i-PI

Release 3.1.5.1

The i-PI developers

Jun 12, 2025

Contents

---

i-PI is a force engine written in Python 3 with the goal of performing standard and advanced molecular simulations. 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. Thus, i-PI effectively decouples the problem of evolving the ionic positions and the problem of computing the system-specific properties

i-PI was initially developed for Path Integral Molecular Dynamics (PIMD) simulations, and contains probably the most comprehensive array of PIMD techniques. However, it has evolved over time to become a general-purpose code with methods ranging from phonon calculators to replica exchange molecular dynamics.

The implementation of i-PI is efficient enough that, by tuning the socket parameters and avoiding excessive I/O, it can be used to run simulations using empirical forcefields or machine learning interatomic potentials with tens of thousands of atoms with only a small overhead.

This documentation is structured as follows:

1 Program Overview

i-PI is a force engine that operates with a client-server paradigm. i-PI acts as a server in command of the evolution of the nuclear positions, while one or more instances of the client code handle the evaluation of system-specific properties. Designed to be universal, i-PI's architecture is agnostic to the identity of the force providers, with a general and flexible backend that accommodates to a wide range of client codes.

The code is written in Python 3, a high-level, general-purpose interpreted programming language known for its emphasis on code readability. This choice facilitates rapid prototyping of new ideas and relatively easy code maintenance when compared to compiled languages traditionally used in scientific computing.

i-PI is structured in a modular way that represents the underlying physics of the target simulation as faithfully as possible. To achieve this, the code is organized around the System class which encodes all the information related to the physical system, such as the number and identity of atoms, the ensemble to be sampled, the number of replicas in path integral simulations, the initialization procedure, and the algorithm for evolving the nuclear positions. Forces are managed through the Forces class, which integrates the individual force components, each of which is computed following the strategy specified in a Forcefield class. This two-layer approach is particu- larly advantageous for algorithms where the total forces and energies are obtained by a combination of these quantities computed at different accuracy levels, or among different system portions. Simulation techniques that require the evolution of many systems simultaneously make use of the SMotion (Systems Motion) class. This class enables the definition of evolution steps that combine the (possibly many) different systems, facil- itating, for example, exchanges between system replicas as done in replica exchange simulations. Finally, estimators can be defined within the code or computed by the provided post-processing tools or user-generated scripts.

A schematic representation of the code structure and the server-client communication is presented in Fig. 1.

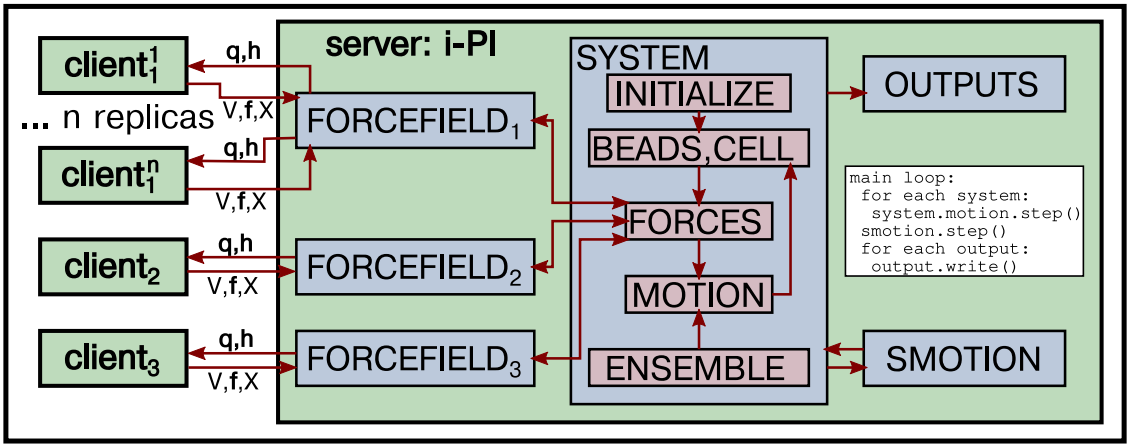

Fig. 1: Figure 1. Schematic representation of the i-PI code structure and the server-client communication.

In Figure 1, the physical system is defined by one or more replicas (beads), and the sampling conditions (e.g. temperature and pressure) by an Ensemble class. The forces acting on the atoms are constructed based on a combination of energy contributions evaluated by one or more instances of the Forcefield class (three in this example). Each Forcefield instance communicates with a different client sending positions (q) and lattice vectors (h), and receiving energy (V), forces (f), stresses and possible extra properties (X) as a JSON formatted string. In simulations with multiple replicas, each Forcefield instance can accept connections from several clients, to achieve parallel evaluation. The Motion class determines the evolution of the atoms (e.g. time integration, or geometry optimization), while "system motion" classes (SMotion) can act on multiple systems, e.g. to carry out replica exchange simulations. The output class handles writing the output and restart files

1.1 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 the 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 allows for fast communication on a single node, whereas the former makes it possible to realise a distributed computing paradigm, with clients running on different nodes or even on different HPC facilities. In order to facilitate the 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 *Distributed execution*.

1.2 Force evaluation

Within i-PI, the evaluation of the forces plays a crucial role, as it is the step requiring communication with the client code. In order to have a flexible infrastructure that makes it possible to perform simulations with advanced techniques, the force evaluation machinery in i-PI might appear complicated at first, and deserves a brief discussion.

This figure provides an overall scheme of the objects involved in the calculation of the forces. The infrastructure comprises a force provider class that deals with the actual subdivision of work among the clients, and a sequence of objects that translate the request of the overall force of the system into atomic evaluations of one component of the force When running path integral simulations, the latter refers to the component of an individual bead: i-PI is built to hide the path integral infrastructure from the client, and so beads must be transferred individually.

Let us discuss for clarity a practical example: a calculation of an empirical water model where the bonded interactions are computed on 32 beads by the program A, and the non-bonded interactions are computed by client B, ring-polymer contracted on 8 beads. Each client "type" is associated with a *forcefield* object in the input. In the case of a *ffsocket* interface, the forcefield object specifies the address to which a client should connect, and so multiple clients of type A or B can connect to i-PI at the same time. Each forcefield object deals with queueing force evaluation requests and computing them in a first-in-first-out fashion, possibly executing multiple requests in parallel.

On the force evaluation side, the task of splitting the request of a force evaluation into individual components and individual beads is accomplished by a chain of three objects, Forces, ForceComponent and ForceBead. is the main force Forces evaluator, that is built from the prototypes listed within the *forces* field of the *system*. Each *forcecomponent* item within the *forces* tag describe one component of the force – in our example one ForceComponent bound to a forcefield of type A, evaluated on 32 beads, and one ForceComponent bound to type B, evaluated on 8 beads. Forces contains the machinery that automatically contracts the actual ring polymer to the number of beads required by each component, and combines the various components with the given weights to provide the overall force, energy and virial where required. Note that in order to support ring polymer contraction (RPC), the RPC procedure is executed even if no contraction was required (i.e. even if all clients contain the full amount of beads). ForceComponent is a very simple helper class that associates with each bead a ForceBead object, which is the entity in charge of filing force requests to the appropriate ForceField object and waiting for completion of the evaluation.

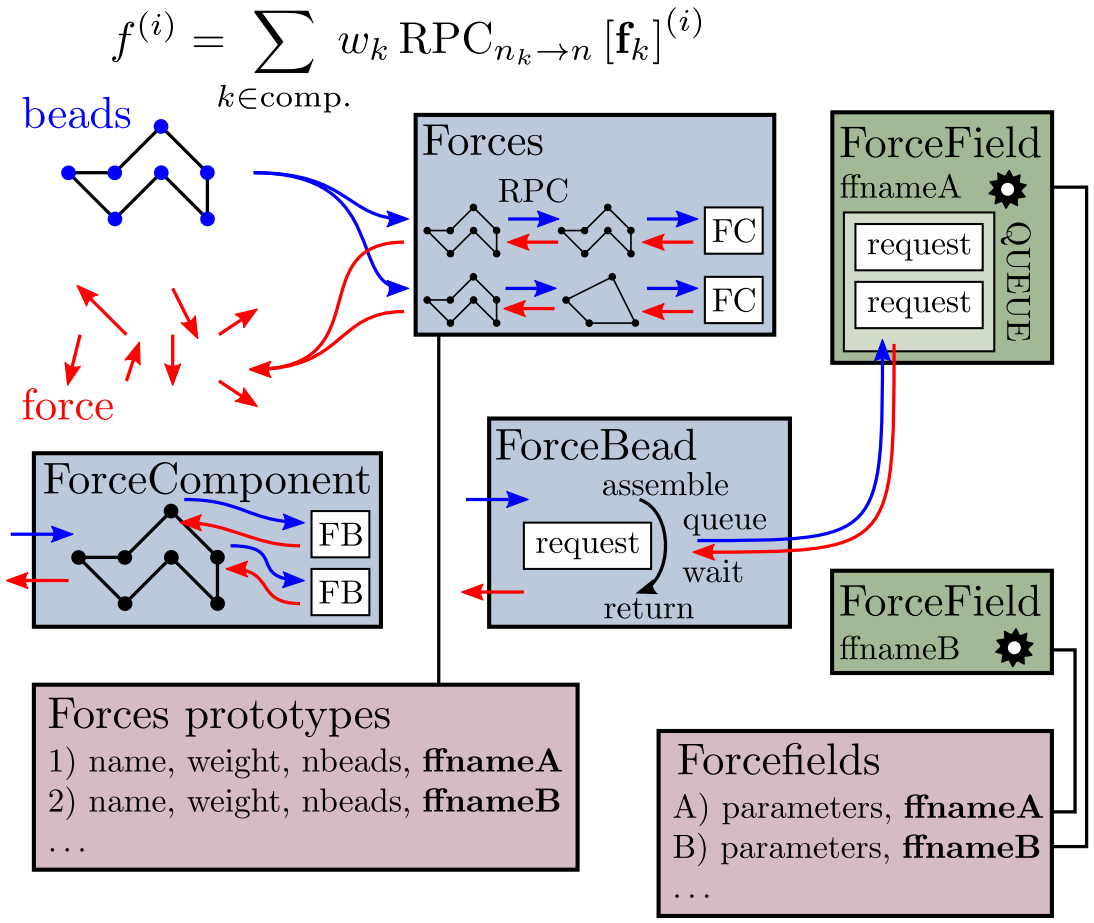

Fig. 2: Schematic representation of the different objects that are involved in the evaluation of the forces. The multiple layers and complex structure are necessary to give the possibility of decomposing the evaluation of the forces between multiple different clients and using different imaginary time partitioning (e.g. one can compute the bonded interactions using one client, and use a different client to compute the long-range electrostatic interactions, contracted on a single bead [MM08]).

1.3 Automated evaluation (depend objects)

i-PI uses a caching mechanism with automatic value updating to make the code used to propagate the dynamics as simply and clearly 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.

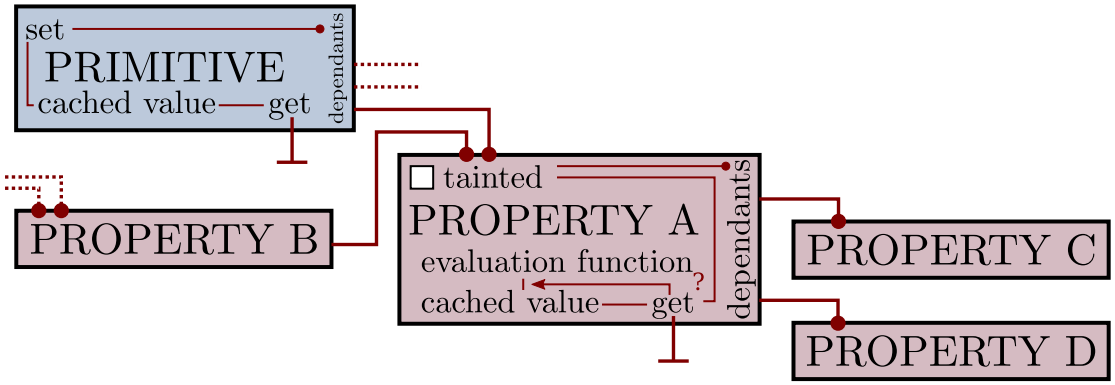

Fig. 3: Schematic overview of the functioning of the *depend* class used as the base for properties and physical quantities in i-PI. A few "primitive" quantities – such as atomic positions or momenta – can be modified directly. For most properties, one defines a function that can compute that property 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 when it is needed (i.e., when the quantity is assessed again).

This is a minimal example of how to implement dependencies in a class

```
from ipi.utils.depend import depend_value, dproperties

class DObject:
   def __init__(self):
      # depend objects are created using an underscore prefix.
      # this is a "primitive" value that doesn't depend on anything
      self._scalar = depend_value(value=1, name="scalar")

      # this is a dependent object. the definition contains a function that
      # is called to determine the value, and specification of what objects
      # it depend on.
      self._double = depend_value(func=lambda: 2*self.scalar, name="double",
                                  dependencies=[self._scalar])

# after the definition of a class, this helper function should be called to
# create property getters and setters (use the names with no leading underscore)
# note that property accessors are added to the class, not to the instances
dproperties(DObject, ["scalar", "double"])

myobj = DObject()
```

(continues on next page)

(continued from previous page)

```
# "primitive values" can be set manually
myobj.scalar = 4
# dependent objects will be computed automatically on demand
print(myobj.double) # prints `8`
```

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.

It is also possible to define dependencies between different objects, in which case it's necessary to make sure that the compute function has access, at runtime, to the value of the other object. A typical usage pattern is

```
# NB: this is meant to be run after the previous code snippet
from ipi.utils.depend import depend_array
import numpy as np

class DOther:
   def __init__(self):

      # depend arrays must be initialized with storage space
      self._vec = depend_array(value=np.ones(4), name="vec")

   def bind(self, factor):

      self.factor = factor # stores a reference to the object holding the value
      self._scaled = depend_array(value=np.ones(4), name="vec",
                                  func=self.get_scaled, dependencies=[self._vec])

      # dependencies (or dependants) can also be added after object creation
      self._scaled.add_dependency(self.factor._double)

   def get_scaled(self):
      # computes a scaled version of the vector
      return self.vec*self.factor.double

dproperties(DOther, ["vec", "scaled"])

myoth = DOther() # creates the object
myoth.bind(myobj) # makes connections

myoth.vec = np.asarray([0,1,2,3])
print(myoth.scaled) # prints [0,8,16,24]

myoth.vec[3] = 0 # depend_arrays can be accessed as normal np.ndarray
print(myoth.scaled) # prints [0,8,16,0]
```

Licence and credits

The code is distributed under the dual GPL and MIT licence. For more details see www.gnu.org/licences/gpl.html and https://fedoraproject.org/wiki/Licensing:MIT.

If you use this code in any future publications, please cite this using [CMM14] for v1, [KRM+19] for v2. [LKF+24] for v3.

1.4 Contributors

i-PI was originally written by M. Ceriotti and J. More at Oxford University, together with D. Manolopoulos. The updated list of developers and contributors can be found here

2 Getting started

2.1 Installing i-PI

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 `bin` directory of the distribution is the main (executable) script, and can be run as long as the system has installed:

- Python version 3.6 or greater (starting from version 2.0, i-PI is not Python2 compatible)
- The Python numerical library NumPy

See *Running i-PI* 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. Patching for use with i-PI should not introduce further dependencies.

Quick Setup

To use i-PI with an existing driver, install and update using *pip*:

Last version:

```
python -m pip install git+https://github.com/i-pi/i-pi.git
```

Last Release:

```
pip install -U ipi
```

Full installation

i-PI

To develop i-PI or test it with the self-contained driver, follow these instructions. It is assumed that i-PI will be run from a Linux environment, with a recent version of Python, Numpy and gfortran, and that the terminal is initially in the i-pi package directory (the directory containing this file), which you can obtain by cloning the repository

```
git clone https://github.com/i-pi/i-pi.git
```

Source the environment settings file *env.sh* as *source env.sh* or . *env.sh*. It is useful to put this in your *.bashrc* or other settings file if you always want to have i-PI available.

Fortran built-in driver

The built-in driver requires a FORTRAN compiler, and can be built as

```
cd drivers/f90
make
cd ../..
```

Python driver and PES

In addition to the FORTRAN drive, the i-PI distribution contains also a Python driver, available in *drivers/py* and through the command-line command *i-pi-py_driver*, which evaluates potential energy surfaces evaluated by simple driver classes, that can be found in *ipi/pes*.

These classes are particularly suitable to perform inference with machine-learning potentials implemented in Python, and it is reasonably simple to add your own, if you need to (see also the *Contributing* section).

These PES files can also be used directly, without the need to go through a client-server interface, using a *ffdirect* forcefield, including in the XML input a block similar to

```
<ffdirect name="ff_name">
   <pes> harmonic </pes>
   <parameters> { k1: 1.0} </parameters>
</ffdirect>
```

Alternative installation using the setup.py module

It is sometimes 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 privileges.

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 it 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
```

2.2 Installing clients

As of today, the following codes provide out-of-the-box an i-PI interface: CP2K, DFTB+, Lammps, Quantum ESPRESSO, Siesta, FHI-aims, Yaff, deMonNano, TBE. Links to the web pages of these codes, including information on how to obtain them, can be found at http://ipi-code.org/.

If you are interested in interfacing your code to i-PI please get in touch, we are always glad to help. We keep some information below in case you are interested in writing a patch to a code.

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, but it also makes running a simulation slightly more complicated, since the two components must be set up and started independently.

Running the i-PI server

i-PI simulations are run using the i-PI Python script found in the "bin" 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.

Running the client code

Below we give examples on how to make different clients communicate with i-PI. Most clients also include descriptions on how to do this from their own documentation.

Built-in, fortran client

i-PI 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/f90", and can be compiled into an executable "i-pi-driver" using the UNIX utility make.

This code currently has several empirical potentials hardcoded into it, including a Lennard-Jones potential, the Silvera-Goldman potential [SG78], a primitive implementation of the qtip4pf potential for water , [HMM09], several toy model potentials, the ideal gas (i.e. no potential interaction), and several more.

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

```
> i-pi-driver [-u] -a address [-p port] -m [model-name] -o [parameters] [-S sockets_
↪prefix] [-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.

-a:
: Is followed in the command line argument list by the hostname (address) 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. Other options should be clear from their description.

-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.

-S:
Optional parameter. If given, overwrite the default socket prefix used in the creation of files for the socket communication. (default "/tmp/ipi_")

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 *Testing the install*.

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 it the documentation of CP2K.

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:socket_name: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/.

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 ipi host_address port
```

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

```
fix  1 all ipi 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. Note that LAMMPS must be compiled with the `yes-user-misc` option to communicate with i-PI. More information from https://lammps.sandia.gov/doc/fix_ipi.html.

FHI-aims

To use FHI-aims 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 `control.in` file:

```
use_pimd_wrapper host_address port
```

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

```
use_pimd_wrapper UNIX:host_address port
```

One can also communicate different electronic-structure quantities to i-PI through the `extra` string from FHI-aims. In this case, the following lines can be added to the `control.in` file:

```
communicate_pimd_wrapper option
```

where option can be, e.g., `dipole, hirshfeld, workfunction, friction`.

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.

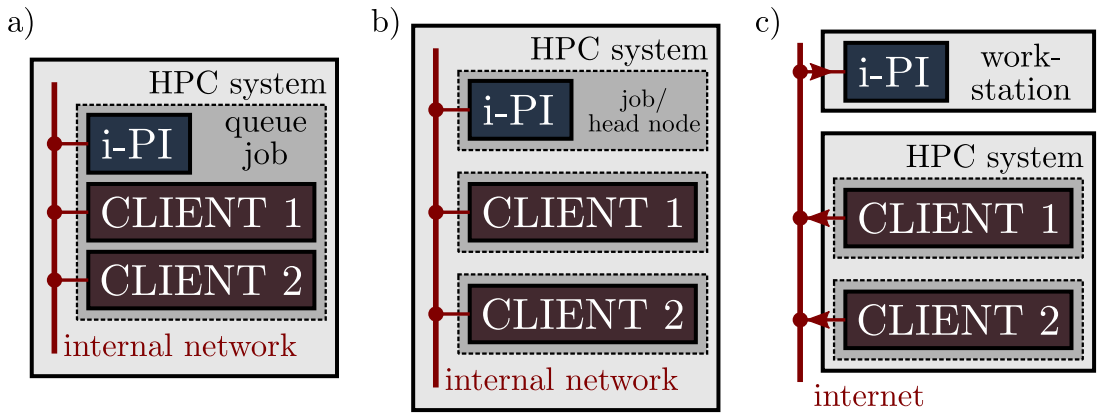

Fig. 4: Different approaches to run i-PI and a number of instances of the forces code on a HPC system: a) running i-PI and the clients in a single job; b) running i-PI and the clients on the same system, but using different jobs, or running i-PI interactively on the login node; c) running i-PI on a local workstation, communicating with the clients (that can run on one or multiple HPC systems) over the internet.

The figure 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 incoming 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 *Distributed execution* for more details).

2.4 Testing the install

test the installation with 'pytest'

There are several test cases included, that can be run automatically with 'i-pi-tests' from the root directory.

```
> i-pi-tests
```

Note 1: pytest and pytest-mock python packages are required to run these tests, but they are required to run i-PI. Note 2: please compile the fortran driver, as explained in *Built-in, fortran client*. Note 3: use the '-h' flag to see all the available tests

test cases and examples

The *examples/* folder contains a multitude of examples for i-PI, covering most of the existing functionalities, and including also simple tests that can be run with different client codes.

The example folder is structured such that each sub-folder is focused on a different aspect of using i-PI:

- **clients:**

  Contains examples that are code-specific, highlighting how the driver code should be set up
  (client-specific syntax and tags) to run it properly with i-PI

- **features :**

  Examples of different functionalities implemented in i-PI.
  All examples can be run locally with the drivers provided with the code.

- **hpc_scripts :**
  Examples of submission scripts on HPC platforms

- **temp :**

  Temporary folder with historical examples that have not yet been adapted
  to the current folder structure

- **init_files:**
  repository of input files shared by many examples

We keep this folder updated as much as we can, and try to run automated tests on these inputs, but in some cases, e.g. when using external clients, we cannot run tests. Please report a bug if you find something that is not working.

3 Core features

i-PI includes a large number of advanced molecular dynamics features, with an obvious focus on path integral molecular dynamics, but also several methods for sampling classical trajectories. This is an (incomplete) list of some of the main features in alphabetical order.

If you implement a major new feature, this is the place to briefly outline what it does and what are the papers to cite. Please follow as closely as possible the template.

3.1 Bosonic and Fermionic Path Integral Molecular Dynamics

PIMD simulations of bosonic particles with a polynomial scaling algorithm. Supports mixtures of distinguishable and bosonic particles. Fermionic statistics can be obtained by a reweigthing procedure by post processing the simulation (see ref. 2 below).

Main contributors: Yotam Feldman, Barak Hirshberg

Theory and Implementation:

Y.M.Y. Feldman and B. Hirshberg *"Quadratic scaling bosonic path integral molecular dynamics simulations"*, J. Chem. Phys. 159, 154107 (2023) DOI: 10.1063/5.0173749 — BibTeX: fetch

Theory:

B. Hirshberg, V. Rizzi and M. Parrinello, *"Path integral molecular dynamics for bosons"*, Proc. Natl. Acad. Sci. U.S.A. 116 (43) 21445-21449 (2019) DOI: 10.1073/pnas.1913365116 — BibTeX: fetch

B. Hirshberg, M. Invernizzi and M. Parrinello, *"Path integral molecular dynamics for fermions: Alleviating the sign problem with the Bogoliubov inequality"*, J. Chem. Phys. 152, 171102 (2020) DOI: 10.1063/5.0008720 — BibTeX: fetch

C. W. Myung, B. Hirshberg and M. Parrinello, *"Prediction of a Supersolid Phase in High-Pressure Deuterium"*, Phys. Rev. Lett., 128 045301 (2022) DOI: 10.1103/PhysRevLett.128.045301 — BibTeX: fetch

3.2 Cavity Molecular Dynamics for Polaritonics

This initial implementation provides an efficient cavity molecular dynamics (CavMD) scheme for simulating strong light-matter interactions between molecules and an optical cavity mode, particularly in the vibrational strong coupling regime. At present, the nuclear partial charges are assumed to be fixed during the simulation. CavMD is implemented with a new force evaluator: *ffcavphsocket*, which is operated similarly to the original *ffsocket* evaluator, but with additional parameters for controlling cavity photons. Hence, with this implementation, users can study different aspects of vibrational strong coupling with many sophisticated methods supported in i-pi.

Main contributors: Tao E. Li

Implementation and theory:

T. E. Li, J. E. Subotnik, and A. Nitzan, *"Cavity molecular dynamics simulations of liquid water under vibrational ultrastrong coupling"*, Proc. Natl. Acad. Sci. 117(31), 18324–18331. (2020)

DOI: 10.1073/pnas.2009272117 — BibTeX: fetch

T. E. Li, A. Nitzan, S. Hammes-Schiffer, and J. E. Subotnik, *"Quantum simulations of vibrational strong coupling via path integrals"*, J. Phys. Chem. Lett. 13(17), 3890–3895. (2022)

DOI: 10.1021/acs.jpclett.2c00613 — BibTeX: fetch

3.3 Committee models

Uncertainty estimation for machine-learning potentials based on committee models. Multiple potentials are used simultaneously (they should have been fitted with randomized training sets). The mean is used to drive the system, the spread is used as an estimate of the uncertainty. This uncertainty can be used to select high-error structures for active learning, to estimate the propagation of the error to thermodynamic averages, or to build a weighted baseline model that falls back to a safe baseline potential when the ML models fail.

Main contributors: Giulio Imbalzano, Venkat Kapil, Yongbin Zhuang, Federico Grasselli, Michele Ceriotti

Implementation and theory:

G. Imbalzano, Y. Zhuang, V. Kapil, K. Rossi, E. A. Engel, F. Grasselli, and M. Ceriotti, *"Uncertainty estimation for molecular dynamics and sampling"*, J. Chem. Phys. 154(7), 074102 (2021)

DOI: 10.1063/5.0036522 — BibTeX: fetch

F. Musil, M. J. Willatt, M. A. Langovoy, and M. Ceriotti, *"Fast and Accurate Uncertainty Estimation in Chemical Machine Learning"*, Journal of Chemical Theory and Computation 15(2), 906–915 (2019)

DOI: 10.1021/acs.jctc.8b00959 — BibTeX: fetch

3.4 Direct Estimators for Isotope Fractionation

A direct estimator to evaluate the isotope fractionation ratios using a single operation (and a single keyword in the input file), without the need for a thermodynamic integration with respect to the mass of the isotope.

Main contributors: Bingqing Cheng, Michele Ceriotti

Implementation and Theory:

B.Cheng, M.Ceriotti, "Direct path integral estimators for isotope fractionation ratios." The Journal of chemical physics 141, 244112 (2015)

DOI: 10.1063/1.4904293 — BibTeX: fetch

3.5 Fast-Forward Langevin Thermostat

This is a modified form of Langevin dynamics in which sluggish high-friction behaviour is corrected for by flipping a particle's momentum when the action of the thermostat causes it to change direction.

Main contributors: Mahdi Hijazi, David Wilkins, Michele Ceriotti

Implementation and Theory:

M. Hijazi, D. M. Wilkins, *"Fast-forward Langevin dynamics with momentum flips"*, J. Chem. Phys. 148, 184109 (2018)

DOI: 10.1063/1.5029833 — BibTeX: fetch

3.6 Finite-differences Suzuki-Chin PIMD

Suzuki-Chin PIMD gives better convergence w.r.t. the number of imaginary time slices as compared to the standard Trotter scheme. The implementation uses a symplectic and time-reversible finite-difference algorithm to compute high order corrections to traditional PIMD for any empirical or ab initio forcefield.

Main contributors: Venkat Kapil, Michele Ceriotti

Implementation and Theory:

V.Kapil, J.Behler, M.Ceriotti *"High order path interals made easy"*, J. Chem. Phys. 145, 234103 (2016)

DOI: 10.1063/1.4971438 — BIBTEX: fetch

Theory:

S.Jang, S.Jang, G.A.Voth *"Applications of higher order composite factorization schemes in imaginary time path integral simulations"*, J. Chem. Phys. 115, 7832 (2001)

DOI: 10.1063/1.1410117 — BIBTEX: fetch

S.A.Chin *"Symplectic integrators from composite operator factorizations"*, Phys. Lett. A 226, 344 (1997)

DOI: 10.1016/s0375-9601(97)00003-0 — BIBTEX: fetch

M.Suzuki *"Hybrid exponential product formulas for unbounded operators with possible applications to Monte Carlo simulations"*, Phys. Lett. A 201, 425 (1995)

DOI: 10.1016/0375-9601(95)00266-6 — BIBTEX: fetch

3.7 Finite-differences Vibrational Analysis

Harmonic vibrations through finite differences for simple evaluation of the harmonic Hessian.

Main contributors: Kapil, Bienvenue

Implementation:

M. Rossi, P. Gasparotto, M. Ceriotti, *"Anharmonic and Quantum Fluctuations in Molecular Crystals: A First-Principles Study of the Stability of Paracetamol"*, Phs. Rev. Lett. 117, 115702 (2016)

DOI: 10.1103/PhysRevLett.117.115702 — BibTeX: fetch

3.8 Free-energy Perturbation Estimators for Isotope Fractionation

Computing isotope fractionation using the thermodynamic integration method requires evaluating the quantum kinetic energy of several systems containing atoms that have different fictitious masses between the physical masses of two isotopes, meaning that a number of PIMD simulations have to be performed. With the help of re-weighting, one has the option of running just one set of simulation with a certain fictitious mass, and obtain the quantum kinetic energy for systems with other masses.

Main contributors: Michele Ceriotti, Thomas Markland

Theory and implementation:

Michele Ceriotti, Thomas E. Markland, "Efficient methods and practical guidelines for simulating isotope effects." The Journal of chemical physics 138(1), 014112 (2013).

DOI: 10.1063/1.4772676 — BibTeX: fetch

3.9 Generalized Langevin Equation Thermostats

The Generalized Langevin Equation provides a very flexible framework to manipulate the dynamics of a classical system, improving sampling efficiency and obtaining quasi-equilibrium ensembles that mimic quantum fluctuations. Parameters for the different modes of operation can be obtained from the GLE4MD website.

Main contributors: Michele Ceriotti

Implementation:

M. Ceriotti, G. Bussi, M. Parrinello, *"M. Colored-Noise Thermostats à la Carte"*, J. Chem. Theory Comput. 6, 1170–1180 (2010)

DOI: 10.1021/ct900563s — BibTeX: fetch

Theory:

Optimal Sampling Efficiency — M. Ceriotti, G. Bussi, and M. Parrinello, *"Langevin Equation with Colored Noise for Constant-Temperature Molecular Dynamics Simulations"*, Phys. Rev. Lett. 102, 20601 (2009)

DOI: 10.1103/PhysRevLett.102.020601 — BibTeX: fetch

Quantum Thermostat — M. Ceriotti, G. Bussi, and M. Parrinello, *"Nuclear Quantum Effects in Solids Using a Colored-Noise Thermostat"*, Phys. Rev. Lett. 103, 30603 (2009)

DOI: 10.1103/PhysRevLett.103.030603 — BibTeX: fetch

Delta Thermostat — M. Ceriotti and M. Parrinello, *"The δ-Thermostat: Selective Normal-Modes Excitation by Colored-Noise Langevin Dynamics"*, Procedia Comput. Sci. 1, 1607 (2010)

DOI: 10.1016/j.procs.2010.04.180 — BibTeX: fetch

MTS Thermostat — J. A. Morrone, T. E. Markland, M. Ceriotti, and B. J. Berne, *"Efficient Multiple Time Scale Molecular Dynamics: Using Colored Noise Thermostats to Stabilize Resonances"*, J. Chem. Phys. 134, 14103 (2011)

DOI: 10.1063/1.3518369 — BibTeX: fetch

"Hot-spot" — R. Dettori, M. Ceriotti, J. Hunger, C. Melis, L. Colombo, and D. Donadio, *"Simulating Energy Relaxation in Pump-Probe Vibrational Spectroscopy of Hydrogen-Bonded Liquids"*, J. Chem. Theory Comput. (2017)

DOI: 10.1021/acs.jctc.6b01108 — BibTeX: fetch

3.10 Geometry Optimization

Several standard algorithms for geometry optimization have been implemented to give the convenience of static calculations that are fully compatible with (PI)MD and other advanced sampling techniques.

Main contributors: Benjamin Helfrecht, Sophie Mutzel, Riccardo Petraglia, Yair Litman, Mariana Rossi

Implementation:

M. Rossi, P. Gasparotto, and M. Ceriotti, *"Anharmonic and Quantum Fluctuations in Molecular Crystals: A First-Principles Study of the Stability of Paracetamol"*, Phys. Rev. Lett. 117, 115702 (2016)

DOI: 10.1103/PhysRevLett.117.115702 — BibTeX: fetch

Theory:

W. H. Press, *"Numerical Recipes: The Art of Scientific Computing"*, (Cambridge University Press, 2007)

3.11 Langevin Sampling for Noisy or Dissipative Forces

A modified Langevin thermostat that allows for constant-temperature dynamics with noisy or dissipative forces by applying additional damping or noise for compensation. The implementation contains a method to adjust the amount of compensation automatically.

Main contributors: Jan Kessler, Thomas D. Kühne

Theory:

T. D. Kühne, M. Krack, F. R. Mohamed, M. Parrinello, *"Efficient and Accurate Car-Parrinello-like Approach to Born-Oppenheimer Molecular Dynamics"*, Phys. Rev. Lett. 98, 066401 (2007)

DOI: 10.1103/PhysRevLett.98.066401 — BibTeX: fetch

F. R. Krajewski, M. Parrinello, *"Linear scaling electronic structure calculations and accurate statistical mechanics sampling with noisy forces"*, Phys. Rev. B 73, 041105 (2006)

DOI: 10.1103/PhysRevB.73.041105 — BibTeX: fetch

Y. Luo, A. Zen, S. Sorella, *"Ab initio molecular dynamics with noisy forces: Validating the quantum Monte Carlo approach with benchmark calculations of molecular vibrational properties"*, J. Chem. Phys. 141, 194112 (2014)

DOI: 10.1063/1.4901430 — BibTeX: fetch

3.12 Multiple Time Step integrators

A multiple time step integration scheme allows for integration of different components of forces with different time steps. It becomes advantageous when the total force can be decomposed into a slowly varying expensive part and a rapidly varying cheap part. A larger time step can be used to integrate the former, there by reducing the number of expensive computations.

Main contributors: Venkat Kapil

Implementation:

V.Kapil, J.VandeVondele, M.Ceriotti *"Accurate molecular dynamics and nuclear quantum effects at low cost by multiple steps in real and imaginary time: using density functional theory to accelerate wavefunction methods"*, J. Chem. Phys. 144, 054111 (2016)

DOI: 10.1063/1.4941091 — BibTeX: fetch

Theory:

M.Tuckerman, B.J.Berne *"Reversible multiple time scale molecular dynamics"*, J. Chem. Phys. 97, 1990 (1992)

DOI: 10.1063/1.463137 — BibTeX: fetch

3.13 Open Path Integrals

Open path integrals and momentum distribution estimators for the computation of the particle momentum distribution including quantum fluctuations of nuclei.

Main contributors: Kapil, Cuzzocrea, Ceriotti

Implementation and Theory:

V. Kapil, A. Cuzzocrea, M. Ceriotti, *"Anisotropy of the Proton Momentum Distribution in Water"*, J. Phys. Chem. B 122, 6048-6054 (2018)

DOI: 10.1021/acs.jpcb.8b03896 — BibTeX: fetch

Theory:

J. A. Morrone, R. Car, *"Nuclear Quantum Effects in Water"*, Phys. Rev. Lett. 101, 017801 (2008)
DOI: 10.1103/PhysRevLett.101.017801 — BibTeX: fetch

3.14 Path Integral GLEs

Generalized Langevin Equations can be combined with a PIMD framework to accelerate convergence of quantum observables while retaining systematic approach to the quantum limit. Parameters formatted for i-PI input can be obtained from the GLE4MD website.

Main contributors: Michele Ceriotti, Joshua More
Implementation:
M. Ceriotti, J. More, D. Manolopoulos, *"i-PI: A Python interface for ab initio path integral molecular dynamics simulations"*, Comp. Phys. Comm. 185(3), 1019 (2014)
DOI: 10.1016/j.cpc.2013.10.027 — BibTeX: fetch
Theory:
PIGLET — M. Ceriotti and D. E. Manolopoulos, *"Efficient First-Principles Calculation of the Quantum Kinetic Energy and Momentum Distribution of Nuclei"*, Phys. Rev. Lett. 109, 100604 (2012)
DOI: 10.1103/PhysRevLett.109.100604 — BibTeX: fetch
PI+GLE — M. Ceriotti, D. E. Manolopoulos, and M. Parrinello, *"Accelerating the Convergence of Path Integral Dynamics with a Generalized Langevin Equation"*, J. Chem. Phys. 134, 84104 (2011)
DOI: 10.1063/1.3556661 — BibTeX: fetch

3.15 Path Integral Molecular Dynamics

The basic PIMD implementation in i-PI relies on a normal-modes integrator, and allows setting non-physical masses, so that both RPMD and CMD can be easily realized.

Main contributors: Michele Ceriotti, Joshua More
Implementation:
M. Ceriotti, J. More, D. Manolopoulos, *"i-PI: A Python interface for ab initio path integral molecular dynamics simulations"*, Comp. Phys. Comm. 185(3), 1019 (2014) DOI: 10.1016/j.cpc.2013.10.027 — BibTeX: fetch
Theory:
R. Feynman, A. Hibbs, *"Quantum Mechanics and Path Integrals"*, McGraw-Hill (1964)
M. Tuckerman, *"Statistical Mechanics and Molecular Simulations"*, Oxford Univ. Press (2008)

3.16 Path Integrals Molecular Dynamics at Constant Pressure

The constant-pressure implementation allows for arbitrary thermostats to be applied to the cell degrees of freedom, and work in both constant-shape and variable-cell mode.

Main contributors: Michele Ceriotti, Joshua More, Mariana Rossi
Implementation:
M. Ceriotti, J. More, D. Manolopoulos, *"i-PI: A Python interface for ab initio path integral molecular dynamics simulations"*, Comp. Phys. Comm. 185(3), 1019 (2014)
DOI: 10.1016/j.cpc.2013.10.027 — BibTeX: fetch

Theory:

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

DOI: 10.1063/1.478193 — BibTeX: fetch

G. Bussi, T. Zykova-Timan, M. Parrinello, *"Isothermal-isobaric molecular dynamics using stochastic velocity rescaling"*, J. Chem. Phys. 130(7), 074101 (2009)

DOI: 10.1063/1.3073889 — BibTeX: fetch

P. Raiteri, J. D. Gale, G. Bussi, *"Reactive force field simulation of proton diffusion in BaZrO3 using an empirical valence bond approach"*, J. Phys. Cond. Matt. 23(33), 334213 (2011)

DOI: 10.1088/0953-8984/23/33/334213 — BibTeX: fetch

3.17 Path Integral Langevin Equation Thermostats

Simple yet efficient Langevin thermostat for PIMD, with normal-modes thermostats optimally coupled to the ideal ring polymer frequencies

Main contributors: Michele Ceriotti

Implementation and Theory:

M. Ceriotti, M. Parrinello, T. E. Markland, and D. E. Manolopoulos, *"Efficient stochastic thermostatting of path integral molecular dynamics"* J. Chem. Phys. 133, 124104 (2010).

DOI: 10.1063/1.3489925 — BibTeX: fetch

3.18 Perturbed Path Integrals

Effectively a zeroth-order cumulant expansion of the high-order PI Hamiltonian, perturbed path integrals offer an attractive approach to compute thermochemistry of materials and molecules including quantum nuclei, as a post-processing of a Trotter trajectory.

Main contributors: Igor Poltavski

Theory:

I. Poltavsky and A. Tkatchenko, *"Modeling Quantum Nuclei with Perturbed Path Integral Molecular Dynamics"*, Chem. Sci. 7, 1368 (2016)

DOI: 10.1039/C5SC03443D — BibTeX: fetch

3.19 Replica Exchange MD

Accelerated convergence of averages by performing Monte Carlo exchanges of configurations between parallel calculations.

Main contributors: Riccardo Petraglia, Robert Meissner, Michele Ceriotti

Implementation: R. Petraglia, A. Nicolaï, M. M. D. Wodrich, M. Ceriotti, C. Corminboeuf, *"Beyond static structures: Putting forth remd as a tool to solve problems in computational organic chemistry"*, J. Comput. Chem. 37(1), 83-92 (2016)

DOI: 10.1002/jcc.24025 — BibTeX: fetch

Theory:

Y. Sugita, Y. Okamoto, *"Replica-exchange molecular dynamics method for protein folding"*, Chem. Phys. Lett. 314(1-2), 141–151 (1999)

DOI: 10.1016/s0009-2614(99)01123-9 — BibTeX: fetch

T. Okabe, M. Kawata, Y. Okamoto, M. Mikami, *"Replica-exchange monte carlo method for the isobaric–isothermal ensemble"*, Chem. Phys. Lett. 335(5-6), 435-439 (2001)

DOI: 10.1016/ s0009-2614(01)00055-0 — BibTeX: fetch

3.20 Quantum Alchemical Transformation

An algorithm that performs Monte Carlo moves to change a chemical species into its isotopes.

Main contributors: Bingqing Cheng, Michele Ceriotti

Implementation:

Cheng, Bingqing, J"{o}rg Behler, Michele Ceriotti, *"Nuclear Quantum Effects in Water at the Triple Point: Using Theory as a Link Between Experiments."* J. Phys. Chem. Lett. 7(12), 2210-2215 (2016)

DOI: 10.1021/acs.jpclett.6b00729 — BibTeX: fetch

Theory:

Michael R. Shirts, David L. Mobley, John D. Chodera, *"Alchemical Free Energy Calculations: Ready for Prime Time?"*, Ann. Rep. Comp. Chem. 41-59 (2007)

DOI: 10.1016/S1574-1400(07)03004-6 — BibTeX: fetch

Jian Liu, Richard S Andino, Christina M Miller, Xin Chen, David M Wilkins, Michele Ceriotti, David E Manolopoulos, *"A surface-specific isotope effect in mixtures of light and heavy water"*, J. Phys. Chem. C 117(6), 2944-2951 (2013)

DOI: 10.1021/jp311986m — BibTeX: fetch

3.21 Reweighting-based high-order PIMD

The Boltzmann weight assciated with the high order correction to standard PIMD is printed out as a property so that the high order estimate of an arbitrary position-dependent observable can be computed as a weighted average.

Main contributors: Michele Ceriotti , Guy A. R. Brian

Implementation:

M.Ceriotti, G.A.R.Brian, O.Riordan, D.E.Manolopolous *"The inefficiency of re-weighted sampling and the curse of system size in high-order path integration"*, Proc. R. Soc. A 468, 2-17 (2011)

DOI: 10.1098/rspa.2011.0413 — BIBTEX: fetch

Theory:

S.Jang, S.Jang, G.A.Voth *"Applications of higher order composite factorization schemes in imaginary time path integral simulations"*, J. Chem. Phys. 115, 7832 (2001)

DOI: 10.1063/1.1410117 — BIBTEX: fetch

S.A.Chin *"Symplectic integrators from composite operator factorizations"*, Phys. Lett. A 226, 344 (1997)

DOI: 10.1016/s0375-9601(97)00003-0 — BIBTEX: fetch

M.Suzuki *"Hybrid exponential product formulas for unbounded operators with possible applications to Monte Carlo simulations"*, Phys. Lett. A 201, 425 (1995)

DOI: 10.1016/0375-9601(95)00266-6 — BIBTEX: fetch

3.22 Ring-Polymer Contraction

A ring-polymer contraction makes it possible to compute different components of the forces on different number of imaginary time slices. In order to reap maximum benefits, the implementation is fully compatible with the multiple time step integrators.

Main contributors: Michele Ceriotti, Venkat Kapil

Implementation:

V.Kapil, J.VandeVondele, M.Ceriotti *"Accurate molecular dynamics and nuclear quantum effects at low cost by multiple steps in real and imaginary time: using density functional theory to accelerate wavefunction methods"*, J. Chem. Phys. 144, 054111 (2016) DOI: 10.1063/1.4941091 — BibTeX: fetch

Theory:

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

DOI: 10.1063/1.2953308 — BibTeX: fetch

3.23 Ring-polymer Instantons

Semiclassical instanton theory is an efficient way of simulating tunneling contributions to reaction rate constants and tunneling splittings, based on a well-defined dominant tunneling pathway. It can be much more efficient than RPMD rate theory, but it is not applicable to condensed phases and includes anharmonicities only along the reaction coordinate.

Main contributors: Yair Litman, Jeremy O. Richardson, Mariana Rossi

Implementation:

Y. Litman, J. O. Richardson, T. Kumagai, M. Rossi, *Elucidating the Quantum Dynamics of Intramolecular Double Hydrogen Transfer in Porphycene*, arXiv:1810.05681 (2018).

V. Kapil et al. *i-PI 2.0: A Universal Force Engine for AdvancedMolecular Simulations*, Comp. Phys. Comm. (2018)

Theory:

W. H. Miller, *Semiclassical limit of quantum mechanical transition state theory for nonseparable systems*, J. Chem. Phys. 62(5) 1899–1906 (1975)

DOI: 10.1063/1.430676 — BibTeX: fetch

J. O. Richardson, *Ring-polymer instanton theory*, Int. Rev. Phys. Chem. 37, 171 (2018)

DOI: 10.1080/0144235X.2018.1472353 — BibTeX: fetch

3.24 Thermodynamic Integrations

Thermodynamic integrations are made easy with i-PI, through the connection of different sockets and the different weights one can assign to different forces. It is possible to do harmonic (Debye model) to anharmonic integration, or integrations between different kinds of potentials. Also, quantum thermodynamic integrations relative to mass are easily done through the manual input of each atom's masses.

Main contributors: Mariana Rossi, Michele Ceriotti

Implementation:

M. Rossi, P. Gasparotto, M.Ceriotti *"Anharmonic and Quantum Fluctuations in Molecular Crystals: A First-Principles Study of the Stability of Paracetamol"*, Phys. Rev. Lett. 117, 115702 (2016)

DOI: 10.1103/PhysRevLett.117.115702 — BIBTEX: fetch

3.25 Thermostatted Ring-polymer Molecular Dynamics

By introducing an internal mode thermostat to RPMD it is possible to reduce the well-known artifacts in the simulation of dynamical properties by path integral methods.

Main contributors: Mariana Rossi, Michele Ceriotti

Implementation:

M.Rossi, M.Ceriotti, D.E.Manolopoulos, *"How to remove the spurious resonances from ring polymer molecular dynamics"*, J. Chem. Phys. 140, 234116 (2014)

DOI: 10.1063/1.4883861 — BibTeX: fetch

Theory:

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

DOI: 10.1063/1.1777575 — BibTeX: fetch

4 Internal units and conventions

4.1 Everything is a.u.

All the units used internally by i-PI are atomic units, as given below. *All* of them, even the temperature. 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 *Overriding default units* and *Properties*.

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

Regarding the specification of these units in the i-PI input files, the user is able to specify units both in the i-PI input file or in the structure file. If the structure file is of the .xyz format, the units specifications should be present in the comment line. Examples of such inputs can be found in `examples/pes-regtest/io-units/` . The code then behaves in the following way, depending on the user's choice:

- If no units are specified in the input, i-PI tries to guess from the comment line of the structure file and it nothing is present, assumes atomic units, except for the cases discussed below.
- If units are specified in both input and structure file and they match, conversion happens just once. If they do not match, an error is raised and i-PI stops.

There are a few exceptions, concerning quantities that are not used to describe the atomic-scale system, or I/O that is constrained by existing standards:

- Quantities that define run-time execution options, such as the simulation wall-clock time limit (specified by the flag <total_time>) or the latency of the forcefield socket (specified by the parameter <latency>), have to be provided in seconds.
- The PDB standard defines units to be in angstrom, and so default units for PDB are angstrom.

- When using ASE format for I/O, the units are dictated by the ASE defaults. So for example, the ASE extended xyz format expects positions in angstroms. (See for example `examples/ase-io/`)

4.2 Lattice parameters

The unit cell is defined internally by i-PI with the lattice vectors stored in the columns - i.e. as a 3×3 array in which `h[i, j]` contains the i-th Cartesian component of the j-th lattice vector. Furthermore, the cell is constrained to be oriented with the first vector along the x axis, the second vector within the xy plane, and the third in an arbitrary position (so that the cell is stored internally as an upper-triangular matrix). This is also reflected in how the cell parameters should be provided in the input file: when given in an array format, the correct format is e.g. `<h units='angstrom'> [10, 1, 2, 0, 9, -1, 0, 0, 11] </h>`.

4.3 The socket flag `pbc=True` or `pbc=False`

Inside the `ffsocket` and almost all other socket-type blocks, there is a flag called `pbc`, which has generated a lot of debate regarding its meaning and regarding its behaviour (see *Input file tags*).

The purpose of this flag is to tell i-PI whether to wrap the positions before sending them to the client or not. It bears no consequence to the type of simulation you are performing (i.e., this is not about whether the simulation is periodic or not).

Before the release of i-PI v.3.0, the default of this flag was set to True. This choice was made especifically because of the LAMMPS client (see below), but for all other clients we had experience with, they either did not care whether this flag was true or false and worked fine regardless or they would break down completely or in corner cases `pbc=True`.

With the release of i-PI v3.0 we set the default of this flag to be *False*. This has improved the user experience in most cases. However, we do recommend using the latest version of LAMMPS with this new default, as we also had to update the LAMMPS client interface, which incidentally became a lot faster (because we avoid triggering the neighbor-list calculation at each step).

If you se issues with your LAMMPS calculation (instability, missing atoms), especially for older versions of LAMMPS, setting `pbc=True` in your socket should fix it.

5 Input files

5.1 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:

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 class structure can be constructed iteratively.

For example, suppose we want to generate a *NPT* ensemble at an external pressure of 10^{-7} atomic pressure units. This would be specified by the following input file:

```
<system> <ensemble> <pressure> 1e-7 </pressure> ... </ensemble> ...
</system>
```

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 at input tags.

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:

```
<system> <ensemble> <pressure units='pascal'> 3e6 </pressure> ...
</ensemble> ... </system>
```

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

```
<system> <ensemble> <pressure units='megapascal'> 3 </pressure> ...
</ensemble> ... </system>
```

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

5.2 Initialization section

The input file can contain a initializer 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.

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 initializer 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 initfile 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 initfile tag will only read the information that can be inferred from the given file type, so for an 'xyz' file that does not contain a cell, the cell parameters will not be initialized.

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 *Checkpoint files* for details). As well as being a valid input for i-PI, a checkpoint can also be used inside an initializer 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.

6 Input file tags

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 . 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.

6.1 al6xxx_kmc

Holds all the information for the KMC dynamics, such as timestep, rates and barriers that control it.

ATTRIBUTES

mode

The KMC algorithm to be used

dtype: string

options: ['rfkmc']

default: rfkmc

FIELDS

optimizer

Option for geometry optimization step

nstep

The number of optimization steps.

dtype: integer

default: 10

a0

FCC lattice parameter

dtype: float

dimension: length

default: 1.0

diffusion_barrier_al

Barrier for vacancy diffusion in pure Al.

dtype: float

dimension: energy

default: 0.01

diffusion_prefactor_al

Prefactor for vacancy diffusion in pure Al.

dtype: float

dimension: frequency

default: 2.4188843e-05

diffusion_barrier_mg

Barrier for vacancy-assisted diffusion of Mg.

dtype: float

dimension: energy

default: 0.0

diffusion_prefactor_mg

Prefactor for vacancy-assisted diffusion of Mg.

dtype: float

dimension: frequency

default: 0.0

diffusion_barrier_si

Barrier for vacancy-assisted diffusion of Si.

dtype: float

dimension: energy

default: 0.0

diffusion_prefactor_si

Prefactor for vacancy-assisted diffusion of Si.

dtype: float

dimension: frequency

default: 0.0

neval

The number of parallel force evaluators.

dtype: integer

default: 4

ncell

The number of repeat cells in each direction.

dtype: integer

default: 4

nvac

The number of vacancies.

dtype: integer

default: 4

nsi

The number of silicon atoms.

dtype: integer

default: 4

nmg

The number of magnesium atoms.

dtype: integer

default: 4

idx

The position of the atoms on the lattice, relative to the canonical ordering.

dtype: integer

default: [ ]

tottime

Total KMC time elapsed

dtype: float

dimension: time

default: 0.0

ecache_file

Filename for storing/loading energy cache

dtype: string

default:

qcache_file

Filename for storing/loading positions cache

dtype: string

default:

max_cache_len

Maximum cache length before oldest entry is deleted

dtype: integer

default: 1000

6.2 alchemy

Holds all the information for doing Monte Carlo alchemical exchange moves.

ATTRIBUTES

mode

dtype: string

options: ['dummy']

default: dummy

FIELDS

names

The names of the atoms to be to exchanged, in the format [name1, name2, ...].

dtype: string

default: [ ]

nxc

The average number of exchanges per step to be attempted

dtype: float

default: 1

ealc

The contribution to the conserved quantity for the alchemical exchanger

dtype: float

default: 0.0

6.3 atoms

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

FIELDS

natoms

The number of atoms.

dtype: integer

default: 0

q

The positions of the atoms, in the format [x1, y1, z1, x2, ...].

dtype: float

dimension: length

default: [ ]

p

The momenta of the atoms, in the format [px1, py1, pz1, px2, …].

dtype: float

dimension: momentum

default: [ ]

m

The masses of the atoms, in the format [m1, m2, …].

dtype: float

dimension: mass

default: [ ]

names

The names of the atoms, in the format [name1, name2, …].

dtype: string

default: [ ]

6.4 atomswap

Holds all the information for doing Monte Carlo atom swap moves.

ATTRIBUTES

mode

Dummy attribute, does nothing.

dtype: string

options: ['dummy']

default: dummy

FIELDS

names

The names of the atoms to be to exchanged, in the format [name1, name2, …].

dtype: string

default: [ ]

nxc

The average number of exchanges per step to be attempted

dtype: float

default: 1

ealc

The contribution to the conserved quantity for the atom swapper

dtype: float

default: 0.0

6.5 barostat

Simulates an external pressure bath.

ATTRIBUTES

mode

The type of barostat. 'isotropic' implements the Bussi-Zykova-Parrinello barostat [doi:10.1063/1.3073889] that isotropically scales the volume while sampling the isothermal isobaric ensemble. The implementation details are given in [doi:10.1016/j.cpc.2013.10.027]. This barostat is suitable for simulating liquids. 'sc-isotropic' implements the same for Suzuki-Chin path integral molecular dynamics [10.1021/acs.jctc.8b01297] and should only be used with the Suzuki-Chin NPT ensemble. 'flexible' implements the path integral version of the Martyna-Tuckerman-Tobias-Klein barostat which incorporates full cell fluctuations while sampling the isothermal isobaric ensemble [doi:10.1063/1.478193]. This is suitable for anisotropic systems such as molecular solids. 'anisotropic' implements the Raiteri-Gale-Bussi barostat which enables cell fluctuations at constant external stress [10.1088/0953-8984/23/33/334213]. It is suitable for simulating solids at given external (non-diagonal) stresses and requires specifying a reference cell for estimating strain. Note that this ensemble is valid only within the elastic limit of small strains. For diagonal stresses (or external pressures) the 'flexible' and the 'anisotropic' modes should give very similar results. 'dummy' barostat does not do anything.

dtype: string

options: ['dummy', 'isotropic', 'flexible', 'anisotropic', 'sc-isotropic']

default: dummy

FIELDS

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.

tau

The time constant associated with the dynamics of the piston.

dtype: float

dimension: time

default: 1.0

p

Momentum (or momenta) of the piston.

dtype: float

dimension: momentum

default: [ ]

h0

Reference cell for Parrinello-Rahman-like barostats. Should be roughly equal to the mean size of the cell averaged over the trajectory. Sampling might be inaccurate if the difference is too large.

dtype: float

dimension: length

***default*:**
[0. 0. 0. 0. 0. 0. 0. 0. 0.]

hfix

A list of the cell entries that should be held fixed (xx, yy, zz, xy, xz, yz). 'offdiagonal' is an alias for xy, xz, yz.

dtype: string

default: [ ]

vol_constraint

If True, the (N, V, sigma_a = 0, T)-ensemble is sampled, which allows for full cell fluctuations while keeping the cell volume fixed. This ensemble was introduced in [doi:10.1021/acs.jctc.5b00748]. Note that it is only implemented for the MTTK ('flexible') barostat, enabling it for any other barostat will not work and result in an error.

dtype: boolean

default: False

6.6 beads

Describes the bead configurations in a path integral simulation.

ATTRIBUTES

natoms

The number of atoms.

dtype: integer

default: 0

nbeads

The number of beads.

dtype: integer

default: 0

FIELDS

q

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

dtype: float

dimension: length

default: [ ]

p

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

dtype: float

dimension: momentum

default: [ ]

m

The masses of the atoms, in the format [m1, m2, ...].

dtype: float

dimension: mass

default: [ ]

names

The names of the atoms, in the format [name1, name2, ...].

dtype: string

default: [ ]

6.7 bec

Deals with the Born Effective Charges tensors

dtype: float

dimension: number

ATTRIBUTES

units

The units the input data is given in.

dtype: string

default: automatic

shape

The shape of the array.

dtype: tuple

default: (0,)

mode

If 'mode' is 'driver', then the array is computed on the fly. If 'mode' is 'manual', then the array is read in directly, then reshaped according to the 'shape' specified in a row-major manner. If 'mode' is 'file' then the array is read in from the file given.

dtype: string

options: ['driver', 'manual', 'file', 'none']

default: none

6.8 bias

Deals with creating all the necessary forcefield objects.

FIELDS

force

The class that deals with how each forcefield contributes to the overall potential, force and virial calculation.

6.9 bosons

Deals with the specification of which atoms are have bosonic indistinguishability. The specified atoms participate in exchange interaction, which forms ring polymers that combine several particles together. The algorithm scales quadratically with the number of atoms and linearly with the number of beads. The implementation is based on Hirshberg et al.'s doi:10.1073/pnas.1913365116 and Feldman and Hirshberg's doi:10.1063/5.0173749.

dtype: string

ATTRIBUTES

shape

The shape of the array.

dtype: tuple

default: (0,)

mode

If 'mode' is 'manual', then the array is read in directly, then reshaped according to the 'shape' specified in a row-major manner. If 'mode' is 'file' then the array is read in from the file given.

dtype: string

options: ['manual', 'file']

default: manual

id

If 'id' is 'index', then bosonic atoms are specified a list of indices (zero-based). If 'id' is 'label' then specify a list of labels.

dtype: string

options: ['index', 'label']

default: index

6.10 cell

Describes with the cell parameters. Takes as array which can be used to initialize the cell vector matrix. N.B.: the cell parameters are stored with the lattice vectors in the columns, and the cell must be oriented in such a way that the array is upper-triangular (i.e. with the first vector aligned along x and the second vector in the xy plane).

dtype: float

dimension: length

ATTRIBUTES

units

The units the input data is given in.

dtype: string

default: automatic

shape

The shape of the array.

dtype: tuple

default: (0,)

mode

If 'mode' is 'manual', then the array is read in directly, then reshaped according to the 'shape' specified in a row-major manner. If 'mode' is 'file' then the array is read in from the file given.

dtype: string

options: ['manual', 'file']

default: manual

6.11 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.

dtype: integer

ATTRIBUTES

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.

dtype: string

default: restart

stride

The number of steps between successive writes.

dtype: integer

default: 1

overwrite

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

dtype: boolean

default: True

6.12 constrained_dynamics

Holds all the information for the MD integrator, such as timestep, the thermostats and barostats that control it.

ATTRIBUTES

mode

The ensemble that will be sampled during the simulation.

dtype: string

options: ['nve', 'nvt']

default: nve

splitting

The integrator used for sampling the target ensemble.

dtype: string

options: ['obabo', 'baoab']

default: baoab

FIELDS

thermostat

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

barostat

Simulates an external pressure bath.

timestep

The time step.

dtype: float

dimension: time

default: 1.0

nmts

Number of iterations for each MTS level (including the outer loop, that should in most cases have just one iteration).

dtype: integer

default: [ ]

nsteps_o

The number of sub steps used in the evolution of the thermostat (used in function step_Oc). Relevant only for GLE thermostats

dtype: integer

default: 1

nsteps_geo

The number of sub steps used in the evolution of the geodesic flow (used in function step_Ag).

dtype: integer

default: 1

csolver

Define a numerical method for computing the projection operators associated with the constraint.

constraint

Define a constraint to be applied onto atoms

6.13 constraint

Generic input value

ATTRIBUTES

mode

The type of constraint.

dtype: string

options: ['distance', 'angle', 'eckart', 'multi', 'multi']

default: distance

FIELDS

atoms

List of atoms indices that are to be constrained.

dtype: integer

default: [ ]

values

List of constraint lengths.

dtype: float

dimension: length

default: [ ]

constraint

One or more constraints that have to be considered coupled

6.14 csolver

Holds all parameters for the numerical method used to solve the contraint.

FIELDS

tolerance

Tolerance value used in the Quasi-Newton iteration scheme.

dtype: float

default: 0.0001

maxit

Maximum number of steps used in the Quasi-Newton iteration scheme.

dtype: integer

default: 1000

norm_order

Order of norm used to determine termination of the Quasi-newton iteration.

dtype: integer

default: 2

6.15 dmd

DrivenMD with external time-dependent driving potential

FIELDS

dmdff

List of names of forcefields that should do driven MD. Accepts ffdmd forcefield types. Currently implemented (2021) for ffdmd only the driving potential similar to the one described in Bowman, .., Brown JCP 119, 646 (2003).

dtype: string

default: [ ]

6.16 driven_dynamics

Holds all the information for a driven dynamics.

ATTRIBUTES

mode

The ensemble that will be sampled during the simulation. eda-nve: nve with an external electric field; eda-nvt: nvt with an external electric field;

dtype: string

options: ['eda-nve', 'eda-nvt']

default: eda-nve

splitting

The Louiville splitting used for sampling the target ensemble.

dtype: string

options: ['obabo', 'baoab']

default: obabo

FIELDS

efield

The external electric field parameters:plane-wave parameters (intensity/amplitude, angular frequency, and phase) and gaussian envelope function parameters (peak time/mean of the gaussian, and pulse duration/standard deviation of the gaussian)

bec

The Born Effective Charges tensors (cartesian coordinates)

dtype: float

dimension: number

default: [ ]

thermostat

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

barostat

Simulates an external pressure bath.

timestep

The time step.

dtype: float

dimension: time

default: 1.0

nmts

Number of iterations for each MTS level (including the outer loop, that should in most cases have just one iteration).

dtype: integer

default: [ ]

6.17 dynamics

Holds all the information for the MD integrator, such as timestep, the thermostats and barostats that control it.

ATTRIBUTES

mode

The ensemble that will be sampled during the simulation. nve: constant-energy-volume; nvt: constant-temperature-volume; npt: constant-temperature-pressure(isotropic); nst: constant-temperature-stress(anisotropic); sc: Suzuki-Chin high-order NVT; scnpt: Suzuki-Chin high-order NpT; nvt-cc: constrained-centroid NVT;

dtype: string

options: ['nve', 'nvt', 'npt', 'nst', 'sc', 'scnpt', 'nvt-cc']

default: nve

splitting

The Louiville splitting used for sampling the target ensemble.

dtype: string

options: ['obabo', 'baoab']

default: obabo

FIELDS

thermostat

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

barostat

Simulates an external pressure bath.

timestep

The time step.

dtype: float

dimension: time

default: 1.0

nmts

Number of iterations for each MTS level (including the outer loop, that should in most cases have just one iteration).

dtype: integer

default: [ ]

6.18 efield

Simulates an external time dependent electric field

FIELDS

amp

The amplitude of the external electric field (in cartesian coordinates)

dtype: float

dimension: electric-field

***default*:**
[0. 0. 0.]

freq

The pulsation of the external electric field

dtype: float

dimension: frequency

default: 0.0

phase

The phase of the external electric field (in rad)

dtype: float

dimension: number

default: 0.0

peak

The time when the external electric field gets its maximum value

dtype: float

dimension: time

default: 0.0

sigma

The standard deviations (time) of the gaussian envelope function of the external electric field

dtype: float

dimension: time

default: inf

6.19 ensemble

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

FIELDS

temperature

The temperature of the system.

dtype: float

dimension: temperature

default: -1.0

pressure

The external pressure.

dtype: float

dimension: pressure

default: -12345.0

stress

The external stress.

dtype: float

dimension: pressure

***default*:**
[-12345. -0. -0. -0. -12345. -0. -0. -0. -12345.]

eens

The ensemble contribution to the conserved quantity.

dtype: float

dimension: energy

default: 0.0

bias

Deals with creating all the necessary forcefield objects.

bias_weights

Bias weights.

dtype: float

default: [ ]

hamiltonian_weights

Hamiltonian weights.

dtype: float

default: [ ]

time

The internal time for this system

dtype: float

dimension: time

default: 0.0

6.20 ffcavphsocket

A cavity molecular dynamics driver for vibraitonal strong coupling. In the current implementation, only a single cavity mode polarized along the x and y directions is coupled to the molecules. Check https://doi.org/10.1073/pnas.2009272117 and also examples/lammps/h2o-cavmd/ for details.

ATTRIBUTES

mode

Specifies whether the driver interface will listen onto a internet socket [inet] or onto a unix socket [unix].

dtype: string

options: ['unix', 'inet']

default: inet

matching

Specifies whether requests should be dispatched to any client, automatically matched to the same client when possible [auto] or strictly forced to match with the same client [lock].

dtype: string

options: ['auto', 'any', 'lock']

default: auto

name

Mandatory. The name by which the forcefield will be identified in the System forces section.

dtype: string

pbc

Applies periodic boundary conditions to the atoms coordinates before passing them on to the driver code.

dtype: boolean

default: False

threaded

Whether the forcefield should use a thread loop to evaluate, or work in serial. Should be set to True for FFSockets

dtype: boolean

default: True

FIELDS

charge_array

The partial charges of all the atoms, in the format [Q1, Q2, …].

dtype: float

dimension: length

default: [ ]

apply_photon

Determines if additional photonic degrees of freedom is included or not.

dtype: boolean

default: False

E0

The value of varepsilon (effective light-matter coupling strength) in atomic units.

dtype: float

default: 0.0

omega_c

This gives the cavity photon frequency at normal incidence.

dtype: float

dimension: frequency

default: 0.01

ph_rep

In the current implementation, two energy-degenerate photon modes polarized along x and y directions are coupled to the molecular system. If 'loose', the cavity photons polarized along the x, y directions are represented by two 'L' atoms; the x dimension of the first 'L' atom is coupled to the molecules, and the y dimension of the second 'L' atom is coupled to the molecules. If 'dense', the cavity photons polarized along the x, y directions are represented by one 'L' atom; the x and y dimensions of this 'L' atom are coupled to the molecules.

dtype: string

options: ['loose', 'dense']

default: loose

address

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

dtype: string

default: localhost

port

This gives the port number that defines the socket.

dtype: integer

default: 65535

slots

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

dtype: integer

default: 4

exit_on_disconnect

Determines if i-PI should quit when a client disconnects.

dtype: boolean

default: False

timeout

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

dtype: float

default: 0.0

latency

The number of seconds the polling thread will wait between exhamining the list of requests.

dtype: float

default: 0.0001

offset

A constant offset that is subtracted from the forcefield energy. Useful when there is a large core energy contribution that is constant throughout a simulation and hides significant changes in the 10th digit.

dtype: float

dimension: energy

default: 0.0

parameters

The parameters of the force field

dtype: dictionary

default: { }

activelist

List with indexes of the atoms that this socket is taking care of. Default: [-1] (corresponding to all)

dtype: integer

***default*:**
[-1]

6.21 ffcommittee

Combines multiple forcefields to build a committee model, that can be used to compute uncertainty-quantified machine-learning models. Each forcefield can be any of the other FF objects, and each should be used with a client that generates a slightly different estimation of energy and forces. These are averaged, and the mean used as the actual forcefield. Statistics about the distribution are also returned as extras fields, and can be printed for further postprocessing. It is also possible for a single FF object to return a JSON-formatted string containing entries *committee_pot*, *committee_force* and *committee_virial*, that contain multiple members at once. These will be unpacked and combined with whatever else is present. Also contains options to use it for uncertainty estimation and for active learning in a ML context, based on a committee model. Implements the approaches discussed in [Musil et al.](http://doi.org/10.1021/acs.jctc.8b00959) and [Imbalzano et al.](http://doi.org/10.1063/5.0036522)

ATTRIBUTES

name

Mandatory. The name by which the forcefield will be identified in the System forces section.

dtype: string

pbc

Applies periodic boundary conditions to the atoms coordinates before passing them on to the driver code.

dtype: boolean

default: False

threaded

Whether the forcefield should use a thread loop to evaluate, or work in serial

dtype: boolean

default: False

FIELDS

latency

The number of seconds the polling thread will wait between exhamining the list of requests.

dtype: float

default: 0.0001

offset

A constant offset that is subtracted from the forcefield energy. Useful when there is a large core energy contribution that is constant throughout a simulation and hides significant changes in the 10th digit.

dtype: float

dimension: energy

default: 0.0

parameters

The parameters of the force field

dtype: dictionary

default: { }

activelist

List with indexes of the atoms that this socket is taking care of. Default: [-1] (corresponding to all)

dtype: integer

***default*:**
[-1]

weights

List of weights to be given to the forcefields. Defaults to 1 for each FF. Note that the components are divided by the number of FF, and so the default corresponds to an average.

dtype: float

default: [ ]

alpha

Scaling of the variance of the model, corresponding to a calibration of the error

dtype: float

default: 1.0

baseline_name

Name of the forcefield object that should be treated as the baseline for a weighted baseline model.

dtype: string

default:

baseline_uncertainty

Corresponds to the expected error of the baseline model. This represents the error on the TOTAL potential energy of the simulation.

dtype: float

dimension: energy

default: -1.0

active_thresh

The uncertainty threshold for active learning. Structure with an uncertainty above this value are printed in the specified output file so they can be used for active learning.

dtype: float

dimension: energy

default: 0.0

active_output

Output filename for structures that exceed the accuracy threshold of the model, to be used in active learning.

dtype: string

default: active_output

parse_json

Tell the model whether to parse extras string looking for committee values of potential, forces, and virials. Default: false.

dtype: boolean

default: False

ffsocket

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

ffdirect

Direct potential that evaluates forces through a Python call, using PES providers from a list of possible external codes. The available PES interfaces are listed into the *ipi/pes* folder, and are the same available for the Python driver. The <*parameters*> field should contain a dictionary of the specific option of the chosen PES.

fflj

Simple, internal LJ evaluator without cutoff, neighbour lists or minimal image convention. Expects standard LJ parameters, e.g. { eps: 0.1, sigma: 1.0 }.

ffdebye

Harmonic energy calculator

ffplumed

Direct PLUMED interface. Can be used to implement metadynamics in i-PI in combination with the <metad> SMotion class. NB: if you use PLUMED for constant biasing (e.g. for umbrella sampling) the bias will be computed but there will be no output as specified in the *plumed.dat* file unless you include a <metad> tag, that triggers the log update.

ffyaff

Uses a Yaff force field to compute the forces.

ffsgdml

A SGDML energy calculator

6.22 ffdebye

Harmonic energy calculator

ATTRIBUTES

name

Mandatory. The name by which the forcefield will be identified in the System forces section.

dtype: string

pbc

Applies periodic boundary conditions to the atoms coordinates before passing them on to the driver code.

dtype: boolean

default: False

threaded

Whether the forcefield should use a thread loop to evaluate, or work in serial

dtype: boolean

default: False

FIELDS

hessian

Specifies the Hessian of the harmonic potential. Default units are atomic. Units can be specified only by xml attribute. Implemented options are: 'atomic_unit', 'ev/ang^2'

dtype: float

dimension: hessian

default: [ ]

x_reference

Minimum-energy configuration for the harmonic potential

dtype: float

dimension: length

default: [ ]

v_reference

Zero-value of energy for the harmonic potential

dtype: float

dimension: energy

default: 0.0

latency

The number of seconds the polling thread will wait between exhamining the list of requests.

dtype: float

default: 0.0001

offset

A constant offset that is subtracted from the forcefield energy. Useful when there is a large core energy contribution that is constant throughout a simulation and hides significant changes in the 10th digit.

dtype: float

dimension: energy

default: 0.0

parameters

The parameters of the force field

dtype: dictionary

default: { }

activelist

List with indexes of the atoms that this socket is taking care of. Default: [-1] (corresponding to all)

dtype: integer

***default*:**
[-1]

6.23 ffdirect

Direct potential that evaluates forces through a Python call, using PES providers from a list of possible external codes. The available PES interfaces are listed into the *ipi/pes* folder, and are the same available for the Python driver. The <*parameters*> field should contain a dictionary of the specific option of the chosen PES.

ATTRIBUTES

name

Mandatory. The name by which the forcefield will be identified in the System forces section.

dtype: string

pbc

Applies periodic boundary conditions to the atoms coordinates before passing them on to the driver code.

dtype: boolean

default: False

threaded

Whether the forcefield should use a thread loop to evaluate, or work in serial

dtype: boolean

default: False

FIELDS

pes

Type of PES that should be used to evaluate the forcefield

dtype: string

options: ['ase', 'bath', 'double_double_well', 'DW', 'DW_bath', 'DW_friction', 'driverdipole', 'dummy', 'elphmod', 'harmonic', 'mace', 'metatensor', 'metatomic', 'pet', 'rascal', 'so3lr', 'Spherical_LJ', 'spline', 'xtb']

default: dummy

latency

The number of seconds the polling thread will wait between exhamining the list of requests.

dtype: float

default: 0.0001

offset

A constant offset that is subtracted from the forcefield energy. Useful when there is a large core energy contribution that is constant throughout a simulation and hides significant changes in the 10th digit.

dtype: float

dimension: energy

default: 0.0

parameters

The parameters of the force field

dtype: dictionary

default: { }

activelist

List with indexes of the atoms that this socket is taking care of. Default: [-1] (corresponding to all)

dtype: integer

***default*:**
[-1]

6.24 ffdmd

Simple, internal DMD evaluator without without neighbor lists, but with PBC. Expects coupling elements (n*(n-1)/2 of them), oscillating frequency and time step.

ATTRIBUTES

name

Mandatory. The name by which the forcefield will be identified in the System forces section.

dtype: string

pbc

Applies periodic boundary conditions to the atoms coordinates before passing them on to the driver code.

dtype: boolean

default: False

threaded

Whether the forcefield should use a thread loop to evaluate, or work in serial

dtype: boolean

default: False

FIELDS

dmd_coupling

Specifies the coupling between atom pairs (should be size N*(N-1)/2 ordered c21, c32, c31, c43, c42, c41 etc. – in atomic units!)

dtype: float

default: [ ]

dmd_freq

Frequency of the oscillation of the time-dependent term

dtype: float

dimension: frequency

default: 0.0

dmd_dt

Time step of the oscillating potential. Should match time step of simulation

dtype: float

dimension: time

default: 0.0

dmd_step

The current step counter for dmd.

dtype: integer

default: 0

latency

The number of seconds the polling thread will wait between exhamining the list of requests.

dtype: float

default: 0.0001

offset

A constant offset that is subtracted from the forcefield energy. Useful when there is a large core energy contribution that is constant throughout a simulation and hides significant changes in the 10th digit.

dtype: float

dimension: energy

default: 0.0

parameters

The parameters of the force field

dtype: dictionary

default: { }

activelist

List with indexes of the atoms that this socket is taking care of. Default: [-1] (corresponding to all)

dtype: integer

***default*:**
[-1]

6.25 fflj

Simple, internal LJ evaluator without cutoff, neighbour lists or minimal image convention. Expects standard LJ parameters, e.g. { eps: 0.1, sigma: 1.0 }.

ATTRIBUTES

name

Mandatory. The name by which the forcefield will be identified in the System forces section.

dtype: string

pbc

Applies periodic boundary conditions to the atoms coordinates before passing them on to the driver code.

dtype: boolean

default: False

threaded

Whether the forcefield should use a thread loop to evaluate, or work in serial

dtype: boolean

default: False

FIELDS

latency

The number of seconds the polling thread will wait between exhamining the list of requests.

dtype: float

default: 0.0001

offset

A constant offset that is subtracted from the forcefield energy. Useful when there is a large core energy contribution that is constant throughout a simulation and hides significant changes in the 10th digit.

dtype: float

dimension: energy

default: 0.0

parameters

The parameters of the force field

dtype: dictionary

default: { }

activelist

List with indexes of the atoms that this socket is taking care of. Default: [-1] (corresponding to all)

dtype: integer

***default*:**
[-1]

6.26 ffplumed

Direct PLUMED interface. Can be used to implement metadynamics in i-PI in combination with the <metad> SMotion class. NB: if you use PLUMED for constant biasing (e.g. for umbrella sampling) the bias will be computed but there will be no output as specified in the *plumed.dat* file unless you include a <metad> tag, that triggers the log update.

ATTRIBUTES

name

Mandatory. The name by which the forcefield will be identified in the System forces section.

dtype: string

pbc

Applies periodic boundary conditions to the atoms coordinates before passing them on to the driver code.

dtype: boolean

default: False

threaded

Whether the forcefield should use a thread loop to evaluate, or work in serial

dtype: boolean

default: False

FIELDS

file

This describes the location to read the reference structure file from.

dtype: string

default:

plumed_dat

The PLUMED input file

dtype: string

default: plumed.dat

plumed_step

The current step counter for PLUMED calls

dtype: integer

default: 0

compute_work

Compute the work done by the metadynamics bias (to correct the conserved quantity). Note that this might require multiple evaluations of the conserved quantities, and can add some overhead.

dtype: boolean

default: True

plumed_extras

List of variables defined in the PLUMED input, that should be transferred to i-PI as *extras* fields. Note that a version of PLUMED greater or equal than 2.10 is necessary to retrieve variables into i-PI, and that, if you are using ring-polymer contraction, the associated force block should use *interpolate_extras* to make sure all beads have values.

dtype: string

default: [ ]

latency

The number of seconds the polling thread will wait between exhamining the list of requests.

dtype: float

default: 0.0001

offset

A constant offset that is subtracted from the forcefield energy. Useful when there is a large core energy contribution that is constant throughout a simulation and hides significant changes in the 10th digit.

dtype: float

dimension: energy

default: 0.0

parameters

The parameters of the force field

dtype: dictionary

default: { }

activelist

List with indexes of the atoms that this socket is taking care of. Default: [-1] (corresponding to all)

dtype: integer

***default*:**
[-1]

6.27 ffsgdml

A SGDML energy calculator

ATTRIBUTES

name

Mandatory. The name by which the forcefield will be identified in the System forces section.

dtype: string

pbc

Applies periodic boundary conditions to the atoms coordinates before passing them on to the driver code.

dtype: boolean

default: False

threaded

Whether the forcefield should use a thread loop to evaluate, or work in serial

dtype: boolean

default: False

FIELDS

sGDML_model

This gives the file name of the sGDML model.

dtype: string

latency

The number of seconds the polling thread will wait between exhamining the list of requests.

dtype: float

default: 0.0001

offset

A constant offset that is subtracted from the forcefield energy. Useful when there is a large core energy contribution that is constant throughout a simulation and hides significant changes in the 10th digit.

dtype: float

dimension: energy

default: 0.0

parameters

The parameters of the force field

dtype: dictionary

default: { }

activelist

List with indexes of the atoms that this socket is taking care of. Default: [-1] (corresponding to all)

dtype: integer

***default*:**
[-1]

6.28 ffsocket

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

ATTRIBUTES

mode

Specifies whether the driver interface will listen onto a internet socket [inet] or onto a unix socket [unix].

dtype: string

options: ['unix', 'inet']

default: inet

matching

Specifies whether requests should be dispatched to any client, automatically matched to the same client when possible [auto] or strictly forced to match with the same client [lock].

dtype: string

options: ['auto', 'any', 'lock']

default: auto

name

Mandatory. The name by which the forcefield will be identified in the System forces section.

dtype: string

pbc

Applies periodic boundary conditions to the atoms coordinates before passing them on to the driver code.

dtype: boolean

default: False

threaded

Whether the forcefield should use a thread loop to evaluate, or work in serial. Should be set to True for FFSockets

dtype: boolean

default: True

FIELDS

address

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

dtype: string

default: localhost

port

This gives the port number that defines the socket.

dtype: integer

default: 65535

slots

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

dtype: integer

default: 4

exit_on_disconnect

Determines if i-PI should quit when a client disconnects.

dtype: boolean

default: False

timeout

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

dtype: float

default: 0.0

latency

The number of seconds the polling thread will wait between exhamining the list of requests.

dtype: float

default: 0.0001

offset

A constant offset that is subtracted from the forcefield energy. Useful when there is a large core energy contribution that is constant throughout a simulation and hides significant changes in the 10th digit.

dtype: float

dimension: energy

default: 0.0

parameters

The parameters of the force field

dtype: dictionary

default: { }

activelist

List with indexes of the atoms that this socket is taking care of. Default: [-1] (corresponding to all)

dtype: integer

***default*:**
[-1]

6.29 ffyaff

Uses a Yaff force field to compute the forces.

ATTRIBUTES

name

Mandatory. The name by which the forcefield will be identified in the System forces section.

dtype: string

pbc

Applies periodic boundary conditions to the atoms coordinates before passing them on to the driver code.

dtype: boolean

default: False

threaded

Whether the forcefield should use a thread loop to evaluate, or work in serial

dtype: boolean

default: False

FIELDS

yaffpara

This gives the file name of the Yaff input parameter file.

dtype: string

default: parameters.txt

yaffsys

This gives the file name of the Yaff input system file.

dtype: string

default: system.chk

yafflog

This gives the file name of the Yaff output log file.

dtype: string

default: yaff.log

rcut

This gives the real space cutoff used by all pair potentials in atomic units.

dtype: float

default: 18.89726133921252

alpha_scale

This gives the alpha parameter in the Ewald summation based on the real-space cutoff: alpha = alpha_scale / rcut. Higher values for this parameter imply a faster convergence of the reciprocal terms, but a slower convergence in real-space.

dtype: float

default: 3.5

gcut_scale

This gives the reciprocale space cutoff based on the alpha parameter: gcut = gcut_scale * alpha. Higher values for this parameter imply a better convergence in the reciprocal space.

dtype: float

default: 1.1

skin

This gives the skin parameter for the neighborlist.

dtype: integer

default: 0

smooth_ei

This gives the flag for smooth truncations for the electrostatic interactions.

dtype: boolean

default: False

reci_ei

This gives the method to be used for the reciprocal contribution to the electrostatic interactions in the case of periodic systems. This must be one of 'ignore' or 'ewald'. The 'ewald' option is only supported for 3D periodic systems.

dtype: string

default: ewald

latency

The number of seconds the polling thread will wait between exhamining the list of requests.

dtype: float

default: 0.0001

offset

A constant offset that is subtracted from the forcefield energy. Useful when there is a large core energy contribution that is constant throughout a simulation and hides significant changes in the 10th digit.

dtype: float

dimension: energy

default: 0.0

parameters

The parameters of the force field

dtype: dictionary

default: { }

activelist

List with indexes of the atoms that this socket is taking care of. Default: [-1] (corresponding to all)

dtype: integer

***default*:**
[-1]

6.30 file

This is the class to initialize from file.

dtype: string

ATTRIBUTES

mode

The input data format. 'xyz' and 'pdb' stand for xyz and pdb input files respectively. 'chk' stands for initialization from a checkpoint file. 'ase' is to read a file with the Atomic Simulation Environment

dtype: string

options: ['xyz', 'pdb', 'chk', 'ase']

default: chk

bead

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

dtype: integer

default: -1

cell_units

The units for the cell dimensions.

dtype: string

default: automatic

6.31 force

The class that deals with how each forcefield contributes to the overall potential, force and virial calculation.

ATTRIBUTES

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.

dtype: integer

default: 0

weight

A scaling factor for this forcefield, to be applied before adding the force calculated by this forcefield to the total force.

dtype: float

default: 1.0

fd_epsilon

The finite displacement to be used for calculaing the Suzuki-Chin contribution of the force. If the value is negative, a centered finite-difference scheme will be used. [in bohr]

dtype: float

default: -0.001

name

An optional name to refer to this force component.

dtype: string

default:

forcefield

Mandatory. The name of the forcefield this force is referring to.

dtype: string

default:

FIELDS

mts_weights

The weight of force in each mts level starting from outer.

dtype: float

dimension: force

***default*:**
[1.]

interpolate_extras

A list of quantities that should be extracted from the 'extra' string returned by the forcefield, that are then treated the same way as energy or forces – that is treated as a numerical, physical quantity, interpolated when changing the number of PI replicas. Same quantities from different force components are summed as well. The names should correspond to entries in the JSON-formatted extra string.

dtype: string

default: [ ]

6.32 forcefield

Base forcefield class that deals with the assigning of force calculation jobs and collecting the data.

ATTRIBUTES

name

Mandatory. The name by which the forcefield will be identified in the System forces section.

dtype: string

pbc

Applies periodic boundary conditions to the atoms coordinates before passing them on to the driver code.

dtype: boolean

default: False

threaded

Whether the forcefield should use a thread loop to evaluate, or work in serial

dtype: boolean

default: False

FIELDS

latency

The number of seconds the polling thread will wait between exhamining the list of requests.

dtype: float

default: 0.0001

offset

A constant offset that is subtracted from the forcefield energy. Useful when there is a large core energy contribution that is constant throughout a simulation and hides significant changes in the 10th digit.

dtype: float

dimension: energy

default: 0.0

parameters

The parameters of the force field

dtype: dictionary

default: { }

activelist

List with indexes of the atoms that this socket is taking care of. Default: [-1] (corresponding to all)

dtype: integer

***default*:**
[-1]

6.33 forces

Deals with creating all the necessary forcefield objects.

FIELDS

force

The class that deals with how each forcefield contributes to the overall potential, force and virial calculation.

6.34 frequencies

Provides a compact way of specifying the ring polymer frequencies

dtype: float

dimension: frequency

ATTRIBUTES

units

The units the input data is given in.

dtype: string

default: automatic

shape

The shape of the array.

dtype: tuple

default: (0,)

mode

If 'mode' is 'manual', then the array is read in directly, then reshaped according to the 'shape' specified in a row-major manner. If 'mode' is 'file' then the array is read in from the file given.

dtype: string

options: ['manual', 'file']

default: manual

style

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.

dtype: string

options: ['pa-cmd', 'wmax-cmd', 'manual', 'rpmd']

default: rpmd

6.35 gle

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

dtype: string

ATTRIBUTES

mode

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

dtype: string

options: ['chk', 'manual']

default: manual

6.36 h0

Describes with the cell parameters. Takes as array which can be used to initialize the cell vector matrix. N.B.: the cell parameters are stored with the lattice vectors in the columns, and the cell must be oriented in such a way that the array is upper-triangular (i.e. with the first vector aligned along x and the second vector in the xy plane).

dtype: float

dimension: length

ATTRIBUTES

units

The units the input data is given in.

dtype: string

default: automatic

shape

The shape of the array.

dtype: tuple

default: (0,)

mode

If 'mode' is 'manual', then the array is read in directly, then reshaped according to the 'shape' specified in a row-major manner. If 'mode' is 'file' then the array is read in from the file given.

dtype: string

options: ['manual', 'file']

default: manual

6.37 init_cell

This is the class to initialize cell.

dtype: string

ATTRIBUTES

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, lattice vectors stored in columns). The 1st element define lattice vector a, the 2nd, 5th elements define lattice vector b, and the 3rd, 6th, 9th elements define lattice vector c. The other elements are ignored, as the cell must be aligned so that it is upper triangular. 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.

dtype: string

options: ['manual', 'pdb', 'chk', 'abc', 'abcABC']

default: manual

6.38 initialize

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

ATTRIBUTES

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.

dtype: integer

FIELDS

positions

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

dtype: string

velocities

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

dtype: string

momenta

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

dtype: string

masses

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

dtype: string

labels

Initializes atomic labels

dtype: string

cell

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

dtype: string

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. The 'units' attribute is deprecated. Append a 'quantity{units}' to the comment line of the xyz or to the 'TITLE' tag of a pdb.

dtype: string

gle

Initializes the additional momenta in a GLE thermostat.

dtype: string

6.39 instanton

A class for instanton calculations

ATTRIBUTES

mode

Defines whether it is an instanton rate or instanton tunneling splitting calculaion

dtype: string

options: ['rate', 'splitting']

default: rate

FIELDS

tolerances

Convergence criteria for optimization.

biggest_step

The maximum step size during the optimization.

dtype: float

default: 0.4

old_pos

The previous step positions during the optimization.

dtype: float

dimension: length

default: [ ]

old_pot

The previous step potential energy during the optimization

dtype: float

dimension: energy

default: [ ]

old_force

The previous step force during the optimization

dtype: float

dimension: force

default: [ ]

opt

The geometry optimization algorithm to be used. For small system sizes nichols is recomended. Lanczos is tailored for big bigger than nbeads*natoms >~38*64. NR works in both cases given that the initial guess is close to the optimized geometry. Finally lbfgs is used for tunneling splitting calculations.

dtype: string

options: ['nichols', 'NR', 'lbfgs', 'lanczos', 'None']

default: None

max_e

Evaluate the forces in a reduced ring polymer such that the potential energy between consecutive replicas is smaller that the provided value.

dtype: float

dimension: energy

default: 0.0

max_ms

Evaluate the forces in a reduced ring polymer such that that mass-scaled distance in a.u. between consecutive replicas is smaller that the provided value.

dtype: float

default: 0.0

discretization

Allows to specified non uniform time discretization as proposed in J. Chem. Phys. 134, 184107 (2011)

dtype: float

default: [ ]

friction

Activates Friction. Add additional terms to the RP related to a position-independent frictional force. See Eq. 20 in J. Chem. Phys. 156, 194106 (2022)

dtype: boolean

default: False

frictionSD

Activates SD Friction. Add additional terms to the RP related to a position-dependent frictional force. See Eq. 32 in J. Chem. Phys. 156, 194106 (2022)

dtype: boolean

default: True

fric_spec_dens

Laplace Transform (LT) of friction. A two column data is expected. First column: w (cm^-1). Second column: LT(eta)(w). See Eq. 11 in J. Chem. Phys. 156, 194106 (2022). Note that within the separable coupling approximation the frequency dependence of the friction tensor is position independent.

dtype: float

default: [ ]

fric_spec_dens_ener

Energy at which the LT of the friction tensor is evaluated in the client code

dtype: float

dimension: energy

default: 0.0

eta

Friction Tensor. Only to be used when frictionSD is disabled.

dtype: float

default: [ ]

alt_out

Alternative output:Prints different formatting of outputs for geometry, hessian and bead potential energies. All quantities are also accessible from typical i-pi output infrastructure. Default to 1, which prints every step. -1 will suppress the output (except the last one). Any other positive number will set the frequency (in steps) with which the quantities are written to file. The instanton geometry is printed in xyz format and the distances are in angrstroms The hessian is printed in one line with the following format: h1_1,h2_1,...,hN_1, h2_2,h2_2,hN_2, ... ,h1_d,h2_d,...,hN_d. Where N represents the total number of replicas, d the number of dimension of each replica (3*n_atoms) and hi_j means the row j of the physical hessian corresponding to the replica i. The physical hessian uses a convention according to the positions convention used in i-pi. Example of 2 particles, the first two rows of the physical hessian reads: 'H_x1_x1, H_x1_y1, H_x1_z1, H_x1_x2, H_x1_y2,H_x1_z2' 'H_x2_x1, H_x2_y1, H_x2_z1, H_x2_x2, H_x2_y2,H_x2_z2'

dtype: integer

default: 1

prefix

Prefix of the output files.

dtype: string

default: instanton

delta

Initial stretch amplitude.

dtype: float

default: 0.1

hessian_init

How to initialize the hessian if it is not fully provided.

dtype: boolean

default: False

hessian

(Approximate) Hessian.

dtype: float

default: [ ]

hessian_update

How to update the hessian after each step.

dtype: string

options: ['powell', 'recompute']

default: powell

hessian_asr

Removes the zero frequency vibrational modes depending on the symmetry of the system.

dtype: string

options: ['none', 'poly', 'crystal']

default: none

fric_hessian

(Approximate) friction second derivative from which a friction Hessian can be built.

dtype: float

default: [ ]

qlist_lbfgs

List of previous position differences for L-BFGS, if known.

dtype: float

default: [ ]

glist_lbfgs

List of previous gradient differences for L-BFGS, if known.

dtype: float

default: [ ]

old_direction

The previous direction in a CG or SD optimization.

dtype: float

default: [ ]

scale_lbfgs

Scale choice for the initial hessian. 0 identity. 1 Use first member of position/gradient list. 2 Use last member of position/gradient list.

dtype: integer

default: 2

corrections_lbfgs

The number of past vectors to store for L-BFGS.

dtype: integer

default: 20

ls_options

Options for line search methods. Includes: tolerance: stopping tolerance for the search, iter: the maximum number of iterations, step: initial step for bracketing, adaptive: whether to update initial step.

energy_shift

Set the zero of energy.

dtype: float

dimension: energy

default: 0.0

hessian_final

Decide if we are going to compute the final big-hessian by finite difference.

dtype: boolean

default: False

6.40 labels

This is the class to initialize atomic labels.

dtype: string

ATTRIBUTES

mode

The input data format. ‘xyz’ and ‘pdb’ stand for xyz and pdb input files respectively. ‘ase’ is to read a file with the Atomic Simulation Environment. ‘chk’ stands for initialization from a checkpoint file. ‘manual’ means that the value to initialize from is giving explicitly as a vector.

dtype: string

options: [‘manual’, ‘xyz’, ‘pdb’, ‘ase’, ‘chk’]

default: chk

index

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

dtype: integer

default: -1

bead

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

dtype: integer

default: -1

6.41 masses

This is the class to initialize atomic masses.

dtype: string

ATTRIBUTES

mode

The input data format. ‘xyz’ and ‘pdb’ stand for xyz and pdb input files respectively. ‘ase’ is to read a file with the Atomic Simulation Environment. ‘chk’ stands for initialization from a checkpoint file. ‘manual’ means that the value to initialize from is giving explicitly as a vector.

dtype: string

options: [‘manual’, ‘xyz’, ‘pdb’, ‘ase’, ‘chk’]

default: chk

index

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

dtype: integer

default: -1

bead

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

dtype: integer

default: -1

6.42 metad

MetaDynamics

FIELDS

metaff

List of names of forcefields that should do metadynamics.

dtype: string

default: [ ]

use_energy

Transfer the potential energy value to PLUMED to use as a collective variable. Can only be used with classical simulations because it requires a rather hacky mechanism to transfer the energy of the system to the forcefield.

dtype: boolean

default: False

6.43 momenta

This is the class to initialize momenta.

dtype: string

ATTRIBUTES

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.

dtype: string

options: ['manual', 'xyz', 'pdb', 'ase', 'chk', 'thermal']

default: chk

index

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

dtype: integer

default: -1

bead

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

dtype: integer

default: -1

6.44 motion

Allow chosing the type of calculation to be performed. Holds all the information that is calculation specific, such as geometry optimization parameters, etc.

ATTRIBUTES

mode

How atoms should be moved at each step in the simulatio. 'replay' means that a simulation is replayed from trajectories provided to i-PI.

dtype: string

options: ['vibrations', 'minimize', 'replay', 'neb', 'string', 'dynamics', 'driven_dynamics', 'constrained_dynamics', 't_ramp', 'p_ramp', 'alchemy', 'atomswap', 'planetary', 'instanton', 'al-kmc', 'dummy', 'scp', 'normalmodes', 'multi']

FIELDS

fixcom

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

dtype: boolean

default: True

fixatoms

Indices of the atoms that should be held fixed.

dtype: integer

default: [ ]

fixatoms_dof

Indices of the degrees of freedom that should be held fixed.

dtype: integer

default: [ ]

optimizer

Option for geometry optimization

neb_optimizer

Option for NEB optimization

string_optimizer

Option for String minimal-energy path optimization

dynamics

Option for (path integral) molecular dynamics

driven_dynamics

Option for driven molecular dynamics

constrained_dynamics

Option for constrained classical molecular dynamics

file

This describes the location to read a trajectory file from. Replay syntax allows using some POSIX wildcards in the filename of trajectory files. If symbols ?*[] are found in a filename, the code expects to find exactly Nbeads files that match the provided pattern. Bead indices will be read from the files, and the files will be ordered ascendingly by their bead indices. Wildcarded files are expected to be in the folder where the simulation runs.

dtype: string

default:

vibrations

Option for phonon computation

normalmodes

Option for solving the vibrational Schroedinger's equations in normal mode coordinates.

scp

Option for self consistent phonons computation

alchemy

Option for alchemical exchanges

atomswap

Option for Monte Carlo atom swap

t_ramp

Option for temperature ramp

p_ramp

Option for pressure ramp

instanton

Option for Instanton optimization

al6xxx_kmc

Option for Al-6xxx KMC

planetary

Option for planetary model calculator

motion

A motion class that can be included as a member of a 'multi' integrator.

6.45 neb_optimizer

Contains the required parameters for performing nudged elastic band (NEB) calculations

ATTRIBUTES

mode

The geometry optimization algorithm to optimize NEB path

dtype: string

options: ['bfgstrm', 'damped_bfgs', 'fire']

default: fire

FIELDS

tolerances

Tolerance criteria to stop NEB optimization. If you work with DFT, do not use these defaults.

old_coord

The previous position in an optimization step.

dtype: float

dimension: length

default: [ ]

full_force

The previous full-dimensional force in an optimization step.

dtype: float

dimension: force

default: [ ]

full_pots

Previous physical potentials of all beads.

dtype: float

dimension: energy

default: [ ]

old_nebpotential

Previous NEB potential energy, which includes spring energy.

dtype: float

default: [ ]

old_nebgradient

The previous gradient including NEB spring forces.

dtype: float

default: [ ]

old_direction

The previous direction.

dtype: float

default: [ ]

biggest_step

The maximum atomic displacement in a single step of optimizations within NEB procedure. If requested step is larger, it will be downscaled so that maximal atomic displacement won't exceed biggest_step.

dtype: float

dimension: length

default: 0.5

scale_lbfgs

Scale choice for the initial hessian. 0 identity. 1 Use first member of position/gradient list. 2 Use last member of position/gradient list.

dtype: integer

default: 2

hessian_bfgs

Approximate Hessian for damped_BFGS, if known.

dtype: float

default: [ ]

qlist_lbfgs

List of previous position differences for L-BFGS, if known.

dtype: float

default: [ ]

glist_lbfgs

List of previous gradient differences for L-BFGS, if known.

dtype: float

default: [ ]

corrections_lbfgs

The number of past vectors to store for L-BFGS.

dtype: integer

default: 5

dtmax_fire

Maximum time interval per step for FIRE.

dtype: float

default: 1.0

v_fire

Current velocity for FIRE

dtype: float

default: [ ]

alpha_fire

velocity mixing factor for FIRE

dtype: float

default: 0.1

N_down_fire

consecutive steps in downhill dierction for FIRE

dtype: integer

default: 0

N_up_fire

consecutive steps in uphill direction

dtype: integer

default: 0

dt_fire

time per step

dtype: float

default: 0.1

endpoints

Geometry optimization of endpoints (not implemented yet)

spring

Uniform or variable spring constants along the elastic band

tangent

How to calculate tangents: simple averaging from the original 1998 paper, or the improved tangent estimate from J. Chem. Phys. 113, 9978 (2000)

dtype: string

options: [‘plain’, ‘improved’]

default: improved

stage

Stage of the NEB pipeline: optimization of endpoints, NEB itself, climbing image

dtype: string

options: [‘endpoints’, ‘neb’, ‘climb’]

default: neb

use_climb

Use climbing image NEB or not

dtype: boolean

default: False

climb_bead

The index of the climbing bead.

dtype: integer

default: -1

6.46 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.

ATTRIBUTES

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.

dtype: string

options: ['fft', 'matrix']

default: fft

propagator

How to propagate the free ring polymer dynamics. Cayley transform is not exact but is strongly stable and avoid potential resonance issues. A bab scheme performs numerical verlet type propagation. All three options work for distinguishable particles. Only the bab propagator can be used with bosonic particles.

dtype: string

options: ['exact', 'cayley', 'bab']

default: exact

fft_threads

The number of threads to be used for the FFT.

dtype: integer

default: 1

fft_float32

Whether to use single precision FFT.

dtype: boolean

default: False

FIELDS

frequencies

Specifies normal mode frequencies for a (closed path) calculation

dtype: float

dimension: frequency

default: [ ]

open_paths

Indices of the atoms whose path should be opened (zero-based).

dtype: integer

default: [ ]

bosons

Specify which atoms are bosons.

dtype: string

default: [ ]

nmts

The number of iterations to perform one bab step.

dtype: integer

default: 10

6.47 normalmodes

Vibrational self-consistent field class. Approximates the vibrational eigenstates and eigenvalues of a system by performing a normal mode expansion of the potential energy surface.

ATTRIBUTES

mode

The algorithm to be used: independent mode framework (imf) and vibrational self consistent field (vscf).

dtype: string

options: ['imf', 'vscf']

default: imf

FIELDS

prefix

Prefix of the output files.

dtype: string

default:

asr

Removes the zero frequency vibrational modes depending on the symmetry of the system for general polyatomic molecules, and periodic crystal structures.

dtype: string

options: [‘none’, ‘poly’, ‘crystal’]

default: none

dynmat

Portion of the dynamical matrix known to the current point in the calculation.

dtype: float

default: [ ]

nprim

Number of primitive unit cells in the simulation cell.

dtype: float

default: 1.0

fnmrms

Fraction of harmonic RMS displacement used to sample along normal mode.

dtype: float

default: 1.0

nevib

Multiple of harm vibr energy up to which BO surface is sampled.

dtype: float

default: 25.0

nint

Integration points for Hamiltonian matrix elements.

dtype: integer

default: 101

pair_range

The range of pair combinations of normal modes to be considered.

dtype: integer

default: [ ]

nbasis

Number of SHO states used as basis for anharmonic wvfn.

dtype: integer

default: 10

athresh

Convergence threshold for absolute error in vibr free energy per degree of freedom.

dtype: float

dimension: energy

default: 3.6749322e-06

ethresh

Convergence thresh for fractional error in vibr free energy.

dtype: float

default: 0.01

alpha

The fraction of mean field potential to mix with the result of the previous SCF iteration.

dtype: float

default: 1.0

nkbt

Threshold for (e - e_gs)/(kB T) of vibr state to be incl in the VSCF and partition function.

dtype: float

default: 4.0

nexc

Minimum number of excited n-body states to calculate (also in MP2 correction).

dtype: integer

default: 5

mptwo

Flag determining whether MP2 correction is calculated.

dtype: boolean

default: False

solve

Flag determining whether the VSCF mean field Schroedinger's equation is solved.

dtype: boolean

default: False

grid

Flag determining whether the coupling potential is gridded or not.

dtype: boolean

default: True

print_mftpot

Flag determining whether MFT potentials are printed to file.

dtype: boolean

default: False

print_1b_map

Flag determining whether the independent mode potentials are printed to file.

dtype: boolean

default: False

print_2b_map

Flag determining whether the two body mapped coupling potentials are printed to file.

dtype: boolean

default: False

print_vib_density

Flag determining whether the vibrational density (psi**2) are printed to file.

dtype: boolean

default: False

threebody

Flag determining whether three-mode coupling terms are accounted for.

dtype: boolean

default: False

nparallel

The number of forces evaluations per i-PI step.

dtype: integer

default: 1

6.48 optimizer

A Geometry Optimization class implementing most of the standard methods

ATTRIBUTES

mode

The geometry optimization algorithm to be used

dtype: string

options: ['sd', 'cg', 'bfgs', 'bfgstrm', 'lbfgs', 'damped_bfgs']

default: lbfgs

FIELDS

ls_options

"Options for line search methods. Includes: tolerance: stopping tolerance for the search (as a fraction of the overall energy tolerance), iter: the maximum number of iterations, step: initial step for bracketing, adaptive: whether to update initial step.

exit_on_convergence

Terminates the simulation when the convergence criteria are met.

dtype: boolean

default: True

tolerances

Convergence criteria for optimization. Default values are extremely conservative. Set them to appropriate values for production runs.

biggest_step

The maximum step size for (L)-BFGS line minimizations.

dtype: float

default: 100.0

scale_lbfgs

Scale choice for the initial hessian. 0 identity. 1 Use first member of position/gradient list. 2 Use last member of position/gradient list.

dtype: integer

default: 2

corrections_lbfgs

The number of past vectors to store for L-BFGS.

dtype: integer

default: 6

old_pos

The previous positions in an optimization step.

dtype: float

dimension: length

default: [ ]

old_pot

The previous potential energy in an optimization step.

dtype: float

dimension: energy

default: [ ]

old_force

The previous force in an optimization step.

dtype: float

dimension: force

default: [ ]

old_direction

The previous direction in a CG or SD optimization.

dtype: float

default: [ ]

invhessian_bfgs

Approximate inverse Hessian for BFGS, if known.

dtype: float

default: [ ]

hessian_trm

Approximate Hessian for trm, if known.

dtype: float

default: [ ]

tr_trm

The trust radius in trm.

dtype: float

dimension: length

default: [ ]

qlist_lbfgs

List of previous position differences for L-BFGS, if known.

dtype: float

default: [ ]

glist_lbfgs

List of previous gradient differences for L-BFGS, if known.

dtype: float

default: [ ]

6.49 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.

ATTRIBUTES

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.

dtype: string

default: i-pi

FIELDS

properties

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

dtype: string

trajectory

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

dtype: string

checkpoint

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

dtype: integer

6.50 p_ramp

PressureRamp Motion class. It just updates the ensemble pressure in steps, between the indicated values, and then holds to the highest value. It should typically be combined with a dynamics class and barostats, using a MultiMotion.

FIELDS

p_start

Initial pressure

dtype: float

dimension: pressure

default: 1.0

p_end

Final pressure

dtype: float

dimension: pressure

default: 1.0

logscale

Change pressure on a logarihthmic scale.

dtype: boolean

default: False

total_steps

Total number of steps for the ramp

dtype: integer

default: 0

current_step

Current step along the ramp

dtype: integer

default: 0

6.51 planetary

Holds all the information for the planetary model frequency matrix calculator.

ATTRIBUTES

mode

The constrained-centroid sampling mode.

dtype: string

options: ['md']

default: md

FIELDS

thermostat

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

timestep

The time step.

dtype: float

dimension: time

default: 1.0

nmts

Number of iterations for each MTS level (including the outer loop, that should in most cases have just one iteration).

dtype: integer

default: [ ]

nsamples

Number of samples to accumulate for each planetary step.

dtype: integer

default: 0

stride

How often the planetary calculation should actually be triggered.

dtype: integer

default: 1

nbeads

Number of beads for centroid-constrained dynamics (default same as main trajectory)

dtype: integer

default: -1

screen

Screening parameter for path-integral frequency matrix.

dtype: float

dimension: length

default: 0.0

6.52 positions

This is the class to initialize positions.

dtype: string

ATTRIBUTES

mode

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

dtype: string

options: ['manual', 'xyz', 'pdb', 'ase', 'chk']

default: chk

index

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

dtype: integer

default: -1

bead

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

dtype: integer

default: -1

6.53 prng

Deals with the pseudo-random number generator.

ATTRIBUTES

n_threads

Use parallel PRNG generator. Will make trajectories less reproducible and is only faster if the arrays are very large.

dtype: integer

default: 1

FIELDS

seed

This specifies the seed number used to generate the initial state of the random number generator. Previously, the default seed was fixed as 123456. Currently, the default seed is random and given by the system time (down to ms). This is done in utils/prng.py.

dtype: integer

default: -1

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.

dtype: string

default:

6.54 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.

dtype: string

ATTRIBUTES

shape

The shape of the array.

dtype: tuple

default: (0,)

mode

If 'mode' is 'manual', then the array is read in directly, then reshaped according to the 'shape' specified in a row-major manner. If 'mode' is 'file' then the array is read in from the file given.

dtype: string

options: ['manual', 'file']

default: manual

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.

dtype: string

default: out

stride

The number of steps between successive writes.

dtype: integer

default: 1

flush

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

dtype: integer

default: 1

verbosity

The level of output on stdout. 'low' will print only properties and column names, 'high' will print information for reproducibility.

dtype: string

options: ['low', 'high']

default: low

6.55 remd

Replica Exchange

FIELDS

stride

Every how often to try exchanges (on average).

dtype: float

default: 1.0

krescale

Rescale kinetic energy upon exchanges.

dtype: boolean

default: True

swapfile

File to keep track of replica exchanges

dtype: string

default: remd_idx

repindex

List of current indices of the replicas compared to the starting indices

dtype: integer

default: [ ]

6.56 scp

Self-consistent phonons class. It variationally optimizes the free energy to calculate the best harmonic approximation to a system.

ATTRIBUTES

mode

The statistics to be used in the calculation of the free energy. Quantum (qn) or classical (cl) Boltzmann statistics.

dtype: string

options: ['qn', 'cl']

default: qn

FIELDS

prefix

Prefix of the output files.

dtype: string

default:

asr

The method used to project out zero modes coming from continuous symmetries: crystal removes the three translational modes; molecule removes the three rotational modes in addition to the translational ones. none keeps all the modes.

dtype: string

options: ['none', 'crystal', 'poly']

default: none

random_type

Chooses the type of random numbers.

dtype: string

options: ['sobol', 'pseudo', 'file']

default: pseudo

displace_mode

The type of optimisation strategy for obtaining the mean position. sd stands for a steepest descent algorithm. ik stands for a Newton-Raphson scheme that requires the inverse of the force constant matrix iK. nmik stands for a Newton-Raphson scheme that only displaces along normal modes directions with statistically significant forces. rnmik same as nmik but performs several optimization steps using a reweighted sampling.

dtype: string

options: ['ik', 'sd', 'nmik', 'rnmik']

default: nmik

dynmat

The dynamical matrix of the trial Hamiltonian.

dtype: float

default: [ ]

max_steps

Maximum number of Monte carlo steps per SCP iteration.

dtype: integer

max_iter

Maximum number of SCP iterations.

dtype: integer

default: 1

tau

Step size along the gradient for the sd displace_mode

dtype: float

default: 1.0

wthreshold

Threshold on minimum Boltzmann weights before more statistics must be accumulated.

dtype: float

default: 0.9

precheck

Flag for checking statistical significance of forces before optimisation of mean position.

dtype: boolean

default: True

checkweights

Flag for checking Boltzmann weights for whether more statistics are required.

dtype: boolean

default: True

chop

Threshold below which frequencies are set to zero.

dtype: float

default: 1e-09

nparallel

The number of Monte Carlo forces to be evaluated (in parallel) per i-PI step.

dtype: integer

default: 1

batch_weight_exponent

The exponent used to suppress low batch weights.

dtype: integer

default: 1

6.57 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.

ATTRIBUTES

verbosity

The level of output on stdout.

dtype: string

options: ['quiet', 'low', 'medium', 'high', 'debug']

default: medium

threading

Whether multiple-systems execution should be parallel. Makes execution non-reproducible due to the random number generator being used from concurrent threads.

dtype: boolean

default: True

mode

What kind of simulation should be run.

dtype: string

options: ['md', 'static']

default: md

safe_stride

Consistent simulation states will be saved every this number of steps. Saving state entails a small overhead, so you may want to set this to the smallest output frequency in your simulation to make i-PI faster. Use at your own risk!

dtype: integer

default: 1

floatformat

A format for all printed floats.

dtype: string

default: %16.8e

sockets_prefix

A prefix prepended to the *address* value to form the UNIX-domain socket location.

dtype: string

default: /tmp/ipi_

FIELDS

prng

Deals with the pseudo-random number generator.

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.

step

The current simulation time step.

dtype: integer

default: 0

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.

dtype: integer

default: 1000

total_time

The maximum wall clock time (in seconds).

dtype: float

default: 0

smotion

Options for a 'super-motion' step between system replicas

system

This is the class which holds all the data which represents a single state of the system.

system_template

Generic input value

ffsocket

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

ffdirect

Direct potential that evaluates forces through a Python call, using PES providers from a list of possible external codes. The available PES interfaces are listed into the *ipi/pes* folder, and are the same available for the Python driver. The <*parameters*> field should contain a dictionary of the specific option of the chosen PES.

fflj

Simple, internal LJ evaluator without cutoff, neighbour lists or minimal image convention. Expects standard LJ parameters, e.g. { eps: 0.1, sigma: 1.0 }.

ffdmd

Simple, internal DMD evaluator without without neighbor lists, but with PBC. Expects coupling elements (n*(n-1)/2 of them), oscillating frequency and time step.

ffdebye

Harmonic energy calculator

ffplumed

Direct PLUMED interface. Can be used to implement metadynamics in i-PI in combination with the <metad> SMotion class. NB: if you use PLUMED for constant biasing (e.g. for umbrella sampling) the bias will be computed but there will be no output as specified in the *plumed.dat* file unless you include a <metad> tag, that triggers the log update.

ffyaff

Uses a Yaff force field to compute the forces.

ffcommittee

Combines multiple forcefields to build a committee model, that can be used to compute uncertainty-quantified machine-learning models. Each forcefield can be any of the other FF objects, and each should be used with a client that generates a slightly different estimation of energy and forces. These are averaged, and the mean used as the actual forcefield. Statistics about the distribution are also returned as extras fields, and can be printed for further postprocessing. It is also possible for a single FF object to return a JSON-formatted string containing entries *committee_pot*, *committee_force* and *committee_virial*, that contain multiple members at once. These will be unpacked and combined with whatever else is present. Also contains options to use it for uncertainty estimation and for active learning in a ML context, based on a committee model. Implements the approaches discussed in [Musil et al.](http://doi.org/10.1021/acs.jctc.8b00959) and [Imbalzano et al.](http://doi.org/10.1063/5.0036522)

ffrotations

Wraps around another forcefield to evaluate it over one or more rotated copies of the physical system. This is useful when interacting with models that are not exactly invariant/covariant with respect to rigid rotations. Besides the parameters defining how averaging is to be performed (using an integration grid, and/or randomizing the orientation at each step) the <ffrotations> should contain either a <ffsocket> or a <ffdirect> block that computes the "base" model. Note that this forcefield should be given a separate *name*, but that you cannot access this "inner" forcefield from other parts of the input file.

ffsgdml

A SGDML energy calculator

ffcavphsocket

A cavity molecular dynamics driver for vibraitonal strong coupling. In the current implementation, only a single cavity mode polarized along the x and y directions is coupled to the molecules. Check https://doi.org/10.1073/pnas.2009272117 and also examples/lammps/h2o-cavmd/ for details.

6.58 smotion

Allow chosing the type of smotion to be performed. Holds all the information that is calculation specific, such as replica exchange parameters, etc.

ATTRIBUTES

mode

Kind of smotion which should be performed.

dtype: string

options: ['dummy', 'remd', 'metad', 'dmd', 'multi']

FIELDS

remd

Option for REMD simulation

metad

Option for REMD simulation

dmd

Option for driven MD simulation

smotion

A smotion class that can be included as a member of a 'multi' Smotion.

6.59 string_optimizer

Contains the required parameters for performing string minimal energy path optimization.

ATTRIBUTES

mode

The geometry optimization algorithm to optimize MEP string

dtype: string

options: ['sd', 'cg', 'bfgs', 'bfgstrm', 'damped_bfgs', 'lbfgs', 'fire', 'euler']

default: bfgstrm

FIELDS

tolerances

Tolerance criteria to stop String optimization. If you work with DFT, do not use these defaults.

old_coord

The previous position in an optimization step.

dtype: float

dimension: length

default: [ ]

full_force

The previous full-dimensional force in an optimization step.

dtype: float

dimension: force

default: [ ]

full_pots

Previous physical potentials of all beads.

dtype: float

dimension: energy

default: [ ]

old_stringpotential

Previous string potential energy.

dtype: float

default: [ ]

old_stringgradient

The previous gradient of the string.

dtype: float

default: [ ]

old_direction

The previous direction.

dtype: float

default: [ ]

biggest_step

The maximum atomic displacement in a single step of optimizations within String MEP procedure. If requested step is larger, it will be downscaled so that maximal atomic displacement won't exceed biggest_step.

dtype: float

dimension: length

default: 0.5

scale_lbfgs

Scale choice for the initial hessian. 0 identity. 1 Use first member of position/gradient list. 2 Use last member of position/gradient list.

dtype: integer

default: 2

hessian_bfgs

Approximate Hessian for damped_BFGS, if known.

dtype: float

default: [ ]

qlist_lbfgs

List of previous position differences for L-BFGS, if known.

dtype: float

default: [ ]

glist_lbfgs

List of previous gradient differences for L-BFGS, if known.

dtype: float

default: [ ]

corrections_lbfgs

The number of past vectors to store for L-BFGS.

dtype: integer

default: 5

tr_trm

Starting value for the trust radius for BFGSTRM.

dtype: float

dimension: length

***default*:**
[1.]

dtmax_fire

Maximum time interval per step for FIRE.

dtype: float

default: 1.0

v_fire

Current velocity for FIRE

dtype: float

default: [ ]

alpha_fire

velocity mixing factor for FIRE

dtype: float

default: 0.1

N_down_fire

consecutive steps in downhill dierction for FIRE

dtype: integer

default: 0

N_up_fire

consecutive steps in uphill direction

dtype: integer

default: 0

dt_fire

time per step

dtype: float

default: 0.1

endpoints

Geometry optimization of endpoints (not implemented yet)

stage

Stage of the String pipeline: optimization of the endpoints, string opt., climbing image opt.

dtype: string

options: ['endpoints', 'string', 'climb']

default: string

use_climb

Use climbing image String MEP or not

dtype: boolean

default: False

climb_bead

The index of the climbing bead.

dtype: integer

default: -1

6.60 system

This is the class which holds all the data which represents a single state of the system.

ATTRIBUTES

prefix

Prepend this string to output files generated for this system.

dtype: string

default:

FIELDS

initialize

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

forces

Deals with creating all the necessary forcefield objects.

ensemble

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

motion

Allow chosing the type of calculation to be performed. Holds all the information that is calculation specific, such as geometry optimization parameters, etc.

beads

Describes the bead configurations in a path integral simulation.

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.

cell

Describes with the cell parameters. Takes as array which can be used to initialize the cell vector matrix. N.B.: the cell parameters are stored with the lattice vectors in the columns, and the cell must be oriented in such a way that the array is upper-triangular (i.e. with the first vector aligned along x and the second vector in the xy plane).

dtype: float

dimension: length

***default*:**
[0. 0. 0. 0. 0. 0. 0. 0. 0.]

6.61 system_template

Generic input value

FIELDS

template

A string that will be read verbatim containing the model for a system to be generated

dtype: string

labels

A list of strings that should be substituted in the template to create multiple systems

dtype: string

instance

A list of strings that should the labels creating one system instance

dtype: string

6.62 t_ramp

TemperatureRamp Motion class. It just updates the ensemble temperature in steps, between the indicated temperatures, and then holds to the highest value. It should typically be combined with a dynamics class and thermostats, using a MultiMotion.

FIELDS

t_start

Initial temperature

dtype: float

dimension: energy

default: 1.0

t_end

Final temperature

dtype: float

dimension: energy

default: 1.0

logscale

Change temperature on a logarihthmic scale.

dtype: boolean

default: False

total_steps

Total number of steps for the ramp

dtype: integer

default: 0

current_step

Current step along the ramp

dtype: integer

default: 0

6.63 thermostat

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

ATTRIBUTES

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. 'cl' represents a modified langevin thermostat which compensates for additional white noise from noisy forces or for dissipative effects. 'ffl' is the fast-forward langevin thermostat, in which momenta are flipped back whenever the action of the thermostat changes its direction. 'multiple' is a special thermostat mode, in which one can define multiple thermostats _inside_ the thermostat tag.

dtype: string

options: ['', 'langevin', 'svr', 'pile_l', 'pile_g', 'gle', 'nm_gle', 'nm_gle_g', 'cl', 'ffl', 'multi']

FIELDS

ethermo

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

dtype: float

dimension: energy

default: 0.0

tau

The friction coefficient for white noise thermostats.

dtype: float

dimension: time

default: 0.0

pile_lambda

Scaling for the PILE damping relative to the critical damping. (gamma_k=2*lambda*omega_k

dtype: float

default: 1.0

pile_centroid_t

Option to set a different centroid temperature wrt. that of the ensemble. Only used if value other than 0.0.

dtype: float

dimension: temperature

default: 0.0

A

The friction matrix for GLE thermostats.

dtype: float

dimension: frequency

default: [ ]

C

The covariance matrix for GLE thermostats.

dtype: float

dimension: temperature

default: [ ]

s

Input values for the additional momenta in GLE.

dtype: float

dimension: ms-momentum

default: [ ]

intau

The inherent noise time scale for compensating langevin thermostats.

dtype: float

dimension: time

default: 0.0

idtau

The inherent dissipation time scale for compensating langevin thermostats.

dtype: float

dimension: time

default: 0.0

apat

The time scale for automatic adjustment of CL thermostat's parameters.

dtype: float

dimension: time

default: 0.0

flip

Flipping type for ffl thermostat ('soft', 'hard', 'rescale', 'none')

dtype: string

default: rescale

thermostat

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

6.64 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.

dtype: string

ATTRIBUTES

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.

dtype: string

default: traj

stride

The number of steps between successive writes.

dtype: integer

default: 1

format

The output file format.

dtype: string

options: ['xyz', 'pdb', 'ase', 'bin']

default: xyz

cell_units

The units for the cell dimensions.

dtype: string

default:

bead

Print out only the specified bead. A negative value means print only one every -(bead) beads, e.g. -2 means print just the even beads, -4 one every four and so on.

dtype: integer

default: -1

flush

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

dtype: integer

default: 1

extra_type

What extra to print from the extra, if it's returned as a JSON dictionary. Can also use 'raw' to print the full data of the unprocessed extra string, or a comma-separated list of keys to print multiple keys, horizontally stacked.

dtype: string

default: raw

6.65 velocities

This is the class to initialize velocities.

dtype: string

ATTRIBUTES

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.

dtype: string

options: ['manual', 'xyz', 'pdb', 'ase', 'chk', 'thermal']

default: chk

index

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

dtype: integer

default: -1

bead

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

dtype: integer

default: -1

6.66 vibrations

Dynamical matrix Class. It calculates phonon modes and frequencies in solids as well as normal vibrational modes and frequencies of aperiodic systems.

ATTRIBUTES

mode

The algorithm to be used: finite differences (fd), normal modes finite differences (nmfd), and energy-scaled normal mode finite differences (enmfd).

dtype: string

options: ['fd', 'nmfd', 'enmfd']

default: fd

FIELDS

pos_shift

The finite displacement in position used to compute derivative of force.

dtype: float

default: 0.01

energy_shift

The finite displacement in energy used to compute derivative of force.

dtype: float

default: 0.0

output_shift

Shift by the dynamical matrix diagonally before outputting.

dtype: float

default: 0.0

prefix

Prefix of the output files.

dtype: string

default: phonons

asr

Removes the zero frequency vibrational modes depending on the symmerty of the system.

dtype: string

options: ['none', 'poly', 'lin', 'crystal']

default: none

dynmat

Portion of the dynamical matrix known up to now.

dtype: float

default: [ ]

refdynmat

Portion of the refined dynamical matrix known up to now.

dtype: float

default: [ ]

7 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 file tags* 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 *Part 1 - NVT Equilibration run*. To give a more in depth explanation of each of these files, they will now be considered in turn.

7.1 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 ->
timepicosecond : 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{units}(...), ... ] </properties>
```

e.g.

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.

See also the documentation of the *properties* tag, and the full *List of available properties*.

7.2 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 file tags* 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 *Configuration files*.

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.

See also the *trajectory* tag, and the full *List of available trajectory files*.

7.3 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 *properties* and *trajectory* 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=/>
```

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.

In order to restart i-PI from a file named "RESTART" one simply has to run

```
> python i-pi RESTART
```

7.4 Reading output files

It can be useful to parse the output files of i-PI into a format that can be readily manipulated in a custom Python script. To this end, i-PI provides a few utilities. *ipi.read_output* that can be used to parse a property output file, that returns data blocks as a dictionary of numpy array, and additional information from the header (such as units, and the description of each property) as a separate dictionary

```
from ipi import read_output
data, info = read_output("simulation.out")
```

Trajectory files can be read with *ipi.read_trajectory*. This reads the trajectory output into a list of *ase.Atoms* objects (hence this functionality has a dependency on *ase*), converting positions and cell to angstrom, and moving other properties to arrays and converting them to ASE units (e.g. forces are in eV/Å). *extras* output files (that contain either numerical data, or raw strings, returned by the driver code in addition to energy and forces) can be processed by using *format='extras'* and an option: the call will then return a dictionary with an entry having the name of the type of extra

(if present) and either a list of the raw strings, or a numpy array with the data. A second dictionary entry contains the list of step numbers.

```
from ipi import read_trajectory
data = read_trajectory("simulation.dipoles", format="extras")
```

8 Output file tags

8.1 List of available properties

The following list briefly describes all the property names that can be listed in the *properties* tag of the *Input files*, and which will be written in the output files.

Eenvelope: The (gaussian) envelope function of the external applied electric field (values go from 0 to 1).

Efield: The external applied electric field (x,y,z components in cartesian axes).

dimension: electric-field; size: 3;

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_f_path: The forces acting on all the beads 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;

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;

atom_x_path: The positions of all the beads of a particle given its index. Takes an argument index (zero based).

dimension: length;

bead_potentials: The physical system potential energy of each bead.

dimension: energy; size: nbeads;

bweights_component: The weight associated one part of the hamiltonian. Takes one mandatory argument index (zero-based) that indicates for which component of the hamiltonian the weight must be returned.

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;

chin_weight: The 3 numbers output are 1) the logarithm of the weighting factor -beta_P delta H, 2) the square of the logarithm, and 3) the weighting factor

size: 3;

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

dimension: energy;

density: The mass density of the physical system.

dimension: density;

dipole: The beads-averaged electric dipole moment or the electric dipole moment of a single bead (x,y,z components in cartesian axes).

dimension: electric-dipole; size: 3;

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.

ensemble_bias: The bias applied to the current ensemble

dimension: energy;

ensemble_lp: The log of the ensemble probability

ensemble_pressure: The target pressure for the current ensemble

dimension: pressure;

ensemble_temperature: The target temperature for the current ensemble

dimension: temperature;

exchange_all_prob: Probability of the ring polymer exchange configuration where all atoms are connected. It is divided by 1/N, so the number is between 0 and N, while the asymptotic value at low temperatures is 1.

size: 1;

exchange_distinct_prob: Probability of the distinguishable ring polymer configuration, where each atom has its own separate ring polymer. A number between 0 and 1, tends to 1 in high temperatures, which indicates that bosonic exchange is negligible

size: 1;

fermionic_sign: Estimator for the fermionic sign, also used for reweighting fermionic observables. Decreases exponentially with beta and the number of particles, but if not too large, can be used to recover fermionic statistics from bosonic simulations, see doi:10.1063/5.0008720.

size: 1;

forcemod: The modulus of the force. With the optional argument 'bead' will print the force associated with the specified bead.

dimension: force;

hweights_component: The weight associated one part of the hamiltonian. Takes one mandatory argument index (zero-based) that indicates for which component of the hamiltonian the weight must be returned.

interaction_energy: The value of the interaction energy due to the external electric field.

dimension: energy;

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;

isotope_zetasc: Returns the (many) terms needed to directly compute the relative probablity of isotope substitution in two different systems/phases. Takes four arguments, 'alpha' , which gives the scaled mass parameter and default to '1.0', and 'atom', which is the label or index of a type of atoms. The 3 numbers output are 1) the average over the excess potential energy for scaled coordinates <sc>, 2) the average of the squares of the excess potential energy <sc**2>, and 3) the average of the exponential of excess potential energy <exp(-beta*sc)>

size: 3;

isotope_zetasc_4th: Returns the (many) terms needed to compute the scaled-coordinates fourth-order direct estimator. Takes two arguments, 'alpha' , which gives the scaled mass parameter and default to '1.0', and 'atom', which is the label or index of a type of atoms. The 5 numbers output are 1) the average over the excess potential energy for an isotope atom substitution <sc>, 2) the average of the squares of the excess potential energy <sc**2>, and 3) the average of the exponential of excess potential energy <exp(-beta*sc)>, and 4-5) Suzuki-Chin and Takahashi-Imada 4th-order reweighing term

size: 5;

isotope_zetatd: Returns the (many) terms needed to directly compute the relative probablity of isotope substitution in two different systems/phases. Takes two arguments, 'alpha' , which gives the scaled mass parameter and default to '1.0', and 'atom', which is the label or index of a type of atoms. The 3 numbers output are 1) the average over the excess spring energy for an isotope atom substitution <spr>, 2) the average of the squares of the excess spring energy <spr**2>, and 3) the average of the exponential of excess spring energy <exp(-beta*spr)>

size: 3;

isotope_zetatd_4th: Returns the (many) terms needed to compute the thermodynamic fourth-order direct estimator. Takes two arguments, 'alpha' , which gives the scaled mass parameter and default to '1.0', and 'atom', which is the label or index of a type of atoms. The 5 numbers output are 1) the average over the excess spring energy for an isotope atom substitution <spr>, 2) the average of the squares of the excess spring energy <spr**2>, and 3) the average of the exponential of excess spring energy <exp(-beta*spr)>, and 4-5) Suzuki-Chin and Takahashi-Imada 4th-order reweighing term

size: 5;

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 optional arguments 'atom', 'bead' or 'nm'. 'atom' can be either an atom label or an index (zero-based) to specify which species or individual atom to output the kinetic energy of. If not specified, all atoms are used and averaged. 'bead' or 'nm' specify whether the kinetic energy should be computed for a single bead or normal mode. If not specified, all atoms/beads/nm are used.

dimension: energy;

kinetic_opsc: 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_prsc: The Suzuki-Chin primitive estimator of the quantum kinetic energy of the physical system

dimension: energy;

kinetic_td: The primitive 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_tdsc: The Suzuki-Chin centroid-virial thermodynamic estimator of the 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_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;

kstress_tdsc: The Suzuki-Chin thermodynamic estimator for pressure of the physical system.

dimension: pressure;

pot_component: The contribution to the system potential from one of the force components. Takes one mandatory argument index (zero-based) that indicates which component of the potential must be returned. The optional argument 'bead' will print the potential associated with the specified bead (interpolated to the full ring polymer). If the potential is weighed, the weight will be applied.

dimension: energy;

pot_component_raw: The contribution to the system potential from one of the force components. Takes one mandatory argument index (zero-based) that indicates which component of the potential must be returned. The optional argument 'bead' will print the potential associated with the specified bead, at the level of discretization of the given component. Potential weights will not be applied.

dimension: energy;

potential: The physical system potential energy. With the optional argument 'bead' will print the potential associated with the specified bead.

dimension: energy;

potential_opsc: The Suzuki-Chin operator estimator for the potential energy of the physical system.

dimension: energy;

potential_tdsc: The Suzuki-chin thermodyanmic estimator for the potential energy of the physical system.

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;

pressure_tdsc: The Suzuki-Chin thermodynamic estimator for pressure of the physical 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;

sc_op_scaledcoords: Returns the estimators that are required to evaluate the scaled-coordinates estimators for total energy and heat capacity for a Suzuki-Chin high-order factorization, as described in Appendix B of J. Chem. Theory Comput. 2019, 15, 3237-3249 (2019). 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 -0.0001. 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;

sc_scaledcoords: Returns the estimators that are required to evaluate the scaled-coordinates estimators for total energy and heat capacity for a Suzuki-Chin fourth-order factorization, 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 -0.0001. 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;

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 -0.0001. 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 optional arguments 'atom', 'bead' or 'nm'. 'atom' 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. 'bead' or 'nm' specify whether the temperature should be computed for a single bead or normal mode.

dimension: temperature;

ti_pot: The correction potential in Takahashi-Imada 4th-order PI expansion. Takes an argument 'atom', which can be either an atom label or index (zero based) to specify which species to find the correction term for. If not specified, all atoms are used.

dimension: energy; size: 1;

ti_weight: The 3 numbers output are 1) the logarithm of the weighting factor -beta_P delta H, 2) the square of the logarithm, and 3) the weighting factor

size: 3;

time: The elapsed simulation time.

dimension: time;

vcom: The center of mass velocity (x,y,z) of the system or of a species. Takes arguments label (default to all species) and bead (zero based). If bead is not specified, refers to the centroid.

dimension: velocity; size: 3;

vir_component: The contribution to the system potential from one of the force components. Takes one mandatory argument index (zero-based) that indicates which component of the potential must be returned. The optional argument 'bead' will print the potential associated with the specified bead (interpolated to the full ring polymer). If the potential is weighed, the weight will be applied.

dimension: pressure; size: 6;

vir_component_raw: The contribution to the system virial from one of the force components. Takes one mandatory argument index (zero-based) that indicates which component of the potential must be returned. The optional argument 'bead' will print the potential associated with the specified bead, at the level of discretization of the given component. Potential weights will not be applied.

dimension: pressure; size: 6;

vir_tdsc: The Suzuki-Chin thermodynamic estimator for pressure of the physical system.

dimension: pressure;

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_fq: Returns the scalar product of force and positions. Useful to compensate for the harmonic component of a potential. Gets one argument 'ref' that should be a filename for a reference configuration, in the style of the FFDebye geometry input, and one that contains the input units.

dimension: energy; size: 1;

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;

8.2 List of available trajectory files

The following list briefly describes all the trajectory types that can be listed in the *trajectory* tag of the *Input files*.

Eforces: The external electric field contribution to the forces

dimension: force;

becx: The x component of the Born Effective Charges in cartesian coordinates.

dimension: number;

becy: The y component of the Born Effective Charges in cartesian coordinates.

dimension: number;

becz: The z component of the Born Effective Charges in cartesian coordinates.

dimension: number;

extras: The additional data returned by the client code. If the attribute "extra_type" is specified, and if the data is JSON formatted, it prints only the specified field. Otherwise (or if extra_type="raw") the full string is printed verbatim. Will print out one file per bead, unless the bead attribute is set by the user.

extras_bias: The additional data returned by the bias forcefield, printed verbatim or expanded as a dictionary. See "extras".

extras_component_raw: The additional data returned by the client code, printed verbatim or expanded as a dictionary. See "extras". Fetches the extras from a specific force component, indicated in parentheses and a specific bead [extras_component_raw(idx; bead=0)]. Never applies weighting or contraction, and does not automatically sum over beads as we don't know if the extras are numeric

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;

forces_component: The contribution to the system forces from one of the force components. Takes one mandatory argument index (zero-based) that indicates which component of the potential must be returned. The optional argument 'bead' will print the potential associated with the specified bead (interpolated to the full ring polymer), otherwise the centoid force is computed. If the potential is weighed, the weight will be applied.

dimension: force;

forces_component_raw: The contribution to the system forces from one of the force components. Takes one mandatory argument index (zero-based) that indicates which component of the potential must be returned. The optional argument 'bead' will print the potential associated with the specified bead (with the level of discretization of the component), otherwise the centoid force is computed. The weight of the potential is not applied.

dimension: force;

forces_sc: The Suzuki-Chin component of force trajectories. Will print out one file per bead, unless the bead attribute is set by the user.

dimension: force;

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

dimension: force;

isotope_zetasc: Scaled-coordinates isotope fractionation direct estimator in the form of ratios of partition functions. Takes two arguments, 'alpha' , which gives the scaled mass parameter and default to '1.0', and 'atom', which is the label or index of a type of atoms. All the atoms but the selected ones will have zero output

isotope_zetatd: Thermodynamic isotope fractionation direct estimator in the form of ratios of partition functions. Takes two arguments, 'alpha' , which gives the scaled mass parameter and default to '1.0', and 'atom', which is the label or index of a type of atoms. All the atoms but the selected ones will have zero output

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;

v_centroid_even: The suzuki-chin centroid velocity.

dimension: velocity;

v_centroid_odd: The suzuki-chin 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;

x_centroid_even: The suzuki-chin centroid coordinates.

dimension: length;

x_centroid_odd: The suzuki-chin centroid coordinates.

dimension: length;

9 Tools and python utilities

9.1 Post-processing tools

Some observables (such as autocorrelation functions, isotope fractionation ratios, particle momentum distributions) require post-processing the raw outputs of a simulation. The folder `tools` contains several post-processing scripts, that can be used to this end - each (tersely) self-documented with a help string and in the code. Some of these tools are also accessible as python functions, so that they can be invoked from custom post-processing workflows. They can be found in the `ipi/utils/tools` folder, and accessed by importing the module, e.g.

```
from ipi.utils.tools import isra_deconvolute
```

is a routine to correct the correlation spectrum computed from a thermostatted simulation, see this example.

9.2 Parsing

i-PI trajectories can be output in extended xyz format, and can be read by ASE, while the property outputs can be simply read with `np.loadtxt`. However, more reliable parsing can be achieved using the `ipi.read_output` and `ipi.read_trajectory` functions, that can also read the native i-PI trajectory formats, and that parse the header of the output file to provide metadata on the type and units of the properties it contains.

9.3 Scripting i-PI

If one wants to run an i-PI simulation within a Python script, it is also possible to use a (somewhat primitive) scripting API, defined in the `ipi.utils.scripting` module. The core component is the `InteractiveSimulation` class, that can be initialized from an *XML* input, advanced for a given number of steps using the `run` method. The calculation requires also the use of a driver, that can communicate through sockets (in which case it must be launched after initialization, and before running) or directly using a Python API using an *ffdirect* block.

Properties can be accessed using the `properties` method, and a snapshot of the configuration by calling `get_structures`. It is also possible to call `set_structures` to change positions, cell and (if available) velocities to those that can be found in an Atoms structure (or a list of structures in case there are multiple systems and/or multiple beads). Several utility functions, also available within the module, facilitate building on the fly an *XML* string to initialize the simulation. An example of usage of this interface goes as follows:

```
from ipi.utils.scripting import InteractiveSimulation, simulation_xml, motion_nvt_xml,␣
↪forcefield_xml

data = ase.io.read("init.xyz")
input_xml = simulation_xml(
    structures=data,
    forcefield=forcefield_xml(name="dummy", mode="direct"),
    motion=motion_nvt_xml(timestep=2.0 * ase.units.fs),
    temperature=300,
    prefix="example",
)

sim = InteractiveSimulation(input_xml)
sim.run(100)
potential = sim.properties("potential")
ase.io.write("final_structure.xyz", sim.get_structures())
```

10 Distributed execution

10.1 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, 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 *Trajectory files*). The string can be formatted in the JSON format, in which case i-PI can extract and process individual fields, that can be printed separately to different files.

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.

10.2 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 *Running on a HPC system*), 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 main process of the client.

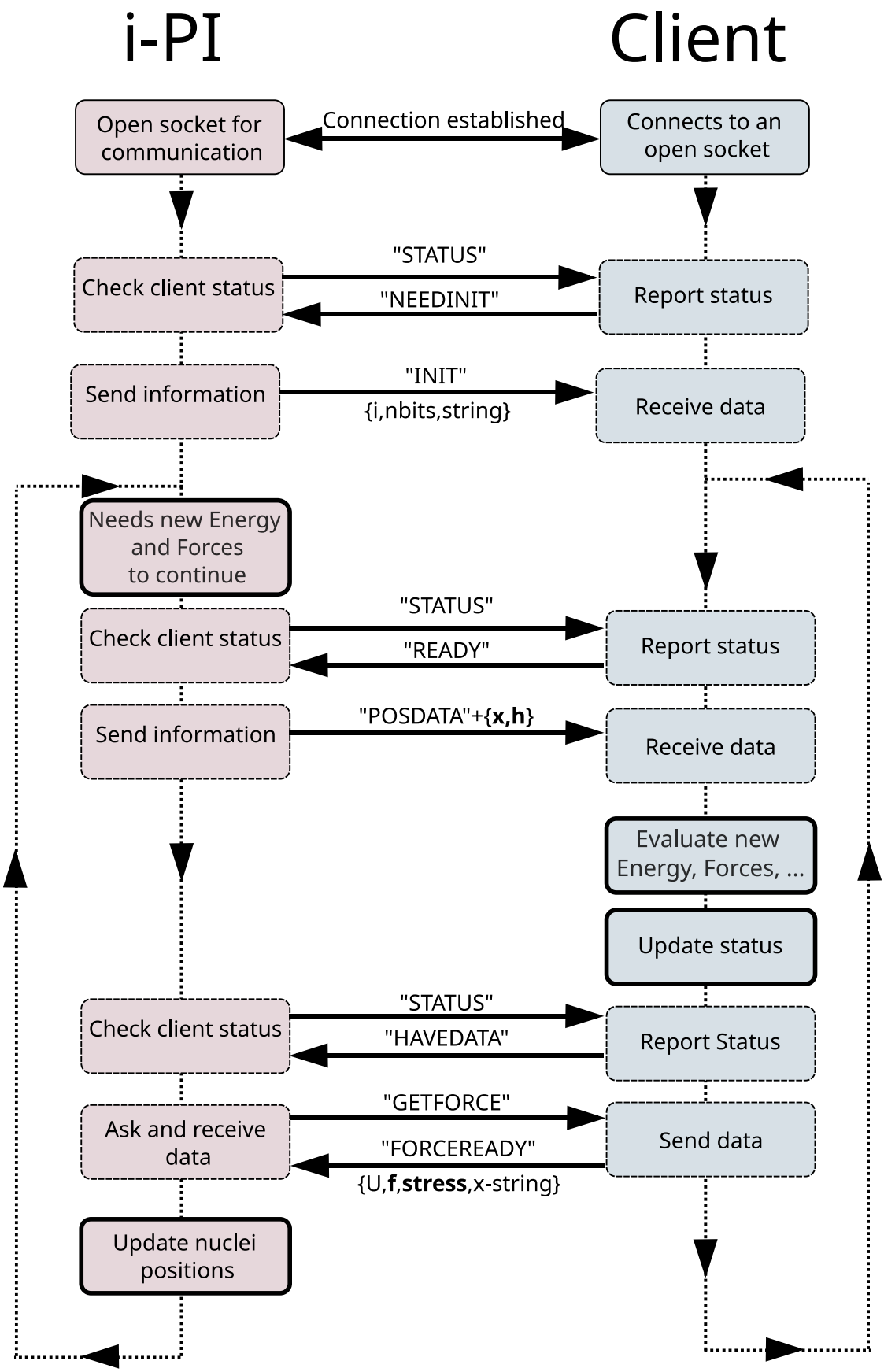

Fig. 5: A schematic simplified representation of the communication protocol. We note that most of the clients do not make use of the 'NEEDINIT' option, The communication is *asynchronous* but we have omitted the 'waiting' blocks for simplicity.

10.3 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 in the xml input file and in the input of the client. By default this file is created based on the address tag, with a */tmp/ipi_* prefix. This can be overridden setting the "sockets_prefix" attribute for the *simulation* tag in the input file, or on the command-line using the *-S* option. Note that several clients do not support changing the default prefix.

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 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 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.

10.4 Running i-PI over the network

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

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.

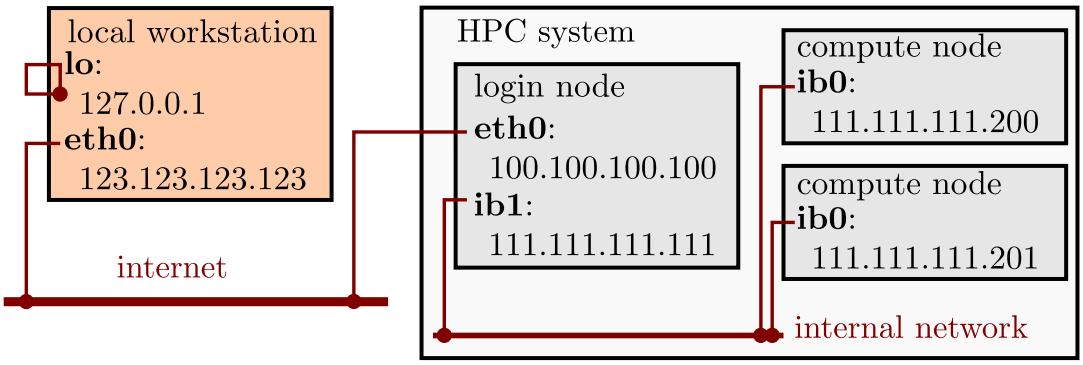

Fig. 6: A schematic representation of the network layout one typically finds when running i-PI and the clients on a HPC system and/or on a local workstation.

The figure 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 panel b of the *i-PI running figure*, then on the HPC system described in this scheme 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 (panel c of *this figure*) 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.

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:

from the job script that launches the client. For instance, with the network layout of *this figure*, 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:

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.

11 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 about previous experience with 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 "demos/para-h2-tutorial/" directory.

The chosen problem is that of a small *NVT* simulation of para-hydrogen, using the Silvera-Goldman potential [SG78]. We will take (N,P,T) = (108, 0, 25 K).

11.1 Part 1 - *NVT* Equilibration run

In this tutorial, we consider the problem of how to use i-PI to run a *NVT* 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 "i-pi-driver" file compiling the code in the "drivers" directory, using the UNIX utility make.

We will first go over the creation of the i-PI input file and afterwards the commands of how to run i-PI and the client code. If you want to skip the first section, against our recommendation if you are a newcomer you can use the input file provided in the "tutorial-1" directory and skip directly to *Running the simulation*

This client code can be used for several different problems (see *Built-in, fortran client*), 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 Bohr.

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 tag, as shown below:

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

xml syntax requires a set of hierarchically 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 input tags, more information on xml syntax can be found in *Input file format and structure*, and possible errors which can occur if the input file is not well-formed can be found in the troubleshooting section.

For the sake of this first tutorial however, we will simply discuss those tags which are needed for a single *NVT* equilibration run. The most important tags are *initialize*, *ensemble*, *motion*, *dynamics*, "total_steps", and *forces*. These correspond to the tag to initialize the atom configurations (initialize), the tag *ensemble* defines the properties of the statistical ensemble considered (like temperature and pressure). While *motion* contains everything regarding the motion of the atoms in general and allows to specify which type of computation is expected (dynamics, geometry optimization, etc.), the tag *dynamics* contains all the specifics relative to the molecular dynamics (such as the barostat, thermostat and the time step, whose total number is defined in the *total_step* tags.). In theory it would be possible to join many different forces into the tag *forces*. Each of the forces needs a force field socket (*ffsocket* tag) that specify who is in charge of computing the forces and sending them to i-PI. Note that the attribute "forcefield" of the *force* tag must correspond to the attribute "name" of one of the *ffsocket* tag. All the tags that define the actual simulation must be within the *system* block. We will also discuss *Output file tags*, which is used to define what output data is generated by the code.

After this short introduction, let's get down to work. We start with an input file that simply looks like this:

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

and in the next subsections, we describe and show code snippets for all the other sections.

Initializing the configurations

First, we consider the *initialize* tag within the *system* block. 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. To run classical molecular dynamics, just set this value to one (nbeads=1).

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
# CELL(abcABC): a b c A B C cell{angstrom} postions{angstrom}
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. The second line is the comment line, and can also contain the cell parameters (a,b, and c are the lattice vectors, and A, B,C the angles) In the example above we use the syntax "cell{angstrom}" and "postions{angstrom}" to indicate the cell parameters and the position coordinates are provided in angstroms.

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 [SG78] and [MM05].

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> position{angstrom} cell{angstrom}
CRYST1 a b c  A B C P 1 1
ATOM 1 atom1 1 1 x1 y1 z1 0.00 0.00 0
ATOM 2 atom2 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, atom1 and atom2 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 https://www.wwpdb.org/documentation/file-format

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 *Overriding default units*.

The "units" attribute is now deprecated and will be removed in the future version of i-PI. The alternative, and the only one available in the future, is to specify the units within the comment line of the xyz or the TITLE line of the pdb

formats (as shown in the examples above). It is also important to put the units only in one place: if the units are present in both, the configuration file with the tag "units" and in the input files (xyz or pdb) the conversion will be applied twice.

A further comment on the cell units and parameters

It is important to note that the units of the cell parameters and the units of the content of the files are specified separately ("positionunits" specify the units of the data and "cellunits" specify the units of the cell). This is necessary because the xyz format can be used to also store quantities which have a different dimension than length (velocities, forces, etc.). Even the cell parameters can now be specified directly within the xyz format. The comment line is parsed looking for a cell specification in the following format:

- "CELL{abcABC}:" followed by six float numbers.
- "CELL{H}:" followed by nine float numbers.
- "CELL{GENH}:" followed by nine float numbers.

The "CELL{abcABC}" must be followed by the length of the vector cell and the three angles between them (as in the CRYST1 field of the pdb format -see above-). The other two must be followed by nine floats specifying, respectively, all the values of the cell matrix (flattened) or all the value of the inverse of the cell matrix (flattened).

Since the units are already specified into the xyz and pdb files, the config file will contain:

If the cell parameters are not specified in the xyz file, then, in the configuration file we must specify them separately. To initialize just the cell parameters, we use the *cell* tag. 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>
```

If the xyz already contains the cell parameters, i-PI will use those which are read the last in the config file (if the "cell" tag follows the "file" specification then the cell parameters are those defined in the "cell" tag. If, otherwise, the "cell" tag compares in the config file before the "file" specification, then the cell parameters of the xyz file are used).

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:

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

The pdb file is specified in a similar way, except that no *cell* tag needs to be specified and the "mode" tag should be set to "pdb" (the units should be specified into the pdb file as shown in the example above):

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

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. With this, the final *initialize* section is:

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

Generating the correct dynamics

We continue within the *system* block and consider the *motion* tag, which determines the computation i-pi will perform. Since we wish to run molecular dynamics, the attribute "mode" of the "motion" tag must be equal to "dynamics". The details of the dynamics integration are given within *dynamics*. 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:

```
<system>
  ...
  <motion mode='dynamics'>
    <dynamics mode='nvt'> ... </dynamics>
  </motion>
</system>
```

This defines the computation that will be performed. 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:

```
<system>
  ...
  <dynamics mode='nvt'>
       ...
       <timestep units='femtosecond'> 1 </timestep>
  </dynamics>
</system>
```

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 [CPMM10], 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:

```
<system>
  ...
 <dynamics mode='nvt'>
     <thermostat mode='pile_g'>
        <tau units='femtosecond'> 25 </tau>
     </thermostat>
     <timestep units='femtosecond'> 1 </timestep>
  </dynamics>
</system>
```

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>
```

The temperature must be specified within the *ensemble*:

```
<system>
    ...
    <ensemble>
        <temperature units='kelvin'> 300 </temperature>
    </ensemble>
    ...
</system>
```

To recap, at this point the input file looks as follows

```
<simulation verbosity='high'>
   <total_steps> 5000 </total_steps>
   <system>
     <initialize nbeads='4'>
       <file mode='pdb'> our_ref.pdb </file>
       <velocities mode='thermal' units='kelvin'> 25 </velocities>
     </initialize>
     <motion mode='dynamics'>
       <dynamics mode='nvt'>
         <thermostat mode='pile_g'>
           <tau units='femtosecond'> 25 </tau>
         </thermostat>
         <timestep units='femtosecond'> 1 </timestep>
       </dynamics>
     </motion>
   </system>
   <ensemble>
       <temperature units='kelvin'> 25 </temperature>
   </ensemble>
</simulation>
```

Please make sure you understand all the lines in the input file before continuing.

Defining the forces

We continue within the the *system* block, and now consider the *forces* tag that defines each of (the possibly many) components of the forces. Within this tag, the user can specify many different "force" tags and the final force will be the sum of the contribution from each "force" tag. In this simple example, however, we consider only one force component, and the corresponding section of the input reads

```
<system>
  ...
  <forces>
     <force forcefield='driver'> </force>
  </forces>
<system>
```

The attribute "forcefield" of the tag *force* is simply a label that allows i-PI to match that particular force component with the corresponding server socket. Note that this apparent unnecessary-complicated syntax makes possible complex setups required by more advanced simulations.

Creating the server socket

Next let us consider the *ffsocket* which deals with communication with the client codes. In this example, we only need to specify a single *ffsocket* tag:

```
<ffsocket> ... </ffsocket>
```

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 *ffsocket* section:

```
<ffsocket mode="inet" name="driver">
    <address> localhost </address>
    <port> 31415 </port>
</ffsocket>
```

and by adding it to the previous sections we have

```
<simulation verbosity='high'>
   <total_steps> 5000 </total_steps>
   <ffsocket mode="inet" name="driver">
     <address> localhost </address>
     <port> 31415 </port>
   </ffsocket>
   <system>
     <initialize nbeads='4'>
       <file mode='pdb'> our_ref.pdb </file>
       <velocities mode='thermal' units='kelvin'> 25 </velocities>
     </initialize>
     <motion mode='dynamics'>
       <dynamics mode='nvt'>
         <thermostat mode='pile_g'>
           <tau units='femtosecond'> 25 </tau>
```

(continues on next page)

(continued from previous page)

```
        </thermostat>
        <timestep units='femtosecond'> 1 </timestep>
      </dynamics>
    </motion>
  </system>
  <ensemble>
      <temperature units='kelvin'> 25 </temperature>
  </ensemble>
</simulation>
```

Note that the *ffsocket* section lives outside the *system* block.

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.

First, we have the standard output. For this output, the amount of data 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 wall time required for the last step at each step and information related to the socket communication.

Second, we have output written to file(s). The content of such output(s) 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 *Output files*, 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's analyze the *output* section as provided in "tutorial-1/tutorial-1.xml" which reads

```
<output prefix='tut1'>
  <checkpoint filename='checkpoint' stride='1000' overwrite='True'> </checkpoint>
  <properties filename='md' stride='1'>
      [step, time{picosecond}, conserved{kelvin}, temperature{kelvin},
```

(continues on next page)

(continued from previous page)

```
      potential{kelvin}, kinetic_cv{kelvin}]
  </properties>
  <trajectory filename='pos' stride='100' format='pdb' cell_units='angstrom'>
          positions{angstrom}
  </trajectory>
  <trajectory filename='forces' stride='100'> forces  </trajectory>
</output>
```

This setup will create 11 files:

checkpoint file: "tut1.checkpoint"

properties file: "tut1.md"

position trajectory files (1 file per bead): "tut1.pos_0.pdb", "tut1.pos_1.pdb", "tut1.pos_2.pdb", and "tut1.pos_3.pdb"

forces trajectory files (1 file per bead): "tut1.forces_0.xyz", "tut1.forces_2.xyz", "tut1.forces_1.xyz", and "tut1.forces_3.xyz"

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" attribute sets 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 the previous snippet we set it to "tut1":

As for the input parameters, the default units are always atomic units. However, this can be modified by the user by specifying an appropriate unit in curly braces after the name of the property or trajectory of interest. In the previous snippet, we have for example set the temperature units to kelvin and position coordinates to angstroms

When using 'pdb' format, it is important to add the "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. 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 at 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:

```
<output prefix='tut1'>
  ...
  <properties filename='force' stride='20'> [atom_f{piconewton}(atom=0;bead=0)] </
↪properties>
</output>
```

This will print out the force vector acting on bead 0 of atom 0.

Input file tutorial-1

If you reached this point, you have been able to specify the input file from scratch, well done! Hopefully, this has helped you to understand the most important syntaxes of the i-PI input file. However, we recommend that next time you use one of the many input files provided within the "examples" and "demos" folder.

For the sake of completeness, we copy the input file we have just created:

```
<simulation verbosity='high'>
  <output prefix='tut1'>
    <checkpoint filename='checkpoint' stride='1000' overwrite='True'> </checkpoint>
    <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>
    <trajectory filename='forces' stride='100'> forces  </trajectory>
    <properties filename='force' stride='20'> [atom_f{piconewton}(atom=0;bead=0)] </
↪properties>
  </output>
  <total_steps> 5000 </total_steps>
  <ffsocket mode='inet' name='driver'>
    <address>localhost</address>
    <port> 31415 </port>
  </ffsocket>
  <system>
    <initialize nbeads='4'>
      <file mode='pdb'> our_ref.pdb </file>
      <velocities mode='thermal' units='kelvin'> 25 </velocities>
    </initialize>
    <forces>
      <force forcefield='driver'> </force>
    </forces>
    <ensemble>
      <temperature units='kelvin'> 25 </temperature>
    </ensemble>
    <motion mode='dynamics'>
      <dynamics mode='nvt'>
        <thermostat mode='pile_g'>
          <tau units='femtosecond'> 25 </tau>
        </thermostat>
        <timestep units='femtosecond'> 1 </timestep>
      </dynamics>
    </motion>
  </system>
</simulation>
```

Running the simulation

If you haven't already, please check out the *Installing i-PI* section of this documentation to set up i-PI.

In the following, we assume that you are in the "demos/para-h2-tutorial/tutorial-1" folder. Now that we have a valid input file, we can run the first part of the tutorial using:

```
> 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:

```
> i-pi-driver -m sg -a localhost -o 15 -p 31415
```

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

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

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", "tut1.forces_1.xyz", "tut1.forces_2.xyz", "tut1.forces_3.xyz", "tut1.forces_4.xyz", 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 "i-pi-driver" 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 considered here with the potential given is a liquid [SG78], 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 -persist  -e "plot 'tut1.md' u 1:3" # step vs conserved quantity
> gnuplot -persist  -e "plot 'tut1.md' u 1:4" # step vs temperature
> gnuplot -persist  -e "plot 'tut1.md' u 1:5" # step vs potential energy
> gnuplot -persist  -e "plot 'tut1.md' u 1:6" # step vs 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 continue with the second part and start the *NPT* run.

11.2 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 $P = 0$.

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 gives:

```
<simulation verbosity='high'>
   <output prefix='tut2a'>
     <properties filename='md' stride='1'>
           [ step, time{picosecond}, conserved{kelvin}, temperature{kelvin}, potential
↪{kelvin},
             kinetic_cv{kelvin}, pressure_cv{megapascal}, volume ]
     </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>
   <total_steps>100000</total_steps>
  ...
 <simulation verbosity='high'>
```

Finally, we must change the *ensemble* and *dynamics* the tags so that the correct ensemble is sampled. The first thing that must be done is adding a "pressure" tag in the ensemble:

```
<ensemble>
   <pressure> 0 </pressure>
    ...
</ensemble>
```

Then, we must also specify the constant pressure algorithm, using the tag *barostat* within the dynamics environment. Do not forget to change the mode attribute of the dynamics from "nvt" to "npt". This example uses a stochastic barostat to apply pressure to an isotropic system, which can be specified with the option "isotropic". See the documentation of the *barostat* object and the examples to see how to apply an anisotropic stress, or to allow for cell shape fluctuations.

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:

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

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

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

with the rest of the *ensemble* and *dynamics* tags being the same as before. Note that in a *NPT* simulation, we have two

thermostats, one applied to the nuclear degrees of freedom and one applied to the volume degrees of freedom.

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 tag *initialize* section with:

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

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".

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.

11.3 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 [MHT99].

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 *ffsocket* tag to "unix":

```
<ffsocket mode='unix' name="driver"> ... </ffsocket>
```

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

```
> i-pi-driver -u -m sg -a 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 > i-pi-driver -u -m sg -a 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.

12 Tutorials, recipes, and on-line resources

12.1 Massive Open Online Course (MOOC)

The theory behind classical and path-integral methods for structure and dynamics along with practical exercises can be found in an online course freely available to everyone, and accessible from this link.

12.2 i-PI resources

For more information about i-PI and to download the source code go to http://ipi-code.org/.

In http://gle4md.org/ one can also obtain colored-noise parameters to run Path Integral with Generalized Langevin Equation thermostat (PI+GLE/PIGLET) calculations.

12.3 Examples and demos

The *examples* and *demos* folders in the i-PI source code contain inputs for many different types of calculations based on i-PI. Examples are typically minimal use-cases of specific features, examples of client codes, or setups for high performance computing, while demos are more structured, tutorial-like examples that show how to realize more complex setups, and also provide a brief discussion of the underlying algorithms.

To run these examples, you should typically start i-PI, redirecting the output to a log file, and then run a couple of instances of the driver code. The progress of the wrapper is followed by monitoring the log file with the *tail* Linux command.

Optionally, you can make a copy of the directory with the example somewhere else if you want to keep the i-PI directory clean. For example:

```
bash
cd demos/para-h2-tutorial/tutorial-1/
i-pi tutorial-1.xml > log &
i-pi-driver -a localhost -p 31415 -m sg -o 15 &
i-pi-driver -a localhost -p 31415 -m sg -o 15 &
tail -f log
```

12.4 Tutorials

A simple tutorial on how to run i-PI can be found in the documentation: a simple tutorial.

You can try a set of guided examples that demonstrate some more advanced features of i-PI. These Tutorials, from the 2021 CECAM Flagship school, are available here: i-PI 2021 tutorials.

Tutorials from the 2023 CECAM Flagship school are available here: i-PI 2023 tutorials.

Tutorial from one of our developers on using machine learning interatomic potentials with i-PI are available here: MLIPs-with-i-PI.

Note that these examples use some features (such as qTIP4P/F water and the Zundel cation potentials in the driver code, and optionally also CP2K). Instructions on how to install and run these codes with i-PI are contained in the tutorial.

12.5 Atomistic recipes

Several examples of usage of i-PI to perform advanced molecular simulations can be found in the i-PI chapter of the atomistic cookbook. These include an introduction to path integral simulations, a discussion of how to compute quantum heat capacities, and an example of path integral metadynamics.

13 Frequently asked questions

13.1 Where to ask for help?

There is distinction between questions and bugs. If your question is related to running i-PI, using or understanding its features (also if you think that something is not documented enough), the best way to ask is to post your question on the i-PI forum: https://groups.google.com/g/ipi-users. In the forum, everything discussed will be available to other members of the community in the future!

If you are seeing an actual bug, please consider reporting an issue on Github: https://github.com/i-pi/i-pi/issues.

13.2 Which units does i-PI use?

Atomic units are used everywhere inside the code. The exact values and more details are given in section *Internal units and conventions*. For I/O operations, various units are accepted. The XML tags responsible for physical settings accept the attribute `units=<unit>`. If nothing is specified, i-PI will assume atomic units. Many of the provided examples have these tags, and the reference section of the manual contains a full list. Another way to use units is related to input `.xyz` files. i-PI can accept declarations of these units in the `.xyz` comment line. One example is:

```
# CELL(abcABC): a b c alpha beta gamma cell{units} positions/forces/velocities{units}
```

Another place worth looking at is `<i-pi-root>/ipi/utils/units.py`, which contains definitions and values for non-default units, such as `electronvolt`, `inversecm` etc.

13.3 How to build i-PI?

i-PI is a Python code, and as such, strictly speaking, does not need to be compiled and installed. `<i-pi-root>/bin/i-pi` file is an executable. `source <i-pi-root>/env.sh` ensures that necessary paths are added to PATH, PYTHONPATH etc. Note that this option will *soon be deprecated* and the usage of i-PI as a package (see below) is recommended.

It is 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. Detailed explanation can be found in the manual.

In addition, i-PI can be installed through `pip` with the command

```
$ pip install -U ipi
```

And to test it (assuming all required packages are also installed)

```
$ i-pi-tests
```

13.4 How to run i-PI with the client code <name>?

i-PI communicates with electronic structure and MD codes via socket communication protocol. For many popular codes (CP2K, FHI-aims, LAMMPS, Quantum ESPRESSO, VASP, etc.) we provide examples in `<i-pi-root>/examples/clients` folder.

Another way of connecting to client codes is using the ASE client. This way you get access to wide variety of codes that are connected to ASE, but for some of them current implementation requires restarting the client code after each step, which may lead to a significant overhead in case of electronic structure calculations. We recommend using the socket connection from the client code to ASE and then from ASE to i-PI, when possible. An example of a "double-socket" setup can be found in:

`https://github.com/i-pi/i-pi/tree/main/examples/clients/ase_client`

13.5 How to run i-PI with VASP?

VASP is not shipped with support to i-PI. The way to get simulations running is to patch the VASP source code. This patch aims to add socket support and we provide patches in `<i-pi-root>/examples/clients/vasp`. Check the README file in that folder for detailed instructions on how to patch VASP.

Note that the patches are version numbered, and we cannot guarantee that a given patch will work on a different VASP version than the one it was created for. We welcome contributions on the updates to these patches to keep up-to-date to the current VASP releases. The most recent VASP version we support by explicit path (as of Jan. 2025) is 6.3.0. The general script `patch_vasp.py` by T. Tian in `<i-pi-root>/examples/clients/vasp` can work for more recent, e.g. 6.4.X, versions, but we recommend testing.

13.6 How to run i-PI on a cluster?

There are different ways of running i-PI on HPC systems, described in details in the manual. Setups differ a lot depending on the type of calculation and the architecture of the system. A few examples based on the Slurm submission system can be taken as the starting point, and can be found in `examples/hpc_scripts`. Other setups are possible, please consult the manual.

13.7 How to setup colored-noise thermostats with meaningful parameters?

If you have an application that will profit from colored noise thermostats, you can download the parameters from http://gle4md.org. This website contains sets of input parameters optimized for a variety of conditions. You can choose i-PI syntax, such that you will only have to copy-paste that section inside the i-PI input file. Also the `<i-pi-root>/examples` (especially withing `lammps`) folder demonstrates the syntax for various thermostats.

13.8 How to perform geometry optimization?

Several examples of geometry optimizaion, including fixing certain atoms, are present in the `<i-pi-root>/examples` folder. The folders with names *geop* correspond to geometry optimization. Note that the optimization algorithms that involve line search have two sets of convergence parameters: one for line search and another for the main algorithm.

14 Troubleshooting

14.1 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 *Input file format and structure* 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.

14.2 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://lab-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.

14.3 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.

14.4 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.

14.5 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.

14.6 I-PI is slow!

i-PI is not designed to be highly efficient, but it should not be the bottleneck in your calculations. Most of the time, if you see a major slow-down, there is a problem with your setup. Here are some tricks you can try.

- *use a UNIX domain socket*: if you run on a single node, it is much faster to use *mode="unix"* in your <*ffsocket*> classes. This uses a shared-memory communication process, that avoids much of the network latency of a TCP/IP socket.
- *reduce latency*: if you have a VERY fast forcefield (with force evaluation below ~1ms) it might help to set the <*latency*> parameter of <*ffsocket*> to a small value, 1e-4s or less

- *reduce I/O*: outputting hundreds of beads configurations at each time step is going to be slow in any scenario, but particularly so when using text files and Python. Reduce the output frequency using a larger *stride*, and/or *flush* less often.
- *reduce the stride of internal checkpoints*: to guarantee that soft exits leave the simulation in a state that can be restarted safely, i-PI stores the internal state of the simulation at each step. Particularly for complicated setups, the overhead can be substantial. You can reduce the frequency by which the internal state is stored using the *safe_stride* attribute in the <*simulation*> tag. Note that doing so increases the risk that the RESTART file saved upon soft exit will be inconsistent with the sate of the outputs, so that restarting a simulaiton will leave broken or discontinuous output files.
- *profile i-PI*: if you still think i-PI is being unreasonably slow, you can contact the developers - your setup might have revealed some kind of bottleneck. It will help us if you can also generate a profiler output from your run (possibly with only a small number of steps). You can generate a profiler log by running i-PI with *-p* option, e.g. *i-pi -p input.xml*. You will need to install the *yappi* profiler.

15 Contributing

i-PI is an open source project, and everyone is welcome to contribute with bug fixes, documentation, examples, new features, regression tests, etc. Any contribution from users or developers is very much welcome! These contributions may be full new methods/algorithms that you have developed or that you just would like to add to i-PI, or simple improvements (that would make your life easier) regarding the code itself, the documentation, or analysis scripts.

15.1 Useful resources

For more information on i-PI in general please consult the following places:

- i-PI homepage: http://ipi-code.org
- i-PI forum: https://groups.google.com/g/ipi-users
- i-PI github repo, including issues page: https://github.com/i-pi/i-pi

15.2 What we expect from a contribution

To start contributing, note the following general guidelines:

- Your contribution should be based on the main branch of the repository
- We expect you to fork the repository and make all the changes in your fork
- When done, you can open a pull request (PR) to our repository, which will be reviewed.

For a PR to be issued, we will expect that:

- You have produced code that is compliant with our formatting requirements (*black* and *flake8*, as explained in the README here: https://github.com/i-pi/i-pi)
- You have written understandable docstrings for all functions and classes you have created
- For all contributions (bug fixes, new functionalities, etc.), ensure that the continuous integration tests all pass.
- In case of a new functionality:
  - You have added an example to the *examples* folder that showcases it. The inputs should be a simple but meaningful example of the functionality.
  - You have added a regression test to *i-pi/ipi_tests/regression_tests* folder (more on this below).

15.3 Creating a regression test

If you code up a new functionality, it is best that it goes directly to the regression test infrastructure, so that we can ensure that it does not break with future code restructuring or the addition of new features in the future. The information below can also be found in the README file in the *i-pi/ipi_tests/regression_tests* folder:

To add a new regression test please provide:

- input.xml (and all the files required by it)
- test_settings.dat This file specifies the driver_model and flags that will be used when running the driver to perform the regression test. For examples including the usage of option flags, please see:

  tests/NVE/NVE_1/harmonic_python/test_settings.dat tests/NVE/NVE_1/harmonic/test_settings.dat

  If your regtest uses a new driver mode, please add it to the `fortran_driver_models` list in the *i-pi/ipi_tests/test_tools.py* file.
- file_to_check.txt specifying the files serving as reference with their respective filenames. For an existing example, please see:

  tests/INSTANTON/100K/files_to_check.txt
- reference files that are listed in the files_to_check.txt file. Currently, the available formats are .xyz and numpy-accessible which should be specified with the keywords 'xyz' and 'numpy'.

Important points to note:

- In the case of multiple drivers needed for a simulation, the *test_settings.dat* file has to include a line for each driver by specifying as many lines as needed of *driver_model xxxx* in the order they should be assigned to the sockets. That is followed by lines decribing the corresponding flags of each driver.
- The extension **.out* appears in the *.gitignore* file. This means that the 'ref_simulation.out' file has to be added manually to your commit. You can do that by typing: "git add -f ref_simulation.out"

If you are developing something new and you are unsure about how to proceed, we encourage opening an issue with your question. Depending on the development, we can open a channel of discussion with the core developers (Slack) in order to see how to best handle your case!

15.4 Adding features involving a client code

If your new development in i-PI is directly related to a specific client code or if there is the need of coding a new native client in i-PI there can be some further adjustments to do. We very much welcome new interfaces and we will be happy to answer your questions.

If you want to enable the communication of a new client code with i-PI, it is not difficult: Please check an example of how it was done in `fortran` and `python` in the *drivers* folder in the repository. It is especially simple to add a new potential energy that is evaluated in Python: it is sufficient to add a file in the *ipi/pes* folder, specifying *__DRIVER_NAME__* (a string that will be used to refer to the PES from the i-PI input or the command line) and *__DRIVER_CLASS__*, the name of the actual class, that should provide, directly or through inheritance, a *__call__(self, cell, pos)* function and return a tuple with *(potential, forces, virial, extras)*. See any of the existing PES files to use as templates - it is particularly simple to create a class that piggybacs on an existing ASE-style calculator.

15.5 Getting recognition for your contribution

We recognize that adding new features or enhancements to any project is a task that needs more recognition. If you want your contribution advertised, we are happy to do so through our webpage. Just let us know and we will add your name, a short description, and links to any publications here.

15.6 Contributing with a bug report

Bug reports are a very welcome feedback! These should be posted in the i-PI Github repository (issues tab). Please include inputs and outputs that allow us to reproduce the bug, and a short description of the bug you are encountering.

15.7 Questions related to i-PI usage

Questions related to usage or general questions are best posted in our forum (see link above to google groups). If we establish that there is the need to open a Github issue related to the question, we will do so.

16 Bibliography

References

[CMM14] Michele Ceriotti, Joshua More, and David E. Manolopoulos. i-PI: A Python interface for ab initio path integral molecular dynamics simulations. *Comp. Phys. Comm.*, 185:1019–1026, 2014.

[CPMM10] Michele Ceriotti, Michele Parrinello, Thomas E Markland, and David E Manolopoulos. Efficient stochastic thermostatting of path integral molecular dynamics. *J. Chem. Phys.*, 133:124104, 2010.

[HMM09] Scott Habershon, Thomas E Markland, and David E Manolopoulos. Competing quantum effects in the dynamics of a flexible water model. *J. Chem. Phys.*, 131:24501, 2009.

[KRM+19] Venkat Kapil, Mariana Rossi, Ondrej Marsalek, Riccardo Petraglia, Yair Litman, Thomas Spura, Bingqing Cheng, Alice Cuzzocrea, Robert H Meißner, David M Wilkins, Benjamin A Helfrecht, Przemysław Juda, Sébastien P Bienvenue, Wei Fang, Jan Kessler, Igor Poltavsky, Steven Vandenbrande, Jelle Wieme, Clemence Corminboeuf, Thomas D Kühne, David E Manolopoulos, Thomas E Markland, Jeremy O Richardson, Alexandre Tkatchenko, Gareth A Tribello, Veronique Van Speybroeck, and Michele Ceriotti. i-PI 2.0: A universal force engine for advanced molecular simulations. *Computer Physics Communications*, 236:214–223, March 2019.

[LKF+24] Yair Litman, Venkat Kapil, Yotam M. Y. Feldman, Davide Tisi, Tomislav Begušić, Karen Fidanyan, Guillaume Fraux, Jacob Higer, Matthias Kellner, Tao E. Li, Eszter S. Pós, Elia Stocco, George Trenins, Barak Hirshberg, Mariana Rossi, and Michele Ceriotti. I-pi 3.0: a flexible and efficient framework for advanced atomistic simulations. *J. Chem. Phys.*, 161:062505, 2024.

[MM08] Thomas E Markland and David E Manolopoulos. An efficient ring polymer contraction scheme for imaginary time path integral simulations. *J. Chem. Phys.*, 129:024105, 2008.

[MHT99] Glenn J Martyna, Adam Hughes, and Mark E Tuckerman. Molecular dynamics algorithms for path integrals at constant pressure. *J. Chem. Phys.*, 110:3275, 1999.

[MM05] Thomas F Miller and David E Manolopoulos. Quantum diffusion in liquid para-hydrogen from ring-polymer molecular dynamics. *J. Chem. Phys.*, 122:184503, 2005.

[SG78] Isaac F. Silvera and Victor V. Goldman. The isotropic intermolecular potential for H2 and D2 in the solid and gas phases. *J. Chem. Phys.*, 69:4209, 1978.