1'>
   <properties filename='md' stride='1'>
      [step, time{picosecond}, conserved{kelvin},
       temperature{kelvin}, potential{kelvin}, kinetic_cv{kelvin}]
   </properties>
   <properties filename='force' stride='20'>
      [atom_f{piconewton}(atom=0;bead=0)]
   </properties>
   <trajectory filename='pos' stride='100' format='pdb' cell_units='angstrom'>
      positions{angstrom}
   </trajectory>
   <checkpoint filename='checkpoint' stride='1000' overwrite='True'/>
</output>
```

5.1.3 Running the simulation

Now that we have a valid input file, we can run the test simulation. The "i-pi" script in the root directory is used to create an i-PI simulation from a xml input file. As explained in 2.3.1 this script is run (if we assume that we are in the "tutorial-1" directory) using:

```
> python ../../../i-pi tutorial-1.xml
```

This will start the i-PI simulation, creating the server socket and initializing the simulation data. This should at this point print out a header message to standard output, followed by a few information messages that end with "starting the polling thread main loop", which signifies that the server socket has been opened and is waiting for connections from client codes.

At this point the driver code is run in a new terminal from the "drivers" directory using the command specified above:

```
> ./driver.x -m sg -h localhost -o 15 -p 31415
```

The i-PI code should now output a message saying that a new client code has connected, and start running the simulation.

5.1.4 Output data

Once the simulation is finished (which should take about half an hour) it should have output "tut1.md", "tut1.force", "tut1.pos_0.xyz", "tut1.pos_1.xyz", "tut1.pos_2.xyz", "tut1.pos_3.xyz", "tut1.checkpoint" and "RESTART".

Firstly, we consider the checkpoint files, "tut1.checkpoint" and "RESTART". As mentioned before, these files can be used as a means of restarting the simulation from a previous point. As an example, the last checkpoint should have been at step 4999, and so we could rerun the last step using

```
> ../../../i-pi tut1.checkpoint
```

followed by running "driver.x" as before.

The difference between these two files is that, while "tut1.checkpoint" was specified by the user, "RESTART" is automatically generated at the end of every i-PI run. This file then is what we will need to initialize the NPT run, since it contains the state of the system after equilibration.

Next, let us look at the trajectory files. Since we have printed out the positions, these should tell us how the spatial distribution has equilibrated, and give us some insight into the atom dynamics. The easiest way to use these files, as discussed earlier, is to use the trajectory files as input to a visualization program such as VMD.

If we do this with these files, we see that the simulation started from a crystalline configuration and then over the course of the simulation began to melt. Since the state point studied and with the potential given para-hydrogen is a liquid [24], this is what we would expect.

Finally, let us check the "tut1.md" file. For the current problem, i.e. checking that we have a suitably equilibrated system, we should do two tests. Firstly, we should check that the conserved quantity does not exhibit any major drift, and second we should check to see if the properties of interest have converged. Using gnuplot, we can plot the relevant graphs using:

```
> gnuplot
> p './tut1.md' u 1:3 # Plots column 1, i.e. current simulation step,
```

```
> p './tut1.md' u 1:4 # against columns 3, 4, 5 and 6,
> p './tut1.md' u 1:5 # i.e. conserved quantity, temperature,
> p './tut1.md' u 1:6 # potential energy and kinetic energy
```

This will show that the conserved quantity has only a small drift upwards, the kinetic and potential energies have equilibrated, and the thermostat is keeping the temperature at the specified value. We have therefore specified a sufficiently short time step, chosen the thermostat parameters sensibly, and have equilibrated the properties of interest. Therefore this stage of the simulation is done, and we are ready to start the *NPT* run.

5.2 Part 2 - *NPT* simulation

Now that we have converged *NVT* simulation data, we can use this to initialize a *NPT* simulation. There are two ways of doing this, both of which involve using the RESTART file generated at the end of the *NVT* run as a starting point. Note that for simplicity we will again take $N = 108, T = 25K$, and use $P = 0$.

5.2.1 Modifying the RESTART file

Firstly, you can use the RESTART file directly, modifying it so that instead of continuing with the original *NVT* simulation it will instead start a new *NPT* simulation. We have included in the "tutorial-2" directory both a RESTART file from tutorial 1 and an adjusted file which will run *NPT* dynamics, "tutorial-2a.xml"

These adjustments start with resetting the "step" tag, so that it starts with the value 0. This can be done by simply removing the tag. Similarly, we can increase the total number of steps so that it is more suitable for collecting the necessary amount of *NPT* data, in this case we will set "total_steps" to 100000.

We will also update the output files, first by setting the filenames to start with "tut2a" rather than "tut1", and secondly by adding the volume and pressure to the list of computed properties so that we can check that the ensemble is being sampled correctly. Putting this together this gives:

```
<output prefix=''>
   <properties shape='(8)' filename='tut2a.md'>
      [ step, time{picosecond}, conserved{kelvin},
        temperature{kelvin}, potential{kelvin}, kinetic_cv{kelvin},
        pressure_cv{megapascal}, volume ]
   </properties>
   <properties stride='20' shape='(1)' filename='tut2a.force'>
      [ atom_f{piconewton}(atom=0;bead=0) ]
   </properties>
   <trajectory filename='tut2a.pos' stride='100' format='pdb' cell_units='angstrom'>
      positions{angstrom}
   </trajectory>
   <checkpoint stride='1000' filename='tut2a.restart'/>
</output>
```

Finally, we must change the "ensemble" tag so that the correct ensemble is sampled. The first thing that must be done is the "mode" tag must be changed from "nvt" to "npt", and a "pressure" tag must be added:

```
<ensemble mode='npt'>
   <pressure> 0 </pressure>
   ...
</ensemble>
```

We must also specify the constant pressure algorithm, using the tag "barostat". At present, i-PI only contains a stochastic barostat to apply pressure to an isotropic system, which can be specified with the option "isotropic". Anisotropic versions of the barostat may be added in future releases of i-PI.

The isotropic barostat also requires a thermostat to deal with the volume degree of freedom, which we will take to be a simple Langevin thermostat. This thermostat is specified in the same way as the one which does the constant temperature algorithms for the atomic degrees of freedom, and we will take its time scale to be 250 femtoseconds:

```
<ensemble mode='npt'>
   <pressure> 0 </pressure>
   <barostat mode='isotropic'>
      <thermostat mode='langevin'>
         <tau units='femtosecond'> 250 </tau>
      </thermostat>
      ...
   </barostat>
   ...
</ensemble>
```

Finally, we will take the barostat time scale to be 250 femtoseconds also, giving:

```
<ensemble mode='npt'>
   <pressure> 0 </pressure>
   <barostat mode='isotropic'>
      <thermostat mode='langevin'>
         <tau units='femtosecond'> 250 </tau>
      </thermostat>
      <tau units='femtosecond'> 250 </tau>
   </barostat>
   ...
</ensemble>
```

with the rest of the "ensemble" tag being the same as before.

5.2.2 Initialization from RESTART

A different way of initializing the simulation is to use the RESTART file as a configuration file, in the same way that the xyz/pdb files were used previously.

Firstly, the original input file "tutorial-1.xml" needs to be modified so that it will do a *NPT* simulation instead of *NVT*. This involves modifying the "total_steps", "output" and "ensemble" tags as above. Next, we replace the "initialize" tag section with:

```
<initialize nbeads='4'>
   <file mode='chk'> tutorial-1_RESTART </file>
</initialize>
```

Note that the "mode" attribute has been set to "chk" to specify that the file is a checkpoint file. This will then use the RESTART file to initialize the bead configurations and velocities and the cell parameters.

Again, there is a file in the "tutorial-2" directory for this purpose, "tutorial-2b.xml".

5.2.3 Running the simulation

Whichever method is used to create the input file, the simulation is run in the same way as before, using either "tutorial-2a.xml" or "tutorial-2b.xml" as the input file. Note how the volume fluctuates with time, as it is no longer held constant in this ensemble.

5.3 Part 3 - A fully converged simulation

As a final example, we note that at this state point 16 replicas and at least 172 particles are actually required to provide converged results. As a last tutorial then, you should repeat tutorials 1 and 2 with this number of replicas and atoms.

The directory "tutorial-3" contains *NVT* and *NPT* input files which can be used to do a fully converged *NPT* simulation from scratch, except that they are missing some of the necessary input parameters.

If these are chosen correctly and the simulation is run properly the volume will be 31 cm^3/mol and the total energy should be -48 K [3].

With this number of beads and atoms, the force calculation is likely to take much longer than it did in either tutorial 1 or 2. To help speed this up, we will now discuss how to parallelize the calculation over the sockets, and how to speed up the data transfer.

Firstly, in this simple case where we are calculating an isotropic, pair-wise interaction, the data transfer time is likely to be a significant proportion of the total calculation time. To help speed this up, there is the option to use a unix domain socket rather than an internet socket. These are optimized for local communication between processes on a single computer, and so for the current problem they will be much faster than internet sockets.

To specify this, we simply set the "mode" attribute of the "socket" tag to "unix":

```
<forces>
   <socket mode='unix'>
      ...
   </socket>
</forces>
```

We then specify that the client code should connect to a unix socket using the -u flag:

```
> ./driver.x -u -m sg -h localhost -o 15 -p 31415
```

Parallelizing the force calculation over the different replicas of the system is similarly easy, all that is required is to run the above command multiple times. For example, if we wish to run 4 client codes, we would use:

```
> for a in 1 2 3 4; do
>    ./driver.x -u -m sg -h localhost -o 15 -p 31415 &
> done
```

Using these techniques should help speed up the calculation considerably, at least in this simple case. Note however, that using unix domain sockets would give a negligible gain in speed in most simulations, since the force calculation usually takes much longer than the data transfer.

Troubleshooting

6.1 Input errors

- *not well-formed (invalid token)*: Seen if the input file does not have the correct xml syntax. Should be accompanied by a line number giving the point in the file where the syntax is incorrect.
- *mismatched tag*: One of the closing tags does not have the same name as the corresponding opening tag. Could be caused either by a misspelling of one of the tags, or by having the closing tag in the wrong place. This last one is a standard part of the xml syntax, if the opening tag of one item is after the opening tag of a second, then its closing tag should be before the closing tag of the second. Should be accompanied by a line number giving the position of the closing tag.
- *Uninitialized value of type* _____ or *Attribute/Field name* _____ *is mandatory and was not found in the input for property* _____: The xml file is missing a mandatory tag, i.e. one without which the simulation cannot be initialized. Find which tag name is missing and add it.
- *Attribute/tag name* _____ *is not a recognized property of* _____ *objects*: The first tag should not be found within the second set of tags. Check that the first tag is spelt correctly, and that it has been put in the right place.
- _____ *is not a valid option (*_____*)*: This attribute/tag only allows a certain range of inputs. Pick one of the items from the list given.
- _____ *is an undefined unit for kind* _____ or _____ *is not a valid unit prefix* or *Unit* _____ *is not structured with a prefix+base syntax*: The unit input by the user is not correct. Make sure it corresponds to the correct dimensionality, and is spelt correctly.
- *Invalid literal for int() with base 10:* _____ or *Invalid literal for float():* _____ or _____ *does not represent a bool value*: The data input by the user does not have the correct data type. See section 3.1.1 for what constitutes a valid integer/float/boolean value.
- *Error in list syntax: could not locate delimiters*: The array input data did not have the required braces. For a normal array use [], for a dictionary use {}, and for a tuple use ().

- *The number of atom records does not match the header of xyz file*: Self-explanatory.
- *list index out of range*: This will normally occur if the configuration is initialized from an invalid input file. This will either cause the code to try to read part of the input file that does not exist, or to set the number of beads to zero which causes this error in a different place. Check that the input file has the correct syntax.

6.2 Initialization errors

- *Negative _____ parameter specified.*: Self-explanatory.
- *If you are initializing cell from cell side lengths you must pass the 'cell' tag an array of 3 floats*: If you are attempting to initialize a cell using the "abc" mode, the code expects three floats corresponding to the three side lengths.
- *If you are initializing cell from cell side lengths and angles you must pass the 'cell' tag an array of 6 floats*: If you are attempting to initialize a cell using the "abcABC" mode, the code expects six floats corresponding to the three side lengths, followed by the three angles in degrees.
- *Cell objects must contain a 3x3 matrix describing the cell vectors.*: If you are attempting to initialize a cell using the "manual" mode, the code expects nine floats corresponding to the cell vector matrix side lengths. Note that the values of the lower-diagonal elements will be set to zero.
- *Array shape mismatch in q/p/m/names in beads input*: The size of the array in question does not have the correct number of elements given the number of atoms and the number of beads used in the rest of the input. If the number of beads is nbeads and the number of atoms natoms, then q and p should have shape (nbeads, 3*natoms) and m and names should have shape (natoms,).
- *No thermostat/barostat tag provided for* NVT/NPT *simulation*: Some ensembles can only be sampled if a thermostat and/or barostat have been defined, and so for simulations at constant temperature and/or pressure these tags are mandatory. If you wish to not use a thermostat/barostat, but still want to keep the ensemble the same, then use "dummy" mode thermostat/barostat, which simply does nothing.
- *Pressure/Temperature should be supplied for constant pressure/temperature simulation*: Since in this case the ensemble is defined by these parameters, these must be input by the user. Add the appropriate tags to the input file.
- *Manual path mode requires (nbeads-1) frequencies, one for each internal mode of the path.*: If the "mode" tag of "normal_modes" is set to "manual", it will expect an array of frequencies, one for each of the internal normal modes of the ring polymers.
- *PA-CMD mode requires the target frequency of all the internal modes.*: If the "mode" tag of "normal_modes" is set to "pa-cmd", it will expect an array of one frequency, to which all the internal modes of the ring polymers will be set.

- *WMAX-CMD mode requires [wmax, wtarget]. The normal modes will be scaled such that the first internal mode is at frequency wtarget and all the normal modes coincide at frequency wmax.*: If the "mode" tag of "normal_modes" is set to "wmax-cmd", it will expect an array of two frequencies, one two set the lowest frequency normal mode, and one for the other normal mode frequencies.

- *Number of beads _____ doesn't match GLE parameter nb= _____*: The matrices used to define the generalized Langevin equations of motion do not have the correct first dimension. If matrices have been downloaded from http://http://imx-cosmo.github.io/gle4md// make sure that you have input the correct number of beads.

- *Initialization tries to match up structures with different atom numbers*: If in the initialization any of the matrices has an array shape which do not correspond to the same number of atoms, then they cannot correspond to the same system. Check the size of the arrays specified if they have been input manually.

- *Cannot initialize single atom/bead as atom/bead index _____ is larger than the number of atoms/beads*: Self-explanatory. However, note that indices are counted from 0, so the first replica/atom is defined by an index 0, the second by an index 1, and so on.

- *Cannot initialize the momenta/masses/labels/single atoms before the size of the system is known.*: In the code, a beads object is created to hold all the information related to the configuration of the system. However, until a position vector has been defined, this object is not created. Therefore, whichever arrays are being initialized individually, the position vector must always be initialized first.

- *Trying to resample velocities before having masses.*: A Maxwell-Boltzmann distribution is partly defined by the atomic masses, and so the masses must be defined before the velocities can be resampled from this distribution.

- *Cannot thermalize a single bead*: It does not make sense to initialize the momenta of only one of the beads, and so i-PI does not give this functionality.

- *Initializer could not initialize _____*: A property of the system that is mandatory to properly run the simulation has not been initialized in either the "initialize" section or the appropriate section in beads.

- *Ensemble does not have a thermostat to initialize* or *There is nothing to initialize in non-GLE thermostats* or *Checkpoint file does not contain usable thermostat data*: These are raised if the user has tried to initialize the matrices for the GLE thermostats with a checkpoint file that either does not have a GLE thermostat or does not have a thermostat at all.

- *Size mismatch in thermostat initialization data*: Called if the shape of the GLE matrices defined in the checkpoint file is different from those defined in the new simulation.

- *Replay can only read from PDB or XYZ files – or a single frame from a CHK file*: If the user specifies a replay ensemble, the state of the system must be defined by either a configuration file or a checkpoint file, and cannot be specified manually.

6.3 Output errors

- *The stride length for the _____ file output must be positive.*: Self-explanatory
- *_____ is not a recognized property/output trajectory*: The string as defined in the "properties"/"trajectory" tag does not correspond to one of the available trajectories. Make sure that both the syntax is correct, and that the property has been spelt correctly.
- *Could not open file _____ for output*: Raised if there is a problem opening the file defined by the "filename" attribute.
- *Selected bead index _____ does not exist for trajectory _____*: You have asked for the trajectory of a bead index greater than the number of the replicas of the system. Note that indices are counted from 0, so the first replica is defined by an index 0, the second by an index 1, and so on.
- *Incorrect format in unit specification _____*: Usually raised if one of the curly braces has been neglected.
- *Incorrect format in argument list _____*: This will be raised either if one of the brackets has been neglected, or if the delimiters between arguments, in this case ";", are not correct. This is usually raised if, instead of separating the arguments using ";", they are instead separated by ",", since this causes the property array to be parsed incorrectly.
- *_____ got an unexpected keyword argument _____*: This will occur if one of the argument lists of one of the properties specified by the user has a keyword argument that does not match any of those in the function to calculate it. Check the properties.py module to see which property this function is calculating, and what the correct keyword arguments are. Then check the "properties" tag, and find which of the arguments has been misspelt.
- *Must specify the index of atom_vec property*: Any property which prints out a vector corresponding to one atom needs the index of that atom, as no default is specified.
- *Cannot output _____ as atom/bead index _____ is larger than the number of atoms/beads*: Self-explanatory. However, note that indices are counted from 0, so the first replica/atom is defined by an index 0, the second by an index 1, and so on.
- *Couldn't find an atom that matched the argument of _____*: For certain properties, you can specify an atom index or label, so that the property is averaged only over the atoms that match it. If however no atom labels match the argument given, then the average will be undefined. Note that for properties which are cumulatively counted rather than averaged, this error is not raised, and if no atom matches the label given 0 will be returned.

6.4 Socket errors

- *Address already in use*: This is called if the server socket is already being used by the host network. There are several possible reasons for getting this error. Firstly, it might simply be that two simulations are running concurrently using the same host and port number. In this case simply change the port number of one of the simulations. Secondly, you can get this error if you try to rerun a simulation that previously threw an exception, since it takes a minute or so before the host will disconnect the server socket if it is not shut down cleanly. In this case, simply wait for it to disconnect, and try again. Finally, you will get this error if you try to use a restricted port number (i.e. below 1024) while not root. You should always use a non-restricted port number for i-PI simulations.

- *Error opening unix socket. Check if a file /tmp/ipi____ exists, and remove it if unused.*: Similar to the above error, but given if you are using a unix socket rather than an internet socket. Since this binds locally the socket can be removed by the user, which means that it is not necessary to wait for the computer to automatically disconnect an unused server socket.

- *Port number ____ out of acceptable range*: The port number must be between 1 and 65535, and should be greater than 1024. Change the port number accordingly.

- *Slot number ____ out of acceptable range*: The slot number must be between 1 and 5. Change the slot number accordingly.

- *'NoneType' object has no attribute 'Up'*: This is called if an exception is raised during writing the data to output, and so the thread that deals with the socket is terminated uncleanly. Check the stack trace for the original exception, since this will be the actual source of the problem. Also note that, since the socket thread was not cleaned up correctly, the server socket may not have been disconnected properly and you may have to wait for a minute before you can restart a simulation using the same host and port number.

6.5 Mathematical errors

- *math domain error*: If the cell parameters are defined using the side lengths and angles, with either a pdb file or using the "abcABC" initialization mode, then for some value of the angles it is impossible to construct a valid cell vector matrix. This will cause the code to attempt to take the square root of a negative number, which gives this exception.

- *overflow encountered in exp*: Sometimes occurs in *NPT* runs when the simulation box "explodes". Make sure you have properly equilibrated the system before starting and that the timestep is short enough to not introduce very large integration errors.

Bibliography

[1] R. P. Feynman, A. R. Hibbs, Quantum Mechanics and Path Integrals, McGraw-Hill, New York, 1964.

[2] M. Ceriotti, M. Parinello, T. E. Markland, M. D. E., Efficient stochastic thermostating of path integral molecular dynamics, J. Phys. Chem. 133 (2010) 124101.

[3] G. J. Martyna, A. Hughes, M. E. Tuckerman, Molecular dynamics algorithms for path integrals at constant pressure, J. Chem. Phys. 110 (7) (1999) 3275.

[4] T. E. Markland, D. E. Manolopoulos, An efficient ring polymer contraction scheme for imaginary time path integral simulations, J. Chem. Phys. 129 (2008) 024105.

[5] M. Ceriotti, D. E. Manolopoulos, M. Parinello, Accelerating the convergence of path integral dynamics with a generalized langevin equation, J. Chem. Phys. 134 (2011) 084104.

[6] M. Suzuki, Hybrid exponential product formulas for unbounded operators with possible applications to monte carlo simulations, Phys. Lett. A 201 (1995) 425.

[7] S. A. Chin, Symplectic integrators from composite operator factorizations, Phys. Lett. A 226 (1997) 344.

[8] M. Ceriotti, G. a. R. Brain, O. Riordan, D. E. Manolopoulos, The inefficiency of re-weighted sampling and the curse of system size in high-order path integration, Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences 468 (2137) (2011) 2–17.

[9] A. Perez, M. E. Tuckerman, Improving the convergence of closed and open path integral molecular dynamics via higher order trotter factorization schemes, J. Chem. Phys. 135 (2011) 064104.

[10] J. Cao, G. A. Voth, A new perspective on quantum time correlation functions, J. Chem. Phys. 99 (12) (1993) 10070–10073.

[11] J. Cao, G. A. Voth, The formulation of quantum statistical mechanics based on the Feynman path centroid density. IV. Algorithms for centroid molecular dynamics, J. Chem. Phys. 101 (7) (1994) 6168–6183.

[12] I. R. Craig, D. E. Manolopoulos, Quantum statistics and classical mechanics: Real time correlation functions from ring polymer molecular dynamics, J. Chem. Phys. 121 (2004) 3368.

[13] B. J. Braams, D. E. Manolopoulos, On the short-time limit of ring polymer molecular dynamics., J. Chem. Phys. 125 (12) (2006) 124105.

[14] S. Habershon, D. E. Manolopoulos, T. E. Markland, T. F. Miller, Ring-polymer molecular dynamics: Quantum effects in chemical dynamics from classical trajectories in an extended phase space, Annu. Rev. Phys. Chem. 64 (2013) 387.

[15] P. Langevin, Sur la theorie du mouvement brownien, C. R. Acad. Sci. 146 (1908) 530.

[16] G. Bussi, M. Parrinello, Stochastic thermostats: comparison of local and global schemes, Comp. Phys. Comm. 179 (2008) 26.

[17] M. Ceriotti, G. Bussi, M. Parinello, Colored-noise thermostats a la carte, J. Chem. Theory Comput. 6 (2009) 1170.

[18] M. Ceriotti, D. E. Manolopoulos, Efficient first principles calculation of the quantum kinetic energy and momentum distribution of nuclei, Phys. Rev. Lett. 109 (2012) 100604.

[19] G. Bussi, T. Zykova-Timan, M. Parinello, Isothermal-isobaric molecular dynamics using stochastic velocity rescaling, J. Phys. Chem. 130 (2009) 074101.

[20] T. M. Yamamoto, Path-integral virial estimator based on the scaling of fluctuation coordinates: Application to quantum clusters with fourth-order propagators, J. Chem. Phys. 123 (2005) 104101.

[21] L. Lin, J. A. Morrone, R. Car, M. Parrinello, Displaced Path Integral Formulation for the Momentum Distribution of Quantum Particles, Phys. Rev. Lett. 105 (2010) 110602.

[22] S. Habershon, G. S. Fanourgakis, D. E. Manolopoulos, Comparison of path integral molecular dynamics methods for the infrared absorption spectrum of liquid water, J. Chem. Phys. 129 (2008) 074501.

[23] T. D. Hone, P. J. Rossky, G. A. Voth, A comparitive study of imaginary time path integral based methods for quantum dynamics, J. Chem. Phys. 124 (2006) 154103.

[24] I. F. Silvera, V. V. Goldman, The isotropic intermolecular potential for h2 and d2 in the solid and gas phases, J. Chem. Phys 69 (1978) 4209.

[25] L. Verlet, Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules, Phys. Rev. 159 (1) (1967) 98.

[26] T. F. Miller, D. E. Manolopoulos, Quantum diffusion in liquid para-hydrogen from ring-polymer molecular dynamics, J. Chem. Phys. 122 (2005) 184503.

[27] S. Habershon, T. E. Markland, D. E. Manolopoulos, Competing quantum effects in the dynamics of a flexible water model, J. Chem. Phys. 131 (2009) 024501.

[28] L. Lindsay, D. A. Broido, Optimized tersoff and brenner empirical potential parameters for lattice dynamics and phonon thermal transport in carbon nanotubes and graphene, Phys. Rev. B 81 (2010) 205441.

[29] L. Lindsay, D. A. Broido, Erratum: Optimized tersoff and brenner empirical potential parameters for lattice dynamics and phonon thermal transport in carbon nanotubes and graphene, Phys. Rev. B 82 (2010) 209903.