ARC (Alkali Rydberg Calculator) is a package of routines written in Python, using object-oriented programming (OOP) to make modular, reusable and extendable collection of routines and data for performing useful calculations of single atom and two-atom properties, like level diagrams, interactions and transition strengths for alkali and alkaline earth metal atoms.
1.1 Installation instructions

1.1.1 Prerequisite: Python

Install Python and packages for scientific computing in Python (scipy, numpy, matplotlib). The package is tested and works with both Python 2.7 and Python 3.*. We recommend installing Python distributions that comes with Numpy that is connected to the optimized numeric libraries like ATLAS. One such distribution is Anaconda, that provides ATLAS support and optimized math kernel.

1.1.2 Recommended installation: via Python pip

Users can simply install the package from command line:

```
pip install ARC-Alkali-Rydberg-Calculator
```

This finishes installation. This should work on all operating systems (Linux, OSX, Windows) with Python 2.7, Python 3.*. Users should contact developers if unexpected errors occur.

That is all, enjoy using ARC package. Check IPython notebook with examples to see some ideas where to start.

Below we describe some old (legacy) installation methods.

1.1.3 Some legacy installation methods

This legacy methods are still useful in development process.

**Download the ARC library/package**

Download latest release for your operating system, unzip the archive and set the folder somewhere within the Python package search path or directly in your project directory. Simply import and use the module:

```
from arc import *
# write your code that uses ARC then.
```
It is important that package is stored somewhere where user has write permissions, so that it can update the databases with atomic properties. **This is the end of the standard installation for majority of the users.**

### Installation of the package globally with setup.py

If you have Linux/UNIX (as it is tested on it) and you are sure that you don’t want to change underlying ARC code. Make sure you have C compiler and **python development headers** installed. To compile and install for local user ARC call from terminal:

```
python setup.py build
python setup.py install
```

Databases that have to be changed with new values will be locally copied from package data location to `~/.arc-data` folder when arc is used for the first time.

### Compiling C extension

If you do need to compile C extension yourself, this is how to do it without installing globally the package (as in the previos section “Installation of the package globally with setup.py”). Optimized version of the Numerov is provided as the C code `arc_c_extensions.c`. **You don't need to perform this step** of manual compilation of that code if you followed recommended installation instruction by downloading precompiled binary distribution for the latest release. Note that path to arc directory should not contain spaces in order to setupc.py script to work.

#### For Windows users

If precompiled binaries don’t work, please contact developers. Compiling Numpy C extensions on Windows is a bit complicated due to use of C89 standard (instead of C99). Procedure is the following. One needs to use [MSVC compiler](https://visualstudio.microsoft.com) in order to compile Numpy extension for Python 2.7 under Windows. For other Python versions (3.5) find correct compiler [here](https://python.org). After installation of the compiler, find in Start menu “Visual C++ 2008 32-bit Command Prompt” (for 32-bit Python) or “Visual C++ 2008 64-bit Command Prompt” (for 64-bit Python). Set the following variables set in the command prompt environment:

```
SET DISTUTILS_USE_SDK=1
SET MSSdk=1
python setupc.py build_ext --inplace
```

This should build C Numpy extension (implementing Numerov integration) under Windows. We recommend, however, using pre-build binaries available on the [release page](https://release.com).

#### For Linux users

Download and install GNU C compiler. Then with terminal open, navigate to arc folder where `setupc.py` file is located execute:

```
python setupc.py build_ext --inplace
```

#### For Mac users

Download and install GNU C compiler. Then with terminal open, navigate to arc folder where `setupc.py` file is located execute:

```
python setupc.py build_ext --inplace
```

### Slow alternative: Numerov implemented in pure Python

Alternative solution, if you don’t want to compile anything, is to use pure Python implementation of the Numerov, provided in the package. This is done by passing `cpp_numerov = False` flag whenever atoms are initialized, e.g:

```
atom = Rubidium(cpp_numerov=False)
```

This is not recommended option for complex calculations, since it will run much more slowly then optimized C version, but is fine if you need just a few numbers.
1.2 Getting started with ARC

1.2.1 IPython notebook with examples

Rydberg atoms - a primer introduces Rydberg atoms and ARC package, and is a good starting point to learn how to use ARC to get relevant information about alkali metal Rydberg atoms. Notebook can also be downloaded in .ipython format here, and can be interactively then modified and used in a Jupyter.

1.2.2 On demand examples from online Atom calculator

You can try using the package without installing anything on your computer. Simply point your web browser from your computer, tablet or phone to atomcalc.jqc.org.uk and use ARC online.

Online version also generates the correct code necessary for answering the questions you ask, which can be downladed and used as a starting point for running the package locally on your computer.

1.2.3 Frequently asked questions (FAQ)

If you have a question how to do a common calculation, we recommend checking above mentioned Rydberg atoms - a primer IPython notebook. For general questions about the package usage check here:

1. How to save calculation (or matrix) for later use?

Calculations of pair-state interactions PairStateInteractions and Stark maps StarkMap can be easily saved at any point by calling alkali_atom_functions.saveCalculation. This can be loaded later by using alkali_atom_functions.loadSavedCalculation and calculation can be continued from that point.

2. How to export results?

If you want to export results e.g. for analysis and plotting in other programs, you can use calculations_atom_pairstate.PairStateInteractions.exportData and calculations_atom_single.StarkMap.exportData to export results of Stark map and Pair-state interaction calculations in .csv format. See documentation of corresponding functions for more details.

3. Calculation is not outputting anything? How long does it take for calculation to finish?

Most of the functions have progressOutput and debugOutput as an optional parameter (by default set to False) - check documentation of individual functions for details. We recommend setting at least progressOutput=True so that you have minimum output about the status of calculations. This often displays percentage of the current calculation that is finished, that you can use to estimate total time. Setting debugOutput=True outputs even more verbose output, like states in the selected basis, and individual coupling strengths etc.

1.3 Detailed documentation of functions

1.3.1 Alkali atom functions

Overview

Classes and global methods
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlkaliAtom(preferQuantumDefects,</td>
<td>Implements general calculations for alkali atoms.</td>
</tr>
<tr>
<td>cpp_numerov)</td>
<td></td>
</tr>
<tr>
<td>NumerovBack(innerLimit, outerLimit, kfun, ...)</td>
<td>Full Python implementation of Numerov integration</td>
</tr>
<tr>
<td>saveCalculation(calculation, fileName)</td>
<td>Saves calculation for future use.</td>
</tr>
<tr>
<td>loadSavedCalculation(fileName)</td>
<td>Loads previously saved calculation.</td>
</tr>
<tr>
<td>printState(n, l, j[, s])</td>
<td>Prints state spectroscopic label for numeric n, l, s label of the state.</td>
</tr>
<tr>
<td>printStateString(n, l, j[, s])</td>
<td>Returns state spectroscopic label for numeric n, l, j label of the state.</td>
</tr>
<tr>
<td>printStateStringLatex(n, l, j[, s])</td>
<td>Returns latex code for spectroscopic label for numeric n, l, j label of the state.</td>
</tr>
</tbody>
</table>

**AlkaliAtom Methods**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlkaliAtom.getDipoleMatrixElement(n1, l1, ...)</td>
<td>Dipole matrix element $\langle n_1 l_1 j_1 m_j 1</td>
</tr>
<tr>
<td>AlkaliAtom.getDipoleMatrixElementHFS(n1, l1, ...)</td>
<td>Dipole matrix element for hyperfine structure resolved transitions $\langle n_1 l_1 j_1 f_1 m_f 1</td>
</tr>
<tr>
<td>AlkaliAtom.getTransitionWavelength(n1, l1, ...)</td>
<td>Calculated transition wavelength (in vacuum) in m.</td>
</tr>
<tr>
<td>AlkaliAtom.getTransitionFrequency(n1, l1, ...)</td>
<td>Calculated transition frequency in Hz</td>
</tr>
<tr>
<td>AlkaliAtom.getRabiFrequency(n1, l1, j1, mj1, ...)</td>
<td>Returns a Rabi frequency for resonantly driven atom in a center of TEM00 mode of a driving field</td>
</tr>
<tr>
<td>AlkaliAtom.getRabiFrequency2(n1, l1, j1, ...)</td>
<td>Returns a Rabi frequency for resonant excitation with a given electric field amplitude</td>
</tr>
<tr>
<td>AlkaliAtom.getStateLifetime(n, l, j[, . . .])</td>
<td>Returns the lifetime of the state (in s)</td>
</tr>
<tr>
<td>AlkaliAtom.getTransitionRate(n1, l1, j1, n2, . . .)</td>
<td>Transition rate due to coupling to vacuum modes (black body included)</td>
</tr>
<tr>
<td>AlkaliAtom.getReducedMatrixElementJ_aymmetric(n1, l1, ...)</td>
<td>Reduced matrix element in $J$ basis, defined in asymmetric notation.</td>
</tr>
<tr>
<td>AlkaliAtom.getReducedMatrixElementJ(n1, l1, ...)</td>
<td>Reduced matrix element in $J$ basis (symmetric notation)</td>
</tr>
<tr>
<td>AlkaliAtom.getReducedMatrixElementL(n1, l1, ...)</td>
<td>Reduced matrix element in $L$ basis (symmetric notation)</td>
</tr>
<tr>
<td>AlkaliAtom.getRadialMatrixElement(n1, l1, ...)</td>
<td>Radial part of the dipole matrix element</td>
</tr>
<tr>
<td>AlkaliAtom.getQuadrupoleMatrixElement(...)</td>
<td>Radial part of the quadrupole matrix element</td>
</tr>
<tr>
<td>AlkaliAtom.getPressure(temperature)</td>
<td>Vapour pressure (in Pa) at given temperature</td>
</tr>
<tr>
<td>AlkaliAtom.getNumberDensity(temperature)</td>
<td>Atom number density at given temperature</td>
</tr>
<tr>
<td>AlkaliAtom.getAverageInteratomicSpacing(...</td>
<td>Returns average interatomic spacing in atomic vapour</td>
</tr>
<tr>
<td>AlkaliAtom.corePotential(l, r)</td>
<td>Core potential felt by valence electron</td>
</tr>
<tr>
<td>AlkaliAtom.effectiveCharge(l, r)</td>
<td>Effective charge of the core felt by valence electron</td>
</tr>
<tr>
<td>AlkaliAtom.potential(l, s, j, r)</td>
<td>Returns total potential that electron feels</td>
</tr>
<tr>
<td>AlkaliAtom.radialWavefunction(l, s, j, ...</td>
<td>Radial part of electron wavefunction</td>
</tr>
<tr>
<td>AlkaliAtom.getEnergy(n, l, j[, s])</td>
<td>Energy of the level relative to the ionisation level (in eV)</td>
</tr>
<tr>
<td>AlkaliAtom.getZeemanEnergyShift(l, ...</td>
<td>Returns linear (paramagnetic) Zeeman shift.</td>
</tr>
<tr>
<td>AlkaliAtom.getQuantumDefect(n, l, j[, s])</td>
<td>Quantum defect of the level.</td>
</tr>
<tr>
<td>AlkaliAtom.getC6term(n, l, j, n1, l1, j1, ...)</td>
<td>C6 interaction term for the given two pair-states</td>
</tr>
</tbody>
</table>

Continued on next page
Table 2 – continued from previous page

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlkaliAtom.getC3term(n, l, j, n1, l1, j1, ...)</td>
<td>C3 interaction term for the given two pair-states</td>
</tr>
<tr>
<td>AlkaliAtom.getEnergyDefect(n, l, j, n1, l1, j1, ...)</td>
<td>Energy defect for the given two pair-states (one of the state has two atoms in the same state)</td>
</tr>
<tr>
<td>AlkaliAtom.getEnergyDefect2(n, l, j, n1, l1, j1, ...)</td>
<td>Energy defect for the given two pair-states</td>
</tr>
<tr>
<td>AlkaliAtom.updateDipoleMatrixElementsFile()</td>
<td>Updates the file with pre-calculated dipole matrix elements.</td>
</tr>
<tr>
<td>AlkaliAtom.getRadialCoupling(n, l, j, n1, l1, j1)</td>
<td>Returns radial part of the coupling between two states (dipole and quadrupole interactions only)</td>
</tr>
<tr>
<td>AlkaliAtom.getAverageSpeed(temperature)</td>
<td>Average (mean) speed at a given temperature</td>
</tr>
<tr>
<td>AlkaliAtom.getLiteratureDME(n1, l1, j1, ...)</td>
<td>Returns literature information on requested transition.</td>
</tr>
</tbody>
</table>

Angular algebra in arc.wigner

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG(j1, m1, j2, m2, j3, m3)</td>
<td>Clebsch–Gordan (CG) coefficients</td>
</tr>
<tr>
<td>Wigner3j(j1, j2, j3, m1, m2, m3)</td>
<td>Evaluates Wigner 3-j symbol</td>
</tr>
<tr>
<td>Wigner6j(j1, j2, j3, J1, J2, J3)</td>
<td>Evaluates Wigner 6-j symbol</td>
</tr>
<tr>
<td>WignerDmatrix(theta, phi[, gamma])</td>
<td>WignerD matrices for different j states in a specified rotated basis.</td>
</tr>
</tbody>
</table>

Detailed documentation

Implements general single-atom calculations

This module calculates single (isolated) atom properties of all alkali metals in general. For example, it calculates dipole matrix elements, quadrupole matrix elements, etc. Also, some helpful general functions are here, e.g. for saving and loading calculations (single-atom and pair-state based), printing state labels etc.

class arc.alkali_atom_functions.AlkaliAtom(preferQuantumDefects=True, cpp_numerov=True)

Implements general calculations for alkali atoms.

This abstract class implements general calculations methods.

Parameters

- **preferQuantumDefects (bool)** – Use quantum defects for energy level calculations. If False, uses NIST ASD values where available. If True, uses quantum defects for energy calculations for principal quantum numbers equal or above minQuantumDefectN which is specified for each element separately. For principal quantum numbers below this value, NIST ASD values are used, since quantum defects don’t reproduce well low-lying states. Default is True.

- **cpp_numerov (bool)** – should the wavefunction be calculated with Numerov algorithm implemented in C++; if False, it uses pure Python implementation that is much slower. Default is True.

I = 0.0
Nuclear spin

Z = 0.0
Atomic number

a1 = [0]
Model potential parameters fitted from experimental observations for different l (electron angular momentum)

a2 = [0]
Model potential parameters fitted from experimental observations for different l (electron angular momentum)
a3 = [0]
Model potential parameters fitted from experimental observations for different l (electron angular momentum)

a4 = [0]
Model potential parameters fitted from experimental observations for different l (electron angular momentum)

abundance = 1.0
relative isotope abundance

alphaC = 0.0
Core polarizability

corePotential(l, r)
core potential felt by valence electron

For more details about derivation of model potential see Ref.\(^2\).

Parameters
• l (int) – orbital angular momentum
• r (float) – distance from the nucleus (in a.u.)

Returns core potential felt by valence electron (in a.u. ???)

Return type float

References

cpp_numerov = True
switch - should the wavefunction be calculated with Numerov algorithm implemented in C++

dipoleMatrixElementFile = ''
location of hard-disk stored dipole matrix elements

effectiveCharge(l, r)
effective charge of the core felt by valence electron

For more details about derivation of model potential see Ref.\(^2\).

Parameters
• l (int) – orbital angular momentum
• r (float) – distance from the nucleus (in a.u.)

Returns effective charge (in a.u.)

Return type float

elementName = 'elementName'
Human-readable element name

extraLevels = []
levels that are for smaller principal quantum number (n) than ground level, but are above in energy due to angular part

getAverageInteratomicSpacing(temperature)
Returns average interatomic spacing in atomic vapour

See calculation of basic properties example snippet.

Parameters temperature (float) – temperature of the atomic vapour

Returns average interatomic spacing in m

Return type float
getAverageSpeed(temperature)
Average (mean) speed at a given temperature

Parameters temperature (float) – temperature (K)

Returns mean speed (m/s)

Return type float
getC3term(n,l,j,n1,l1,j1,n2,l2,j2,s=0.5)
C3 interaction term for the given two pair-states

Calculates $C_3$ interaction term for $|n,l,j,n,l,j⟩ ↔ |n_1,l_1,j_1,n_2,l_2,j_2⟩$

Parameters

- n (int) – principal quantum number
- l (int) – orbital angular momentum
- j (float) – total angular momentum
- n1 (int) – principal quantum number
- l1 (int) – orbital angular momentum
- j1 (float) – total angular momentum
- n2 (int) – principal quantum number
- l2 (int) – orbital angular momentum
- j2 (float) – total angular momentum
- s (float) – optional, total spin angular momentum of state. By default 0.5 for Alkali atoms.

Returns $C_3 = \frac{(n,l,j)er[n_1,l_1,j_1](n,l,j)er[n_2,l_2,j_2]}{4\pi\varepsilon_0} (h \text{ Hz m }^3)$.

Return type float
getC6term(n,l,j,n1,l1,j1,n2,l2,j2,s=0.5)
C6 interaction term for the given two pair-states

Calculates $C_6$ interaction term for $|n,l,j,n,l,j⟩ ↔ |n_1,l_1,j_1,n_2,l_2,j_2⟩$. For details of calculation see Ref.3.

Parameters

- n (int) – principal quantum number
- l (int) – orbital angular momentum
- j (float) – total angular momentum
- n1 (int) – principal quantum number
- l1 (int) – orbital angular momentum
- j1 (float) – total angular momentum
- n2 (int) – principal quantum number
- l2 (int) – orbital angular momentum
- j2 (float) – total angular momentum
- s (float) – optional, total spin angular momentum of state. By default 0.5 for Alkali atoms.

---

Returns \[ C_6 = \frac{1}{4\pi\varepsilon_0} \left| \langle n_1, l_1, j_1 | e r | n_1, l_1, j_1 \rangle \right|^2 \left| \langle n_2, l_2, j_2 | e r | n_2, l_2, j_2 \rangle \right|^2 \left( E(n_1, l_1, j_1, n_2, l_2, j_2) - E(n_1, l_1, j_1, n_1, l_1, j_1) \right) \] (h Hz m 6).

Return type float

Example

We can reproduce values from Ref.3 for C3 coupling to particular channels. Taking for example channels described by the Eq. (50a-c) we can get the values:

```python
from arc import *

channels = [[70, 0, 0.5, 70, 1, 1.5, 69, 1, 1.5],
            [70, 0, 0.5, 70, 1, 1.5, 69, 1, 0.5],
            [70, 0, 0.5, 69, 1, 1.5, 70, 1, 0.5],
            [70, 0, 0.5, 70, 1, 0.5, 69, 1, 0.5]]

print(" = = = Caesium = = =")
atom = Caesium()
for channel in channels:
    print("%.0f GHz (mu m)^6" % (atom.getC6term(*channel) / C_h * 1.e27 ))

print(" = = = Rubidium = = =")
atom = Rubidium()
for channel in channels:
    print("%.0f GHz (mu m)^6" % (atom.getC6term(*channel) / C_h * 1.e27 ))
```

Returns:

```
= = = Caesium = = =
722 GHz (mu m)^6
316 GHz (mu m)^6
383 GHz (mu m)^6
228 GHz (mu m)^6

= = = Rubidium = = =
799 GHz (mu m)^6
543 GHz (mu m)^6
589 GHz (mu m)^6
437 GHz (mu m)^6
```

which is in good agreement with the values cited in the Ref.3. Small discrepancies for Caesium originate from slightly different quantum defects used in calculations.

References

getDipoleMatrixElement \( (n l, l l, j l, m j l, n 2, l 2, j 2, m j 2, q, s=0.5) \)
Dipole matrix element \( (n_1 l_1 j_1 m_{j_1} | e r | n_2 l_2 j_2 m_{j_2}) \) in units of \( a_0 e \)

Parameters

- \( l l, j l, m j l (n 1,) \) – principal, orbital, total angular momentum, and projection of total angular momentum for state 1
- \( l 2, j 2, m j 2 (n 2,) \) – principal, orbital, total angular momentum, and projection of total angular momentum for state 2
- \( q (i n t) \) – specifies transition that the driving field couples to, +1, 0 or -1 corresponding to driving \( \sigma^+ \), \( \pi \) and \( \sigma^- \) transitions respectively.
• \( s (\text{float}) \) – optional, total spin angular momentum of state. By default 0.5 for Alkali atoms.

**Returns** dipole matrix element( \( a_0e \))

**Return type** float

### Example

For example, calculation of \( 5S_{1/2} m_j = - \frac{1}{2} \rightarrow 5P_{3/2} m_j = - \frac{3}{2} \) transition dipole matrix element for laser driving \( \sigma^- \) transition:

```python
from arc import *
atom = Rubidium()
# transition 5 S_{1/2} m_j=-0.5 \rightarrow 5 P_{3/2} m_j=-1.5
# for laser driving sigma- transition
print(atom.getDipoleMatrixElement(5, 0, 0.5, -0.5, 5, 1, 1.5, -1.5, -1))
```

### `getDipoleMatrixElementHFS` (\( n_1, l_1, j_1, f_1, m_f_1, n_2, l_2, j_2, f_2, m_f_2, q, s=0.5 \))

Dipole matrix element for hyperfine structure resolved transitions \( \langle n_1 l_1 j_1 f_1 m_f_1 | e r | n_2 l_2 j_2 f_2 m_f_2 \rangle \) in units of \( a_0 e \)

For hyperfine resolved transitions, the dipole matrix element is

\[
\langle n_1 l_1 j_1 f_1 m_f_1 | r | n_2 l_2 j_2 f_2 m_f_2 \rangle = (-1)^{j_1+m_f_1} \left( \begin{array}{cc} f_1 & 1 \\ -m_f_1 & q \end{array} \right) \langle n_1 l_1 j_1 f_1 | r || n_2 l_2 j_2 f_2 \rangle,
\]

where

\[
\langle n_1 l_1 j_1 f_1 | r || n_2 l_2 j_2 f_2 \rangle = (-1)^{j_1+l+F_2+1} \sqrt{(2f_1+1)(2f_2+1)} \left\{ \begin{array}{cc} F_1 & 1 \\ F_2 & j_1 \end{array} \right\} \langle n_1 l_1 j_1 | r || n_2 l_2 j_2 \rangle.
\]

**Parameters**

- \( n_1, l_1, j_1, f_1, m_f_1 \) – principal, orbital, total orbital, fine basis (total atomic) angular momentum, and projection of total angular momentum for state 1
- \( n_2, l_2, j_2, f_2, m_f_2 \) – principal, orbital, total orbital, fine basis (total atomic) angular momentum, and projection of total angular momentum for state 2
- \( q (\text{int}) \) – specifies transition that the driving field couples to, +1, 0 or -1 corresponding to driving \( \sigma^+, \pi \) and \( \sigma^- \) transitions respectively.
- \( s (\text{float}) \) – optional, total spin angular momentum of state. By default 0.5 for Alkali atoms.

**Returns** dipole matrix element( \( a_0e \))

**Return type** float

### `getEnergy` (\( n, l, s, s=0.5 \))

Energy of the level relative to the ionisation level (in eV)

Returned energies are with respect to the center of gravity of the hyperfine-split states. If `preferQuantumDefects` =False (set during initialization) program will try use NIST energy value, if such exists, falling back to energy calculation with quantum defects if the measured value doesn’t exist. For `preferQuantumDefects` =True, program will always calculate energies from quantum defects (useful for comparing quantum defect calculations with measured energy level values).

**Parameters**

- \( n (\text{int}) \) – principal quantum number
- \( l (\text{int}) \) – orbital angular momentum
- \( j (\text{float}) \) – total angular momentum
- \( s (\text{float}) \) – optional, total spin angular momentum. Default value of 0.5 is correct for Alkali atoms, and has to be specified explicitly for divalent atoms.

**Returns** state energy (eV)
Return type  float

gEnergyDefect(n, l, j, n1, l1, j1, n2, l2, j2, s=0.5)
Energy defect for the given two pair-states (one of the state has two atoms in the same state)
Energy difference between the states $E(n_1, l_1, j_1, n_2, l_2, j_2) - E(n, l, j, n, l, j)\$

Parameters

• n (int) – principal quantum number
• l (int) – orbital angular momentum
• j (float) – total angular momentum
• n1 (int) – principal quantum number
• l1 (int) – orbital angular momentum
• j1 (float) – total angular momentum
• n2 (int) – principal quantum number
• l2 (int) – orbital angular momentum
• j2 (float) – total angular momentum
• s (float) – optional. Spin angular momentum (default 0.5 for Alkali)

Returns  energy defect (SI units: J)

Return type  float

gEnergyDefect2(n, l, j, nn, ll, jj, n1, l1, j1, n2, l2, j2, s=0.5)
Energy defect for the given two pair-states
Energy difference between the states $E(n_1, l_1, j_1, n_2, l_2, j_2) - E(n, l, j, nn, ll, jj)\$
See pair-state energy defects example snippet.

Parameters

• n (int) – principal quantum number
• l (int) – orbital angular momentum
• j (float) – total angular momentum
• nn (int) – principal quantum number
• ll (int) – orbital angular momentum
• jj (float) – total angular momentum
• n1 (int) – principal quantum number
• l1 (int) – orbital angular momentum
• j1 (float) – total angular momentum
• n2 (int) – principal quantum number
• l2 (int) – orbital angular momentum
• j2 (float) – total angular momentum
• s (float) – optional. Spin angular momentum (default 0.5 for Alkali)

Returns  energy defect (SI units: J)

Return type  float

gLiteratureDME(n1, l1, j1, n2, l2, j2, s=0.5)
Returns literature information on requested transition.

Parameters
• \( n_1, l_1, j_1 \) – one of the states we are coupling
• \( n_2, l_2, j_2 \) – the other state to which we are coupling

Returns

hasLiteratureValue?, dme, referenceInformation

If Boolean value is True, a literature value for dipole matrix element was found and reduced DME in J basis is returned as the number. The third returned argument (array) contains additional information about the literature value in the following order [typeOfSource, errorEstimate, comment, reference, reference DOI] upon success to find a literature value for dipole matrix element:

• typeOfSource=1 if the value is theoretical calculation; otherwise, if it is experimentally obtained value typeOfSource=0
• comment details where within the publication the value can be found
• errorEstimate is absolute error estimate
• reference is human-readable formatted reference
• reference DOI provides link to the publication.

Boolean value is False, followed by zero and an empty array if no literature value for dipole matrix element is found.

Return type boo, flo, [int, flo, str, str, str, str]

Note: The literature values are stored in /data folder in <element name>_literature_dme.csv files as a ; separated values. Each row in the file consists of one literature entry, that has information in the following order:

• n1
• l1
• j1
• n2
• l2
• j2
• dipole matrix element reduced l basis (a.u.)
• comment (e.g. where in the paper value appears?)
• value origin: 1 for theoretical; 0 for experimental values
• accuracy
• source (human readable formatted citation)
• doi number (e.g. 10.1103/RevModPhys.82.2313 )

If there are several values for a given transition, program outputs the value that has smallest error (under column accuracy). The list of values can be expanded - every time program runs this file is read and the list is parsed again for use in calculations.

\( \text{getNumberDensity}(\text{temperature}) \)

Atom number density at given temperature

See calculation of basic properties example snippet.

Parameters \( \text{temperature}(\text{float}) \) – temperature in K

Returns atom concentration in \( 1/m^3 \)
Return type float

generatePressure(temperature)
Vapour pressure (in Pa) at given temperature

Parameters temperature (float) – temperature in K

Returns vapour pressure in Pa

Return type float

generateQuadrupoleMatrixElement(n1, l1, j1, n2, l2, j2, s=0.5)
Radial part of the quadrupole matrix element

Calculates \( \int dr R_{n_1,l_1,j_1}(r) \cdot R_{n_1,l_1,j_1}(r) \cdot r^4 \). See Quadrupole calculation example snippet.

Parameters
• n1 (int) – principal quantum number of state 1
• l1 (int) – orbital angular momentum of state 1
• j1 (float) – total angular momentum of state 1
• n2 (int) – principal quantum number of state 2
• l2 (int) – orbital angular momentum of state 2
• j2 (float) – total angular momentum of state 2
• s (float) – optional, total spin angular momentum of state. By default 0.5 for Alkali atoms.

Returns quadrupole matrix element \( (a_2^0e) \).

Return type float

generateQuantumDefect(n, l, j, s=0.5)
Quantum defect of the level.

For an example, see Rydberg energy levels example snippet.

Parameters
• n (int) – principal quantum number
• l (int) – orbital angular momentum
• j (float) – total angular momentum
• s (float) – (optional). Total spin angular momentum. Default value of 0.5 correct for Alkali atoms. For divalent atoms it has to be explicitly defined.

Returns quantum defect

Return type float

generateRabiFrequency(n1, l1, j1, m1j1, n2, l2, j2, q, laserPower, laserWaist, s=0.5)
Returns a Rabi frequency for resonantly driven atom in a center of TEM00 mode of a driving field

Parameters
• n1, l1, j1, m1j1 – state from which we are driving transition
• n2, l2, j2 – state to which we are driving transition
• q – laser polarization (-1,0,1 correspond to \( \sigma^- \), \( \pi \) and \( \sigma^+ \) respectively)
• laserPower – laser power in units of W
• laserWaist – laser \( 1/e^2 \) waist (radius) in units of m
• s (float) – optional, total spin angular momentum of state. By default 0.5 for Alkali atoms.
Returns Frequency in rad $^{-1}$. If you want frequency in Hz, divide by returned value by $2\pi$

Return type float

getRabiFrequency2 ($n_1$, $l_1$, $j_1$, $m_{j1}$, $n_2$, $l_2$, $j_2$, $q$, electricFieldAmplitude, $s=0.5$)
Returns a Rabi frequency for resonant excitation with a given electric field amplitude

Parameters
• $n_1$, $l_1$, $j_1$, $m_{j1}$ – state from which we are driving transition
• $n_2$, $l_2$, $j_2$ – state to which we are driving transition
• $q$ – laser polarization (-1,0,1 correspond to $\sigma^-$, $\pi$ and $\sigma^+$ respectively)
• electricFieldAmplitude – amplitude of electric field driving (V/m)
• $s$ (float) – optional, total spin angular momentum of state. By default 0.5 for Alkali atoms.

Returns Frequency in rad $^{-1}$. If you want frequency in Hz, divide by returned value by $2\pi$

Return type float

getRadialCoupling ($n$, $l$, $j$, $n_1$, $l_1$, $j_1$, $s=0.5$)
Returns radial part of the coupling between two states (dipole and quadrupole interactions only)

Parameters
• $n_1$ (int) – principal quantum number of state 1
• $l_1$ (int) – orbital angular momentum of state 1
• $j_1$ (float) – total angular momentum of state 1
• $n_2$ (int) – principal quantum number of state 2
• $l_2$ (int) – orbital angular momentum of state 2
• $j_2$ (float) – total angular momentum of state 2
• $s$ (float) – optional, total spin angular momentum of state. By default 0.5 for Alkali atoms.

Returns radial coupling strength (in a.u.), or zero for forbidden transitions in dipole and quadrupole approximation.

Return type float

getRadialMatrixElement ($n_1$, $l_1$, $j_1$, $n_2$, $l_2$, $j_2$, $s=0.5$, useLiterature=True)
Radial part of the dipole matrix element
Calculates $\int dr \, R_{n_1,l_1,j_1}(r) \cdot R_{n_2,l_2,j_2}(r) \cdot r^3$.

Parameters
• $n_1$ (int) – principal quantum number of state 1
• $l_1$ (int) – orbital angular momentum of state 1
• $j_1$ (float) – total angular momentum of state 1
• $n_2$ (int) – principal quantum number of state 2
• $l_2$ (int) – orbital angular momentum of state 2
• $j_2$ (float) – total angular momentum of state 2
• $s$ (float) – optional, total spin angular momentum of state 1. By default 0.5 for Alkali atoms.

Returns dipole matrix element ($a_0e$).

Return type float
getReducedMatrixElementJ\((n_1, l_1, j_1, n_2, l_2, j_2, s=0.5)\)
Reduced matrix element in \(J\) basis (symmetric notation)

**Parameters**
- \(n_1\) (int) – principal quantum number of state 1
- \(l_1\) (int) – orbital angular momentum of state 1
- \(j_1\) (float) – total angular momentum of state 1
- \(n_2\) (int) – principal quantum number of state 2
- \(l_2\) (int) – orbital angular momentum of state 2
- \(j_2\) (float) – total angular momentum of state 2
- \(s\) (float) – optional, total spin angular momentum of state. By default 0.5 for Alkali atoms.

**Returns** reduced dipole matrix element in \(J\) basis \(\langle j||\mathcal{E}r||j'\rangle (a_0e)\).
**Return type** float

getReducedMatrixElementJ_asymmetric\((n_1, l_1, j_1, n_2, l_2, j_2, s=0.5)\)
Reduced matrix element in \(J\) basis, defined in asymmetric notation.

Note that notation for symmetric and asymmetrically defined reduced matrix element is not consistent in the literature. For example, notation is used e.g. in Steck\(^1\) precisely the opposite.

**Note:** Note that this notation is asymmetric: \(\langle j||\mathcal{E}r||j'\rangle \neq \langle j'||\mathcal{E}r||j\rangle\). Relation between the two notation is \(\langle j||\mathcal{E}r||j'\rangle = \sqrt{2J+1}\langle j'||\mathcal{E}r||j'\rangle\). This function always returns value for transition from lower to higher energy state, independent of the order of states entered in the function call.

**Parameters**
- \(n_1\) (int) – principal quantum number of state 1
- \(l_1\) (int) – orbital angular momentum of state 1
- \(j_1\) (float) – total angular momentum of state 1
- \(n_2\) (int) – principal quantum number of state 2
- \(l_2\) (int) – orbital angular momentum of state 2
- \(j_2\) (float) – total angular momentum of state 2
- \(s\) (float) – optional, total spin angular momentum of state. By default 0.5 for Alkali atoms.

**Returns** reduced dipole matrix element in Steck notation \(\langle j||\mathcal{E}r||j'\rangle (a_0e)\).
**Return type** float

getReducedMatrixElementL\((n_1, l_1, j_1, n_2, l_2, j_2, s=0.5)\)
Reduced matrix element in \(L\) basis (symmetric notation)

**Parameters**
- \(n_1\) (int) – principal quantum number of state 1
- \(l_1\) (int) – orbital angular momentum of state 1
- \(j_1\) (float) – total angular momentum of state 1
- \(n_2\) (int) – principal quantum number of state 2
- \(l_2\) (int) – orbital angular momentum of state 2

---

• $j_2$ (float) – total angular momentum of state 2

**Returns** reduced dipole matrix element in $L$ basis $\langle l || e r || l' \rangle (\alpha_0 e)$.

**Return type** float

### getStateLifetime ($n, l, j, temperature=0, includeLevelsUpTo=0, s=0.5$)

Returns the lifetime of the state (in s)

For non-zero temperatures, user must specify up to which principal quantum number levels, that is above the initial state, should be included in order to account for black-body induced transitions to higher lying states. See Rydberg lifetimes example snippet.

**Parameters**

• $l, j (n, l)$ – specifies state whose lifetime we are calculating

• temperature – optional. Temperature at which the atom environment is, measured in K. If this parameter is non-zero, user has to specify transitions up to which state (due to black-body decay) should be included in calculation.

• includeLevelsUpTo (int) – optional and not needed for atom lifetimes calculated at zero temperature. At non zero temperatures, this specify maximum principal quantum number of the state to which black-body induced transitions will be included. Minimal value of the parameter in that case is $n + 1$

• $s$ (float) – optional, total spin angular momentum of state. By default 0.5 for Alkali atoms.

**Returns** State lifetime in units of s (seconds)

**Return type** float

See also:

getTransitionRate for calculating rates of individual transition rates between the two states

### getTransitionFrequency ($n_1, l_1, j_1, n_2, l_2, j_2, s_1=0.5, s_2=0.5$)

Calculated transition frequency in Hz

Returned values is given relative to the centre of gravity of the hyperfine-split states.

**Parameters**

• $n_1$ (int) – principal quantum number of the state from which we are going

• $l_1$ (int) – orbital angular momentum of the state from which we are going

• $j_1$ (float) – total angular momentum of the state from which we are going

• $n_2$ (int) – principal quantum number of the state to which we are going

• $l_2$ (int) – orbital angular momentum of the state to which we are going

• $j_2$ (float) – total angular momentum of the state to which we are going

• $s_1$ (float) – optional, spin of the initial state (for Alkali this is fixed to 0.5)

• $s_2$ (float) – optional, spin of the final state (for Alkali this is fixed to 0.5)

**Returns** transition frequency (in Hz). If the returned value is negative, level from which we are going is above the level to which we are going.

**Return type** float

### getTransitionRate ($n_1, l_1, j_1, n_2, l_2, j_2, temperature=0.0, s=0.5$)

Transition rate due to coupling to vacuum modes (black body included)

Calculates transition rate from the first given state to the second given state $|n_1, l_1, j_1\rangle \rightarrow |n_2, l_2, j_2\rangle$ at given temperature due to interaction with the vacuum field. For zero temperature this returns Einstein
A coefficient. For details of calculation see Ref.\textsuperscript{4} and Ref.\textsuperscript{5}. See Black-body induced population transfer example snippet.

**Parameters**

- \( n1 \) (int) – principal quantum number
- \( l1 \) (int) – orbital angular momentum
- \( j1 \) (float) – total angular momentum
- \( n2 \) (int) – principal quantum number
- \( l2 \) (int) – orbital angular momentum
- \( j2 \) (float) – total angular momentum
- \([temperature]\) (float) – temperature in K
- \( s \) (float) – optional, total spin angular momentum of state. By default 0.5 for Alkali atoms.

**Returns** transition rate in s\(^{-1}\) (SI)

**Return type** float

**References**

getTransitionWavelength \((n1, l1, j1, n2, l2, j2, s1=0.5, s2=0.5)\)

Calculated transition wavelength (in vacuum) in m.

Returned values is given relative to the centre of gravity of the hyperfine-split states.

**Parameters**

- \( n1 \) (int) – principal quantum number of the state from which we are going
- \( l1 \) (int) – orbital angular momentum of the state from which we are going
- \( j1 \) (float) – total angular momentum of the state from which we are going
- \( n2 \) (int) – principal quantum number of the state to which we are going
- \( l2 \) (int) – orbital angular momentum of the state to which we are going
- \( j2 \) (float) – total angular momentum of the state to which we are going
- \( s1 \) (float) – optional, spin of the initial state (for Alkali this is fixed to 0.5)
- \( s2 \) (float) – optional, spin of the final state (for Alkali this is fixed to 0.5)

**Returns** transition wavelength (in m). If the returned value is negative, level from which we are going is above the level to which we are going.

**Return type** float

getZeemanEnergyShift \((l, j, mj, magneticFieldBz, s=0.5)\)

Returns linear (paramagnetic) Zeeman shift.

\[ \mathcal{H}_P = \frac{\mu_B B_z}{h} (\hat{L}_z + g_S S_z) \]

**Parameters**

- \( l \) (int) – orbital angular momentum
- \( j \) (float) – total angular momentum
- \( mj \) (float) – projection of total angular momentum along z-axis

• `magneticFieldBz(float)` – applied magnetic field (along the z-axis only) in units of T (Tesla)

• `s(float)` – optional, total spin angular momentum of state. By default 0.5 for Alkali atoms.

**Returns** energy offset of the state (in J)

**Return type** float

`groundStateN = 0`
principal quantum number for the ground state

`levelDataFromNIST = ''`
location of stored NIST values of measured energy levels in eV

`literatureDMEfilename = ''`
Filename of the additional literature source values of dipole matrix elements. These additional values should be saved as reduced dipole matrix elements in J basis.

`mass = 0.0`
atomic mass in kg

`meltingPoint = 0`
melting point of the element at standard conditions

`minQuantumDefectN = 0`
mimal quantum number for which quantum defects can be used; uses measured energy levels otherwise

`potential(l, s, j, r)`
returns total potential that electron feels

Total potential = core potential + Spin-Orbit interaction

**Parameters**

• `l(int)` – orbital angular momentum

• `s(float)` – spin angular momentum

• `j(float)` – total angular momentum

• `r(float)` – distance from the nucleus (in a.u.)

**Returns** potential (in a.u.)

**Return type** float

`quadrupoleMatrixElementFile = ''`
location of hard-disk stored dipole matrix elements

`quantumDefect = [[0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]]`
Contains list of modified Rydberg-Ritz coefficients for calculating quantum defects for [[S_{1/2}, P_{1/2}, D_{3/2}, F_{5/2}], [S_{1/2}, P_{3/2}, D_{5/2}, F_{7/2}]].

`radialWavefunction(l, s, j, stateEnergy, innerLimit, outerLimit, step)`
Radial part of electron wavefunction

Calculates radial function with Numerov (from outside towards the core). Note that wavefunction might not be calculated all the way to the requested `innerLimit` if the divergence occurs before. In that case third returned argument gives nonzero value, corresponding to the first index in the array for which wavefunction was calculated. For quick example see Rydberg wavefunction calculation snippet.

**Parameters**

• `l(int)` – orbital angular momentum

• `s(float)` – spin angular momentum

• `j(float)` – total angular momentum
• **stateEnergy** (*float*) – state energy, relative to ionization threshold, should be given in atomic units (Hartree)
• **innerLimit** (*float*) – inner limit at which wavefunction is requested
• **outerLimit** (*float*) – outer limit at which wavefunction is requested
• **step** (*float*) – radial step for integration mesh (a.u.)

**Returns**

\[
\begin{align*}
r & \quad R(r) \cdot r \\
\end{align*}
\]

**Return type** List[float], List[float], int

---

**Note:** Radial wavefunction is not scaled to unity! This normalization condition means that we are using spherical harmonics which are normalized such that

\[
\int d\theta d\phi \ Y(l, m_l)^* \times Y(l', m_{l'}) = \delta(l, l') \delta(m_l, m_{l'}).
\]

---

**Note:** Alternative calculation methods can be added here (potenital package expansion).

---

**rc = [0]**

Model potential parameters fitted from experimental observations for different \(l\) (electron angular momentum)

**sEnergy = 0**

state energies from NIST values \(sEnergy[n, l] = \) state energy for \(n, l, j = l-1/2\) \(sEnergy[l, n] = \) state energy for \(j = l+1/2\)

**updateDipoleMatrixElementsFile()**

Updates the file with pre-calculated dipole matrix elements.

This function will add the the file all the elements that have been calculated in the previous run, allowing quick access to them in the future calculations.

**arc.alkali_atom_functions.NumerovBack(innerLimit, outerLimit, kfun, step, init1, init2)**

Full Python implementation of Numerov integration

Calculates solution function \(rad(r)\) with discrete step in \(r\) size of \(step\), integrating from \(outerLimit\) towards the \(innerLimit\) (from outside, inwards) equation

\[
\frac{d^2 rad(r)}{dr^2} = kfun(r) \cdot rad(r).
\]

**Parameters**

• **innerLimit** (*float*) – inner limit of integration
• **outerLimit** (*float*) – outer limit of integration
• **kfun** (*function(double)*) – pointer to function used in equation (see longer explanation above)
• **step** – discrete step size for integration
• **init1** (*float*) – initial value, \(rad'(outerLimit' + 'step')\)
• **init2** (*float*) – initial value, \(rad'(outerLimit' + :math:'2cdot step')\)

**Returns** \(r\) (a.u.), \(rad(r)\);

**Return type** numpy array of float, numpy array of float, int

---

**Note:** Returned function is not normalized!
Note: If `AlkaliAtom.cpp_numerov switch` is set to True (default), much faster C implementation of the algorithm will be used instead. That is recommended option. See documentation installation instructions for more details.

```
arc.alkali_atom_functions.loadSavedCalculation(fileName)
```

Loads previously saved calculation.

```
Loads calculations_atom_pairstate.PairStateInteractions and calculations_atom_single.StarkMap calculation instance from file named filename where it was previously saved with saveCalculation.
```

**Example**

See example for `saveCalculation`.

**Parameters**

- `fileName` – name of the file where calculation will be saved

**Returns**

saved calculation

```
arc.alkali_atom_functions.printState(n,l,j,s=None)
```

Prints state spectroscopic label for numeric `n`, `l`, `s` label of the state

**Parameters**

- `n` (int) – principal quantum number
- `l` (int) – orbital angular momentum
- `j` (float) – total angular momentum
- `s` (float) – (optional) total spin momentum

```
arc.alkali_atom_functions.printStateString(n,l,j,s=None)
```

Returns state spectroscopic label for numeric `n`, `l`, `j` label of the state.

Optionally users can define `s`, prompting printing `2S + 1` index too (commonly used for Alkaline Earth atoms, while it is usually omitted for Alkali atoms)

**Parameters**

- `n` (int) – principal quantum number
- `l` (int) – orbital angular momentum
- `j` (float) – total angular momentum
- `s` (float) – (optional) total spin momentum

**Returns**

label for the state in standard spectroscopic notation

**Return type**

string

```
arc.alkali_atom_functions.printStateStringLatex(n,l,j,s=None)
```

Returns latex code for spectroscopic label for numeric `n`, `l`, `j` label of the state.

**Parameters**

- `n` (int) – principal quantum number
- `l` (int) – orbital angular momentum
- `j` (float) – total angular momentum
- `s` (float) – (optional) total spin momentum

**Returns**

label for the state in standard spectroscopic notation

**Return type**

string
arc.alkali_atom_functions.saveCalculation(calculation, fileName)

Saves calculation for future use.

Saves calculations_atom_pairstate.PairStateInteractions and calculations_atom_single.StarkMap calculations in compact binary format in file named filename. It uses cPickle serialization library in Python, and also zips the final file.

Calculation can be retrieved and used with loadSavedCalculation

Parameters

• calculation – class instance of calculations (instance of calculations_atom_pairstate.PairStateInteractions or calculations_atom_single.StarkMap) to be saved.

• fileName – name of the file where calculation will be saved

Example

Let’s suppose that we did the part of the calculation_atom_pairstate.PairStateInteractions calculation that involves generation of the interaction matrix. After that we can save the full calculation in a single file:

```python
calc = PairStateInteractions(Rubidium(),
  60,0,0.5,
  60,0,0.5,
  0.5,0.5)
calc.defineBasis(0,0, 5,5, 25.e9)
calc.diagonalise(np.linspace(0.5,10.0,200),150)
saveCalculation(calc, "mySavedCalculation.pkl")
```

Then, at a later time, and even on the another machine, we can load that file and continue with calculation. We can for example explore the calculated level diagram:

```python
calc = loadSavedCalculation("mySavedCalculation.pkl")
calc.plotLevelDiagram()
calc.showPlot()
rvdw = calc.getVdwFromLevelDiagram(0.5,14,
  minStateContribution=0.5,
  showPlot = True)
```

Or, we can do additional matrix diagonalization, in some new range, then and find C6 by fitting the obtained level diagram:

```python
calc = loadSavedCalculation("mySavedCalculation.pkl")
calc.diagonalise(np.linspace(3,6.0,200),20)
calc.getC6fromLevelDiagram(3,6.0,showPlot=True)
```

Note that for all loading of saved calculations we’ve been using function loadSavedCalculation.

Note: This doesn’t save results of plotLevelDiagram for the corresponding calculations. Call the plot function before calling showPlot function for the corresponding calculation.

arc.alkali_atom_functions.setup_data_folder()

Set up the data folder in the users home directory.

arc.wigner.CG(j1, m1, j2, m2, j3, m3)

Clebsch–Gordan (CG) coefficients

Parameters j1, m1, j2, m2, j3, m3 – parameters of \( \langle j_1, m_1, j_2, m_2 | j_3, m_3 \rangle \)
arc.wigner.Wigner3j(j1, j2, j3, m1, m2, m3)
Evaluates Wigner 3-j symbol

Parameters j1, j2, j3, m1, m2, m3 (float) – parameters of \( \begin{pmatrix} j_1 & j_2 & j_3 \\ m_1 & m_2 & m_3 \end{pmatrix} \)

arc.wigner.Wigner6j(j1, j2, j3, J1, J2, J3)
Evaluates Wigner 6-j symbol

Parameters j1, j2, j3, J1, J2, J3 (float) – parameters of \( \begin{pmatrix} j_1 & j_2 & j_3 \\ J_1 & J_2 & J_3 \end{pmatrix} \)

class arc.wigner.WignerDmatrix(theta, phi, gamma=0.0)
WignerD matrices for different j states in a specified rotated basis.

This matrix converts components of angular momentum j given in one basis into components of angular momentum calculated in the basis which is rotated by phi around z-axis, and then by theta around y-axis. Use:

```
wdg = WignerDmatrix(theta, phi)
# let's rotate state with angular momentum 1
dMatrix = wdg.get(j)
stateNewBasis = dMatrix.dot(stateOldBasis)
```

Parameters

- **theta (float)** – rotation around y-axis
- **phi (float)** – rotation around z-axis
- **gamma (float)** – optional, last rotation around z-axis (rotations are in order z-y-z). By default 0.

get(j)
WignerD matrix for specified basis for states with angular momentum j.

Parameters j (float) – angular momentum of states.

Returns matrix of dimensions (2*j+1,2*j+1). state in new basis = wignerDmatrix * state in original basis

### 1.3.2 Alkali atom data

| Hydrogen([preferQuantumDefects, cpp_numerov]) | Properties of hydrogen atoms |
| Lithium6([preferQuantumDefects, cpp_numerov]) | Properties of lithium 6 atoms |
| Lithium7([preferQuantumDefects, cpp_numerov]) | Properties of lithium 7 atoms |
| Sodium([preferQuantumDefects, cpp_numerov]) | Properties of sodium 23 atoms |
| Potassium([preferQuantumDefects, cpp_numerov]) | Properties of potassium 39 atoms |
| Potassium39([preferQuantumDefects, cpp_numerov]) | Properties of potassium 39 atoms |
| Potassium40([preferQuantumDefects, cpp_numerov]) | Properties of potassium 40 atoms |
| Potassium41([preferQuantumDefects, cpp_numerov]) | Properties of potassium 41 atoms |
| Rubidium([preferQuantumDefects, cpp_numerov]) | Properties of potassium 41 atoms |

backward compatibility: before only one class for Potassium existed and it corresponded to Potassium 39
backward compatibility: before there was only one Rubidium class, and that one corresponded to Rubidi-um85

Continued on next page
This module specifies properties of individual alkali metals.

If you want to change e.g. coefficients used for model potential, quantum defects, or other numerical values, this is the place to look at.

**How to delete precalculated dipole/quadrupole matrix elements values and/or start a new database?**

To delete precalculated values, simply delete files, whose names are stated in `dipoleMatrixElementFile`, `quadrupoleMatrixElementFile` and `precalculatedDB` variables for the corresponding atom type, from data/ folder. Alternatively, if you want to keep old values, but want to also start completely new calculation of dipole matrix elements (e.g. because you changed parameters of energy levels significantly or model potential parameters), simply set new values for `dipoleMatrixElementFile`, `quadrupoleMatrixElementFile` and `precalculatedDB` variables.

Note that by default isotopes of Rubidium and Potassium are sharing precalculated dipole and quadrupole matrix elements. This is because the small energy level differences typically don’t change this matrix elements within a typical accuracy.

**Data sources**

**Module**

```python
class arc.alkali_atom_data.Caesium(preferQuantumDefects=True, cpp_numerov=True)

Bases: arc.alkali_atom_functions.AlkaliAtom

Properties of caesium atoms

a1 = [3.49546309, 4.69366096, 4.32466196, 3.01048361]
model potential parameters from\(^1\)

a2 = [1.475338, 1.71398344, 1.61365288, 1.40000001]
model potential parameters from\(^1\)

a3 = [-9.72143084, -24.6562428, -6.7012885, -3.20036138]
model potential parameters from\(^1\)

a4 = [0.02629242, -0.09543125, -0.74095193, 0.00034538]
model potential parameters from\(^1\)

alphaC = 15.644
model potential parameters from\(^1\)

extraLevels = [[5, 2, 2.5], [5, 2, 1.5], [5, 3, 3.5], [5, 3, 2.5], [5, 4, 4.5], [5, 4, 3.5], [4, 3, 3.5], [4, 3, 2.5]]
levels that are for smaller n than ground level, but are above in energy due to angular part

getPressure(temperature)
Pressure of atomic vapour at given temperature.

Uses equation and values from\(^3\). Values from table 2. (accuracy +/- 5\%) are used for Cs in solid phase. Values from table 3. (accuracy +/- 1\%) are used for Cs in liquid phase.

ionisationEnergy = 3.893905726389478
(eV), Ref.\(^9\).
```

---


meltingPoint = 301.59
in K

quantumDefect = [[[4.04935665, 0.2377037, 0.255401, 0.00378, 0.25486, 0.0],
                  [3.5915895, 0.360926, 0.41905, 0.64388, ... -2.1867, -1.5532, -56.6739]],
                  [0.03341424, -0.198674, 0.28953, -0.2601, 0.0, 0.0],
                  [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]]

quantum defects for $S_{1/2}$, $nP_{1/2}$, $D_{5/2}$, $F_{5/2}$ and $G_{7/2}$ are from\(^2\), while quantum defects for $nP_{3/2}$:math:D_{3/2} are from\(^10\).

**Note:** f$_{7/2}$ quantum defects are PUT TO BE EXACTLY the same as f$_{5/2}$ (~10MHz difference??)

rc = [1.9204693, 2.13383095, 0.93007296, 1.99969677]
model potential parameters from\(^1\)

scaledRydbergConstant = 13.605636960556954
in eV

class arc.alkali_atom_data.Cesium(preferQuantumDefects=True, cpp_numerov=True)
Bases: arc.alkali_atom_data.Caesium

supports for American English spelling

class arc.alkali_atom_data.Hydrogen(preferQuantumDefects=True, cpp_numerov=True)
Bases: arc.alkali_atom_functions.AlkaliAtom

Properties of hydrogen atoms

ionisationEnergy = 13.598433
(eV), Ref.\(^{12}\).

mass = 1.6735328383153192e-27
source NIST, Atomic Weights and Isotopic Compositions\(^{14}\)

potential $(l, s, j, r)$
returns total potential that electron feels
Total potential = core potential + Spin-Orbit interaction

Parameters

- $l$ (int) – orbital angular momentum
- $s$ (float) – spin angular momentum
- $j$ (float) – total angular momentum
- $r$ (float) – distance from the nucleus (in a.u.)

Returns potential (in a.u.)

Return type float

class arc.alkali_atom_data.Lithium6(preferQuantumDefects=True, cpp_numerov=True)
Bases: arc.alkali_atom_functions.AlkaliAtom

Properties of lithium 6 atoms

a1 = [2.47718079, 3.45414648, 2.51909839, 2.51909839]
model potential parameters from\(^1\)

a2 = [1.84150932, 2.5515108, 2.4371245, 2.4371245]
model potential parameters from\(^1\)

a3 = [−0.02169712, −0.21646561, 0.32505524, 0.32505524]
model potential parameters from\(^1\)

---


\(^{12}\) NIST, P. Mohr and S. Kotochigova, unpublished calculations (2000). The wavelengths for the Balmer-alpha and Balmer-beta transitions at 6563 and 4861 \(\times c/\delta\) include only the stronger components of more extensive fine structures.


1.3. Detailed documentation of functions
\[ a_4 = [-0.11988362, -0.06990078, 0.1060243, 0.1060243] \]
model potential parameters from\(^1\)

\[ \text{abundance} = 0.0759 \]
source NIST, Atomic Weights and Isotopic Compositions\(^{14}\)

\[ \text{alphaC} = 0.1923 \]
model potential parameters from\(^1\)

\[ \text{getPressure}(\text{temperature}) \]
Pressure of atomic vapour at given temperature.

Uses equation and values from\(^3\). Values from table 3. (accuracy +/- 1 %) are used both for liquid and solid phase of Li.

\[ \text{mass} = 9.98346544927492\times10^{-27} \]
source NIST, Atomic Weights and Isotopic Compositions\(^{14}\)

\[ \text{meltingPoint} = 453.68999999999994 \]
in K

\[ \text{quantumDefect} = \begin{bmatrix} 0.3995101, 0.029, 0.0, 0.0, 0.0, 0.0 \end{bmatrix}, \begin{bmatrix} 0.0471835, -0.024, 0.0, 0.0, 0.0, 0.0 \end{bmatrix}, \begin{bmatrix} 0.002129, 0.000000, -0.000000, 0.000000, 0.000000, 0.000000 \end{bmatrix}, \begin{bmatrix} -7.7e-05, 0.021856, -0.4211, 2.3891, 0.0, 0.0 \end{bmatrix}, \begin{bmatrix} 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 \end{bmatrix} \]
quantum defects for \( nS \) and \( nP \) are from Ref.\(^8\). Quantum defects for \( D_j \) and \( F_j \) are from Ref.\(^{11}\) (note that this defects in Ref.\(^{11}\) are for Li\(^7\), differences are expected not be too big).

\[ \text{rc} = [0.61340824, 0.61566441, 2.34126273, 2.34126273] \]
model potential parameters from\(^1\)

class arc.alkali_atom_data.Lithium7 (preferQuantumDefects=True, cpp_numerov=True)
Bases: arc.alkali_atom_functions.AlkaliAtom

Properties of lithium 7 atoms

\[ a_1 = [2.47718079, 3.45414648, 2.51909839, 2.51909839] \]
model potential parameters from\(^1\)

\[ a_2 = [1.84150932, 2.5515108, 2.4371245, 2.4371245] \]
model potential parameters from\(^1\)

\[ a_3 = [-0.02169712, -0.21646561, 0.32505524, 0.32505524] \]
model potential parameters from\(^1\)

\[ a_4 = [-0.11988362, -0.06990078, 0.1060243, 0.1060243] \]
model potential parameters from\(^1\)

\[ \text{abundance} = 0.9241 \]
source NIST, Atomic Weights and Isotopic Compositions\(^{14}\)

\[ \text{alphaC} = 0.1923 \]
model potential parameters from\(^1\)

\[ \text{getPressure}(\text{temperature}) \]
Pressure of atomic vapour at given temperature (in K).

Uses equation and values from\(^3\). Values from table 3. (accuracy +/- 1 %) are used for both liquid and solid phase of Li.

\[ \text{ionisationEnergy} = 5.391719 \] (eV) NIST Ref.\(^{13}\)

\[ \text{mass} = 1.1650347797874157\times10^{-26} \]
source NIST, Atomic Weights and Isotopic Compositions\(^{14}\)

---


meltingPoint = 453.68999999999994
in K
quantumDefect = [[[0.3995101, 0.0, 0.0, 0.0, 0.0, 0.0], [0.047178, -0.024, 0.0, 0.0, 0.0, 0.0], [0.002129, 0.0, 0.0, 0.0, -0.1759, -0.8507, 0.0, 0.0], [-7.7e-05, 0.021856, -0.4211, 2.3891, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]]
quantum defects for nS and nP states are from Ref.8. Quantum defects for D_{j} and F_{j} states are from11.
rc = [0.61340824, 0.61566441, 2.34126273, 2.34126273]
model potential parameters from1

class arc.alkali_atom_data.Potassium (preferQuantumDefects=True, cpp_numerov=True)
Bases: arc.alkali_atom_data.Potassium39
backward compatibility: before only one class for Potassium existed and it corresponded to Potassium 39

class arc.alkali_atom_data.Potassium39 (preferQuantumDefects=True, cpp_numerov=True)
Bases: arc.alkali_atom_functions.AlkaliAtom
Properties of potassium 39 atoms
a1 = [3.56079437, 3.65670429, 4.12713694, 1.42310446]
model potential parameters from1
a2 = [1.83909642, 1.67520788, 1.79837462, 1.27861156]
model potential parameters from1
a3 = [-1.74701102, -2.07416615, -1.69935174, 4.77441476]
model potential parameters from1
a4 = [-1.03237313, -0.89030421, -0.98913582, -0.94829262]
model potential parameters from1
abundance = 0.932581
source NIST, Atomic Weights and Isotopic Compositions14
alphaC = 5.331
model potential parameters from1
extraLevels = [[3, 2, 2.5], [3, 2, 1.5]]
levels that are for smaller n than ground level, but are above in energy due to angular part

getPressure (temperature)
Pressure of atomic vapour at given temperature.
Uses equation and values from3. Values from table 2. (accuracy +- 5%) are used for Na in solid phase. Values from table 3. (accuracy +-1 %) are used for Na in liquid phase.
ionisationEnergy = 4.340663717174086 (eV), weighted average of values in Ref.11.
mass = 6.470075680020302e-26
source NIST, Atomic Weights and Isotopic Compositions14
meltingPoint = 336.65
in K
quantumDefect = [[[2.1801985, 0.13558, 0.0759, 0.117, -0.206, 0.0], [1.713892, 0.233294, 0.16137, 0.5345, -0.234, 2.3891], [0.010098, -0.100224, 1.56334, -12.6851, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]]
quantum defects from Ref.11.
rc = [0.83167545, 0.85235381, 0.83216907, 6.50294371]
model potential parameters from1

class arc.alkali_atom_data.Potassium40 (preferQuantumDefects=True, cpp_numerov=True)
Bases: arc.alkali_atom_functions.AlkaliAtom
Properties of potassium 40 atoms
a1 = [3.56079437, 3.65670429, 4.12713694, 1.42310446]
model potential parameters from

a2 = [1.83909642, 1.67520788, 1.79837462, 1.27861156]
model potential parameters from

a3 = [-1.74701102, -2.07416615, -1.69935174, 4.77441476]
model potential parameters from

a4 = [-1.03237313, -0.89030421, -0.98913582, -0.94829262]
model potential parameters from

abundance = 0.000117
source NIST, Atomic Weights and Isotopic Compositions

alphaC = 5.331
model potential parameters from

eXtraLevels = [[3, 2, 2.5], [3, 2, 1.5]]
levels that are for smaller n than ground level, but are above in energy due to angular part

getPressure (temperature)
Pressure of atomic vapour at given temperature.
Uses equation and values from. Values from table 2. (accuracy +- 5%) are used for Na in solid phase. Values from table 3. (accuracy +-1 %) are used for Na in liquid phase.

ionisationEnergy = 4.340663717174086
(eV), weighted average of values in Ref.11.

mass = 6.636178021217375e-26
source NIST, Atomic Weights and Isotopic Compositions

meltingPoint = 336.65
in K

quantumDefect = [[[2.1801985, 0.13558, 0.0759, 0.117, -0.206, 0.0], [1.713892, 0.233294, 0.16137, 0.5345, -0.234, 0.0]], [10.0053, -19.0244, 0.0], [0.010098, -0.100224, 1.56334, -12.6851, 0.0, 0.0]], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]]
quantum defects from Ref.11.

rc = [0.83167545, 0.85235381, 0.83216907, 6.50294371]
model potential parameters from

scaledRydbergConstant = 13.605506356008
in eV

class arc.alkali_atom_data.Potassium41 (preferQuantumDefects=True, cpp_numerov=True)

Bases: arc.alkali_atom_functions.AlkaliAtom

Properties of potassium 41 atoms

a1 = [3.56079437, 3.65670429, 4.12713694, 1.42310446]
model potential parameters from

a2 = [1.83909642, 1.67520788, 1.79837462, 1.27861156]
model potential parameters from

a3 = [-1.74701102, -2.07416615, -1.69935174, 4.77441476]
model potential parameters from

a4 = [-1.03237313, -0.89030421, -0.98913582, -0.94829262]
model potential parameters from

abundance = 0.067302
source NIST, Atomic Weights and Isotopic Compositions

alphaC = 5.331
model potential parameters from
extraLevels = [[3, 2, 2.5], [3, 2, 1.5]]
levels that are for smaller n than ground level, but are above in energy due to angular part

generatePressure(temperature)
Pressure of atomic vapour at given temperature.
Uses equation and values from\textsuperscript{3}. Values from table 2. (accuracy +\textpm 5\%) are used for Na in solid phase.
Values from table 3. (accuracy +\textpm 1\%) are used for Na in liquid phase.

ionisationEnergy = 4.340663717174086
(eV), weighted average of values in Ref.\textsuperscript{11}.
mass = 6.801871107998557e-26
source NIST, Atomic Weights and Isotopic Compositions\textsuperscript{14}
meltingPoint = 336.65
in K
quantumDefect = [[2.1801985, 0.13558, 0.0759, 0.117, -0.206, 0.0], [1.713892, 0.233294, 0.16137, 0.5345, -0.234, 10.0053, -19.0244, 0.0], [0.010098, -0.100224, 1.56334, -12.6851, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]]
quantum defects from Ref.\textsuperscript{11}.
rc = [0.83167545, 0.85235381, 0.83216907, 6.50294371]
model potential parameters from\textsuperscript{1}
scaledRydbergConstant = 13.60510905549485
in eV
class arc.alkali_atom_data.Rubidium(preferQuantumDefects=True, cpp_numerov=True)
Bases: arc.alkali_atom_data.Rubidium85
backward compatibility: before there was only one Rubidium class, and that one corresponded to Rubidium85

class arc.alkali_atom_data.Rubidium85(preferQuantumDefects=True, cpp_numerov=True)
Bases: arc.alkali_atom_functions.AlkaliAtom

Properties of rubidium 85 atoms
a1 = [3.69628474, 4.44088978, 3.78717363, 2.39848933]
model potential parameters from\textsuperscript{1}
a2 = [1.64915255, 1.92828831, 1.57027864, 1.76810544]
model potential parameters from\textsuperscript{1}
a3 = [-9.86069196, -16.7959777, -11.6558897, -12.0710678]
model potential parameters from\textsuperscript{1}
a4 = [0.19579987, -0.8163314, 0.52942835, 0.77256589]
model potential parameters from\textsuperscript{1}
abundance = 0.7217
source NIST, Atomic Weights and Isotopic Compositions\textsuperscript{14}
alphaC = 9.076
model potential parameters from\textsuperscript{1}
extralLevels = [[4, 2, 2.5], [4, 2, 1.5], [4, 3, 3.5], [4, 3, 2.5]]
levels that are for smaller n than ground level, but are above in energy due to angular part

generatePressure(temperature)
Pressure of atomic vapour at given temperature.
Uses equation and values from\textsuperscript{3}. Values from table 2. (accuracy +\textpm 5\%) are used for Rb in solid phase.
Values from table 3. (accuracy +\textpm 1\%) are used for Rb in liquid phase.
ionisationEnergy = 4.17712652376669 (eV) Ref.15

mass = 1.409993440747079e-25

source NIST, Atomic Weights and Isotopic Compositions14

meltingPoint = 312.46 in K

quantumDefect = [[3.1311804, 0.1784, 0.0, 0.0, 0.0, 0.0],
                 [2.6548849, 0.29, 0.0, 0.0, 0.0, 0.0],
                 [1.34809171, -0.596, 0.0, 0.0, 0.0, 0.0],
                 [0.0165437, -0.086, 0.0, 0.0, 0.0, 0.0],
                 [0.00405, 0.0, 0.0, 0.0, 0.0, 0.0]]

quantum defects for \( n_F \) states are from5. Quantum defects for \( n_G \) states are from7. All other quantum defects are from from4

rc = [1.66242117, 1.50195124, 4.86851938, 4.79831327]

model potential parameters from1

scaledRydbergConstant = 13.605605218602195 in eV

class arc.alkali_atom_data.Rubidium87 (preferQuantumDefects=True, cpp_numerov=True)

Bases: arc.alkali_atom_functions.AlkaliAtom

Properties of rubidium 87 atoms

a1 = [3.69628474, 4.40488978, 3.78717363, 2.39848933]
model potential parameters from1

a2 = [1.64915255, 1.92828831, 1.57027864, 1.76810544]
model potential parameters from1

a3 = [-9.86069196, -16.7959777, -11.6558897, -12.0710678]
model potential parameters from1

a4 = [0.19579987, -0.8163314, 0.52942835, 0.77256589]
model potential parameters from1

abundance = 0.2783

source NIST, Atomic Weights and Isotopic Compositions14

alphaC = 9.076
model potential parameters from1

extraLevels = [[4, 2, 2.5], [4, 2, 1.5], [4, 3, 3.5], [4, 3, 2.5]]
levels that are for smaller \( n \) than ground level, but are above in energy due to angular part

getPressure (temperature)
Pressure of atomic vapour at given temperature.

Uses equation and values from3. Values from table 2. (accuracy \( \pm 5\% \)) are used for Rb in solid phase. Values from table 3. (accuracy \( \pm 1\% \)) are used for Rb in liquid phase.

ionisationEnergy = 4.177127321633609 (eV) Ref.5

mass = 1.4431608951791763e-25

source NIST, Atomic Weights and Isotopic Compositions14

meltingPoint = 312.46 in K
quantumDefect = 
[[3.1311804, 0.1784, 0.0, 0.0, 0.0, 0.0],
 [2.6548849, 0.29, 0.0, 0.0, 0.0, 0.0],
 [1.34809171, 0.0, 0.0, 0.0, 0.0, 0.0],
 [0.0165437, -0.086, 0.0, 0.0, 0.0, 0.0],
 [0.00405, 0.0, 0.0, 0.0, 0.0, 0.0]]

quantum defects for \(nF\) states are from \(^5\). Quantum defects for \(nG\) states are from \(^1\). All other quantum defects are from from \(^4\).

\[\text{rc} = [1.66242117, 1.50195124, 4.86851938, 4.79831327]\]
model potential parameters from \(^1\)

\[\text{scaledRydbergConstant} = 13.60560723878171\]
in eV (M_ion core = m_atomic - m_electron)

class arc.alkali_atom_data.Sodium(preferQuantumDefects=True, cpp_numerov=True)
Bases: arc.alkali_atom_functions.AlkaliAtom

Properties of sodium 23 atoms

\[\text{a1} = [4.82223117, 5.08382502, 3.53324124, 1.11056646]\]
model potential parameters from \(^1\)

\[\text{a2} = [2.45449865, 2.18226881, 2.48697936, 1.05458759]\]
model potential parameters from \(^1\)

\[\text{a3} = [-1.12255048, -1.19534623, -0.75688448, 1.73203428]\]
model potential parameters from \(^1\)

\[\text{a4} = [-1.42631393, -1.03142861, -1.27852357, -0.09265696]\]
model potential parameters from \(^1\)

\[\text{abundance} = 1.0\]
source NIST, Atomic Weights and Isotopic Compositions \(^{14}\)

\[\text{alphaC} = 0.9448\]
model potential parameters from \(^1\)

\[\text{getPressure} (\text{temperature})\]
Pressure of atomic vapour at given temperature.
Uses equation and values from \(^3\). Values from table 2. (accuracy +- 5\%) are used for Na in solid phase.
Values from table 3. (accuracy +-1 \%) are used for Na in liquid phase.

\[\text{ionisationEnergy} = 5.13907592528896\]
(eV) from Ref. \(^{11}\)

\[\text{mass} = 3.8175410024881635e-26\]
source NIST, Atomic Weights and Isotopic Compositions \(^{14}\)

\[\text{meltingPoint} = 370.94399999999996\]
in K

\[\text{quantumDefect} = [[1.347964, 0.060673, 0.0233, -0.0085, 0.0, 0.0], [0.85538, 0.11363, 0.0384, 0.1412, 0.0, 0.0], [0.7958, -4.0513, 0.0, 0.0], [0.001453, 0.017312, -0.7809, 7.021, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]]\]
Quantum defects are from Ref. \(^{11}\). Note that we are using modified Rydberg-Ritz formula. In literature both modified and non-modified coefficients appear. For more details about the two equations see page 301. of Ref. \(^{11}\).

\[\text{rc} = [0.45489422, 0.45798739, 0.71875312, 28.6735059]\]
model potential parameters from \(^1\)

\[\text{scaledRydbergConstant} = 13.60536846146495\]
(eV)

1.3.3 Divalent atom functions

Note: This is module to be released in the forthcoming ARC 3.0.0 version. To used it now as a beta feature do:

```
from arc.beta import *
```
Publication describing this upgrade is in preparation (check this place soon). For now cite as “E. J. Robertson, N. Šibalić, R. M. Potvliege and M. P. A. Jones, *in preparation* (2020).

**Overview**

**DivalentAtom Methods**

- `DivalentAtom.getDipoleMatrixElement(n1, l1, ..., j1, m1)`: Dipole matrix element $\langle n_1l_1jm_1 | e^{r} | n_2l_2jm_2 \rangle$ in units of $a_0 e$.
- `DivalentAtom.getTransitionWavelength(n1, l1, ..., j1, m1)`: Calculated transition wavelength (in vacuum) in m.
- `DivalentAtom.getTransitionFrequency(n1, l1, ..., j1, m1)`: Calculated transition frequency in Hz.
- `DivalentAtom.getRabiFrequency(n1, l1, ..., j1, m1)`: Returns a Rabi frequency for resonantly driven atom in a center of TEM00 mode of a driving field.
- `DivalentAtom.getRabiFrequency2(n1, l1, ..., j1, m1)`: Returns a Rabi frequency for resonant excitation with a given electric field amplitude.
- `DivalentAtom.getStateLifetime(n, l, j, m, ...)`: Returns the lifetime of the state (in s).
- `DivalentAtom.getTransitionRate(n1, l1, ..., j1, m1)`: Transition rate due to coupling to vacuum modes (black body included).
- `DivalentAtom.getReducedMatrixElementJ_asymmetric(n1, ..., j1, m1)`: Reduced matrix element in $J$ basis, defined in asymmetric notation.
- `DivalentAtom.getReducedMatrixElementJ(n1, ..., j1, m1)`: Reduced matrix element in $J$ basis (symmetric notation).
- `DivalentAtom.getReducedMatrixElementL(n1, ..., j1, m1)`: Reduced matrix element in $L$ basis (symmetric notation).
- `DivalentAtom.getRadialMatrixElement(n1, l1, ..., j1, m1)`: Radial part of the dipole matrix element.
- `DivalentAtom.getQuadrupoleMatrixElement(n1, l1, ..., j1, m1)`: Radial part of the quadrupole matrix element.
- `DivalentAtom.getPressure(temperature)`: Vapour pressure (in Pa) at given temperature.
- `DivalentAtom.getNumberDensity(temperature)`: Atom number density at given temperature.
- `DivalentAtom.getAverageInteratomicSpacing(...)`: Returns average interatomic spacing in atomic vapour.
- `DivalentAtom.getEnergy(n, l, j, [s])`: Energy of the level relative to the ionisation level (in eV).
- `DivalentAtom.getZeemanEnergyShift(l, j, m, ...)`: Returns linear (paramagnetic) Zeeman shift.
- `DivalentAtom.getQuantumDefect(n, l, j, [s])`: Quantum defect of the level.
- `DivalentAtom.getC6term(n, l, j, n1, l1, j1, ...)`: C6 interaction term for the given two pair-states.
- `DivalentAtom.getC3term(n, l, j, n1, l1, j1, ...)`: C3 interaction term for the given two pair-states.
- `DivalentAtom.getEnergyDefect(n, l, j, n1, ...)`: Energy defect for the given two pair-states (one of the state has two atoms in the same state).
- `DivalentAtom.getEnergyDefect2(n, l, j, ...)`: Energy defect for the given two pair-states.
- `DivalentAtom.updateDipoleMatrixElementsFile()`: Updates the file with pre-calculated dipole matrix elements.
- `DivalentAtom.getRadialCoupling(n, l, j, n1, ...)`: Returns radial part of the coupling between two states (dipole and quadrupole interactions only).
- `DivalentAtom.getAverageSpeed(temperature)`: Average (mean) speed at a given temperature.

Continued on next page
DivalentAtom.getLiteratureDME(n1, l1, j1, ...)  
Returns literature information on requested transition.

Table 5 – continued from previous page

**Detailed documentation**

```python
class arc.divalent_atom_functions.DivalentAtom (preferQuantumDefects=True,  
cpp_numerov=True)

Bases: arc.alkali_atom_functions.AlkaliAtom

Implements general calculations for Alkaline Earths, and other divalent atoms.

This class inherits arc.alkali_atom_functions.AlkaliAtom. Most of the methods can be directly used from there, and the source for them is provided in the base class. Few methods that are implemented differently for Alkaline Earths are defined here.

**Parameters**

- **preferQuantumDefects (bool)** – Use quantum defects for energy level calculations. If False, uses NIST ASD values where available. If True, uses quantum defects for energy calculations for principal quantum numbers within the range specified in `defectFittingRange` which is specified for each element and series separately. For principal quantum numbers below this value, NIST ASD values are used if existing, since quantum defects. Default is True.

- **cpp_numerov (bool)** – This switch for Alkaline Earths at the moment doesn’t have any effect since wavefunction calculation function is not implemented (d.m.e. and quadrupole matrix elements are calculated directly semiclassically).

**corePotential (l, r)**

Not implemented for Alkaline earths

**defectFittingRange = {}**

Used for AlkalineEarths to define minimum and maximum principal quantum number for which quantum defects are valid. Ranges are stored under keys defined as state terms ("stateLabel":>[minN, maxN], e.g. ‘1S0’). Dictionary returns array stating minimal and maximal principal quantum number for which quantum defects were fitted. For example:

```python
limits = self.defectFittingRange['1S0']
print("Minimal n = \%d" % limits[0])
print("Maximal n = \%d" % limits[1])
```

**effectiveCharge (l, r)**

Not implemented for Alkaline earths

**energyLevelsExtrapolated = False**

flag that is turned to True if the energy levels of this atom were calculated by extrapolating with quantum defects values outside the quantum defect fitting range.

**getAverageInteratomicSpacing (temperature)**

Returns average interatomic spacing in atomic vapour

See calculation of basic properties example snippet.

**Parameters**  
**temperature (float)** – temperature of the atomic vapour

**Returns**  
average interatomic spacing in m

**Return type**  
float

**getAverageSpeed (temperature)**

Average (mean) speed at a given temperature

**Parameters**  
**temperature (float)** – temperature (K)

**Returns**  
mean speed (m/s)
Return type float

getC3term(n, l, j, n1, l1, j1, n2, l2, j2, s=0.5)
C3 interaction term for the given two pair-states

Calculates $C_3$ interaction term for $|n, l, j, n, l, j\rangle \leftrightarrow |n_1, l_1, j_1, n_2, l_2, j_2\rangle$

Parameters

- n (int) – principal quantum number
- l (int) – orbital angular momentum
- j (float) – total angular momentum
- n1 (int) – principal quantum number
- l1 (int) – orbital angular momentum
- j1 (float) – total angular momentum
- n2 (int) – principal quantum number
- l2 (int) – orbital angular momentum
- j2 (float) – total angular momentum
- s (float) – optional, total spin angular momentum of state. By default 0.5 for Alkali atoms.

Returns $C_3 = \langle n, l, j | \epsilon r | n_1, l_1, j_1 \rangle \langle n, l, j | \epsilon r | n_2, l_2, j_2 \rangle / (4\pi\varepsilon_0) (h \text{ Hz m}^3)$.

Return type float

getC6term(n, l, j, n1, l1, j1, n2, l2, j2, s=0.5)
C6 interaction term for the given two pair-states

Calculates $C_6$ interaction term for $|n, l, j, n, l, j\rangle \leftrightarrow |n_1, l_1, j_1, n_2, l_2, j_2\rangle$. For details of calculation see Ref.2.

Parameters

- n (int) – principal quantum number
- l (int) – orbital angular momentum
- j (float) – total angular momentum
- n1 (int) – principal quantum number
- l1 (int) – orbital angular momentum
- j1 (float) – total angular momentum
- n2 (int) – principal quantum number
- l2 (int) – orbital angular momentum
- j2 (float) – total angular momentum
- s (float) – optional, total spin angular momentum of state. By default 0.5 for Alkali atoms.

Returns $C_6 = \left( \frac{|\langle n, l, j | \epsilon r | n_1, l_1, j_1 \rangle |^2 |\langle n, l, j | \epsilon r | n_2, l_2, j_2 \rangle |^2}{E(n_1, l_1, j_1, n_2, l_2, j_2) - E(n_1, l_1, j_1) \} (h \text{ Hz m}^6)$.

Return type float

Example

We can reproduce values from Ref.\(^2\) for C\(_3\) coupling to particular channels. Taking for example channels described by the Eq. (50a-c) we can get the values:

```python
from arc import *

channels = [
    [70,0,0.5, 70, 1,1.5, 69,1, 1.5],
    [70,0,0.5, 70, 1,1.5, 69,1, 0.5],
    [70,0,0.5, 69, 1,1.5, 70,1, 0.5],
    [70,0,0.5, 70, 1,0.5, 69,1, 0.5]]

print(" = = = Caesium = = =")
atom = Caesium()
for channel in channels:
    print("%.0f GHz (mu m)^6" % ( atom.getC6term(*channel) / C_h * 1.e27 ))

print("= = = Rubidium = = =")
atom = Rubidium()
for channel in channels:
    print("%.0f GHz (mu m)^6" % ( atom.getC6term(*channel) / C_h * 1.e27 ))
```

Returns:

```
= = = Caesium = = =
722 GHz (mu m)^6
316 GHz (mu m)^6
383 GHz (mu m)^6
228 GHz (mu m)^6

= = = Rubidium = = =
799 GHz (mu m)^6
543 GHz (mu m)^6
589 GHz (mu m)^6
437 GHz (mu m)^6
```

which is in good agreement with the values cited in the Ref.\(^2\). Small discrepancies for Caesium originate from slightly different quantum defects used in calculations.

References

getDipoleMatrixElement\(\langle n_1 l_1 j_1 m_{j_1} | e_r | n_2 l_2 j_2 m_{j_2}\rangle\) in units of \(a_0 e\)

Parameters

- \(l_1, j_1, m_{j_1}\) \((n_1.)\) – principal, orbital, total angular momentum, and projection of total angular momentum for state 1
- \(l_2, j_2, m_{j_2}\) \((n_2.)\) – principal, orbital, total angular momentum, and projection of total angular momentum for state 2
- \(q\) \((\text{int})\) – specifies transition that the driving field couples to, +1, 0 or -1 corresponding to driving \(\sigma^+\), \(\pi\) and \(\sigma^-\) transitions respectively.
- \(s\) \((\text{float})\) – optional, total spin angular momentum of state. By default 0.5 for Alkali atoms.

Returns dipole matrix element( \(a_0 e\) )

Return type float
Example

For example, calculation of $5S_{1/2} m_j = -\frac{1}{2} \rightarrow 5P_{3/2} m_j = -\frac{3}{2}$ transition dipole matrix element for laser driving $\sigma^-$ transition:

```python
from arc import *
atom = Rubidium()
# transition 5 S_{1/2} m_j=-0.5 -> 5 P_{3/2} m_j=-1.5
print(atom.getDipoleMatrixElement(5,0,0.5,-0.5,5,1,1.5,-1.5,-1))
```

getDipoleMatrixElementHFS$(n1,l1,j1,f1, mf1, n2,l2,j2, f2, mf2, q, s=0.5)$
Dipole matrix element for hyperfine structure resolved transitions $\langle n_1 l_1 j_1 f_1 m_{f_1} | e r | n_2 l_2 j_2 f_2 m_{f_2} \rangle$ in units of $\alpha_0 e$

For hyperfine resolved transitions, the dipole matrix element is $\langle n_1 l_1 j_1 f_1 m_{f_1} | \hat{r} \cdot e \rangle |n_2 l_2 j_2 f_2 m_{f_2}\rangle$, where

$$\langle n_1 l_1 j_1 f_1 | e | n_2 l_2 j_2 f_2 \rangle = (1)^{j_1 + l + F_2 + 1} \sqrt{(2f_1 + 1)(2f_2 + 1)} \left\{ \begin{array}{cc} F_1 & 1 \\ f_2 \end{array} \right\} \left\{ \begin{array}{cc} F_2 & 1 \\ j_1 \end{array} \right\} \langle n_1 l_1 j_1 | e | n_2 l_2 j_2 \rangle$$

Parameters

- **$l1, j1, f1, mf1 (n1.)$** – principal, orbital, total orbital, fine basis (total atomic) angular momentum, and projection of total angular momentum for state 1
- **$l2, j2, f2, mf2 (n2.)$** – principal, orbital, total orbital, fine basis (total atomic) angular momentum, and projection of total angular momentum for state 2
- **$q (int)$** – specifies transition that the driving field couples to, +1, 0 or -1 corresponding to driving $\sigma^+$, $\pi$ and $\sigma^-$ transitions respectively.
- **$s (float)$** – optional, total spin angular momentum of state. By default 0.5 for Alkali atoms.

Returns dipole matrix element( $\alpha_0 e$)

Return type **float**

getEnergy$(n, l, j, s=None)$
Energy of the level relative to the ionisation level (in eV)

Returned energies are with respect to the center of gravity of the hyperfine-split states. If **preferQuantumDefects =False** (set during initialization) program will try use NIST energy value, if such exists, falling back to energy calculation with quantum defects if the measured value doesn’t exist. For **preferQuantumDefects =True**, program will always calculate energies from quantum defects (useful for comparing quantum defect calculations with measured energy level values).

Parameters

- **$n (int)$** – principal quantum number
- **$l (int)$** – orbital angular momentum
- **$j (float)$** – total angular momentum
- **$s (float)$** – optional, total spin angular momentum. Default value of 0.5 is correct for Alkali atoms, and has to be specified explicitly for divalent atoms.

Returns state energy (eV)

Return type **float**

getEnergyDefect$(n, l, j, n1, l1, j1, n2, l2, j2, s=0.5)$
Energy defect for the given two pair-states (one of the state has two atoms in the same state)

Energy difference between the states $E(n_1,l_1,j_1,n_2,l_2,j_2) - E(n,l,j,n,l,j)$
Parameters

- n (int) – principal quantum number
- l (int) – orbital angular momentum
- j (float) – total angular momentum
- n1 (int) – principal quantum number
- l1 (int) – orbital angular momentum
- j1 (float) – total angular momentum
- n2 (int) – principal quantum number
- l2 (int) – orbital angular momentum
- j2 (float) – total angular momentum
- s (float) – optional. Spin angular momentum (default 0.5 for Alkali)

Returns energy defect (SI units: J)

Return type float

gEnergyDefect2 (n, l, j, nn, ll, jj, n1, l1, j1, n2, l2, j2, s=0.5)
Energy defect for the given two pair-states
Energy difference between the states $E(n_1, l_1, j_1, n_2, l_2, j_2) - E(n, l, j, nn, ll, jj)$
See pair-state energy defects example snippet.

Parameters

- n (int) – principal quantum number
- l (int) – orbital angular momentum
- j (float) – total angular momentum
- nn (int) – principal quantum number
- ll (int) – orbital angular momentum
- jj (float) – total angular momentum
- n1 (int) – principal quantum number
- l1 (int) – orbital angular momentum
- j1 (float) – total angular momentum
- n2 (int) – principal quantum number
- l2 (int) – orbital angular momentum
- j2 (float) – total angular momentum
- s (float) – optional. Spin angular momentum (default 0.5 for Alkali)

Returns energy defect (SI units: J)

Return type float

gLiteratureDME (n1, l1, j1, n2, l2, j2, s=0)
Returns literature information on requested transition.

Parameters

- n1, l1, j1 – one of the states we are coupling
- n2, l2, j2 – the other state to which we are coupling
- s – (optional) spin of the state. Default s=0.
Returns

hasLiteratureValue?, dme, referenceInformation

If Boolean value is True, a literature value for dipole matrix element was found and reduced DME in J basis is returned as the number. The third returned argument (array) contains additional information about the literature value in the following order [ typeOfSource, errorEstimate, comment, reference, reference DOI] upon success to find a literature value for dipole matrix element:

• typeOfSource=1 if the value is theoretical calculation; otherwise, if it is experimentally obtained value typeOfSource=0
• comment details where within the publication the value can be found
• errorEstimate is absolute error estimate
• reference is human-readable formatted reference
• reference DOI provides link to the publication.

Boolean value is False, followed by zero and an empty array if no literature value for dipole matrix element is found.

Return type bool, float, [int, float, string, string, string]

Note: The literature values are stored in /data folder in <element name>_literature_dme.csv files as a ; separated values. Each row in the file consists of one literature entry, that has information in the following order:

• n1
• l1
• j1
• n2
• l2
• j2
• s
• dipole matrix element reduced l basis (a.u.)
• comment (e.g. where in the paper value appears?)
• value origin: 1 for theoretical; 0 for experimental values
• accuracy
• source (human readable formatted citation)
• doi number (e.g. 10.1103/RevModPhys.82.2313 )

If there are several values for a given transition, program outputs the value that has smallest error (under column accuracy). The list of values can be expanded - every time program runs this file is read and the list is parsed again for use in calculations.

getNumberDensity(temperature)
Atom number density at given temperature

See calculation of basic properties example snippet.

Parameters temperature (float) – temperature in K

Returns atom concentration in $1/m^3$

Return type float
getPressure(temperature)
Vapour pressure (in Pa) at given temperature
Parameters temperature (float) – temperature in K
Returns vapour pressure in Pa
Return type float

getQuadrupoleMatrixElement(n1, l1, j1, n2, l2, j2, s=0.5)
Radial part of the quadrupole matrix element
Calculates \( \int dr R_{n_1,l_1,j_1}(r) \cdot R_{n_1,l_1,j_1}(r) \cdot r^4 \). See Quadrupole calculation example snippet.
Parameters
• n1 (int) – principal quantum number of state 1
• l1 (int) – orbital angular momentum of state 1
• j1 (float) – total angular momentum of state 1
• n2 (int) – principal quantum number of state 2
• l2 (int) – orbital angular momentum of state 2
• j2 (float) – total angular momentum of state 2
• s (float) – optional. Spin of the state. Default 0.5 is for Alkali
Returns quadrupole matrix element (a\(e^2\)).
Return type float

getQuantumDefect(n, l, j, s=0.5)
Quantum defect of the level.
For an example, see Rydberg energy levels example snippet.
Parameters
• n (int) – principal quantum number
• l (int) – orbital angular momentum
• j (float) – total angular momentum
• s (float) – (optional). Total spin angular momentum. Default value of 0.5 correct for Alkali atoms. For divalent atoms it has to be explicitly defined.
Returns quantum defect
Return type float

getRabiFrequency(n1, l1, j1, mj1, n2, l2, j2, q, laserPower, laserWaist, s=0.5)
Returns a Rabi frequency for resonantly driven atom in a center of TEM00 mode of a driving field
Parameters
• n1, l1, j1, mj1 – state from which we are driving transition
• n2, l2, j2 – state to which we are driving transition
• q – laser polarization (-1,0,1 correspond to \(\sigma^-\), \(\pi\) and \(\sigma^+\) respectively)
• laserPower – laser power in units of W
• laserWaist – laser 1/e\(^2\) waist (radius) in units of m
• s (float) – optional, total spin angular momentum of state. By default 0.5 for Alkali atoms.
Returns Frequency in rad\(^{-1}\). If you want frequency in Hz, divide by returned value by 2\(\pi\)
Return type float

1.3. Detailed documentation of functions
getRabiFrequency2 \((n1, l1, j1, m1j1, n2, l2, j2, q, electricFieldAmplitude, s=0.5)\)

Returns a Rabi frequency for resonant excitation with a given electric field amplitude

**Parameters**

- \(n1, l1, j1, m1j1\) – state from which we are driving transition
- \(n2, l2, j2\) – state to which we are driving transition
- \(q\) – laser polarization (-1,0,1 correspond to \(\sigma^-\), \(\pi\) and \(\sigma^+\) respectively)
- \(electricFieldAmplitude\) – amplitude of electric field driving (V/m)
- \(s\) (float) – optional, total spin angular momentum of state. By default 0.5 for Alkali atoms.

**Returns** Frequency in \(\text{rad}^{-1}\). If you want frequency in Hz, divide by returned value by \(2\pi\)

**Return type** float

getRadialCoupling \((n, l, n1, l1, j1, s=0.5)\)

Returns radial part of the coupling between two states (dipole and quadrupole interactions only)

**Parameters**

- \(n1\) (int) – principal quantum number
- \(l1\) (int) – orbital angular momentum
- \(j1\) (float) – total angular momentum
- \(n2\) (int) – principal quantum number
- \(l2\) (int) – orbital angular momentum
- \(j2\) (float) – total angular momentum
- \(s\) (float) – optional, total spin angular momentum of state. By default 0.5 for Alkali atoms.

**Returns** radial coupling strength (in a.u.), or zero for forbidden transitions in dipole and quadrupole approximation.

**Return type** float

getRadialMatrixElement \((n1, l1, j1, n2, l2, j2, s=None, useLiterature=True)\)

Radial part of the dipole matrix element

Calculates \(\int dr \cdot R_{n1,l1,j1}(r) \cdot R_{n2,l2,j2}(r) \cdot r^3\).

**Parameters**

- \(n1\) (int) – principal quantum number of state 1
- \(l1\) (int) – orbital angular momentum of state 1
- \(j1\) (float) – total angular momentum of state 1
- \(n2\) (int) – principal quantum number of state 2
- \(l2\) (int) – orbital angular momentum of state 2
- \(j2\) (float) – total angular momentum of state 2
- \(s\) (float) – is required argument, total spin angular momentum of state. Specify \(s=0\) for singlet state or \(s=1\) for triplet state.

**Returns** dipole matrix element \((a_0 e)\).

**Return type** float

getReducedMatrixElementJ \((n1, l1, j1, n2, l2, j2, s=0.5)\)

Reduced matrix element in \(J\) basis (symmetric notation)

**Parameters**
• \( n_1 (\text{int}) \) – principal quantum number of state 1
• \( l_1 (\text{int}) \) – orbital angular momentum of state 1
• \( j_1 (\text{float}) \) – total angular momentum of state 1
• \( n_2 (\text{int}) \) – principal quantum number of state 2
• \( l_2 (\text{int}) \) – orbital angular momentum of state 2
• \( j_2 (\text{float}) \) – total angular momentum of state 2
• \( s (\text{float}) \) – optional, total spin angular momentum of state. By default 0.5 for Alkali atoms.

Returns reduced dipole matrix element in \( J \) basis \( \langle j || e r || j' \rangle (a_0 e) \).

Return type float

\textbf{getReducedMatrixElementJ\_asymmetric} (\( n_1, l_1, j_1, n_2, l_2, j_2, s=0.5 \))

Reduced matrix element in \( J \) basis, defined in asymmetric notation.

Note that notation for symmetric and asymmetrically defined reduced matrix element is not consistent in the literature. For example, notation is used e.g. in Steck\(^1\) is precisely the opposite.

Note: Note that this notation is asymmetric: \( \langle j || e r || j' \rangle \neq \langle j' || e r || j \rangle \). Relation between the two notation is \( \langle j || e r || j' \rangle = \sqrt{2j+1} \langle j || e r || j' \rangle \). This function always returns value for transition from lower to higher energy state, independent of the order of states entered in the function call.

Parameters

• \( n_1 (\text{int}) \) – principal quantum number of state 1
• \( l_1 (\text{int}) \) – orbital angular momentum of state 1
• \( j_1 (\text{float}) \) – total angular momentum of state 1
• \( n_2 (\text{int}) \) – principal quantum number of state 2
• \( l_2 (\text{int}) \) – orbital angular momentum of state 2
• \( j_2 (\text{float}) \) – total angular momentum of state 2
• \( s (\text{float}) \) – optional, total spin angular momentum of state. By default 0.5 for Alkali atoms.

Returns reduced dipole matrix element in Steck notation \( \langle j || e r || j' \rangle (a_0 e) \).

Return type float

\textbf{getReducedMatrixElementL} (\( n_1, l_1, j_1, n_2, l_2, j_2, s=0.5 \))

Reduced matrix element in \( L \) basis (symmetric notation)

Parameters

• \( n_1 (\text{int}) \) – principal quantum number of state 1
• \( l_1 (\text{int}) \) – orbital angular momentum of state 1
• \( j_1 (\text{float}) \) – total angular momentum of state 1
• \( n_2 (\text{int}) \) – principal quantum number of state 2
• \( l_2 (\text{int}) \) – orbital angular momentum of state 2
• \( j_2 (\text{float}) \) – total angular momentum of state 2

Returns reduced dipole matrix element in \( L \) basis \( \langle l || e r || l' \rangle (a_0 e) \).

---

Return type  float

**getStateLifetime**  \((n, l, j, \text{temperature}=0, \text{includeLevelsUpTo}=0, s=0)\)

Returns the lifetime of the state (in s)

For non-zero temperatures, user must specify up to which principal quantum number levels, that is **above** the initial state, should be included in order to account for black-body induced transitions to higher lying states. See Rydberg lifetimes example snippet.

**Parameters**

- \(l, j (n, \) – specifies state whose lifetime we are calculating
- \(\text{temperature} \) – optional. Temperature at which the atom environment is, measured in K. If this parameter is non-zero, user has to specify transitions up to which state (due to black-body decay) should be included in calculation.
- \(\text{includeLevelsUpTo} (\text{int}) \) – optional and not needed for atom lifetimes calculated at zero temperature. At non zero temperatures, this specify maximum principal quantum number of the state to which black-body induced transitions will be included. Minimal value of the parameter in that case is \(n + 1\)
- \(s (\text{float}) \) – optional, total spin angular momentum of state. By default 0.5 for Alkali atoms.

**Returns**  State lifetime in units of s (seconds)

Return type  float

See also:

**getTransitionRate** for calculating rates of individual transition rates between the two states

**getTransitionFrequency**  \((n1, l1, j1, n2, l2, j2, s1=0.5, s2=0.5)\)

Calculated transition frequency in Hz

Returned values is given relative to the centre of gravity of the hyperfine-split states.

**Parameters**

- \(n1 (\text{int}) \) – principal quantum number of the state from which we are going
- \(l1 (\text{int}) \) – orbital angular momentum of the state from which we are going
- \(j1 (\text{float}) \) – total angular momentum of the state from which we are going
- \(n2 (\text{int}) \) – principal quantum number of the state to which we are going
- \(l2 (\text{int}) \) – orbital angular momentum of the state to which we are going
- \(j2 (\text{float}) \) – total angular momentum of the state to which we are going
- \(s1 (\text{float}) \) – optional, spin of the initial state (for Alkali this is fixed to 0.5)
- \(s2 (\text{float}) \) – optional, spin of the final state (for Alkali this is fixed to 0.5)

**Returns**  transition frequency (in Hz). If the returned value is negative, level from which we are going is **above** the level to which we are going.

Return type  float

**getTransitionRate**  \((n1, l1, j1, n2, l2, j2, \text{temperature}=0.0, s=0.5)\)

Transition rate due to coupling to vacuum modes (black body included)

Calculates transition rate from the first given state to the second given state \(|n1, l1, j1\rangle \rightarrow |n2, j2, j2\rangle\) at given temperature due to interaction with the vacuum field. For zero temperature this returns Einstein A coefficient. For details of calculation see Ref.\(^3\) and Ref.\(^4\). See Black-body induced population transfer example snippet.


Parameters

• \( n_1 \) (int) – principal quantum number
• \( l_1 \) (int) – orbital angular momentum
• \( j_1 \) (float) – total angular momentum
• \( n_2 \) (int) – principal quantum number
• \( l_2 \) (int) – orbital angular momentum
• \( j_2 \) (float) – total angular momentum
• [temperature] (float) – temperature in K
• \( s \) (float) – optional, total spin angular momentum of state. By default 0.5 for Alkali atoms.

Returns transition rate in s\(^{-1}\) (SI)

Return type float

References

getTransitionWavelength \((n_1, l_1, j_1, n_2, l_2, j_2, s_1=0.5, s_2=0.5)\)

Calculated transition wavelength (in vacuum) in m.

Returned values is given relative to the centre of gravity of the hyperfine-split states.

Parameters

• \( n_1 \) (int) – principal quantum number of the state from which we are going
• \( l_1 \) (int) – orbital angular momentum of the state from which we are going
• \( j_1 \) (float) – total angular momentum of the state from which we are going
• \( n_2 \) (int) – principal quantum number of the state to which we are going
• \( l_2 \) (int) – orbital angular momentum of the state to which we are going
• \( j_2 \) (float) – total angular momentum of the state to which we are going
• \( s_1 \) (float) – optional, spin of the intial state (for Alkali this is fixed to 0.5)
• \( s_2 \) (float) – optional, spin of the final state (for Alkali this is fixed to 0.5)

Returns transition wavelength (in m). If the returned value is negative, level from which we are going is above the level to which we are going.

Return type float

getZeemanEnergyShift \((l, j, m_j, magneticFieldBz, s=0.5)\)

Retuns linear (paramagnetic) Zeeman shift.

\[
\mathcal{H}_P = \frac{\mu_B B_z}{\hbar} (\hat{L}_z + g_S S_z)
\]

Parameters

• \( l \) (int) – orbital angular momentum
• \( j \) (float) – total angular momentum
• \( m_j \) (float) – projection of total angular momentum alon z-axis
• magneticFieldBz (float) – applied magnetic field (alon z-axis only) in units of T (Tesla)
• \( s \) (float) – optional, total spin angular momentum of state. By default 0.5 for Alkali atoms.

Returns energy offset of the state (in J)
Return type float

```
levelDataFromNIST = ''
file with .csv data, each row is [n, l, s, j, energy, source, absolute uncertainty]
```

```
minQuantumDefectN = None
Not used with DivalentAtom, see defectFittingRange instead.
```

```
modelPotential_coef = {}
Model potential parameters fitted from experimental observations for different l (electron angular momentum)
```

```
potential (l, s, j, r)
Not implemented for Alkaline earths
```

```
quantumDefect = [[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]],
Contains list of modified Rydberg-Ritz coefficients for calculating quantum defects for [1S0, 1P1, 1D2, 1F3], [3S0, 3P0, 3D1, 3F2], [3S0, 3P1, 3D2, 3F3], [3S1, 3P2, 3D3, 3F4].
```

```
radialWavefunction (l, s, j, stateEnergy, innerLimit, outerLimit, step)
Not implemented for Alkaline earths
```

```
updateDipoleMatrixElementsFile()
Updates the file with pre-calculated dipole matrix elements.
This function will add the the file all the elements that have been calculated in the previous run, allowing quick access to them in the future calculations.
```

### 1.3.4 Divalent atom data

**Note:** This is module to be released in the forthcoming ARC 3.0.0 version. To used it now as a beta feature do:

```
from arc.beta import *
```

Publication describing this upgrade is in preparation (check this place soon). For now cite as “E. J. Robertson, N. Šibalić, R. M. Potvliege and M. P. A. Jones, in preparation (2020)”.

<table>
<thead>
<tr>
<th>Strontium88([preferQuantumDefects, cpp_numerov])</th>
<th>Properties of Strontium 88 atoms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calcium40([preferQuantumDefects, cpp_numerov])</td>
<td>Properties of Calcium 40 atoms</td>
</tr>
<tr>
<td>Ytterbium174([preferQuantumDefects, cpp_numerov])</td>
<td>Properties of Ytterbium 174 atoms</td>
</tr>
</tbody>
</table>

**Data sources**

```
class arc.divalent_atom_data.Calcium40 (preferQuantumDefects=True, cpp_numerov=True)
Bases: arc.divalent_atom_functions.DivalentAtom
Properties of Calcium 40 atoms
```

```
defectFittingRange = {'1D2': [36, 66], '1F3': [10, 25], '1P1': [14, 28], '1S0':
Quantum defect principal quantum number fitting ranges for different series
```

```
extraLevels = []
TODO unkown if such exist at time of writing
```

```
getPressure (temperature)
Pressure of atomic vapour at given temperature.
```
Calculates pressure based on Ref.\textsuperscript{17} (accuracy +- 5%).

\begin{verbatim}
ionisationEnergy = 6.113155417663086
 eV ref.\textsuperscript{15}
levelDataFromNIST = 'ca_level_data.csv'
Sources Refs.\textsuperscript{1,5,9,14}
mass = 6.635944355805756e-26
Ref.\textsuperscript{18}
meltingPoint = 1115.15
in K
quantumDefect = [[[2.33793, -3.96, 0.0, 0.0, 0.0, 0.0], [1.885584, -0.114, -23.8, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]]
Contains list of modified Rydberg-Ritz coefficients for calculating quantum defects for \[1\] S\(0\), \[1\] P\(1\), \[1\] D\(2\), \[1\] F\(3\], \[3\] S\(0\), \[3\] P\(0\), \[3\] D\(1\), \[3\] F\(2\], \[3\] S\(0\), \[3\] P\(1\), \[3\] D\(2\), \[3\] F\(3\], \[3\] S\(1\), \[3\] P\(2\), \[3\] D\(3\), \[3\] F\(4\].
scaledRydbergConstant = 13.605506438624705
todo source class arc.divalent_atom_data.Strontium88 (preferQuantumDefects=True, cpp_numerov=True)
Bases: arc.divalent_atom_functions.DivalentAtom
Properties of Strontium 88 atoms
defectFittingRange = {'1D2': [36, 66], '1F3': [10, 25], '1P1': [14, 28], '1S0': [14, 34], '3D1': [20, 32], ...
24], '3F3': [10, 24], '3F4': [10, 24], '3P0': [8, 15], '3P1': [8, 22], '3P2': [8, 18], '3S1': [13, 45]}
Quantum defect principal quantum number fitting ranges for different series
getPressure (temperature)
Pressure of atomic vapour at given temperature.
Calculates pressure based on Ref.\textsuperscript{17} (accuracy +- 5%).

\begin{verbatim}
ionisationEnergy = 5.6948674
(eV) ref.\textsuperscript{3}
levelDataFromNIST = 'sr_level_data.csv'
Sources Refs.\textsuperscript{1,2,3,4,5,6,7,8,10}
mass = 1.4597071452315522e-25
Ref.\textsuperscript{18}
\end{verbatim}

\textsuperscript{5} P. Escherick, J. J. Wynne and J. A. Armstrong, Spectroscopy of 3P\(0\) states of alkaline earths, Optics Letters 1, 19 (1977).
\textsuperscript{14} Thomas R. Gentile, Barbara J. Hughey, Daniel Kleppner and Theodore W. Ducas, Microwave spectroscopy of calcium Rydberg states, Physical Review A 42, 440 (1990)
\textsuperscript{18} NIST Standard reference database, https://dx.doi.org/10.18434/T4FW23
\textsuperscript{2} R. Beigang, K. Lücke, A. Timmermann, P.J.West and D.Frölich, Determination of absolute level energies of 5sns1S\(0\) and 5snd1D\(2\) Rydberg series of Sr, Opt. Commun. 19, 1982.
\textsuperscript{6} P Escherick, Bound, even-parity J = 0 and J = 2 spectra of Sr, Physical Review A 15, 1920 (1977).

1.3. Detailed documentation of functions 45
meltingPoint = 1050.15
in K

quantumDefect = [[[3.26923346261, -0.252029996277, 12.6529707842, 0.0, 0.0, 0.0], [2.73329407388, -5.97060805042, ... 0.0, 0.0, 0.0], [0.121422935185, -2.86416051794, 157.683861744, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]]
Contains list of modified Rydberg-Ritz coefficients for calculating quantum defects for [[\text{1}S_0, \text{1}P_1, \text{1}D_2, \text{1}F_3], [\text{3}S_0, \text{3}P_0, \text{3}D_1, \text{3}F_2], [\text{3}S_0, \text{3}P_1, \text{3}D_2, \text{3}F_3], [\text{3}S_1, \text{3}P_2, \text{3}D_3, \text{3}F_4]].
scaledRydbergConstant = 13.60560773714796
TODO source
class arc.divalent_atom_data.Ytterbium174 (preferQuantumDefects=True, cpp_numerov=True)
Bases: arc.divalent_atom_functions.DivalentAtom
Properties of Ytterbium 174 atoms
defectFittingRange = {'1D2': [28, 75], '1P1': [35, 53], '1S0': [15, 43], '3D2': [10, 52]}
Quantum defect principal quantum number fitting ranges for different series
extraLevels = []
TODO unknown if such exist at time of writing
getPressure (temperature)
Pressure of atomic vapour at given temperature.
Calculates pressure based on Ref.\textsuperscript{17} (accuracy +- 5%).
levelDataFromNIST = 'yb_level_data.csv'
Sources Refs.\textsuperscript{11,12,13}
mass = 2.888322828573181e-25
Ref.\textsuperscript{18}
meltingPoint = 1092.15
in K
quantumDefect = [[[4.27914, -7.06, 565, 0.0, 0.0, 0.0], [3.95433, -12.33, 1729, 0.0, 0.0, 0.0], [2.71363, -2.01, 571, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]],
Contains list of modified Rydberg-Ritz coefficients for calculating quantum defects for [[\text{1}S_0, \text{1}P_1, \text{1}D_2, \text{1}F_3], [\text{3}S_0, \text{3}P_0, \text{3}D_1, \text{3}F_2], [\text{3}S_0, \text{3}P_1, \text{3}D_2, \text{3}F_3], [\text{3}S_1, \text{3}P_2, \text{3}D_3, \text{3}F_4]].
scaledRydbergConstant = 13.60560773714796
TODO source

1.3.5 Single atom calculations

Note: Some of the functions (Wavefunction, AtomSurfaceVdW, OpticalLattice1D, DynamicPolarizability, and optical materials properties) in this modules will be released in the forthcoming ARC 3.0.0 version. To used them now as a beta feature do:

```
from arc.beta import *
```

Publication describing this upgrade is in preparation (check this place soon). For now cite as “E. J. Robertson, N. Šibalić, R. M. Potvliege and M. P. A. Jones, in preparation (2020)”.

Overview

Wavefunction Methods

- **Wavefunction.getRtimesPsiSpherical**(theta, phi, r) 
  Calculates list of $r \cdot \psi_m(\theta, \phi, r)$

- **Wavefunction.getRtimesPsi**(x, y, z) 
  Calculates list of $r \cdot \psi_m(x, y, z)$

- **Wavefunction.getPsi**(x, y, z) 
  Calculates list of $\psi_m(x, y, z)$

- **Wavefunction.getRtimesPsiSquaredInPlane**(…) 
  Calculates $|r \cdot \psi|^2$ on a mesh in a given plane.

- **Wavefunction.plot2D**(plane, pointsPerAxis, …) 
  2D colour plot of $|r \cdot \psi|^2$ wavefunction in a requested plane.

- **Wavefunction.plot3D**(plane, pointsPerAxis, …) 
  3D colour surface plot of $|r \cdot \psi|^2$ wavefunction in a requested plane.

StarkMap Methods

- **StarkMap.defineBasis**(n, l, j, mj, nMin, …) 
  Initializes basis of states around state of interest

- **StarkMap.diagonalise**(eFieldList[…]) 
  Finds atom eigenstates in a given electric field

- **StarkMap.plotLevelDiagram**(units, …) 
  Makes a plot of a stark map of energy levels

- **StarkMap.showPlot**(interactive) 
  Shows plot made by plotLevelDiagram

- **StarkMap.savePlot**(filename) 
  Saves plot made by plotLevelDiagram

- **StarkMap.getPolarizability**(maxField, …) 
  Returns the polarizability of the state (set during the initialization process).

- **StarkMap.getState**(state, electricField, …) 
  Returns basis states and coefficients that make up for a given electric field the eigenstate with largest contribution of the original state.

LevelPlot Methods

- **LevelPlot.makeLevels**(nFrom, nTo, lFrom, lTo) 
  Constructs energy level diagram in a given range

- **LevelPlot.drawLevels**(…) 
  Draws a level diagram plot

- **LevelPlot.showPlot**(…) 
  Shows a level diagram plot

AtomSurfaceVdW Methods

- **AtomSurfaceVdW.getC3contribution**(n1, l1, j1, …) 
  Contribution to $C_3^\text{v} \text{dW}$ of $|n_1, \ell_1, j_1\rangle$ state due to dipole coupling to $|n_2, \ell_2, j_2\rangle$ state.

- **AtomSurfaceVdW.getTateC3**(n, l, j, …, s, …) 
  Van der Waals atom-surface interaction coefficient for a given state ($C_3$ in units of $J \cdot m^3$)

OpticalLattice1D Methods

- **OpticalLattice1D.getRecoilEnergy**(…) 
  Recoil energy for atoms in given optical lattice

- **OpticalLattice1D.getTrappingFrequency**(…) 
  Atom’s trapping frequency for given trapth depth

- **OpticalLattice1D.defineBasis**(lLimit) 
  Define basis for Bloch band calculations

Continued on next page
DynamicPolarizability Methods

- DynamicPolarizability\_defineBasis(nMin, nMax)
  - Defines basis for calculation of dynamic polarizability
- DynamicPolarizability\_getPolarizability()
  - Calculates scalar and tensor polarizability
- DynamicPolarizability\_plotPolarizability()
  - Plots of polarisability for a range of wavelengths.

Optical material properties

- OpticalMaterial
  - Abstract class implementing calculation of basic properties for optical materials.
- Air()
  - Air as an optical material at normal conditions
- Sapphire()
  - Sapphire as optical material.

Detailed documentation

This module provides calculations of single-atom properties.

Included calculations are Stark maps, level plot visualisations, lifetimes and radiative decays.

**class** arc.calculations\_atom\_single.AtomSurfaceVdW(\textit{atom}, surfaceMaterial=None)

Calculates atom-surface Van der Waals interaction.

Energy of atom state |i\rangle at distance z from the surface of material is offseted in energy by \( V_{VdW} \) at small distances \( z \ll \min(\lambda_i,j) \), where \( \lambda_i,j \) are the wavelengths from atom state |i\rangle to all strongly-coupled states j, due to (unretarded) atom-surface interaction, also called Van der Waals interaction. The interaction potential can be expressed as

\[
V_{VdW} = -\frac{C_3}{z^3}
\]

This class calculates \( C_3 \) for individual states |i\rangle.

**Parameters**

- \textit{atom} (AlkaliAtom or DivalentAtom) – specified Alkali or Alkaline Earth atom whose interaction with surface we want to explore
- \textit{material} (from arc.materials) – specified surface material

**Note:** To find frequency shift of a transition |a\rangle \rightarrow |b\rangle, one needs to calculate difference in \( C_3 \) coefficients obtained for the two states |a\rangle and |b\rangle respectively. See example TODO (TO-DO)

**getC3contribution** (\textit{n1}, \textit{l1}, \textit{j1}, \textit{n2}, \textit{l2}, \textit{j2}, s=0.5)

Contribution to \( C_3 \) of \( |n_1, \ell_1, j_1\rangle \) state due to dipole coupling to \( |n_2, \ell_2, j_2\rangle \) state.

Calculates \[
\frac{1}{4\pi\varepsilon_0} \frac{\frac{1}{4}(\langle a|D_x|b\rangle)^2-1}{(\langle a|D_y|b\rangle)^2+1}\left(\frac{\langle a|D_x|b\rangle^2}{\langle a|D_y|b\rangle^2+2}\frac{\langle a|D_z|b\rangle^2}{16}\right)
\]
where \( |a\rangle \equiv |n_1, \ell_1, j_1\rangle, \quad |b\rangle \equiv |n_2, \ell_2, j_2\rangle \), \( \mathbf{D} \equiv e \cdot \mathbf{r} \equiv \hat{x}D_x + \hat{y}D_y + \hat{z}D_z \) is atomic dipole operator and \( n(\omega_{ab}) \) is refractive index of the considered surface at transition frequency \( \omega_{ab} \).

**Parameters**

- \( n_1 \) (int) – principal quantum number of state 1
- \( \ell_1 \) (int) – orbital angular momentum of state 1
- \( j_1 \) (float) – total angular momentum of state 1
- \( n_2 \) (int) – principal quantum number of state 2
- \( \ell_2 \) (int) – orbital angular momentum of state 2
- \( j_2 \) (float) – total angular momentum of state 2
- \( s \) (float, optional) – total spin angular momentum of states. Default value of 0.5 is correct for AlkaliAtoms. For DivalentAtom it has to be explicitly stated

**Returns**

contribution to VdW coefficient \( C_3 \), estimated error \( \delta C_3 \) (in units of \( J \cdot m^3 \)), and refractive index \( n \) of the surface material for the given transition.

**Return type** float, float, float

**Warning:** This is just contribution of one transition to the level shift of a particular state. To calculate total level shift, check `AtomSurfaceVdW.getStateC3`

### getStateC3 \((n, l, j, \text{coupledStatesList}, s=0.5, \text{debugOutput}=False)\)

Van der Waals atom-surface interaction coefficient for a given state \((C_3 \text{ in units of } J \cdot m^3)\)

**Parameters**

- \( n \) (int) – principal quantum number of the state
- \( l \) (int) – orbital angular momentum of the state
- \( j \) (int) – total angular momentum of state
- \( \text{coupledStatesList} \) (array) – array of states that are strongly dipole-coupled to the initial state, whose contribution to \( C_3 \) will be take into account. Format \([\{n1, l1, j1\}, \ldots]\)
- \( s \) (float, optional) – total spin angular momentum for the considered state. Default value of 0.5 is correct for AlkaliAtoms, but it has to be explicitly specified for DivalentAtom.
- \( \text{debugOutput} \) (bool, optional) – prints additional output information, False by default.

**Returns** \( C_3 \) (in units of \( J \cdot m^3 \)), estimated error \( \delta C_3 \)

**Return type** float, float

### class arc.calculations_atom_single.DynamicPolarizability \((atom, n, l, j, s=0.5)\)

Calculations of magic wavelengths and dynamic polarizability (scalar and tensor).

**Parameters**

- \( atom \) – alkali or alkaline element of choice
- \( n \) (int) – principal quantum number of the selected state
- \( l \) (int) – orbital angular momentum of the selected state
- \( j \) (float) – total angular momentum of selected state
- \( s \) (float) – optional, spin state of the atom. Default value of 0.5 is correct for Alkali atoms, but it has to be explicitly specified for DivalentAtom.
**defineBasis**(nMin, nMax)

Defines basis for calculation of dynamic polarizability

**Parameters**

- nMin (int) – minimal principal quantum number of states to be taken into account for calculation
- nMax (int) – maximal principal quantum number of states to be taken into account for calculation

**getPolarizability**(driveWavelength, mj=None, units='SI')

Calculates of scalar and tensor polarizability

**Parameters**

- driveWavelength (float) – wavelength of driving field (in units of m)
- mj (float) – optional, mj of the state for which we calculate polarizability. Default value is value of `self.j`.
- units (string) – optional, ‘SI’ or ‘a.u.’ (equivalently ‘au’), switches between SI units for returned result (\(HzV^{-2}m^2\)) and atomic units (“\(a_0^6\”)). Default ‘SI’

**plotPolarizability**(wavelengthList, mj=None, addToPlotAxis=None, line='b-', units='SI', debugOutput=False)

Plots of polarisability for a range of wavelengths.

Can be combined for different states to allow finding magic wavelengths for pairs of states. See example

---

**Todo:** Add link to example calculation of magic wavelengths

---

**class arc.calculations_atom_single.LevelPlot**(atomType)

Single atom level plots and decays (a Grotrian diagram, or term diagram)

For an example see Rydberg energy levels example snippet.

**Parameters**

- wavelengthList (array) – wavelengths for which we want to calculate polarizability (in units of m).
- mj (float) – optional, mj projection of the total angular momentum for the states for which we are calculating polarizability. By default it’s +j.
- line (string) – optional, line style short definition to be passed to matplotlib when plotting calculated polarisabilities
- units (string) – optional, ‘SI’ or ‘a.u.’ (equivalently ‘au’), switches between SI units for returned result (\(HzV^{-2}m^2\)) and atomic units (“\(a_0^6\”)”). Default ‘SI’.
- debugOutput (bool) – optional. Print additional output on resonances Default value False.

**class arc.calculations_atom_single.LevelPlot**(atomType)

Single atom level plots and decays (a Grotrian diagram, or term diagram)

For an example see Rydberg energy levels example snippet.

**Parameters**

- atom (AlkaliAtom) – =

  - alkali_atom_data.Lithium6
  - alkali_atom_data.Lithium7
  - alkali_atom_data.Sodium
  - alkali_atom_data.Potassium39
  - alkali_atom_data.Potassium40
  - alkali_atom_data.Potassium41
  - alkali_atom_data.Rubidium85
  - alkali_atom_data.Rubidium87
  - alkali_atom_data.Caesium

  Alkali atom type whose levels we want to examine

**drawLevels()**

Draws a level diagram plot
**makeLevels** (nFrom, nTo, lFrom, lTo, sList=[0.5])

Constructs energy level diagram in a given range

**Parameters**

- **nFrom** (*int*) – minimal principal quantum number of the states we are interested in
- **nTo** (*int*) – maximal principal quantum number of the states we are interested in
- **lFrom** (*int*) – minimal orbital angular momentum of the states we are interested in
- **lTo** (*int*) – maximal orbital angular momentum of the states we are interested in
- **sList** (*float*) – optional, spin angular momentum. Default value of [0.5] corresponds to Alkali atoms. For Alkaline Earth it has to be specified. For divalent atoms one can plot either one spin state by setting for example sList=[0]', or both spin states sList=[0,1]'

**showPlot()**

Shows a level diagram plot

**class** arc.calculations_atom_single.OpticalLattice1D (atom, trapWavenegth)

Atom properties in optical lattices in 1D

**Parameters**

- **atom** – one of AlkaliAtom or DivalentAtom
- **trapWavenegth** (*float*) – wavelength of trapping laser light (in units of m)

**BlochWavefunction** (trapPotentialDepth, quasimomentum, blochBandIndex)

Bloch wavefunction as a function of 1D coordinate.

**Example**

Returns Bloch wavefunction. Use as following:

```
trapPotentialDepth = 40  # units of recoil energy
quasimomentum = 0
blochBandIndex = 0  # Bloch band lowest in energy is 0
wf = lattice.BlochWavefunction(trapPotentialDepth, quasimomentum, blochBandIndex)
wf(x)  # returns complex number corresponding to value of Bloch
   # wavefunction at point x (coordinate given in units of
   # 1/k where k = 2 \pi / trapWavenegth )
   # by default k=1, so one full wavelength is 2\pi
```

**Parameters**

- **trapPotentialDepth** (*float*) – (in units of recoil energy
  OpticalLattice1D.getRecoilEnergy)
- **quasimomentum** (*float*) – (in units of 2 pi / OpticalLattice1D.
  trapWavenegth; note that reciprocal lattice momentum in this units is 2, and that
  full range of quasimomentum is from -1 to +1)

**Returns** Bloch wavefunction as a function of coordinate (see call example above)

**defineBasis** (lLimit=35)

Define basis for Bloch band calculations

Bloch states are calculated suming up all relevant states with momenta in range [-lLimit * 4 * pi /trapWavenegth, +lLimit * 4 * pi /trapWavenegth] Note that factor of 4 occurs since potential lattice period is twice the trapWavelength for standing wave.
**Parameters**

**lLimit (integer)** – Optional, defines maximal momentum to be taken for calculation of Bloch States as \( lLimit \times 4 \times \pi / \text{trapWavenegth} \). By default set to 35.

**diagonalise (trapPotentialDepth, quasimomentumList, saveBandIndex=None)**

Calculates energy levels (Bloch bands) for given quasimomentumList

Energy levels and their quasimomentum are saved in internal variables `energy` and `quasimomentum`. Energies are saved in units of recoil energy, and quasimomentum in units of The optional parameter `saveBandIndex` specifies index of the Bloch band for which eigenvectors should be saved. If provided, eigenvectors for each value `quasimomentumList[i]` are saved in `savedBlochBand[i]`.

**Parameters**

- **latticePotential (float)** – lattice depth formed by the standing wave of laser, with wavelength specified during initialisation of the lattice (in units of recoil energy).

- **quasimomentumList (array)** – array of quasimomentum values for which energy levels will be calculated (in units of \( \hbar \cdot k \), where \( k \) is trapping laser wavevector; since reciprocal lattice has twice the trapping laser wavevector due to standing wave, full range of quasimomentum is from -1 to +1)

- **saveBandIndex (int)** – optional, default None. If provided, specifies for which Bloch band should the eigenvectors be also saved. `saveBlochBand=0` corresponds to lowest energy band.

```python
energy = []
energy of states obtained by `OpticalLattice1D.diagonalise` method in format `[[energies for quasimomentum1], [energies for quasimomentum2], ...]`
```

**getRecoilEnergy ()**

Recoil energy for atoms in given optical lattice

**Returns** recoil energy in units of J

**getTrappingFrequency (trapPotentialDepth)**

Atom’s trapping frequency for given trapth depth

**Parameters** **trapPotentialDepth (float)** – lattice depth (in units of J)

**Returns** trapping frequency (in Hz)

**getWannierFunction (x, latticeIndex=0, k=1)**

Gives value at coordinate x of a Wannier function localized at given lattice index.

**Parameters**

- **x (float)** – spatial coordinate (in units of \( 2\pi / k \); for default value of laser driving wavevector \( k = 1 \), one trappingWavelength is \( 2\pi \)). Coordinate origin is at `latticeIndex=0`.

- **latticeIndex (int)** – optional, lattice index at which the Wannier function is localised. By default 0.

- **k (float)** – optional; laser driving wavevector, defines unit of length. Default value is 1, making one trapping laser wavelength equal to \( 2\pi \)

**plotLevelDiagram ()**

Plots energy level diagram (Bloch bands).

Based on diagonalisation of the lattice potential, plots discrete eigen energy spectra obtained for each value of the quasimomentum used in `OpticalLattice1D.diagonalise` method.

**Returns** matplotlib figure with a Bloch bands
quasimomentum = []

list of quasimomentum for which the energies of states was calculated by OpticalLattice1D.
diagonalise method in format [quasimomentum1, quasimomentum2, ...]

savedBlochBand = []

list of saved eigen energy state compositions for each of the Calculated quasimomentums for the se-
lected index of the Bloch band in OpticalLattice1D.diagonalise method in format [[eigen
state decomposition for quasimomentum 1], [eigen state decomposition for quasimomentum 2], ...]

trapPotentialDepth = 0

save slattice trap potential depth for which calculation OpticalLattice1D.diagonalise was
done

class arc.calculations_atom_single.StarkMap(atom)

Calculates Stark maps for single atom in a field

This initializes calculation for the atom of a given type. For details of calculation see Zimmerman1. For a
quick working example see Stark map example snippet.

Parameters atom (AlkaliAtom) =

alkali_atom_data.Lithium6,

alkali_atom_data.Lithium7,

alkali_atom_data.Potassium39,

alkali_atom_data.Potassium40,

alkali_atom_data.Potassium41,

alkali_atom_data.Rubidium85,

alkali_atom_data.Rubidium87, alkali_atom_data.Caesium ) Select the
alkali metal for energy level diagram calculation

Examples

State 28 S_{1/2} |m_j| = 0.5 polarizability calculation

```python
>>> from arc import *
>>> calc = StarkMap(Caesium())
>>> calc.defineBasis(28, 0, 0.5, 0.5, 23, 32, 20)
>>> calc.diagonalise(np.linspace(00.,6000,600))
>>> print("%.5f MHz cm^2 / V^2 " % calc.getPolarizability())
0.76705 MHz cm^2 / V^2
```

Stark map calculation

```python
>>> from arc import *
>>> calc = StarkMap(Caesium())
>>> calc.defineBasis(28, 0, 0.5, 0.5, 23, 32, 20)
>>> calc.diagonalise(np.linspace(00.,60000,600))
>>> calc.plotLevelDiagram()
>>> calc.showPlot()

<< matplotlib plot will open containing a Stark map >>
```

Examples

Advanced interfacing of Stark map calculations (StarkMap class) Here we show one easy way to
obtain the Stark matrix (from diagonal mat1 and off-diagonal part mat2 ) and basis states (stored in
basisStates ), if this middle-product of the calculation is needed for some code build on top of the
existing ARC package.

```python
>>> from arc import *
>>> calc = StarkMap(Caesium())
>>> calc.defineBasis(28, 0, 0.5, 0.5, 23, 32, 20)
>>> # Now we have matrix and basis states, that we can used in our own code
```

>>> # Let's say we want Stark map at electric field of 0.2 V/m
>>> eField = 0.2  # V/m
>>> # We can easily extract Stark matrix
>>> # as diagonal matrix (state detunings)
>>> # + off-diagonal matrix (proportional to electric field)
>>> matrix = calc.mat1+calc.mat2*eField
>>> # and the basis states as array [ [n,l,j,mj] , ...]
>>> basisStates = calc.basisStates
>>> # you can do your own calculation now...

References

basisStates = None
List of basis states for calculation in the form [ [n,l,j,mj], ...]. Calculated by defineBasis.

defineBasis(n, l, j, mj, nMin, nMax, maxL, Bz=0, progressOutput=False, debugOutput=False, s=0.5)
Initializes basis of states around state of interest

Defines basis of states for further calculation. n, l, j, mj specify state whose neighbourhood and polarization we want to explore. Other parameters specify basis of calculations. This method stores basis in basisStates, while corresponding interaction matrix is stored in two parts. First part is diagonal electric-field independent part stored in mat1, while the second part mat2 corresponds to off-diagonal elements that are proportional to electric field. Overall interaction matrix for electric field eField can be then obtained as fullStarkMatrix = mat1 + mat2 * eField

Parameters

• n (int) – principal quantum number of the state
• l (int) – angular orbital momentum of the state
• j (float) – total angular momentum of the state
• mj (float) – projection of total angular momentum of the state
• nMin (int) – minimal principal quantum number of the states to be included in the basis for calculation
• nMax (int) – maximal principal quantum number of the states to be included in the basis for calculation
• maxL (int) – maximal value of orbital angular momentum for the states to be included in the basis for calculation
• Bz (float) – optional, magnetic field directed along z-axis in units of Tesla. Calculation will be correct only for weak magnetic fields, where paramagnetic term is much stronger then diamagnetic term. Diamagnetic term is neglected.
• progressOutput (bool, optional) – if True prints the progress of calculation; Set to false by default.
• debugOutput (bool, optional) – if True prints additional information usefull for debugging. Set to false by default.
• s (float) – optional. Total spin angular momentum for the state. Default value of 0.5 is correct for Alkaline Atoms, but value has to be specified explicitly for divalent atoms (e.g. s=0 or s=1 for singlet and triplet states, that have total spin angular momentum equal to 0 or 1 respectively).

diagonalise(eFieldList, drivingFromState=[0, 0, 0, 0], progressOutput=False, debugOutput=False)
Finds atom eigenstates in a given electric field
Eigenstates are calculated for a list of given electric fields. To extract polarizability of the originally stated state see `getPolarizability` method. Results are saved in `eFieldList`, `y` and `highlight`.

**Parameters**

- `eFieldList (array)` – array of electric field strength (in V/m) for which we want to know energy eigenstates
- `progressOutput (bool, optional)` – if True prints the progress of calculation; Set to false by default.
- `debugOutput (bool, optional)` – if True prints additional information useful for debugging. Set to false by default.

**eFieldList = None**

Saves electric field (in units of V/m) for which energy levels are calculated

See also:

`y`, `highlight`, `diagonalise`

**exportData (fileBase, exportFormat='csv')**

Exports StarkMap calculation data.

Only supported format (selected by default) is .csv in a human-readable form with a header that saves details of calculation. Function saves three files: 1) `filebase_eField.csv`; 2) `filebase_energyLevels`; 3) `filebase_highlight`

For more details on the format, see header of the saved files.

**Parameters**

- `filebase (string)` – filebase for the names of the saved files without format extension. Add as a prefix a directory path if necessary (e.g. saving outside the current working directory)
- `exportFormat (string)` – optional. Format of the exported file. Currently only .csv is supported but this can be extended in the future.

**getPolarizability (maxField=10000000000.0, showPlot=False, debugOutput=False, minStateContribution=0.0)***

Returns the polarizability of the state (set during the initialization process).

Fits offset of the energy level of the state to \( \frac{1}{2} \alpha_0 E^2 \), where \( E \) is the applied static electric field, and returns fitted value \( \alpha_0 \)

**Parameters**

- `maxField (float, optional)` – maximum field (in V/m) to be used for fitting the polarizability. By default, max field is very large, so it will use eigenvalues calculated in the whole range.
- `showPlot (bool, optional)` – shows plot of calculated eigenValues of the given state (dots), and the fit (solid line) for extracting polarizability
- `debugOutput (bool, optional)` – if True prints additional information useful for debugging. Set to false by default.

**Returns** scalar polarizability in units of MHz cm \(^2\) / V \(^2\)

**Return type** `float`

**getState (state, electricField, minN, maxN, maxL, accountForAmplitude=0.95, debugOutput=False)**

Returns basis states and coefficients that make up for a given electric field the eigenstate with largest contribution of the original state.

**Parameters**
• **state** (*array*) – target basis state in format \([n, \ell, j, m_j]\) corresponding to the state whose composition we want to track as we apply the electric field

• **electricField** (*float*) – applied DC electric field in units of V/m.

• **minN** (*int*) – minimal principal quantum number to be taken for calculation of the Stark mixing

• **maxN** (*int*) – maximal principal quantum number to be take for calculation of the Stark mixing

• **maxL** (*int*) – maximal orbital angular momentum of states that should be taken in calculation of the Stark mixing

• **accountForAmplitude** (*float*) – optional, relative amplitude of state that should be reached with the subset of the eigen states returned. The returned eigen states will be sorted in the declining relative contribution to the final eigen state, and once total accounted amplitude of the state reaches 0.95, further output of additional small contribution of the other basis states to the final states will be suppressed. Default value of 0.95 will force output until basis state accounts for 95% of the state amplitude.

• **debugOutput** (*bool*) – optional, prints additional debug information if True. Default False.

**Returns** array of states in format \([n_1, \ell_1, j_1, m_{j_1}], \ldots\] and array of complex coefficients in format \([c_1, c_2, \ldots]\) corresponding the projections of the eigenstate (that has largest contribution of the original state in the given electric field) on the basis states, and energy of the found state in (eV)

highlight = None
highlight[i][j] is an array of values measuring highlighted feature in the eigenstates at electric field intensity eFieldList[i]. E.g. highlight[i][j] measures highlighted feature of the state with energy y[i][j] at electric field eFieldList[i]. What will be highlighted feature is defined in the call of diagonalise (see that part of documentation for details).

See also:
eFieldList, y, diagonalise

indexOfCoupledState = None
Index of coupled state (initial state passed to defineBasis) in basisStates list of basis states

mat1 = None
diagonal elements of Stark-matrix (detuning of states) calculated by defineBasis in the basis basisStates.

mat2 = None
off-diagonal elements of Stark-matrix divided by electric field value. To get off diagonal elements multiply this matrix with electric field value. Full Stark matrix is obtained as fullStarkMatrix = mat1 + mat2 * eField. Calculated by defineBasis in the basis basisStates.

plotLevelDiagram(*units=1, highlightState=True, progressOutput=False, debugOutput=False, highlightColour='red', addToExistingPlot=False*)
Makes a plot of a stark map of energy levels
To save this plot, see savePlot. To print this plot see showPlot.

**Parameters**

• **units** (*int, optional*) – possible values \([1, 2]\) ; if the value is 1 (default) Stark diagram will be plotted in energy units cm \(^{-1}\); if value is 2, Stark diagram will be plotted as energy \(/ h\) in units of GHz

• **highlightState** (*bool, optional*) – False by default. If True, scatter plot colour map will map in red amount of original state for the given eigenState

• **progressOutput** (*bool, optional*) – if True prints the progress of calculation; Set to False by default.
- **debugOutput** (`bool`, optional) – if True prints additional information useful for debugging. Set to False by default.

- **addToExistingPlot** (`bool`, optional) – if True adds points to existing old plot. Note that then interactive plotting doesn’t work. False by default.

```python
s = None
```

Spin manifold in which we are working default value of 0.5 is correct for Alkaline Atoms. Otherwise it has to be specified when calling `defineBasis` as `s=0` or `s=1` for singlet and triplet states respectively.

```python
savePlot(filename='StarkMap.pdf')
```

Saves plot made by `plotLevelDiagram`

**Parameters**

- **filename** (`str`, optional) – file location where the plot should be saved

```python
showPlot(interactive=True)
```

Shows plot made by `plotLevelDiagram`

```python
y = None
```

`y[i]` is an array of eigenValues corresponding to the energies of the atom states at the electric field `eFieldList[i]`. For example `y[i][j]` is energy of the `j` eigenvalue (energy of the state) measured in cm$^{-1}$ relative to the ionization threshold.

**See also:**

- `eFieldList`, `highlight`, `diagonalise`

```python
class arc.calculations_atom_single.Wavefunction(atom, basisStates, coefficients)
```

Calculates and plots electron wavefunctions.

**Parameters**

- **atom** – atom type considered (for example `Rubidum87()`)

- **basisStates** (`array`) – array of states in fine basis that contribute to the state whose wavefunction is requested. `[n_1, l_1, j_1, m_j_1, ...]`. For efficient calculation do not pass all the possible basis states, but just the ones that have significant contribution to the requested state.

- **coefficients** (`array`) – array `[c_1, ...]` of complex coefficients $c_i = \langle \psi_i | \psi \rangle$ corresponding to decomposition of required state $|\psi\rangle$ on basis states $|\psi_i\rangle$.

```python
getPsi(x, y, z)
```

Calculates list of $\psi_{m_s}(x, y, z)$

At point define by Cartesian coordinates returns list of $\psi_{m_s}(x, y, z)$ wavefunction values corresponding to different electron spin projection values $m_s$.

**Parameters**

- **x** (`float`) – Cartesian coordinates of selected point, relative to the atom core. (in atomic units of Bohr radius $a_0$)

- **y** (`float`) – Cartesian coordinates of selected point, relative to the atom core. (in atomic units of Bohr radius $a_0$)

- **z** (`float`) – Cartesian coordinates of selected point, relative to the atom core. (in atomic units of Bohr radius $a_0$)

**Returns** list of complex values corresponding to $\psi_{m_s}(\theta, \phi, r)$ for different spin states $m_s$ contributing to the state in decreasing order of $m_s$. For example, for `arc.AlkaliAtom` returns $\psi_{m_s=+1/2}(\theta, \phi, r)$ and $\psi_{m_s=-1/2}(\theta, \phi, r)$.

```python
getRtimesPsi(x, y, z)
```

Calculates list of $r \cdot \psi_{m_s}(x, y, z)$

At a point defined by Cartesian coordinates returns list of $r \cdot \psi_{m_s}(x, y, z)$ wavefunction values for different...
electron spin projection values $m_s$.

### Parameters

- **$x$** (float) – Cartesian coordinates of selected point, relative to the atom core. (in atomic units of Bohr radius $\alpha_0$)
- **$y$** (float) – Cartesian coordinates of selected point, relative to the atom core. (in atomic units of Bohr radius $\alpha_0$)
- **$z$** (float) – Cartesian coordinates of selected point, relative to the atom core. (in atomic units of Bohr radius $\alpha_0$)

### Returns

List of complex values corresponding to $r \cdot \psi_{m_s}(\theta, \phi, r)$ for different spin states $m_s$ contributing to the state in decreasing order of $m_s$. For example, for arc.AlkaliAtom returns $r \cdot \psi_{m_s=+1/2}(\theta, \phi, r)$ and $r \cdot \psi_{m_s=-1/2}(\theta, \phi, r)$.

### getRtimesPsiSpherical$(\theta, \phi, r)$

Calculates list of $r \cdot \psi_{m_s}(\theta, \phi, r)$ at point defined by spherical coordinates, returns list of $r \cdot \psi_{m_s}(\theta, \phi, r)$ wavefunction values for different electron spin projection values $m_s$.

Coordinates are defined relative to atomic core.

### Parameters

- **$\theta$** (float) – polar angle (angle between $z$ axis and vector pointing towards selected point) (in units of radians).
- **$\phi$** (float) – azimuthal angle (angle between $x$ axis and projection at $x-y$ plane of vector pointing towards selected point) (in units of radians).
- **$r$** (float) – distance between coordinate origin and selected point. (in atomic units of Bohr radius $\alpha_0$)

### Returns

List of complex values corresponding to $\psi_{m_s}(\theta, \phi, r)$ for different spin states $m_s$ contributing to the state in decreasing order of $m_s$. For example, for arc.AlkaliAtom returns $r \cdot \psi_{m_s=+1/2}(\theta, \phi, r)$ and $r \cdot \psi_{m_s=-1/2}(\theta, \phi, r)$.

### getRtimesPsiSquaredInPlane$(plane='x-z', \text{pointsPerAxis}=150, \text{axisLength}=None, \text{units}=\text{atomic}')$

Calculates $|r \cdot \psi|^2$ on a mesh in a given plane.

### Parameters

- **$plane$** (str) – optiona, set’s calculation plane to ‘x-y’ or ‘x-z’. Default value ‘x-y’
- **pointsPerAxis** (int) – optional, a number of mesh points per Carthesian axis. Default value of 150, gives a mesh with total size of $150 \times 150 = 22500$ points.
- **axisLength** (float) – optional, length of the square in the selected plane on which wavefunction will be calculated. By default it is large enough to fit the whole wavefunction (in atomic units of Bohr radius $\alpha_0$).
- **units** (str) – optional, units of length in which calculated mesh will be returned (note that axisLength is on the other hand always in atomi units.). Supported values are ‘atomic’ or ‘nm’. Default value ‘atomic’.

### Returns

MeshCoordinate1, MeshCoordinate2 and $|r \cdot \psi|^2 = \sum_{m_s} |r \cdot \psi_{m_s}|^2$, where sum is over possible electron spin projection values $m_s$.

### plot2D$(plane='x-z', \text{pointsPerAxis}=150, \text{axisLength}=None, \text{units}=\text{atomic}', \text{colorbar}=True, \text{labels}=True)$

2D colour plot of $|r \cdot \psi|^2$ wavefunction in a requested plane.

### Parameters

- **$plane$** (str) – optiona, set’s calculation plane to ‘x-y’ or ‘x-z’. Default value ‘x-y’
• **pointsPerAxis** (*int*) – optional, a number of mesh points per Carthesian axis. Default value of 150, gives a mesh with total size of 150 \( \times 150 = 22500 \) points.

• **axisLength** (*float*) – optional, length of the square in the selected plane on which wavefunction will be calculated. By default it is large enough to fit the whole wavefunction (in atomic units of Bohr radius \( a_0 \)).

• **units** (*str*) – optional, units of length in which calculated mesh will be returned (note that axisLength is on the other hand always in atomi units.). Supported values are ‘atomic’ or ‘nm’. Default value ‘atomic’.

• **colorbar** (*bool*) – optional, determines if the colour bar scale of should be shown. Default value is `True`.

• **labels** (*bool*) – optional, determines if the labels on the axis of the plot should be shown. Default value is `True`.

Returns `matplotlib.pyplot.figure` object with a requested plot. Use `show()` method to see figure.

```python
def plot3D(plane='x-z', pointsPerAxis=150, axisLength=None, units='atomic', labels=True):
    3D colour surface plot of \(|r \cdot \psi|^2\) wavefunction in a requested plane.
```

Parameters

• **plane** (*str*) – optional, set’s calculation plane to ‘x-y’ or ‘x-z’. Default value ‘x-y’

• **pointsPerAxis** (*int*) – optional, a number of mesh points per Carthesian axis. Default value of 150, gives a mesh with total size of 150 \( \times 150 = 22500 \) points.

• **axisLength** (*float*) – optional, length of the square in the selected plane on which wavefunction will be calculated. By default it is large enough to fit the whole wavefunction (in atomic units of Bohr radius \( a_0 \)).

• **units** (*str*) – optional, units of length in which calculated mesh will be returned (note that axisLength is on the other hand always in atomi units.). Supported values are ‘atomic’ or ‘nm’. Default value ‘atomic’.

• **labels** (*bool*) – optional, determines if the labels on the axis of the plot should be shown. Default value is `True`.

Returns `matplotlib.figure` object with a requested plot. Use `show()` method to see figure.

```python
class arc.materials.Air:
    Bases: arc.materials.OpticalMaterial
    Air as an optical material at normal conditions
    
    getN(vacuumWavelength=None)
        Assumes temperature: 15 °C, pressure: 101325 Pa
```

```python
class arc.materials.OpticalMaterial:
    Bases: object
    
    Abstract class implementing calculation of basic properties for optical materials.
    
    getN()
        Refractive index of material
    
    name = ''
        Human-friendly name of material
    
    sources = []
        List of .csv files listing refractive index measurements first column in these files is wavelength (in mu m), the second refractive index
    
    sourcesComment = []
        Any notes about measured values
```

1.3. Detailed documentation of functions 59
sourcesRange = []

Array of max and minimal wavelength pairs [lambdaMin, lambdaMax] for each of the sources. Automatically loaded from sources list

class arc.materials.Sapphire

Bases: arc.materials.OpticalMaterial

Sapphire as optical material.

getN(vacuumWavelength=None, airWavelength=None, axis='ordinary')

1.3.6 Pair-state basis calculations

Preliminaries

Relative orientation of the two atoms can be described with polar angle $\theta$ (range $0 - \pi$) and azimuthal angle $\phi$ (range $0 - 2\pi$). The $\hat{z}$ axis is here specified relative to the laser driving. For circularly polarized laser light, this is the direction of laser beam propagation. For linearly polarized light, this is the plane of the electric field polarization, perpendicular to the laser direction.

Internal coupling between the two atoms in $|n, l, j, m1\rangle$ and $|nn, ll, jj, m2\rangle$ is calculated easily for the two atoms positioned so that $\theta = 0$, and for other angles wignerD matrices are used to change a basis and perform calculation in the basis where couplings are more clearly seen.

Overview

PairStateInteractions Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>defineBasis(theta, ...)</td>
<td>Finds relevant states in the vicinity of the given pair-state</td>
</tr>
<tr>
<td>getC6perturbatively(...)</td>
<td>Calculates $C_6$ from second order perturbation theory.</td>
</tr>
<tr>
<td>getLeRoyRadius()</td>
<td>Returns Le Roy radius for initial pair-state.</td>
</tr>
<tr>
<td>diagonalise(rangeR, ...)</td>
<td>Finds eigenstates in atom pair basis.</td>
</tr>
<tr>
<td>plotLevelDiagram(...)</td>
<td>Plots pair state level diagram</td>
</tr>
<tr>
<td>showPlot([interactive])</td>
<td>Shows level diagram printed by PairStateInteractions.plotLevelDiagram</td>
</tr>
<tr>
<td>exportData(fileBase[, ...])</td>
<td>Exports PairStateInteractions calculation data.</td>
</tr>
<tr>
<td>getC6fromLevelDiagram(...)</td>
<td>Finds $C_6$ coefficient for original pair state.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 14 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PairStateInteractions.getC3fromLevelDiagram</td>
<td>Finds $C_3$ coefficient for original pair state.</td>
</tr>
<tr>
<td>PairStateInteractions.getVdwFromLevelDiagram</td>
<td>Finds $r_{vdW}$ coefficient for original pair state.</td>
</tr>
</tbody>
</table>

StarkMapResonances Methods

- StarkMapResonances.findResonances(nMin, ...) Finds near-resonant dipole-coupled pair-states
- StarkMapResonances.showPlot([interactive]) Plots initial state Stark map and its dipole-coupled resonances

Detailed documentation

**Pair-state basis level diagram calculations**

Calculates Rydberg spaghetti of level diagrams, as well as pertubative C6 and similar properties. It also allows calculation of Foster resonances tuned by DC electric fields.

**Example**

Calculation of the Rydberg eigenstates in pair-state basis for Rubidium in the vicinity of the $|60 S_{1/2} m_j = 1/2, 60 S_{1/2} m_j = 1/2\rangle$ state. Colour highlights coupling strength from state $6 F_{1/2} m_j = 1/2$ with $\pi (q = 0)$ polarized light. Eigenstates:

```python
from arc import *
calc1 = PairStateInteractions(Rubidium(), 60, 0, 0.5, 60, 0, 0.5, 0.5, 0.5, 0.5)
calc1.defineBasis(0., 0., 4, 5, 10e9)
# optionally we can save now results of calculation for future use
saveCalculation(calc1, "mycalculation.pkl")
calc11.diagonalise(linspace(1,10.0,30),250,progressOutput = True,
    drivingFromState=[6,1,0.5,0.5,0])
calc1.plotLevelDiagram()
calc1.ax.set_xlim(1,10)
calc1.ax.set_ylim(-2,2)
calc1.showPlot()
```

**class arc.calculations_atom_pairstate.PairStateInteractions (atom, n, l, j, nn, ll, jj, m1, m2, interactionsUpTo=1, s=0.5, s2=None, atom2=None)**

Calculates Rydberg level diagram (spaghetti) for the given pair state

Initializes Rydberg level spaghetti calculation for the given atom in the vicinity of the given pair state. For details of calculation see Ref. 1. For a quick start point example see interactions example snippet.

**Parameters**


1.3. Detailed documentation of functions 61
Select the alkali metal for energy level diagram calculation

- \( n (\text{int}) \) – principal quantum number for the \textbf{first} atom
- \( l (\text{int}) \) – orbital angular momentum for the \textbf{first} atom
- \( j (\text{float}) \) – total angular momentum for the \textbf{first} atom
- \( n_2 (\text{int}) \) – principal quantum number for the \textbf{second} atom
- \( l_2 (\text{int}) \) – orbital angular momentum for the \textbf{second} atom
- \( j_2 (\text{float}) \) – total angular momentum for the \textbf{second} atom
- \( m_1 (\text{float}) \) – projection of the total angular momentum on z-axis for the \textbf{first} atom
- \( m_2 (\text{float}) \) – projection of the total angular momentum on z-axis for the \textbf{second} atom
- \( \text{interactionsUpTo} (\text{int}) \) – Optional. If set to 1, includes only dipole-dipole interactions. If set to 2 includes interactions up to quadrupole-quadrupole. Default value is 1.
- \( s_2 (\text{float}) \) – optional, spin state of the second atom. If not specified (left to default value None) it will assume spin state of the first atom.
- \( \text{atom2} (\text{AlkaliAtom or DivalentAtom}) \) – optional, specifies atomic species for the second atom, enabling calculation of \textbf{inter-species pair-state interactions}. If not specified (left to default value None) it will assume spin state of the first atom.

References

Examples

Advanced interfacing of pair-state is2=None, atom2=None interactions calculations (PairStateInteractions class). This is an advanced example intended for building up extensions to the existing code. If you want to directly access the pair-state interaction matrix, constructed by \texttt{defineBasis}, you can assemble it easily from diagonal part (stored in \texttt{matDiagonal}) and off-diagonal matrices whose spatial dependence is \( R^{-3}, R^{-4}, R^{-5} \) stored in that order in \texttt{matR}. Basis states are stored in \texttt{basisStates} array.

```
>>> from arc import *
>>> calc = PairStateInteractions(Rubidium(), 60,0,0.5, 60,0,0.5, interactionsUpTo = 1)
>>> # theta=0, phi = 0, range of pqn, range of l, deltaE = 25e9
>>> calc.defineBasis(0 ,0 , 5, 5, 25e9, progressOutput=True)
>>> # now calc stores interaction matrix and relevant basis
>>> # we can access this directly and generate interaction matrix
>>> # at distance rval :
>>> rval = 4 # in mum
>>> matrix = calc.matDiagonal
>>> rX = (rval*1.e-6)**3
>>> for matRX in self.matR:
>>>     matrix = matrix + matRX/rX
>>> rX *= (rval*1.e-6)
>>> # matrix variable now holds full interaction matrix for
>>> # interacting atoms at distance rval calculated in
>>> # pair-state basis states can be accessed as
>>> basisStates = calc.basisStates

atom1 = None
atom type

basisStates = None
List of pair-states for calculation. In the form \([n1,l1,j1,mj1,n2,l2,j2,mj2], \ldots \). Each state is an
array \([n_{1,1}, j_{1,1}, m_{j_{1,1}}, n_{2,1}, j_{2,1}, m_{j_{2,1}}]\) corresponding to \(|n_1, l_1, j_1, m_j, n_2, l_2, j_2, m_j\rangle\) state. Calculated by \texttt{defineBasis}.

\texttt{channel = None}

states relevant for calculation, defined in J basis (not resolving \(m_j\)). Used as intermediary for full interaction matrix calculation by \texttt{defineBasis}.

\texttt{coupling = None}

List of matrices defining coupling strengths between the states in J basis (not resolving \(m_j\)). Basis is given by \texttt{channel}. Used as intermediary for full interaction matrix calculation by \texttt{defineBasis}.

\texttt{defineBasis(theta, phi, nRange, lrange, energyDelta, Bz=0, progressOutput=False, debugOutput=False)}

Finds relevant states in the vicinity of the given pair-state

Finds relevant pair-state basis and calculates interaction matrix. Pair-state basis is saved in \texttt{basisStates}. Interaction matrix is saved in parts depending on the scaling with distance. Diagonal elements \texttt{matDiagonal}, corresponding to relative energy defects of the pair-states, don’t change with interatomic separation. Off diagonal elements can depend on distance as \(R^{-3}, R^{-4}\) or \(R^{-5}\), corresponding to dipole-dipole (\(C_3\)), dipole-quadrupole (\(C_4\)) and quadrupole-quadrupole coupling (\(C_5\)) respectively. These parts of the matrix are stored in \texttt{matR} in that order. I.e. \texttt{matR[0]} stores dipole-dipole coupling (\(\propto R^{-3}\)), \texttt{matR[0]} stores dipole-quadrupole couplings etc.

Parameters

- \texttt{theta (float)} – relative orientation of the two atoms (see figure on top of the page), range 0 to \(\pi\)
- \texttt{phi (float)} – relative orientation of the two atoms (see figure on top of the page), range 0 to \(2\pi\)
- \texttt{nRange (int)} – how much below and above the given principal quantum number of the pair state we should be looking?
- \texttt{lrange (int)} – what is the maximum angular orbital momentum state that we are including in calculation
- \texttt{energyDelta (float)} – what is maximum energy difference (\(\Delta E/h\) in Hz) between the original pair state and the other pair states that we are including in calculation
- \texttt{Bz (float)} – optional, magnetic field directed along z-axis in units of Tesla. Calculation will be correct only for weak magnetic fields, where paramagnetic term is much stronger then diamagnetic term. Diamagnetic term is neglected.
- \texttt{progressOutput (bool)} – optional, False by default. If true, prints information about the progress of the calculation.
- \texttt{debugOutput (bool)} – optional, False by default. If true, similarly to \texttt{progressOutput=True}, this will print information about the progress of calculations, but with more verbose output.

See also:

\texttt{alkali_atom_functions.saveCalculation} and \texttt{alkali_atom_functions.loadSavedCalculation} for information on saving intermediate results of calculation for later use.

\texttt{diagonalise(rangeR, noOfEigenVectors, drivingFromState=[0, 0, 0, 0], eigenstateDetuning=0.0, sortEigenvectors=False, progressOutput=False, debugOutput=False)}

Finds eigenstates in atom pair basis.

\texttt{ARPACK (scipy.sparse.linalg.eigsh)} calculation of the \texttt{noOfEigenVectors} eigenvectors closest to the original state. If \texttt{drivingFromState} is specified as \([n, l, j, m_j]\) coupling between the pair-states and the situation where one of the atoms in the pair state basis is in \(|n, l, j, m_j\rangle\) state due to driving with a laser field that drives \(q\) transition (+1,0,-1 for \(\sigma^-, \pi\) and \(\sigma^+\) transitions respectively) is calculated and marked by the colourmapping these values on the obtained eigenvectors.

1.3. Detailed documentation of functions 63
Parameters

- **rangeR** (array) – Array of values for distance between the atoms (in μ m) for which we want to calculate eigenstates.

- **noOfEigenvectors** (int) – number of eigen vectors closest to the energy of the original (unperturbed) pair state. Has to be smaller then the total number of states.

- **eigenstateDetuning** (float, optional) – Default is 0. This specifies detuning from the initial pair-state (in Hz) around which we want to find noOfEigenvectors eigenvectors. This is useful when looking only for couple of off-resonant features.

- **drivingFromState** ([int, int, float, float, int]) – Optional. State of one of the atoms from the original pair-state basis from which we try to drive to the excited pair-basis manifold, assuming that the first of the two atoms is already excited to the specified Rydberg state. By default, program will calculate just contribution of the original pair-state in the eigenstates obtained by diagonalization, and will highlight it’s admixture by colour mapping the obtained eigenstates plot. State is specified as [n, ℓ, j, mj, d] where d is +1, 0 or -1 for driving σ−, π and σ+ transitions respectively.

- **sortEigenvectors** (bool) – optional, False by default. Tries to sort eigenvectors so that given eigenvector index corresponds to adiabatically changing eigenstate, as determined by maximising overlap between old and new eigenvectors.

- **progressOutput** (bool) – optional, False by default. If true, prints information about the progress of the calculation.

- **debugOutput** (bool) – optional, False by default. If true, similarly to progressOutput=True, this will print information about the progress of calculations, but with more verbose output.

**exportData** (fileBase, exportFormat='csv')

Exports PairStateInteractions calculation data.

Only supported format (selected by default) is .csv in a human-readable form with a header that saves details of calculation. Function saves three files: 1) filebase _r.csv; 2) filebase _energyLevels 3) filebase _highlight

For more details on the format, see header of the saved files.

**getC3fromLevelDiagram** (rStart, rStop, showPlot=False, minStateContribution=0.0, resonantBranch=1)

Finds C3 coefficient for original pair state.

Function first finds for each distance in the range [rStart, rStop] the eigen state with highest contribution of the original state. One can set optional parameter minStateContribution to value in the range [0,1), so that function finds only states if they have contribution of the original state that is bigger then minStateContribution.

Once original pair-state is found in the range of interatomic distances, from smallest rStart to the biggest rStop, function will try to perform fitting of the corresponding state energy E(R) at distance R to the function A + C3/R3 where A is some offset.

Parameters

- **rStart** (float) – smallest inter-atomic distance to be used for fitting

- **rStop** (float) – maximum inter-atomic distance to be used for fitting
• **showPlot** *(bool)* – If set to true, it will print the plot showing fitted energy level and the obtained best fit. Default is False

• **minStateContribution** *(float)* – valid values are in the range [0,1). It specifies minimum amount of the original state in the given energy state necessary for the state to be considered for the adiabatic continuation of the original unperturbed pair state.

• **resonantBranch** *(int)* – optional, default +1. For resonant interactions we have two branches with identical state contributions. In this case, we will select only positively detuned branch (for resonantBranch = +1) or negatively detuned branch (for resonantBranch = -1) depending on the value of resonantBranch optional parameter

**Returns** $C_3$ measured in GHz $\mu$m$^6$ on success; If unsuccessful returns False.

**Return type** float

---

**Note:** In order to use this functions, highlighting in *diagonalise* should be based on the original pair state contribution of the eigenvectors (that this, *drivingFromState* parameter should not be set, which corresponds to *drivingFromState* = [0,0,0,0,0]).

---

**getC6fromLevelDiagram** *(rStart, rStop, showPlot=False, minStateContribution=0.0)*

Finds $C_6$ coefficient for original pair state.

Function first finds for each distance in the range [rStart, rStop] the eigen state with highest contribution of the original state. One can set optional parameter *minStateContribution* to value in the range [0,1), so that function finds only states if they have contribution of the original state that is bigger then minStateContribution.

Once original pair-state is found in the range of interatomic distances, from smallest rStart to the biggest rStop, function will try to perform fitting of the corresponding state energy $E(R)$ at distance $R$ to the function $A + C_6/R^6$ where $A$ is some offset.

**Parameters**

• **rStart** *(float)* – smallest inter-atomic distance to be used for fitting

• **rStop** *(float)* – maximum inter-atomic distance to be used for fitting

• **showPlot** *(bool)* – If set to true, it will print the plot showing fitted energy level and the obtained best fit. Default is False

• **minStateContribution** *(float)* – valid values are in the range [0,1). It specifies minimum amount of the original state in the given energy state necessary for the state to be considered for the adiabatic continuation of the original unperturbed pair state.

**Returns** $C_6$ measured in GHz $\mu$m$^6$ on success; If unsuccessful returns False.

**Return type** float

---

**Note:** In order to use this functions, highlighting in *diagonalise* should be based on the original pair state contribution of the eigenvectors (that this, *drivingFromState* parameter should not be set, which corresponds to *drivingFromState* = [0,0,0,0,0]).

---

**getC6perturbatively** *(theta, phi, nRange, energyDelta, degeneratePertubation=False)*

Calculates $C_6$ from second order perturbation theory.

Calculates $C_6 = \sum_{r',r''} |\langle r',r''|V| r_1,r_2 \rangle|^2 / \Delta_{r',r''}$, where $\Delta_{r',r''} \equiv E(r',v'') - E(r_1,r_2)$

This calculation is faster then full diagonalization, but it is valid only far from the so called spaghetti region that occurs when atoms are close to each other. In that region multiple levels are strongly
coupled, and one needs to use full diagonalization. In region where perturbative calculation is correct, energy level shift can be obtained as $V(R) = -C_6/R^6$

See perturbative C6 calculations example snippet.

**Parameters**

- **theta** *(float)* – orientation of inter-atomic axis with respect to quantization axis ($z$) in Euler coordinates (measured in units of radian)
- **phi** *(float)* – orientation of inter-atomic axis with respect to quantization axis ($z$) in Euler coordinates (measured in units of radian)
- **nRange** *(int)* – how much below and above the given principal quantum number of the pair state we should be looking
- **energyDelta** *(float)* – what is maximum energy difference ($\Delta E/\hbar$ in Hz) between the original pair state and the other pair states that we are including in calculation
- **degeneratePertubation** *(bool)* – optional, default False. Should one use degenerate pertubation theory. This should be used whenever angle between quantisation and interatomic axis is non-zero, as well as when one considers non-stretched states.

**Returns** if degeneratePertubation=False, returns $C_6$ measured in GHz $\mu m^6$; if degeneratePertubation=True, returns array of $C_6$ measured in GHz $\mu m^6$ AND array of corresponding eigenvectors in $m_{j_1} = -j_1, \ldots, m_{j_1} = +j_1 \otimes m_{j_2} = -j_2, \ldots, m_{j_2} = +j_2$ basis

**Return type** float

**Example**

If we want to quickly calculate $C_6$ for two Rubidium atoms in state $62D_{3/2}m_j = \frac{3}{2}$, positioned in space along the shared quantization axis:

```python
from arc import *
calculation = PairStateInteractions(Rubidium(), 62, 2, 1.5, 62, 2, 1.5, 1.5)
c6 = calculation.getC6perturbatively(0,0, 5, 25e9)
print "C_6 = %.0f GHz (mu m)^6" % c6
```

Which returns:

```
C_6 = 767 GHz (mu m)^6
```

Quick calculation of angular anisotropy of for Rubidium $D_{2/5}, m_j = \frac{5}{2}$ states:

```python
# Rb 60 D_{2/5}, mj=2.5 , 60 D_{2/5}, mj=2.5 pair state
calculation1 = PairStateInteractions(Rubidium(), 60, 2, 2.5, 60, 2, 2.5, 2.5)
# list of atom orientations
thetalist = np.linspace(0,pi,30)
# do calculation of C6 pertubatively for all atom orientations
c6 = []
for theta in thetalist:
    value = calculation1.getC6perturbatively(theta,0,5,25e9)
    c6.append(value)
print ("theta = %.2f * pi $\mu m^6$ % (theta/pi,value))
```

(continues on next page)
ylabel(r"$C_6$ (GHz $\mu$m$^6$)")

show()

getLeRoyRadius()

Returns Le Roy radius for initial pair-state.

Le Roy radius is defined as $2\langle r_1^2 \rangle^{1/2} + 2\langle r_2^2 \rangle^{1/2}$, where $r_1$ and $r_2$ are electron coordinates for the first and the second atom in the initial pair-state. Below this radius, calculations are not valid since electron wavefunctions start to overlap.

- Returns Le Roy radius measured in $\mu$m
- Return type float

References

getVdwFromLevelDiagram(rStart, rStop, showPlot=False, minStateContribution=0.0)

Finds $r_{vdW}$ coefficient for original pair state.

Function first finds for each distance in the range [rStart, rStop] the eigen state with highest contribution of the original state. One can set optional parameter minStateContribution to value in the range [0,1), so that function finds only states if they have contribution of the original state that is bigger than minStateContribution.

Once original pair-state is found in the range of interatomic distances, from smallest rStart to the biggest rStop, function will try to perform fitting of the corresponding state energy $E(R)$ at distance $R$ to the function $A + B \frac{1-\sqrt{1+\frac{r_{vdW}}{r}}}{1-\sqrt{1+r_{vdW}^6}}$

where $A$ and $B$ are some offset.

- Parameters
  - rStart (float) – smallest inter-atomic distance to be used for fitting
  - rStop (float) – maximum inter-atomic distance to be used for fitting
  - showPlot (bool) – If set to true, it will print the plot showing fitted energy level and the obtained best fit. Default is False
  - minStateContribution (float) – valid values are in the range [0,1). It specifies minimum amount of the original state in the given energy state necessary for the state to be considered for the adiabatic continuation of the original unperturbed pair state.

- Returns $r_{vdW}$ measured in $\mu$m on success; If unsuccessful returns False.
- Return type float

Note: In order to use this functions, highlighting in diagonalise should be based on the original pair state contribution of the eigenvectors (that this, drivingFromState parameter should not be set, which corresponds to drivingFromState = [0,0,0,0]).

interactionsUpTo = None

" Specifies up to which approximation we include in pair-state interactions. By default value is 1, corresponding to pair-state interactions up to dipole-dipole coupling. Value of 2 is also supported, corresponding to pair-state interactions up to quadrupole-quadrupole coupling.


1.3. Detailed documentation of functions 67
matDiagonal = None
Part of interaction matrix in pair-state basis that doesn’t depend on inter-atomic distance. E.g. diagonal elements of the interaction matrix, that describe energies of the pair states in unperturbed basis, will be stored here. Basis states are stored in basisStates. Calculated by defineBasis.

matR = None
Stores interaction matrices in pair-state basis that scale as $1/R^3$, $1/R^4$ and $1/R^5$ with distance in matR[0], matR[1] and matR[2] respectively. These matrices correspond to dipole-dipole ($C_3$), dipole-quadrupole ($C_4$) and quadrupole-quadrupole ($C_5$) interactions coefficients. Basis states are stored in basisStates. Calculated by defineBasis.

matrixElement = None
matrixElement[i] gives index of state in channel basis (that doesn’t resolve $m_j$ states), for the given index $i$ of the state in basisStates ($m_j$ resolving) basis.

originalPairStateIndex = None
index of the original n,l,j,m1,n,l,j,m2 pair-state in the basisStates basis.

plotLevelDiagram (highlightColor='red', highlightScale='linear')
Plots pair state level diagram
Call showPlot if you want to display a plot afterwards.

Parameters

• **highlightColor** *(string)* – optional, specifies the colour used for state highlighting

• **highlightScale** *(string)* – optional, specifies scaling of state highlighting. Default is ‘linear’. Use ‘log-2’ or ‘log-3’ for logarithmic scale going down to 1e-2 and 1e-3 respectively. Logarithmic scale is useful for spotting weakly admixed states.

s1 = None
total spin angular momentum, optional (default 0.5)

savePlot (filename='PairStateInteractions.pdf')
Saves plot made by plotLevelDiagram

Parameters filename *(str, optional)* – file location where the plot should be saved

showPlot (interactive=True)
Shows level diagram printed by PairStateInteractions.plotLevelDiagram
By default, it will output interactive plot, which means that clicking on the state will show the composition of the clicked state in original basis (dominant elements)

Parameters interactive *(bool)* – optional, by default it is True. If true, plotted graph will be interactive, i.e. users can click on the state to identify the state composition

Note: interactive=True has effect if the graphs are explored in usual matplotlib pop-up windows. It doesn’t have effect on inline plots in Jupyter (IPython) notebooks.

class arc.calculations_atom_pairstate.StarkMapResonances (atom1, state1, atom2, state2)
Calculates pair state Stark maps for finding resonances
Tool for finding conditions for Foster resonances. For a given pair state, in a given range of the electric fields, looks for the pair-state that are close in energy and coupled via dipole-dipole interactions to the original pair-state.

See Stark resonances example snippet.

Parameters

• **atom** *(AlkaliAtom)* – =

  alkali_atom_data.Lithium6,
  alkali_atom_data.Lithium7,
  alkali_atom_data.Sodium,
the first atom in the pair-state

- **state1**((int, int, float, float, (float))) – specification of the state of the first state as an array of values \([n, l, j, m_j]\). For obj: `DivalentAtom` and other divalent atoms, 5th value should be added specifying total spin angular momentum \(s\). Full definition of state then has format \([n, l, j, m_j, s]\).


- **state2**((int, int, float, float, (float))) – specification of the state of the first state as an array of values \([n, l, j, m_j]\). For obj: `DivalentAtom` and other divalent atoms, 5th value should be added specifying total spin angular momentum \(s\). Full definition of state then has format \([n, l, j, m_j, s]\).

**FindResonances**\(\text{findResonances}(nMin, nMax, maxL, eFieldList, energyRange=[-5000000000.0, 5000000000.0], Bz=0, progressOutput=False)\)

Finds near-resonant dipole-coupled pair-states

For states in range of principal quantum numbers \([nMin, nMax]\) and orbital angular momentum \([0, maxL]\), for a range of electric fields given by `eFieldList` function will find near-resonant pair states.

Only states that are in the range given by `energyRange` will be extracted from the pair-state Stark maps.

**Parameters**

- **nMin** (int) – minimal principal quantum number of the state to be included in the StarkMap calculation
- **nMax** (int) – maximal principal quantum number of the state to be included in the StarkMap calculation
- **maxL** (int) – maximum value of orbital angular momentum for the states to be included in the calculation
- **eFieldList**([(float)]) – list of the electric fields (in V/m) for which to calculate level diagram (StarkMap)
- **Bz** (float) – optional, magnetic field directed along z-axis in units of Tesla. Calculation will be correct only for weak magnetic fields, where paramagnetic term is much stronger then diamagnetic term. Diamagnetic term is neglected.
- **energyRange**([(float, float)]) – optional argument. Minimal and maximal energy of that some dipole-coupled state should have in order to keep it in the plot (in units of Hz). By default it finds states that are ±5 GHz
- **progressOutput** (bool, optional) – if True prints the progress of calculation; Set to false by default.

**ShowPlot**\(\text{showPlot}(\text{interactive}=True)\)

Plots initial state Stark map and its dipole-coupled resonances
Parameters interactive *(optional, bool)* – if True (by default) points on plot will be clickable so that one can find the state labels and their composition (if they are heavily admixed).

**Note:** Zero is given by the initial states of the atom given in initialisation of calculations, calculated in absence of magnetic field B_z. In other words, for non-zero magnetic field the initial states will have offset from zero even for zero electric field due to Zeeman shift.

### 1.3.7 Advanced functions

Below is the list of extension modules for advanced ARC calculations, contributed by the research community. When using advanced extension modules for ARC please cite *both* original ARC paper *and* paper that introduced extension.

**Overview**

<table>
<thead>
<tr>
<th>getPopulationLifetime(atom, n, l, j, [\ldots])</th>
<th>Calculates lifetime of atomic population taking into account redistribution of population to other states under spontaneous and black body induced transitions.</th>
</tr>
</thead>
</table>

**Detailed documentation**

```
arc.advanced.population_lifetime.getPopulationLifetime(atom, n, l, j, 
              temperature=0, 
              includeLevelsUpTo=0, 
              period=1, 
              plotting=1, 
              thresholdState=False, 
              detailedOutput=False)
```

Calculates lifetime of atomic population taking into account redistribution of population to other states under spontaneous and black body induced transitions.

It simulates the time evolution of a system in which all the states, from the fundamental one to the highest state which you want to include, are taken into account. The orbital angular momenta taken into account are only S,P,D,F.

This function is based on getStateLifetime but it takes into account the re-population processes due to BBR-induced transitions. For this reason lifetimes of Rydberg states are slightly longer than those returned by getStateLifetime up to 5-10%.
This function creates a .txt file, plots the time evolution of the population of the Rydberg states and yields the lifetime values by using the fitting method from Ref. 1.

**Contributed by:** Alessandro Greco (alessandrogreco08 at gmail dot com), Dipartimento di Fisica E. Fermi, Università di Pisa, Largo Bruno Pontecorvo 3, 56127 Pisa, Italy. The simulations have been compared with experimental data2.

**Please cite as:** original ARC paper and paper introducing extension2

References:

Some definitions:

What are the ensemble, the support, the ground? According to https://arxiv.org/abs/1907.01254 The sum of the populations of every state which is detected as Rydberg state (above the threshold state which must be set) is called ensemble The sum of the populations of every state which is detected as Rydberg state, without the target state, is called support The sum of the populations of every state which cannot be detected as Rydberg state (under the threshold state which must be set) is called ground

\[ \gamma_{\text{Target Spont}} \] is the rate which describes transitions from Target State towards all the levels under the threshold state, i.e. Ground State \[ \gamma_{\text{Target BBR}} \] is the rate which describes transitions towards all the levels above the threshold state, i.e. Support State \[ \gamma_{\text{Support Spont}} \] is the rate which describes transitions from the Support State towards all the levels under the threshold state, i.e. Ground State \[ \gamma_{\text{Support BBR}} \] is the rate which describes transitions from the Support State towards all the levels above the threshold state, i.e. Support State

Parameters

- \( n (\text{int}) \) – principal quantum number of the state whose population lifetime we are calculating, it’s called the Target state and its color is green in the plot.
- \( l (\text{int}) \) – orbital angular momentum number of the state whose population lifetime we are calculating, it’s called the Target state and its color is green in the plot.
- \( j (\text{float}) \) – total angular momentum of the state whose population lifetime we are calculating, it’s called the Target state and its color is green in the plot.
- \( \text{temperature} (\text{float}) \) – Temperature at which the atom environment is, measured in K. If this parameter is non-zero, user has to specify transitions up to which state (due to black-body decay) should be included in calculation.
- \( \text{includeLevelsUpTo} (\text{int}) \) – At non zero temperatures, this specifies maximum principal quantum number of the state to which black-body induced transitions will be included. Minimal value of the parameter in that case is \( \text{n}+1 \)
- \( \text{period} \) – Specifies the period that you want to consider for the time evolution, in microseconds.
- \( \text{plotting} (\text{int}) \) – optional. It is set to 1 by default. The options are (see also image at the bottom of documentation): \( \text{plotting}=0 \) no plot; \( \text{plotting}=1 \) plots the population of the target \((n,l,j)\) state with its fit and it yields the value of the target lifetime in microseconds; \( \text{plotting}=2 \) plots the whole system (Ensemble, Support, Target), no fit; \( \text{plotting}=3 \) plots the whole system (Ensemble, Support, Target) and it fits the Ensemble and Target curves, it yields the values of the Ensemble lifetime and Target lifetime in microseconds; \( \text{plotting}=4 \) it plots the whole system (Ensemble, Support, Target) + the Ground (which is the complementary of the ensemble). It considers the whole system like a three-level model (Ground State, Support State, Target State) and yields four transition rates.

---

1 https://people.duke.edu/~ccc14/sta-663/CalibratingODEs.html
• **thresholdState** *(int)* – optional. It specifies the principal quantum number $n$ of the lowest state (it’s referred to S state!) which is detectable by your experimental apparatus, it directly modifies the Ensemble and the Support (whose colors are red and blue respectively in the plot). It is necessary to define a threshold state if plotting = 2, 3 or 4 has been selected. It is not necessary to define a threshold state if plotting = 0 or 1 has been selected.

• **detailedOutput=True** – optional. It writes a .txt file with the time evolution of all the states. It is set to false by default. (The first column is the time, the other are the population of all the states. The order is time, nS, nP0.5, nP1.5, nD1.5, nD2.5, nF2.5, nF3.5, and n is ordered from the lowest state to the highest one. For example: time, 4S, 5S, 6S, ecc... includeLevelsUpToS, 4P0.5, 5P0.5, 6P0.5, ecc... includeLevelsUpToP0.5, 4P1.5, 5P1.5, 6P1.5, ecc... )

**Returns** Plots and a .txt file. **plotting = 0,1** create a .txt file with two columns (time target population); **plotting = 2,3,4** create a .txt file with four columns (time ensemble population support population target population)

**Example**

```python
>>> from arc import *
>>> from arc.advanced.population_lifetime import getPopulationLifetime
>>> atom = Rubidium()
>>> getPopulationLifetime(atom, 10, 1, 1.5, temperature = 300, includeLevelsUpTo=15, detailedOutput=True, plotting=1)
```

![Fig. 1: Examples of possible outputs of getPopulationLifetime for different plotting parameter settings.](image)

**1.4 How to contribute to the project**

Both new data and calculations for the core library are welcome. We will also include codes that solve specific research questions in the arc.advanced. Ideally, this package/library will grow into a community project, as a
community-maintained resource for atomic physics community. Full code is accessible from GitHub, so please fork the project, and submit improvements, additional modules, new features, or just suggest ideas. For inspiration, we have a list of possible features for development below.

1.4.1 Ideas for development

This is incomplete list of some of the modules that can be added to the library:

- Photoionization
- Collisional cross-sections
- Dressing potentials
- ... (add your own ideas)

1.4.2 Naming conventions

For the sake of consistency, readability and cross-linking with the written literature, please follow the following for contributions:

- Names and method/subdivision should reflect **structure of knowledge in atomic physics**, NOT low-level implementation structure.
- Names should be sensible to atomic physicists (even if they are not familiar with coding).
- Use long self-descriptive variable names (so that the code is self-documented and readable in itself) and write short comment on functions of code subsections.
- Use **Google style docstrings for code documentation** (we are using Sphinx Napoleon extension for generating documentation)
- Add references to original papers in comments and docstrings.

Finally, this is the naming convention of the original package. For consistency, we suggest following the same naming convention.

- Submodules are lower case, separated by underscore. Example:

  ```
  import my_module_name
  ```

- Classes are named in CamelCase, for example:

  ```
  class MyNewClass:
  ... 
  ```

- Class methods that return a value start with get in their name, and follow camelCase convention, for example:

  ```
  def getSumOfTwoNumbers(a,b):
      return a+b
  ```

- Class methods don’t return a value are named in camelCase, for example:

  ```
  def defineBasis():
  ... 
  ```

**Note:** Support for divalent atoms and some of the functions (**Wavefunction, AtomSurfaceVdW, OpticalLattice1D, DynamicPolarizability**, and optical materials properties) in this modules will be released in the forthcoming ARC 3.0.0 version. To used them now as a beta feature do:
from arc.beta import *

Publication describing this upgrade is in preparation (check this place soon). For now cite as “E. J. Robertson, N. Šibalić, R. M. Potvliege and M. P. A. Jones, in preparation (2020)".
Fig. 1: Overview of modules and interdependencies in the arc package. Click on image to enlarge.
CHAPTER 3

Indices and tables

• genindex
• modindex
• search
CHAPTER 4

Credits

Authors  Nikola Šibalić, Jonathan D. Pritchard, Charles S. Adams, Kevin J. Weatherill
Licence  BSD 3-Clause
Version  3.0.0 of 2020/01/17
a

arc.advanced.population_lifetime, 70
arc.alkali_atom_data, 24
arc.alkali_atom_functions, 7
arc.calculations_atom_pairstate, 61
arc.calculations_atom_single, 48
arc.divalent_atom_data, 44
arc.divalent_atom_functions, 33
arc.materials, 59
arc.wigner, 22
A
a1 (arc.alkali_atom_data.Caesium attribute), 24
a1 (arc.alkali_atom_data.Lithium6 attribute), 25
a1 (arc.alkali_atom_data.Lithium7 attribute), 26
a1 (arc.alkali_atom_data.Potassium39 attribute), 27
a1 (arc.alkali_atom_data.Potassium40 attribute), 27
a1 (arc.alkali_atom_data.Potassium41 attribute), 28
a1 (arc.alkali_atom_data.Rubidium85 attribute), 29
a1 (arc.alkali_atom_data.Rubidium87 attribute), 30
a1 (arc.alkali_atom_data.Sodium attribute), 31
a1 (arc.alkali_atom_functions.AlkaliAtom attribute), 7
a2 (arc.alkali_atom_data.Caesium attribute), 24
a2 (arc.alkali_atom_data.Lithium6 attribute), 25
a2 (arc.alkali_atom_data.Lithium7 attribute), 26
a2 (arc.alkali_atom_data.Potassium39 attribute), 27
a2 (arc.alkali_atom_data.Potassium40 attribute), 28
a2 (arc.alkali_atom_data.Potassium41 attribute), 28
a2 (arc.alkali_atom_data.Rubidium85 attribute), 29
a2 (arc.alkali_atom_data.Rubidium87 attribute), 30
a2 (arc.alkali_atom_data.Sodium attribute), 31
a2 (arc.alkali_atom_functions.AlkaliAtom attribute), 7
a3 (arc.alkali_atom_data.Caesium attribute), 24
a3 (arc.alkali_atom_data.Lithium6 attribute), 25
a3 (arc.alkali_atom_data.Lithium7 attribute), 26
a3 (arc.alkali_atom_data.Potassium39 attribute), 27
a3 (arc.alkali_atom_data.Potassium40 attribute), 28
a3 (arc.alkali_atom_data.Potassium41 attribute), 28
a3 (arc.alkali_atom_data.Rubidium85 attribute), 29
a3 (arc.alkali_atom_data.Rubidium87 attribute), 30
a3 (arc.alkali_atom_data.Sodium attribute), 31
a3 (arc.alkali_atom_functions.AlkaliAtom attribute), 7
a4 (arc.alkali_atom_data.Caesium attribute), 24
a4 (arc.alkali_atom_data.Lithium6 attribute), 26
a4 (arc.alkali_atom_data.Lithium7 attribute), 26
a4 (arc.alkali_atom_data.Potassium39 attribute), 27
a4 (arc.alkali_atom_data.Potassium40 attribute), 28
a4 (arc.alkali_atom_data.Potassium41 attribute), 28
a4 (arc.alkali_atom_data.Rubidium85 attribute), 29
a4 (arc.alkali_atom_data.Rubidium87 attribute), 30
a4 (arc.alkali_atom_data.Sodium attribute), 31
a4 (arc.alkali_atom_functions.AlkaliAtom attribute), 8
abundance (arc.alkali_atom_data.Lithium6 attribute), 26
abundance (arc.alkali_atom_data.Lithium7 attribute), 26
abundance (arc.alkali_atom_data.Potassium39 attribute), 27
abundance (arc.alkali_atom_data.Potassium40 attribute), 28
abundance (arc.alkali_atom_data.Potassium41 attribute), 28
abundance (arc.alkali_atom_data.Rubidium85 attribute), 29
abundance (arc.alkali_atom_data.Rubidium87 attribute), 30
abundance (arc.alkali_atom_data.Sodium attribute), 31
abundance (arc.alkali_atom_functions.AlkaliAtom attribute), 8
Air (class in arc.materials), 59
AlkaliAtom (class in arc.alkali_atom_functions), 7
alphaC (arc.alkali_atom_data.Caesium attribute), 24
alphaC (arc.alkali_atom_data.Lithium6 attribute), 26
alphaC (arc.alkali_atom_data.Lithium7 attribute), 26
alphaC (arc.alkali_atom_data.Potassium39 attribute), 27
alphaC (arc.alkali_atom_data.Potassium40 attribute), 28
alphaC (arc.alkali_atom_data.Potassium41 attribute), 28
alphaC (arc.alkali_atom_data.Rubidium85 attribute), 29
alphaC (arc.alkali_atom_data.Rubidium87 attribute), 30
alphaC (arc.alkali_atom_data.Sodium attribute), 31
alphaC (arc.alkali_atom_functions.AlkaliAtom attribute), 8
arc.advanced.population_lifetime (module), 70
arc.alkali_atom_data (module), 24
arc.alkali_atom_functions (module), 7
arc.calculations_atom_pairstate (module), 61
arc.calculations_atom_single (module), 48
arc.divalent_atom_data (module), 44
arc.divalent_atom_functions (module), 33
arc.materials (module), 39

Index
arc.wigner  (module), 22
dipoleMatrixElementFile  (arc.alkali_atom_functions.AlkaliAtom attribute), 8
atom1  (arc.calculations_atom_pairstate.PairStateInteractions attribute), 62
DivalentAtom  (class in arc.divalent_atom_functions), 33
drawLevels()  (arc.calculations_atom_single.LevelPlot method), 50
D
defectFittingRange  (arc.divalent_atom_functions.Calcium40 attribute), 44
defectFittingRange  (arc.divalent_atom_functions.Strontium88 attribute), 45
defectFittingRange  (arc.divalent_atom_functions.Ytterbium174 attribute), 46
defectFittingRange  (arc.divalent_atom_functions.DivalentAtom attribute), 33
defineBasis()  (arc.calculations_atom_pairstate.PairStateInteractions method), 63
defineBasis()  (arc.calculations_atom_single.DynamicPolarizability method), 49
defineBasis()  (arc.calculations_atom_single.OpticalLattice1D method), 51
defineBasis()  (arc.calculations_atom_single.StarkMap method), 54
diagonalise()  (arc.calculations_atom_pairstate.PairStateInteractions method), 63
diagonalise()  (arc.calculations_atom_single.OpticalLattice1D method), 52
diagonalise()  (arc.calculations_atom_single.StarkMap method), 54

C
Caesium  (class in arc.alkali_atom_data), 24
Calcium40  (class in arc.divalent_atom_data), 44
Cesium  (class in arc.alkali_atom_data), 25
cg()  (in module arc.wigner), 22
couple()  (arc.calculations_atom_pairstate.PairStateInteractions method), 63
corePotential()  (arc.alkali_atom_functions.AlkaliAtom method), 8
corePotential()  (arc.divalent_atom_functions.DivalentAtom method), 33
cp_numerov  (arc.alkali_atom_functions.AlkaliAtom attribute), 8
elementName  (arc.alkali_atom_functions.AlkaliAtom attribute), 5
energy  (arc.calculations_atom_single.OpticalLattice1D attribute), 52
energyLevelsExtrapolated  (arc.divalent_atom_functions.DivalentAtom attribute), 33
exportData()  (arc.calculations_atom_pairstate.PairStateInteractions method), 64
extraLevels  (arc.alkali_atom_data.Caesium attribute), 24
extraLevels  (arc.alkali_atom_data.Potassium39 attribute), 27
extraLevels  (arc.alkali_atom_data.Potassium40 attribute), 28
extraLevels  (arc.alkali_atom_data.Potassium41 attribute), 28
extraLevels  (arc.alkali_atom_data.Rubidium85 attribute), 29
extraLevels  (arc.alkali_atom_data.Rubidium87 attribute), 30
extraLevels  (arc.alkali_atom_functions.AlkaliAtom attribute), 8
extraLevels  (arc.divalent_atom_data.Calcium40 attribute), 44
extraLevels  (arc.divalent_atom_data.Ytterbium174 attribute), 46

B
AtomSurfaceVdW  (class in arc.calculations_atom_single), 48
BasisStates  (arc.calculations_atom_pairstate.PairStateInteractions attribute), 62
BasisStates  (arc.calculations_atom_single.StarkMap attribute), 54
BlochWavefunction()  (arc.calculations_atom_single.OpticalLattice1D method), 51

E
effectiveCharge()  (arc.alkali_atom_functions.AlkaliAtom method), 8
effectiveCharge()  (arc.divalent_atom_functions.DivalentAtom method), 33
eFieldList  (arc.calculations_atom_single.StarkMap attribute), 55
extraLevels  (arc.divalent_atom_data.Calcium40 attribute), 44
extraLevels  (arc.divalent_atom_data.Ytterbium174 attribute), 46

G
findResonances()  (arc.calculations_atom_pairstate.StarkMapResonances method), 69
get()  (arc.wigner.WignerDmatrix method), 23

Index
getAverageInteratomicSpacing() (arc.alkali_atom_functions.AlkaliAtom method), 8
getAverageInteratomicSpacing() (arc.divalent_atom_functions.DivalentAtom method), 33
getAverageSpeed() (arc.alkali_atom_functions.AlkaliAtom method), 9
getAverageSpeed() (arc.divalent_atom_functions.DivalentAtom method), 33
getC3contribution() (arc.calculations_atom_single.AtomSurfaceVdW method), 48
getC3fromLevelDiagram() (arc.calculations_atom_pairstate.PairStateInteractions method), 64
getC3term() (arc.alkali_atom_functions.AlkaliAtom method), 9
getC3term() (arc.divalent_atom_functions.DivalentAtom method), 34
getC6fromLevelDiagram() (arc.calculations_atom_pairstate.PairStateInteractions method), 65
getC6perturbatively() (arc.calculations_atom_pairstate.PairStateInteractions method), 65
getC6term() (arc.alkali_atom_functions.AlkaliAtom method), 9
getC6term() (arc.divalent_atom_functions.DivalentAtom method), 34
getDipoleMatrixElement() (arc.alkali_atom_functions.AlkaliAtom method), 10
getDipoleMatrixElement() (arc.divalent_atom_functions.DivalentAtom method), 35
getDipoleMatrixElementHFS() (arc.alkali_atom_functions.AlkaliAtom method), 11
getDipoleMatrixElementHFS() (arc.divalent_atom_functions.DivalentAtom method), 36
getEnergy() (arc.alkali_atom_functions.AlkaliAtom method), 11
getEnergy() (arc.divalent_atom_functions.DivalentAtom method), 36
getEnergyDefect() (arc.alkali_atom_functions.AlkaliAtom method), 12
getEnergyDefect() (arc.divalent_atom_functions.DivalentAtom method), 36
getEnergyDefect2() (arc.alkali_atom_functions.AlkaliAtom method), 12
getEnergyDefect2() (arc.divalent_atom_functions.DivalentAtom method), 37
getLeRoyRadius() (arc.calculations_atom_pairstate.PairStateInteractions method), 67
getLiteratureDME() (arc.alkali_atom_functions.AlkaliAtom method), 12
getLiteratureDME() (arc.divalent_atom_functions.DivalentAtom method), 37
getN() (arc.materials.Air method), 59
getN() (arc.materials.OpticalMaterial method), 59
getN() (arc.materials.Sapphire method), 60
getNumberDensity() (arc.alkali_atom_functions.AlkaliAtom method), 13
getNumberDensity() (arc.divalent_atom_functions.DivalentAtom method), 38
getPolarizability() (arc.calculations_atom_single.DynamicPolarizability method), 50
getPolarizability() (arc.calculations_atom_single.StarkMap method), 55
getPressure() (arc.alkali_atom_data.Caesium method), 24
getPressure() (arc.alkali_atom_data.Lithium6 method), 26
getPressure() (arc.alkali_atom_data.Lithium7 method), 26
getPressure() (arc.alkali_atom_data.Potassium39 method), 27
getPressure() (arc.alkali_atom_data.Potassium40 method), 28
getPressure() (arc.alkali_atom_data.Potassium41 method), 29
getPressure() (arc.alkali_atom_data.Rubidium85 method), 30
getPressure() (arc.alkali_atom_data.Rubidium87 method), 31
getPressure() (arc.alkali_atom_data.Sodium method), 31
getPsi() (arc.calculations_atom_single.Wavefunction method), 57

Index 85
getQuadrupoleMatrixElement()  
  (arc.alkali_atom_functions.AlkaliAtom method), 14
getQuadrupoleMatrixElement()  
  (arc.divalent_atom_functions.DivalentAtom method), 39
getQuantumDefect()  
  (arc.alkali_atom_functions.AlkaliAtom method), 14
getQuantumDefect()  
  (arc.divalent_atom_functions.DivalentAtom method), 39
getRabiFrequency()  
  (arc.alkali_atom_functions.AlkaliAtom method), 14
getRabiFrequency()  
  (arc.divalent_atom_functions.DivalentAtom method), 39
getRabiFrequency2()  
  (arc.alkali_atom_functions.AlkaliAtom method), 15
getRabiFrequency2()  
  (arc.divalent_atom_functions.DivalentAtom method), 39
getRadialCoupling()  
  (arc.alkali_atom_functions.AlkaliAtom method), 15
getRadialCoupling()  
  (arc.divalent_atom_functions.DivalentAtom method), 40
getRadialMatrixElement()  
  (arc.alkali_atom_functions.AlkaliAtom method), 15
getRadialMatrixElement()  
  (arc.divalent_atom_functions.DivalentAtom method), 40
getRecoilEnergy()  
  (arc.calculations_atom_single.OpticalLattice1D method), 52
getReducedMatrixElementJ()  
  (arc.alkali_atom_functions.AlkaliAtom method), 15
getReducedMatrixElementJ()  
  (arc.divalent_atom_functions.DivalentAtom method), 40
getReducedMatrixElementJ_asymmetric()  
  (arc.alkali_atom_functions.AlkaliAtom method), 16
getReducedMatrixElementJ_asymmetric()  
  (arc.divalent_atom_functions.DivalentAtom method), 41
getReducedMatrixElementL()  
  (arc.alkali_atom_functions.AlkaliAtom method), 16
getReducedMatrixElementL()  
  (arc.divalent_atom_functions.DivalentAtom method), 41
getRtimesPsi()  
  (arc.calculations_atom_single.Wavefunction method), 57
getRtimesPsiSpherical()  
  (arc.calculations_atom_single.Wavefunction method), 58
getRtimesPsiSquaredInPlane()  
  (arc.calculations_atom_single.Wavefunction method), 58
getState()  
  (arc.calculations_atom_single.StarkMap method), 55
getStateC3()  
  (arc.calculations_atom_single.AtomSurfaceVdW method), 49
getStateLifetime()  
  (arc.alkali_atom_functions.AlkaliAtom method), 17
getStateLifetime()  
  (arc.divalent_atom_functions.DivalentAtom method), 42
getTransitionFrequency()  
  (arc.alkali_atom_functions.AlkaliAtom method), 17
getTransitionFrequency()  
  (arc.divalent_atom_functions.DivalentAtom method), 42
getTransitionRate()  
  (arc.alkali_atom_functions.AlkaliAtom method), 17
getTransitionRate()  
  (arc.divalent_atom_functions.DivalentAtom method), 42
getTransitionWavelength()  
  (arc.alkali_atom_functions.AlkaliAtom method), 18
getTransitionWavelength()  
  (arc.divalent_atom_functions.DivalentAtom method), 43
getTrappingFrequency()  
  (arc.calculations_atom_single.OpticalLattice1D method), 52
getVdwFromLevelDiagram()  
  (arc.calculations_atom_pairstate.PairStateInteractions method), 67
getWannierFunction()  
  (arc.calculations_atom_single.OpticalLattice1D method), 52
getZeemanEnergyShift()  
  (arc.alkali_atom_functions.AlkaliAtom method), 18
getZeemanEnergyShift()  
  (arc.divalent_atom_functions.DivalentAtom method), 43
groundStateN  
  (arc.alkali_atom_functions.AlkaliAtom attribute), 19
H highlight  
  (arc.calculations_atom_single.StarkMap attribute), 56
Hydrogen  
  (class in arc.alkali_atom_data), 25
I (arc.alkali_atom_functions.AlkaliAtom attribute), 7
indexOfCoupledState (arc.calculations_atom_single.StarkMap attribute), 56
interactionsUpTo (arc.calculations_atom_pairstate.PairStateInteractions attribute), 67
ionisationEnergy (arc.alkali_atom_data.Caesium attribute), 24
ionisationEnergy (arc.alkali_atom_data.Hydrogen attribute), 25
ionisationEnergy (arc.alkali_atom_data.Lithium7 attribute), 26
ionisationEnergy (arc.alkali_atom_data.Potassium39 attribute), 27
ionisationEnergy (arc.alkali_atom_data.Potassium40 attribute), 28
ionisationEnergy (arc.alkali_atom_data.Potassium41 attribute), 29
ionisationEnergy (arc.alkali_atom_data.Rubidium85 attribute), 30
ionisationEnergy (arc.alkali_atom_data.Rubidium87 attribute), 30
ionisationEnergy (arc.alkali_atom_data.Sodium attribute), 31
ionisationEnergy (arc.alkali_atom_functions.AlkaliAtom attribute), 19
ionisationEnergy (arc.divalent_atom_data.Calcium40 attribute), 45
ionisationEnergy (arc.divalent_atom_data.Strontium88 attribute), 45
ionisationEnergy (arc.divalent_atom_data.Ytterbium174 attribute), 46
levelDataFromNIST (arc.alkali_atom_functions.AlkaliAtom attribute), 19
levelDataFromNIST (arc.divalent_atom_data.Calcium40 attribute), 45
levelDataFromNIST (arc.divalent_atom_data.Strontium88 attribute), 45
levelDataFromNIST (arc.divalent_atom_data.Ytterbium174 attribute), 46
levelDataFromNIST (arc.divalent_atom_functions.DivalentAtom attribute), 44
LevelPlot (class in arc.calculations_atom_single), 50
literatureDMEfilename (arc.alkali_atom_functions.AlkaliAtom attribute), 19
Lithium6 (class in arc.alkali_atom_data), 25
Lithium7 (class in arc.alkali_atom_data), 26
loadSavedCalculation() (in module arc.alkali_atom_functions), 21
makeLevels() (arc.calculations_atom_single.LevelPlot method), 50
mass (arc.alkali_atom_data.Hydrogen attribute), 25
mass (arc.alkali_atom_data.Lithium6 attribute), 26
mass (arc.alkali_atom_data.Lithium7 attribute), 26
mass (arc.alkali_atom_data.Potassium39 attribute), 27
mass (arc.alkali_atom_data.Potassium40 attribute), 28
mass (arc.alkali_atom_data.Potassium41 attribute), 29
mass (arc.alkali_atom_data.Rubidium85 attribute), 30
mass (arc.alkali_atom_data.Rubidium87 attribute), 30
mass (arc.alkali_atom_data.Sodium attribute), 31
mass (arc.alkali_atom_functions.AlkaliAtom attribute), 19
mass (arc.divalent_atom_data.Calcium40 attribute), 45
mass (arc.divalent_atom_data.Strontium88 attribute), 45
mass (arc.divalent_atom_data.Ytterbium174 attribute), 46
mat1 (arc.calculations_atom_single.StarkMap attribute), 56
mat2 (arc.calculations_atom_single.StarkMap attribute), 56
matDiagonal (arc.calculations_atom_pairstate.PairStateInteractions attribute), 67
matR (arc.calculations_atom_pairstate.PairStateInteractions attribute), 68
matrixElement (arc.calculations_atom_pairstate.PairStateInteractions attribute), 68
meltingPoint (arc.alkali_atom_data.Caesium attribute), 24
meltingPoint (arc.alkali_atom_data.Lithium6 attribute), 26
meltingPoint (arc.alkali_atom_data.Lithium7 attribute), 26
meltingPoint (arc.alkali_atom_data.Potassium39 attribute), 27
meltingPoint (arc.alkali_atom_data.Potassium40 attribute), 28
meltingPoint (arc.alkali_atom_data.Potassium41 attribute), 29
meltingPoint (arc.alkali_atom_data.Rubidium85 attribute), 30
meltingPoint (arc.alkali_atom_data.Rubidium87 attribute), 30
meltingPoint ([arc.alkali_atom_data.Sodium attribute], 31
meltingPoint ([arc.alkali_atom_functions.AlkaliAtom method], 19
meltingPoint ([arc.divalent_atom_data.Calcium40 attribute], 45
meltingPoint ([arc.divalent_atom_data.Strontium88 attribute], 45
meltingPoint ([arc.divalent_atom_data.Ytterbium174 attribute], 46
minQuantumDefectN ([arc.alkali_atom_functions.AlkaliAtom attribute], 19
minQuantumDefectN ([arc.divalent_atom_functions.DivalentAtom attribute], 44
modelPotential_coef ([arc.divalent_atom_functions.DivalentAtom attribute], 44
N
name ([arc.materials.OpticalMaterial attribute], 59
NumerovBack () (in module [arc.alkali_atom_functions], 20
O
OpticalLattice1D (class in [arc.calculations_atom_single], 51
OpticalMaterial (class in [arc.materials], 59
originalPairStateIndex ([arc.calculations_atom_pairstate.PairStateInteractions attribute], 68
P
PairStateInteractions (class in [arc.calculations_atom_pairstate], 61
plot2D () ([arc.calculations_atom_single.Wavefunction method], 58
plot3D () ([arc.calculations_atom_single.Wavefunction method], 59
plotLevelDiagram () ([arc.calculations_atom_pairstate.PairStateInteractions attribute], 68
plotLevelDiagram () ([arc.calculations_atom_single.OpticalLattice1D method], 52
plotLevelDiagram () ([arc.calculations_atom_single.StarkMap method], 56
plotPolarizability () ([arc.calculations_atom_single.DynamicPolarizability method], 50
Potential (class in [arc.alkali_atom_data], 27
Potassium (class in [arc.alkali_atom_data], 27
Potassium39 (class in [arc.alkali_atom_data], 27
Potassium40 (class in [arc.alkali_atom_data], 27
Potassium41 (class in [arc.alkali_atom_data], 27
potential () ([arc.alkali_atom_data.Hydrogen method], 25
potential () ([arc.alkali_atom_functions.AlkaliAtom method], 19
potential () ([arc.divalent_atom_functions.DivalentAtom method], 44
printState () (in module [arc.alkali_atom_functions], 21
printStateString () (in module [arc.alkali_atom_functions], 21
printStateStringLatex () (in module [arc.alkali_atom_functions], 21
Q
quadrupoleMatrixElementFile ([arc.alkali_atom_functions.AlkaliAtom attribute], 19
quantumDefect ([arc.alkali_atom_data.Caesium attribute], 19
quantumDefect ([arc.alkali_atom_data.Lithium6 attribute], 26
quantumDefect ([arc.alkali_atom_data.Lithium7 attribute], 30
quantumDefect ([arc.alkali_atom_data.Potassium39 attribute], 27
quantumDefect ([arc.alkali_atom_data.Potassium40 attribute], 28
quantumDefect ([arc.alkali_atom_data.Potassium41 attribute], 29
quantumDefect ([arc.alkali_atom_data.Rubidium85 attribute], 30
quantumDefect ([arc.alkali_atom_data.Rubidium87 attribute], 30
quantumDefect ([arc.alkali_atom_data.Sodium attribute], 31
quantumDefect ([arc.alkali_atom_functions.AlkaliAtom method], 19
quantumDefect ([arc.divalent_atom_data.Calcium40 attribute], 45
quantumDefect ([arc.divalent_atom_data.Strontium88 attribute], 46
quantumDefect ([arc.divalent_atom_data.Ytterbium174 attribute], 46
quantumDefect ([arc.divalent_atom_functions.DivalentAtom attribute], 44
quasimomentum ([arc.calculations_atom_single.OpticalLattice1D attribute], 52
R
radialWavefunction () ([arc.alkali_atom_functions.AlkaliAtom method], 19
radialWavefunction () ([arc.divalent_atom_functions.DivalentAtom method], 44
rc ([arc.alkali_atom_data.Caesium attribute], 25
rc ([arc.alkali_atom_data.Lithium6 attribute], 26
rc ([arc.alkali_atom_data.Lithium7 attribute], 27
rc ([arc.alkali_atom_data.Potassium39 attribute], 27
rc ([arc.alkali_atom_data.Potassium40 attribute], 28
rc (arc.alkali_atom_data.Potassium41 attribute), 29
rc (arc.alkali_atom_data.Rubidium85 attribute), 30
rc (arc.alkali_atom_data.Rubidium87 attribute), 31
rc (arc.alkali_atom_functions.AlkaliAtom attribute), 20
Rubidium (class in arc.alkali_atom_data), 29
Rubidium85 (class in arc.alkali_atom_data), 29
Rubidium87 (class in arc.alkali_atom_data), 30

S
s (arc.calculations_atom_single.StarkMap attribute), 57
s1 (arc.calculations_atom_pairstate.PairStateInteractions attribute), 68
Sapphire (class in arc.materials), 60
saveCalculation() (in module arc.alkali_atom_functions), 21
savedBlochBand (arc.calculations_atom_single.OpticalLattice1D attribute), 53
savePlot() (arc.calculations_atom_pairstate.PairStateInteractions method), 68
savePlot() (arc.calculations_atom_single.StarkMap method), 57

T
trapPotentialDepth

U
updateDipoleMatrixElementsFile() (arc.alkali_atom_functions.AlkaliAtom method), 20
updateDipoleMatrixElementsFile() (arc.divalent_atom_functions.DivalentAtom method), 44

W
Wavefunction (class in arc.calculations_atom_single), 57
Wigner3j() (in module arc.wigner), 22
Wigner6j() (in module arc.wigner), 23
WignerDmatrix (class in arc.wigner), 23

Y
y (arc.calculations_atom_single.StarkMap attribute), 57
Ytterbium174 (class in arc.divalent_atom_data), 46

Z
Z (arc.alkali_atom_functions.AlkaliAtom attribute), 7