atsim_potentials Documentation

Release 0.1.1

M.J.D. Rushton

Sep 27, 2017

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.

The following codes are supported for pair-potential tabulation:

- LAMMPS
- DL_POLY

Embedded Atom Model (EAM) potential tabulation is supported in the following formats:

- DYNAMO - as used by LAMMPS and several other codes.
- DL_POLY

CHAPTER 1

Contents

Installation

Install Using Pip

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

```
pip install atsim.potentials
```

Install from Source

The source is hosted on bitbucket and can be cloned using mercurial as follows:

```
hg clone https://bitbucket.org/mjdr/atsim_potentials
```

alternatively a tarball of the source can be downloaded from the bitbucket downloads page 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):

```
python setup.py build_sphinx
```

This will place documents in `build/sphinx/html` within the source tree.

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

Quick-Start

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

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`.

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

import atsim.potentials
from atsim.potentials import potentialforms

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 = atsim.potentials.plus(buck_OU, morse_OU)

    # Construct list of Potential objects
    potential_objects = [
        atsim.potentials.Potential('O', 'O', f_OO),
        atsim.potentials.Potential('U', 'U', f_UU),
        atsim.potentials.Potential('O', 'U', f_OU)
    ]
    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)
    with open('TABLE', 'wb') as outfile:
```

[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

```
        atsim.potentials.writePotentials(
           'DL_POLY',
           potential_objects,
           6.5, 6500,
           out = 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* together with labels name 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):

```
    # 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 = atsim.potentials.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)
    with open('TABLE', 'wb') as outfile:
        atsim.potentials.writePotentials(
           'DL_POLY',
           potential_objects,
           6.5, 6500,
           out = outfile)
```

- First the `makePotentialObjects()` function is called, returning a list of *`Potential`* that are stored in the `potential_objects` variable.
- The *`writePotentials()`* function is then called with this list to create potentials with a maximum cut-off of 6.5Å and 6500 rows (i.e. a grid increment of 0.001 Å).
- 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)
    with open('Basak.lmptab', 'wb') as outfile: # <-- Filename changed from 'TABLE'
        atsim.potentials.writePotentials(
           'LAMMPS', # <-- This has been changed from 'DL_POLY'
           potential_objects,
           6.5, 6500,
           out = outfile)
```

Only the two highlighted lines have been changed:

1. the first changes the output filename to `Basak.lmptab`
2. the second has been changed from `DL_POLY` to `LAMMPS` in order to select the desired tabulation format.

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 the convenience function *atsim.potentials.writePotentials()*. This function is supplied with a list of Potential objects, which have *PotentialInterface.energy()* and *PotentialInterface.force()* methods called during tabulation to obtain potential-energy as a function of separation and its derivative respectively.

atsim.potentials.writePotentials()

The *atsim.potentials.writePotentials()* function is used to tabulate pair-potentials for the supported simulation codes.

The process by which *writePotentials()* is used can be summmarised as follows:

1. Define python functions describing energy of interactions (see *Example: Instantiating atsim.potentials.Potential Objects* and *Predefined Potential Forms*)
2. Wrap these functions in *Potential* objects.
3. Call *writePotentials()* with list of *Potential* objects choosing:
   - Tabulation type
   - Potential cut-off
   - Number of rows in tabulation
   - Python file like object into which data should be written.

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 *atsim.potentials.Potential.energy()* method. The forces calculated by the *atsim.potentials.Potential.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:

```
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:

```
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:

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

Predefined Potential Forms

In the previous example (*instantiate_potential_object*), 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:

```
import math

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

can be rewritten as:

```
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 *Quick-Start* example, where the oxygen-uranium pair potential was the combination of a Buckingham and Morse potential forms.

Such potential combinations can be made using the `plus()` function from the `atsim.potentials` module:

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.

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.

Example: Splining ZBL Potential onto Buckingham Potential

As mentioned above, 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)[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:

[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

- 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 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.

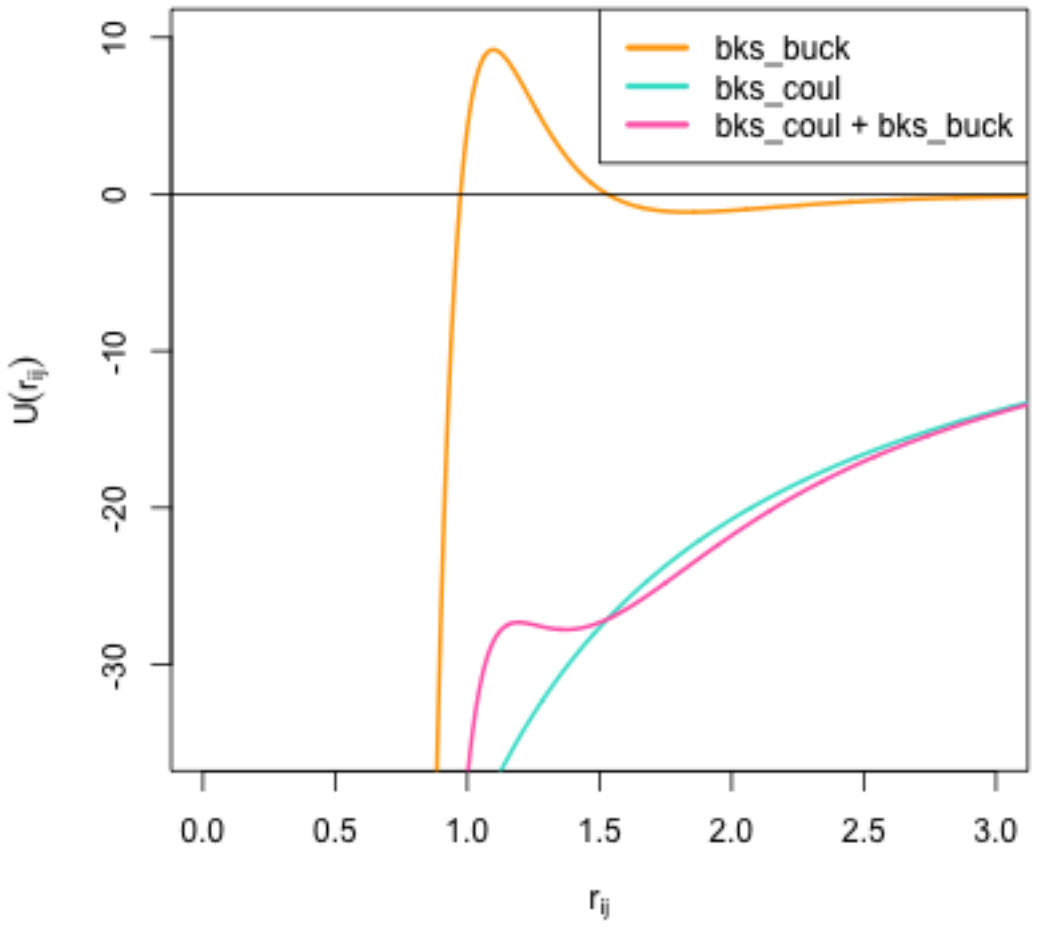

Fig. 1.1: Plot of BKS Si-O potential showing the short-range (bks_buck) component, electrostatic (bks_coul) and the effective Si-O interaction (bks_buck + bks_coul). This shows that this potential turns over at small separations making it unsuitable for use where high energies may be experienced such as high-temperature or radiation damage cascade simulations.

The first step to using *SplinePotential* is to choose appropriate detachment and attachment points. This is per-

haps 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.

```
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)

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:

Finally, the potential can be tabulated in a format suitable for LAMMPS using *atsim.potentials.writePotentials()*:

```
bks_SiO = atsim.potentials.Potential('Si', 'O', spline)
with open('bks_SiO.lmptab', 'wb') as outfile:
    atsim.potentials.writePotentials('LAMMPS', [bks_SiO], 10.0, 5000, out = 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. In addition to the differences in table files expected by different simulation codes, there are several variations on the embedded atom method, in order to support this variety, the `atsim_potentials` module contains several tabulation 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*

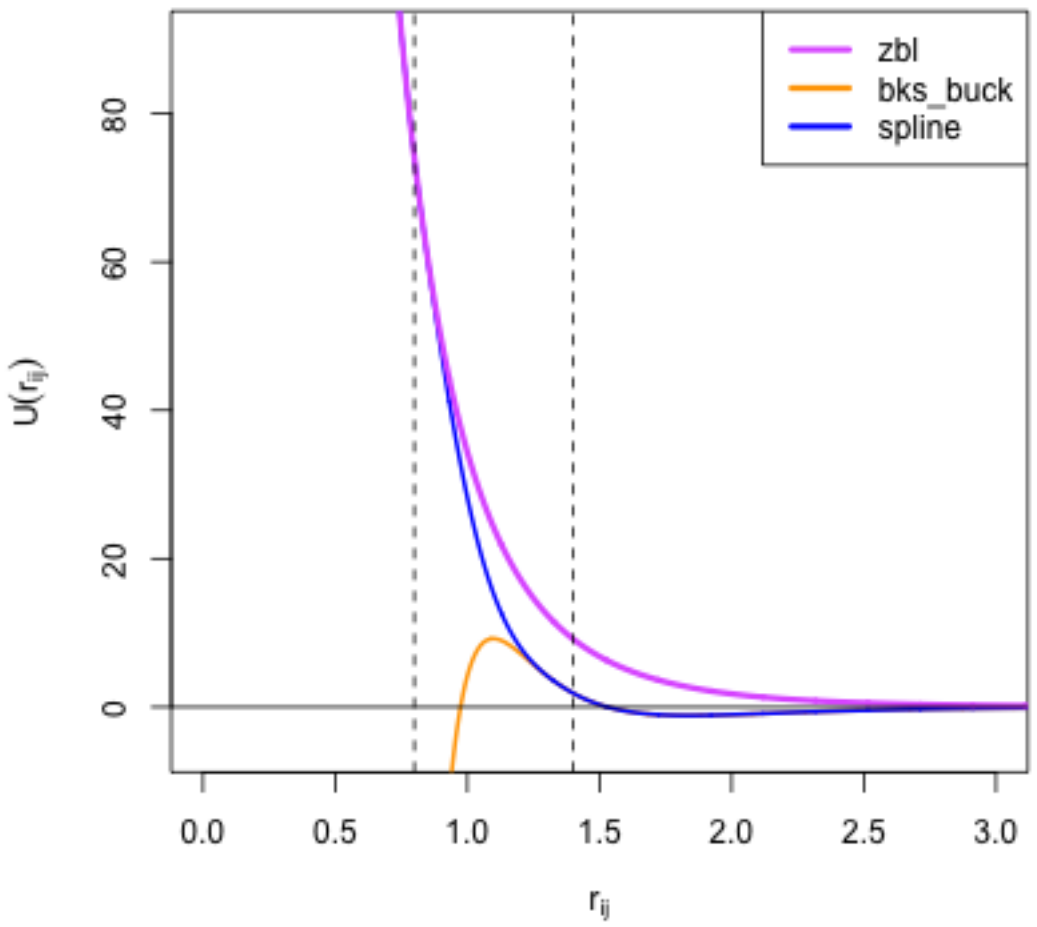

Fig. 1.2: Plot of BKS Si-O interaction showing the short-range (buck) and ZBL functions plotted with the curve generated by `SplinePotential` (spline). This joins them with a an interpolating spline acting between the detachment point at $r_{ij} = 0.8$ and re-attachment point at $r_{ij} = 1.4$ shown by dashed lines.

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`.

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.

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:

```
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
```

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

$\sqrt{\frac{\phi(r_{ij})r_{ij}}{27.2\times 0.529}}$

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 *Potential* objects, can be used to tabulate the Ag potential into the `Ag.eam` file:

```
  nrho = 50000
  drho = 0.001

  nr = 12000
  dr = 0.001

  from atsim.potentials import writeFuncFL

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

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.

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:

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

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

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:

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

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

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):
  # 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 xrange(4)])

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

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:

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

Now we have all the objects required for *writeSetFL()*. The next excerpt call `makeObjects()` to get the EAM and pair-potential objects before invoking the tabulation function, writing the data into a file called `Zhou_AlCu.setfl`:

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

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

  nr = 2000
  dr = 0.003

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

Putting this all 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.setfl` 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
```

```
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 xrange(4)])

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

  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()

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

```
  nr = 2000
  dr = 0.003

  with open("Zhou_AlCu.setfl", 'wb') 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()
```

Using the `Zhou_AlCu.setfl` file within LAMMPS

Within LAMMPS the `setfl` files generated by *`writeSetFL()`* 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.setfl Al
```

Likewise if a copper system was being simulated:

```
pair_style eam/alloy
pair_coeff * * Zhou_AlCu.setfl 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.setfl Al Cu
```

Or if Cu was 1 and Al 2:

```
pair_style eam/alloy
pair_coeff * * Zhou_AlCu.setfl Cu Al
```

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. See the tabulation script for this example: `eam_tabulate_example2b.py`.

The *EAMPotential* and *Potential* lists are created in exactly the same way as *Example 2a*, however rather than calling *writeSetFL()* the `main()` function is modified to use the DL_POLY specific *writeTABEAM()* function instead and to write into a file named `TABEAM`. The `main()` function of `eam_tabulate_example2b.py` is now given:

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

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

  nr = 2000
  dr = 0.003

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

Using the `TABEAM` file with DL_POLY

Running `eam_tabulate_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 `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 file format created by *writeSetFLFinnisSinclair()* 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

[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.

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 to *`writeSetFLFinnisSinclair()`*.

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 `writeSetFLFinnisSinclair` 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_tabulate_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 $F_{\mathrm{Fe}}(\rho)$.
- **Density-Functions:**
  - `def AlAlDensityFunction(r):` - Al density function $\rho_{\mathrm{AlAl}}(r_{ij})$.
  - `def FeFeDensityFunction(r):` - Fe density function $\rho_{\mathrm{AlAl}}(r_{ij})$.
  - `def FeAlDensityFunction(r):` - Al-Fe density function $\rho_{\mathrm{AlFe}}(r_{ij})$.

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_tabulate_example3a.py`.

These functions are used within the `main()` function of the `eam_tabulate_example3a.py` file which is now shown:

```
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", "wb") as outfile:
  writeSetFLFinnisSinclair(
    nrho, drho,
    nr, dr,
    eamPotentials,
    pairPotentials,
    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:

```
# 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 the use of *writeSetFLFinnisSinclair()* differs from *writeSetFL()*, as a dictionary of density functions is passed to the constructor instead of the single function used previously (see highlighted lines):

```
# 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') ]
```

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 the *writeSetFLFinnisSinclair()* function is called, in this case writing the data to `Mendelev_Al_Fe.eam.fs` in the current directory:

```
# 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", "wb") as outfile:
  writeSetFLFinnisSinclair(
    nrho, drho,
    nr, dr,
    eamPotentials,
    pairPotentials,
    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 `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", "wb") 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", "wb") as outfile:
    writeTABEAMFinnisSinclair(
      nrho, drho,
      nr, dr,
      eamPotentials,
      pairPotentials,
      outfile)
```

That's it, nothing else has changed.

Using the `TABEAM` file with DL_POLY

Running `eam_tabulate_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.

List of Examples

The following page gives a list of the examples that are distributed across the documentation:

API Reference

atsim.potentials

Pair Potential Tabulation

class atsim.potentials.**Potential**(*speciesA*, *speciesB*, *potentialFunction*)

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. This callable is also used when calculating forces $\frac{-dU}{dr}$ through the *force()* method. Forces are calculated by using a simple finite difference method to find the linear gradient of the energy for a given separation.

__init__(*speciesA*, *speciesB*, *potentialFunction*)

Create a Potential object from a python function or callable that returns energy at a given separation.

Parameters

- **speciesA** – Label of first species in the potential pair
- **speciesB** (*str*) – Label of second species in the potential pair
- **potentialFunction** – Python callable which accepts a single parameter (separation) and returns energy for that separation

energy(*r*)

Parameters **r** – Separation

Returns Energy for given separation

force(*r*, *h=1e-06*)

Calculate force for this potential at a given separation.

Force is calculated as -dU/dr by performing numerical differentiation of function.

Parameters

- **r** (*float*) – Separation
- **h** (*float*) – Distance increment used when calculating energy derivative centred on r

Returns -dU/dr at given separation

Return type float

speciesA

speciesB

atsim.potentials.**writePotentials**(*outputType*, *potentialList*, *cutoff*, *gridPoints*, *out=<open file '<stdout>', mode 'w'>*)

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*.

–`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.

∗For a working example see *Quick-Start: LAMMPS*

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

Embedded Atom Method Tabulation

class atsim.potentials.**EAMPotential**(*species*, *atomicNumber*, *mass*, *embeddingFunction*, *electronDensityFunction*, *latticeConstant=0.0*, *latticeType='fcc'*)

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.

__init__(*species*, *atomicNumber*, *mass*, *embeddingFunction*, *electronDensityFunction*, *latticeConstant=0.0*, *latticeType='fcc'*)

Create EAMPotential object.

Note on electronDensityFunction parameter

In the conventional Embedded Atom Method each `EAMPotential` encapsulates a single density function (density functions are mixed by the simulation code for alloy systems). In such cases the `electronDensityFunction` constructor parameter is a single python callable.

The Finnis-Sinclair extension of the Embdedded Atom Method allows density functions for specific pairs of species to be used. This requires a callable to be given for each species pair. This is achieved by passing a dictionary of callables/functions to the `electronDensityFunction` constructor parameter. This has the general form:

```
{ SPECIES_1 : DENSITY_FUNCTION_1,
  SPECIES_2 : DENSITY_FUNCTION_2,
  ...
  SPECIES_N : DENSITY_FUNCTION_N}
```

Where the keys `SPECIES_N` represent the species of atoms surrounding a central atom with the `species` passed to the `EAMPotential` constructor and `DENSITY_FUNCTION_N` is a unary function that takes the separation between the central atom and the surrounding atom and returns the electron density due to the surrounding atom.

Example: For a binary system containing Al and Cu, the `EAMPotential` constructor may be invoked as follows to create an object representing Al for use with the Finnis-Sinclair style EAM tabulation functions:

```
alPotential = EAMPotential(
   "Al",
   13,
   26.98,
   embeddingFunction,
   {"Al" : density_Al_Al, # specifies Al-Al density function
    "Cu" : density_Al_Cu, # specifies Al-Cu density function
   })
```

See also:

- For working examples of how to use `EAMPotential` with the conventional Embedded Atom Model:
  - *Example 1: Using writeFuncFL() to Tabulate Ag Potential for LAMMPS*
  - *Example 2a: Tabulate Al-Cu Alloy Potentials Using writeSetFL() for LAMMPS*
  - *Example 2b: Tabulate Al-Cu Alloy Potentials Using writeTABEAM() for DL_POLY*
- For complete examples using the Finnis-Sinclair version of the method:
  - *Example 3a: Tabulate Al-Fe Finnis-Sinclair Potentials Using writeSetFLFinnisSinclair() for LAMMPS*

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

Parameters

- **species** – Species label
- **atomicNumber** – Atomic number
- **mass** – Atomic mass
- **embeddingFunction** – Python callable that retuns embedding values as function of density
- **electronDensityFunction** – Python callable that returns density as function of separation in angstroms. For write functions that accept multiple density functions (e.g. *writeSetFLFinnisSinclair()* and *writeTABEAMFinnisSinclair()*) a dictionary mapping species to a function object is expected for this parameter. See above.
- **latticeConstant** – Lattice constant for this species
- **latticeType** – Lattice type e.g. 'fcc' for this

electronDensity(*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.

embeddingValue(*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

atsim.potentials.**writeFuncFL**(*nrho*, *drho*, *nr*, *dr*, *eampots*, *pairpots*, *out=<open file '<stdout>', mode 'w'>*, *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=<open file '<stdout>', mode 'w'>, 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=<open file '<stdout>', mode 'w'>, 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 *EAMPotential.__init__()* for how these are specified to the *EAMPotential* constructors.

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.**writeTABEAM**(*nrho*, *drho*, *nr*, *dr*, *eampots*, *pairpots*, *out=<open file '<stdout>', mode 'w'>*, *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=<open file '<stdout>', mode 'w'>*, *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 *EAMPotential.__init__()* for how these are specified to the *EAMPotential* constructors.

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 *EAMPotential* objects
- **pairpots** (*list*) – Potentials List of *Potential* objects
- **out** (*file object*) – Python file object to which `TABEAM` data should be written
- **title** (*str*) – Title of TABEAM file

Miscellaneous Functions

`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.`**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.**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

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.

Example:

To combine *buck()* and *hbnd()* functions from the `potentialsforms` 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)`

class atsim.potentials.**SplinePotential**(*startPotential*, *endPotential*, *detachmentX*, *attachmentX*)

Callable to allow splining of one potential to another

__init__(*startPotential*, *endPotential*, *detachmentX*, *attachmentX*)

Joins `startPotential` to `endPotential` using exponential spline of the form:

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

The spline coefficients $B_{0\ldots5}$ can be obtained using the `splineCoefficients()` property.

See also:

- *Spline Interpolation*
- *Example: Splining ZBL Potential onto Buckingham Potential*

Parameters

- **startPotential** – Function defining potential for rij <= detachmentX
- **endPotential** – Function defining potential for rij => attachmentX
- **detachmentX** – rij value at which startPotential should end
- **attachmentX** – rij value at which splines join endPotential

attachmentX

Returns Point at which spline should end

detachmentX

Returns Point at which spline should start

endPotential

Returns Function defining potential for separations > `attachmentX`

interpolationFunction

Returns Exponential spline function connecting startPotential and endPotential for separations `detachmentX` < rij < `attachmentX`

splineCoefficients

Returns Tuple containing the six coefficients of the spline polynomial

startPotential

Returns Function defining potential for separations < `detachmentX`

class `atsim.potentials.`**TableReader**(*fileobject*)

Callable that allows pretabulated data to be used with a Potential object.

datReader

Returns _tablereaders.DatReader associated with this callable

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.

`atsim.potentials.potentialforms.`**bornmayer**(*A*, *rho*)

Return a Born-Mayer potential function for the given parameters

$$U(r_{ij}) = A \exp\left(\frac{-r_{ij}}{\rho}\right)$$

Parameters

- **A** – Potential parameter
- **rho** – Potential parameter ρ

Returns Function that will evaluate Born-Mayer potential for given A and rho parameters

atsim.potentials.potentialforms.**buck**(*A*, *rho*, *C*)

Returns a Buckingham potential function for a given set of A, rho and C parameters

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

Parameters

- **A** – Buckingham A parameter
- **rho** – Buckingham rho parameter ρ
- **C** – Buckingham C parameter

Returns Function that will evaulate Buckingham potential for given A, rho and C

atsim.potentials.potentialforms.**coul**(*qi*, *qj*)

Coulomb potential (including $4\pi\epsilon_0$ term).

$$U(r_{ij}) = \frac{q_i q_j}{4\pi\epsilon_0 r_{ij}}$$

Note: Constant value appropriate for r_{ij} in angstroms and energy in eV.

Parameters

- **qi** – Charge on species i
- **qj** – Charge on species j

Returns Coulomb callable

atsim.potentials.potentialforms.**hbnd**(*A*, *B*)

Returns a function with the DL_POLY hbnd form:

$$U(r_{ij}) = \frac{A}{r_{ij}^{12}} - \frac{B}{r_{ij}^{10}}$$

Parameters

- **A** – Potential A parameter
- **B** – Potentials' B parameter

Returns Function that will evaulate energy for potential of hbnd form for given A and B parameters

atsim.potentials.potentialforms.**lj**(*epsilon*, *sigma*)

Lennard-Jones 12-6 potential.

$$U(r_{ij}) = 4\epsilon\left(\frac{\sigma^{12}}{r_{ij}^{12}} - \frac{\sigma^6}{r_{ij}^6}\right)$$

Parameters

- **epsilon** – Epsilon parameter ϵ
- **sigma** – Sigma parameter σ

Returns Lennard Jones callable

atsim.potentials.potentialforms.**morse**(*gamma*, *r_star*, *D*)

Return morse function parametrised with gamma, r_start and D.

Potential function is (where r is interatomic separation):

$$U(r_{ij}) = D\left[\exp\left(-2\gamma(r_{ij} - r_*)\right) - 2\exp\left(-\gamma(r - r_*)\right)\right]$$

Parameters

- **gamma** – Potential parameter γ
- **r_star** – Potential parameter r_*
- **D** – Potential parameter

Returns Morse potential callable

atsim.potentials.potentialforms.**zbl**(*z1*, *z2*)

ZBL potential.

Ziegler-Biersack-Littmark screened nuclear repulsion for describing high energy interactions.

Parameters

- **z1** – Atomic number of species i
- **z2** – Atomic number of species j

Returns ZBL callable

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 *SplinePotential*.

Changes

- 0.1.1: 2014-03-25
  - Added "Changes" to documentation.
  - Added link to http://atsim.readthedocs.org to README.rst.
- 0.1.0: 2014-03-22

- Initial release.

License

atsim.potentials is released under the terms of the Apache License

Apache License

Version 2.0, January 2004

http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

   "Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

> To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright 2014 M.J.D. Rushton

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

> http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

CHAPTER 2

Contact

`atsim.potentials` was developed by Michael Rushton, if you have any problems, suggestions or queries please get in touch (contact details are available from my homepage).

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

Symbols

A

B

C

D

E

F

H

I

L

M

P

S

T

W

Z