Alkali Rydberg Calculator (ARC) Documentation

Release 2.0.6

Nikola Šibalić

Jan 31, 2019

Contents

ARC (Alkali Rydberg Calculator) is 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 metal atoms.

CHAPTER 1

Contents

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 3.6**. 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 or Python 3.6. Note this is a new feature for Windows, so Windows users should contact developers if unexpected errors occur. **Below we describe some old (legacy) installation methods.**

1.1.3 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.**

1.1.4 Installation of the package globally with setup.py

This is tested on Linux so far

Do this only 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.

1.1.5 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 in order to compile Numpy extension for Python 2.7 under Windows. For other Python versions (3.5) find correct compiler here . 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 .

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.

Finally...

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

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

AlkaliAtom([preferQuantumDefects, cpp_numerov])	Implements general calculations for alkali atoms.
NumerovBack(innerLimit, outerLimit, kfun, ...)	Full Python implementation of Numerov integration
saveCalculation(calculation, fileName)	Saves calculation for future use.
loadSavedCalculation(fileName)	Loads previously saved calculation.
printState(n, l, j)	Prints state spectroscopic label for numeric n, l, s label of the state
printStateString(n, l, j)	Returns state spectroscopic label for numeric n, l, s label of the state

AlkaliAtom Methods

AlkaliAtom.getDipoleMatrixElement(n1, l1, ...)	Dipole matrix element $\langle n_1 l_1 j_1 m_{j_1} \vert e\mathbf{r} \vert n_2 l_2 j_2 m_{j_2} \rangle$ in units of $a_0 e$
AlkaliAtom.getTransitionWavelength(n1, l1, ...)	Calculated transition wavelength (in vacuum) in m.
AlkaliAtom.getTransitionFrequency(n1, l1, ...)	Calculated transition frequency in Hz
AlkaliAtom.getRabiFrequency(n1, l1, j1, mj1, ...)	Returns a Rabi frequency for resonantly driven atom in a center of TEM00 mode of a driving field
AlkaliAtom.getRabiFrequency2(n1, l1, j1, ...)	Returns a Rabi frequency for resonant excitation with a given electric field amplitude
AlkaliAtom.getStateLifetime(n, l, j[, ...])	Returns the lifetime of the state (in s)
AlkaliAtom.getTransitionRate(n1, l1, j1, n2, ...)	Transition rate due to coupling to vacuum modes (black body included)
AlkaliAtom.getReducedMatrixElementJ_asymmetric(n1, ...)	Reduced matrix element in J basis, defined in asymmetric notation.
AlkaliAtom.getReducedMatrixElementJ(n1, l1, ...)	Reduced matrix element in J basis (symmetric notation)
AlkaliAtom.getReducedMatrixElementL(n1, l1, ...)	Reduced matrix element in L basis (symmetric notation)
AlkaliAtom.getRadialMatrixElement(n1, l1, ...)	Radial part of the dipole matrix element
AlkaliAtom.getQuadrupoleMatrixElement(n1, ...)	Radial part of the quadrupole matrix element
AlkaliAtom.getPressure(temperature)	Vapour pressure (in Pa) at given temperature
AlkaliAtom.getNumberDensity(temperature)	Atom number density at given temperature
AlkaliAtom.getAverageInteratomicSpacing()	Returns average interatomic spacing in atomic vapour
AlkaliAtom.corePotential(l, r)	core potential felt by valence electron
AlkaliAtom.effectiveCharge(l, r)	effective charge of the core felt by valence electron
AlkaliAtom.potential(l, s, j, r)	returns total potential that electron feels
AlkaliAtom.radialWavefunction(l, s, j, ...)	Radial part of electron wavefunction
AlkaliAtom.getEnergy(n, l, j)	Energy of the level relative to the ionisation level (in eV)
AlkaliAtom.getZeemanEnergyShift(l, j, mj, ...)	Retuns linear (paramagnetic) Zeeman shift.
AlkaliAtom.getQuantumDefect(n, l, j)	Quantum defect of the level.
AlkaliAtom.getC6term(n, l, j, n1, l1, j1, ...)	C6 interaction term for the given two pair-states
AlkaliAtom.getC3term(n, l, j, n1, l1, j1, ...)	C3 interaction term for the given two pair-states
AlkaliAtom.getEnergyDefect(n, l, j, n1, l1, ...)	Energy defect for the given two pair-states (one of the state has two atoms in the same state)
AlkaliAtom.getEnergyDefect2(n, l, j, nn, ll, ...)	Energy defect for the given two pair-states

Continued on next page

Table 2 – continued from previous page

AlkaliAtom.updateDipoleMatrixElementsFile()	Updates the file with pre-calculated dipole matrix elements.
AlkaliAtom.getRadialCoupling(n, l, j, n1, l1, j1)	Returns radial part of the coupling between two states (dipole and quadrupole interactions only)
AlkaliAtom.getAverageSpeed(temperature)	Average (mean) speed at a given temperature
AlkaliAtom.getLiteratureDME(n1, l1, j1, n2, ...)	Returns literature information on requested transition.

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, quandrupole 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.

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
swich - 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*)
C3 interaction term for the given two pair-states

Calculates C_3 intaraction 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

[2] M. Marinescu, H. R. Sadeghpour, and A. Dalgarno PRA **49**, 982 (1994), https://doi.org/10.1103/PhysRevA.49.982

- **l** (*int*) – orbital angular momenutum
- **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

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

Return type float

getC6term(*n*, *l*, *j*, *n1*, *l1*, *j1*, *n2*, *l2*, *j2*)

C6 interaction term for the given two pair-states

Calculates C_6 intaraction 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.[3].

Parameters

- **n** (*int*) – principal quantum number
- **l** (*int*) – orbital angular momenutum
- **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

Returns $C_6 = \frac{1}{4\pi\varepsilon_0}\frac{|\langle n,l,j|er|n_1,l_1,j_1\rangle|^2|\langle n,l,j|er|n_2,l_2,j_2\rangle|^2}{E(n_1,l_1,j_2,n_2,j_2,j_2)-E(n,l,j,n,l,j)}$ (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:

```
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␣
↪))
```

(continues on next page)

[3] T. G. Walker, M. Saffman, PRA **77**, 032723 (2008) https://doi.org/10.1103/PhysRevA.77.032723

(continued from previous page)

```
print("\n = = = 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 (*n1*, *l1*, *j1*, *mj1*, *n2*, *l2*, *j2*, *mj2*, *q*)
Dipole matrix element $\langle n_1 l_1 j_1 m_{j_1} | e\mathbf{r} | n_2 l_2 j_2 m_{j_2} \rangle$ in units of $a_0 e$

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 σ^- transition:

```
from arc import *
atom = Rubidium()
# transition 5 S_{1/2} m_j=-0.5 -> 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))
```

getEnergy (*n*, *l*, *j*)
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

Returns state energy (eV)

Return type float

getEnergyDefect(*n*, *l*, *j*, *n1*, *l1*, *j1*, *n2*, *l2*, *j2*)

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 momenutum
- **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

Returns energy defect (SI units: J)

Return type float

getEnergyDefect2(*n*, *l*, *j*, *nn*, *ll*, *jj*, *n1*, *l1*, *j1*, *n2*, *l2*, *j2*)

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 momenutum
- **j** (*float*) – total angular momentum
- **nn** (*int*) – principal quantum number
- **ll** (*int*) – orbital angular momenutum
- **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

Returns energy defect (SI units: J)

Return type float

getLiteratureDME(*n1*, *l1*, *j1*, *n2*, *l2*, *j2*)

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

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. 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
- 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 will output 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*)
Radial part of the quadrupole matrix element

Calculates $\int \mathbf{d}r\ R_{n_1,l_1,j_1}(r) \cdot R_{n_1,l_1,j_1}(r) \cdot r^4$. 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

Returns quadrupole matrix element ($a_0^2 e$).

Return type float

getQuantumDefect(*n*, *l*, *j*)
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

Returns quantum defect

Return type float

getRabiFrequency(*n1*, *l1*, *j1*, *mj1*, *n2*, *l2*, *j2*, *q*, *laserPower*, *laserWaist*)
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 σ^-, π and σ^+ respectively)
- **laserPower** – laser power in units of W
- **laserWaist** – laser $1/e^2$ waist (radius) in units of m

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

Return type float

getRabiFrequency2(*n1*, *l1*, *j1*, *mj1*, *n2*, *l2*, *j2*, *q*, *electricFieldAmplitude*)
Returns a Rabi frequency for resonant excitation with a given electric field amplitude

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 σ^-, π and σ^+ respectively)
- **electricFieldAmplitude** – amplitude of electric field driving (V/m)

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

Return type float

getRadialCoupling(*n*, *l*, *j*, *n1*, *l1*, *j1*)

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

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*, *useLiterature=True*)

Radial part of the dipole matrix element

Calculates $\int \mathbf{d}r\ R_{n_1,l_1,j_1}(r) \cdot R_{n_1,l_1,j_1}(r) \cdot r^3$.

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

Returns dipole matrix element ($a_0 e$).

Return type float

getReducedMatrixElementJ(*n1*, *l1*, *j1*, *n2*, *l2*, *j2*)

Reduced matrix element in J basis (symmetric notation)

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

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

Return type float

getReducedMatrixElementJ_asymmetric(*n1*, *l1*, *j1*, *n2*, *l2*, *j2*)
Reduced matrix element in J basis, defined in asymmetric notation.

Note that notation for symmetric and asymmetricly defined reduced matrix element is not consistent in the literature. For example, notation is used e.g. in Steck[1] is precisely the oposite.

Note: Note that this notation is asymmetric: $(j||er||j') \neq (j'||er||j)$. Relation between the two notation is $\langle j||er||j'\rangle = \sqrt{2j+1}(j||er||j')$. 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

- **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

Returns reduced dipole matrix element in Steck notation $(j||er||j')$ $(a_0 e)$.

Return type float

getReducedMatrixElementL(*n1*, *l1*, *j1*, *n2*, *l2*, *j2*)
Reduced matrix element in L basis (symmetric notation)

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

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

Return type float

getStateLifetime(*n*, *l*, *j*, *temperature=0*, *includeLevelsUpTo=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
- **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.

[1] Daniel A. Steck, "Cesium D Line Data," (revision 2.0.1, 2 May 2008). http://steck.us/alkalidata

- **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 $n + 1$

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*)
Calculated transition frequency in Hz

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

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*, *temperature=0.0*)
Transition rate due to coupling to vacuum modes (black body included)

Calculates transition rate from the first given state to the second given state $|n_1, l_1, j_1\rangle \rightarrow |n_2, j_2, j_2\rangle$ at given temperature due to interaction with the vacuum field. For zero temperature this returns Einstein A coefficient. For details of calculation see Ref.[4] and Ref.[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

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

Return type float

[4] C. E. Theodosiou, PRA **30**, 2881 (1984) https://doi.org/10.1103/PhysRevA.30.2881
[5] I. I. Beterov, I. I. Ryabtsev, D. B. Tretyakov, and V. M. Entin, PRA **79**, 052504 (2009) https://doi.org/10.1103/PhysRevA.79.052504

References

getTransitionWavelength (*n1*, *l1*, *j1*, *n2*, *l2*, *j2*)
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

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*)
Retuns linear (paramagnetic) Zeeman shift.

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

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

minQuantumDefectN = 0
minimal 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
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 (Hatree)
- **innerLimit** (*float*) – inner limit at which wavefunction is requested
- **outerLimit** (*float*) – outer limit at which wavefunction is requested
- **step** (*flaot*) – radial step for integration mesh (a.u.)

Returns

r

$R(r) \cdot r$

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

Note: Radial wavefunction is not scaled to unity! This normalization condition means that we are using spherical harmonics which are normalized such that $\int \mathrm{d}\theta\ \mathrm{d}\psi\ Y(l, m_l)^* \times Y(l', m_{l'}) = \delta(l, l')\ \delta(m_l, m_{l'})$.

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

---

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

scaledRydbergConstant = 0
in eV

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 descrete step in r size of *step*, integrating from *outerLimit* towards the *innerLimit* (from outside, inwards) equation $\frac{\mathrm{d}^2 rad(r)}{\mathrm{d}r^2} = kfun(r) \cdot rad(r)$.

Parameters

- **innerLimit** (*float*) – inner limit of integration
- **outerLimit** (*flaot*) – outer limit of integration
- **kfun** (*function(double)*) – pointer to function used in equation (see longer explanation above)
- **step** – descrete 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* swich is set to True (default option), 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)
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

arc.alkali_atom_functions.**printStateString**(n, l, j)
Returns 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

Returns label for the state in standard spectroscopic notation

Return type string

arc.alkali_atom_functions.**printStateStringLatex**(n, l, j)
Returns latex code for 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

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:

```
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:

```
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:

```
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**()

Setup the data folder in the users home directory.

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])	backward compatibility: before only one class for Potassium existed and it corresponded to Potassium 39
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])	backward compatibility: before there was only one Rubidium class, and that one corresponded to Rubidium85
Rubidium85([preferQuantumDefects, cpp_numerov])	Properites of rubidium 85 atoms
Rubidium87([preferQuantumDefects, cpp_numerov])	Properites of rubidium 87 atoms
Caesium([preferQuantumDefects, cpp_numerov])	Properties of caesium atoms

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

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

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]

[1] M. Marinescu, H. R. Sadeghpour, and A. Dalgarno, *Phys.Rev.A* **49**, 982 (1994) https://doi.org/10.1103/PhysRevA.49.982

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,
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 = 3.8939056946689456
(eV), Ref.[9].

quantumDefect = [[[4.04935665, 0.2377037, 0.255401, 0.00378, 0.25486, 0.0], [3.59158
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 = 13.605636850154157
in eV

class arc.alkali_atom_data.**Hydrogen**(*preferQuantumDefects=True*, *cpp_numerov=True*)
Properties of hydrogen atoms

ionisationEnergy = 13.598433
(eV), Ref.[12].

mass = 1.6735328115071732e-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*)
Properties of lithium 6 atoms

[3] C.B.Alcock, V.P.Itkin, M.K.Horrigan, *Canadian Metallurgical Quarterly*, **23**, 309 (1984) http://dx.doi.org/10.1179/cmq.1984.23.3.309
[9] Johannes Deiglmayr, Holger Herburger, Heiner Sassmannshausen, Paul Jansen, Hansjurg Schmutz, Frederic Merkt, *Phys. Rev. A* **93**, 013424 (2016) https://doi.org/10.1103/PhysRevA.93.013424
[2] K.-H. Weber and Craig J. Sansonetti, *Phys.Rev.A* **35**, 4650 (1987)
[10] C. -J. Lorenzen, and K. Niemax, *Z. Phys. A* **315**, 127 (1984) dx.doi.org/ 10.1007/BF01419370
[12] NIST, P. Mohr and S. Kotochigova, unpublished calculations (2000). The wavelengths for the Balmer-alpha and Balmer-beta transitions at 6563 and 4861 $xC5$ include only the stronger components of more extensive fine structures.
[14] J. S. Coursey, D. J. Schwab, J. J. Tsai, and R. A. Dragoset, (2015), Atomic Weights and Isotopic Compositions (version 4.1). Online Available: http://physics.nist.gov/Comp (2017, March, 14). National Institute of Standards and Technology, Gaithersburg, MD.

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]

a4 = [-0.11988362, -0.06990078, 0.1060243, 0.1060243]
model potential parameters from[1]

abundance = 0.0759
source NIST, Atomic Weights and Isotopic Compositions[14]

alphaC = 0.1923
model potential parameters from[1]

getPressure(*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.

mass = 9.988346384925222e-27
source NIST, Atomic Weights and Isotopic Compositions[14]

quantumDefect = [[[0.3995101, 0.029, 0.0, 0.0, 0.0, 0.0], [0.0471835, -0.024, 0.0, 0
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 Li7, differences are expected not be too big).

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*)
Properties of lithium 7 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]

a4 = [-0.11988362, -0.06990078, 0.1060243, 0.1060243]
model potential parameters from[1]

abundance = 0.9241
source NIST, Atomic Weights and Isotopic Compositions[14]

alphaC = 0.1923
model potential parameters from[1]

getPressure(*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.

ionisationEnergy = 5.391719
(eV) NIST Ref.[13].

[8] P. Goy, J. Liang, M. Gross, and S. Haroche, *Phys. Rev. A* **34**, 2889 (1986) https://doi.org/10.1103/PhysRevA.34.2889

[11]
3. -J. Lorenzen, and K. Niemax, *Physica Scripta* **27**, 300 (1983)

[13]
18. (l) Kelly, *J. Phys. Chem. Ref. Data* **16**, Suppl. 1 (1987).

mass = 1.1650347611248465e-26
source NIST, Atomic Weights and Isotopic Compositions[14]

quantumDefect = [[[0.3995101, 0.029, 0.0, 0.0, 0.0, 0.0], [0.047178, -0.024, 0.0, 0
quantum defects for nS and nP states are from Ref.[8]. Quantum defects for D_j and F_j states are from[11].

rc = [0.61340824, 0.61566441, 2.34126273, 2.34126273]
model potential parameters from[1]

class arc.alkali_atom_data.**Potassium**(*preferQuantumDefects=True*, *cpp_numerov=True*)
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*)
Properties of potassium 39 atoms

a1 = [3.56079437, 3.65670429, 4.12713694, 1.42310446]
model potential parameters from[1]

a2 = [1.83909642, 1.67520788, 1.79837462, 1.27861156]
model potential parameters from[1]

a3 = [-1.74701102, -2.07416615, -1.69935174, 4.77441476]
model potential parameters from[1]

a4 = [-1.03237313, -0.89030421, -0.98913582, -0.94829262]
model potential parameters from[1]

abundance = 0.932581
source NIST, Atomic Weights and Isotopic Compositions[14]

alphaC = 5.331
model potential parameters from[1]

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[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.

ionisationEnergy = 4.340663681814173
(eV), weighted average of values in Ref.[11].

mass = 6.470075576376843e-26
source NIST, Atomic Weights and Isotopic Compositions[14]

quantumDefect = [[[2.1801985, 0.13558, 0.0759, 0.117, -0.206, 0.0], [1.713892, 0.23
quantum defects from Ref.[11].

rc = [0.83167545, 0.85235381, 0.83216907, 6.50294371]
model potential parameters from[1]

class arc.alkali_atom_data.**Potassium40**(*preferQuantumDefects=True*, *cpp_numerov=True*)
Properties of potassium 40 atoms

a1 = [3.56079437, 3.65670429, 4.12713694, 1.42310446]
model potential parameters from[1]

a2 = [1.83909642, 1.67520788, 1.79837462, 1.27861156]
model potential parameters from[1]

a3 = [-1.74701102, -2.07416615, -1.69935174, 4.77441476]
model potential parameters from[1]

a4 = [-1.03237313, -0.89030421, -0.98913582, -0.94829262]
model potential parameters from[1]

abundance = 0.000117
source NIST, Atomic Weights and Isotopic Compositions[14]

alphaC = 5.331
model potential parameters from[1]

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[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.

ionisationEnergy = 4.340663681814173
(eV), weighted average of values in Ref.[11].

mass = 6.63617791491314e-26
source NIST, Atomic Weights and Isotopic Compositions[14]

quantumDefect = [[[2.1801985, 0.13558, 0.0759, 0.117, -0.206, 0.0], [1.713892, 0.23
quantum defects from Ref.[11].

rc = [0.83167545, 0.85235381, 0.83216907, 6.50294371]
model potential parameters from[1]

scaledRydbergConstant = 13.6055062456062
in eV

class arc.alkali_atom_data.**Potassium41**(*preferQuantumDefects=True*, *cpp_numerov=True*)
Properties of potassium 41 atoms

a1 = [3.56079437, 3.65670429, 4.12713694, 1.42310446]
model potential parameters from[1]

a2 = [1.83909642, 1.67520788, 1.79837462, 1.27861156]
model potential parameters from[1]

a3 = [-1.74701102, -2.07416615, -1.69935174, 4.77441476]
model potential parameters from[1]

a4 = [-1.03237313, -0.89030421, -0.98913582, -0.94829262]
model potential parameters from[1]

abundance = 0.067302
source NIST, Atomic Weights and Isotopic Compositions[14]

alphaC = 5.331
model potential parameters from[1]

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[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.

ionisationEnergy = 4.340663681814173
(eV), weighted average of values in Ref.[11].

mass = 6.801870999040102e-26
source NIST, Atomic Weights and Isotopic Compositions[14]

quantumDefect = [[[2.1801985, 0.13558, 0.0759, 0.117, -0.206, 0.0], [1.713892, 0.233
quantum defects from Ref.[11].

rc = [0.83167545, 0.85235381, 0.83216907, 6.50294371]
model potential parameters from[1]

scaledRydbergConstant = 13.605510795147651
in eV

class arc.alkali_atom_data.**Rubidium**(*preferQuantumDefects=True*, *cpp_numerov=True*)
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*)
Properites of rubidium 85 atoms

a1 = [3.69628474, 4.44088978, 3.78717363, 2.39848933]
model potential parameters from[1]

a2 = [1.64915255, 1.92828831, 1.57027864, 1.76810544]
model potential parameters from[1]

a3 = [-9.86069196, -16.7959777, -11.6558897, -12.0710678]
model potential parameters from[1]

a4 = [0.19579987, -0.8163314, 0.52942835, 0.77256589]
model potential parameters from[1]

abundance = 0.7217
source NIST, Atomic Weights and Isotopic Compositions[14]

alphaC = 9.076
model potential parameters from[1]

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 from[3]. Values from table 2. (accuracy +- 5%) are used for Rb in solid phase. Values from table 3. (accuracy +-1 %) are used for Rb in liquid phase.

ionisationEnergy = 4.177126489738963
(eV) Ref.[15]

mass = 1.409993418160543e-25
source NIST, Atomic Weights and Isotopic Compositions[14]

quantumDefect = [[[3.1311804, 0.1784, 0.0, 0.0, 0.0, 0.0], [2.6548849, 0.29, 0.0, 0
quantum defects for nF states are from[5]. Quantum defects for nG states are from[7]. All other quantum defects are from from[4]

rc = [1.66242117, 1.50195124, 4.86851938, 4.79831327]
model potential parameters from[1]

[15] B. Sanguinetti, H. O. Majeed, M. L. Jones and B. T. H. Varcoe, *J. Phys. B* **42**, 165004 (2009) http://iopscience.iop.org/article/10.1088/0953-4075/42/16/165004/meta

[5] Jianing Han, Yasir Jamil, D. V. L. Norum, Paul J. Tanner, and T. F. Gallagher, *Phys. Rev. A* **74**, 054502 (2006) https://doi.org/10.1103/PhysRevA.74.054502

[7] K. Afrousheh, P. Bohlouli-Zanjani, J. A. Petrus, and J. D. D. Martin, *Phys. Rev. A* **74**, 062712 (2006) https://doi.org/10.1103/PhysRevA.74.062712

[4] Wenhui Li, I. Mourachko, M. W. Noel, and T. F. Gallagher, *Phys. Rev. A* **67**, 052502 (2003) https://doi.org/10.1103/PhysRevA.67.052502

scaledRydbergConstant = 13.605605108199638
in eV

class arc.alkali_atom_data.**Rubidium87**(*preferQuantumDefects=True*, *cpp_numerov=True*)
Properites of rubidium 87 atoms

a1 = [3.69628474, 4.44088978, 3.78717363, 2.39848933]
model potential parameters from[1]

a2 = [1.64915255, 1.92828831, 1.57027864, 1.76810544]
model potential parameters from[1]

a3 = [-9.86069196, -16.7959777, -11.6558897, -12.0710678]
model potential parameters from[1]

a4 = [0.19579987, -0.8163314, 0.52942835, 0.77256589]
model potential parameters from[1]

abundance = 0.2783
source NIST, Atomic Weights and Isotopic Compositions[14]

alphaC = 9.076
model potential parameters from[1]

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 from[3]. Values from table 2. (accuracy +- 5%) are used for Rb in solid phase. Values from table 3. (accuracy +-1 %) are used for Rb in liquid phase.

ionisationEnergy = 4.177127287605897
(eV) Ref.[6]

mass = 1.4431608720613343e-25
source NIST, Atomic Weights and Isotopic Compositions[14]

quantumDefect = [[[3.1311804, 0.1784, 0.0, 0.0, 0.0, 0.0], [2.6548849, 0.29, 0.0, 0
quantum defects for nF states are from[5]. Quantum defects for nG states are from[7]. All other quantum defects are from from[4]

rc = [1.66242117, 1.50195124, 4.86851938, 4.79831327]
model potential parameters from[1]

scaledRydbergConstant = 13.60560712837914
in eV (M_ion core = m_atomic - m_electron)

class arc.alkali_atom_data.**Sodium**(*preferQuantumDefects=True*, *cpp_numerov=True*)
Properties of sodium 23 atoms

a1 = [4.82223117, 5.08382502, 3.53324124, 1.11056646]
model potential parameters from[1]

a2 = [2.45449865, 2.18226881, 2.48697936, 1.05458759]
model potential parameters from[1]

a3 = [-1.12255048, -1.19534623, -0.75688448, 1.73203428]
model potential parameters from[1]

a4 = [-1.42631393, -1.03142861, -1.27852357, -0.09265696]
model potential parameters from[1]

[6] Markus Mack, Florian Karlewski, Helge Hattermann, Simone Hockh, Florian Jessen, Daniel Cano, and Jozsef Fortagh, *Phys. Rev. A* **83**, 052515 (2011), https://doi.org/10.1103/PhysRevA.83.052515

abundance = 1.0
source NIST, Atomic Weights and Isotopic Compositions[14]

alphaC = 0.9448
model potential parameters from[1]

getPressure(*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.

ionisationEnergy = 5.139075550664961
(eV) from Ref.[11]

mass = 3.8175409413353766e-26
source NIST, Atomic Weights and Isotopic Compositions[14]

quantumDefect = [[[1.347964, 0.060673, 0.0233, −0.0085, 0.0, 0.0], [0.85538, 0.11361
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].

rc = [0.45489422, 0.45798739, 0.71875312, 28.6735059]
model potential parameters from[1]

scaledRydbergConstant = 13.605368351064202
(eV)

1.3.3 Single atom calculations

Overview

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.exportData(fileBase[, exportFormat])	Exports StarkMap calculation data.
StarkMap.getPolarizability([maxField, ...])	Returns the polarizability of the state (set during the initalization process)

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

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.**LevelPlot**(*atomType*)

Single atom level plots and decays

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*)
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

showPlot()
Shows a level diagram plot

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 Zimmerman[1]. For a quick working example see Stark map 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 } Select the alkali metal for energy level diagram calculation

Examples

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

```
>>> 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

```
>>> 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 >>
```

[1] M. L. Zimmerman et.al, PRA **20**:2251 (1979) https://doi.org/10.1103/PhysRevA.20.2251

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.

```
>>> 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 (propotional 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*)
Initializes basis of states around state of interest

Defines basis of states for further calculation. n, l, j, m_j specify state whose neighbourhood and polarizability 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 propotional 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** (*flaot*) – 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 debuging. Set to false by default.

diagonalise(*eFieldList, drivingFromState=[0, 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 originaly 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 usefull for debuging. 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 initalization process)

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 usefull for debuging. Set to false by default.

Returns scalar polarizability in units of MHz cm ^{2} / V ^{2}

Return type float

highlight = None
highlight[i] 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 elemements 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*, *highlighState=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 usefull for debuging. 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.

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

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

showPlot(*interactive=True*)
Shows plot made by *plotLevelDiagram*

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*

arc.calculations_atom_single.**printState**(*n*, *l*, *j*)
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

arc.calculations_atom_single.**printStateString**(*n*, *l*, *j*)
Returns 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

Returns label for the state in standard spectroscopic notation

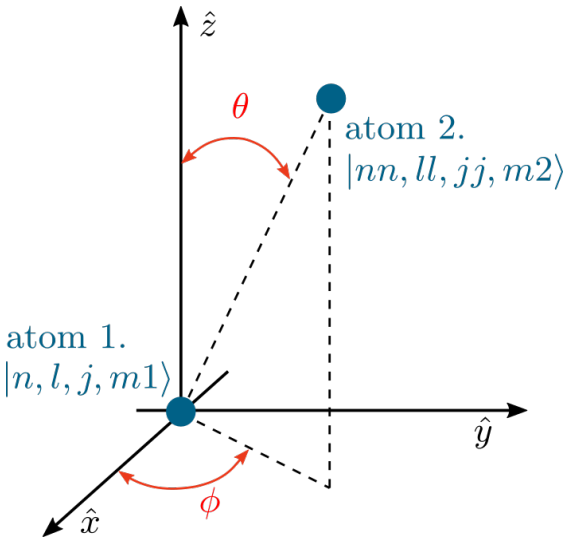
Return type string

1.3.4 Pair-state basis calculations

Preliminaries

Relative orientation of the two atoms can be described with polar angle θ (range $0 - \pi$) and azimuthal angle ϕ (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 $|n, l, j, m_1\rangle$ and $|nn, ll, jj, m_2\rangle$ 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

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

Continued on next page

Table 6 – continued from previous page

PairStateInteractions.getC3fromLevelDiagram()	Finds C_3 coefficient for original pair state.
PairStateInteractions.getVdwFromLevelDiagram()	Finds r_{vdW} coefficient for original pair state.

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 $P_{1/2}\ m_j = 1/2$ with π ($q = 0$) polarized light. eigenstates:

```
from arc import *
calc1 = PairStateInteractions(Rubidium(), 60, 0, 0.5, 60, 0, 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")
calculation1.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*)

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

- **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,

[1] T. G Walker, M. Saffman, PRA **77**, 032723 (2008) https://doi.org/10.1103/PhysRevA.77.032723

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

- **n** (*int*) – principal quantum number for the *first* atom
- **l** (*int*) – orbital angular momentum for the *first* atom
- **j** (*float*) – total angular momentum for the *first* atom
- **nn** (*int*) – principal quantum number for the *second* atom
- **ll** (*int*) – orbital angular momentum for the *second* atom
- **jj** (*float*) – total angular momentum for the *second* atom
- **m1** (*float*) – projection of the total angular momentum on z-axis for the *first* atom
- **m2** (*float*) – projection of the total angular momentum on z-axis for the *second* atom
- **interactionsUpTo** (*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.

References

Examples

Advanced interfacing of pair-state 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 *defineBasis*, you can assemble it easily from diagonal part (stored in *matDiagonal*) and off-diagonal matrices whose spatial dependence is R^{-3}, R^{-4}, R^{-5} stored in that order in *matR*. Basis states are stored in *basisStates* array.

```
>>> from arc import *
>>> calc = PairStateInteractions(Rubidium(), 60,0,0.5, 60,0,0.5, 0.5,0.5,
↪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
```

atom = None
atom type

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

channel = None
states relevant for calculation, defined in J basis (not resolving m_j. Used as intermediary for full interaction matrix calculation by *defineBasis*.

coupling = None
List of matrices defineing coupling strengths between the states in J basis (not resolving m_j). Basis is given by `channel`. Used as intermediary for full interaction matrix calculation by `defineBasis`.

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 `basisStates`. Interaction matrix is saved in parts depending on the scaling with distance. Diagonal elements `matDiagonal`, correponding 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-qudrupole (C_4) and quadrupole-quadrupole coupling (C_5) respectively. These parts of the matrix are stored in `matR` in that order. I.e. `matR[0]` stores dipole-dipole coupling ($\propto R^{-3}$), `matR[0]` stores dipole-quadrupole couplings etc.

Parameters

- **theta** (`float`) – relative orientation of the two atoms (see figure on top of the page), range 0 to π
- **phi** (`float`) – relative orientation of the two atoms (see figure on top of the page), range 0 to 2π
- **nRange** (`int`) – how much below and above the given principal quantum number of the pair state we should be looking?
- **lrange** (`int`) – what is the maximum angular orbital momentum state that we are including in calculation
- **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
- **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, False by default. If true, prints information about the progress of the calculation.
- **debugOutput** (`bool`) – optional, False by default. If true, similar to progressOutput=True, this will print information about the progress of calculations, but with more verbose output.

See also:

`alkali_atom_functions.saveCalculation` and `alkali_atom_functions.loadSavedCalculation` for information on saving intermediate results of calculation for later use.

diagonalise(*rangeR, noOfEigenvectors, drivingFromState=[0, 0, 0, 0, 0], eigenstateDetuning=0.0, sortEigenvectors=False, progressOutput=False, debugOutput=False*)
Finds eigenstates in atom pair basis.

ARPACK (`scipy.sparse.linalg.eigsh`) calculation of the *noOfEigenvectors* eigenvectors closest to the original state. If *drivingFromState* is specified as *[n,l,j,mj,q]* 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 σ^-, π and σ^+ transitions respectively) is calculated and marked by the colourmaping these values on the obtained eigenvectors.

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 the one of the atoms from the original pair-state basis from which we try to dribe to the excited pair-basis manifold. By default, program will calculate just contribution of the original pair-state in the eigenstates obtained by diagonalization, and will highlight it's admixure by colour mapping the obtained eigenstates plot.
- **sortEigenvectors** (*bool*) – optional, False by default. Tries to sort eigenvectors so that given eigen vector index corresponds to adiabatically changing eigenstate, as detirmined 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, similar 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.

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.

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

Finds C_3 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 $A + C_3/R^3$ 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 (fore resonantBranch = -1) depending on the value of resonantBranch optional parameter

Returns C_3 measured in GHz μm^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 μm^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*)

Calculates C_6 from second order perturbation theory.

Calculates $C_6 = \sum_{\mathrm{r}',\mathrm{r}''} |\langle \mathrm{r}', \mathrm{r}''|V|\mathrm{r1}, \mathrm{r2}\rangle|^2/\Delta_{\mathrm{r}',\mathrm{r}''}$, where $\Delta_{\mathrm{r}',\mathrm{r}''} \equiv E(\mathrm{r}', \mathrm{r}'') - E(\mathrm{r1}, \mathrm{r2})$

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 ($\Delta E/h$ in Hz) between the original pair state and the other pair states that we are including in calculation

Returns C_6 measured in GHz μm^6

Return type float

Example

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

```
from arc import *
calculation = PairStateInteractions(Rubidium(), 62, 2, 1.5, 62, 2, 1.5, 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 $D_{2/5}, m_j = 5/2$ states:

```
# 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, 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       C6 = %.2f GHz  mum^6" % (theta/pi,
↪value))
# plot results
plot(thetaList/pi,c6,"b-")
title("Rb, pairstate  60 $D_{5/2},m_j = 5/2$, 60 $D_{5/2},m_j = 5/2$")
xlabel(r"$\Theta /\pi$")
ylabel(r"$C_6$ (GHz $\mu$m${}^6$")
show()
```

getLeRoyRadius()

Returns Le Roy radius for initial pair-state.

Le Roy radius[2] is defined as $2(\langle r_1^2\rangle^{1/2} + \langle r_2^2\rangle^{1/2})$, 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 LeRoy radius measured in μm

Return type float

[2] R.J. Le Roy, Can. J. Phys. **52**, 246 (1974) http://www.nrcresearchpress.com/doi/abs/10.1139/p74-035

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 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 $A + B\frac{1-\sqrt{1+(r_{\mathrm{vdW}}/r)^6}}{1-\sqrt{1+r_{\mathrm{vdW}}^6}}$

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 μm 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]).

---

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.

j = None

pair-state definition – total angular momentum of the first atom

jj = None

pair-state definition – total angular momentum of the second atom

l = None

pair-state definition – orbital angular momentum of the first atom

ll = None

pair-state definition – orbital angular momentum of the second atom

m1 = None

pair-state definition – projection of the total ang. momentum for the *first* atom

m2 = None

pair-state definition – projection of the total angular momentum for the *second* atom

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.

n = None
pair-state definition – principal quantum number of the first atom

nn = None
pair-state definition – principal quantum number of the second atom

originalPairStateIndex = None
index of the original n,l,j,m1,nn,ll,jj,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.

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,

alkali_atom_data.Potassium39, alkali_atom_data.Potassium40, alkali_atom_data.Potassium41, alkali_atom_data.Rubidium85, alkali_atom_data.Rubidium87, alkali_atom_data.Caesium }

the first atom in the pair-state

- **state1** ([int, int, float, float]) – specification of the state of the first state as an array of values $[n, l, j, m_j]$
- **atom** – ={ 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 }

  the second atom in the pair-state

- **state2** ([int, int, float, float]) – specification of the state of the first state as an array of values $[n, l, j, m_j]$

Note: In checking if certain state is dipole coupled to the original state, only the highest contributing state is checked for dipole coupling. This should be fine if one is interested in resonances in weak fields. For stronger fields, one might want to include effect of coupling to other contributing base states.

---

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*]) – optinal 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(*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).

1.4 How to contribute to the project

Ideally, this package/library will grow into a community project, as a community-maintained resource for atomic physics community. Full code will be accessible from GitHub, so please fork the project, and submit improvements, additional modules, new features, or just suggest ideas. We have a list of features that can be potentially included.

1.4.1 Ideas for development

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

- Dressing potentials
- Magic wavelengths
- Atom-wall interactions
- Photoionization
- Collisional cross-sections
- Tensor polarisability
- ... (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():
    ...
```

CHAPTER 2

Package structure

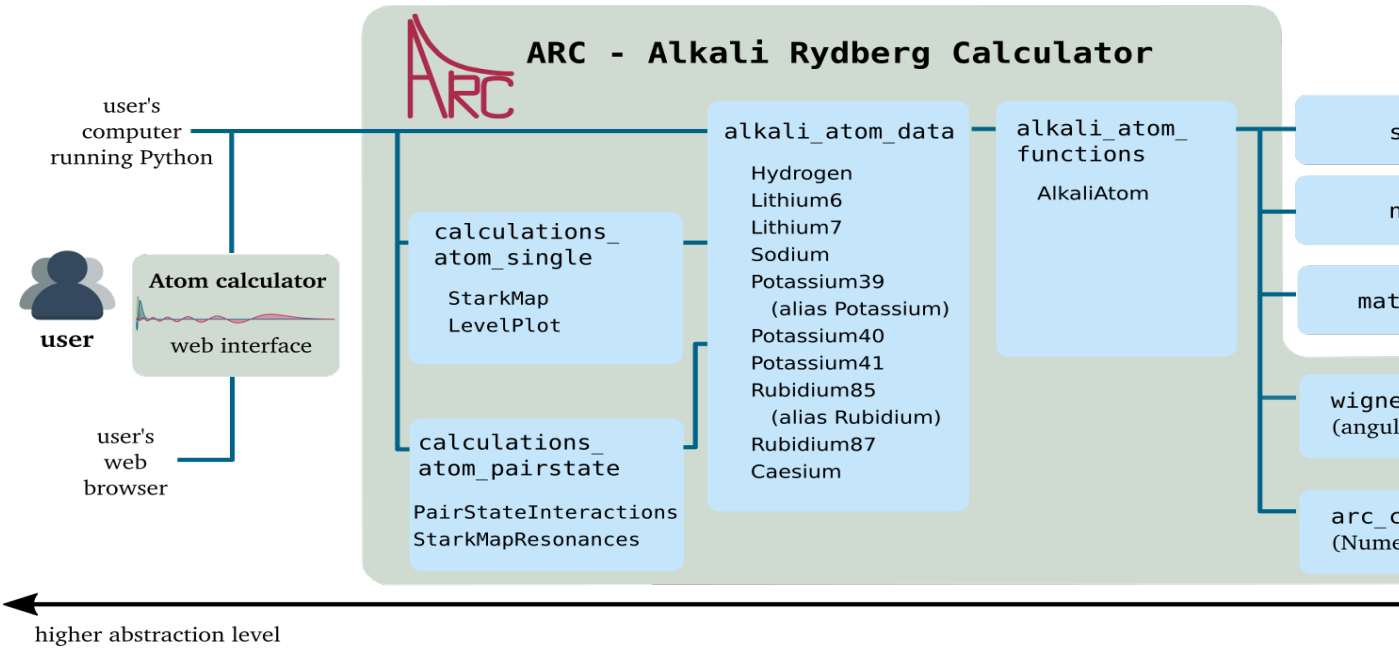


Fig. 1: Overview of modules and interdependencies in the `arc` package. Click on image to enlarge.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

CHAPTER 4

Credits

Authors Nikola Šibalić, Jonathan D. Pritchard, Charles S. Adams, Kevin J. Weatherill

Version 1.2.0 of 2017/06/10

Python Module Index

a

Index

C

D

E

F

G

H

I

J

L

M

U

Y

Z