atsim_potentials Documentation

M.J.D. Rushton

May 20, 2021

Contents

Classical simulation codes typically contain a good selection of analytical forms for describing atomic interactions. Sometimes however, you may need to use a potential that is not directly supported by the code. Luckily, most simulation codes allow you to provide tabulated potentials in which energies and forces, for a range of interatomic separations, are pre-calculated and specified as rows within a text file. The `atsim.potentials` package provides python modules to make the specification and tabulation of pair- and many-body potentials straightforward and consistent.

CHAPTER 1

Features

- **Pair-Potential Tabulation:** Effective pair-potentials can be tabulated for multiple codes including:
  - GULP
  - LAMMPS
  - DL_POLY
- **Many-Bodied Potentials:** Embedded Atom Model (EAM) potential tabulation is supported in the following formats:
  - DYNAMO - as used by LAMMPS and several other codes:
    - Support for LAMMPS `eam`, `eam/fs`, `eam/alloy`, `adp` pair-styles.
  - DL_POLY: write `TABEAM` formatted files.
- **No programming required:** `atsim.potentials` can be driven using its own potential definition format. Using simple configuration files complex models can be defined and tabulated without requiring any programming experience.
- **Potential forms:** comes pre-loaded with a wide range of common-potential types.
- **Potential splining:** join different potentials together with splines.
- **Flexible:** the `atsim.potentials` potential definition format allows the use of arbitrary mathematical formulae to define new potential functions. If this isn't sufficient it also provides a powerful Python API which should allow most tabulation tasks to be achieved.

CHAPTER 2

Contents

2.1 Quick-Start

The following provides a quick example of how a pair potential model can be defined and then tabulated for different simulation codes. For more advanced features such as EAM potentials, defining custom potential forms, splining or using the python interface, please see *User Guide*.

In this example we will define Basak's [Basak2003] potential model for UO_2. This has been chosen as it presents an issue that is often solved by using tabulated potentials. The U-O interaction combines Buckingham and Morse potential forms. Although most simulation codes natively provide these as analytical forms, some don't then allow them to be used in combination for the same pair-interaction. As a result, it becomes necessary to combine them externally and feed them into the code in tabulated form. This tutorial will show how this can be achieved using `atsim.potentials`.

1. **Installation** (see *Installation* for more detail):

   ```
   pip install atsim.potentials
   ```

2. **Write input file**

   - In the Basak model [Basak2003] the O-O and U-U interactions are defined entirely using the Buckingham potential form. Whilst the O-U pair is the sum of a Buckingham and Morse potential (see *below* for full details of the potential model). Both forms are provided by `atsim.potentials` and are specified as:

     $$\begin{aligned} V_{\text{Buck}}^{\text{atsim}}(r_{ij}) &= A_{ij} \exp\left(-\frac{r_{ij}}{\rho_{ij}}\right) - \frac{C_{ij}}{r_{ij}^6} \\ V_{\text{Morse}}^{\text{atsim}}(r_{ij}) &= D_{ij}\left[\exp\{-2\gamma_{ij}(r_{ij}-r_{ij}^*\} - 2\exp\{-\gamma_{ij}(r_{ij}-r_{ij}^*\}\right] \end{aligned} \tag{2.1}$$

   - The Buckingham potential is parametrised using values for A_ij, ρ_{ij} and C_{ij}, specific to each species pair.

   - The Morse potential takes D_{ij}, $\gamma_{ij}$and r_{ij}^*.

   - Parameters for the Basak model are given in Table 2.1.

Table 2.1: Basak parameters for use with standard form of Buckingham and Morse potentials

Parameters	O-O	U-U	O-U
A_{ij}/eV	1633.010243	294.640906	693.650934
ρ_{ij}/Å	0.327022	0.327022	0.327022
C_{ij}/eV^6	3.948787	0.0	0.0
D_{ij}/eV	NA	NA	0.577190
$\gamma_{ij}/^{-1}$	NA	NA	1.650000
r_{ij}^*/Å	NA	NA	2.369000

- The *potable* tool is used to generate table files.
- It accepts input in a straightforward format, reminiscent of `.ini` configuration files.
- The potential parameters from Table 2.1 have been transferred into a file suitable for *potable*: `basak.aspot`:

```
[Tabulation]
target :  DL_POLY
cutoff : 6.5
nr : 652

[Pair]
O-O = as.buck 1633.010242995040 0.327022 3.948787
U-U = as.buck 294.640906285709 0.327022 0.0
O-U = sum(as.buck 693.650933805978 0.327022 0.0,
                  as.morse 1.65 2.369 0.577189831995)
```

3. **Generate tabulated potential**

- Download the `basak.aspot` file and then generate a tabulation in `DL_POLY` format (see *below* for information on other formats) by running this command:

```
potable basak.aspot TABLE
```

- This will generate a tabulation in the file named `TABLE`.

And that's it! The rest of this page now gives details on what you've just done. Read on for more.

2.1.1 What are tabulated potentials?

Pair potentials are functions that relate potential energy to the separation of two interacting particles: $U_{ij}(r_{ij})$. These are typically defined as equations, with a particular analytical form, that can be tailored to the chemistry of a given pair of species through a set of parameters (e.g. A_ij, ρ_{ij} and C_ij in the case of the Buckingham form shown above).

Using these analytical potentials, an entire potential model, comprising of many interactions, can be described in a very compact form. Simulation codes come with large libraries of analytical potential forms, even so, it is impossible for all codes to support all potential forms. As a way of providing flexibility, and as an alternative to requiring users to edit and recompile simulation codes whenever they want to use an unusual form, most support table files.

Tabulated potential approximate a $U_{ij}(r_{ij})$ functions as a series of [separation, potential-energy] points that are stored in a file, readable by the code. For values not stored in the file, the simulation code performs interpolation to obtain energies and forces for in-between values.

Fig. 2.1 shows the process followed in producing one of these tables. The desired analytical potential is defined and energy and forces (the first derivative of energy with respect to separation) are sampled at regular intervals. These samples are then written to a file in the format required by the simulation code.

The aim of `atsim.potentials` is to make this process simple, flexible and transferable across simulation codes.

Fig. 2.1: Tabulated potentials are obtained by sampling a mathematical formula at regular separations.

2.1.2 Potential model

Before moving on to the tabulation procedure, it is useful to understand what it is we're trying to tabulate.

Basak's [Basak2003] model employs the following potential form:

$$V(r_{ij}) = V_{\mathrm{Coul}}(r_{ij}) + V_{\mathrm{Buck}}(r_{ij}) + V_{\mathrm{Morse}}(r_{ij}) \tag{2.2}$$

Where $V(r_{ij})$ is the potential energy of two atoms (i and j) separated by r_{ij}. The first term in eqn. (2.2) defines the long range electrostatic interaction between the two atoms (with charges q_i and q_j):

$$V_{\mathrm{Coul}}(r_{ij}) = \frac{q_i q_j}{4\pi\epsilon_0 r_{ij}} \tag{2.3}$$

Where ϵ_0 is the permittivity of free space.

In a periodic system like UO_2, it is necessary to employ various mathematical tricks to get the Coulomb sum to converge quickly and reliably (see for instance Ewald, cell-multipole or particle-mesh methods). As a result the Coulomb part of a potential model isn't normally included in a tabulation file and therefore it won't appear further in this example.

This leaves the short-range components of $V(r_{ij})$ from eqn. (2.2) for us to bother about, namely $V_{\mathrm{Buck}}(r_{ij})$ and $V_{\mathrm{Morse}}(r_{ij})$. In Basak's [Basak2003] paper these are defined as follows:

$$\begin{aligned} V_{\mathrm{Buck}}(r_{ij}) &= f_0 b_{ij} \exp\left(\frac{a_{ij} - r_{ij}}{b_{ij}}\right) - \frac{C_{ij}}{r_{ij}^6} \\ V_{\mathrm{Morse}}(r_{ij}) &= f_0 d_{ij}\left[\exp\{-2\gamma_{ij}(r_{ij} - r_{ij}^*)\} - 2\exp\{-\gamma_{ij}(r_{ij} - r_{ij}^*)\}\right] \end{aligned} \tag{2.4}$$

The f_0, a_{ij}, b_{ij}, C_{ij}, D_{ij}, γ_{ij} and r_{ij}^* specifying each pair interaction are given in the following table.

Table 2.2: Potential parameters for Basak model taken from original paper [Basak2003] (units have been converted for certain values).

Parameters	O-O	U-U	O-U
a_{ij}/	3.82	3.26	3.54
b_{ij}/	0.327022	0.327022	0.327022
C_{ij}/eV^6	3.948787	0.0	0.0
d_{ij}/r_{ij}^*	NA	NA	13.6765
γ_{ij}/r_{ij}^*	NA	NA	1.65
r_{ij}^*	NA	NA	2.369
f_0/eV^{-1}	0.042203	0.042203	0.042203

The `atsim.potentials` package comes pre-loaded with a large number of potential-forms, so that you don't have to constantly redefine functional forms (see *List of Potential Forms*). In this tutorial you will use the pre-defined versions of the Buckingham and Morse potentials.

If you compare the two definitions of the Buckingham and Morse potentials given in eqns. (2.1) and (2.4) you will see there are some differences. The definitions of the Morse potential are almost identical, allowing γ_{ij} and r_{ij}^* to be used directly with the `atsim` Morse form. D_{ij}, is however obtained as the product of f_0 and d_{ij}

$$D_i j = f_0 d_{ij} \tag{2.5}$$

The Buckingham potential used by Basak has more significant differences to the `atsim` standard. The C_{ij} parameter can be used directly in both versions, however we need to manipulate the function from eqn. (2.4) to show how A_{ij} and ρ_{ij} may be obtained from the original Basak parameter set (Table 2.2).

Let's do this now by rearranging the Buckingham potential from eqn. (2.4):

$$\begin{aligned} V_{\text{Buck}}(r_{ij}) &= f_0 b_{ij} \exp\left(\frac{a_{ij} - r_{ij}}{b_{ij}}\right) - \frac{C_{ij}}{r_{ij}^6} \\ &= f_0 b_{ij} \exp\left(\frac{a_{ij}}{b_{ij}} - \frac{r_{ij}}{b_{ij}}\right) - \frac{C_{ij}}{r_{ij}^6} \\ &= f_0 b_{ij} \frac{\exp\left(\frac{a_{ij}}{b_{ij}}\right)}{\exp\left(\frac{r_{ij}}{b_{ij}}\right)} - \frac{C_{ij}}{r_{ij}^6} \\ &= f_0 b_{ij} \exp\left(\frac{a_{ij}}{b_{ij}}\right) \exp\left(-\frac{r_{ij}}{b_{ij}}\right) - \frac{C_{ij}}{r_{ij}^6} \end{aligned} \tag{2.6}$$

By comparing coefficients between this equation, eqn. (2.6) and the `atsim` form, eqn. (2.1), it becomes obvious that the Basak parameters can be brought into the standard form using these relationships:

$$\begin{aligned} A_{ij} &= f_0 b_{ij} \exp\left(\frac{a_{ij}}{b_{ij}}\right) \\ \rho_{ij} &= b_{ij} \end{aligned} \tag{2.7}$$

Using these relationships together with D_{ij} from eqn. (2.5) the table of potential parameters given above was obtained (Table 2.1).

2.1.3 Writing the potential definition

Using the parameters from Table 2.1, the model was described in the `basak.aspot` file:

```
[Tabulation]
target :  DL_POLY
cutoff : 6.5
nr : 652

[Pair]
O-O = as.buck 1633.010242995040 0.327022 3.948787
U-U = as.buck 294.640906285709 0.327022 0.0
O-U = sum(as.buck 693.650933805978 0.327022 0.0,
                  as.morse 1.65 2.369 0.577189831995)
```

This file contains two configuration blocks:

- `[Tabulation]`: this defines the table's output format
  - `target :  DL_POLY` means a DL_POLY TABLE file will be produced.
  - `cutoff :  6.5` gives the maximum separation to include in the tabulation ($r_{ij} = 6.5$ Å).
  - `nr :  652` the tabulation should contain 652 rows.
- `[Pair]`: this section defines the O-O, U-U and O-U pair interactions:
  - The basic form of each line is `SPECIES_A-SPECIES_B = POTENTIAL_FORM PARAMETER_1 PARAMETER_2 ... PARAMETER_N`
    - Where `SPECIES_A` and `SPECIES_B` define each pair of interacting species.

   * POTENTIAL_FORM is a label identifying the functional form of the interaction. Here the as.buck and as.morse types are used. The as. prefix indicates these are standard forms provided by atsim.potentials (see *List of Potential Forms* for a complete list).

The *Buckingham (buck)* potential takes three parameters, separated by spaces:

```
as.buck A  C
```

And the *Morse (morse)* parameters are:

```
as.morse  r* D
```

By comparing the *input file* and Table 2.1, it should be apparent how the pair-interactions have been parametrised.

The O-U interaction makes use of a *potential modifier* to combine the as.buck and as.morse forms. This is achieved using the *sum()* modifier which adds up all the contributions of the comma separated list of potentials defined inside the brackets.

2.1.4 Generating the tabulation

Save the basak.aspot to your drive. Then generate the TABLE file by executing the *potable* command:

```
potable basak.aspot TABLE
```

This will interpret the potential model described above and write it in DL_POLY format to a file named TABLE.

For an example of how this TABLE file can be used in a DL_POLY simulation see *Using the TABLE File in DL_POLY*.

Using the TABLE File in DL_POLY

A set of DL_POLY files are provided allowing a simple NPT molecular dynamics equilibration simulation to be run using a TABLE file created with atsim.potentials. Copy the files linked from the following table into the same directory as the TABLE file:

File	Description
CONFIG	4×4×4 UO2:sub:2 super-cell containing 768 atoms.
CONTROL	Defines 300K equilibration run under NPT ensemble lasting 10ps.
FIELD	File defining potentials and charges.

- The FIELD file contains the directives relevant to the TABLE file:

```
UO2.cif. Supercell: 4 x 4 x 4
units eV
molecules 1
UO2.cif. Supercell: 4 x 4 x 4
nummols 1
atoms 768
             O    15.999400    -1.200000    512    0
             U   238.028910     2.400000    256    0
finish
vdw 3
O O tab
U U tab
O U tab
CLOSE
```

- The following lines define the atom multiplicity and charges (O=-1.2*e* and U=2.4*e*):

```
nummols 1
atoms 768
             O     15.999400     -1.200000     512    0
             U    238.028910      2.400000     256    0
finish
```

- The `vdw` section states that the O-O, U-U and O-U interactions should be read from the `TABLE` file:

```
vdw 3
O O tab
U U tab
O U tab
CLOSE
```

- Once all the files are in the same directory, the simulation can be started by invoking `DL_POLY`:

```
DLPOLY.Z
```

2.1.5 Specifying other tabulation targets

Once the potential model has been defined, creating tabulations for different codes in different formats is simple.

LAMMPS

To target LAMMPS it is just a case of changing the `target` option in the `[Tabulation]` section of the `basak.aspot` file to `LAMMPS` e.g.

```
[Tabulation]
target : LAMMPS
cutoff : 6.5
nr : 652
```

A `LAMMPS` table file named `Basak.lmptab` would then be generated by re-running *potable*:

```
potable basak.aspot Basak.lmptab
```

An example of how to use this file in LAMMPS is given here: *Using Basak.lmptab in LAMMPS*.

GULP

In the previous section a new tabulation target was specified by editing the `basak.aspot` file. For temporary changes, the *potable* allows configuration options to be overridden from the command line. Let's do this now to make a table file suitable for GULP. This is achieved with the *–override-item* option, like this:

```
potable --override-item Tabulation:target=GULP basak.aspot potentials.lib
```

The `potentials.lib` file can be used with the following GULP input file to run an energy minimisation: `basak.gin`

Using `Basak.lmptab` in LAMMPS

LAMMPS input files are provided for use with the table file:

- `UO2.lmpstruct`: structure file for single UO_2 cell, that can be read with read_data when atom_style `full` is used.
- `equilibrate.lmpin`: input file containing LAMMPS instructions. Performs 10ps of 300K NPT equilibration, creating a 4x4x4 super-cell.

Copy these files into the same directory as `Basak.lmptab`, the simulation can then be run using:

```
lammps -in equilibrate.lmpin -log equilibrate.lmpout -echo both
```

The section of `equilibrate.lmpin` which defines the potential model and makes use of the table file is as follows:

```
variable O equal 1
variable U equal 2

set type $O charge -1.2
set type $U charge 2.4

kspace_style pppm 1.0e-6

pair_style hybrid/overlay coul/long ${SR_CUTOFF} table linear 6500 pppm
pair_coeff * * coul/long
pair_coeff $O $O table Basak.lmptab O-O
pair_coeff $O $U table Basak.lmptab O-U
pair_coeff $U $U table Basak.lmptab U-U
```

Notes:

1. As LAMMPS uses ID numbers to define species the `variable` commands associate:
   - index 1 with variable `$O`
   - index 2 with `$U` to aid readability.
3. The `set type SPECIES_ID charge` lines define the charges of oxygen and uranium.
3. Uses the hybrid/overlay `pair_style` to combine the coul/long and table styles.

   ```
   pair_style hybrid/overlay coul/long ${SR_CUTOFF} table␣
   ↪linear 6500 pppm
   ```

   - The coul/long style is used to calculate electrostatic interactions using the pppm `kspace_style` defined previously.
   - `table linear 6500 pppm`:
     – linear interpolation of table values should be used
     – all 6500 rows of the table are employed
     – corrections appropriate to the pppm `kspace_style` will be applied.
4. Means that electrostatic interactions should be calculated between all pairs of ions.

   ```
   pair_coeff * * coul/long
   ```

5. Each `pair_coeff` reads an interaction from the `Basak.lmptab` file.

```
pair_coeff $O $O table Basak.lmptab O-O
pair_coeff $O $U table Basak.lmptab O-U
pair_coeff $U $U table Basak.lmptab U-U
```

- The general form is:
  - `pair_coeff SPECIES_A SPECIES_A table TABLE_FILENAME TABLE_KEYWORD`
  - Here the `SPECIES_*` use the `$O` and `$U` variables defined earlier.
  - `TABLE_KEYWORD` - the table file contains multiple blocks, each defining a single interaction.
  - The `TABLE_KEYWORD` is the title of the block. The labels are of the form `LABEL_A-LABEL_B` albeit with the species sorted into alphabetical order.

2.2 Installation

2.2.1 Install Using Pip

If you have Pip type the following to install `atsim.potentials`:

```
pip install atsim.potentials
```

2.2.2 Install from Source

The source is hosted on github and can be cloned using git as follows:

```
git clone https://github.com/mjdrushton/atsim-potentials.git
```

alternatively a tarball of the source can be downloaded here

From the source directory install `atsim.potentials` using the following command:

```
python setup.py install
```

Build the Documentation

The documentation (which you are currently reading) can be built from source using (assuming sphinx is installed):

```
sphinx-build docs html
```

This will place documents in `html` within the project directory.

Alternatively this documentation is hosted at http://atsimpotentials.readthedocs.org

2.3 User Guide

This user-guide is broadly split into two sections. The first (*Using potable*) is for users of the *potable*. This tool allows potential tabulation without knowledge of Python and this section of the user-guide is aimed at describing *potable*input files and how they can be used to describe a wide-range of potential models.

The second half (*Using the Python API*) describes the python interface to `atsim.potentials`. This is provided to allow tabulation tasks to be automated, included in other scripts or for when *potable* is not expressive enough.

2.3.1 Using `potable`

This section of the user-guide will start by first explaining *pair-potential tabulation* before moving on to describe *many body potentials*, these contain additional terms that add more complexity to their tabulation.

Pair-potential models

The *Quick Start guide* should have already given you an idea of what a *potable* potential definition looks like. We will now delve a little deeper and describe this input in more detail.

File Structure

`[Tabulation]` section

This section defines the file format in which `potable` will write its output files through the *target* configuration item.

See also:

- Input format reference: *[Tabulation]*.

The pair-tabulation *target* can be one of:

- `DL_POLY` (or `DLPOLY`): this creates output in the `TABLE` format accepted by the `DL_POLY` simulation code.
- `LAMMPS`: creates output suitable for use with the LAMMPS `pair_style table` (see *Using Basak.lmptab in LAMMPS*).
- `GULP` produces a table for the `GULP` code by defining a set of separation, energy pairs using the `GULP spline` directive.

Defining table cut-off

The extent of the table is defined in the `[Tabulation]` section using the *nr*, *dr* and *cutoff* options:

- `dr` defines the row increment (step-size) between table rows.
- `cutoff` gives the maximum separation to be tabulated.
- `nr` determines the number of rows in the tabulation.

Any two of `nr`, `dr`and `cutoff` can be used to define the extent and resolution of the tabulation. As an example all three of the following `[Tabulation]` sections would produce a table with 1000 rows, each separated by 0.01:

1. `cutoff` and `nr`

```
[Tabulation]
target: LAMMPS
cutoff : 9.99
nr : 1000
```

2. `cutoff` and `dr`

```
[Tabulation]
target: LAMMPS
cutoff : 9.99
dr : 0.01
```

3. `nr` and `dr`

```
[Tabulation]
target: LAMMPS
nr : 1000
dr : 0.01
```

`[Pair]` section

In the `[Pair]` section of the model definition, potential-forms are combined with parameters that tailor them to a given pair of species.

The *basic* form of an entry in this section is:

```
SPECIES_A-SPECIES_B : POTENTIAL_FORM PARAM_1 PARAM_2 ... PARAM_N
```

- **The label before the colon: `SPECIES_A-SPECIES_B` identifies the pair interaction being parametrised.**
    - This label consists of two species identifiers (`SPECIES_A` and `SPECIES_B`) separated by a hyphen `-`.
    - The order in which the species labels are specified does not matter. That is, `Au-Ag` and `Ag-Au` would be equivalent.
    - The `[Pair]` section can only contain one entry per unique species pair.
- The potential definition, specified after the colon, consists of the name of the potential-form (`POTENTIAL_FORM`) followed by the numeric parameters it requires.

Pre-defined potential-forms

A number of *pre-defined potential forms are provided*. These all have names pre-fixed by `as.`

Each entry in the *list of potentials* provides an entry called `potable signature`. This shows the order in which parameters should be given to create a potential.

For the *Buckingham* potential the `potable signature` is:

`as.buck` A ρ C

which is associated with the formula:

$$V(r_{ij}) = A \exp\left(-\frac{r_{ij}}{\rho}\right) - \frac{C}{r_{ij}^6}$$

This means that if we were defining a potential between Si and O that had A_{ij} = 18003, ρ_{ij} = 0.205 and C_{ij} = 133.36 then the entry in the `[Pair]` section would be:

```
[Pair]
Si-O = as.buck 18003.0 0.205 133.36
```

Please refer to the `potable signature` when using the `as.*` potential-forms; specifying parameters in the wrong order will cause you problems.

It is also possible to define your own potential-forms in the `[Potential-Form]` section of `potable` file. These are parametrised here in the `[Pair]` section in the same way as the pre-defined `as.*` potential-forms. This usage is documented later here: *[Potential-Form] section*.

Potential modifiers

If you followed the *Quick-Start* guide, you will have already seen a potential modifier. The `[Pair]` section from the `basak.aspot` used in the *Quick-Start* is repeated here:

```
[Pair]
O-O = as.buck 1633.010242995040 0.327022 3.948787
U-U = as.buck 294.640906285709 0.327022 0.0
O-U = sum(as.buck 693.650933805978 0.327022 0.0,
                  as.morse 1.65 2.369 0.577189831995)
```

You can see that the O-O and U-U pairs use the basic definition we have just seen. The U-O interaction however uses the modified form:

```
O-U = sum(as.buck 693.650933805978 0.327022 0.0,
                  as.morse 1.65 2.369 0.577189831995)
```

Here `sum()` takes two basic pair-definitions (one for *as.buck* and one for *as.morse*) and creates a pair-potential that is the sum of both. Here `sum()` is acting as a potential-modifier.

Potential-modifiers take the input or output of other potentials and produce outputs that have been altered in some way. A number of modifiers are provided with `atsim.potentials` and these are listed.

See also:

See *List of Potential Modifiers*.

Multi-range potentials

The potential definition syntax used in the `[Pair]` section supports an extension which allows a series of potential-forms to be concatenated to each other, allowing each to act over a particular range of separations. These are defined as multi-range potentials. Concrete examples of where they are useful are provided in *Multi-range potential examples* however the basic syntax defining multi-range potentials is introduced here.

Suppose we want to define a potential acting between Mg and O using two potential-forms: `pot_A` and `pot_B`. The first is to be parametrised with values of 5.3 and 1.2 and `pot_B` with 9.6 and 2.4. Now say we want `pot_A` to act over the separations $0 \geq r_{ij} \leq 3$ and `pot_B` $3 < r_{ij} \leq 8$ and for the pair-potential to evaluate to zero when $r_{ij} > 8$.

Fig. 2.2: Illustration of multi-range potential definition described in the text.

The multi-range potential just described is summarised in Fig. 2.2. This would be defined as follows in the `potable` input file:

```
[Pair]
Mg-O : >=0 pot_A 5.3 1.2 >3 pot_B 9.6 2.4 >8 as.zero
```

Notice that we used the *as.zero* potential-form to provide a constant value of 0.0 when $r_{ij} > 8$ (equally `as.constant 0.0` could have been used).

The syntax for a multi-range potential can be summarised as:

- A series of single potential definitions delimited by range markers.
- Range markers take the form:
  - `>=R` which indicates that the potential definition, following the marker, will be used at separations **greater than or equal** to the value specified by `R`
  - `>R` which means the same but acts only for separations **greater** than `R`.

Note: `potable` defines all potentials to have the initial range of `>0` unless a range is explicitly defined. This is to avoid divide by zero errors when the potential is evaluated for $r_{ij} = 0$. As this separation is unimportant to most physically relevant simulation.

To include $r_{ij} = 0$ in you tabulations simply make sure that your potential starts with `>=0`

```
>=0 POTENTIAL_DEFN
```

See also:

- Examples of multi-range potentials can be found here: *Multi-range potential examples*

`[Potential-Form]` section

This section of the input file is used for defining formulae that can be used as potential-forms and functions elsewhere in the model definition.

This allows for potential-forms that are not described in the standard file itself. Entries in this section have the general form:

```
LABEL([ARG_1, ARG_2, ... , ARG_N]) : FORMULA
```

- `LABEL` is a unique identifier for the function (this can be used to refer to this function in the *[Pair]*).
- `ARG ...` defines the function signature by naming the arguments it takes.
- `FORMULA` mathematical expression defining the function.

Formula syntax

The `FORMULA` strings used in this section of the configuration file support a rich range of mathematical expressions. These are parsed using the cexprtk python module which is built on top of the exprtk library. As a result the supported operators and functions are listed here .

In addition, users can make use of functions from the python math module. In `FORMULA` definitions these are called with a `pymath.*` prefix. A list of the supported python math functions are given here: *Python maths functions supported in mathematical expressions*.

pymath formula example

As an example of where a `pymath` function might be useful, `exprtk` does not natively provide a means of calculating factorials. Instead, the python factorial() function can be used in a `FORMULA` as shown in this potential form definition:

```
N(zeta, n) = (2*zeta)**n * sqrt((2*zeta)/pymath.factorial(2*n))
```

Example: using custom-potential forms to define Basak potential

We will revisit the Basak [Basak2003] UO_2 model. In the *Quick-Start* guide you will have seen that the published potential parameters required considerable manipulation to make them compatible with the *Buckingham (buck)* and *Morse (morse)* potential-forms defined in `atsim.potentials` (see *Potential model*). Rather than transforming model parameters in this way, it may be easier to use the pair-potential equations and parameters directly as they appear in a paper. The `[Potential-Form]` section is the mechanism by which this may be achieved.

The Buckingham potential used in the Basak paper has the form:

$$V_{\text{Buck}}(r_{ij}) = f_0 b_{ij} \exp\left(\frac{a_{ij} - r_{ij}}{b_{ij}}\right) - \frac{C_{ij}}{r_{ij}^6}$$

We can write this as an entry in `[Potential-Form]` as:

```
[Potential-Form]
basak_buck(r,f0,a,b,c) = f0*b*exp((a-r)/b) - c/r^6
```

Note: You may have noticed in this equation that we defined one of the terms using `r^6` (r to the sixth power), where python syntax would define this as `r**6`. This is because, the formulae defined in this section are parsed using the exprtk library (via its python wrapper cexprtk).

To understand the functions, operators and syntax, supported for formulae please refer to the exprtk documentation.

It is also possible to call the standard `as.*` *potential-forms* in `[Potential-Form]` expressions. This is shown here to define a `basak_morse` function, where *as.morse* will be used to provide a function that can be used directly with the parameters from the Basak paper (Table 2.2). For reference, *as.morse* has the `potable signature`:

`as.morse` $\gamma\ r_*\ D$

Remembering that the D parameter is given as $f_0 \times D$ using parameters from Table 2.2 (see *Potential model*) we can now define our second potential function:

```
[Potential-Form]
basak_buck(r,f0,a,b,c) = f0*b*exp((a-r)/b) - c/r^6
basak_morse(r, f0, d, gamma, r_star) = as.morse(r,gamma, r_star, f0*d)
```

Note: It should also be noted that not-all the `as.*` potential-forms are available as functions within these formulae (for instance *as.buck4* isn't). If you would like to check, please refer to the *List of Potential Forms* and make sure that `potential-function` is listed as one of its `Features`.

Now that we have both functions, we need to parametrise them for each interaction using values from Table 2.2. This is achieved in the normal way in the *[Pair]* section:

```
[Pair]
O-O : basak_buck 0.042203 3.82 0.327022 3.948787
U-U : basak_buck 0.042203 3.26 0.327022 0.0
O-U : sum(
        basak_buck  0.042203 3.54 0.327022 0.0,
        basak_morse 0.042203 13.6765 1.65 2.369)
```

Notice that for the `O-U` interaction we continue to use the *sum()* potential-modifier to combine our Buckingham and Morse potentials (see *Potential modifiers*).

The order in which parameters are specified in the `[Pair]` entries correspond to the arguments in the function signatures for `basak_buck` and `basak_morse`, as is now shown:

Note: By convention the `as.*` potentials take `r` (separation) as their first argument when used in formulae in the `[Potential-Form]` section.

This represents a subtle to difference to when they appear in the `[Pair]` section and the argument list defined by the `potable signature` entries in *List of Potential Forms*.

For instance where `as.buck` could be parametrised as `as.buck 1000.0 0.2 32.0` in the `[Pair]` section it would be defined as `as.buck(r, 1000.0, 0.2, 32.0)` in a `[Potential-Form]` formula.

The model is now fully defined and gives the following potable input:

```
[Tabulation]
target :  LAMMPS
nr : 1000
dr : 0.01

[Pair]
O-O : basak_buck 0.042203 3.82 0.327022 3.948787
U-U : basak_buck 0.042203 3.26 0.327022 0.0
O-U : sum(
        basak_buck  0.042203 3.54 0.327022 0.0,
        basak_morse 0.042203 13.6765 1.65 2.369)


[Potential-Form]
basak_buck(r,f0,a,b,c) = f0*b*exp((a-r)/b) - c/r^6
basak_morse(r, f0, d, gamma, r_star) = as.morse(r,gamma, r_star, f0*d)
```

This input file can be downloaded as `basak_custom_potential_form_a.aspot` and tabulated thus:

```
potable basak_custom_potential_form_a.aspot Basak.lmptab
```

Section *Using Basak.lmptab in LAMMPS* describes how this table can then be used to perform a molecular dynamics simulation.

Alternative descriptions

The potential-forms used in the previous example could have been defined in a number of different ways. Some of these are now shown to illustrate the flexibility of the `potable` system:

- `basak_custom_potential_form_b.aspot`. In this example, a third potential-form `basak_buckmorse` is defined. This adds `basak_buck()` to `basak_morse` as an alternative to using the *sum()* potential modifier in the *[Pair]*.

  ```
  [Tabulation]
  target :  LAMMPS
  nr : 1000
  dr : 0.01
  ```

(continues on next page)

(continued from previous page)

```
[Pair]
O-O : basak_buck 0.042203 3.82 0.327022 3.948787
U-U : basak_buck 0.042203 3.26 0.327022 0.0
O-U : basak_buckmorse 0.042203 3.54 0.327022 0.0 13.6765 1.65 2.369


[Potential-Form]
basak_buck(r,f0,a,b,c) = f0*b*exp((a-r)/b) - c/r^6
basak_morse(r, f0, d, gamma, r_star) = as.morse(r,gamma, r_star, f0*d)
basak_buckmorse(r,f0,a,b,c,d,gamma,r_star) = basak_buck(r,f0,a,b,c) +␣
↪basak_morse(r, f0,d,gamma,r_star)
```

- `basak_custom_potential_form_c.aspot`. In this example the Morse potential is described directly rather than delegating to the `as.morse()` function:

```
[Tabulation]
target :  LAMMPS
nr : 1000
dr : 0.01

[Pair]
O-O : basak_buck 0.042203 3.82 0.327022 3.948787
U-U : basak_buck 0.042203 3.26 0.327022 0.0
O-U : basak_buckmorse 0.042203 3.54 0.327022 0.0 13.6765 1.65 2.369


[Potential-Form]
basak_buck(r,f0,a,b,c) = f0*b*exp((a-r)/b) - c/r^6
basak_morse(r, f0, d, gamma, r_star) = f0*d*(exp(-2*gamma*(r-r_star)) -␣
↪2*exp(-gamma*(r-r_star)))
basak_buckmorse(r,f0,a,b,c,d,gamma,r_star) = basak_buck(r,f0,a,b,c) +␣
↪basak_morse(r, f0,d,gamma,r_star)
```

- `basak_custom_potential_form_d.aspot`. This example shows that `[Potential-Form]` formulae can refer to each other.
  - In order to use the standard `as.buck()` potential-function, its A_{ij} parameter must be calculated from the `f0`, `a` and `b` Basak parameters (see *Potential model*).
  - Here an `A_ij()` formula is defined which is then invoked from inside the `basak_buck()` function. This sort of modularisation allows well structured and hence simpler expressions to be define.

```
[Tabulation]
target :  LAMMPS
nr : 1000
dr : 0.01

[Pair]
O-O : basak_buck 0.042203 3.82 0.327022 3.948787
U-U : basak_buck 0.042203 3.26 0.327022 0.0
O-U : basak_buckmorse 0.042203 3.54 0.327022 0.0 13.6765 1.65 2.369


[Potential-Form]
A_ij(f0, a,b) = f0*b*exp(a/b)
basak_buck(r,f0,a,b,c) = as.buck(r, A_ij(f0,a,b), b, c)
```

(continues on next page)

(continued from previous page)

```
basak_morse(r, f0, d, gamma, r_star) = as.morse(r,gamma, r_star, f0*d)
basak_buckmorse(r,f0,a,b,c,d,gamma,r_star) = basak_buck(r,f0,a,b,c) +␣
↪basak_morse(r, f0,d,gamma,r_star)
```

`[Table-Form]` section

This section of the input serves a similar purpose to the *[Potential-Form]* section as it allows custom potential functions to be defined. However, instead of being defined using a mathematical expression they are specified as tables of x,y points with interpolation providing intermediate values.

It is sometimes to convenient to use pre-tabulated values for very complex expressions however you should always use caution. It is recommended that when using `[Table-Form]` that you plot the resulting functions and derivatives. This is because interpolation can sometimes introduce spurious effects, so it's worth checking that nothing odd has happened.

Defining a Table Form

The general form of a `[Table-Form]` section is:

```
[Table-Form:NAME]
interpolation : INTERPOLATION_TYPE
xy : DATA
```

Multiple `[Table-Form]` sections can appear in a `potable` input file and are distinguished by the `NAME` identifier included after the colon in the section header. This identifier can be used elsewhere, such as in `[Pair]` to use the function.

The value of `INTERPOLATION_TYPE` specifies how values are calculated between data-points. A list of supported interpolation schemes can be found *here*, but for the examples that follow we will use `cubic_spline` interpolation.

Finally the function's data is provided through the `xy` option. The value of `DATA` is a list of space separated x,y pairs. To aid readability these can appear on separate lines as long as they are indented. Alternatively, table data can be specified as separate arrays of x and y values *see here for more* .

Example: [Table-Form] pair-potential

Revisiting the example from earlier (see *Potential model*) the following shows how the O-U interaction from the Basak model [Basak2003] can be represented as a `[Table-Form]`:

The data points from Fig. 2.3 (blue) have been included in the `potable` input file `basak_table_form.aspot` using the *xy* option.

The third line of the `[Pair]` section deserves notice:

```
O-U = tabulated
```

The potential-form `tabulated` specified for the `O-U` interaction refers to the name of the table-form: `[Table-Form:tabulated]`. It should also be noted that instances of the potential form do not take any parameters.

```
[Tabulation]
target :  LAMMPS
cutoff : 6.5
nr : 652

[Pair]
O-O = as.buck 1633.010242995040 0.327022 3.948787
U-U = as.buck 294.640906285709 0.327022 0.0
O-U = tabulated

[Table-Form:tabulated]
interpolation : cubic_spline
xy :  0.019969278       1939.293892
      0.169738863       1188.215091
      0.319508449       726.7144999
      0.469278034       443.3947902
      0.619047619       269.6704607
      0.768817204       163.3187664
      0.91858679       98.35155402
      1.068356375       58.77819733
      1.21812596       34.76429963
      1.367895545       20.26573351
      1.517665131       11.57134322
      1.667434716       6.405364338
      1.817204301       3.374649118
      1.966973886       1.62831462
      2.116743472       0.648256084
      2.266513057       0.120274645
      2.416282642       -0.14514429
      2.566052227       -0.261481615
      2.715821813       -0.29602021
      2.865591398       -0.288108481
      3.015360983       -0.260312815
      3.165130568       -0.225205576
      3.314900154       -0.189478838
      3.464669739       -0.15642305
      3.614439324       -0.127408946
      3.764208909       -0.102764146
      3.913978495       -0.082284087
      4.06374808       -0.065523547
      4.213517665       -0.051957769
      4.36328725       -0.04106708
      4.513056836       -0.032377452
      4.662826421       -0.025476347
      4.812596006       -0.020015289
      4.962365591       -0.015705784
      5.112135177       -0.012312372
      5.261904762       -0.009644843
      5.411674347       -0.007550718
      5.561443932       -0.005908467
      5.711213518       -0.004621653
      5.860983103       -0.00361401
      6.010752688       -0.002825384
      6.160522273       -0.002208426
      6.310291859       -0.001725927
      6.460061444       -0.001348681
      6.5 0.0
```

The following figure shows the good match between the analytical and tabulated forms resulting from the use of the `[Table-Form]`.

Fig. 2.3: Plot showing the table points used in the example (blue circles) and the function resulting from cubic spline interpolation (red). The original analytical form is shown in grey.

Potable files for pair-potential models may contain the following sections:

```
[Tabulation]
...

[Pair]
...

[Potential-Form]
...

[Table-Form:NAME]
...

[Species]
...
```

Each section may contain a number of configuration options. These have the general form:

```
ITEM : VALUE
```

or an equals sign may be used instead:

```
ITEM = VALUE
```

Where `ITEM` identifies the configuration option's name and `VALUE` its value.

Lines maybe commented out using `#` and line continuation is supported according to identation (see here for more details).

Each section of the file has a specific purpose and not all sections will be required in all cases:

- ***[Tabulation] section***
    - describes how the file should be converted into a table file by the *potable* command. Contains information such as cutoff, output table format and cutoff.
- ***[Pair] section***
    - this is where pair interactions are defined by parametrising a potential-form.
- ***[Potential-Form] section***
    - this section allows custom potential-forms to be defined. This may be required when you can't find an appropriate function from those *supplied* with `atsim.potentials`. However in many cases this won't be necessary and this section needn't appear in your model definition.
- ***[Table-Form] section***
    - Multiple `[Table-Form]` sections may be specified. These allow potential-forms to be defined from tables of x,y points. These tabulated forms can be used in the same way as potnetials defined in `[Potential-Form]`.
- **`[Species]`**

  - this is used to provide meta-data about the species being tabulated. In most cases this section can be omitted (as very little species data is used during pair-tabulation). It is however, sometimes useful to include atomic charges etc, here so that the input file represents a complete description of a given potential model.

For completeness the `[EAM-Embed]` and `[EAM-Density]` sections are mentioned here. These are not required for pair potential models but are used in many body models:

- `[EAM-Density]` is described *here*.
- `[EAM-Embed]` is described *here*.

Combining potentials

It is often useful to describe the different regions of a pair-potential using different functional forms. This section of the user-guide will show ways in which different potentials can be combined.

First, examples will be given demonstrating the use *multi-range potentials* described earlier.

Methods for linking potential-forms using splines will then be considered.

Multi-range potential examples

This section shows how potential-forms can be combined using the multi-range syntax introduced here: *Multi-range potentials*.

Example: truncating a potential

The starting structure for a simulation may contain overlapping atoms (for example if generated from random coordinates). In these cases it is useful to move atoms apart before starting the simulation proper.

In principle this should be possible by using an energy minimisation run, however some potential forms are badly behaved at very small separations (for instance the Buckingham catastrophe) leading to unexpected results.

One way of removing atom overlap is to temporarily substitute the potentials with some that are guaranteed to work, even when two atoms are completely on top of each other. An example of a potential-form that could be used for this purpose is now described (this is available natively in LAMMPS as pair_style soft):

$$V_{ij}(r_{ij}) = A\left[1 + \cos\left(\frac{\pi r_{ij}}{r_c}\right)\right]$$

Let's start by plotting this function with sensible parameter values of A = 10 and r_c = 1.6:

To make this useful, the potential is truncated at r_c (i.e. $r_{ij} < r_c$). This means that the potential is repulsive below r_c and doesn't act above:

Now, having been suitably truncated, the potential will gently push atoms apart in an MD or energy minimisation run. This can be implemented for *potable* as follows; first define the cosine function:

```
[Potential-Form]
soft(r, A, rc) = A * (1+cos((pi*r)/rc))
```

The truncation at r_c can then be applied using a multi-range `[Pair]` deinition.

```
[Pair]
Si-O : soft 10.0 1.6 >1.6 as.zero
O-O  : soft  5.0 2.4 >2.4 as.zero
```

Here the Si-O interaction have A = 10.0 and an r_c value of 1.6Å (the position of the Si-O peak in a silica glass radial distribution function) after which the *Zero (zero)* potential-form takes over. Similarly, the O-O pair has a weaker repulsion ($A = 5.0$) and a cutoff-radius, r_c, of 2.4Å (again this is about the position of the O-O peak in the RDF for silica glass).

In this way the cosine function has been truncated. Specifying an Si-Si interaction will be left as an exercise for the reader.

The complete input file for this example can be downloaded here `soft_a.aspot` and is:

```
[Tabulation]
target : GULP
cutoff : 10.0
dr : 0.01

[Pair]
Si-O : soft 10.0 1.6 >1.6 as.zero
O-O  : soft  5.0 2.4 >2.4 as.zero

[Potential-Form]
soft(r, A, rc) = A * (1+cos((pi*r)/rc))
```

Example: truncating using `if()` statement

This example shows an alternative to using the multi-range syntax to implement the soft cosine potential described above. The exprtk package used to evaluate expressions in the *[Potential-Form]* section, support an `if ... then ... else` construct through its `if()` function. This has the form:

```
if (CONDITION, FORMULA_IF_TRUE, FORMULA_IF_FALSE)
```

This means the $r > r_c$ condition could have been included in *[Potential-Form]* like this:

```
[Potential-Form]
cos_form(r, A, rc) = A * (1+cos((pi*r)/rc))
soft(r, A, rc) = if(r>rc, 0, cos_form(r, A, rc))
```

This simplifies the definition of the `[Pair]` section by avoiding the repetition of the r_c parameter when defining the multi-range version of the potential:

```
[Pair]
Si-O : soft 10.0 1.6
O-O  : soft  5.0 2.4
```

The *potable* input using this version of the potential-form can be downloaded here: `soft_if.aspot`.

Example: parametrising a model using published spline coefficients

The UO_2 potential model by Morelon describes U-O and O-O interactions [Morelon2003]. The former employs the *Born-Mayer (bornmayer)* potential-form while the latter uses the combination of a 3rd and 5th order polynomial spline to link an $A \exp\left(-\frac{r_{ij}}{\rho}\right)$ term to a $-\frac{C}{r_{ij}^6}$ term. The spline-coefficients for the Morelon model are available in a paper by Potashnikov et al. [Potashnikov2011]. They give the O-O interaction as:

$$V(r_{ij}) = \begin{cases} 11272.6 \exp\left(\frac{0.1363}{r_{ij}}\right) & : r_{ij} < 1.2 \\ 479.955 - 1372.53 r_{ij} + 1562.22 r_{ij}^2 - 881.969 r_{ij}^3 + 246.435 r_{ij}^4 - 27.2447 r_{ij}^5 & : 1.2 \leq r_{ij} < 2.1 \\ 42.8917 - 55.4965 r_{ij} + 23.0774 r_{ij}^2 - 31.13140 r_{ij}^3 & : 2.1 \leq r_{ij} < 2.6 \\ -\frac{134}{r_{ij}^6} & : r_{ij} \geq 2.6 \end{cases} \tag{2.8}$$

The following figure plots the O-O interaction to show how its constituent parts relate to each other.

Fig. 2.4: The O-O interaction of the Morelon model is made up of several potential-forms acting over different ranges. The upper plot shows the un-trimmed versions of the individual functions. The lower plot shows how these combine to form the overall interaction.

Note: in a simulation the O-O interaction would also include a repulsive electrostatic term which isn't plotted here.

Using this information we can write the Morelon model as *potable* input (and can be downloaded as `morelon.aspot`):

```
[Tabulation]
target : LAMMPS
cutoff : 10.0
nr : 1001

[Pair]
O-U : as.bornmayer 566.498 0.42056
O-O : as.bornmayer 11272.6 0.1363
      >1.2
          as.polynomial 479.955 -1372.53 1562.22 -881.969 246.435 -27.2447
      >2.1
          as.polynomial 42.8917 -55.4965 23.0774 -3.13140
      >2.6
          as.buck 0.0 1.0 134.0
```

There are a few features here that are worth noting:

- The four distinct regions are defined as multi-range potentials.
- The third and fifth order polynomials are both defined using the *as.polynomial* form:
  - The order of the polynomial is implicitly defined by the number of parameters.
- The $-\frac{C}{r_{ij}^6}$ term is implemented using the *Buckingham (buck)* form. In order to only use the attractive part of the potential an `A` parameter of 0.0 is used as its first argument and a non-zero `rho` is then specified.

Splining

Splines are curves that are used to smoothly connect different functional forms across different ranges of a potential-form. Typically splining curves are polynomials. Care must be taken when determining the coefficients of the spline function as it is important that potentials are smooth and do not have steps in their derivatives. For this reason spline parameters are determined so that at its end-points, it gives the same potential-energy and derivatives (1st and 2nd) as the functions it joins.

In the previous example (see *Example: parametrising a model using published spline coefficients*), spline coefficients were provided but their calculation can be quite involved. As a result `potable` provides services to make this easier.

Splining is performed in the `[Pair]` section of the input through the `spline()` *potential-modifier*.

`spline()` takes a single argument which has the form of a *multi-range potential definition*. This has three distinct parts, representing the three ranges of the splined potential:

1. The region described by the starting potential.
2. The interpolation region given by the spline.
3. The final part defined by the end potential.

In the description that follows, the separation at which the starting potential ends will be referred to as the **detachment-point**. Similarly, the end of the splined region is the **attachment-point**.

Bearing in mind its similarity to the *multi-range* syntax `spline()` definitions have this basic form:

```
spline(POT_A_DEFN >R_DETACH SPLINE_DEFN >R_ATTACH POT_B_DEFN)
```

Where:

- `POT_A_DEFN` - this is the potential definition for the starting-potential. This is typically a potential-form label followed by potential parameters (see *[Pair] section*).
- `R_DETACH` - detachment-point (**note:** as this is a multi-range potential definition the inclusive form `>=R_DETACH` is also valid).
- `SPLINE_DEFN` - the type of spline to be used along with any parameters it takes. This will be described properly in the next paragraph.
- `R_ATTACH` - attachment point (again `>=R_ATTACH` is also accepted).
- `POT_B_DEFN` - the potential definition of the end-potential.

`SPLINE_DEFN` has the same format as a potential-definition: an identifying label followed by a list of parameters (**note:** some spline-types do not take any parameters):

```
SPLINE_LABEL PARAM_1 PARAM_2 ... PARAM_N
```

Currently `SPLINE_LABEL` can be *exp_spline* or *buck4_spline*. These spline-types will now be described.

Exponential Spline `exp_spline`

The spline used when `exp_spline` is specified is given as:

$$V(r_{ij}) = \exp\left(B_0 + B_1 r_{ij} + B_2 r_{ij}^2 + B_3 r_{ij}^3 + B_4 r_{ij}^4 + B_5 r_{ij}^5\right) \tag{2.9}$$

Where $B_{0..5}$ are the spline coefficients calculated automatically during splining.

The exponential spline does not take any parameters.

The functional form given in eqn. (2.9) is also available as a potential-form. Although the user is then responsible for calculating their own spline-coefficients. See: *Exponential Spline (exp_spline)*.

Example: splining to the `zbl` potential form using `exp_spline`

This example will show how to use `spline()` with the `exp_spline` type to add the repulsive *Ziegler-Biersack-Littmark (zbl)* to a *Buckingham (buck)* potential that is unphysical at small separations.

For certain parameterisations, popular potential forms can exhibit unphysical behaviour for some interatomic separations. A popular model for the description of silicate and phosphate systems is that due to van Beest, Kramer and van

Santen (the BKS potential set) [VanBeest1990]. In the current example, the Si-O interaction from this model will be considered. This uses the Buckingham potential form with the following parameters:

- A = 18003.7572 eV
- ρ = 0.205204
- C = 133.5381 eV ^{6}
- **Charges:**
  - Si = 2.4 *e*
  - O = -1.2 *e*

The plot in Fig. 2.5 shows the combined coulomb and short-range contributions for this interaction plotted as a function of separation. The large C term necessary to describe the equilibrium properties of silicates means that as r_{ij} gets smaller, the $\frac{C}{r_{ij}^6}$ overwhelms the repulsive Born-Mayer component of the Buckingham potential meaning that it turns over. This creates only a relatively shallow minimum around the equilibrium Si-O separation. Within simulations containing high velocities (e.g. high temperatures or collision cascades) atoms could easily enter the very negative, attractive portion of the potential at low r_{ij} - effectively allowing atoms to collapse onto each other. In order to overcome this deficiency a ZBL potential will be splined onto the Si-O interaction within this example.

Fig. 2.5: Plot showing Si-O pair-potential and its electrostatic and short-range components. A small separations it becomes attractive which is unphysical.

The first step to using `exp_spline` is to choose appropriate detachment and attachment points. This is perhaps best done by plotting the two potential functions to be splined. The *as.buck* and *as.zbl* curves in Fig. 2.6a, show that detachment and attachment at separations of 0.8 and 1.4 may be appropriate.

Fig. 2.6: The result of splining using the `exp_spline` potential.

We now have enough information to spline the two potentials together. This gives is defined in the `[Pair]` section:

```
[Pair]
Si-O : spline(
                   as.zbl 14 8
             >=0.8
                   exp_spline
             >=1.4
                   as.buck 18003.7572 0.205204 133.5381 )
```

The result of splining can be seen in Fig. 2.6b (here it plotted with the Coulomb interaction included).

The complete `potable` input can be downloaded here: `exp_spline.aspot`.

See also:

- This example *Example: Splining ZBL Potential on to Buckingham Potential* shows how to achieve the same result using the Python API.

Buckingham-4 Spline `buck4_spline`

When used with `buck4_spline` the overall potential has the form:

$$V(r_{ij}) = \begin{cases} V_{\text{PotA}}(r_{ij}) & : 0 \le r_{ij} \le r_{\text{detach}} \\ a_0 + a_1 r_{ij} + a_2 r_{ij}^2 + a_3 r_{ij}^3 + a_4 r_{ij}^4 + a_5 r_{ij}^5 & : r_{\text{detach}} < r_{ij} < r_{\text{min}} \\ b_0 + b_1 r_{ij} + b_2 r_{ij}^2 + b_3 r_{ij}^3 & : r_{\text{min}} \le r_{ij} < r_{\text{attach}} \\ V_{\text{PotB}}(r_{ij}) & : r_{ij} \ge r_{\text{attach}} \end{cases} \tag{2.10}$$

The spline form can be seen to be the combination of a fifth order and cubic polynomial. As for other spline types the coefficients ($a_{0...5}$ and $b_{0...3}$) are chosen so that first and second derivatives (with respect to separation) are continuous through the attachment and detachment points. In addition, the coefficients are determined to give a stationary point where the two polynomials meet at r_{min}.

The `SPLINE_DEFN` part of the `spline()` definition for `buck4_spline` is:

`buck4_spline` r_{min}

Relationship to `as.buck4` potential-form

As its name suggests, the `buck4_spline` is closely related to the *Buckingham-4 (buck4)* potential-form. When used as a potential-form, the $V_{\text{PotA}}(r_{ij})$ and $V_{\text{PotB}}(r_{ij})$ terms from eqn. (2.10) are pre-defined as the first and last terms of the *Buckingham* potential:

$$V_{\text{PotA}}(r_{ij}) = A_{ij} \exp\left(-\frac{r_{ij}}{\rho}\right)$$
$$V_{\text{PotB}}(r_{ij}) = -\frac{C}{r_{ij}}$$

The potential-form is therefore shorthand for the following:

```
spline(as.buck A RHO 0.0 >R_DETACH buck4_spline R_MIN >R_ATTACH as.buck 0 1.0 C)
```

And would be specified as:

```
as.buck4 A RHO C R_DETACH R_MIN R_ATTACH
```

Where: `A`, `RHO` and `C` are the potential parameters and `R_DETACH`, `R_MIN` and `R_ATTACH` define the splined region.

In most cases users will prefer the `as.buck4` form unless they have a specific reason to change the initial and final potential functions.

Example: redefining the Morelon model using `buck4_spline`

In this example the Morelon potential model will be re-implemented first using the `spline()` modifier then with the *as.buck4*.

This model was discussed in a previous example: *Example: parametrising a model using published spline coefficients* where O-O and O-U interactions were defined like this:

```
[Pair]
O-U : as.bornmayer 566.498 0.42056
O-O : as.bornmayer 11272.6 0.1363
      >1.2
```

(continues on next page)

(continued from previous page)

```
        as.polynomial 479.955 -1372.53 1562.22 -881.969 246.435 -27.2447
    >2.1
        as.polynomial 42.8917 -55.4965 23.0774 -3.13140
    >2.6
        as.buck 0.0 1.0 134.0
```

From this it can be seen that the O-U interaction is relatively simple whilst the O-O interaction has four distinct parts (also defined in eqn. (2.8)), and when examined more closely it can be seen it matches the four range Buckingham form.

The polynomial terms, with their pre-supplied coefficients, can be replaced by using the `spline()` modifier with the `buck4_spline` spline type. This is now shown and can be downloaded as: `morelon_buck4_spline.aspot`

```
1 [Pair]
2 O-U : as.bornmayer 566.498 0.42056
3 O-O : spline(
4             as.bornmayer 11272.6 0.1363
5              >1.2
6                buck4_spline 2.1
7              >2.6
8                as.buck 0.0 1.0 134.0 )
```

To change the starting potential and end potential lines 4 and 8 would be edited. To edit the detach and attach points then lines 5 and 7 would be altered. To change the position of the stationary point in the splined region, the value of 2.1 at the end of line 6 would be edited.

This input file could also be re-written to use the `buck4` potential-form, like this:

```
[Pair]
O-U : as.bornmayer 566.498 0.42056
O-O : as.buck4 11272.6 0.1363 134.0 1.2 2.1 2.6
```

This version of the model can be downloaded here: `morelon_buck4.aspot`

Many body models

Models that use the Embedded Atom Method (EAM) can be tabulated using *potable*. Embedded atom models take the general form

$$E_i = F_\alpha \left(\sum_{j \neq i} \rho_\beta(r_{ij}) \right) + \frac{1}{2} \sum_{j \neq i} \phi_{\alpha\beta}(r_{ij}) \tag{2.11}$$

- Where:
  - $\rho_\beta(r_{ij})$ is the density function which gives the electron density for atom j with species β as a function of its separation from atom i, r_{ij}.
  - The electron density for atom i is obtained by summing over the density ($\rho_\beta(r_{ij})$ contributions due to its neighbours.
  - The embedding function $F_\alpha(\rho)$ is used to calculate the many-bodied energy contribution from this summed electron density.
  - The sum $\frac{1}{2} \sum_{j \neq i} \phi_{\alpha\beta}(r_{ij})$ gives the pair-potential contribution to atom i's energy.
  - $\phi_{\alpha\beta}(r_{ij})$ are simply pair potentials that describe the energy between two atoms as a function of their separation.

In order to support the description of EAM when compared to *pair-potential models*, additional sections in the input file are required. These are:

- `[EAM-Density]`: defines EAM density functions.
- `[EAM-Embed]`: describe the model's embedding functions.

As before, the pairwise section of the forcefield is specified in the *[Pair] section* of the input file.

The EAM sections of the input are defined in much the same way as the *[Pair] section*. Both `[EAM-Density]`, `[EAM-Embed]` allow the use of *multi-range potential definitions* and *potential modifiers*.

As for pair only models, many-bodied force fields can specify *[Potential-Form]* and *[Table-Form]* sections can be provided if custom functional forms are required in `[EAM-Density]`, `[EAM-Embed]` or `[Pair]`.

[EAM-Density]

The density functions for embedded atom models are specified in this section. For the EAM form shown in equation (2.11) entries in this section take the form:

```
SPECIES : POTENTIAL_FORM PARAM_1 PARAM_2 ... PARAM_N
```

- Where:
  - `SPECIES` species for which density should be calculated
  - `POTENTIAL_FORM PARAM_1 PARAM_2 ... PARAM_N` potential form definition.

Density functions are tabulate in r_{ij} space, therefore their extent and resolution are controlled by the the *dr*, *nr* and *cutoff* fields in the *[Tabulation]* section in the same way as for `[Pair]` potentials.

Note: The `SPECIES` label can take the form `ALPHA->BETA` for Finnis-Sinclair tabulations. See *Finnis Sinclair Models, below* and *[EAM-Density]* in the reference section .

Example

A density function for silver may look something like this:

```
[EAM-Density]
Ag : as.exponential 4681.013008649 -6
```

This is taken from *Sutton Ag EAM Example*. As you can determine from the parameters given there, and just for fun, this could have been defined using potential-modifiers as this, which makes the original parameters self evident:

```
[EAM-Density]
Ag : pow(
         product(as.constant 4.09,
                 pow(
                     as.polynomial 0 1,
                     as.constant -1)),
         as.constant 6)
```

[EAM-Embed]

Embedding functions are defined in this section.

Entries have the following form:

```
SPECIES : POTENTIAL_FORM PARAM_1 PARAM_2 ... PARAM_N
```

Where:

- `SPECIES` is atomic type at which the surrounding electron density will be embedded using the specified potential form.
- `POTENTIAL_FORM PARAM_1 ...`: embedding functions instantiate potential forms in the same way as in the *[Pair]* section.

Note: Embedding functions are tabulated using rho values. The resolution and extent of functions in rho are defined by `drho`, `nrho` and `cutoff_rho` in the *[Tabulation] section*.

Example

This shows the embedding function used in the *Sutton Ag EAM Example* for Ag:

```
[EAM-Embed]
Ag : product(as.constant 2.5415e-3, as.sqrt -144.41)
```

Note the use of the *product()* modifier to apply a constant multiplication factor to the square root embedding function.

Standard EAM Examples

Sutton Ag EAM Example

This provides an example of using `potable` to tabulate the Ag model given by Sutton and Chen in[1].

Potential Model

$$E_i = \epsilon \left[\frac{1}{2} \sum_{i \neq j} \sum \phi_{\alpha\beta}(r_{ij}) - c \sum_i \sqrt{\rho_i} \right] \tag{2.12}$$

Where:

- E_T is the energy of ths system.
- r_{ij} is the separation between atoms i and j.
- c and ϵ are adjustable parameters specific to interacting species.
- Inside the square brackets the first term $V(r_{ij})$ are the pair potentials.
- The second is the many body term: $c \sum_i \sqrt{\rho_i}$. Where ρ_i is the electron density.

[1] A.P. Sutton, and J. Chen, "Long-range Finnis-Sinclair potentials", *Philos. Mag. Lett.* **61** (1990) 139 doi:10.1080/09500839008206493.

Pair potential form:

$$\phi_{\alpha\beta}(r_{ij}) = (a/r_{ij})^n$$

Where:

- a and n are potential parameters.

This must be multiplied by the ϵ term from equation (2.12) above:

$$\phi_{\alpha\beta}(r_{ij}) = \epsilon(a/r_{ij})^n$$

To make things easier later on, this will be re-expressed as:

$$\phi_{\alpha\beta}(r_{ij}) = \epsilon a^n r_{ij}^n$$

This will allow this functional form to be written using the provided *as.exponential* potential-form.

Many body terms

Density function:

The density function is:

$$\rho_i = \left(\frac{a}{r_{ij}}\right)^m$$

Again to allow the use of the *as.exponential* potential-form this will be re-written as:

$$\rho_i = a^m r_{ij}^{-m}$$

Embedding function:

Examining the many-body term from (2.12) it can be seen that the embedding function is:

$$c\sqrt{\rho_i}$$

Taking the the ϵ term from outside the square brackets and pre-multiplying the expression this becomes:

$$\epsilon c\sqrt{\rho_i}$$

Potential parameters

The potential parameters for Ag are:

Table 2.3: Potential parameters for Ag

Parameter	Value
m	6
n	12
ϵ	2.5415×10^{-3} eV
a	4.09
c	144.41

Potable input

The `potable` input for this model can be downloaded as `Ag_sutton.aspot` and will now be described:

```
 1  [Tabulation]
 2  target : setfl
 3  #
 4  cutoff_rho : 600
 5  drho : 0.005
 6  #
 7  cutoff : 12.0
 8  dr : 0.001
 9
10  [EAM-Embed]
11  Ag : product(as.constant 2.5415e-3, as.sqrt -144.41)
12
13  [EAM-Density]
14  Ag : as.exponential 4681.013008649 -6
15
16  [Pair]
17  Ag-Ag : product(as.constant 2.5415e-3, as.exponential 21911882.787 -12)
```

Notes:

- **lines 1-8 [Tabulation]:**
  - **lines 4,5:** gives the resolution and extent of the function in `[EAM-Embed]`.
  - **lines 7,8:** defines resolution and extent of the tables generated for the `[Pair]` and `[EAM-Density]` functions.
- **lines 10 and 11 [EAM-Embed]:**
  - Defines the embedding function.
  - Note the use of the `product()` *potential modifier* to multiply the square root embedding function by the value of ϵ.
- **lines 13 and 14 [EAM-Density]:**
  - Describes the density function.
  - The value of 4681.013008649 is obtained as $a^m = 4.09^6$.
- **lines 16 and 17 [Pair]:**
  - Defines the pair potential component of the model.
  - As above, the `product()` *potential modifier* has been used to multiply the function by ϵ.
  - Here the first parameter to the `as.exponential` form is $a^n = 4.09^{12}$ = 21911882.78.

Making and testing the tabulation

To tabulate the potential `download the aspot file` and run it through `potable`

```
potable Ag_sutton.aspot Ag_sutton.eam.alloy
```

A LAMMPS input file is provided to allow you to test the `Ag_sutton.eam.alloy` file produced by potable. This input file can be downloaded here: `Ag_sutton_fcc.lmpin` and will energy minimize the structure and then perform an NPT MD equilibration at T=300K. Frames will be dumped every 1000 timesteps (1ps) and dumped to a LAMMPS dump file named `dump.atom`(this is suitable for visualisation in Ovito).

In terms of the table file the important part of the LAMMPS input is:

```
pair_style eam/alloy
pair_coeff * * Ag_sutton.eam.alloy Ag
```

This tells LAMMPS to accept a `setfl` formatted file (`pair_style eam/alloy`). The `Ag` at the end of the `pair_coeff` line says that LAMMPS should associate atom type 1 with the `Al` species label in the table file `Ag_sutton_eam.alloy`.

Placing both the LAMMPS and table file in the same directory run LAMMPS as follows:

```
mpirun lammps -in Ag_sutton_fcc.lmpin -log Ag_sutton_fcc.lmpout
```

Footnotes:

More complete examples of EAM tabulation are listed in the following table:

Example	Description
Sutton Ag EAM Example	An example of how to tabulate a single component EAM potential for Ag, to use in LAMMPS

Finnis-Sinclair Style EAM Models

A variation of the standard EAM is supported allowing different density functions to be specified for each pair of species. Before looking at this let's have another look at original definition of the EAM given in (2.11):

$$E_i = F_\alpha\left(\sum_{j\neq i}\rho_\beta(r_{ij})\right) + \frac{1}{2}\sum_{j\neq i}\phi_{\alpha\beta}(r_{ij})$$

Here the many body term is:

$$F_\alpha\left(\sum_{j\neq i}\rho_\beta(r_{ij})\right)$$

From this it can be seen that for any atom type β surrounding atom i, the same density function is used for β, no matter the species (α) of the central atom. So in the standard EAM, the density due to a B atom neighbouring an A atom would be calculatd by $\rho_B(r_{ij})$. Similarly a B atom next to another B atom would also have its density calculated using the same $\rho_B(r_{ij})$ function.

By comparison, the EAM variant (referred to as Finnis-Sinclair by LAMMPS) has the following form:

$$E_i = F_\alpha\left(\sum_{j\neq i}\rho_{\alpha\beta}(r_{ij})\right) + \frac{1}{2}\sum_{j\neq i}\phi_{\alpha\beta}(r_{ij}) \tag{2.13}$$

The difference is subtle, but has important implications for the expressiveness of the potential model. The density function now becomes $\rho_{\alpha\beta}(r_{ij})$ meaning it is now specific the types of the interacting species. So the density due to a B atom around an A atom would be given by $\rho_{AB}(r_{ij})$ whilst a different function $\rho_{BB}(r_{ij})$ would be used for B atoms around A atoms.

The *potable* tool allows this concept to be expressed by using a slightly different style of `[EAM-Density]` section. To define the two density functions described in the previous paragraph this would look like this:

```
[EAM-Density]
A->B : DENSITY_AB
B->B : DENSITY_BB
```

Where:

- `DENSITY_AB` and `DENSITY_BB` would be the two potential-forms for the 'B surrounding A' and 'B surrounding B' density functions.

Summarising, the `[EAM-Density]` section of Finnis-Sinclair style `potable` files has contains `SPECIES` labels of the form:

```
ALPHA->BETA
```

Where:

- `ALPHA` is the central atom species for the density function.
- `BETA` is the surrrounding atom species for the density function.

Finnis-Sinclair style models can be used with the following tabulation targets:

- `excel_eam_fs`
- `DL_POLY_EAM_fs`
- `setfl_fs`

Example

The following example aims to demonstrate the difference between the standard and Finnis-Sinclair potential models and how to tabulate them. For simplicity a 'toy' atomic configuration will be used, this is illustrated in Fig. 2.7 and can be downloaded as a LAMMPS file: `toy_structure.lmpstruct` (`atom_style charge`).

Fig. 2.7: The atomic configuration used in this example (viewed down z-axis).

The coordination environments of the A and B atoms in this structure are as follows:

Table 2.4: Coordination environment of the different atoms.

Central Atom	Surrounding Atoms	r_{ij}
A	$4 \times$ B	2.0
B	$1 \times$ A	2.0
	$1 \times$ B	4.0
	$2 \times$ B	$2\sqrt{2}$

Standard EAM

For our toy example, let's define the density due to an A atom as $\rho_A(r_{ij}) = 2r_{ij}$ and that due to B as $\rho_B(r_{ij}) = 3r_{ij}$. Using these we can extend our table to calculate the density expected around each type of atom:

Table 2.5: Density calculation

Central Atom	Surrounding Atoms	r_{ij}	$\rho_A(r_{ij}) = 2r_{ij}$	$\rho_B(r_{ij}) = 3r_{ij}$	Total ρ
A	$4 \times$ B	2.0		$4 \times 3 \times 2.0 = 24.0$	**24.0**
B	$1 \times$ A	2.0	$1 \times 2 \times 2.0 = 4.0$		4.0 + 12.0 + 16.971 = **32.971**
	$1 \times$ B	4.0		$1 \times 3 \times 4.0 = 12.0$	
	$2 \times$ B	$2\sqrt{2}$		$2 \times 3 \times 2\sqrt{2} = 16.971$	

Looking at the calculation in Table 2.5 we can see that the density around an **A** atom is **24.0** and that around **B** is **32.971**.

Let's confirm that this is the case by running a LAMMPS calculation that reproduces the hand calculation given in the table using the following `potable` input (`standard_eam.aspot`)

```
[Tabulation]
target : setfl
cutoff = 5.0
dr = 0.1
cutoff_rho = 50.0
drho = 0.1

[Species]
A.atomic_mass = 1
A.atomic_number = 1
B.atomic_mass = 2
B.atomic_number = 2

[EAM-Embed]
A = as.polynomial 0 1
B = as.zero

[EAM-Density]
A = as.polynomial 0 2
B = as.polynomial 0 3

[Pair]
```

Notes:

- The `[Species]` section has been included so we can use **A** and **B** as species labels rather than proper element names.
- The `[Pair]` section has been left empty as we haven't defined any pair potentials to use with this model.

The `[EAM-Density]` section uses the *as.polynomial* potential form to give our $\rho_A(r_{ij}) = 2r_{ij}$ and $\rho_B(r_{ij}) = 3r_{ij}$ density functions:

```
[EAM-Density]
A = as.polynomial 0 2
B = as.polynomial 0 3
```

More unusual is the use of the *as.polynomial* and *as.zero* potential forms in the `[EAM-Embed]` section. Our example is in no way trying to reproduce the physics of atomic bonding but is instead showing how the EAM and its tabulations works. As a result the `[EAM-Embed]` section looks like this:

```
[EAM-Embed]
A = as.polynomial 0 1
B = as.zero
```

Here the embedding function for **A** has been set to *as.polynomial 0 1*. This is useful for debugging as this is an identity function meaning that the 'energy' calculated by LAMMPS for each **A** atom will instead be its summed density.

The embedding function for **B** has been set to *as.zero* meaning the density around these atoms will not contribute to the value output by LAMMPS. We will turn the **B** embedding function back on later in the example, however for our first run the `potable` tabulation should result in the density surrounding the single **A** atom in Fig. 2.7 being produced. According to the calculation in Table 2.5 this should be **24.0**.

Run potable on `standard_eam.aspot` to produce a table file named `standard_eam.eam`

```
potable standard_eam.aspot standard_eam.eam
```

Now download the structure file (`toy_structure.lmpstruct`) and the following LAMMPS input script (`standard_evaluate.lmpin`) into the same directory as your table:

```
units metal
atom_style charge

read_data toy_structure.lmpstruct

pair_style eam/alloy
pair_coeff * * standard_eam.eam A B

run 0

print ENERGY:$(pe)
```

Now let's run this through LAMMPS to evaluate our table file for the example structure:

```
lammps -in standard_evaluate.lmpin  -log standard_evaluate.lmpout
```

This should produce output similar to this:

```
LAMMPS (3 Mar 2020)
units metal
atom_style charge

read_data toy_structure.lmpstruct
  orthogonal box = (0 0 0) to (20 20 20)
  1 by 1 by 1 MPI processor grid
  reading atoms ...
  5 atoms
  read_data CPU = 0.004225 secs

pair_style eam/alloy
pair_coeff * * standard_eam.eam A B

run 0
WARNING: No fixes defined, atoms won't move (../verlet.cpp:52)
Neighbor list info ...
  update every 1 steps, delay 10 steps, check yes
  max neighbors/atom: 2000, page size: 100000
  master list distance cutoff = 7.1
```

(continues on next page)

(continued from previous page)

```
  ghost atom cutoff = 7.1
  binsize = 3.55, bins = 6 6 6
  1 neighbor lists, perpetual/occasional/extra = 1 0 0
  (1) pair eam/alloy, perpetual
      attributes: half, newton on
      pair build: half/bin/atomonly/newton
      stencil: half/bin/3d/newton
      bin: standard
Per MPI rank memory allocation (min/avg/max) = 3.436 | 3.436 | 3.436 Mbytes
Step Temp E_pair E_mol TotEng Press
       0            0           24            0           24   -1602.1765
Loop time of 1e-06 on 1 procs for 0 steps with 5 atoms

100.0% CPU use with 1 MPI tasks x no OpenMP threads

MPI task timing breakdown:
Section |  min time  |  avg time  |  max time  |%varavg| %total
---------------------------------------------------------------
Pair    | 0          | 0          | 0          |   0.0 |  0.00
Neigh   | 0          | 0          | 0          |   0.0 |  0.00
Comm    | 0          | 0          | 0          |   0.0 |  0.00
Output  | 0          | 0          | 0          |   0.0 |  0.00
Modify  | 0          | 0          | 0          |   0.0 |  0.00
Other   |            | 1e-06      |            |       |100.00

Nlocal:    5 ave 5 max 5 min
Histogram: 1 0 0 0 0 0 0 0 0 0
Nghost:    0 ave 0 max 0 min
Histogram: 1 0 0 0 0 0 0 0 0 0
Neighs:    10 ave 10 max 10 min
Histogram: 1 0 0 0 0 0 0 0 0 0

Total # of neighbors = 10
Ave neighs/atom = 2
Neighbor list builds = 0
Dangerous builds = 0

print ENERGY:$(pe)
print ENERGY:24.000000000000014211
ENERGY:24.000000000000014211
Total wall time: 0:00:00
```

As you can see from the penultimate line we obtain our expected value of 24.0 for the density of the **A** atom:

```
ENERGY:24.000000000000014211
```

Now change the `[EAM-Embed]` section as follows, to turn off the **A** density and turn on the **B** density:

```
[EAM-Embed]
A = as.zero
B = as.polynomial 0 1
```

Running the same process as above we get the following `ENERGY` line in our output:

```
ENERGY:131.88225099390862738
```

Remembering that we have four **B** atoms in our system, each with an expected density of 32.971 then it can be seen that this equates well to what we would expect: 4×32.971 = 131.884.

Finnis-Sinclair

We will now extend this example to use different density functions for the AB and BA interactions. The following functions will be used:

- $\rho_{AB}(r_{ij}) = 3r_{ij}$
- $\rho_{BB}(r_{ij}) = 5r_{ij}$
- $\rho_{BA}(r_{ij}) = 2r_{ij}$

Using these functions will produce a density around the **A** atom that is the same as in the standard EAM example above. This is because the $\rho_{AB}(r_{ij})$ function is the same as the $\rho_A(r_{ij})$ used earlier. However, the function for calculating the B density surrounding a B atom is now different from the earlier example ($\rho_{BB}(r_{ij}) = 5r_{ij}$). As the $\rho_{BA}(r_{ij})$ function also matches the equivalent ρ_A function from earlier, it is only the change introduced by the new $\rho_{BB}(r_{ij}) = 5r_{ij}$ function for B-B pairs, that will change the LAMMPS output for this example. Due to the single A atom there are no density contribution from A-A pairs in this system.

The density calculation for the Finnis-Sinclair model can now be performed by hand as shown in Table 2.6.

Table 2.6: Density calculation

Central Atom	Surrounding Atoms	r_{ij}	$\rho_{AB}(r_{ij}) = 3r_{ij}$	$\rho_{BA}(r_{ij}) = 2r_{ij}$	$\rho_{BB}(r_{ij}) = 5r_{ij}$	Total ρ
A	4 × B	2.0	4 × 3 × 2.0 = 24.0			**24.0**
B	1 × A	2.0		1 × 2 × 2.0 = 4.0		4.0 + 20.0 + 28.284 = **52.284**
	1 × B	4.0			1 × 5 × 4.0 = 20.0	
	2 × B	$2\sqrt{2}$			2 × 5 × $2\sqrt{2}$ = 28.284	

This can be described in the `potable` format as follows (`finnis_sinclair_eam.aspot`):

```
[Tabulation]
target : setfl_fs
cutoff = 5.0
dr = 0.1
cutoff_rho = 50.0
drho = 0.1

[Species]
A.atomic_mass = 1
A.atomic_number = 1
B.atomic_mass = 2
B.atomic_number = 2

[EAM-Embed]
A = as.zero
B = as.polynomial 0 1

[EAM-Density]
A->B = as.polynomial 0 3
B->A = as.polynomial 0 2
```

(continues on next page)

(continued from previous page)

```
B->B = as.polynomial 0 5

[Pair]
```

Note that the `[Tabulation]` section now specifies a Finnis-Sinclair compatible tabulation target:

```
[Tabulation]
target : setfl_fs
```

The important differences between this file and the standard EAM example can be found in the `[EAM-Density]`

```
[EAM-Density]
A->B = as.polynomial 0 3
B->A = as.polynomial 0 2
B->B = as.polynomial 0 5
```

The first entry in this section defines the $\rho_{AB}(r_{ij}) = 3r_{ij}$ function and it should be apparent how the remaining density functions are specified.

Tabulate the `finnis_sinclair_eam.aspot` file:

```
potable finnis_sinclair_eam.aspot finnis_sinclair.eam.fs
```

Copy the `toy_structure.lmpstruct` structure file and the following LAMMPs input file (`finnis_sinclair_evaluate.lmpin`) to the same directory as your tabulation:

```
units metal
atom_style charge

read_data toy_structure.lmpstruct

pair_style eam/fs
pair_coeff * * finnis_sinclair.eam.fs A B

run 0

print ENERGY:$(pe)
```

Running LAMMPS to obtain the total **B** density:

```
lammps -in finnis_sinclair_evaluate.lmpin -log finnis_sinclair_evaluate.lmpout
```

gives the following output:

```
ENERGY:209.13708498984715334
```

Referring back to Table 2.6 and remembering that there are four **B** atoms in the system the value expected from the hand calculation was 52.284 × 4 = 209.136. This very closely matches the value obtained from LAMMPS.

ADP Style EAM Models

Added in: 0.4.0

The `potable` tool allows EAM models using the angular dependent potential (ADP) extension to be tabulated.

$$
\begin{aligned}
E_i &= F_\alpha\left(\sum_{j\neq i}\rho_\beta(r_{ij})\right) + \frac{1}{2}\sum_{j\neq i}\phi_{\alpha\beta}(r_{ij}) + \frac{1}{2}\sum_s(\mu_i^s)^2 + \frac{1}{2}\sum_{s,t}(\lambda_i^{st})^2 - \frac{1}{6}\nu_i^2 \\
\mu_i^s &= \sum_{j\neq i} u_{\alpha\beta}(r_{ij})r_{ij}^s \\
\lambda_i^{st} &= \sum_{j\neq i} w_{\alpha\beta}(r_{ij})r_{ij}^s r_{ij}^t \\
\nu_i &= \sum_s \lambda_i^{ss}
\end{aligned}
\tag{2.14}
$$

ADP extends the standard EAM (2.11) with additional dipole (μ_i^s) and quadrupole terms (λ_i^{st}). The dipoles and quadrupoles are defined via $u_{\alpha\beta}(r_{ij})$ and $w_{\alpha\beta}(r_{ij})$ functions respectively.

As the u and w functions are specified for pairs of species they are defined in a `potable` file in the same way as pair potentials (see *[Pair] section*). The $u_{\alpha\beta}(r_{ij})$ functions are given in the section named `[EAM-ADP-Dipole]` and the $w_{\alpha\beta}(r_{ij})$ functions appear in `[EAM-ADP-Quadrupole]`. Other than these new sections ADP models are defined in the same way as the standard EAM (see *Many body models*). Consequently the `potable` file for an ADP model minimally contains the sections:

- *[Tabulation]*
  - For ADP the `target` value should be set as `eam_adp`.
- *[EAM-Density]*
- *[EAM-Embed]*
- *[Pair]*
- *[EAM-ADP-Dipole]*
- *[EAM-ADP-Quadrupole]*

Troubleshooting Potable Input Files

Checking tabulated functions

Sometimes you will want to check if the tabulated functions defining your potential model are as they should be. Perhaps the easiest way to do this is to use the `excel` tabulation targets:

- **excel:** dumps pair potential models into an Excel spreadsheet file,
- **excel_eam:** as above but for EAM potentials,
- **excel_eam_fs:** for Finnis-Sinclair EAM potential models.

If given as the value of the `target` field in a potable input file's `[Tabulation]` section each one will dump the model into an Excel `.xlsx` formatted spreadsheet.

Depending on the type of model the spreadsheet can contain **Pair**, **EAM-Density** and **EAM-Embed** sheets. The first column of each is either the *r* or *rho* values of the function with the remaining columns giving the function values.

To enable quick checks it is often more convenient to use potable's `--override-item` option to temporarily set the target, rather than having to edit the input file. For example, it is used here to set the `excel` target:

```
potable --override-item=Tabulation:target=excel potential_model.aspot test.xlsx
```

2.3.2 Using the Python API

Python API Getting Started

The following example gives a complete python script showing how the potential API can be used to tabulate potentials for DL_POLY .

See also:

- Also see *Quick-Start* which allows you to achieve the same using the `potable` tool without requiring programming experience.

The following example (`basak_tabulate.py`) shows how the UO_2 potential model of Basak[1] can be tabulated:

- the U + O interaction within this model combines Buckingham and Morse potential forms. Although DL_POLY natively supports both potential forms they cannot be combined with the code itself. By creating a `TABLE` file the Basak model can be described to DL_POLY.
- when executed from the command line this script will write tabulated potentials into a file named `TABLE`.

```python
#! /usr/bin/env python

from atsim.potentials import Potential, plus, potentialforms
from atsim.potentials.pair_tabulation import DLPoly_PairTabulation


def makePotentialObjects():
    # O-O Interaction:
    # Buckingham
    # A = 1633.00510, rho = 0.327022, C = 3.948790
    f_OO = potentialforms.buck(1633.00510, 0.327022, 3.948790)

    # U-U Interaction:
    # Buckingham
    # A = 294.640000, rho = 0.327022, C = 0.0
    f_UU = potentialforms.buck(294.640000, 0.327022, 0.0)

    # O-U Interaction
    # Buckingham + Morse.
    # Buckingham:
    # A = 693.648700, rho = 693.648700, C = 0.0
    # Morse:
    # D0 = 0.577190, alpha = 1.6500, r0 = 2.36900
    buck_OU = potentialforms.buck(693.648700, 0.327022, 0.0)
    morse_OU = potentialforms.morse(1.6500, 2.36900, 0.577190)

    # Compose the buckingham and morse functions into a single function
    # using the atsim.potentials.plus() function
    f_OU = plus(buck_OU, morse_OU)

    # Construct list of Potential objects
    potential_objects = [
        Potential('O', 'O', f_OO),
        Potential('U', 'U', f_UU),
        Potential('O', 'U', f_OU)
    ]
```

(continues on next page)

[1] Basak, C. (2003). Classical molecular dynamics simulation of UO2 to predict thermophysical properties. *Journal of Alloys and Compounds*, **360** (1-2), 210–216. http://dx.doi.org/doi:10.1016/S0925-8388(03)00350-5

(continued from previous page)

```
    return potential_objects


def main():
    potential_objects = makePotentialObjects()
    # Tabulate into file called TABLE
    # using short-range cutoff of 6.5 Angs with grid
    # increment of 1e-3 Angs (6500 grid points)

    tabulation = DLPoly_PairTabulation(potential_objects,
                                       6.5, 6500)

    with open('TABLE', 'w') as outfile:
        tabulation.write(outfile)


if __name__ == '__main__':
    main()
```

Tabulating the Potentials

Defining the Potentials

The first step to tabulating pair potentials is to define `Potential` objects (see *Potential Objects*). Normally this involves creating a python function for the desired pair interaction before passing this to the `atsim.potentials.Potential()` constructor to provide labels for the species pair pertinent to the interaction.

- The functions `f_OO` and `f_UU` use the Buckingham form and are created using `buck()` function factory (see *Predefined Potential Forms* for more on the pre-defined forms provided):

```
def makePotentialObjects():
    # O-O Interaction:
    # Buckingham
    # A = 1633.00510, rho = 0.327022, C = 3.948790
    f_OO = potentialforms.buck(1633.00510, 0.327022, 3.948790)

    # U-U Interaction:
    # Buckingham
    # A = 294.640000, rho = 0.327022, C = 0.0
    f_UU = potentialforms.buck(294.640000, 0.327022, 0.0)
```

- The O-U interaction is a little more tricky to define as Buckingham and Morse potentials need to be combined. Pre-canned implementations of both of these are provided in `atsim.potentials.potentialforms` as `buck()` and `morse()`. Two functions are created, one for each component of the O-U interaction and stored in the `buck_OU` and `morse_OU` variables:

```
    # O-U Interaction
    # Buckingham + Morse.
    # Buckingham:
    # A = 693.648700, rho = 693.648700, C = 0.0
    # Morse:
    # D0 = 0.577190, alpha = 1.6500, r0 = 2.36900
    buck_OU = potentialforms.buck(693.648700, 0.327022, 0.0)
    morse_OU = potentialforms.morse(1.6500, 2.36900, 0.577190)
```

- These are then composed into the desired function, `f_OU`, using the *`plus()`* function (see *Combining Potential Forms*):

```
    f_OU = plus(buck_OU, morse_OU)
```

Make `TABLE` File

The table file is written from the `main()` function of `basak_tabulate.py`

```
def main():
    potential_objects = makePotentialObjects()
    # Tabulate into file called TABLE
    # using short-range cutoff of 6.5 Angs with grid
    # increment of 1e-3 Angs (6500 grid points)

    tabulation = DLPoly_PairTabulation(potential_objects,
                                       6.5, 6500)

    with open('TABLE', 'w') as outfile:
        tabulation.write(outfile)
```

- First the `makePotentialObjects()` function is called, returning a list of *`Potential`* objects that are stored in the `potential_objects` variable.
- An instance of *`DLPoly_PairTabulation`* is created by passing this list of potentials a cut-off value of 6.5Å and specifying 6500 rows (i.e. a grid increment of 0.001 Å) to its constructor:

```
    tabulation = DLPoly_PairTabulation(potential_objects,
                                       6.5, 6500)
```

- The *`write()`* method of the `Tabulation` object is then called with the file object into which the tabulation is written:

```
    with open('TABLE', 'w') as outfile:
        tabulation.write(outfile)
```

- Now run the `basak_tabulate.py` file (making sure you have *installed* `atsim.potentials` first):

```
python basak_tabulate.py
```

- This will create a DL_POLY `TABLE` file in the working directory.

Using the `TABLE` File in DL_POLY

A set of DL_POLY files are provided allowing a simple NPT molecular dynamics equilibration simulation to be run against the `TABLE` file created in the previous step using *`writePotentials`*. Copy the files linked from the following table into the same directory as the `TABLE` file:

File	Description
`CONFIG`	4×4×4 UO2:sub:2 super-cell containing 768 atoms.
`CONTROL`	Defines 300K equilibration run under NPT ensemble lasting 10ps.
`FIELD`	File defining potentials and charges.

- The FIELD file contains the directives relevant to the TABLE file:

```
UO2.cif. Supercell: 4 x 4 x 4
units eV
molecules 1
UO2.cif. Supercell: 4 x 4 x 4
nummols 1
atoms 768
             O     15.999400     -1.200000     512    0
             U    238.028910      2.400000     256    0
finish
vdw 3
O O tab
U U tab
O U tab
CLOSE
```

- The following lines define the atom multiplicity and charges (O=-1.2*e* and U=2.4*e*):

```
nummols 1
atoms 768
             O     15.999400     -1.200000     512    0
             U    238.028910      2.400000     256    0
finish
```

- The vdw section states that the O-O, U-U and O-U interactions should be read from the TABLE file:

```
vdw 3
O O tab
U U tab
O U tab
CLOSE
```

- Once all the files are in the same directory, the simulation can be started by invoking DL_POLY:

```
DLPOLY.Z
```

Quick-Start: LAMMPS

Once the potential model has been defined as a series of *Potential* creating tabulations for different codes in different formats is fairly simple. The script described in this example is given in basak_tabulate_lammps.py. This contains the same potential definition as the *previous example*, however the main() function has been modified to create a table suitable for LAMMPS :

```
def main():
    potential_objects = makePotentialObjects()
    # Tabulate into file called Basak.lmptab
    # using short-range cutoff of 6.5 Angs with grid
    # increment of 1e-3 Angs (6500 grid points)
    tabulation = LAMMPS_PairTabulation(potential_objects, 6.5, 6500) # <-- The␣
↪tabulation class has been changed

    with open('Basak.lmptab', 'w') as outfile:  # <-- Filename changed from 'TABLE'
        tabulation.write(outfile)
```

Only the two highlighted lines have been changed:

1. the first changes the tabulation class to *LAMMPS_PairTabulation*. This describes the same interface as the the previous *DLPoly_PairTabulation* class meaning it is a drop in replacement.
2. the second changes the output filename to `Basak.lmptab`

Running the file creates the `Basak.lmptab` file:

```
python basak_tabulate_lammps.py
```

Using `Basak.lmptab` in LAMMPS

LAMMPS input files are provided for use with the table file:

- `UO2.lmpstruct`: structure file for single UO:sub:*2* cell, that can be read with read_data when atom_style `full` is used.
- `equilibrate.lmpin`: input file containing LAMMPS instructions. Performs 10ps of 300K NPT equilibration, creating a 4×4×4 super-cell.

Copy these files into the same directory as `Basak.lmptab`, the simulation can then be run using:

```
lammps -in equilibrate.lmpin -log equilibrate.lmpout -echo both
```

The section of `equilibrate.lmpin` which defines the potential model and makes use of the table file is as follows:

```
variable O equal 1
variable U equal 2

set type $O charge -1.2
set type $U charge 2.4

kspace_style pppm 1.0e-6

pair_style hybrid/overlay coul/long ${SR_CUTOFF} table linear 6500␣
↪pppm
pair_coeff * * coul/long
pair_coeff $O $O table Basak.lmptab O-O
pair_coeff $O $U table Basak.lmptab O-U
pair_coeff $U $U table Basak.lmptab U-U
```

Notes:

1. As LAMMPS uses ID numbers to define species the `variable` commands associate:
   - index 1 with variable `$O`
   - index 2 with `$U` to aid readability.
3. The `set type SPECIES_ID charge` lines define the charges of oxygen and uranium.
3. Uses the hybrid/overlay `pair_style` to combine the coul/long and table styles.

   ```
   pair_style hybrid/overlay coul/long ${SR_CUTOFF} table␣
   ↪linear 6500 pppm
   ```

   - The coul/long style is used to calculate electrostatic interactions using the pppm `kspace_style` defined previously.
   - `table linear 6500 pppm`:

   - linear interpolation of table values should be used
   - all 6500 rows of the table are employed
   - corrections appropriate to the pppm `kspace_style` will be applied.

4. Means that electrostatic interactions should be calculated between all pairs of ions.

```
pair_coeff * * coul/long
```

5. Each `pair_coeff` reads an interaction from the `Basak.lmptab` file.

```
pair_coeff $O $O table Basak.lmptab O-O
pair_coeff $O $U table Basak.lmptab O-U
pair_coeff $U $U table Basak.lmptab U-U
```

- The general form is:
  - `pair_coeff SPECIES_ID_1 SPECIES_ID_2 table TABLE_FILENAME TABLE_KEYWORD`
  - Here the `SPECIES_IDs` use the `$O` and `$U` variables defines earlier.
  - `TABLE_KEYWORD` - the table file contains multiple blocks, each defining a single interaction.
  - The `TABLE_KEYWORD` is the title of the block. The *`writePotentials()`* function creates labels of the form `LABEL_A-LABEL_B` albeit with the species sorted into alphabetical order. This label format is described in greater detail *`here`*.

Pair Potential Tabulation

Pair potentials are tabulated using `PairTabulation` objects, a tabulation class is provided for each target (note an alternative procedural interface is also provided *`atsim.potentials.writePotentials()`*):

- ***`atsim.potentials.pair_tabulation.DLPoly_PairTabulation`***
  - Class for creating DL_POLY `TABLE` files.
- ***`atsim.potentials.pair_tabulation.Excel_PairTabulation`***
  - Produces Excel spreadsheet files from *`atsim.potentials.Potential`* objects. This is useful for plotting potentials and debugging purposes (also see *Troubleshooting Potable Input Files*).
- ***`atsim.potentials.pair_tabulation.GULP_PairTabulation`***
  - Suitable for producing files usable with the GULP code.
- ***`atsim.potentials.pair_tabulation.LAMMPS_PairTabulation`***
  - Produces files for use with LAMMPS pair_style table.

Using `Tabulation` objects

The constructor of `PairTabulation` classes have the following basic signature:

```
PairTabulation(self, potentials, cutoff, nr)
```

Where:

- **`potentials` is a list of `Potential` objects.**

  - `Potential` objects have *`energy()`* and *`force()`* methods called during tabulation to obtain potential-energy as a function of separation and its derivative respectively.
  - see *Example: Instantiating atsim.potentials.Potential Objects* and *Predefined Potential Forms*.
- `cutoff` is a float giving the maximum separation represented by the tabulation.
- `nr` the number of rows to be included in the tabulation.

Therefore to create a *`LAMMPS_PairTabulation`* with a 10 Å cutoff with 5000 rows from a list of potentials stored in the variable `potentials` the following would be used:

```
tabulation = LAMMPS_PairTabulation(potentials, 10, 5000)
```

The pair potential model can then be written to a file-like object using the *`write()`* method:

```
with open("tabulation.lmptab", "w") as outfile:
    tabulation.write(outfile)
```

See also:

- *Python API Getting Started* provides a complete example of using `PairTabulation` objects.

Potential Objects

Potential objects should implement the following interface:

`class PotentialInterface`

`speciesA`
(str): Attribute giving first species in pair being described by pair-potential

`speciesB`
(str): Attribute giving second species in pair described by pair-potential

`energy`(*self*, *r*)
Calculate energy between atoms for given separation.

Parameters **r** (*`float`*) – Separation between atoms of speciesA and speciesB

Returns Energy in eV for given separation.

Return type float

`force`(*self*, *r*)
Calculate force (-dU/dr) for interaction at a given separation.

Parameters **r** (*`float`*) – Separation

Returns -dU/dr at *r* in eV per Angstrom.

Return type float

In most cases the *`atsim.potentials.Potential`* class provided in *`atsim.potentials`* can be used. This wraps a python callable that returns potential energy as a function of separation to provide the values returned by the *`energy()`* method. The forces calculated by the *`force()`* method are obtained by taking the numerical derivative of the wrapped function.

Example: Instantiating `atsim.potentials.Potential` Objects

The following example shows how a Born-Mayer potential function can be described and used to create a Potential object for the interaction between Gd and O. The Born-Mayer potential is given by:

$$U_{\text{Gd-O}}(r_{ij}) = A \exp\left(\frac{-r_{ij}}{\rho}\right)$$

Where $U_{\text{Gd-O}}(r_{ij})$ is the potential energy between atoms i and j of types Gd and O, separated by r_{ij}. The parameters A and ρ will be taken as 1000.0 and 0.212.

The Gd-O potential function can be defined as:

```python
import math
from atsim.potentials import Potential

def bornMayer_Gd_O(rij):
    energy = 1000.0 * math.exp(-rij/0.212)
    return energy
```

This is then passed to `Potential`'s constructor along with the species names:

```python
pot = Potential('Gd', 'O', bornMayer_Gd_O)
```

The energy and force at a separation of 1 can then be obtained by calling the `energy()` and `force()` methods:

```python
>>> pot.energy(1.0)
8.942132960434881
>>> pot.force(1.0)
42.17987245936639
```

Predefined Potential Forms

In the previous example, a function named `bornMayer_Gd_O()` was defined for a single pair-interaction, with the potential parameters hard-coded within the function. Explicitly defining a function for each interaction quickly becomes tedious for anything but the smallest parameter sets. In order to make the creation of functions using standard potential forms easier, a set of function factories are provided within the `atsim.potentials.potentialsforms` module.

Using the `potentialsforms` module, the function:

```python
import math

def bornMayer_Gd_O(rij):
    energy = 1000.0 * math.exp(-rij/0.212)
    return energy
```

can be rewritten as:

```python
from atsim.potentials import potentialforms
bornMayer_Gd_O = potentialsforms.bornmayer(1000.0, 0.212)
```

See API reference for list of available potential forms: *`atsim.potentials.potentialforms`*

Combining Potential Forms

Pair interactions are often described using a combination of standard potential forms. This was seen for the Basak potentials used within the *Python API Getting Started* example, where the oxygen-uranium pair potential was the combination of a Buckingham and Morse potential forms. This combination was made using the `plus()` function. This returns a callable which, when invoked, returns the sum of the values returned by the callables originally passed to `plus()`.

The combination functions listed below will return a wrapped function that correctly evaluate the first and second derivatives of the combined callables. That is, when the callables provide `.deriv()` and `.deriv2()` methods, these will, where possible be used in the evaluation. In this way accurate analytical derivatives can be combined and will appear in the resulting tabulation. If any callable does not implement these methods, the system will revert to using numerical evaluation of derivatives.

- **Combination functions:**
  - **`atsim.potentials.plus()`**
    - Sum the return values of constituent callables.
  - **`atsim.potentials.pow()`**
    - Takes two functions and returns a third which when evaluated returns the result of `a(r)**b(r)`
  - **`atsim.potentials.product()`**
    - Takes two callables and returns a third which when evaluated returns the result of `a(r) * b(r)`.

Spline Interpolation

The `SplinePotential` class can be used to smoothly interpolate between two different potential forms within the same potential curve: one potential function acts below a given cutoff (referred to as the detachment point) and the other potential function takes over at larger separations (acting above a second cutoff called the attachment point). An exponential interpolating spline acts between the detachment and attachment points to provide a smooth transition between the two potential curves.

See also:

- *Splining with potable* - description of how to do splining with potable rather than using the Python API.

The `SplinePotential` class aims to automatically determine spline coefficients such that the resultant, interpolated, potential curve is continuous in its first and second derivatives. The analytical form of the interpolating spline is (where r_{ij} is interatomic separation and $B_{0..5}$ are the spline coefficients calculated by the `SplinePotential` class):

$$U(r_{ij}) = \exp\left(B_0 + B_1 r_{ij} + B_2 r_{ij}^2 + B_3 r_{ij}^3 + B_4 r_{ij}^4 + B_5 r_{ij}^5\right)$$

The `SplinePotential` has a number of applications, for example:

- certain potential forms can become attractive in an unphysical manner at small separations (an example is the so-called Buckingham catastrophe); `SplinePotential` can be used to combine an appropriate repulsive potential at short separations whilst still using the other form for equilibrium and larger separations.
- similarly different potential forms may be better able to express certain separations than others. For instance the `zbl()` potential is often used to describe the high energy interactions found in radiation damage cascades but must be combined with another potential to describe equilibrium properties.

The *atsim.potentials.spline.Buck4_SplinePotential* can also be used to connect two potential functions. The splined region of this potentialform is described via an instance of *atsim.potentials.spline.Buck4_Spline*.

Both *atsim.potentials.spline.Buck4_SplinePotential* and *atsim.potentials.spline.SplinePotential* inherit from *atsim.potentials.spline.Custom_SplinePotential*. This provides the useful property *splineCoefficients* which can be used to access the coefficients used to describe the polynomial connecting the two potential functions. These are often quoted in journal papers as they allow the same spline to be reproduced exactly by readers.

Example: Splining ZBL Potential on to Buckingham Potential

As mentioned above, for certain parameterisations, popular potential forms can exhibit unphysical behaviour for some interatomic separations.

See also:

- A version of this example which uses potable instead of the Python API is given here: *Example: splining to the zbl potential form using exp_spline*.

A popular model for the description of silicate and phosphate systems is that due to van Beest, Kramer and van Santen (the BKS potential set)[1]. In the current example, the Si-O interaction from this model will be considered. This uses the Buckingham potential form with the following parameters:

- A = 18003.7572 eV
- ρ = 0.205204
- C = 133.5381 eV ^{6}
- **Charges:**
  - Si = 2.4 *e*
  - O = -1.2 *e*

The following plot shows the combined coulomb and short-range contributions for this interaction plotted as a function of separation. The large C term necessary to describe the equilibrium properties of silicates means that as r_{ij} gets smaller, the $\frac{C}{r_{ij}^6}$ overwhelms the repulsive Born-Mayer component of the Buckingham potential meaning that it turns over. This creates only a relatively shallow minimum arround the equilibrium Si-O separation. Within simulations containing high velocities (e.g. high temperatures or collision cascades) atoms could easily enter the very negative, attractive portion of the potential at low r_{ij} - effectively allowing atoms to collapse onto each other. In order to overcome this deficiency a ZBL potential will be splined onto the Si-O interaction within this example.

The first step to using *SplinePotential* is to choose appropriate detachment and attachment points. This is perhaps best done plotting the two potential functions to be splined. The *potentials* module contains the convenience functions *atsim.potentials.plot()* and *atsim.potentials.plotToFile()* to make this task easier. The following piece of code first defines the ZBL and Buckingham potentials before plotting them into the files `zbl.dat` and `bks_buck.dat`. These files each contain two, space delimited, columns giving r_{ij} and energy, and may be easily plotted in Excel or GNU Plot.

```python
from atsim.potentials import potentialforms
import atsim.potentials

zbl = potentialforms.zbl(14, 8)
bks_buck = potentialforms.buck(18003.7572, 1.0/4.87318, 133.5381)
```

(continues on next page)

[1] Van Beest, B. W. H., Kramer, G. J., & van Santen, R. A. (1990). Force fields for silicas and aluminophosphates based on ab initio calculations. *Physical Review Letters* , **64** (16), 1955–1958. http://dx.doi.org/doi:10.1103/PhysRevLett.64.1955

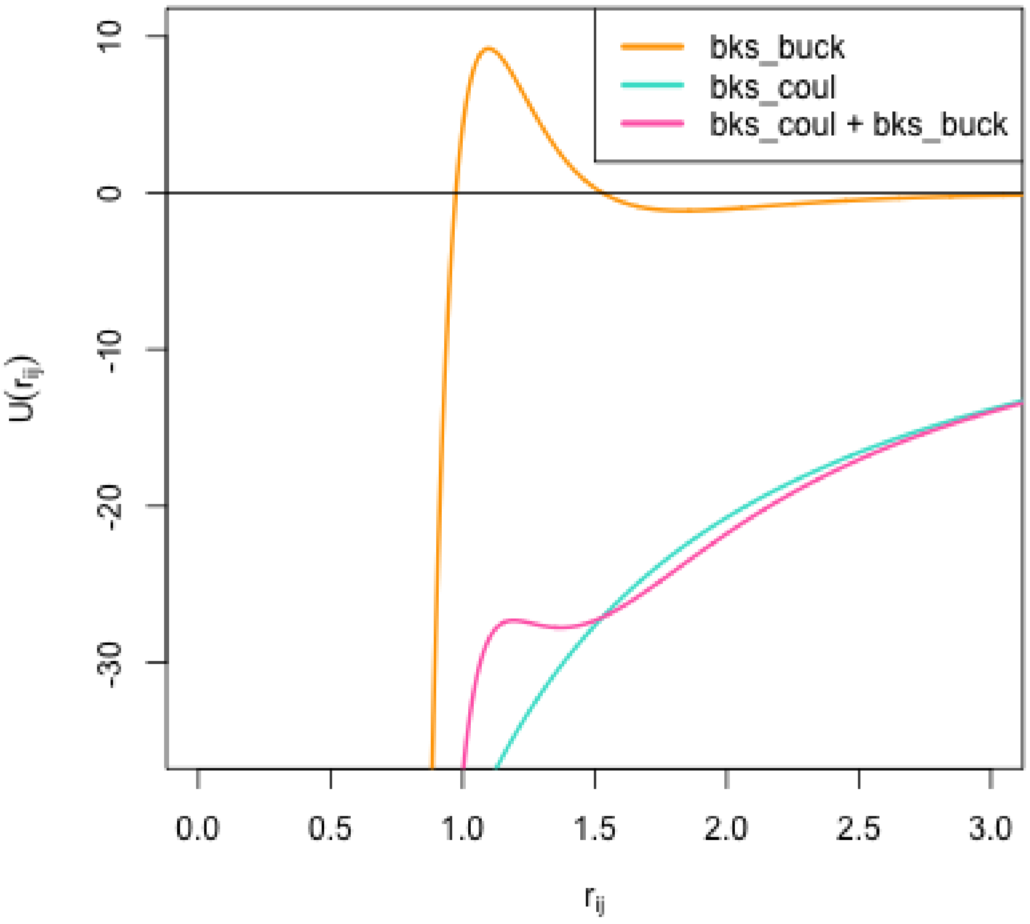

Fig. 2.8: Plot of BKS Si-O potential showing the short-range (bks_buck) component, electrostatic (bks_coul) and the effective Si-O interaction (bks_buck + bks_coul). This shows that this potential turns over at small separations making it unsuitable for use where high energies may be experienced such as high-temperature or radiation damage cascade simulations.

(continued from previous page)

```
atsim.potentials.plot( 'bks_buck.dat', 0.1, 10.0, bks_buck, 5000)
atsim.potentials.plot( 'zbl.dat', 0.1, 10.0, zbl, 5000)
```

Plotting these files show that `detachmentX` and `attachmentX` values of 0.8 and 1.4 may be appropriate. The `zbl` and `bks_buck` functions can then be splined between these points as follows:

```
spline = atsim.potentials.SplinePotential(zbl, bks_buck, 0.8, 1.4)
```

Plot data can then be created for the combined functions with the interpolating spline:

```
atsim.potentials.plot( 'spline.dat', 0.1, 10.0, spline, 5000)
```

Plotting the splined Si-O potential together with the original `buck` and `zbl` functions allows the smooth transition between the two functions to be observed, as shown in the following function:

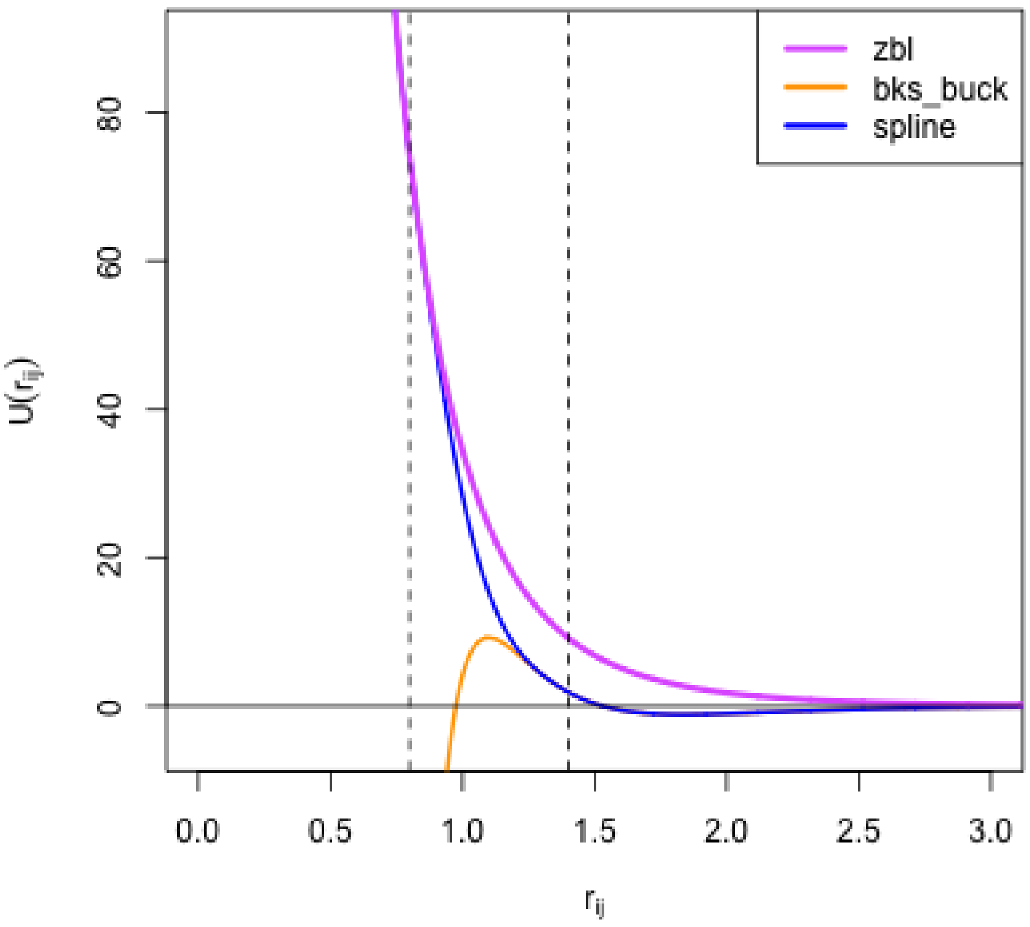

Fig. 2.9: Plot of BKS Si-O interaction showing the short-range (buck) and ZBL functions plotted with the curve generated by `SplinePotential` (spline). This joins them with a an interpolating spline acting between the detachment point at $r_{ij} = 0.8$ and re-attachment point at $r_{ij} = 1.4$ shown by dashed lines.

Finally, the potential can be tabulated in a format suitable for LAMMPS:

```
bks_SiO = atsim.potentials.Potential('Si', 'O', spline)
tabulation = atsim.potentials.pair_tabulation.LAMMPS_PairTabulation(
    [bks_SiO],
    10.0, 5000)
with open('bks_SiO.lmptab', 'w') as outfile:
    tabulation.write(outfile)
```

Procedural EAM Tabulation

As an alternative to the `EAM_Tabulation` objects described here *Embedded Atom Method (EAM) Tabulation* a procedural interface is also provided for EAM tabulation using the following functions:

Function	File-Format	Simulation Code	Example
writeFuncFL()	funcfl	LAMMPS	*Example 1: Ag in LAMMPS*
writeSetFL()	setfl	LAMMPS	*Example 2a: Al-Cu in LAMMPS*
writeTABEAM()	TABEAM	DL_POLY	*Example 2b: Al-Cu in LAMMPS*
writeSetFLFinnisSinclair()	setfl	LAMMPS	*Example 3a: Al-Fe Finnis-Sinclair in LAMMPS*
writeTABEAMFinnisSinclair()	TABEAM	DL_POLY	*Example 3b: Al-Fe Finnis-Sinclair in DL_POLY*

Examples

Example 1: Using `writeFuncFL()` to Tabulate Ag Potential for LAMMPS

This example shows how to use *writeFuncFL()* function to tabulate an EAM model for the simulation of Ag metal. How to use this tabulation within LAMMPS will then be demonstrated. The final tabulation script can be found in `eam_tabulate_example1.py`.

The same model as used for the *SetFL_EAMTabulation* example (*Example 1: Using SetFL_EAMTabulation to Tabulate Ag Potential for LAMMPS*) and is described the same way in python. In terms of the code, the only significant difference between the object based example and this one, is the use of *writeFuncFL()* to tabulate the model into a file. The output format used in this example is also different, it uses the simpler `funcfl` format. Each `funcfl` file contains a single species, making alloy systems less convenient. Further more, alloy models are simulated by combining the `funcfl` files using pre-determined mixing rules meaning there is much less control over the specific interactions between the various elements in the alloy. To prodce the same `setfl` files as produced by the *SetFL_EAMTabulation* class, the *writeSetFL()* function can be used (an example of which is given *below*).

The `embed()` and `density()` functions are defined for $F_{\mathrm{Ag}}(\rho)$ and ρ_{Ag} respectively:

```
import math
from atsim.potentials import EAMPotential
from atsim.potentials import Potential

def embed(rho):
  return -math.sqrt(rho)


def density(rij):
  if rij == 0:
```

(continues on next page)

(continued from previous page)

```
    return 0.0
  return (2.928323832 / rij) ** 6.0
```

The embedding and density functions should then be wrapped in an *EAMPotential* object to create a single item list:

```
  # Create EAMPotential
  eamPotentials = [ EAMPotential("Ag", 47, 107.8682, embed, density) ]
```

Similarly the pair potential component, $\phi_{\mathrm{Ag-Ag}}(r_{ij}$, of the model can easily be defined as:

```
def pair_AgAg(rij):
    if rij == 0:
      return 0.0
    return (2.485883762/rij) ** 12
```

This can then be wrapped in a *atsim.potentials.Potential* object to create a list of pair potentials.

```
  pairPotentials = [ Potential('Ag', 'Ag', pair_AgAg) ]
```

Note: *writeFuncFL()* only accepts a single *Potential* object and this should be the X-X interaction (where X is the species for which the `funcfl` tabulation is being created). Within the 'pair'-potential is tabulated as

$$\sqrt{\frac{\phi(r_{ij})r_{ij}}{27.2\times0.529}}$$

The numerical constants (27.2 and 0.529) convert from eV and Å into the units of Hartree and Bohr radius used by the `funcfl` format. The square rooting of the potential function is important: the simulation code effectively reconstitutes a pair potential by multiplying two of these tabulated square-rooted functions (one for each species in each interacting pair) together. If atoms i and j in an interacting pair, have the same species then effectively the original pair-potential is obtained (albeit multiplied by r_{ij}).

By comparison, if multiple `funcfl` files are used to define multiple species within a simulation (e.g. for alloy systems), then the pair potential functions of each species are effectively 'mixed' when they are multiplied together for heterogeneous atom pairs. If more control is required, with pair-potential functions specific to distinct pairs of species being necessary, then the `setfl` format produced by the *writeSetFL()* and *writeSetFLFinnisSinclair()* functions should be used instead.

Now all the components of the model have been defined a table file can be created in the `funcfl` format. Before doing this, it is necessary to choose appropriate density and separation cut-offs together with dr_{ij} and $d\rho$ increments for the density/pair functions and embedding function respectively:

- Here a $d\rho$ value of 0.001 will be used and 50000 density values tabulated.
- This means the maximum density that can be accepted by the embedding function is $49999 \times 0.001 = 49.999$
- $dr = 0.001$ Å using 12000 rows.
- The pair-potential cut-off and the maximum r_{ij} value for the density function is therefore 11.999 Å.

Invoking the *writeFuncFL()* function with these values and the *EAMPotential* and `potentialsPotential` objects, can be used to tabulate the Ag potential into the `Ag.eam` file:

```
  nrho = 50000
  drho = 0.001

  nr = 12000
```

(continues on next page)

(continued from previous page)

```
  dr = 0.001

  from atsim.potentials import writeFuncFL

  with open("Ag.eam", 'w') as outfile:
    writeFuncFL(
            nrho, drho,
            nr, dr,
            eamPotentials,
            pairPotentials,
            out= outfile,
            title='Sutton Chen Ag')
```

Putting this together the following script is obtained (this script can also be downloaded `eam_tabulate_example1.py`:

```
#! /usr/bin/env python
import math
from atsim.potentials import EAMPotential
from atsim.potentials import Potential

def embed(rho):
  return -math.sqrt(rho)


def density(rij):
  if rij == 0:
    return 0.0
  return (2.928323832 / rij) ** 6.0


def pair_AgAg(rij):
    if rij == 0:
      return 0.0
    return (2.485883762/rij) ** 12


def main():
  # Create EAMPotential
  eamPotentials = [ EAMPotential("Ag", 47, 107.8682, embed, density) ]
  pairPotentials = [ Potential('Ag', 'Ag', pair_AgAg) ]

  nrho = 50000
  drho = 0.001

  nr = 12000
  dr = 0.001

  from atsim.potentials import writeFuncFL

  with open("Ag.eam", 'w') as outfile:
    writeFuncFL(
            nrho, drho,
            nr, dr,
            eamPotentials,
            pairPotentials,
```

(continues on next page)

(continued from previous page)

```
            out= outfile,
            title='Sutton Chen Ag')

if __name__ == "__main__":
  main()
```

Running this script will produce a table file named `Ag.eam` in the same directory as the script:

```
python eam_tabulate_example1.py
```

Using the `Ag.eam` file within LAMMPS

This section of the example will now demonstrate how the table file can be used used to perform a static energy minimisation of an FCC Ag structure in LAMMPS.

Place the following in a file called `fcc.lmpstruct` in the same directory as the `Ag.eam` file you created previously. This describes a single FCC cell with a wildly inaccurate lattice parameter:

```
Title


4 atoms
1 atom types
0.0 5.000000 xlo xhi
0.0 5.000000 ylo yhi
0.0 5.000000 zlo zhi
0.000000 0.000000 0.000000 xy xz yz



Masses

1 107.86820000000000163709 #Ag

Atoms

1 0 1 0.000000 0.000000 0.000000 0.000000
2 0 1 0.000000 2.500000 2.500000 0.000000
3 0 1 0.000000 0.000000 2.500000 2.500000
4 0 1 0.000000 2.500000 0.000000 2.500000
```

The following LAMMPS input file describes a minimisation run. The lines describing potentials are highlighted. Put its contents in a file called `example1_minimize.lmpin`:

```
units metal
boundary p p p

atom_style full
read_data fcc.lmpstruct

pair_style eam
pair_coeff 1 1 Ag.eam
```

(continues on next page)

(continued from previous page)

```
fix 1 all box/relax x 0.0 y 0.0 z 0.0

minimize 0.0 1.0e-8 1000 100000
```

The `pair_style eam` command tells LAMMPS to use the EAM and expect `pair_coeff` commands mapping atom types to particular table files:

```
pair_style eam
```

The following `pair_coeff` directive indicates that the interaction between atom-type 1 (Ag) with itself should use the `funcfl` formatted file contained within `Ag.eam`:

```
pair_coeff 1 1 Ag.eam
```

The example can then be run by invoking LAMMPS:

```
lammps -in example1_minimize.lmpin
```

Example 2a: Tabulate Al-Cu Alloy Potentials Using `writeSetFL()` for LAMMPS

Within the following example the process required to generate and use a `setfl` file that tabulates the Al-Cu alloy model of Zhou et al[2]. By comparison to the `funcfl` format, `setfl` allows multiple elements to be given in the same file and additionally pair-potentials for particular pairs of interacting species can be specified (`funcfl` relies on the simulation code to 'mix' pair-potentials within alloy systems). The `eam_tabulate_example2a.py` gives a complete example of how the Zhou model can be tabulated.

This example is almost entirely the same as that given for the object based interface (*Example 2a: Tabulate Al-Cu Alloy Potentials Using SetFL_EAMTabulation for LAMMPS*) with the only difference being the use of the `writeSetFL()` function for the final tabulation. For a description of the Zhou model and how it is coded in python please *see here*.

Putting everything together gives the following script (which can also be downloaded using the following link `eam_tabulate_example2a.py`:). Running this (`python eam_tabulate_example2a.py`) produces the `Zhou_AlCu.eam.alloy` file in current working directory.

```
#! /usr/bin/env python

from atsim.potentials import writeSetFL
from atsim.potentials import Potential
from atsim.potentials import EAMPotential

import math


def makeFunc(a, b, r_e, c):
    # Creates functions of the form used for density function.
    # Functional form also forms components of pair potential.
    def func(r):
        return (a * math.exp(-b*(r/r_e - 1)))/(1+(r/r_e - c)**20.0)
    return func
```

(continues on next page)

[2] X. Zhou, R. Johnson and H. Wadley, "Misfit-energy-increasing dislocations in vapor-deposited CoFe/NiFe multilayers", *Phys. Rev. B.* **69** (2004) 144113.

(continued from previous page)

```
def makePairPotAA(A, gamma, r_e, kappa,
                  B, omega, lamda):
    # Function factory that returns functions parameterised for homogeneous pair␣
↪interactions
    f1 = makeFunc(A, gamma, r_e, kappa)
    f2 = makeFunc(B, omega, r_e, lamda)

    def func(r):
        return f1(r) - f2(r)
    return func


def makePairPotAB(dens_a, phi_aa, dens_b, phi_bb):
    # Function factory that returns functions parameterised for heterogeneous pair␣
↪interactions
    def func(r):
        return 0.5 * ((dens_b(r)/dens_a(r) * phi_aa(r)) + (dens_a(r)/dens_b(r) * phi_
↪bb(r)))
    return func


def makeEmbed(rho_e, rho_s, F_ni, F_i, F_e, eta):
    # Function factory returning parameterised embedding function.
    rho_n = 0.85*rho_e
    rho_0 = 1.15*rho_e

    def e1(rho):
        return sum([F_ni[i] * (rho/rho_n - 1)**float(i) for i in range(4)])

    def e2(rho):
        return sum([F_i[i] * (rho/rho_e - 1)**float(i) for i in range(4)])

    def e3(rho):
        return F_e * (1.0 - eta*math.log(rho/rho_s)) * (rho/rho_s)**eta

    def func(rho):
        if rho < rho_n:
            return e1(rho)
        elif rho_n <= rho < rho_0:
            return e2(rho)
        return e3(rho)
    return func


def makePotentialObjects():
    # Potential parameters
    r_eCu = 2.556162
    f_eCu = 1.554485
    gamma_Cu = 8.127620
    omega_Cu = 4.334731
    A_Cu = 0.396620
    B_Cu = 0.548085
    kappa_Cu = 0.308782
    lambda_Cu = 0.756515
```

(continues on next page)

(continued from previous page)

```
    rho_e_Cu = 21.175871
    rho_s_Cu = 21.175395
    F_ni_Cu = [-2.170269, -0.263788, 1.088878, -0.817603]
    F_i_Cu = [-2.19, 0.0, 0.561830, -2.100595]
    eta_Cu = 0.310490
    F_e_Cu = -2.186568

    r_eAl = 2.863924
    f_eAl = 1.403115
    gamma_Al = 6.613165
    omega_Al = 3.527021
    # A_Al       = 0.134873
    A_Al = 0.314873
    B_Al = 0.365551
    kappa_Al = 0.379846
    lambda_Al = 0.759692

    rho_e_Al = 20.418205
    rho_s_Al = 23.195740
    F_ni_Al = [-2.807602, -0.301435, 1.258562, -1.247604]
    F_i_Al = [-2.83, 0.0, 0.622245, -2.488244]
    eta_Al = 0.785902
    F_e_Al = -2.824528

    # Define the density functions
    dens_Cu = makeFunc(f_eCu, omega_Cu, r_eCu, lambda_Cu)
    dens_Al = makeFunc(f_eAl, omega_Al, r_eAl,  lambda_Al)

    # Finally, define embedding functions for each species
    embed_Cu = makeEmbed(rho_e_Cu, rho_s_Cu, F_ni_Cu, F_i_Cu, F_e_Cu, eta_Cu)
    embed_Al = makeEmbed(rho_e_Al, rho_s_Al, F_ni_Al, F_i_Al, F_e_Al, eta_Al)

    # Wrap them in EAMPotential objects
    eamPotentials = [
        EAMPotential("Al", 13, 26.98, embed_Al, dens_Al),
        EAMPotential("Cu", 29, 63.55, embed_Cu, dens_Cu)]

    # Define pair functions
    pair_CuCu = makePairPotAA(A_Cu, gamma_Cu, r_eCu, kappa_Cu,
                              B_Cu, omega_Cu, lambda_Cu)

    pair_AlAl = makePairPotAA(A_Al, gamma_Al, r_eAl, kappa_Al,
                              B_Al, omega_Al, lambda_Al)

    pair_AlCu = makePairPotAB(dens_Cu, pair_CuCu, dens_Al, pair_AlAl)

    # Wrap them in Potential objects
    pairPotentials = [
        Potential('Al', 'Al', pair_AlAl),
        Potential('Cu', 'Cu', pair_CuCu),
        Potential('Al', 'Cu', pair_AlCu)]

    return eamPotentials, pairPotentials


def main():
    eamPotentials, pairPotentials = makePotentialObjects()
```

(continues on next page)

(continued from previous page)

```
    # Perform tabulation
    # Make tabulation
    nrho = 2000
    drho = 0.05

    nr = 2000
    dr = 0.003

    with open("Zhou_AlCu.eam.alloy", 'w') as outfile:
        writeSetFL(
            nrho, drho,
            nr, dr,
            eamPotentials,
            pairPotentials,
            out=outfile,
            comments=['Zhou Al Cu', "", ""])  # <-- Note: title lines given as list␣
↪of three strings


if __name__ == '__main__':
    main()
```

See also:

- See *Using the Zhou_AlCu.eam.alloy file within LAMMPS* for details of how to use the tabulation file with LAMMPS.

Example 2b: Tabulate Al-Cu Alloy Potentials Using `writeTABEAM()` for DL_POLY

The tabulation script used with *Example 2a* can be easily modified to produce the `TABEAM` format expected by the DL_POLY simulation code by using the *`writeTABEAM()`*. See the tabulation script for this example: `eam_tabulate_example2b.py`.

```
def main():
  eamPotentials, pairPotentials = makePotentialObjects()

  # Perform tabulation
  # Make tabulation
  nrho = 2000
  drho = 0.05

  nr = 2000
  dr = 0.003

  with open("TABEAM", 'w') as outfile:
    writeTABEAM(
      nrho, drho,
      nr, dr,
      eamPotentials,
      pairPotentials,
      out = outfile)
```

See also:

- See the object oriented version of this example *Example 2b: Tabulate Al-Cu Alloy Potentials Using*

TABEAM_EAMTabulation for DL_POLY.

Example 3a: Tabulate Al-Fe Finnis-Sinclair Potentials Using `writeSetFLFinnisSinclair()` for LAMMPS

This example will show how to reproduce the EAM model described by Mendelev et al. for Fe segregation at grain boundaries within Al[3]. As a result this example effectively shows how to reproduce the `AlFe_mm.eam.fs` file provided with the LAMMPS source distribution using the *`writeSetFLFinnisSinclair()`* function.

The example uses the *`writeSetFLFinnisSinclair()`* function to produce files supported by the LAMMPS `pair_style eam/fs` command.

The potential model and definition of potential objects is detailed in *Example 3b: Tabulate Al-Fe Finnis-Sinclair Potentials Using TABEAM_FinnisSinclair_EAMTabulation for DL_POLY* which uses a tabulation class but is otherwise very similar to this example. Having defined the list of *`EAMPotential`* instances the *`writeSetFLFinnisSinclair()`* function is called, in this case writing the data to `Mendelev_Al_Fe.eam.fs` in the current directory:

```
def main():
  # Define list of pair potentials
  pairPotentials = [
    Potential('Al', 'Al', ppfuncAlAl),
    Potential('Al', 'Fe', ppfuncAlFe),
    Potential('Fe', 'Fe', ppfuncFeFe)]

  # Assemble the EAMPotential objects
  eamPotentials = [
    #Al
    EAMPotential('Al', 13, 26.98154, AlEmbedFunction,
      { 'Al' : AlAlDensityFunction,
        'Fe' : FeAlDensityFunction },
      latticeConstant = 4.04527,
      latticeType = 'fcc'),
    #Fe
    EAMPotential('Fe', 26, 55.845, FeEmbedFunction,
      { 'Al': FeAlDensityFunction,
        'Fe' : FeFeDensityFunction},
      latticeConstant = 2.855312,
      latticeType = 'bcc') ]

  # Number of grid points and cut-offs for tabulation.
  nrho = 10000
  drho = 3.00000000000000E-2
  nr   = 10000
  dr   = 6.50000000000000E-4

  with open("Mendelev_Al_Fe.eam.fs", "w") as outfile:
    writeSetFLFinnisSinclair(
      nrho, drho,
      nr, dr,
      eamPotentials,
      pairPotentials,
      outfile)
```

The full tabulation script can be downloaded as `eam_tabulate_example3a.py`.

[3] M.I. Mendelev, D.J. Srolovitz, G.J. Ackland, and S. Han, “Effect of Fe Segregation on the Migration of a Non-Symmetric Σ5 Tilt Grain Boundary in Al”, *J. Mater. Res.* **20** (2011) 208.

Example 3b: Tabulate Al-Fe Finnis-Sinclair Potentials Using `writeTABEAMFinnisSinclair()` for DL_POLY

Using exactly the same model definition as for *Example 3a*, the Al-Fe model can be re-tabulated for DL_POLY with minimal modification to the `main()` function. The modified version of the tabulation script can be found in `eam_tabulate_example3b.py`.

The `main()` function is given below:

```
def main():
  # Define list of pair potentials
  pairPotentials = [
    Potential('Al', 'Al', ppfuncAlAl),
    Potential('Al', 'Fe', ppfuncAlFe),
    Potential('Fe', 'Fe', ppfuncFeFe)]

  # Assemble the EAMPotential objects
  eamPotentials = [
    #Al
    EAMPotential('Al', 13, 26.98154, AlEmbedFunction,
      { 'Al' : AlAlDensityFunction,
        'Fe' : FeAlDensityFunction },
      latticeConstant = 4.04527,
      latticeType = 'fcc'),
    #Fe
    EAMPotential('Fe', 26, 55.845, FeEmbedFunction,
      { 'Al': FeAlDensityFunction,
        'Fe' : FeFeDensityFunction},
      latticeConstant = 2.855312,
      latticeType = 'bcc') ]

  # Number of grid points and cut-offs for tabulation.
  nrho = 10000
  drho = 3.00000000000000E-2
  nr   = 10000
  dr   = 6.50000000000000E-4
  cutoff = 6.5

  with open("TABEAM", "w") as outfile:
    writeTABEAMFinnisSinclair(
      nrho, drho,
      nr, dr,
      eamPotentials,
      pairPotentials,
      outfile)
```

Excluding the import statement at the top of the file, only two lines have been changed (highlighted). The first changes the filename to `TABEAM` whilst the second tells python to call *`writeTABEAMFinnisSinclair()`* instead of *`writeSetFLFinnisSinclair()`*:

```
  with open("TABEAM", "w") as outfile:
    writeTABEAMFinnisSinclair(
      nrho, drho,
      nr, dr,
      eamPotentials,
      pairPotentials,
      outfile)
```

Embedded Atom Method (EAM) Tabulation

An EAM model is defined by constructing instances of *atsim.potentials.EAMPotential* describing each species within the model. `EAMPotential` encapsulates the density and embedding functions specific to each species' many bodied interactions. In addition the purely pairwise interactions within the EAM are defined using a list of *atsim.potentials.Potential* objects.

Once the EAM model has been described in terms of *EAMPotential* and *Potential* objects it can be tabulated for specific simulation codes. This is done by using the `EAM_Tabulation` objects from the *atsim.potentials.eam_tabulation* module:

Class	Format	Simulation Code	Example
SetFL_EAMTabulation	`setfl`, pair_style eam/alloy	LAMMPS	*Example 1: Using SetFL_EAMTabulation to Tabulate Ag Potential for LAMMPS* *Example 2a: Tabulate Al-Cu Alloy Potentials Using SetFL_EAMTabulation for LAMMPS*
SetFL_FS_EAMTabulation	pair_style eam/fs	LAMMPS	*Example 3a: Tabulate Al-Fe Finnis-Sinclair Potentials Using SetFL_FS_EAMTabulation for LAMMPS*
TABEAM_EAMTabulation	`TABEAM`	DL_POLY	*Example 2b: Tabulate Al-Cu Alloy Potentials Using TABEAM_EAMTabulation for DL_POLY*
TABEAM_FinnisSinclair_EAMTabulation	`EEAM TABEAM`	DL_POLY	*Example 3b: Tabulate Al-Fe Finnis-Sinclair Potentials Using TABEAM_FinnisSinclair_EAMTabulation for DL_POLY*
Excel_EAMTabulation	`.xlsx`		
Excel_FinnisSinclair_EAMTabulation	`.xlsx`		

Even though the use of `EAM_Tabulation` objects is preferred a legacy procedural interface is also provided. This is described here: *Procedural EAM Tabulation*.

Examples

Example 1: Using `SetFL_EAMTabulation` to Tabulate Ag Potential for LAMMPS

This example shows how to use the *SetFL_EAMTabulation* class to tabulate an EAM model for the simulation of Ag metal. How to use this tabulation within LAMMPS will then be demonstrated. The final tabulation script can be found in `eam_example1.py`.

See also:

- A `potable` version of this example is given here: *Sutton Ag EAM Example*.

Model Description

Within this example the Ag potential of Sutton will be tabulated[1]. Within the EAM the energy (E_i) of an atom i whose species is α is given by:

$$E_i = F_\alpha \left(\sum_{j \neq i} \rho_\beta(r_{ij}) \right) + \frac{1}{2} \sum_{j \neq i} \phi_{\alpha\beta}(r_{ij})$$

Note:

- $\rho_\beta(r_{ij})$ is the density function which gives the electron density for atom j with species β as a function of its separation from atom i, r_{ij}.
- The electron density for atom i is obtained by summing over the density ($\rho_\beta(r_{ij})$ contributions due to its neighbours.
- The embedding function $F_\alpha(\rho)$ is used to calculate the many-bodied energy contribution from this summed electron density.
- The sum $\frac{1}{2} \sum_{j \neq i} \phi_{\alpha\beta}(r_{ij})$ gives the pair-potential contribution to atom i's energy.
- $\phi_{\alpha\beta}(r_{ij})$ are simply pair potentials that describe the energy between two atoms as a function of their separation.

The embedding function used by Sutton is:

$$F_\alpha(\rho) = -\sqrt{\rho}$$

and the density function is:

$$\rho_\beta(r_{ij}) = \left(\frac{a}{r_{ij}} \right)^m$$

whilst pair interactions are given by:

$$\phi_{\alpha\beta}(r_{ij}) = \left(\frac{b}{r_{ij}} \right)^n$$

The model parameters are given as:

Parameter	Value
m	6
n	12
a	$2.928323832 \text{Å} \text{eV}^{\frac{1}{3}}$
b	$2.485883762 \text{eV}^{\frac{1}{12}} \text{Å}$

[1] A.P. Sutton, and J. Chen, "Long-range Finnis-Sinclair potentials", *Philos. Mag. Lett.* **61** (1990) 139 doi:10.1080/09500839008206493.

Define the Model

It is now necessary to describe the model in python code. Hard-coding the model parameters from the previous table, `embed()` and `density()` functions can be defined for $F_{\mathrm{Ag}}(\rho)$ and ρ_{Ag} respectively:

```python
import math

from atsim.potentials import EAMPotential, Potential
from atsim.potentials.eam_tabulation import SetFL_EAMTabulation


def embed(rho):
    return -math.sqrt(rho)


def density(rij):
    if rij == 0:
        return 0.0
    return (2.928323832 / rij) ** 6.0
```

The embedding and density functions should then be wrapped in an *EAMPotential* object to create a single item list:

```python
    eam_potentials = [EAMPotential("Ag", 47, 107.8682, embed, density)]
```

Similarly the pair potential component, $\phi_{\mathrm{Ag-Ag}}(r_{ij})$, of the model can easily be defined as:

```python
def pair_AgAg(rij):
    if rij == 0:
        return 0.0
    return (2.485883762/rij) ** 12
```

This can then be wrapped in a *atsim.potentials.Potential* object to create a list of pair potentials.

```python
    pair_potentials = [Potential('Ag', 'Ag', pair_AgAg)]
```

Now all the components of the model have been defined a table file can be created in the `setfl` format. Before doing this, it is necessary to choose appropriate density and separation cut-offs together with the number of rows in the density/pair functions (`nr`) and embedding function (`nrho`) respectively:

- Here 50000 density values will be tabulated to a cutoff of 50.0.
- The pair-potential cut-off and the maximum r_{ij} value for the density function is 12 Å and both will have 12000 rows.

An instance of *atsim.potentials.eam_tabulation.SetFL_EAMTabulation* is created with the *EAMPotential* and *Potential* objects. This object is then used to tabulate the Ag potential by calling the *write()* method with the `Ag.eam.alloy` file object:

```python
    pair_potentials = [Potential('Ag', 'Ag', pair_AgAg)]

    cutoff_rho = 50.0
    nrho = 50000

    cutoff = 12.0
    nr = 12000
```

(continues on next page)

(continued from previous page)

```
    tabulation = SetFL_EAMTabulation(
        pair_potentials,
        eam_potentials,
        cutoff, nr,
        cutoff_rho, nrho)

    with open("Ag.eam.alloy", 'w') as outfile:
        tabulation.write(outfile)
```

Putting this together the following script is obtained (this script can also be downloaded `eam_example1.py`:

```python
#! /usr/bin/env python
import math

from atsim.potentials import EAMPotential, Potential
from atsim.potentials.eam_tabulation import SetFL_EAMTabulation


def embed(rho):
    return -math.sqrt(rho)


def density(rij):
    if rij == 0:
        return 0.0
    return (2.928323832 / rij) ** 6.0


def pair_AgAg(rij):
    if rij == 0:
        return 0.0
    return (2.485883762/rij) ** 12


def main():
    # Create EAMPotential
    eam_potentials = [EAMPotential("Ag", 47, 107.8682, embed, density)]
    pair_potentials = [Potential('Ag', 'Ag', pair_AgAg)]

    cutoff_rho = 50.0
    nrho = 50000

    cutoff = 12.0
    nr = 12000

    tabulation = SetFL_EAMTabulation(
        pair_potentials,
        eam_potentials,
        cutoff, nr,
        cutoff_rho, nrho)

    with open("Ag.eam.alloy", 'w') as outfile:
        tabulation.write(outfile)


if __name__ == "__main__":
    main()
```

Running this script will produce a table file named `Ag.eam.alloy` in the same directory as the script:

```
python eam_example1.py
```

Using the `Ag.eam.alloy` file within LAMMPS

This section of the example will now demonstrate how the table file can be used used to perform a static energy minimisation of an FCC Ag structure in LAMMPS.

Place the following in a file called `fcc.lmpstruct` in the same directory as the `Ag.eam.alloy` file you created previously. This describes a single FCC cell with a wildly inaccurate lattice parameter:

```
Title


4 atoms
1 atom types
0.0 5.000000 xlo xhi
0.0 5.000000 ylo yhi
0.0 5.000000 zlo zhi
0.000000 0.000000 0.000000 xy xz yz



Masses

1 107.86820000000000163709 #Ag

Atoms

1 0 1 0.000000 0.000000 0.000000 0.000000
2 0 1 0.000000 2.500000 2.500000 0.000000
3 0 1 0.000000 0.000000 2.500000 2.500000
4 0 1 0.000000 2.500000 0.000000 2.500000

```

The following LAMMPS input file describes a minimisation run. The lines describing potentials are highlighted. Put its contents in a file called `example_eam_alloy_minimize.lmpin`:

```
units metal
boundary p p p

atom_style full
read_data fcc.lmpstruct

pair_style eam/alloy
pair_coeff * * Ag.eam.alloy Ag

fix 1 all box/relax x 0.0 y 0.0 z 0.0

minimize 0.0 1.0e-8 1000 100000
```

The `pair_style eam/alloy` command tells LAMMPS to use the EAM and expect `pair_coeff` commands mapping atom types to particular table files:

```
pair_style eam/alloy
```

The following `pair_coeff` directive indicates that the interaction between atom-type 1 (Ag) with itself should use the `setfl` formatted file contained within `Ag.eam.alloy`. The `Ag` label at the end of the line indicates that atom-type 1 should be associated with this label in the table file:

```
pair_coeff * * Ag.eam.alloy Ag
```

The example can then be run by invoking LAMMPS:

```
lammps -in example_eam_alloy_minimize.lmpin
```

Example 2a: Tabulate Al-Cu Alloy Potentials Using `SetFL_EAMTabulation` for LAMMPS

Within the following example the process required to generate and use a `setfl` file that tabulates the Al-Cu alloy model of Zhou et al[2]. In comparison to the previous example this example contains density and embedding functions for multiple elements and includes pair-potentials specific to pairs of interacting species. The `eam_example2a.py` file gives a complete example of how the Zhou model can be tabulated.

Model Description

The model makes use of the EAM as described above (see Example 1 *Model Description*) . The density function, $\rho_\beta(r_{ij})$ for an atom j of species β separated from atom i by r_{ij} is:

$$\rho_\beta(r_{ij}) = \frac{f_e \exp\left[-\omega(r_{ij}/r_e - 1)\right]}{1 + (r_{ij}/r_e - \lambda)^{20}}$$

where f_e, r_e, ω and λ are parameters specific to species β. The pair-potential function acting between species α–β is obtained by combining the density functions of the interacting species:

$$\phi_{\alpha\beta}(r_{ij}) = \frac{1}{2}\left[\frac{\rho_\beta(r_{ij})}{\rho_\alpha(r_{ij})}\phi_{\alpha\alpha}(r_{ij}) + \frac{\rho_\alpha(r_{ij})}{\rho_\beta(r_{ij})}\phi_{\beta\beta}(r_{ij})\right]$$

The homogeneous pair-interactions, $\phi_{\alpha\alpha}(r_{ij})$ and $\phi_{\beta\beta}(r_{ij})$ have the form:

$$\phi_{\alpha\alpha}(r_{ij}) = \frac{A\exp\left[-\gamma(r_{ij}/r_e - 1)\right]}{1 + (r_{ij}/r_e - \kappa)^{20}} - \frac{B\exp\left[-\omega(r_{ij}/r_e - 1)\right]}{1 + (r_{ij}/r_e - \lambda)^{20}}$$

again, A, B, γ, ω, κ and ω are parameters specific to the species α.

The embedding function for each species, $F_\alpha(\rho)$, is defined over three density ranges using the following:

$$F_\alpha(\rho) = \begin{cases} \sum_{i=0}^{3} F_{ni}\left(\frac{\rho}{\rho_n} - 1\right)^i & \rho < \rho_n, \\ \rho_n = 0.85\rho_e & \\ \sum_{i=0}^{3} F_i\left(\frac{\rho}{\rho_e} - 1\right)^i & \rho_n \le \rho < \rho_0, \\ \rho_0 = 1.15\rho_e & \\ F_e\left[1 - \eta\ln\left(\frac{\rho}{\rho_s}\right)\right]\left(\frac{\rho}{\rho_s}\right)^\eta & \rho_0 \le \rho \end{cases}$$

The model parameters for Cu and Al are given in the following table:

2 X. Zhou, R. Johnson and H. Wadley, "Misfit-energy-increasing dislocations in vapor-deposited CoFe/NiFe multilayers", *Phys. Rev. B.* **69** (2004) 144113.

Parameter	Cu	Al
r_e	2.556162	2.863924
f_e	1.554485	1.403115
ρ_e	21.175871	20.418205
ρ_s	21.175395	23.195740
γ	8.127620	6.613165
ω	4.334731	3.527021
A	0.396620	0.314873
B	0.548085	0.365551
κ	0.308782	0.379846
λ	0.756515	0.759692
F_{n0}	-2.170269	-2.807602
F_{n1}	-0.263788	-0.301435
F_{n2}	1.088878	1.258562
F_{n3}	-0.817603	-1.247604
F_0	-2.19	-2.83
F_1	0	0
F_2	0.561830	0.622245
F_3	-2.100595	-2.488244
η	0.310490	0.785902
F_e	-2.186568	-2.824528

Note: The Al A value is given as 0.134873 in Zhou's original *Phys. Rev. B* paper. However parameter file provided by Zhou for this model, at http://www.ctcms.nist.gov/potentials/Zhou04.html gives the parameter as 0.314873. It is this latter value that is used here.

In addition the final term of the embedding function has been modified to match that used in fortran tabulation code also provided at http://www.ctcms.nist.gov/potentials/Zhou04.html

Define the Model

A series of python functions are defined to describe the embedding, density and pair interaction functions. To encourage code re-use a number of function factories are defined. Using the parameters passed to them they return specialised functions appropriate for the parameters. The given factory functions make use of python's support for closures in their implementation.

The `makeFunc()` factory function is used to define density functions. As this functional form is also used as a component of the pair-potentials `makeFunc()` is re-used within the `makePairPotAA()` factory function.

```
def makeFunc(a, b, r_e, c):
    # Creates functions of the form used for density function.
    # Functional form also forms components of pair potential.
    def func(r):
        return (a * math.exp(-b*(r/r_e - 1)))/(1+(r/r_e - c)**20.0)
    return func
```

The following factory returns the functions used to describe the homogeneous Al-Al and Cu-Cu pair-potential interactions:

```
def makePairPotAA(A, gamma, r_e, kappa,
                  B, omega, lamda):
```

(continues on next page)

(continued from previous page)

```
    # Function factory that returns functions parameterised for homogeneous pair␣
↪interactions
    f1 = makeFunc(A, gamma, r_e, kappa)
    f2 = makeFunc(B, omega, r_e, lamda)

    def func(r):
        return f1(r) - f2(r)
    return func
```

Whilst `makePairPotAB()` describes the Al-Cu pair-potential:

```
def makePairPotAB(dens_a, phi_aa, dens_b, phi_bb):
    # Function factory that returns functions parameterised for heterogeneous pair␣
↪interactions
    def func(r):
        return 0.5 * ((dens_b(r)/dens_a(r) * phi_aa(r)) + (dens_a(r)/dens_b(r) * phi_
↪bb(r)))
    return func
```

The `makeEmbed()` function describes the embedding function:

```
def makeEmbed(rho_e, rho_s, F_ni, F_i, F_e, eta):
    # Function factory returning parameterised embedding function.
    rho_n = 0.85*rho_e
    rho_0 = 1.15*rho_e

    def e1(rho):
        return sum([F_ni[i] * (rho/rho_n - 1)**float(i) for i in range(4)])

    def e2(rho):
        return sum([F_i[i] * (rho/rho_e - 1)**float(i) for i in range(4)])

    def e3(rho):
        return F_e * (1.0 - eta*math.log(rho/rho_s)) * (rho/rho_s)**eta

    def func(rho):
        if rho < rho_n:
            return e1(rho)
        elif rho_n <= rho < rho_0:
            return e2(rho)
        return e3(rho)
    return func
```

Lists of *`EAMPotential`* and *`Potential`* objects are created and returned as a tuple by the `makePotentialObjects()` function within `eam_example2a.py`. Before invoking the factory functions we just defined, the model parameters are assigned to easily identifiable variables within this function:

```
def makePotentialObjects():
    # Potential parameters
    r_eCu = 2.556162
    f_eCu = 1.554485
    gamma_Cu = 8.127620
    omega_Cu = 4.334731
    A_Cu = 0.396620
    B_Cu = 0.548085
    kappa_Cu = 0.308782
```

(continues on next page)

(continued from previous page)

```
    lambda_Cu = 0.756515

    rho_e_Cu = 21.175871
    rho_s_Cu = 21.175395
    F_ni_Cu = [-2.170269, -0.263788, 1.088878, -0.817603]
    F_i_Cu = [-2.19, 0.0, 0.561830, -2.100595]
    eta_Cu = 0.310490
    F_e_Cu = -2.186568

    r_eAl = 2.863924
    f_eAl = 1.403115
    gamma_Al = 6.613165
    omega_Al = 3.527021
    # A_Al       = 0.134873
    A_Al = 0.314873
    B_Al = 0.365551
    kappa_Al = 0.379846
    lambda_Al = 0.759692

    rho_e_Al = 20.418205
    rho_s_Al = 23.195740
    F_ni_Al = [-2.807602, -0.301435, 1.258562, -1.247604]
    F_i_Al = [-2.83, 0.0, 0.622245, -2.488244]
    eta_Al = 0.785902
    F_e_Al = -2.824528

    # Define the density functions
    dens_Cu = makeFunc(f_eCu, omega_Cu, r_eCu, lambda_Cu)
    dens_Al = makeFunc(f_eAl, omega_Al, r_eAl,  lambda_Al)

    # Finally, define embedding functions for each species
    embed_Cu = makeEmbed(rho_e_Cu, rho_s_Cu, F_ni_Cu, F_i_Cu, F_e_Cu, eta_Cu)
    embed_Al = makeEmbed(rho_e_Al, rho_s_Al, F_ni_Al, F_i_Al, F_e_Al, eta_Al)

    # Wrap them in EAMPotential objects
    eam_potentials = [
        EAMPotential("Al", 13, 26.98, embed_Al, dens_Al),
        EAMPotential("Cu", 29, 63.55, embed_Cu, dens_Cu)]

    # Define pair functions
    pair_CuCu = makePairPotAA(A_Cu, gamma_Cu, r_eCu, kappa_Cu,
                              B_Cu, omega_Cu, lambda_Cu)

    pair_AlAl = makePairPotAA(A_Al, gamma_Al, r_eAl, kappa_Al,
                              B_Al, omega_Al, lambda_Al)

    pair_AlCu = makePairPotAB(dens_Cu, pair_CuCu, dens_Al, pair_AlAl)

    # Wrap them in Potential objects
    pair_potentials = [
        Potential('Al', 'Al', pair_AlAl),
        Potential('Cu', 'Cu', pair_CuCu),
        Potential('Al', 'Cu', pair_AlCu)]

    return eam_potentials, pair_potentials
```

Now the functions required by the `EAMPotential` instances for Al and Cu can be created:

```
    # Define the density functions
    dens_Cu = makeFunc(f_eCu, omega_Cu, r_eCu, lambda_Cu)
    dens_Al = makeFunc(f_eAl, omega_Al, r_eAl,  lambda_Al)

    # Finally, define embedding functions for each species
    embed_Cu = makeEmbed(rho_e_Cu, rho_s_Cu, F_ni_Cu, F_i_Cu, F_e_Cu, eta_Cu)
    embed_Al = makeEmbed(rho_e_Al, rho_s_Al, F_ni_Al, F_i_Al, F_e_Al, eta_Al)
```

Now these are wrapped up in `EAMPotential` objects to give the `eamPotentials` list:

```
    eam_potentials = [
        EAMPotential("Al", 13, 26.98, embed_Al, dens_Al),
        EAMPotential("Cu", 29, 63.55, embed_Cu, dens_Cu)]
```

Similarly, using the `makePairPotAA()` and `makePairPotAB()` function factories the `Potential` objects required for the tabulation are defined:

```
    # Define pair functions
    pair_CuCu = makePairPotAA(A_Cu, gamma_Cu, r_eCu, kappa_Cu,
                              B_Cu, omega_Cu, lambda_Cu)

    pair_AlAl = makePairPotAA(A_Al, gamma_Al, r_eAl, kappa_Al,
                              B_Al, omega_Al, lambda_Al)

    pair_AlCu = makePairPotAB(dens_Cu, pair_CuCu, dens_Al, pair_AlAl)

    # Wrap them in Potential objects
    pair_potentials = [
        Potential('Al', 'Al', pair_AlAl),
        Potential('Cu', 'Cu', pair_CuCu),
        Potential('Al', 'Cu', pair_AlCu)]
```

Now we have all the objects required for `SetFL_EAMTabulation`. The next excerpt calls `makeObjects()` to get the EAM and pair-potential objects before creating the tabulation object, and invoking its `write()` method to write the data into a file called `Zhou_AlCu.eam.alloy`:

```
def main():
    eam_potentials, pair_potentials = makePotentialObjects()

    # Perform tabulation
    # Make tabulation
    cutoff_rho = 100.0
    nrho = 2000

    cutoff = 6.0
    nr = 2000

    tabulation = SetFL_EAMTabulation(
        pair_potentials,
        eam_potentials,
        cutoff, nr,
        cutoff_rho, nrho
    )

    with open("Zhou_AlCu.eam.alloy", 'w') as outfile:
        tabulation.write(outfile)
```

Putting this all together gives the following script (which can also be downloaded using the following link eam_example2a.py:). Running this (python eam_example2a.py) produces the Zhou_AlCu.eam.alloy file in current working directory.

```
#! /usr/bin/env python

import math

from atsim.potentials import EAMPotential, Potential
from atsim.potentials.eam_tabulation import SetFL_EAMTabulation


def makeFunc(a, b, r_e, c):
    # Creates functions of the form used for density function.
    # Functional form also forms components of pair potential.
    def func(r):
        return (a * math.exp(-b*(r/r_e - 1)))/(1+(r/r_e - c)**20.0)
    return func


def makePairPotAA(A, gamma, r_e, kappa,
                  B, omega, lamda):
    # Function factory that returns functions parameterised for homogeneous pair␣
↪interactions
    f1 = makeFunc(A, gamma, r_e, kappa)
    f2 = makeFunc(B, omega, r_e, lamda)

    def func(r):
        return f1(r) - f2(r)
    return func


def makePairPotAB(dens_a, phi_aa, dens_b, phi_bb):
    # Function factory that returns functions parameterised for heterogeneous pair␣
↪interactions
    def func(r):
        return 0.5 * ((dens_b(r)/dens_a(r) * phi_aa(r)) + (dens_a(r)/dens_b(r) * phi_
↪bb(r)))
    return func


def makeEmbed(rho_e, rho_s, F_ni, F_i, F_e, eta):
    # Function factory returning parameterised embedding function.
    rho_n = 0.85*rho_e
    rho_0 = 1.15*rho_e

    def e1(rho):
        return sum([F_ni[i] * (rho/rho_n - 1)**float(i) for i in range(4)])

    def e2(rho):
        return sum([F_i[i] * (rho/rho_e - 1)**float(i) for i in range(4)])

    def e3(rho):
        return F_e * (1.0 - eta*math.log(rho/rho_s)) * (rho/rho_s)**eta

    def func(rho):
        if rho < rho_n:
            return e1(rho)
        elif rho_n <= rho < rho_0:
```

(continues on next page)

(continued from previous page)

```
            return e2(rho)
        return e3(rho)
    return func


def makePotentialObjects():
    # Potential parameters
    r_eCu = 2.556162
    f_eCu = 1.554485
    gamma_Cu = 8.127620
    omega_Cu = 4.334731
    A_Cu = 0.396620
    B_Cu = 0.548085
    kappa_Cu = 0.308782
    lambda_Cu = 0.756515

    rho_e_Cu = 21.175871
    rho_s_Cu = 21.175395
    F_ni_Cu = [-2.170269, -0.263788, 1.088878, -0.817603]
    F_i_Cu = [-2.19, 0.0, 0.561830, -2.100595]
    eta_Cu = 0.310490
    F_e_Cu = -2.186568

    r_eAl = 2.863924
    f_eAl = 1.403115
    gamma_Al = 6.613165
    omega_Al = 3.527021
    # A_Al       = 0.134873
    A_Al = 0.314873
    B_Al = 0.365551
    kappa_Al = 0.379846
    lambda_Al = 0.759692

    rho_e_Al = 20.418205
    rho_s_Al = 23.195740
    F_ni_Al = [-2.807602, -0.301435, 1.258562, -1.247604]
    F_i_Al = [-2.83, 0.0, 0.622245, -2.488244]
    eta_Al = 0.785902
    F_e_Al = -2.824528

    # Define the density functions
    dens_Cu = makeFunc(f_eCu, omega_Cu, r_eCu, lambda_Cu)
    dens_Al = makeFunc(f_eAl, omega_Al, r_eAl,  lambda_Al)

    # Finally, define embedding functions for each species
    embed_Cu = makeEmbed(rho_e_Cu, rho_s_Cu, F_ni_Cu, F_i_Cu, F_e_Cu, eta_Cu)
    embed_Al = makeEmbed(rho_e_Al, rho_s_Al, F_ni_Al, F_i_Al, F_e_Al, eta_Al)

    # Wrap them in EAMPotential objects
    eam_potentials = [
        EAMPotential("Al", 13, 26.98, embed_Al, dens_Al),
        EAMPotential("Cu", 29, 63.55, embed_Cu, dens_Cu)]

    # Define pair functions
    pair_CuCu = makePairPotAA(A_Cu, gamma_Cu, r_eCu, kappa_Cu,
                              B_Cu, omega_Cu, lambda_Cu)
```

(continues on next page)

(continued from previous page)

```
    pair_AlAl = makePairPotAA(A_Al, gamma_Al, r_eAl, kappa_Al,
                              B_Al, omega_Al, lambda_Al)

    pair_AlCu = makePairPotAB(dens_Cu, pair_CuCu, dens_Al, pair_AlAl)

    # Wrap them in Potential objects
    pair_potentials = [
        Potential('Al', 'Al', pair_AlAl),
        Potential('Cu', 'Cu', pair_CuCu),
        Potential('Al', 'Cu', pair_AlCu)]

    return eam_potentials, pair_potentials


def main():
    eam_potentials, pair_potentials = makePotentialObjects()

    # Perform tabulation
    # Make tabulation
    cutoff_rho = 100.0
    nrho = 2000

    cutoff = 6.0
    nr = 2000

    tabulation = SetFL_EAMTabulation(
        pair_potentials,
        eam_potentials,
        cutoff, nr,
        cutoff_rho, nrho
    )

    with open("Zhou_AlCu.eam.alloy", 'w') as outfile:
        tabulation.write(outfile)


if __name__ == '__main__':
    main()
```

Using the `Zhou_AlCu.eam.alloy` file within LAMMPS

Within LAMMPS the `setfl` files generated by *`SetFL_EAMTabulation`* are used with the eam/alloy pair_style. The `pair_coeff` directive used with this `pair_style` effectively maps LAMMPS species numbers to the element names within the table file.

Single Element Systems

Assuming a LAMMPS system containing only Al (i.e. Al is species 1) then the `pair_style` and `pair_coeff` directives would be given as:

```
pair_style eam/alloy
pair_coeff * * Zhou_AlCu.eam.alloy Al
```

Likewise if a copper system was being simulated:

```
pair_style eam/alloy
pair_coeff * * Zhou_AlCu.eam.alloy Cu
```

Mixed Al-Cu System

For an Al-Cu system where Al is species 1 and Cu species 2 then the directives would be:

```
pair_style eam/alloy
pair_coeff * * Zhou_AlCu.eam.alloy Al Cu
```

Or if Cu was 1 and Al 2:

```
pair_style eam/alloy
pair_coeff * * Zhou_AlCu.eam.alloy Cu Al
```

Example 2b: Tabulate Al-Cu Alloy Potentials Using `TABEAM_EAMTabulation` for DL_POLY

The tabulation script used with *Example 2a* can be easily modified to produce the `TABEAM` format expected by the DL_POLY simulation code. See the tabulation script for this example: `eam_example2b.py`.

The *`EAMPotential`* and *`Potential`* lists are created in exactly the same way as *Example 2a*, however rather than creating an instance of *`SetFL_EAMTabulation`* in the `main()` function it is modified to use the DL_POLY specific *`TABEAM_EAMTabulation`* class instead and to write into a file named `TABEAM`. The `main()` function of `eam_example2b.py` is now given:

```
def main():
    eamPotentials, pairPotentials = makePotentialObjects()

    # Perform tabulation
    # Make tabulation
    cutoff_rho = 100.0
    nrho = 2000

    cutoff = 6
    nr = 2000

    tabulation = TABEAM_EAMTabulation(
        pairPotentials, eamPotentials, cutoff, nr, cutoff_rho, nrho)

    with open("TABEAM", 'w') as outfile:
        tabulation.write(outfile)
```

Using the `TABEAM` file with DL_POLY

Running `eam_example2b.py` will create a file named `TABEAM` in the working directory. This should be copied into the simulation directory containing the DL_POLY input files (`CONTROL`, `CONFIG` and `FIELD`).

The following should be added at the bottom of the `FIELD` file:

```
metal 3
Al Al eam
Cu Cu eam
Al Cu eam
```

Example 3a: Tabulate Al-Fe Finnis-Sinclair Potentials Using `SetFL_FS_EAMTabulation` for LAMMPS

This example will show how to reproduce the EAM model described by Mendelev et al. for Fe segregation at grain boundaries within Al[3]. As a result this example effectively shows how to reproduce the `AlFe_mm.eam.fs` file provided with the LAMMPS source distribution using the *`SetFL_FS_EAMTabulation`* class.

See also:

- *Finnis-Sinclair example using potable*
- *Example 3a: Tabulate Al-Fe Finnis-Sinclair Potentials Using writeSetFLFinnisSinclair() for LAMMPS*

The file format created by *`atsim.potentials.eam_tabulation.SetFL_FS_EAMTabulation`* is supported by the LAMMPS `pair_style eam/fs` command. This adds an additional level of flexibility in comparison to the `eam/alloy` style; when calculating the density surrounding an atom with species α, each neighbouring atom's contribution to the density is calculated as a function of its separation from the central atom using $\rho_{\alpha\beta}(r_{ij})$. This means that the density function is now specific to both the central atom species, α **and** that of the surrounding atom, β. By comparison when using `eam/alloy` tabulations the same $\rho_{\beta}(r_{ij})$ function is used, no matter the type of the central atom. This means that the equation describing `eam/fs` style models becomes:

$$E_i = F_\alpha\left(\sum_{j\neq i}\rho_{\alpha\beta}(r_{ij})\right) + \frac{1}{2}\sum_{j\neq i}\phi_{\alpha\beta}(r_{ij})$$

Here a binary Al, Fe, model is being described and the resultant `eam/fs` file should contain definitions for the following:

- **Pair-Potentials**: $\phi_{\mathrm{AlAl}}(r_{ij})$, $\phi_{\mathrm{FeFe}}(r_{ij})$ and $\phi_{\mathrm{AlFe}}(r_{ij})$.
- **Embedding-Functions**: $F_{\mathrm{Al}}(\rho)$ and $F_{\mathrm{Fe}}(\rho)$.
- **Density-Functions**: $\rho_{\mathrm{AlAl}}(r_{ij})$, $\rho_{\mathrm{FeFe}}(r_{ij})$, $\rho_{\mathrm{AlFe}}(r_{ij})$ and $\rho_{\mathrm{FeAl}}(r_{ij})$.

From this it can be seen that, when using `eam/fs` style potentials, the density functions must have both the $\alpha\beta$ **and** $\beta\alpha$ interactions specified.

Although both the $\alpha\beta$ and $\beta\alpha$ can be described using `eam/fs` files, the Mendelev model used in this example uses the same density function for both Al-Fe and Fe-Al cross density functions[3].

Using `SetFL_FS_EAMTabulation` to Tabulate the Model

As in previous examples it is necessary to define pair, density and embedding functions in python code that are then wrapped in *`EAMPotential`* and *`Potential`* objects to be passed to the tabulation function. For brevity only the names of the functions, as defined in the attached example file (`eam_example3a.py`) are now given:

- **Pair-Potentials:**
  - `def ppfuncAlAl(r):` - Al-Al pair-potential $\phi_{\mathrm{AlAl}}(r_{ij})$.
  - `def ppfuncAlFe(r):` - Al-Fe pair-potential $\phi_{\mathrm{AlFe}}(r_{ij})$.
  - `def ppfuncFeFe(r):` - Fe-Fe pair-potential $\phi_{\mathrm{FeFe}}(r_{ij})$.
- **Embedding-Functions:**
  - `def AlEmbedFunction(rho):` - Al embedding function $F_{\mathrm{Al}}(\rho)$.
  - `def FeEmbedFunction(rho):` - Fe embedding function $F_{\mathrm{Fe}}(\rho)$.

[3] M.I. Mendelev, D.J. Srolovitz, G.J. Ackland, and S. Han, "Effect of Fe Segregation on the Migration of a Non-Symmetric $\Sigma5$ Tilt Grain Boundary in Al", *J. Mater. Res.* **20** (2011) 208.

- **Density-Functions:**
  - `def AlAlDensityFunction(r):` - Al density function $\rho_{\text{AlAl}}(r_{ij})$.
  - `def FeFeDensityFunction(r):` - Fe density function $\rho_{\text{AlAl}}(r_{ij})$.
  - `def FeAlDensityFunction(r):` - Al-Fe density function $\rho_{\text{AlFe}}(r_{ij})$.

Note: The functional forms used within the Mendelev paper[3] are somewhat long, and including their implementations here would detract from the readability of this example. However, they are included in the attached python file: `eam_example3a.py`.

These functions are used within the `main()` function of the `eam_example3a.py` file which is now shown:

```python
def main():
    # Define list of pair potentials
    pairPotentials = [
        Potential('Al', 'Al', ppfuncAlAl),
        Potential('Al', 'Fe', ppfuncAlFe),
        Potential('Fe', 'Fe', ppfuncFeFe)]

    # Assemble the EAMPotential objects
    eamPotentials = [
        # Al
        EAMPotential('Al', 13, 26.98154, AlEmbedFunction,
                     {'Al': AlAlDensityFunction,
                      'Fe': FeAlDensityFunction},
                     latticeConstant=4.04527,
                     latticeType='fcc'),
        # Fe
        EAMPotential('Fe', 26, 55.845, FeEmbedFunction,
                     {'Al': FeAlDensityFunction,
                      'Fe': FeFeDensityFunction},
                     latticeConstant=2.855312,
                     latticeType='bcc')]

    # Number of grid points and cut-offs for tabulation.
    cutoff_rho = 300.0
    nrho = 10000

    cutoff = 6.5
    nr = 10000

    tabulation = SetFL_FS_EAMTabulation(
        pairPotentials, eamPotentials, cutoff, nr, cutoff_rho, nrho)

    with open("Mendelev_Al_Fe.eam.fs", "w") as outfile:
        tabulation.write(outfile)
```

1. Breaking `main()` into its components, first a list of *`Potential`* objects is created, this is common with the other tabulation methods already discussed:

```python
    # Define list of pair potentials
    pairPotentials = [
        Potential('Al', 'Al', ppfuncAlAl),
        Potential('Al', 'Fe', ppfuncAlFe),
        Potential('Fe', 'Fe', ppfuncFeFe)]
```

2. Next, the *EAMPotential* objects for Al and Fe are instantiated. This is where a Finnis-Sinclair model differs from the standard EAM seen earlier. Instead of a single density callable being passed to the *EAMPotential* constructor a dictionary of density functions is passed instead (see highlighted lines):

```
    eamPotentials = [
        # Al
        EAMPotential('Al', 13, 26.98154, AlEmbedFunction,
                        {'Al': AlAlDensityFunction,
                         'Fe': FeAlDensityFunction},
                        latticeConstant=4.04527,
                        latticeType='fcc'),
        # Fe
        EAMPotential('Fe', 26, 55.845, FeEmbedFunction,
                        {'Al': FeAlDensityFunction,
                         'Fe': FeFeDensityFunction},
                        latticeConstant=2.855312,
                        latticeType='bcc')]
```

3. The density function dictionary keys refer to the β species in each $\alpha\beta$ pair. This means that:

- for the Al *EAMPotential* instance:
  - ρ_{AlAl} = `AlAlDensityFunction()`,
  - ρ_{AlFe} = `FeAlDensityFunction()`.
- for the Fe *EAMPotential* instance:
  - ρ_{FeAl} = `FeAlDensityFunction()`,
  - ρ_{FeFe} = `FeFeDensityFunction()`.

4. Finally, having defined the list of *EAMPotential* instances these are passed to the constructor of *SetFL_FS_EAMTabulation*, in this case writing the data to `Mendelev_Al_Fe.eam.fs` in the current directory:

```
    tabulation = SetFL_FS_EAMTabulation(
        pairPotentials, eamPotentials, cutoff, nr, cutoff_rho, nrho)

    with open("Mendelev_Al_Fe.eam.fs", "w") as outfile:
        tabulation.write(outfile)
```

Using the `Mendelev_Al_Fe.eam.fs` file within LAMMPS

For a binary system where Al and Fe have IDs of 1 and 2 the `Mendelev_Al_Fe.eam.fs` file is specified to LAMMPS as follows:

```
pair_style eam/fs
pair_coeff * * Mendelev_Al_Fe.eam.fs Al Fe
```

Example 3b: Tabulate Al-Fe Finnis-Sinclair Potentials Using `TABEAM_FinnisSinclair_EAMTabulation` for DL_POLY

Using exactly the same model definition as for *Example 3a*, the Al-Fe model can be re-tabulated for DL_POLY with minimal modification to the `main()` function. The modified version of the tabulation script can be found in `eam_example3b.py`.

The `main()` function is given below:

```python
def main():
    # Define list of pair potentials
    pairPotentials = [
        Potential('Al', 'Al', ppfuncAlAl),
        Potential('Al', 'Fe', ppfuncAlFe),
        Potential('Fe', 'Fe', ppfuncFeFe)]

    # Assemble the EAMPotential objects
    eamPotentials = [
        # Al
        EAMPotential('Al', 13, 26.98154, AlEmbedFunction,
                     {'Al': AlAlDensityFunction,
                      'Fe': FeAlDensityFunction},
                     latticeConstant=4.04527,
                     latticeType='fcc'),
        # Fe
        EAMPotential('Fe', 26, 55.845, FeEmbedFunction,
                     {'Al': FeAlDensityFunction,
                      'Fe': FeFeDensityFunction},
                     latticeConstant=2.855312,
                     latticeType='bcc')]

    # Number of grid points and cut-offs for tabulation.
    cutoff_rho = 300.0
    nrho = 10000

    cutoff = 6.5
    nr = 10000

    tabulation = TABEAM_FinnisSinclair_EAMTabulation(
        pairPotentials, eamPotentials, cutoff, nr, cutoff_rho, nrho)

    with open("TABEAM", "w") as outfile:
        tabulation.write(outfile)
```

Excluding the import statement at the top of the file, only two lines have been changed (highlighted). The first changes the filename to `TABEAM` whilst the second tells python to create an object of *`TABEAM_FinnisSinclair_EAMTabulation`* instead of *`atsim.potentials.eam_tabulation.SetFL_FS_EAMTabulation`*:

```python
    tabulation = TABEAM_FinnisSinclair_EAMTabulation(
        pairPotentials, eamPotentials, cutoff, nr, cutoff_rho, nrho)

    with open("TABEAM", "w") as outfile:
        tabulation.write(outfile)
```

That's it, nothing else has changed.

Using the `TABEAM` file with DL_POLY

Running `eam_example3b.py` produces a file names `TABEAM` within the working directory. This should be placed in the same directory as the other DL_POLY input files (`CONTROL`, `CONFIG` and `FIELD`). Then the following should be added to the end of the `FIELD` file:

```
metal 3
Al Al eeam
Fe Fe eeam
Al Fe eeam
```

Note: The Extended EAM (eeam) variant of the `TABEAM` file generated here is only supported in DL_POLY versions >= 4.05.

Working with `potable` files in Python

Using `Configuration` class to create `Pair_Tabulation` and `EAM_Tabulation` objects from `potable` input

`atsim.potentials.config.Configuration` is a factory class which accepts `potable` input and uses it to create tabulation objects.

Tabulation objects are typically created by passing a file like object containing `potable` input to the *`read()`* method of *`Configuration`*.

Example

The following example demonstrates how to create a `atsim.potentials.pair_tabulation.Pair_Tabulation` object from `potable` input by using the *`atsim.potentials.config.Configuration`* class.

The aim of the example is to find the spline coefficients for following potable input (described elsewhere *Splining*)

```
[Tabulation]
target : GULP
cutoff : 10.0
dr : 0.01

[Pair]
Si-O : spline(
                    as.zbl 14 8
              >=0.8
                    exp_spline
              >=1.4
                    as.buck 18003.7572 0.205204 133.5381 )
```

Running the following script `python_potable_api.py` will print out the spline coefficients:

```
import io

from atsim.potentials.config import Configuration

potable_input = """
[Tabulation]
target : GULP
cutoff : 10.0
dr : 0.01

```

(continues on next page)

(continued from previous page)

```
[Pair]
Si-O : spline(
                    as.zbl 14 8
              >=0.8
                    exp_spline
              >=1.4
                    as.buck 18003.7572 0.205204 133.5381 )
"""


def main():
    # Make a file like object from the potable input string given above.
    potable_input_file = io.StringIO(potable_input)

    # Create a Configuration() object and read input from the input file.
    configuration = Configuration()
    # ... Configuration is a factory class for PairTabulation and EAMTabulation
    #     objects. In the current case it will return a GULP_PairTabulation object.
    tabulation = configuration.read(potable_input_file)

    # The potable input defines a single pair potential.
    # Potential objects are accessible from the tabulation object through
    # its .potentials property.
    potential_Si_O = tabulation.potentials[0]

    # The potential-form for this interaction is now accessed.
    multirange_potentialform = potential_Si_O.potentialFunction

    # The potential-forms created from potable input are Multi_Range_Potential_Form
    # objects. This is true even if only one range is defined, as is the case here.
    #
    # Let's get hold of the spline potential form through the Multi_Range_Potential_
↪Form
    # .range_defns property (which returns a list of MultiRangeDefinitionTuple)
    #
    spline_potentialform = multirange_potentialform.range_defns[0].potential_form

    # Now let's get hold of the spline coefficients
    spline_coefficients = spline_potentialform.splineCoefficients

    print("Spline coefficients are: {}".format(spline_coefficients))


if __name__ == "__main__":
    main()
```

Overriding and adding items

Items can be amended or added to the `potable` input before it is passed to the *Configuration* class. This is done by passing a *atsim.potentials.config.ConfigParser* to the *read_from_parser()* method of the *Configuration* object.

The constructor of *ConfigParser* accepts `overrides` and `additional` parameters, each of which accept lists of *atsim.potentials.config.ConfigParserOverrideTuple*.

ConfigParserOverrideTuple is a `collections.namedtuple` with three properties `section`, `key` and `value`. The first two uniquely identify a location in the `potable` input whilst `value` specifies what should be added or changed.

So to add an additional pair-potential to potable input contained in a file given by `fp` the *`ConfigParser`* would be defined as:

```
cp = ConfigParser(fp,
                  additional=[
                      ConfigParserOverrideTuple(
                          "Pair", "O-O", "as.buck 444.7686 0.402 0.0")
                  ])
```

This would be the same as if the following had been given in the original `potable` input:

```
[Pair]
O-O : as.buck 444.7686 0.402 0.0
```

Similarly to change the tabulation target of a potable file you could use:

```
cp = ConfigParser(fp,
                  overrides=[
                      ConfigParserOverrideTuple(
                          "Tabulation", "target", "LAMMPS"),
                  ])
```

A tabulation object is then obtained by combining the *`ConfigParser`* with *`Configuration`*:

```
tabulation = Configuration().read_from_parser(cp)
```

Example

This example shows the use of overrides and additional items. Again the potable input from *Splining* is used.

```
[Tabulation]
target : GULP
cutoff : 10.0
dr : 0.01

[Pair]
Si-O : spline(
                    as.zbl 14 8
              >=0.8
                    exp_spline
              >=1.4
                    as.buck 18003.7572 0.205204 133.5381 )
```

In `python_potable_api.py` the tabulation target and potential cutoff in the `[Tabulation]` section are overriden. An additional `O-O` interaction is added to the `[Pair]` section.

This is then used to create a tabulation object which is finally output to the screen:

```
import io
import sys

from atsim.potentials.config import (ConfigParser, ConfigParserOverrideTuple,
                                     Configuration)
```

(continues on next page)

(continued from previous page)

```
potable_input = """
[Tabulation]
target : GULP
cutoff : 10.0
dr : 0.01

[Pair]
Si-O : spline(
                    as.zbl 14 8
              >=0.8
                    exp_spline
              >=1.4
                    as.buck 18003.7572 0.205204 133.5381 )
"""


def main():
    # Make a file like object from the potable input string given above.
    potable_input_file = io.StringIO(potable_input)

    # Create a Configuration() object and read input from the input file.
    configuration = Configuration()

    # This example shows how to override and add items to potable input before it
    # is passed to the Configuration object.
    #    The tabulation target will be change to 'LAMMPS'
    #    The cutoff will be reduced to 6.5
    #    An additional pair-interaction will be given for O-O

    cp = ConfigParser(potable_input_file,
                      overrides=[
                          ConfigParserOverrideTuple(
                              "Tabulation", "target", "LAMMPS"),
                          ConfigParserOverrideTuple(
                              "Tabulation", "cutoff", "6.5")
                      ],
                      additional=[
                          ConfigParserOverrideTuple(
                              "Pair", "O-O", "as.buck 444.7686 0.402 0.0")
                      ])

    # Create the tabulation by passing the Config_Parser object to the Configuration.
↪read_from_parser method.
    tabulation = configuration.read_from_parser(cp)

    # Now write tabulation to console
    tabulation.write(sys.stderr)


if __name__ == "__main__":
    main()
```

2.4 Reference

2.4.1 List of Potential Forms

Key to features:

The **Features** field in the following potential description may contain the following values:

- **potential-form** - can be used in the `[Pair]`, `[EAM-Embed]` and `[EAM-Density]` sections of a *potable* input file.
- **potential-function** - can also be used as a function in sections of input files that accept mathematical expressions.
- **deriv** - potential form provides an analytical derivative with respect to separation.
- **deriv2** - provides an analytical second derivative with respect to separation.

Born-Mayer (bornmayer)

$$V(r_{ij}) = A \exp\left(\frac{-r_{ij}}{\rho}\right)$$

potable signature *as.bornmayer* A ρ

Features potential-form, potential-function, deriv, deriv2

See also:

- *Buckingham (buck)*

Buckingham (buck)

Potential form due to R.A. Buckingham [Buckingham1938]

$$V(r_{ij}) = A\exp\left(-\frac{r_{ij}}{\rho}\right) - \frac{C}{r_{ij}^6}$$

potable signature `as.buck` A ρ C

Features potential-form, potential-function, deriv, deriv2.

See also:

- *Born-Mayer (bornmayer)*
- Wikipedia - Buckingham potential

Buckingham-4 (buck4)

Four-range Buckingham potential due to B. Vessal et al. [Vessal1989], [Vessal1993].

$$V(r_{ij}) = \begin{cases} A\exp(-r_{ij}/\rho), & 0 \le r_{ij} \le r_{\text{detach}} \\ a_0 + a_1 r_{ij} + a_2 r_{ij}^2 + a_3 r_{ij}^3 + a_4 r_{ij}^4 + a_5 r_{ij}^5, & r_{\text{detach}} < r_{ij} < r_{\text{min}} \\ b_0 + b_1 r_{ij} + b_2 r_{ij}^2 + b_3 r_{ij}^3, & r_{\text{min}} \le r_{ij} < r_{\text{attach}} \\ -\frac{C}{r_{ij}^6}, & r_{ij} \ge r_{\text{attach}} \end{cases}$$

In other words this is a Buckingham potential in which the *Born-Mayer* component acts at small separations and the disprsion term acts at larger separation. These two parts are linked by a fifth then third order polynomial (with a minimum formed in the spline at r_{min}).

The spline parameters ($a_{0...5}$ and $b_{0...3}$) are subject to the constraints that $V(r_{ij})$, first and second derivatives must be equal at the boundary points and the function must have a stationary point at r_{min}. The spline coefficients are automatically calculated by this potential-form.

Note: Due to the complexity of calculating the spline-coefficients this potential form does not have an equivalent in the atsim.potentials.potentialfunctions module.

potable signature `as.buck4` A ρ C r_{detach} r_{min} r_{attach}

Features potential-form, deriv, deriv2

See also:

- *Buckingham-4 Spline buck4_spline*
- *Splining*

Constant (constant)

Potential form that always evaluates to a constant value.

$$V(r_{ij}) = C$$

potable signature `as.constant` C

Features potential-form, potential-function, deriv, deriv2

Coulomb (coul)

Electrostatic interaction between two point charges.

$$V(r_{ij}) = \frac{q_i q_j}{4\pi\epsilon_0 r_{ij}}$$

Note: Constant value appropriate for r_{ij} in angstroms and energy in eV.

potable signature `as.coul` q_i q_j

Features potential-form, potential-function, deriv, deriv2

Exponential (exponential)

General exponential form.

$$V(r_{ij}) = A r_{ij}^n$$

potable signature `as.exponential` A n

Features potential-form, potential-function, deriv, deriv2

Exponential Spline (exp_spline)

Exponential spline function (as used in splining routines).

$$V(r_{ij}) = \exp\left(B_0 + B_1 r_{ij} + B_2 r_{ij}^2 + B_3 r_{ij}^3 + B_4 r_{ij}^4 + B_5 r_{ij}^5\right) + C$$

Where B_0, B_1, B_2, B_3, B_4, B_5, C are spline coefficients.

potable signature `as.exp_spline` B_0 B_1 B_2 B_3 B_4 B_5 C

Features potential-form, potential-function, deriv, deriv2

See also:

- *Exponential Spline exp_spline*
- *Splining*

Hydrogen Bond 12-10 (hbnd)

$$V(r_{ij}) = \frac{A}{r_{ij}^{12}} - \frac{B}{r_{ij}^{10}}$$

portable signature `as.hbnd` A B

Features potential-form, potential-function, deriv, deriv2

Lennard-Jones 12-6 (lj)

Potential form first proposed by John Lennard-Jones in 1924 [Lennard-Jones1924].

$$V(r_{ij}) = 4\epsilon \left(\frac{\sigma^{12}}{r_{ij}^{12}} - \frac{\sigma^6}{r_{ij}^6} \right)$$

ϵ defines depth of potential well and σ is the separation at which $V(r_{ij})$ is zero.

potable signature `as.lj` ϵ σ

Features potential-form, potential-function, deriv, deriv2

See also:

- Wikipedia - Lennard-Jones potential

Morse (morse)

$$V(r_{ij}) = D\left[\exp\left(-2\gamma(r_{ij} - r_*)\right) - 2\exp\left(-\gamma(r - r_*)\right)\right]$$

$-D$ is the potential well depth at an equilibrium separation of r_*.

potable signature `as.morse` γ r_* D

Features potential-form, potential-function, deriv, deriv2

See also:

- Wikipedia - Morse potential

Polynomial (polynomial)

Polynomial of arbitrary order.

$$V(r_{ij}) = C_0 + C_1 r_{ij} + C_2 r_{ij}^2 + \cdots + C_n r_{ij}^n$$

This function accepts a variable number of arguments which are $C_0, C_1, \ldots, C_n$ respectively.

potable signatures `as.polynomial` $C_0 ... C_n$

Features potential-form, potential-function, deriv, deriv2

See also:

- *Example: parametrising a model using published spline coefficients*

Square Root (sqrt)

Potential function is:

$$U(r_{ij}) = G\sqrt{r_{ij}}$$

potable signature `as.sqrt` G

Features potential-form, potential-function, deriv, deriv2

Tang-Toennies (tang_toennies)

This potential form was derived to describe the Van der Waal's interactions between the noble gases (He to Rn) by Tang and Toennies [Tang2003].

This has the following form:

$$V(r) = A\exp(-br) - \sum_{n=3}^{N} f_{2N}(bR)\frac{C_{2N}}{R^{2N}}$$

Where:

$$f_{2N}(x) = 1 - \exp(-x)\sum_{k=0}^{2n}\frac{x^k}{k!}$$

potable signature `as.tang_toennies` A b C_6 C_8 C_{10}

Features potential-form, potential-function, deriv, deriv2

Zero (zero)

Potential form which returns zero for all separations.

$$V(r) = 0$$

potable signature `as.zero`

Features potential-form, potential-function, deriv, deriv2

Ziegler-Biersack-Littmark (zbl)

Ziegler-Biersack-Littmark screened nuclear repulsion for describing high energy interactions [Ziegler2015].

$$V(r) =$$

$$\frac{1}{4\pi\epsilon_0}\frac{Z_1}{Z_2}\phi(r/a) + S(r)$$

$$a =$$

$$\frac{0.46850}{Z_i^{0.23} + Z_j^{0.23}}$$

$$\phi(x) =$$

$$0.18175\exp(-3.19980x)$$

$$+0.50986\exp(-0.94229x)$$

$$+0.28022\exp(-0.40290x)$$

$$+0.02817\exp(-0.20162x)$$

Where Z_i and Z_j are the atomic numbers of two species.

potable signature `as.zbl` Z_i Z_j

Features potential-form, potential-function, deriv, deriv2

2.4.2 `potable` input format

[EAM-ADP-Dipole]

Added in: 0.4.0

Section defining the dipole functions for angular dependent (ADP) EAM models. See *ADP Style EAM Models*.

Potential forms are defined between pairs of species in the same way as in the *[Pair]* section:

```
SPECIES_A-SPECIES_B : POTENTIAL_FORM PARM_1 PARAM_2 ... PARAM_N
```

Where:

- `SPECIES_A-SPECIESB` gives the pair of species for which the dipole function is defined. e.g. *Al-Cu* would define a function for aluminium and copper.
- `POTENTIAL_FORM PARAM_1 PARAM_2 ... PARAM_N` defines the potential form in the same way as in the *[Pair]* section.

[EAM-ADP-Quadrupole]

Added in: 0.4.0

Section defining the quadrupole functions for angular dependent (ADP) EAM models. See *ADP Style EAM Models*.

Potential forms are defined between pairs of species in the same way as in the *[Pair]* section:

```
SPECIES_A-SPECIES_B : POTENTIAL_FORM PARM_1 PARAM_2 ... PARAM_N
```

Where:

- `SPECIES_A-SPECIESB` gives the pair of species for which the dipole function is defined. e.g. *Al-Cu* would define a function for aluminium and copper.
- `POTENTIAL_FORM PARAM_1 PARAM_2 ... PARAM_N` defines the potential form in the same way as in the *[Pair]* section.

[EAM-Density]

The density functions for embedded atom models are specified in this section. The input takes different forms depending on whether the standard embedded atom model or Finnis-Sinclair variant are being used.

Both standards have the following general form:

```
INTERACTION : POTENTIAL_FORM PARAM_1 PARAM_2 ... PARAM_N
```

Where:

- `POTENTIAL_FORM PARAM_1 ...`: density functions use the same rules to instantiate potential forms as in the *[Pair]* section.
- `INTERACTION` specifies the density this potential-form represents:
  - **Standard EAM:** standard EAM uses the same function for the density surrounding any central atom of any given species. Consequently in these cases `INTERACTION` is a single species label. So the density function of aluminium would take the form:

```
[EAM-Density]
Al : POTENIAL_FORM ...
```

- **Finnis-Sinclair:** in this variant of EAM density functions change depending on types of the cental atom and surrounding atom to be embedded. Consequently the following form is used:

```
[EAM-Density]
A->B : POTENTIAL_FORM ...
```

- Where `A` is the central atom type and `B` is the type of the embedding atom. To define a density function for the density of nickel being embedded at an aluminium site this would be used:

```
[EAM-Density]
Al->Ni : POTENTIAL_FORM ...
```

- It should be noted that `A->B` and `B->A` must be specified separately even if the same density function is used for both. If not given null (i.e. `as.zero`) density functions are implicitly defined for missing interactions.

See also:

- *Many body models*

[EAM-Embed]

Embedding functions for many-body models are defined in this section.

Entries have the following form:

```
SPECIES : POTENTIAL_FORM PARAM_1 PARAM_2 ... PARAM_N
```

Where:

- `SPECIES` is atomic type at which the surrounding electron density will be embedded using the specified potential form.
- `POTENTIAL_FORM PARAM_1 ...`: embedding functions instantiate potential forms in the same way as in the *[Pair]* section.

Note: Embedding functions are tabulated using rho values. The resolution and extent of functions in rho are defined by `drho`, `nrho` and `cutoff_rho` in the *[Tabulation] section*.

See also:

- *Many body models*

[Pair]

Pair-potentials are defined in this section of the file. See *[Pair] section* for full description.

See also:

See also:

- **Potential-forms are parametrised in this section:**
  - *List of Potential Forms* - reference list of pre-defined potential forms.

    - Custom functions are defined in the `[Potential-Form]` section:
        * *[Potential-Form] section* - custom potential-forms are introduced here.
        * *[Potential-Form]* - reference information for `[Potential-Form]` section.

- **Potential-modifiers are described in thiese sections:**
    - *Potential modifiers* - are introduced here.
    - *List of Potential Modifiers* - list of potential-modifiers.

[Potential-Form]

Custom functional forms are defined in this section. See *[Potential-Form] section* where it is introduced.

See also:

- The syntax used by the mathematical expressions defined in the `[Potential-Form]` is defined here.

Python maths functions supported in mathematical expressions

The mathematical expressions used in the `[Potential-Form]` section of `potable` input allow a subset of functions from the math module to be used. These are accesible via the `pymath.*` namespace prefix. An example of this is provided here: *Formula syntax*

The list of functions accessible through `pymath.*` are below. In general, functions that return multiple values do not appear:

- acos(x)
- acosh(x)
- asinh(x)
- atan(x)
- atan2(x,y)
- atanh(x)
- cos(x)
- cosh(x)
- degrees(x)
- exp(x)
- factorial(x)
- fsum(*args)
    - This function is called slightly differently than in native Python.
    - In Python you pass in a single iterable to this function. This expression: `math.fsum([1,2,3,4])` would be written `pymath.fsum(1,2,3,4)` in a `potable` formula.
- gcd(a,b)
- hypot(x,y)
- ldexp(a,b)
- log(*args)

- log10(x)
- log1p(x)
- log2(x)
- pow(x,a)
- radians(x)
- sin(x)
- sinh(x)
- sqrt(x)
- sqrt(x)
- tan(x)
- tanh(x)
- trunc(x)

[Tabulation]

The section of the input file which defines how a model should be tabulated.

Fields

cutoff

Item `cutoff`

Format float

Description Defines upper bound of functions tabulated in terms of separation. This is used in a pair with ns tabulated in terms of separation. This directive is used together with *nr (number of rows)* or *dr (step size)* to give the extent and resolution of a tabulated function.

cutoff_rho

Item `cutoff_rho`

Format float

Description Used to define cutoff for functions tabulated in terms of electron density (rho) e.g. for *[EAM-Embed]* functions. This option defines the upper bound of rho values included in the tabulation of these functions. This directive is used together with *nrho* or *cutoff_rho* to define resolution and extent of density functions.

dr

Item `dr`

Format float

Description Defines the step size between rows of functions tabulated in terms of separation. This directive is used together with *nr* or *cutoff* to define resolution and extent of these functions.

drho

Item `drho`

Format float

Description Used to define resolution of functions tabulated in terms of electron density (rho) e.g. for *[EAM-Embed]* functions. This option defines the rho increment for such functions. This directive is used together with *nrho* or *cutoff_rho* to define resolution and extent of these functions.

nr

Item `nr`

Format int

Description Defines the number of rows when functions are tabulated in terms of separation. This directive is used either with *dr* or *cutoff* to give the range and resolution of the tabulated function.

nrho

Item `nrho`

Format int

Description Used to define cutoff (in conjunction with `drho`) for functions tabulated in terms of electron density (rho) e.g. for *[EAM-Embed]* functions. This option defines the number of rho values included in the tabulation of these functions. This directive is used together with *nrho* or *cutoff_rho* to define resolution and extent of density functions.

target

Item `target`

Format str

Valid Options `DL_POLY|DLPOLY`, `DL_POLY_EAM_fs`, `DL_POLY_EAM`, `eam_adp`, `excel`, `excel_eam`, `excel_eam_fs`, `GULP`, `LAMMPS_eam_alloy|setfl`, `LAMMPS`, `setfl_fs`

Description Specifies the format that tabulation will be written in.

[Table-Form]

The `[Table-Form]` section is used to define functions from pre-tabulated data that may be used in the same way as a custom `[Potentia-Form]`. Data is specified using the `x` and `y` options or the `xy` option.

To provide a continuous function interpolation is performed between data points, the interpolation method is set using the `interpolation` option.

Naming Table Form

To allow a [Table-Form] to be used in sections such as [Pair], [EAM-Embed] and [EAM-Density] it is necessary to give it a unique label. This is done by including it in the section header following a colon:

```
[Table-Form:NAME]
```

Therefore to create a [Table-Form] named tabulated the following definition could be used:

```
[Table-Form:tabulated]
interpolation: cubic_spline
x : 0.0 1.0 2.0 3.0
y : 0.0 2.0 3.0 4.0
```

This could then be referenced in another section using this name. e.g.

```
[Pair]
Si-O : tabulated
```

Fields

interpolation

Item interpolation

Format Currently this option only accepts cubic_spline

Description Sets interpolation type.

x

Item x

Format List of space separated float values.

Description Define x values of tabulated data. Must be used with y option.

Example To define a linear function the following could be used:

```
[Table-Form:linear]
interpolation: cubic_spline
x : 0.0 1.0 2.0 3.0
y : 0.0 2.0 3.0 4.0
```

xy

Item xy

Format List of space separated float values.

Description Allows x and y values of data to be specified as series of pairs.

Example To define a linear function the following could be used:

```
[Table-Form:linear]
interpolation: cubic_spline
xy: 0.0 0.0
    1.0 2.0
    2.0 3.0
    3.0 4.0
```

y

Item `y`

Format List of space separated float values.

Description Define y values of tabulated data. Must be used with `x` option.

Example See documentation for *x* option.

[Variables]

Added in: 0.4.0

This section allows values to be specified for use in multiple places in the `potable` file. Values are actually string snippets with variable place-holders replaced throughout the file before potential tabulation is performed. Variables are specified like this:

```
[Variables]
VARIABLE_NAME_1 : VARIABLE_VALUE_1
VARIABLE_NAME_2 : VARIABLE_VALUE_2
...
VARIABLE_NAME_N : VARIABLE_VALUE_N
```

These values may then be referenced elsewhere in the file through place-holders with the this form `${VARIABLE_NAME}`. With the place-holder replaced with the value from the `[Variables]` section before tabulation is performed.

This feature makes use of the string interpolation from Python's `configparser` module using the extended interpolation syntax. This allows values from other sections in the file to be referenced using this placeholder format: `${SECTION:NAME}`.

Example

```
[Variables]
nsteps : 10000
rho : 0.32

[Tabulation]
target : LAMMPS
nr : ${nsteps}
dr : 0.1

[Species]
Gd.atomic_number : 64
O.atomic_number : 8

```

(continues on next page)

(continued from previous page)

```
[Pair]
Gd-O : spline(
                as.zbl ${Species:Gd.atomic_number} ${Species:O.atomic_number}
                >=0.8
                    as.buck 1000.0 ${rho} 0.0)
O-O : as.buck 500 ${rho} 32.0
```

2.4.3 List of Potential Modifiers

Potential modifiers are described here: *Potential modifiers*.

A list of available potential modifiers is provided here.

pow()

Modifier that raises each potential-form to the power of the next.

If the potentials provide analytical derivatives, the `pow()` modifier will combine these correctly.

Example:

To take the square of the *sum()* of a series of potential forms you could use:

```
[Pair]
# This would evaluate to 0.16
A-B : pow(sum(as.constant -1, as.constant 0.1, as.constant 0.5),
          as.constant 2)
```

`pow()` can take more than two potential forms as its arguments:

```
[Pair]
# This would evaluate to 2^(2^3) = 256
A-B : pow(as.constant 2, as.constant 3, as.constant 2)
```

You aren't restricted to using constant values as arguments:

```
[Pair]
# This is equivalent to 2^(0.5r + r^2)
A-B : pow(as.constant 2, as.polynomial 0 0.5 1)
```

product()

Modifier that takes the product of the potential-forms provided to it as arguments.

If the potentials provide analytical derivatives the `product()` modifier will combine these correctly.

Example:

Any number of potential instances can be multiplied by each other:

```
[Pair]
# Evaluates to 16
A-A : product(as.constant 2.0, as.constant 2.0, as.constant 4.0)

# Apply a soft-cutoff at 2.5 Angs to a Buckingham potential
# This defines a custom function in the [Potential-Form] section
# based on the complementary error function for this purpose.
B-B : product(as.buck 1000.0 0.2 32.0,
              truncate 2.5)

[Potential-Form]
truncate(rij, cutoff) = erfc(4*(rij-cutoff))/2.0
```

spline()

Modifier that smoothly splines between two potential forms by linking them with an intermediate spline.

`spline()` takes a single argument which is defined as a *multi-range potential*. This must define three ranges:

1. Start potential
2. Interpolating spline
3. End potential

The **Interpolating spline** section has the form:

```
SPLINE_LABEL SPLINE_PARAMETERS
```

Where the `SPLINE_LABEL` defines the type of spline to be used and the (optional) `SPLINE_PARAMETERS` is a list of space separated options taken by the spline function.

A list of spline types usable with `SPLINE_LABEL` is now given:

buck4_spline

Spline Signature `buck4_spline` r_{min}

Description Combination of a fifth and third order polynomial joined by a stationary point at r_{min}. This is the spline used in the well-known *four-range Buckingham potential form*.

See also

- *Buckingham-4 Spline buck4_spline*
- *Buckingham-4 (buck4)*

exp_spline

Spline Signature `exp_spline`

Description Exponential of fifth order polynomial.

See also

- *Exponential Spline exp_spline*
- *Exponential Spline (exp_spline)*

Example:

A configuration string might be defined as:

```
[Pair]

Si-O : spline(>0 as.zbl 14 8 >=0.8 exp_spline >=1.4 as.buck 180003 0.3 32.0)
```

This would create a *zbl* and *Buckingham* potential connected by an exponential spline when *r* is between 0.8 and 1.4.

See also:

- Splining is introduced in more detail here: *Splining*.
- List of examples:
  - *Example: splining to the zbl potential form using exp_spline*.
  - *Example: redefining the Morelon model using buck4_spline*.

`sum()`

Modifier that sums all the potentials given as arguments.

If the potentials provide analytical derivatives the `sum()` modifier will combine these correctly.

Example:

Any number of potential instances can be summed:

```
[Pair]
# Evaluates to 3
A-A : sum(as.constant 1.0, as.constant 2.0)
# Evaluates to 6
B-B : sum(as.constant 1.0, as.constant 2.0, as.constant 3.0)
```

See also:

- This modifier is used in the following examples:
  - *Quick-Start*
  - *Example: using custom-potential forms to define Basak potential*

`trans()`

Modifier that applies the following transformation to a given potential function:

```
potential(r+X)
```

Where X is the transformation value.

This modifier takes two arguments, the first is a potential form instance. The second must be an instance of `as.constant` that takes `X` as its argument.

Example

To shift a Buckingham paair potential two angstroms to the left the `trans()` modifier could be used like this:

```
[Pair]
A-B : trans(as.buck 1000.0 0.1 32.0, as.constant 2)
```

2.4.4 Command Line Tools

potable

The **`potable`** tool is the interface for working with potential definition files. In addition to converting a potential model definition into a tabulation it allows their contents to be queried, filtered, overridden and plotted.

Usage

```
potable [-h]
            [--list-items | --list-item-labels | --item-value SECTION_NAME:KEY]
            [--include-species [SPECIES [SPECIES ...]] | --exclude-species
            [SPECIES [SPECIES ...]]]
            [--override-item [SECTION_NAME:KEY=VALUE [SECTION_NAME:KEY=VALUE ...]]]
            [--add-item [SECTION_NAME:KEY=VALUE [SECTION_NAME:KEY=VALUE ...]]]
            [--remove-item [SECTION_NAME:KEY [SECTION_NAME:KEY ...]]]
            POTENTIAL_DEFN_FILE [OUTPUT_FILE]
```

Tabulate potential models for common atomistic simulation codes. This is part of the atsim.potentials package.

Positional Arguments:

- `POTENTIAL_DEFN_FILE` File containing definition of potential model.
- `OUTPUT_FILE` File into which data will be tabulated.

Optional Arguments:

`-h, --help`
show this help message and exit

Query

Query items in the configuration file

`--list-items, -l`
List items in configuration file to STD_OUT. One is listed per line with format `SECTION_NAME:KEY=VALUE`

`--list-item-labels`
List item in configuration file to STD_OUT. One item per line with format `SECTION_NAME:KEY`

`--item-value` `SECTION_NAME:KEY`
Return the value for given item in configuration file

Filter

Filter items from the configuration file

`--include-species` `[SPECIES [SPECIES ...]]`
: If specified, only those `SPECIES` provided will be included in tabulation.

`--exclude-species` `[SPECIES [SPECIES ...]]`
: `SPECIES` provided to this option will NOT be included in tabulation.

Override

Add or override values in the configuration file

`--override-item` `[SECTION_NAME:KEY=VALUE [SECTION_NAME:KEY=VALUE ...]]`, **`-e`** `[SECTION_NAME:KEY`
: Use `VALUE` for item `SECTION_NAME:KEY` instead of value contained in the configuration file

`--add-item` `[SECTION_NAME:KEY=VALUE [SECTION_NAME:KEY=VALUE ...]]`, **`-a`** `[SECTION_NAME:KEY=VALU`
: Add item to configuration file

`--remove-item` `[SECTION_NAME:KEY [SECTION_NAME:KEY ...]]`, **`-r`** `[SECTION_NAME:KEY [SECTION_NAMI`
: Remove item from configuration file

Examples:

Various examples of the use of this tool are given throughout the documentation:

- *Quick-Start*.
- *Quick Start: Generating Basak Tabulation for DL_POLY*.
- *Quick Start: Generating Basak Tabulation for LAMMPS*.
- *Quick Start: Generating Basak Tabulation for GULP*. This provides an example of the `--override-item` option.
- *User Guide: Making and Testing the Tabulation - Sutton Ag Example*.
- *Troubleshooting Potable Input Files*: provides an example of the `--override-item` option.

2.4.5 API Reference

This page contains auto-generated API reference documentation[1].

`atsim`

Subpackages

`atsim.potentials`

A collection of classes and functions related to defining potentials

[1] Created with sphinx-autoapi

Subpackages

`atsim.potentials.config`

Package Contents

Classes

`ConfigParser`(fp, overrides=[], additional=[])	Performs initial stage (tokenizing) of generating a potential model
`FilteredConfigParser`(config_parser, exclude=[], include=[])	Class that wraps around ConfigParser instances and
`Potential_Form_Registry`(cfg, register_standard=False, register_pymath_functions=False)	Factory class that takes [Potential-Form] and [Table-Form] definitions
`Modifier_Registry`()	Registry of factories for potential modifiers
`Configuration`()	Factory class that allows Tabulation objects to be built from .ini files

class `atsim.potentials.config.`**`ConfigParser`**(*fp*, *overrides=[]*, *additional=[]*)

Bases: `object`

Performs initial stage (tokenizing) of generating a potential model suitable for tabulation functions.

`pair`

Returns the contents of the config file's [Pair] section.

Returns List of tuples of (SpeciesPair, potential_form_label, params) Where params = [p1, p2, . . . , pn] and p1 etc are the potential parameters

`potential_form`

Return the contents of the config file's [Potential-Form] section.

Returns List of (PotentialFormSignature, formula_string) pairs.

`tabulation`

Return the parsed contents of the config file's [Tabulation] section.

This defines what type of model (pair, EAM) the config file contains and also how the model should be tabulated.

Returns _Tabulation_Section object

`table_form`

Returns parsed content of config file's [Table-Form] section.

This allows pre-tabulated data to be used within atsim.potentials.

Returns List of TableFormTuple instance tuples.

`eam_embed`

Return the parsed contents of the configuration file's [EAM-Embed] section.

Returns List of (SPECIES, potential_form_label, params) Where params = [p1, p2, . . . , pn] and p1 etc are the embedding function parameters)

`eam_density`

Return the parsed contents of the configuration file's [EAM-Density] section.

Returns List of (SPECIES, potential_form_label, params) Where params = [p1, p2, . . . , pn] and p1 etc are the density function parameters)

eam_density_fs

Return the parsed contents of the configuration file's [EAM-Density] section.

This assumes Finnis-Sinclair parsing rules. This means that SPECIES (below) is parsed as a *EAMFSDensitySpeciesTuple* with *from_species* and *to_species* attributes.

Returns List of (SPECIES, potential_form_label, params) Where params = [p1, p2, . . . , pn] and p1 etc are the density function parameters)

parsed_sections

Returns a list of relevant sections found inside configuration file.

Names are returned as the *ConfigParser* attribute names which could be used to access each parsed section. So *[Pair]* becomes *pair* and *[EAM-Density]* is *eam_density*.

Returns List of attribute names representing parseable sections of the configuration file

orphan_sections

Returns list of section keys, in current configuration file, that are not relevant to the *ConfigParser* class.property

Returns List of section labels.

raw_config_parser

species

Return reference data for atomic species.

Data is returned as a dictionary relating each species label to a dictionary mapping property name to propety value.

Returns Dictionary of dictionaries.

parse_pair_like(*self*, *section_name*)

Parse a section as if it contains pair potentials.

Parameters **section_name** – Name of section that should be parsed in the same way as the [Pair] section.

Returns List of tuples of (SpeciesPair, potential_form_label, params) Where params = [p1, p2, . . . , pn] and p1 etc are the potential parameters

`atsim.potentials.config.`**ConfigParserOverrideTuple**

exception `atsim.potentials.config.`**ConfigOverrideException**

Bases: `atsim.potentials.config._common.ConfigParserException`

Common base class for all non-exit exceptions.

class `atsim.potentials.config.`**FilteredConfigParser**(*config_parser*, *exclude=[]*, *include=[]*)

Bases: `wrapt.ObjectProxy`

Class that wraps around ConfigParser instances and filters out entries for particular, unwanted species

pair

eam_embed

eam_density

eam_density_fs

class atsim.potentials.config.**Potential_Form_Registry**(*cfg*, *register_standard=False*, *register_pymath_functions=False*)

Bases: `object`

Factory class that takes [Potential-Form] and [Table-Form] definitions from ConfigParser and turns them into Potential_Form objects

registered

Returns the labels for the potentials registered here.

__getitem__(*self*, *k*)

class atsim.potentials.config.**Modifier_Registry**

Bases: `object`

Registry of factories for potential modifiers

__getitem__(*self*, *k*)

class atsim.potentials.config.**Configuration**

Bases: `object`

Factory class that allows Tabulation objects to be built from .ini files

read(*self*, *fp*)

Read potential data from the file object *fp* and return a *PairTabulation* or *EAMTabulation* object.

Params fp File like object containing potential information.

Returns Tabulation object

read_from_parser(*self*, *cp*)

Read potential data from the *ConfigParser* object *cp* and return a *PairTabulation* or *EAMTabulation* instance.

Parameters **cp** – atsim.potentials.config.ConfigParser instance.

Returns Tabulation object

exception atsim.potentials.config.**ConfigParserException**

Bases: `atsim.potentials.config._common.ConfigurationException`

Common base class for all non-exit exceptions.

exception atsim.potentials.config.**Potential_Form_Registry_Exception**

Bases: `atsim.potentials.config._common.ConfigurationException`

Common base class for all non-exit exceptions.

exception atsim.potentials.config.**Potential_Form_Exception**

Bases: `atsim.potentials.config._common.ConfigurationException`

Common base class for all non-exit exceptions.

`atsim.potentials.referencedata`

Package Contents

Classes

Reference_Data(extra_data={})	Class providing data about atomic species

class atsim.potentials.referencedata.**Reference_Data**(*extra_data={}*)

Bases: `object`

Class providing data about atomic species

get(*self*, *species*, *property_name*)

Get a property value for a given species.

Parameters

- **species** – Species label.
- **property_name** – Propety identifier.

Returns Property value for given combination of species and property name.

exception atsim.potentials.referencedata.**Unknown_Species_Exception**

Bases: `atsim.potentials.referencedata._reference_data.Reference_Data_Exception`

Common base class for all non-exit exceptions.

exception atsim.potentials.referencedata.**Unknown_Property_Exception**

Bases: `atsim.potentials.referencedata._reference_data.Reference_Data_Exception`

Common base class for all non-exit exceptions.

exception atsim.potentials.referencedata.**Reference_Data_Exception**

Bases: `Exception`

Common base class for all non-exit exceptions.

atsim.potentials.spline

Package Contents

Classes

Spline_Point(potential_function, r)	Class for the attachment and detachment points of potential objects and region to be splined
Exp_Spline(detach_point, attach_point)	Class for represention splines of the form:
Buck4_Spline(detach_point, attach_point, r_min)	Class for representing the splined part of the four ranged Buckingham potential.
Custom_SplinePotential(spline)	Callable to allow splining of one potential to another
SplinePotential(startPotential, endPotential, detachmentX, attachmentX)	Callable to allow splining of one potential to another using an exponential spline
Buck4_SplinePotential(startPotential, endPotential, detachmentX, attachmentX, r_min)	Callable to allow splining of one potential to another using the Buck4 spline type

Functions

gradient(func, h=1e-06)	Function wrapper that returns derivative of func.

`atsim.potentials.spline.`**`gradient`**(*func*, *h=1e-06*)

Function wrapper that returns derivative of func.

If the callable, *func* provides a *.deriv(r)* method this will be used to evaluate the derivative of the function, if not the returned function will use num_deriv() in gradient evaluation.

If the callable additionally provides a *.deriv2(r)* method, representing its second derivative, the function returned by this routine will have a *deriv()* method which will delegate to func.deriv2() when called.

By providing .deriv() and .deriv2() on the *func* callable analytical descriptions of a potential's first and second derivatives may be specified.

Parameters

- **`func`** – Function to be wrapped
- **`h`** – Step size used when performing numerical differentiation

Returns Function that returns derivative of func

`atsim.potentials.spline.`**`polynomial`**

`atsim.potentials.spline.`**`exp_spline`**

`class` `atsim.potentials.spline.`**`Spline_Point`**(*potential_function*, *r*)

Bases: `object`

Class for the attachment and detachment points of potential objects and region to be splined

`potential_function`

Potential function

`r`

Value at which splining takes place

`v`

Value of *potential_function* at *r*

`deriv`

First derivative of *potential_function*: dv/dr(r)

`deriv2`

Second derivative of *potential_function*: d2v/dr^2(r)

`deriv_callable`

`deriv2_callable`

`class` `atsim.potentials.spline.`**`Exp_Spline`**(*detach_point*, *attach_point*)

Bases: `object`

Class for represention splines of the form:

$$U(r_{ij}) = \exp\left(B_0 + B_1 r_{ij} + B_2 r_{ij}^2 + B_3 r_{ij}^3 + B_4 r_{ij}^4 + B_5 r_{ij}^5\right) + C$$

The spline coefficients $B_{0...5}$ and *C* can be obtained using the `spline_coefficients()` property.

`detach_point`

Spline_Point giving start of splined region

`attach_point`

Spline_Point giving end of splined region

spline_coefficients
Coefficients for spline_function

__call__(*self*, *r*)

deriv(*self*, *r*)

deriv2(*self*, *r*)

class atsim.potentials.spline.**Buck4_Spline**(*detach_point*, *attach_point*, *r_min*)
Bases: `object`

Class for representing the splined part of the four ranged Buckingham potential.

Between the detachment point and *r_min* this is a 5th order polynomial:

$$U(r_{ij}) = A_0 + A_1 r_{ij} + A_2 r_{ij}^2 + A_3 r_{ij}^3 + A_4 r_{ij}^4 + A_5 r_{ij}^5$$

and between *r_min* and the re-attachment point a 3rd order spline is used:

$$U(r_{ij}) = B0 + B_1 r_{ij} + B_2 r_{ij}^2 + B_3 r_{ij}^3$$

The spline coefficients $A_{0..5}$ and $B_{0..3}$ are solved such that the the spline values match with the potential functions at the detach and re-attachment points and r_min. They are continuous in their first and second derivatives across these points and where the two splines meet at *r_min*. Finally, the derivative at *r_min* is set to be 0 with the aim of creating a minimum.

detach_point
Spline_Point giving start of splined region

attach_point
Spline_Point giving end of splined region

r_min
Position of minimum

spline_coefficients
Spline coefficients as list of form [A_0, A_1, A_2, A_3, A_4, A_5, B_0, B_1, B_2, B_3]

spline5
Callable (atsim.potentials.potentialfunctions.polynomial) object representing the fifth order section of the buck4 spline - between *detach_point* and *r_min*

spline3
Callable (atsim.potentials.potentialfunctions.polynomial) object representing the fifth order section of the buck4 spline - between *detach_point* and *r_min*

__call__(*self*, *r*)

deriv(*self*, *r*)

deriv2(*self*, *r*)

class atsim.potentials.spline.**Custom_SplinePotential**(*spline*)
Bases: `object`

Callable to allow splining of one potential to another

startPotential

Returns Function defining potential for separations < `detachmentX`

endPotential

Returns Function defining potential for separations > `attachmentX`

interpolationFunction

Returns Spline object connecting startPotential and endPotential for separations `detachmentX` < rij < `attachmentX`

detachmentX

Returns Point at which spline should start

attachmentX

Returns Point at which spline should end

splineCoefficients

Returns Tuple containing the seven coefficients of the spline polynomial

__call__(*self*, *rij*)

Parameters **rij** – separation at which to evaluate splined potential

Returns spline value

class atsim.potentials.spline.**SplinePotential**(*startPotential*, *endPotential*, *detachmentX*, *attachmentX*)

Bases: *atsim.potentials.spline.Custom_SplinePotential*

Callable to allow splining of one potential to another using an exponential spline

class atsim.potentials.spline.**Buck4_SplinePotential**(*startPotential*, *endPotential*, *detachmentX*, *attachmentX*, *r_min*)

Bases: *atsim.potentials.spline.Custom_SplinePotential*

Callable to allow splining of one potential to another using the Buck4 spline type

atsim.potentials.tools

Subpackages

atsim.potentials.tools.potable

Front-end script for atsim.potentials. Allows potentials to be tabulated using simple .ini based configuration files

Package Contents

Classes

ConfigParser(fp, overrides=[], additional=[])	Performs initial stage (tokenizing) of generating a potential model
FilteredConfigParser(config_parser, exclude=[], include=[])	Class that wraps around ConfigParser instances and

Functions

main()

`atsim.potentials.tools.potable.`**`ConfigParserOverrideTuple`**

class `atsim.potentials.tools.potable.`**`ConfigParser`**(*fp*, *overrides=[]*, *additional=[]*)
Bases: `object`

Performs initial stage (tokenizing) of generating a potential model suitable for tabulation functions.

`pair`
Returns the contents of the config file's [Pair] section.

Returns List of tuples of (SpeciesPair, potential_form_label, params) Where params = [p1, p2, . . . , pn] and p1 etc are the potential parameters

`potential_form`
Return the contents of the config file's [Potential-Form] section.

Returns List of (PotentialFormSignature, formula_string) pairs.

`tabulation`
Return the parsed contents of the config file's [Tabulation] section.

This defines what type of model (pair, EAM) the config file contains and also how the model should be tabulated.

Returns _Tabulation_Section object

`table_form`
Returns parsed content of config file's [Table-Form] section.

This allows pre-tabulated data to be used within atsim.potentials.

Returns List of TableFormTuple instance tuples.

`eam_embed`
Return the parsed contents of the configuration file's [EAM-Embed] section.

Returns List of (SPECIES, potential_form_label, params) Where params = [p1, p2, . . . , pn] and p1 etc are the embedding function parameters)

`eam_density`
Return the parsed contents of the configuration file's [EAM-Density] section.

Returns List of (SPECIES, potential_form_label, params) Where params = [p1, p2, . . . , pn] and p1 etc are the density function parameters)

`eam_density_fs`
Return the parsed contents of the configuration file's [EAM-Density] section.

This assumes Finnis-Sinclair parsing rules. This means that SPECIES (below) is parsed as a *EAMFSDensitySpeciesTuple* with *from_species* and *to_species* attributes.

Returns List of (SPECIES, potential_form_label, params) Where params = [p1, p2, . . . , pn] and p1 etc are the density function parameters)

`parsed_sections`
Returns a list of relevant sections found inside configuration file.

Names are returned as the *ConfigParser* attribute names which could be used to access each parsed section. So *[Pair]* becomes *pair* and *[EAM-Density]* is *eam_density*.

Returns List of attribute names representing parseable sections of the configuration file

`orphan_sections`
Returns list of section keys, in current configuration file, that are not relevant to the *ConfigParser* class.property

Returns List of section labels.

`raw_config_parser`

`species`

Return reference data for atomic species.

Data is returned as a dictionary relating each species label to a dictionary mapping property name to propety value.

Returns Dictionary of dictionaries.

`parse_pair_like`(*self*, *section_name*)

Parse a section as if it contains pair potentials.

Parameters **`section_name`** – Name of section that should be parsed in the same way as the [Pair] section.

Returns List of tuples of (SpeciesPair, potential_form_label, params) Where params = [p1, p2, . . . , pn] and p1 etc are the potential parameters

class `atsim.potentials.tools.potable.`**`FilteredConfigParser`**(*config_parser*, *exclude=[]*, *include=[]*)

Bases: `wrapt.ObjectProxy`

Class that wraps around ConfigParser instances and filters out entries for particular, unwanted species

`pair`

`eam_embed`

`eam_density`

`eam_density_fs`

exception `atsim.potentials.tools.potable.`**`ConfigurationException`**

Bases: `Exception`

Common base class for all non-exit exceptions.

`atsim.potentials.tools.potable.`**`main`**()

Submodules

`atsim.potentials.eam_tabulation`

Module Contents

Classes

`SetFL_EAMTabulation`(potentials, eam_potentials, cutoff, nr, cutoff_rho, nrho)	Class for tabulating setfl formatted embedded atom potentials suitable
`SetFL_FS_EAMTabulation`(potentials, eam_potentials, cutoff, nr, cutoff_rho, nrho)	Class for tabulating setfl Finnis-Sinclair formatted embedded atom potentials suitable
`TABEAM_EAMTabulation`(potentials, eam_potentials, cutoff, nr, cutoff_rho, nrho)	Class for tabulating TABEAM formatted embedded atom potentials for the DL_POLY code.

Continued on next page

Table 2.13 – continued from previous page

TABEAM_FinnisSinclair_EAMTabulation(potentials, eam_potentials, cutoff, nr, cutoff_rho, nrho)	Class for tabulating EEAM TABEAM formatted Finnis-Sinclair style embedded atom potentials for the DL_POLY code.
Excel_EAMTabulation(potentials, eam_potentials, cutoff, nr, cutoff_rho, nrho)	Class for dumping EAM model into a spreadsheet
Excel_FinnisSinclair_EAMTabulation(potentials, eam_potentials, cutoff, nr, cutoff_rho, nrho)	Class for dumping EAM model into a spreadsheet
ADP_EAMTabulation(potentials, eam_potentials, dipole_potentials, quadrupole_potentials, cutoff, nr, cutoff_rho, nrho)	Class for tabulating setfl formatted embedded atom potentials with the ADP, angular dependent extension,

class atsim.potentials.eam_tabulation.**SetFL_EAMTabulation**(*potentials*, *eam_potentials*, *cutoff*, *nr*, *cutoff_rho*, *nrho*)

Bases: `atsim.potentials.eam_tabulation._EAMTabulationAbstractbase`

Class for tabulating setfl formatted embedded atom potentials suitable for use with LAMMPS' pair_style eam/alloy

write(*self*, *fp*)

Write the tabulation to the file object *fp*.

Parameters **fp** – File object into which data should be written.

class atsim.potentials.eam_tabulation.**SetFL_FS_EAMTabulation**(*potentials*, *eam_potentials*, *cutoff*, *nr*, *cutoff_rho*, *nrho*)

Bases: `atsim.potentials.eam_tabulation._EAMTabulationAbstractbase`

Class for tabulating setfl Finnis-Sinclair formatted embedded atom potentials suitable for use with LAMMPS' pair_style eam/fs

write(*self*, *fp*)

Write the tabulation to the file object *fp*.

Parameters **fp** – File object into which data should be written.

class atsim.potentials.eam_tabulation.**TABEAM_EAMTabulation**(*potentials*, *eam_potentials*, *cutoff*, *nr*, *cutoff_rho*, *nrho*)

Bases: `atsim.potentials.eam_tabulation._EAMTabulationAbstractbase`

Class for tabulating TABEAM formatted embedded atom potentials for the DL_POLY code.

write(*self*, *fp*)

Write the tabulation to the file object *fp*.

Parameters **fp** – File object into which data should be written.

class atsim.potentials.eam_tabulation.**TABEAM_FinnisSinclair_EAMTabulation**(*potentials*, *eam_potentials*, *cutoff*, *nr*, *cutoff_rho*, *nrho*)

Bases: `atsim.potentials.eam_tabulation._EAMTabulationAbstractbase`

Class for tabulating EEAM TABEAM formatted Finnis-Sinclair style embedded atom potentials for the DL_POLY code.

write(*self*, *fp*)
Write the tabulation to the file object *fp*.

Parameters **fp** – File object into which data should be written.

class atsim.potentials.eam_tabulation.**Excel_EAMTabulation**(*potentials*, *eam_potentials*, *cutoff*, *nr*, *cutoff_rho*, *nrho*)
Bases: atsim.potentials.eam_tabulation._EAMTabulationAbstractbase

Class for dumping EAM model into a spreadsheet

workbook

write(*self*, *fp*)
Write the tabulation to the file object *fp*.

Parameters **fp** – File object into which data should be written.

classmethod open_fp(*cls*, *filename*)
Creates a file object with a given path suitable for writing potential data to.

Parameters **filename** – Filename of output file object.

Returns File object suitable for passing to write() method

class atsim.potentials.eam_tabulation.**Excel_FinnisSinclair_EAMTabulation**(*potentials*, *eam_potentials*, *cutoff*, *nr*, *cutoff_rho*, *nrho*)
Bases: *atsim.potentials.eam_tabulation.Excel_EAMTabulation*

Class for dumping EAM model into a spreadsheet

class atsim.potentials.eam_tabulation.**ADP_EAMTabulation**(*potentials*, *eam_potentials*, *dipole_potentials*, *quadrupole_potentials*, *cutoff*, *nr*, *cutoff_rho*, *nrho*)
Bases: *atsim.potentials.eam_tabulation.SetFL_EAMTabulation*

Class for tabulating setfl formatted embedded atom potentials with the ADP, angular dependent extension, suitable for use with LAMMPS' pair_style adp

write(*self*, *fp*)
Write the tabulation to the file object *fp*.

Parameters **fp** – File object into which data should be written.

atsim.potentials.pair_tabulation

Module Contents

Classes

PairTabulation_AbstractBase(potentials, cutoff, nr, target)	Base class for PairTabulation objects.
LAMMPS_PairTabulation(potentials, cutoff, nr)	Class for tabulating pair-potential models for LAMMPS
DLPoly_PairTabulation(potentials, cutoff, nr)	Class for tabulating pair-potential models for DLPOLY
GULP_PairTabulation(potentials, cutoff, nr)	Class for tabulating pair-potential models for the GULP code.
Excel_PairTabulation(potentials, cutoff, nr)	Class for dumping pair-potential models into an Excel formatted spreadsheet

class atsim.potentials.pair_tabulation.**PairTabulation_AbstractBase**(*potentials*, *cutoff*, *nr*, *target*)

Bases: `object`

Base class for PairTabulation objects.

Child classes must implement: write() method

type

target

nr

cutoff

potentials

dr

classmethod open_fp(*self*, *filename*)

Creates a file object with a given path suitable for writing potential data to.

Parameters **filename** – Filename of output file object.

Returns File object suitable for passing to write() method

write(*self*, *fp*)

Write the tabulation to the file object *fp*.

Parameters **fp** – File object into which data should be written.

class atsim.potentials.pair_tabulation.**LAMMPS_PairTabulation**(*potentials*, *cutoff*, *nr*)

Bases: *atsim.potentials.pair_tabulation.PairTabulation_AbstractBase*

Class for tabulating pair-potential models for LAMMPS

write(*self*, *fp*)

Write the tabulation to the file object *fp*.

Parameters **fp** – File object into which data should be written.

class atsim.potentials.pair_tabulation.**DLPoly_PairTabulation**(*potentials*, *cutoff*, *nr*)

Bases: *atsim.potentials.pair_tabulation.PairTabulation_AbstractBase*

Class for tabulating pair-potential models for DLPOLY

write(*self*, *fp*)

Write tabulation to the file object *fp*.

Parameters **fp** – File object into which data should be written.

class atsim.potentials.pair_tabulation.**GULP_PairTabulation**(*potentials*, *cutoff*, *nr*)

Bases: *atsim.potentials.pair_tabulation.PairTabulation_AbstractBase*

Class for tabulating pair-potential models for the GULP code.

write(*self*, *fp*)

Write tabulation to the file object *fp*.

Parameters **fp** – File object into which data should be written.

class atsim.potentials.pair_tabulation.**Excel_PairTabulation**(*potentials*, *cutoff*, *nr*)

Bases: *atsim.potentials.pair_tabulation.PairTabulation_AbstractBase*

Class for dumping pair-potential models into an Excel formatted spreadsheet

workbook

Property which returns an openpyxl.Workbook instance containing potential data

write(*self*, *fp*)

Write tabulation to the file object *fp* (note: fp should be opened in binary mode).

Parameters **fp** – File object into which data should be written.

classmethod open_fp(*self*, *filename*)

Creates a file object with a given path suitable for writing potential data to.

Parameters **filename** – Filename of output file object.

Returns File object suitable for passing to write() method

atsim.potentials.potentialforms

Functions representing different potential forms.

The functions contained herein are function factories returning a function that takes separation as its sole argument.

See *List of Potential Forms* for descriptions of these potential forms.

Module Contents

Functions

potential(func)	Decorator for callables that should be tagged as potential-forms or potential-functions
is_potential(obj)	Identifies if an object is a potential-form or potential-function
buck4(A, rho, C, r_detach, r_min, r_attach)	Returns a potential form describing the four-range Buckingham potential.

atsim.potentials.potentialforms.**potential**(*func*)

Decorator for callables that should be tagged as potential-forms or potential-functions

atsim.potentials.potentialforms.**is_potential**(*obj*)

Identifies if an object is a potential-form or potential-function

atsim.potentials.potentialforms.**buck4**(*A*, *rho*, *C*, *r_detach*, *r_min*, *r_attach*)

Returns a potential form describing the four-range Buckingham potential.

The potential form is:

$$V(r_{ij}) = \begin{cases} A\exp(-r_{ij}/\rho), & 0 \leq r_{ij} \leq r_{\text{detach}} \\ a_0 + a_1 r_{ij} + a_2 r_{ij}^2 + a_3 r_{ij}^3 + a_4 r_{ij}^4 + a_5 r_{ij}^5, & r_{\text{detach}} < r_{ij} < r_{\text{min}} \\ b_0 + b_1 * r_{ij} + b_2 * r_{ij}^2 + b_3 * r_{ij}^3, & r_{\text{min}} \leq r_{ij} < r_{\text{attach}} \\ -\frac{C}{r_{ij}^6}, & r_{ij} \geq r_{\text{attach}} \end{cases}$$

In other words this is a Buckingham potential in which the Born-Mayer component acts at small separations and the disprsion term acts at larger separation. These two parts are linked by a fifth then third order polynomial (with a minimum formed in the spline at r_extmin).

The spline parameters are subject to the constraints that $V(r_{ij})$, first and second derivatives must be equal at the boundary points and the function must have a stationary point at *r_min*.

See also:

- `atsim.potentials.Buck4_Spline`
- `atsim.potentials.Buck4_SplinePotential`

Note: Due to the complexity of calculating the spline-coefficients this potential form does not have an equivalent in the atsim.potentials.potentialfunctions module.

Parameters

- **`A`** – A potential parameter.
- **`rho`** – potential parameter.
- **`C`** – C parameter.
- **`r_detach`** – Separation where spline starts.
- **`r_min`** – Location of stationary point.
- **`r_attach`** – End of splined region.

Returns Splined potential.

`atsim.potentials.potentialforms.`**`buck`**

`atsim.potentials.potentialforms.`**`bornmayer`**

`atsim.potentials.potentialforms.`**`coul`**

`atsim.potentials.potentialforms.`**`constant`**

`atsim.potentials.potentialforms.`**`exponential`**

`atsim.potentials.potentialforms.`**`hbnd`**

`atsim.potentials.potentialforms.`**`lj`**

`atsim.potentials.potentialforms.`**`morse`**

`atsim.potentials.potentialforms.`**`polynomial`**

`atsim.potentials.potentialforms.`**`sqrt`**

`atsim.potentials.potentialforms.`**`tang_toennies`** atsim.potentials.potentialforms.**zbl**

atsim.potentials.potentialforms.**zero**

atsim.potentials.potentialforms.**exp_spline**

atsim.potentials.potentialfunctions

Functions for different potential forms.

Most of the potentials in this module are implemented as callable _Potential_Function_Bases. The potential energy is evaluated by calling one of these objects. By convention the first argument of each is the atomic separation *r*, with other potential parameters following after. For instance, to evaluate a Buckingham potential at *r* = 2.0 the following could be called for *A*, *rho* and *C* values 1000.0, 0.2 and 32.0 respectively:

```
atsim.potentialfunctions.buck(2.0, 1000.0, 0.2, 32.0)
```

The callable objects also have other useful methods. Perhaps most importantly is the *.deriv()* method this returns the first derivative of the given potential (force). Again using the Buckingham potential as an example its derivative can be evaluated for *r* = 2.0 as follows:

```
atsim.potentialfunctions.buck.deriv(2.0, 1000.0, 0.2, 32.0)
```

See *List of Potential Forms* for descriptions of these potential forms.

Module Contents

atsim.potentials.potentialfunctions.**buck**

atsim.potentials.potentialfunctions.**bornmayer**

atsim.potentials.potentialfunctions.**coul**

atsim.potentials.potentialfunctions.**constant**

atsim.potentials.potentialfunctions.**exponential**

atsim.potentials.potentialfunctions.**hbnd**

atsim.potentials.potentialfunctions.**lj**

atsim.potentials.potentialfunctions.**morse**

atsim.potentials.potentialfunctions.**polynomial**

atsim.potentials.potentialfunctions.**sqrt**

atsim.potentials.potentialfunctions.**tang_toennies**

atsim.potentials.potentialfunctions.**zbl**

atsim.potentials.potentialfunctions.**zero**

atsim.potentials.potentialfunctions.**exp_spline**

atsim.potentials.tableforms

Module Contents

Classes

Cubic_Spline_Table_Form(x_data, y_data)	Potential form that takes tabulated data and returns interpolated values.

class atsim.potentials.tableforms.**Cubic_Spline_Table_Form**(*x_data*, *y_data*)

Bases: `object`

Potential form that takes tabulated data and returns interpolated values.

This potential uses cubic spline interpolation. It is simply a wrapper around the scipy.interpolate.InterpolatedUnivariateSpline class

config_label = cubic_spline

is_potential = True

interpolant

This class is a wrapper around instances of scipy.interpolate.InterpolatedUnivariateSpline This property returns the scipy object used internally

__call__(*self*, *x*)

Returns interpolated value at x

deriv(*self*, *x*)

Returns derivative of potential form at x

deriv2(*self*, *x*)

Returns second derivative of potential form at x

Package Contents

Classes

Potential(speciesA, speciesB, potentialFunction, h=1e-06)	Class used to describe a potential to the *writePotentials()* function.
EAMPotential(species, atomicNumber, mass, embeddingFunction, electronDensityFunction, latticeConstant=0.0, latticeType='fcc')	Class used to describe a particular species within EAM potential models.
SplinePotential(startPotential, endPotential, detachmentX, attachmentX)	Callable to allow splining of one potential to another using an exponential spline
Multi_Range_Defn(range_type, start, potential_form, **kwargs)	
TableReader(fileobject)	Callable that allows pretabulated data to be used with a Potential object.

Functions

gradient(func, h=1e-06)	Function wrapper that returns derivative of func.
num_deriv(r, func, h=1e-06)	Returns numerical derivative of the callable *func*

Continued on next page

Table 2.18 – continued from previous page

deriv(r, func, h=1e-06)	Evaluates the derivative of a unary callable, *func* at a value of *r*.
writeTABEAM(nrho, drho, nr, dr, eampots, pairpots, out=sys.stdout, title=")	Create `TABEAM` file for use with the `DL_POLY` simulation code.
writeTABEAMFinnisSinclair(nrho, drho, nr, dr, eampots, pairpots, out=sys.stdout, title=")	Create Exended EAM variant of DL_POLY `TABEAM` file.
writeFuncFL(nrho, drho, nr, dr, eampots, pairpots, out=sys.stdout, title=")	Creates a DYNAMO `funcfl` formatted file suitable for use with lammps pair_style eam
writeSetFL(nrho, drho, nr, dr, eampots, pairpots, out=sys.stdout, comments=['', '', ''], cutoff=None)	Creates EAM potential in the DYNAMO `setfl` format. This format is suitable for
writeSetFLFinnisSinclair(nrho, drho, nr, dr, eampots, pairpots, out=sys.stdout, comments=['', '', ''], cutoff=None)	Creates Finnis-Sinclar EAM potential in the DYNAMO `setfl` format. The format should be used with the
potential(func)	Decorator for callables that should be tagged as potential-forms or potential-functions
is_potential(obj)	Identifies if an object is a potential-form or potential-function
buck4(A, rho, C, r_detach, r_min, r_attach)	Returns a potential form describing the four-range Buckingham potential.
create_Multi_Range_Potential_Form(*range_tuples, **kwargs)	Creates Multi_Range_Potential_Form or sub-class instance, from list of Multi_Range_Defn
plus(a, b)	Takes two functions and returns a third which when evaluated returns the result of a(r) + b(r)
product(a, b)	Takes two callables and returns a third which when evaluated returns the result of a(r) * b(r)
pow(a, b)	Takes two callables and returns a third which when evaluated returns the result of a(r)**b(r)
plotToFile(fileobj, lowx, highx, func, steps=10000)	Convenience function for plotting the potential functions contained herein.
plot(filename, lowx, highx, func, steps=10000)	Convenience function for plotting the potential functions contained herein.
plotPotentialObject(filename, lowx, highx, potentialObject, steps=10000)	Convenience function for plotting energy of pair interactions
plotPotentialObjectToFile(fileobj, lowx, highx, potentialObject, steps=10000)	Convenience function for plotting energy of pair interactions
writePotentials(outputType, potentialList, cutoff, gridPoints, out=sys.stdout)	Tabulates pair-potentials in formats suitable for multiple simulation codes.

class atsim.potentials.**Potential**(*speciesA*, *speciesB*, *potentialFunction*, *h=1e-06*)

Bases: `object`

Class used to describe a potential to the *writePotentials()* function.

Potential objects encapsulate a python function or callable which is used by the *energy()* method to calculate potential energy.

The *force()* method returns $\frac{-dU}{dr}$. If the energy callable provides *.deriv()* and *.deriv2()* methods these are used for evaluating the first and second derivatives of energy with respect to sepration. This allows analytical derivatives to be defined to the Potential object. When not defined, numerical derivatives are used instead.

The *gradient()* function is used to wrap the energy callable so that the correct derivative implementation is used.

speciesA

speciesB

potentialFunction

energy(*self*, *r*)

Parameters **r** – Separation

Returns Energy for given separation

force(*self*, *r*)

Calculate force for this potential at a given separation.

If this object's potentialFunction has a .deriv() method this will be used to calculate force (allowing analytical derivatives to be specified).

If potentialFunction doesn't have a deriv method then a numerical derivative of the potential function will be returned instead.

Parameters **r** (`float`) – Separation

Returns -dU/dr at given separation

Return type float

class `atsim.potentials.`**EAMPotential**(*species*, *atomicNumber*, *mass*, *embeddingFunction*, *electronDensityFunction*, *latticeConstant=0.0*, *latticeType='fcc'*)

Bases: `object`

Class used to describe a particular species within EAM potential models.

This class is a container for the functions and attributes necesary for describing the many-body component of an Embedded Atom potential Model.

embeddingValue(*self*, *density*)

Method that returns energy for given electron density.

This method simply passes `density` to the callable stored in the `embeddingFunction` and returns its value.

Parameters **density** (`float`) – Electron density.

Returns Energy for given density (as given by `self.embeddingFunction`).

Return type float

electronDensity(*self*, *separation*)

Gives the 'electron' density for an atom separated from current species by `separation`.

This is a pass-through method to callable stored in current instance's `electronDensityFunction` attribute.

Parameters **separation** (`float.`) – Separation (in angstroms) between atom represented by this object and another atom.

Returns Contribution to electron density due to given pair separation.

Return type float.

class `atsim.potentials.`**SplinePotential**(*startPotential*, *endPotential*, *detachmentX*, *attachmentX*)

Bases: *`atsim.potentials.spline.Custom_SplinePotential`*

Callable to allow splining of one potential to another using an exponential spline `atsim.potentials.`**`gradient`**`(`*func*, *h=1e-06*`)`

Function wrapper that returns derivative of func.

If the callable, *func* provides a *.deriv(r)* method this will be used to evaluate the derivative of the function, if not the returned function will use num_deriv() in gradient evaluation.

If the callable additionally provides a *.deriv2(r)* method, representing its second derivative, the function returned by this routine will have a *deriv()* method which will delegate to func.deriv2() when called.

By providing .deriv() and .deriv2() on the *func* callable analytical descriptions of a potential's first and second derivatives may be specified.

Parameters

- **`func`** – Function to be wrapped
- **`h`** – Step size used when performing numerical differentiation

Returns Function that returns derivative of func

`atsim.potentials.`**`num_deriv`**`(`*r*, *func*, *h=1e-06*`)`

Returns numerical derivative of the callable *func*

Parameters

- **`r`** – Value at which derivative of *func* should be evaluated.
- **`func`** – Function whose gradient is to be evaluated.
- **`h`** – Step size used when performing numerical differentiation.

Returns Numerical derivative of func at *r*.

`atsim.potentials.`**`deriv`**`(`*r*, *func*, *h=1e-06*`)`

Evaluates the derivative of a unary callable, *func* at a value of *r*.

If the object *func* has a unary method *deriv(r)*, this will be used to evauluate the derivative (allowing analytical derivatives to be used).

If *func* does not have a specific *deriv(r)* method then its numerical-derivative of will be taken by calling num_deriv()

Parameters

- **`r`** – Value at which derivative of *func* should be evaluated.
- **`func`** – Function whose derivative is to be evaluated.
- **`h`** – Step size used when performing numerical differentiation.

Returns Derivative of func at *r*.

`atsim.potentials.`**`writeTABEAM`**`(`*nrho*, *drho*, *nr*, *dr*, *eampots*, *pairpots*, *out=sys.stdout*, *title=''*`)`

Create `TABEAM` file for use with the `DL_POLY` simulation code.

See also:

For a working example using this function see *Example 2b: Tabulate Al-Cu Alloy Potentials Using writeTABEAM() for DL_POLY*

Parameters

- **`nrho`** (`int`) – Number of entries in tabulated embedding functions
- **`drho`** (`float`) – Step size between consecutive embedding function entries
- **`nr`** (`int`) – Number of entries in tabulated pair potentials and density functions

- **dr** (*float*) – Step size between entries in tabulated pair potentials and density functions
- **eampots** – Potentials List of potentials.EAMPotential objects
- **pair** – Potentials List of potentials.Potential objects
- **out** (*file object*) – Python file object to which TABEAM data should be written
- **title** (*str*) – Title of TABEAM file

`atsim.potentials.`**`writeTABEAMFinnisSinclair`**(*nrho, drho, nr, dr, eampots, pairpots, out=sys.stdout, title="*)

Create Exended EAM variant of DL_POLY `TABEAM` file.

The *EAMPotential* instances within the `eampots` list are expected to provide individual density functions for each species pair in the species being tabulated. See `__init__()` for how these are specified to the *EAMPotential* constructor.

Note: The Extended EAM variant for which this function creates `TABEAM` files (i.e. metal potential type = eeam) is only supported in DL_POLY versions >= 4.05.

See also:

For a working example using this function see *Example 3b: Tabulate Al-Fe Finnis-Sinclair Potentials Using writeTABEAMFinnisSinclair() for DL_POLY*

Parameters

- **nrho** (*int*) – Number of entries in tabulated embedding functions
- **drho** (*float*) – Step size between consecutive embedding function entries
- **nr** (*int*) – Number of entries in tabulated pair potentials and density functions
- **dr** (*float*) – Step size between entries in tabulated pair potentials and density functions
- **eampots** – Potentials List of *atsim.potentials.EAMPotential* objects
- **pairpots** (*list*) – Potentials List of *atsim.potentials.Potential* objects
- **out** (*file object*) – Python file object to which `TABEAM` data should be written
- **title** (*str*) – Title of TABEAM file

`atsim.potentials.`**`writeFuncFL`**(*nrho, drho, nr, dr, eampots, pairpots, out=sys.stdout, title="*)

Creates a DYNAMO `funcfl` formatted file suitable for use with lammps pair_style eam potential form. For the pair_style eam/alloy see *writeSetFL()*.

See also:

For a working example using this function see *Example 1: Using writeFuncFL() to Tabulate Ag Potential for LAMMPS*

Parameters

- **nrho** (*int*) – Number of points used to describe embedding function
- **drho** (*float*) – Step size between rho values used to describe embedding function
- **nr** (*int*) – Number of points used for the pair-potential, and density functions
- **dr** (*float*) – Step size between r values in effective charge and density functions

- **eampots** (*list*) – List containing a single *EAMPotential* instance for species to be tabulated.
- **pairpots** (*list*) – List containing a single PairPotential instance for the X-X interaction (where X is the species represented by EAMPotential in eampots list)
- **out** (*file object*) – Python file object to which eam table file will be written
- **title** (*str*) – Title to be written as table file header

atsim.potentials.**writeSetFL**(*nrho, drho, nr, dr, eampots, pairpots, out=sys.stdout, comments=['', '', ''], cutoff=None*)

Creates EAM potential in the DYNAMO setfl format. This format is suitable for use with the LAMMPS pair_style eam/alloy.

See also:

For a working example using this function see *Example 2a: Tabulate Al-Cu Alloy Potentials Using writeSetFL() for LAMMPS*

Parameters

- **nrho** (*int*) – Number of points used to describe embedding function
- **drho** (*float*) – Increment used when tabulating embedding function
- **nr** (*int*) – Number of points used to describe density and pair potentials
- **dr** (*float*) – Separation increment used when tabulating density function and pair potentials
- **eampots** (*list*) – Instances of lammps.writeEAMTable.EAMPotential() which encapsulate information about each species
- **pairpots** (*list*) – Instance of potentials.Potential, these describe repulsive pair potential component of EAM potential
- **out** (*file object*) – Python file object into which EAM potential data should be written
- **comments** (*list*) – List containing three strings, these form the header of the created file
- **cutoff** (*float*) – Pair potential and density cutoff, if None then value of nr * dr is used.

atsim.potentials.**writeSetFLFinnisSinclair**(*nrho, drho, nr, dr, eampots, pairpots, out=sys.stdout, comments=['', '', ''], cutoff=None*)

Creates Finnis-Sinclar EAM potential in the DYNAMO setfl format. The format should be used with the LAMMPS eam/fs pair_style.

The *EAMPotential* instances within the eampots list are expected to provide individual density functions for each species pair in the species being tabulated. See atsim.potentials.EAMPotential.__init__() for how these are specified to the *atsim.potentials.EAMPotential* constructor.

See also:

For a working example using this function see *Example 3a: Tabulate Al-Fe Finnis-Sinclair Potentials Using writeSetFLFinnisSinclair() for LAMMPS*

Parameters

- **nrho** (*int*) – Number of points used to describe embedding function
- **drho** (*float*) – Increment used when tabulating embedding function

- **nr** (*int*) – Number of points used to describe density and pair potentials
- **dr** (*float*) – Separation increment used when tabulating density function and pair potentials
- **eampots** (*list*) – Instances of lammps.writeEAMTable.EAMPotential() which encapsulate information about each species
- **pairpots** (*list*) – Instance of potentials.Potential, these describe repulsive pair potential component of EAM potential
- **out** (*file object*) – Python file object into which EAM potential data should be written
- **comments** (*list*) – List containing three strings, these form the header of the created file
- **cutoff** (*float*) – Pair potential and density cutoff. If None then value of `nr * dr` is used.

atsim.potentials.**potential**(*func*)

Decorator for callables that should be tagged as potential-forms or potential-functions

atsim.potentials.**is_potential**(*obj*)

Identifies if an object is a potential-form or potential-function

atsim.potentials.**buck4**(*A*, *rho*, *C*, *r_detach*, *r_min*, *r_attach*)

Returns a potential form describing the four-range Buckingham potential.

The potential form is:

$$V(r_{ij}) = \begin{cases} A\exp(-r_{ij}/\rho), & 0 \leq r_{ij} \leq r_{\text{detach}} \\ a_0 + a_1 r_{ij} + a_2 r_{ij}^2 + a_3 r_{ij}^3 + a_4 r_{ij}^4 + a_5 r_{ij}^5, & r_{\text{detach}} < r_{ij} < r_{\text{min}} \\ b_0 + b_1 * r_{ij} + b_2 * r_{ij}^2 + b_3 * r_{ij}^3, & r_{\text{min}} \leq r_{ij} < r_{\text{attach}} \\ -\frac{C}{r_{ij}^6}, & r_{ij} \geq r_{\text{attach}} \end{cases}$$

In other words this is a Buckingham potential in which the Born-Mayer component acts at small separations and the disprsion term acts at larger separation. These two parts are linked by a fifth then third order polynomial (with a minimum formed in the spline at r_extmin).

The spline parameters are subject to the constraints that $V(r_{ij})$, first and second derivatives must be equal at the boundary points and the function must have a stationary point at *r_min*.

See also:

- `atsim.potentials.Buck4_Spline`
- `atsim.potentials.Buck4_SplinePotential`

Note: Due to the complexity of calculating the spline-coefficients this potential form does not have an equivalent in the atsim.potentials.potentialfunctions module.

Parameters

- **A** – A potential parameter.
- **rho** – potential parameter.
- **C** – C parameter.
- **r_detach** – Separation where spline starts.

- **r_min** – Location of stationary point.
- **r_attach** – End of splined region.

Returns Splined potential.

atsim.potentials.**buck**

atsim.potentials.**bornmayer**

atsim.potentials.**coul**

atsim.potentials.**constant**

atsim.potentials.**exponential**

atsim.potentials.**hbnd**

atsim.potentials.**lj**

atsim.potentials.**morse**

atsim.potentials.**polynomial**

atsim.potentials.**sqrt**

atsim.potentials.**tang_toennies**

atsim.potentials.**zbl**

atsim.potentials.**zero**

atsim.potentials.**exp_spline**

atsim.potentials.**create_Multi_Range_Potential_Form**(**range_tuples*, ***kwargs*)

Creates Multi_Range_Potential_Form or sub-class instance, from list of Multi_Range_Defn instances in *range_tuples*.

If any Multi_Range_Defn object's *.has_deriv2* are True then an instance of Multi_Range_Potential_Form_Deriv2 is returned.

If any Multi_Range_Defn object's *.has_deriv* property is True but all *.has_deriv2* are False then an instance of Multi_Range_Potential_Form_Deriv is returned.

If non of the Multi_Range_Defn objects provide analytical deriv or deriv2 methods, return Multi_Range_Potential_Form.

Parameters

- **range_tuples** – List of Multi_Range_Defn instances.
- **kwargs** – Keyword arguments passed to Multi_Range_Potential_Form constructor.

Returns See above

class atsim.potentials.**Multi_Range_Defn**(*range_type*, *start*, *potential_form*, ***kwargs*)

Bases: object

range_type

start

potential_form

has_deriv

Returns True if the potential callable provides an analytical derivative through a *.deriv()* method.

has_deriv2
Returns True if the potential callable provides an analytical derivative through a *.deriv2()* method.

deriv (*self*, *r*)

deriv2 (*self*, *r*)

atsim.potentials.**plus** (*a*, *b*)
Takes two functions and returns a third which when evaluated returns the result of a(r) + b(r)

This function is useful for combining existing potentials.

Derivatives:

If either of the potential callables (*a* and *b*) provide a .deriv() method the function returned by *plus()* will also have a *.deriv()* method. This allows analytical derivatives to be specified. If only one of *a* or *b* provide *.deriv()* then the derivative of the other callable will be evaluated numerically.

If neither function has a .deriv() method then the function returned here will also *not* have a .deriv() method.

Example:

To combine *buck()* and *hbnd()* functions from the *atsim.potentials.potentialforms* module to give:

```
A*(-r/rho) + C/r**6 + D/r**12 - E/r**10
```

this function can then be used as follows:

```
plus(buck(A,rho,C), hbnd(D,E))
```

Parameters

- **a** – First callable
- **b** – Second callable

Returns Function that when evaulated returns `a(r) + b(r)`

atsim.potentials.**product** (*a*, *b*)
Takes two callables and returns a third which when evaluated returns the result of a(r) * b(r)

This function is useful for combining existing potentials.

Derivatives:

If either of the potential callables (*a* and *b*) provide a .deriv() method the function returned by *product()* will also have a *.deriv()* method. This allows analytical derivatives to be specified. If only one of *a* or *b* provide *.deriv()* then the derivative of the other callable will be evaluated numerically.

If neither function has a .deriv() method then the function returned here will also *not* have a .deriv() method.

Parameters

- **a** – First callable
- **b** – Second callable

Returns Function that when evaulated returns `a(r) * b(r)`

atsim.potentials.**pow** (*a*, *b*)
Takes two callables and returns a third which when evaluated returns the result of a(r)**b(r)

This function is useful for combining existing potentials.

Derivatives:

If either of the potential callables (*a* and *b*) provide a .deriv() method the function returned by *pow()* will also have a *.deriv()* method. This allows analytical derivatives to be specified. If only one of *a* or *b* provide *.deriv()* then the derivative of the other callable will be evaluated numerically.

If neither function has a .deriv() method then the function returned here will also *not* have a .deriv() method.

Parameters

- **`a`** – First callable
- **`b`** – Second callable

Returns Function that when evaluated returns `a(r)**b(r)` (a to the power of b)

`class` `atsim.potentials.`**`TableReader`**(*fileobject*)

Bases: `object`

Callable that allows pretabulated data to be used with a Potential object.

`datReader`

Returns _tablereaders.DatReader associated with this callable

`__call__`(*self*, *separation*)

`atsim.potentials.`**`plotToFile`**(*fileobj*, *lowx*, *highx*, *func*, *steps=10000*)

Convenience function for plotting the potential functions contained herein.

Data is written to a text file as two columns (r and E) separated by spaces with no header.

Parameters

- **`fileobj`** – Python file object into which data should be plotted
- **`lowx`** – X-axis lower value
- **`highx`** – X-axis upper value
- **`func`** – Function to be plotted
- **`steps`** – Number of data points to be plotted

`atsim.potentials.`**`plot`**(*filename*, *lowx*, *highx*, *func*, *steps=10000*)

Convenience function for plotting the potential functions contained herein.

Data is written to a text file as two columns (r and E) separated by spaces with no header.

Parameters

- **`filename`** – File into which data should be plotted
- **`lowx`** – X-axis lower value
- **`highx`** – X-axis upper value
- **`func`** – Function to be plotted
- **`steps`** – Number of data points to be plotted

`atsim.potentials.`**`plotPotentialObject`**(*filename*, *lowx*, *highx*, *potentialObject*, *steps=10000*)

Convenience function for plotting energy of pair interactions given by instances of *`atsim.potentials.Potential`* obtained by calling *potential .energy()* method.

Data is written to a text file as two columns (r and E) separated by spaces with no header.

Parameters

- **`filename`** – File into which data should be plotted
- **`lowx`** – X-axis lower value
- **`highx`** – X-axis upper value
- **`func`** – *`atsim.potentials.Potential`* object.
- **`steps`** – Number of data points to be plotted

`atsim.potentials.`**`plotPotentialObjectToFile`**(*fileobj*, *lowx*, *highx*, *potentialObject*, *steps=10000*)

Convenience function for plotting energy of pair interactions given by instances of *`atsim.potentials.Potential`* obtained by calling *potential .energy()* method.

Data is written to a text file as two columns (r and E) separated by spaces with no header.

Parameters

- **`fileobj`** – Python file object into which data should be plotted
- **`lowx`** – X-axis lower value
- **`highx`** – X-axis upper value
- **`func`** – *`atsim.potentials.Potential`* object.
- **`steps`** – Number of data points to be plotted

`exception` `atsim.potentials.`**`UnsupportedTabulationType`**

Bases: `Exception`

Exception thrown by writePotentials() when unknown tabulation type specified

`atsim.potentials.`**`writePotentials`**(*outputType*, *potentialList*, *cutoff*, *gridPoints*, *out=sys.stdout*)

Tabulates pair-potentials in formats suitable for multiple simulation codes.

- The `outputType` parameter can be one of the following:
  - `DL_POLY`:
    - This function creates output that can be written to a `TABLE` and used within DL_POLY.
    - for a working example see *Quick-Start: DL_POLY*.
  - `GULP`:
    - Creates output for the GULP code
    - Output is in the form of a series of spline potential forms
    - The generated file can be loaded into GULP using the library command
  - `LAMMPS`:
    - Creates files readable by LAMMPS pair_style table
    - Each file can contain multiple potentials:
      - the block representing each potential has a title formed from the `speciesA` and `speciesB` attributes of the *`Potential`* instance represented by the block. These are sorted into their natural order and separated by a hyphen to form the title.
      - **Example:**
      - For a *`Potential`* where `speciesA` = Xe and `speciesB` = O the block title would be: `O-Xe`.
      - If `speciesA` = B and `speciesB` = O the block title would be: `B-O`.

- within LAMMPSthe block title is used as the `keyword` argument to the pair_style table `pair_coeff` directive.

Parameters

- **`outputType`** (*`str`*) – The type of output that should be created can be one of: `DL_POLY` or `LAMMPS`
- **`potentialList`** (*`list`*) – List of Potential objects to be tabulated.
- **`cutoff`** (*`float`*) – Largest separation to be tabulated.
- **`gridPoints`** (*`int`*) – Number of rows in tabulation.
- **`out`** (*`file`*) – Python file like object to which tabulation should be written

2.5 List of Examples

The following page gives a list of the examples that are distributed across the documentation:

Description	Link
Quick-Start: Tabulating Basak Potentials for DL_POLY	*potable* *Python API*
Quick-Start: Tabulating Basak Potentials for LAMMPS	*potable* *Python API*
[Potential-Form] Using custom-potential forms to define Basak potential	*potable*
Splining ZBL Potential on to Buckingham Potential	*potable* *Python API*
Defining the Morelon model using the Buck4 spline	*potable*
Parametrising a model using published spline coefficients	*potable*
Truncating a potential (describing LAMMPS *pair_style soft*)	*potable*
Truncating a potential using *if()* (describing LAMMPS *pair_style soft*)	*potable*
[Table-Form] pair-potential. | Including pre-tabulated data in a model.	*potable*
Instantiating `atsim.potentials.Potential` Objects	*Python API*
Tabulating EAM Ag model for LAMMPS	*potable*, *Python API (object oriented)* *Python API (procedural)*
Tabulate Al-Cu EAM Alloy Potentials	*Python API (LAMMPS object oriented)* *Python API (LAMMPS procedural)* *Python API (DL_POLY object oriented)* *Python API (DL_POLY procedural)*
Tabulate Al-Fe Finnis-Sinclair EAM potentials	*Python API (LAMMPS object oriented)* *Python API (LAMMPS procedural)* *Python API (DL_POLY object oriented)* *Python API (DL_POLY procedural)*
Finnis-Sinclair Tabulation using potable	*potable*
Working with `potable` files in Python	*Python API*
Working with `potable` files in Python Overriding and adding items	*Python API*

2.6 Credits

`atsim.potentials` is developed and maintained by Michael Rushton. It was intially developed to support the activities of the Atomistic Simulation Group located in the Department of Materials at Imperial College London. Thanks go to Prof. Robin Grimes and the rest of the group. Particular thanks must goes to:

- Michael Cooper who helped test and debug the Embedded Atom Method tabulation methods whilst we developed our actinide potential model for the following:
  - M.W.D. Cooper, M.J.D. Rushton and R. W. Grimes, "A many-body potential approach to modelling the thermomechanical properties of actinide oxides", *J. Phys. Condens. Matter*, 2014 **26** 105401. doi:10.1088/0953-8984/26/10/105401
- **Dr. Clare Bishop** for providing an early implementation of the spline interpolation method implemented within *`atsim.potentials.spline.SplinePotential`*.

2.7 Changes

2.7.1 0.4.0

New Features

- Added support for angular dependent potential (ADP) models (LAMMPS `pair_style adp`). See *ADP Style EAM Models*.
- Support for `[Variables]` section in `potable` files which allows use of string snippets and string interpolation. See *[Variables]*.

2.7.2 0.3.0 (2020-8-24)

New Features

- Introduced the new potable tool to allow tabulation without needing to write a python script.
- Revamped python api.

Bug-Fixs

- The order that the density functions were specified to the writeSetFLFinnisSinclair() function was the reverse of what would be expected. This has been fixed.

2.7.3 0.2.1 (2018-05-19)

Bug-Fixes

- Fix to the plot functions. Previously all x-axis values were being set to the `lowx` value given to the function.

2.7.4 0.2.0 (2018-05-01)

New Features

- Support for Python 3

2.7.5 0.1.1 (2014-03-25)

- Initial Release

2.8 License

atsim.potentials is released under the terms of the Apache License

2.9 References

CHAPTER 3

Contact

`atsim.potentials` was developed by Michael Rushton, if you have any problems, suggestions or queries please get in touch at m.j.d.rushton@gmail.com

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Bibliography

[Basak2003] C.Basak, "Classical molecular dynamics simulation of UO_2 to predict thermophysical properties", *Journal of Alloys and Compounds*, **360** (2003) 210. http://doi.org/doi:10.1016/S0925-8388(03)00350-5

[VanBeest1990] B.W.H. van Beest, G.J. Kramer, R.A. van Santen, "Force fields for silicas and aluminophosphates based on ab initio calculations", *Phys. Rev. Lett.* **64** (1990) 1955. http://doi.org/10.1103/PhysRevLett.64.1955

[Buckingham1938] R. A. Buckingham, "The Classical Equation of State of Gaseous Helium, Neon and Argon", *Proc. R. Soc. London. Ser. A, Math. Phys. Sci.*, **168** (1938) 264. http://doi.org/10.1098/rspa.1938.0173

[Lennard-Jones1924] J.E. Lennard-Jones, "On the Determination of Molecular Fields. — II. From the Equation of State of a Gas", *Proc. R. Soc. Lond. A*, **106** (1924) 463. http://10.1098/rspa.1924.0082

[Morelon2003] N.-D. Morelon, D. Ghaleb, J.-M. Delaye, L. Van Brutzel, "A new empirical potential for simulating the formation of defects and their mobility in uranium dioxide", *Philos. Mag.* **83** (2003) 1533. http://doi.org/10.1080/1478643031000091454

[Potashnikov2011] S.I. Potashnikov, A.S. Boyarchenkov, K.A. Nekrasov, A.Y. Kupryazhkin, "High-precision molecular dynamics simulation of UO_2–PuO_2: Pair potentials comparison in UO_2", *J. Nucl. Mater.* **419** (2011) 217. http://doi.org/10.1016/j.jnucmat.2011.08.033

[Tang2003] K.T. Tang, J.P. Toennies, "The van der Waals potentials between all the rare gas atoms from He to Rn", *J. Chem. Phys.* **118** (2003) 4976. https://doi.org/10.1063/1.1543944

[Vessal1989] B. Vessal, M. Amini, D. Fincham, C.R.A Catlow, "Water-like melting behaviour of SiO_2 investigated by the molecular dynamics simulation technique", *Philos. Mag. B* **60** (1989) 753. http://doi.org/10.1080/13642818908209741

[Vessal1993] B. Vessal, M. Amini, C.R.A. Catlow, "Computer simulation of the structure of silica glass", *J. Non. Cryst. Solids.* **159** (1993) 184. http://doi.org/10.1016/0022-3093(93)91295-E

[Ziegler2015] J.F. Ziegler, J.P. Biersack, M.D. Ziegler, SRIM - The Stopping and Range of Ions in Matter, 15th ed., IIT Co., 2015. http://www.lulu.com/shop/james-ziegler/srim-the-stopping-and-range-of-ions-in-matter/hardcover/product-22155781.html

Python Module Index

Index