
This version is available at https://strathprints.strath.ac.uk/60863/

Strathprints is designed to allow users to access the research output of the University of Strathclyde. Unless otherwise explicitly stated on the manuscript, Copyright © and Moral Rights for the papers on this site are retained by the individual authors and/or other copyright owners. Please check the manuscript for details of any other licences that may have been applied. You may not engage in further distribution of the material for any profitmaking activities or any commercial gain. You may freely distribute both the url (https://strathprints.strath.ac.uk/) and the content of this paper for research or private study, educational, or not-for-profit purposes without prior permission or charge.

Any correspondence concerning this service should be sent to the Strathprints administrator: strathprints@strath.ac.uk

The Strathprints institutional repository (https://strathprints.strath.ac.uk) is a digital archive of University of Strathclyde research outputs. It has been developed to disseminate open access research outputs, expose data about those outputs, and enable the management and persistent access to Strathclyde's intellectual output.
ARC: An open-source library for calculating properties of alkali Rydberg atoms

N. Šibalić, J.D. Pritchard, C.S. Adams, K.J. Weatherill

Joint Quantum Center (JQC) Durham-Newcastle, Department of Physics, Durham University, South Road, Durham, DH1 3LE, United Kingdom

Department of Physics, SUPA, University of Strathclyde, 107 Rottenrow East, Glasgow, G4 0NG, United Kingdom

A R T I C L E   I N F O

Article history:
Received 13 December 2016
Received in revised form 12 June 2017
Accepted 17 June 2017
Available online 5 August 2017

Keywords:
Alkali atom
Matrix elements
Dipole–dipole interactions
Stark shift
 Förster resonances

A B S T R A C T

We present an object-oriented Python library for the computation of properties of highly-excited Rydberg states of alkali atoms. These include single-body effects such as dipole matrix elements, excited-state lifetimes (radiative and black-body limited) and Stark maps of atoms in external electric fields, as well as two-atom interaction potentials accounting for dipole and quadrupole coupling effects valid at both long and short range for arbitrary placement of the atomic dipoles. The package is cross-referenced to precise measurements of atomic energy levels and features extensive documentation to facilitate rapid upgrade or expansion by users. This library has direct application in the field of quantum information and quantum optics which exploit the strong Rydberg dipolar interactions for two-qubit gates, robust atom–light interfaces and simulating quantum many-body physics, as well as the field of metrology using Rydberg atoms as precise microwave electrometers.

Program summary

Program Title: ARC: Alkali Rydberg Calculator
Program Files doi: http://dx.doi.org/10.17632/hm5n8w628c.1
Licensing provisions: BSD-3-Clause
Programming language: Python 2.7 or 3.5, with C extension
External Routines: NumPy [1], SciPy [1], Matplotlib [2]
Nature of problem: Calculating atomic properties of alkali atoms including lifetimes, energies, Stark shifts and dipole–dipole interaction strengths using matrix elements evaluated from radial wavefunctions.
Solution method: Numerical integration of radial Schrödinger equation to obtain atomic wavefunctions, which are then used to evaluate dipole matrix elements. Properties are calculated using second order perturbation theory or exact diagonalisation of the interaction Hamiltonian, yielding results valid even at large external fields or small interatomic separation.
Restrictions: External electric field fixed to be parallel to quantisation axis.
Supplementary material: Detailed documentation (.html), and Jupyter notebook with examples and benchmarking runs (.html and .ipynb).


© 2017 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

Highly-excited atoms, in so called Rydberg states, provide strong atom–atom interactions, and large optical nonlinearities. They are a flourishing field for quantum information processing [1,2] and quantum optics [3–5] in the few to single excitation regime, as well as many-body physics [6–11], in the many-excitations limit. Their exploration requires detailed knowledge of both the single-atom properties, such as lifetimes, energies and transition dipoles elements, as well as atom pair properties, such as their interactions and strongly perturbed energy levels for atoms at distances in the micrometre range. Rydberg states are also highly sensitive to external DC and AC fields, making them excellent candidates for precision electrometry and imaging in the microwave [12,13] and terahertz [14] range, as well as performing state engineering to tune pair-state interaction potentials [15–17].
Although many of the relevant calculations share the same primitives, such as numeric integration of atomic wavefunctions based on measured energy levels and model core potentials, these basic efforts have been repeated by many groups independently so far. To date no single common resource has emerged for building complex calculations, or for performing quick numerical estimates. To this end, we have developed Alkali Rydberg Calculator (ARC) [18], a library of routines written in Python, using object-oriented programming to make a 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.

The hierarchical nature of the package helps organise possible calculations into abstraction levels, allowing one to pick the information at the relevant level for the calculation at hand. The data for individual atomic species is provided as classes that inherit calculation methods defined as abstract classes, allowing one easily to check and update relevant data, should future measurements improve some of the experimentally estimated values. Detailed documentation is provided for all the ARC’s modules [19]. In addition, the code is commented, cross-referenced in-line and uses self-descriptive names. Whenever possible, the class and function names reflect the hierarchical structure of atomic physics knowledge and the natural decompositions of the calculations, not the low-level implementation details. In addition to the documentation, ARC has example snippets provided as an IPython notebook [20], giving an overview of Rydberg physics and how to perform calculations using the package library. This is a good starting point for new users.

To facilitate the initial adoption of the package and to allow quick calculations useful in the research planning stages, we are also providing a web-interface to basic functions of the package [21]. This allows any device with a web browser to access the web-server, that will use the ARC package to perform the calculations and obtain results that are transferred back to the users’ web browser. In the process, the web service self-generates and outputs the code, so it can be used as an example-on-demand service, providing a starting point for more complex calculations.

This paper is organised as follows. An overview of the ARC architecture is presented in Section 2, where we introduce the theoretical framework for performing Rydberg state calculations, e.g. calculating wavefunctions and diagonalising interaction Hamiltonians. Here we also provide illustrative examples for building up calculations and visualising results with the provided tools. Initial setup for the ARC library is presented in Section 3, and specific details relating to the implementation are discussed in Section 4. Finally, Section 5 briefly summarises the package and outlines future possible expansions for the library, with a complete list of ARC classes, methods and functions detailed in Appendix A. Detailed documentation of the module is provided in .html format in the Supplemental Material [19] or available from the ARC website [21], along with an IPython notebook [20] that contains numerous additional examples and code snippets that reproduce many of the results from the literature.

2. ARC architecture and modules

2.1. Overview

The structure of the ARC library is shown in Fig. 1. At the lowest level, the wigner module implements angular momentum algebra (Wigner 3-j and 6-j coefficients and the WignerD rotation matrix), and arc_c_extensions is a Python extension coded in C to provide fast calculation of the radial part of the atomic wavefunctions. On a higher level, alkali_atom_functions uses these low-level functions to build general methods for calculating single-atom properties, which are contained within the abstract class AlkaliAtom that implements calculation of dipole matrix elements, transition rates, energy levels etc. The module alkali_atom_data defines an explicit class for each alkali element (e.g. Rubidium87(),Caesium()) that encodes all relevant physical parameters and inherits the calculation methods from the parent AlkaliAtom class. These atom classes can be passed as arguments to either of the core calculation modules, calculations_atom_single that implements interactive energy level diagrams (LevelPlot) and calculates Stark maps for atoms in external fields (StarkMap), or calculations_atom_pairstate for dealing with two-atom effects such as long-range dipole–dipole interactions. This pair-state module provides a sophisticated interface to automatically identify Förster resonances for atoms in weak electric fields (StarkMapResonances) and calculate atom interaction potentials at both long and short range including up to quadrupole–quadrupole couplings (PairStateInteractions). In the following we will outline the basic functionality of the ARC module, provide the theoretical framework for the various modules and give examples of relevant calculations implemented in the library.

2.2. The AlkaliAtom class

Almost all calculated quantities in the ARC library can be derived from knowledge of the Rydberg state energy levels and matrix elements. The functions to calculate these values along with other primitive single-atom properties are encapsulated within the methods of the abstract class AlkaliAtom defined by the module alkali_atom_functions. This class is used in alkali_atom_data where each alkali-metal atom (Lithium6, Lithium7, Sodium, Potassium39, Potassium40, Potassium41, Rubidium85, Rubidium87 and Caesium, as well as Hydrogen) inherits calculation methods from this abstract class, and specifies all the necessary numerical values for a given atom. Calculations with the ARC library begin by initiating an atom class associated with the relevant atom, as shown in the example below for caesium.

```python
from arc import *  # initialise ARC library
atom = Caesium()  # create atom Object
```

Physical properties can then be determined using the methods in the form atom.functionName(parameters), where a complete list of available methods is listed in Table B.2 and documentation [19]. The sections below outline the key properties.

2.2.1. Rydberg atom energy levels

Energy of the Rydberg state with principal quantum number n and orbital and total angular momentum ℓ and j respectively, are calculated from the Rydberg formula

\[
E_{n,\ell,j} = E_P - \frac{\text{Ry}}{(n - \delta_{n,\ell,j})^2},
\]

where \(E_P\) is the ionisation energy threshold \(\text{ionisationEnergy}\), \(\text{Ry}\) is the Rydberg constant corrected for the reduced mass \(\text{scaledRydbergConstant}\), \(\text{Ry} = m_e/(m + m_H)\text{Ry}\), and \(\delta_{n,\ell,j}\) is the quantum defect \(\text{getQuantumDefect}\) given by

\[
\delta_{n,\ell,j} = \delta_0 + \frac{\delta_2}{(n - \delta_0)^2} + \frac{\delta_4}{(n - \delta_0)^4} + \ldots,
\]

where \(\delta_0, \delta_2, \ldots\) are modified Rydberg–Ritz coefficients obtained by fitting precise measurements of the atomic energy levels [22].

Energies \(E\) for a given Rydberg state relative to the ionisation limit can be obtained using \(\text{getEnergy}\) (in eV), as well as methods \(\text{getTransitionWavelength}\) and \(\text{getTransitionFrequency}\) to return the wavelength (m) or frequency (Hz) of a transition between two Rydberg states. Energies and transitions are given relative to the centres of gravity of the hyperfine split ground and excited states.
2.2.2. Rydberg atom wavefunctions

In order to calculate matrix elements for electric dipole and quadrupole couplings of the Rydberg states, it is necessary to calculate the radial wavefunctions \( R(r) \) by numerically integrating the Radial Schrödinger equation for the valence electron. Using the substitution \( \rho(r) = rR(r) \), we can remove the first order differential to obtain an equation in the form:

\[
-\frac{1}{2\mu} \frac{d^2\rho}{dr^2} + \left[ \ell(\ell + 1) \frac{1}{2r^2} + V(r) \right] \rho(r) = E \rho(r),
\]

where \( V(r) \) is the spherically symmetric central potential (potential). For hydrogen, and high orbital angular momentum states \( \ell > 3 \), this is simply the Coulomb potential of \( V(r) = -1/r + V_0(r) \), with added (relativistic) spin–orbit interaction \( V_0(r) = \alpha L \cdot S/(2r^3) \), where \( \alpha \) is the fine structure constant. However for alkali atoms with closed shells it is necessary to introduce a model potential that gives a Coulomb potential at long-range and at short range accounts for effects of the core penetration of the valence electron. We adopt the core potential \( V_0(r) \) of Marinescu et al. [23] given by the form

\[
V(r) = -\frac{Z_i(r)}{r} - \frac{\alpha_e}{2r}(1-e^{-r/a_i}) + V_0(r),
\]

where \( \alpha_e \) is the core polarisability, \( r_i \) is a cutoff radius introduced to avoid divergence of the polarisation potential at short range and the radial charge (effective charge) is given by

\[
Z_i(r) = 1 + (z - 1)e^{-\alpha r} - r(a_3 + a_4 r)e^{-\alpha r},
\]

with \( \ell \)-dependent parameters \( a_{1,4} \) and \( r_i \) taken from Table 1 of Ref. [23] which were obtained by fitting the model potential to measured energy levels for each element.

Using this model potential and the known Rydberg energy from above, the radial wavefunctions can be calculated by numerically integrating Eq. (3), as shown on Fig. 2. This is achieved by performing a transformation to integrate the function \( X(r) = R(r)r^{3/4} \) in terms of the scaled co-ordinate \( x = \sqrt{r} \) [24] that gives an approximately constant number of points across each period of oscillation in the wavefunction. The result is a second-order differential equation of the form

\[
\frac{d^2X}{dx^2} = g(x) X,
\]

which is efficiently solved using the Numerov method [25,26]

\[
[1 - T(x + h)] X(x + h) + [1 - T(x - h)] X(x - h) = [2 + 10T(x)] X(x) + O(h^4),
\]

where \( h \) is the step size, \( T(x) = h^2 g(x)/12 \) and

\[
g(x) = 8 \mu x^2 (V(r) - E) + \frac{\ell(\ell + \frac{1}{2})}{x^2} (2\ell + \frac{3}{2}).
\]

It is necessary to truncate the range of integration as at short range the model becomes unphysical and diverges, whilst at long range the wavefunction decays to zero. Following Ref. [27], the limits of integration are set to use an inner radius of \( r_i = \sqrt{\alpha} \), and an outer radius of \( r_o = 2n(n + 15) \) which is much larger than the classical turning point of the wavefunction. To minimise errors introduced by the approximate model potential at short range, the integration is performed inwards, starting at \( r_o \). For high orbital momentum states \( \ell > 3 \) divergence can occur even before \( r_i \), which is automatically detected and integration is stopped at the closest wavefunction node before divergence occurred. The wavefunctions are then normalised, and returned by \( \text{radialWavefunction} \).

2.2.3. Matrix elements

Relevant properties for alkali atoms such as lifetimes (Section 2.2.5), Stark shifts (Section 2.3.2) and atomic interaction potentials (Section 2.4.1) require evaluation of the electric dipole and electric quadrupole matrix elements from state \( |n, \ell, m_\ell \rangle \) to \( |n', \ell', m'_{\ell'} \rangle \). For electric dipole transitions, the interaction is dependent upon matrix elements of the form \( \hat{\mathbf{r}} = -\mathbf{e} \cdot \hat{\mathbf{e}} \) where \( \hat{\mathbf{e}} \) is the electric field polarisation vector. Expanding the operator in

\[ Fig. 1. \] Overview of the Alkali Rydberg Calculator (ARC) package for Python. Object-oriented structure with hierarchy reflecting the structure of atomic physics calculations is used. This allows the user to choose the abstraction level at which one wants to work, from low-level implementations of basic functions of \texttt{alkali\_atom\_functions.AlkaliAtom} for finding dipole matrix elements and lifetimes, to high-level functions that allow automatic Förster resonance finding and exploration of complex energy-level diagrams of atomic pair-states at small inter-atomic separations in \texttt{calculations\_atom\_pairstate}.

\[ Fig. 2. \] Electron wavefunctions are calculated by Numerov integration, here shown for caesium 18 $S_{1/2}m_i = 1/2$ state. Solving the radial part of the Schrödinger equation in the model potential from [23], using state energy from quantum defects or NIST ASD database [28]. These calculations are the starting step in calculating all atom-light couplings (dipolar and higher order).

the spherical basis and using the fact that the calculation can be separated into radial overlap, and angular overlap of the electron wavefunction with operator $\hat{r}$, the resulting matrix element can be evaluated in terms of angular momentum terms and radial matrix elements using the Wigner–Ekart theorem [29]

$$\langle n, \ell, m_l | r_q | n', \ell', m'_l \rangle = (-1)^{\ell - m_q} \left( \frac{\ell}{-m_c} \frac{1}{-q} \frac{\ell'}{m'_l} \right) \times \langle \ell | r | \ell' \rangle,$$

where $q$ represents the electric field polarisation ($q = \pm 1$, 0 driving $\sigma^\pm$, $\pi$ transitions respectively), and the braces denote a Wigner–3j symbol ($Wigner3j$). We use Condon–Shortley phase convention [30] for spherical harmonics. The reduced matrix element ($\langle \ell | r | \ell' \rangle$ (getReducedMatrixElement$)$ is given by

$$\langle \ell | r | \ell' \rangle = (-1)^\ell \sqrt{(2\ell + 1)(2\ell' + 1)} \left( \frac{\ell}{0} \frac{1}{0} \frac{\ell}{0} \right),$$

where the radial matrix element (getRadialMatrixElement$)$ is evaluated from

$$\mathcal{R}_{n\ell \to n'\ell'} = \int_0^\infty R_{n\ell}(r) R_{n'\ell'}(r) r^2 \, dr,$$

using numerical integration of the calculated wavefunctions.

Transforming these to the fine-structure basis Eq. (9) can be expressed in terms of states $j, m_j$ (getDipoleMatrixElement$)$ as

$$\langle j, m_j | r_q | j', m'_j \rangle = (-1)^{j - m_q} \sqrt{(2j + 1)(2j' + 1)} \left( \frac{j}{-m_c} \frac{1}{-q} \frac{j'}{m'_j} \right) \times \langle j | r | j' \rangle,$$

where curly braces denote a Wigner–6j symbol ($Wigner6j$).

To account for higher order moments in the interaction between atom pairs (see Section 2.4.1), it is also necessary to calculate quadrupole matrix elements (getQuadrupoleMatrixElement$)$ of the form

$$\mathcal{R}_Q^{j, j'} = \int_0^\infty R_{j\ell}(r) R_{j'\ell'}(r) r^2 \, dr.$$

As with the quantum defect model for the energies, these numerical approaches provide accurate estimates of the dipole and quadrupole terms for highly-excited states but have a large error for low-lying states where the electron density is weighted close to the atomic core where integration is most sensitive to the divergence of the model potential. To overcome this limitation, values for dipole matrix elements available in the literature either from direct measurement or more complex coupled-cluster calculations [31,32] are tabulated (accessible through the function getLiteratureDME$)$.

Before calculating a matrix element, the ARC library first checks if a literature value exists and if so utilises the tabulated value with the smallest estimated error. Otherwise a numerical integration is performed using the method outlined above. For each element, these tabulated values are contained within an easily readable .csv file (file name specified in literatureDME$filename$) that lists the value along with relevant bibliographical reference information, making it easy for users to add new values at a later date.

### 2.2.4. Rabi frequency

An important parameter in experiments with Rydberg atoms is the Rabi frequency $\Omega = d \cdot \vec{E} / \hbar$ where $d$ is the dipole matrix element for the transition and $\vec{E}$ is the electric field of the laser driving the transition. For a transition from state $|n_1, \ell_1, m_{1j} \rangle$ to $|n_2, \ell_2, m_{2j} \rangle$ using a laser with power $P$ and $1/e^2$ beam radius $w$, the Rabi frequency (in rad s$^{-1}$) can be obtained as

$$\Omega = \frac{2 \pi \hbar \ell}{\lambda_0} \frac{P}{w^2},$$

where $\lambda_0$ is the transition wavelength, $P$ is the laser power, and $w$ is the beam radius. The related function atom.getRabiFrequency$($n1,l1,j1,mj1,n2,l2,j2,mj2,P,w$)$ returns the Rabi frequency calculated from the electric field amplitude.

#### 2.2.5. Excited state lifetimes

Radiative lifetimes of alkali atoms can be calculated using dipole matrix elements to determine the Einstein $A$-coefficient [33] for transitions (see getTransitionRate$)$,

$$A_{n\ell \rightarrow n'\ell'} = \frac{4\pi^3}{3c^3} \frac{\ell}{2\ell + 1} \frac{\alpha^2}{\lambda_0^4} \left| \langle j | r | j' \rangle \right|^2,$$

where $\omega_{\text{tot}}$ is the frequency of the transition from $|n\ell\rangle$ to $|n'\ell'\rangle$. The total radiative decay $\Gamma_0$ is then obtained by summing over all dipole-coupled states $|n'\ell'\rangle$ of lower energy,

$$\Gamma_0 = \sum_{n'\ell'} A_{n\ell \rightarrow n'\ell'}.$$

For an environment at finite temperature, it is necessary to account for the interaction of the atom with black-body radiation (BBR) [34]. This causes stimulated emission and absorption which depend on the effective number of BBR photons per mode, $n_{\text{eff}}$, at temperature $T$ given by the Planck distribution

$$n_{\text{eff}} = \frac{1}{e^{h\nu/kT} - 1},$$

multiplied by the $A$-coefficient resulting in a BBR transition rate

$$\Gamma_{\text{BBR}} = \sum_{n'\ell'} A_{n\ell \rightarrow n'\ell'} n_{\text{eff}}.$$

where the summation $n', \ell'$ includes also states higher in energy, since BBR can drive these transitions. Finally, the effective lifetime $\tau$ (see getEffectiveLifetime$)$ is calculated from $1/\tau = \Gamma_0 + \Gamma_{\text{BBR}}$.

Fig. 3 shows the relative contribution to the excited-state lifetime of Rb 30 $S_{1/2}$ due to radiative decay (dominated by low-lying states due to the $\alpha^2$ scaling in Eq. (14)) and black-body decay at 300 K calculated using ARC (for full code see Appendix A). Comparison of our calculated lifetimes [20] with previous work on radiative [33] and BBR induced lifetimes for alkali–metals [34] yields excellent agreement.
2.2.6. Atomic vapour properties

For experiments using thermal vapours of alkali atoms, a number of useful functions are provided for returning atomic vapour pressure (getPressure) or number density (getNumberDensity), as well as the average interatomic distance (getAverageInteratomicSpacing) and atomic speed (getAverageSpeed) at a given temperature. For full listing of functions see Table B.2 and full documentation in Supplemental [19].

2.3. Single-atom calculations

The module calculations_atom_single provides calculations of single-atom properties. It supports plotting of atom energy levels with interactive finding of associated transition wavelengths and frequencies (Section 2.3.1), i.e. Grotrian diagrams, and atom-energy-level shifts in electric fields (Section 2.3.2), i.e. Stark maps.

2.3.1. Level plots

The LevelPlot class facilitates simple plotting of atomic energy levels. It provides an interactive plot for exploring transition wavelengths and frequencies. For example, generation and plotting of the caesium energy level diagram, including ℓ states from S to D for principal quantum numbers from n = 6 to n = 60, can be realised with:

```python
atom = Caesium()
levels = LevelPlot(atom)
# parameters: nmin, nmax, lmin, lmax
levels.makeLevels(6, 60, 0, 3)
levels.drawLevels()
levels.showPlot()
```

This simple example also demonstrates that the more complicated calculations are implemented as a compact class, whose initialisation and methods closely follow naming and stages one would perform in a manual calculation. Yet, working on high abstraction level, one can obtain information directly relevant for research in just a couple of high-level commands [20].

2.3.2. Stark shifts

Calculation of atomic Stark shifts in external static electric fields provides a tool for both precision electrometry and a mechanism for tuning interatomic interactions to a Förster resonance to exploit strong resonant dipole–dipole interactions (Section 2.4.4). To find the energy of the atom in an electric field E applied along the z-axis it is necessary to find the eigenvalues of the system described by the Stark Hamiltonian

\[ H = \hat{H}_0 + \hat{E} \hat{\mathbf{\Delta}}, \]

where \( \hat{H}_0 \) is the Hamiltonian for the unperturbed atomic energy levels and \( \hat{E} \) is the applied electric field. The electric field term causes a mixing of the bare atomic energy levels due to coupling by the Stark interaction matrix elements \( n, \ell, j, m_\ell | \mathbf{E}^2 | n', \ell', j', m_\ell' \) which can be evaluated from Eq. (12) with \( q = 0 \). The selection rules of the Stark Hamiltonian give \( \Delta m_\ell = 0, \Delta \ell = \pm 1 \) such that only states with projection \( m_\ell \) are coupled together, so each Stark map can be constructed by taking basis states with the same \( m_\ell \) value. Following the method of Zimmerman et al. [27], Stark shifts are calculated by exact diagonalisation of the Hamiltonian.

Stark shift calculations are handled using the StarkMap class, which is initialised by passing the appropriate atom class. For a target state \( n, \ell, j, m_\ell \), a range of \( n \) values from \( n_{\text{min}} \) to \( n_{\text{max}} \) with values of \( \ell \) up to \( \ell_{\text{max}} \) is required to define the basis states for the Stark Hamiltonian. Convergence is typically achieved using \( \ell_{\text{max}} \) of 20 and \( n_{\text{max}} = n_{\text{min}} \sim 10 \), however for large applied fields or higher values of \( n \) it will be necessary to increase the basis size to account for the strong mixing of the energy levels. Finally, the Hamiltonian is diagonalised for each value of the electric field (defined in V/cm). As an example, to calculate the Stark map shown in Fig. 4 for the 28 S1/2 \( m_\ell = 1/2 \) state in Cs we include states \( n_{\text{min}} = 23 \) to \( n_{\text{max}} = 32 \) with \( l_{\text{max}} = 20 \) for 600 equidistant electric field values in the range from 0 to 600 V/cm. The corresponding program call is:

```python
calc = StarkMap(Caesium())
# parameters: n, 1, j, mj, nmin, nmax, lmax
calc.defineBasis(28, 0, 0.5, 0.5, 23, 32, 20)
calc.diagonalise(np.linspace(0.,600*1e2,600))
calc.plotLevelDiagram()
calc.showPlot()
```

In interactive mode, the plot can be clicked to obtain the dominant contribution of the basis states within each eigenstate and by default the figure is highlighted in proportion to the fractional contribution of the target state (in this case 28 S1/2 \( m_\ell = 1/2 \)). An alternative option is to highlight the plot proportional to the transition probability for laser excitation from state \( |n', \ell', j', m_\ell' \rangle \) with a laser polarisation driving \( q \) polarised transition (where \( q = \pm 1, 0 \) corresponds to \( \sigma^+, \pi \)). This is enabled using the optional parameter drivingFromState in the call to the diagonalise method of StarkMap. For example, the corresponding code for driving \( \sigma^+ \) transition from 7 S1/2 \( m_\ell = -1/2 \) would be:

```python
calc = StarkMap(Caesium())
calc.defineBasis(28, 0, 0.5, 0.5, 23, 32, 20)
calc.diagonalise(np.linspace(0.,6000,600),
    drivingFromState=[7,0,0.5,-0.5,1])
```

Finally, to evaluate the static polarisability \( \alpha_0 \) of the target state in units of MHz/(V/cm)^2, the getPolarizability method is called on the StarkMap object, i.e.

```python
print("%.5f MHz cm^-2 / V^-2 * calc.getPolarizability()")
```

2.4. Pair-state calculations

The module calculations_atom_pairstate contains classes and methods for the calculation and visualisation of long-range and short range interactions (PairStateInteractions), as well as an automated tool for identifying Förster resonances (StarkMapResonances) for electric field tuning of the interaction potential.
2.4.1. Interatomic interactions

Pair-wise interactions between two atoms with internuclear separation \( R \), and electron coordinates \( \mathbf{r}_1 \) and \( \mathbf{r}_2 \) relative to the respective nuclei, can be expanded in multipole form as [35]

\[
V(R) = \sum_{L_1, L_2 = 1}^{\infty} \frac{V_{1+2}(L_1, L_2)}{R_{1+2}^{L_1+2L_2+1}},
\]

where \( L_1 + L_2 = 2, L_1 + L_2 = 3 \) and \( L_1 + L_2 = 4 \) correspond respectively to dipole–dipole, dipole–quadrupole and quadrupole–quadrupole interactions, and

\[
V_{1+2}(L_1, L_2) = \frac{(-1)^{L_1} 4\pi}{\sqrt{2L_1 + 1} \sqrt{2L_2 + 1}} \times \sum_m \left( \begin{array}{c} L_1 + L_2 \cr L_1 + m \cr L_2 + m \end{array} \right) r_1^{L_1} r_2^{L_2} \times Y_{L_1,m}(\mathbf{r}_1) Y_{L_2,-m}(\mathbf{r}_2),
\]

where \( \binom{n}{m} \) is the binomial coefficient and \( Y_{L,m}(\mathbf{r}) \) spherical harmonics. In Eq. (20) the quantisation axis is oriented along internuclear axis \( \mathbf{R} \). On the other hand, the atomic states’ quantisation axis is not oriented along \( \mathbf{R} \), be described with polar angle \( \theta \) and azimuthal angle \( \phi \), so that inter-atomic axis defines the quantisation direction [29]. The coupling can then be easily calculated between the rotated states \( |s\rangle = w_0(\theta, \phi)|j_1, m_1\rangle \). Note that \( |j_1, m_1\rangle \) in the rotated basis \( |j', m'\rangle \) is now, in general, a superposition of \( m \) states.

This multipole expansion is valid as long as the wavefunctions are described with polar angle \( \theta \) and azimuthal angle \( \phi \). For a real system, whilst the pair-state with the smallest \( \Delta r', r'' \) dominates the resulting interaction shift, it is necessary to consider the effects of all near-resonant pair-states that are coupled by the \( V(R) \) interaction term above. Calculations of these pair-wise interaction potentials are handled by the PairStateInteractions class that is initialised by specifying the element name and target pair-state \( n_1, l_1, j_1, n_2, l_2, j_2, m_j, m_j' \) whose behaviour we want to explore.

\[
cal = \text{PairStateInteractions(\text{atom()}, n_1, l_1, j_1, n_2, l_2, j_2, m_j, m_j')} \]

2.4.2. Long-range limit

At large separation, for off-resonantly \( (\Delta r', r'' \neq 0) \) coupled states, the dipole–dipole interactions dominate to give an interaction of the form \(-C_6/R^6\) van der Waals potential. For a real system, whilst the pair-state with the smallest \( \Delta r', r'' \) energy difference from the pair-states \( |\tau\rangle \) energy for \( \Delta r', r'' < \Delta E \), where \( \Delta E \) is some maximal energy defect that provides a truncation of the basis states. This calculation is performed using the method getC6Perturbatively, returning \( C_6 \) in units of \( \text{GHz} \mu \text{m}^6 \). Users must specify the relative orientation of the atoms, the range of principal quantum numbers for the states used in calculation, and the maximal energy defect. For example for the interaction of two rubidium atoms in 60 \( S_1/2 \), \( m_j = 1/2, m_j = -1/2 \) states, when \( \Delta \theta = \pi/6 \) respect with to the quantisation axis [Fig. 5(a)], the \( C_6 \) interaction term can be calculated from states with principal quantum number differing maximally \( \delta n = 5 \) from the \( n = 60 \), and maximal energy difference in pair-state energies of \( h \times 25 \text{ GHz} \) as

\[
C_6 = \frac{\Delta \langle r''|V(R)|r'\rangle^2}{\Delta r', r''},
\]

where the sum runs over all pair-states \( |r''\rangle \) whose energy differs from the pair-states \( |\tau\rangle \) energy for \( \Delta r', r'' < \Delta E \).

For a real system, the pair-state with the smallest \( \Delta r', r'' \) dominates the resulting interaction shift, it is necessary to consider the effects of all near-resonant pair-states that are coupled by the \( V(R) \) interaction term above. Calculations of these pair-wise interaction potentials are handled by the PairStateInteractions class that is initialised by specifying the element name and target pair-state \( n_1, l_1, j_1, n_2, l_2, j_2, m_j, m_j' \) whose behaviour we want to explore.

\[
cal = \text{PairStateInteractions(\text{atom()}, n_1, l_1, j_1, n_2, l_2, j_2, m_j, m_j')} \]

2.4.3. Exact interaction potential

To evaluate the interaction potential for arbitrary separation [valid down to the Le Roy radius, Eq. (21)], it is necessary to diagonalise the matrix \( V(R) \) containing all the interatomic couplings for each separation \( R \) to obtain exact values of the eigenvalues and eigenstates describing the interaction between atomic pair-states. Due to the strong admixing of states, this process requires careful choice of atomic basis states, including higher order orbital angular momentum states up to \( \ell_{\text{max}} \). For each distance \( R \), the matrix is diagonalised using an efficient ARPACK package provided through Numpy, and the \( n_{\text{e}} \) eigenstates whose eigenvalues are closest to the target pair-state are returned.

Fig. 6 provides an example of the interaction potential for a pair of atoms initially in the \( 60 \ S_{1/2}, m_j = -1/2, 60 \ S_{1/2}, m_j = -1/2 \)
Eq. with orbital angular momentum up to $l$ during the diagonalisation to $l_{\text{max}} = 4$ created using the code

$$\text{calc} \equiv \text{PairStateInteractions} \left( \text{Rubidium}() , 60, 0, 0.5, 60, 0, 0.5, 0.5, -0.5 \right)$$

# parameters: theta, phi, nMax, lMax, maxEnergy
calc. defineBasis (0, 0, 5, 4, 25.e9)
calc. diagonalise (np.linspace (0.5,10.0,200), 150)
calc. plotLevelDiagram()
calc. showPlot()

As with the StarkMap method, the figure is interactive allowing users to determine the dominant composition (expressed in the pair-state basis) of a given eigenstate. The default behaviour is to highlight the eigenvalues proportional to the fractional contribution of the original target pair-state. Alternatively, as shown on Fig. 6, the optional parameter drivingFromState is used in the call to diagonalise to give highlighting in proportion to the relative laser coupling strength from a given state, assuming that one atom is already in one of the two states that make the initially specified pair-state.

By default, PairStateInteractions includes only dipole coupling between the states in calculating level diagrams. Interactions up to quadrupole–quadrupole term [including all terms $L_1 + L_2 \leq 4$ in Eq. (20)] can be included by setting optional parameter interactionUpTo = 2 during the PairStateInteractions initialisation, which can be important for the short-distance structure of level diagram [37]. For evaluation of long-range potential curves this additional term makes only small perturbations to the asymptotic $C_6$ behaviour. However, for accurate determination of the molecular levels of the short-range potential wells it is important to include the higher order multipole terms. Convergence should also be checked by increasing the basis size and re-evaluating the relevant parameters to ensure all the relevant states are included in the calculation basis.

Following diagonalisation of the pair-state interaction matrix, the long-range ($C_6$) and short range ($C_3$) dispersion coefficients can be evaluated using the methods getCSfromLevelDiagram and getC3fromLevelDiagram respectively, which perform a fit to the eigen-energies associated with the state containing the largest admixture of the target state. A method for finding the cross-over distance between van der Waals and resonant dipole–dipole interactions, i.e. van der Waals radius $R_{vdw}$, is also provided (getVdwFromLevelDiagram).

2.4.4. Stark tuned Förster resonances

As outlined in Section 2.4.1, the finite pair-state energy defects $\Delta$ associated with the closest dipole-coupled pair-states lead to a transition from first order resonant dipole–dipole interactions ($\propto \frac{1}{R^3}$) to second order van der Waals ($\propto \frac{1}{R^6}$) at the van der Waals radius. Using external electric fields however, it is possible to Stark shift the pair-states into resonance to obtain long-range $\propto \frac{1}{R^3}$ interactions for all values of $R$, known as a Förster resonance [38–41].

To identify suitable Förster resonances the StarkMapResonances class is used, taking in a pair of target pair-states and performing diagonalisation of the Stark Hamiltonian of Eq. (18) in the pair-state basis. Due to the angular momentum selection rules of the dipole operator $V(R)$, the target pair-state can be dipole-coupled to pair-states with $\Delta m_j = \pm 1, 0$. Following diagonalisation, only pair-states with energy close to the target state being discarded. As the electric field coupling leads to significant mixing of the zero-field pair-states, the algorithm identifies the basis state containing the largest target state fraction at a given electric field strength to test for the closest dipole-coupled pair-states. Finally, an interactive plot routine enables users to identify states that have a Förster
3.1. Optimised Numerov integrator

Integration of the atomic wavefunctions to obtain dipole matrix elements is numerically intensive and by default ARC uses an optimised Numerov integration routine implemented with C extensions for Python (arc_c_extensions). In the unlikely case the precompiled executable arc_c_extensions.so is incompatible with the installed system, please install a C compiler and compile arc_c_extensions.c located in the ARC root folder using by calling from command line

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

Note that path to arc directory should not contain spaces in order for setupc.py script to work. Alternatively, the native Python solver can be used by setting the optional argument `cpp_numerov=False` when initialising the atom class, however this is not recommended for intensive calculations as it results in substantially slower execution.

3.2. Getting started

To initialise the library, use the following code at the start of a Python script or interactive IPython notebook with

```python
# locate ARC Directory
import sys, os
rootDir = '/path/to/root/directory/for/arc'
sys.path.append(rootDir)
os.chdir(rootDir)
# import ARC library
from arc import *
```

This first sets a path to the directory containing ARC package directory on your computer.\(^\text{1}\) This is the recommended way of using the package in a research environment, since users can easily access, check and change the underlying code and constants according to their needs. ARC is now ready for use, and can be tested using the example code above. Numerous additional examples are provided in IPython notebook “An Introduction to Rydberg atoms with ARC” [20].

4. Implementation

4.1. Physical constants

As mentioned above, the atomic properties are encapsulated in classes which contain the relevant atomic properties determined from the literature along with model potential coefficients taken from Marinescu et al. [23] which have been optimised against measured energy levels. For each atom, asymptotic expansions of the quantum defects are used to determine the energy levels of states with high principal quantum number to high accuracy, using measured values of the ionisation energy, Rydberg constant and quantum defects taken from Li [44,22], Na [22], K [22], Rb [45–48], and Cs [49–51]. For the low-lying states, the quantum defects do not accurately reproduce the measured energies and instead energy levels are determined from data in the NIST ASD database [28]. By default, the cut-off between tabulated and calculated energies is determined by the point at which the error in the calculated energy exceeds 0.02%.

---

\(^{1}\) Note that in the directory path, a backslash (\) should be used on Windows machines, instead of forward slash used on UNIX based machines (Linux, MacOS).
4.2. Tracking calculation progress

For tracking progress of calculations, most functions (see documentation [19] for details) provide optional bool arguments, that turn on printing of additional information about calculations. For example, progressOutput prints basic status information, and is recommended for everyday use. If debugOutput is set to True, more verbose information about basis states and couplings will be printed in the standard output.

4.3. Memoization

In order to achieve good performance, memoization is used throughout. That is, when functions receive request for calculation, memorised previous results are checked, and the value is retrieved from memory if it already exists. Both an in-application SQL database (SQLite) and standard arrays are used for this. Memoization is used for dipole and quadrupole matrix element calculations and angular coupling factors, that are independent of principal quantum number of the considered state, including Wigner-J coefficients and WignerD matrices. Single-atom and pair-state calculations will automatically update databases if new values are encountered, speeding up the future calculations. Updating dipole and quadrupole matrix element database can be done manually too, by calling updateDipoleMatrixElementsFile.

4.4. Saving and retrieving calculations

Large self-contained calculations, such as Stark maps or pair-state interaction potentials, are wrapped as classes to enable easy access to the calculations parameters, with associated visualisation and exploration methods to make the results easy to disseminate. An added benefit is that calculations can be saved for future use, as illustrated in the example below:

```python
import PairStateInteractions

from ARC import StarkMatrix

calc = PairStateInteractions(Rubidium(), 60,0,0,5,60,0,0,5, 0.5,0,5)

# do something with initialised calculation

calc.defineBasis(0,0,5,5,25.e9)

calc.diagonalise(np.linspace(0.5,10.0,200),150)

saveCalculation(calc, "myCalculation.pkl")
```

This calculation can then be retrieved at a later time, or even on another machine, and continued, or further explored. For example:

```python
loadSavedCalculation("myCalculation.pkl")

# continue calculation

calc.plotLevelDiagram()
calc.showPlot()
```

4.5. Exporting data to file

If one wants to use the obtained results in another program, for further analysis and calculation, both PairStateInteractions and StarkMap provide exportData method that allows saving data in .csv format, that is human-readable and easy to import in other software tools. As a commented header of the exported files, ARC will record details of the calculation parameters, in human-readable form. Provided that the initial calculation is variable calc, its data is exported by calling

```python
calc.exportData("rootFileName")
```

The program will output list of files (typically three files) storing the calculation data, whose names will be starting with "rootFileName".

4.6. Advanced interfacing of ARC with other projects

In addition to graphical exploration, and exporting relevant data as .csv files, advanced users incorporating ARC in their own projects might want to access directly Stark or pair-state interaction matrices, and corresponding basis states that are generated by defineBasis methods of the corresponding calculation classes. Basis (pair-) states are accessible for both methods in basisStates array. Stark matrix can be assembled as the sum of the diagonal matrix mat1 recording state energies, and the off-diagonal matrix that depends on applied field and can be obtained as product of electric field and mat2. For pair-state interactions, the interaction matrix can be obtained as a sum of interatomic–distance independent matrix matDiagonal, recording energy defects of pair-states, and distance dependent matrices stored in matR, where matR[0], matR[1] and matR[2] store respectively dipole–dipole, dipole–quadrupole and quadrupole–quadrupole interaction coefficients for interaction terms [Eq. (20)] scaling respectively with distance $R$ as $\propto R^{-3}$ and $\propto R^{-5}$. For more details about this, examples of use, and other accessible calculated information, we refer the reader to the detailed documentation [19].

5. Outlook

This paper describes version 1.2 of ARC that can be downloaded from [21]. Supplementary information provides full documentation [19] for functions listed in Appendix B, as well as an IPython notebook [20] that contains more elaborate examples, code snippets and benchmarks against published results. This notebook provides an introduction to most of the capabilities of the ARC, as well as a good starting point for users, providing useful code snippets. Numerous examples also provide quantitative interactive introduction for anyone starting in the field of Rydberg atomic physics. In addition, for quick estimates, when one is in the lab, conference or other meeting, we provide a web-interface to the package [21]. For the latest versions of the package and documentation, please download material from the ARC GitHub page [18].

In the future, the package can be extended to include calculations of dressing potentials [52], magic wavelengths [53], atom-wall interactions [54,55], photoionisation, collisional cross-sections [56], tensor polarisability, molecular bound states [57], effects of magnetic fields, microwave tuning of interactions [17] and other atomic properties. These can be added as calculation tools, in separate classes, built on top of the existing library. Alkaline earth elements can be included too. While these were beyond the scope of the current project, we hope that this project can provide an initial seed for a much bigger community project. The code is hosted on GitHub [18], allowing easy community involvement and improvements. A proposal of some basic philosophy for the development is provided in the documentation [19].

We hope the library will increase accessibility to the existing knowledge. Up to now a lot of relevant information, although in principle derivable from existing literature, required quite lengthy and error-prone calculations. The developed hierarchical object-oriented structure allows one to retrieve relevant information at the appropriate level of abstraction, without the need to deal with lower-level details. In addition, we hope to establish growing code base for common calculations, in the spirit of other open-source community projects like, Numpy and SciPy [58], adding to the growing stack of atomic physics tools like ElecSus [59], The Software Atom [60] and QuTIP [61]. We especially highlight recent related development of C++ program PairInteraction for pair-state calculations [62]. We hope that these efforts will allow more research groups to explore the rich physics achievable when one uses all the available transitions in the atoms. In addition to exploring Rydberg physics, library can be useful for practical
expansion of multi-photon schemes for alkali-atoms excitation [63–65], ionisation [66,67] and upconversion [68–70], as well as state control, e.g. through dressing of the intermediate levels [71] and adiabatic transfers [72].

Acknowledgements

We thank D. Whiting, J. Keaveney, H. Busche, P. Huillery, T. Billam, C. Nicholas, I. Hughes, R. Potvliege and M. Jones for stimulating discussions. This work is supported by Durham University, EPSRC grant EP/N003527/1, FET-PROACT project “RySQ” (H2020-FETPROACT-2014-640378-RYSQ), EPSRC grant “Rydberg soft matter” (EP/M014398/1), DSTL and EPSRC grant EP/M013103/1. Data for the presented examples can be generated by the IPython notebook available in Supplementary material [20]. In the final stages of this project we have been informed by Sebastian Weber about a related efforts on C++ program for pair-state calculations [62].

Appendix A. Example program

Code below generates Fig. 3. For many more examples see Supplemental material [20].

Table B.1
Class and function listing of alkali_atom_functions module.

<table>
<thead>
<tr>
<th>Name (parameters)</th>
<th>Short description</th>
<th>Typical rel. accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlkaliAtom([preferQuantumDefects, cppnumeros])</td>
<td>Implements general calculations for alkali atoms (see Table B.2)</td>
<td>-10^-2</td>
</tr>
<tr>
<td>NumerovBac(innerLimit, outerLimit, kfun, ...)</td>
<td>Full Python implementation of Numerov integration</td>
<td>-10^-6</td>
</tr>
<tr>
<td>loadCalculation(calculation, fileName)</td>
<td>Saves calculation for future use</td>
<td>-10^-6</td>
</tr>
<tr>
<td>saveCalculation(fileName)</td>
<td>Loads previously saved calculation</td>
<td>-10^-2</td>
</tr>
<tr>
<td>printStateString(n, i, j)</td>
<td>Returns state spectroscopic label for</td>
<td>-10^-2</td>
</tr>
</tbody>
</table>

Table B.2
Methods and function listing of alkali_atom_functions AlkaliAtom class. Typical relative uncertainties are obtained from comparison to measured values.

<table>
<thead>
<tr>
<th>Name (parameters)</th>
<th>Short description</th>
<th>Typical rel. accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>getDipoleMatrixElement(n1, l1, ...)</td>
<td>Reduced dipole matrix element (a_e)</td>
<td>-10^-2</td>
</tr>
<tr>
<td>getTransitionWavelength(n1, l1, ...)</td>
<td>Calculated transition wavelength in vacuum (m)</td>
<td>-10^-6</td>
</tr>
<tr>
<td>getTransitionFrequency(n1, l1, ...)</td>
<td>Calculated transition frequency (Hz)</td>
<td>-10^-6</td>
</tr>
<tr>
<td>getRabiFrequency(n1, l1, j1, ..., m1)</td>
<td>Returns a Rabi frequency (angular, i.e. $\Omega = 2\pi \times \nu$) for resonant excitation</td>
<td>-10^-2</td>
</tr>
<tr>
<td>getRabiFrequency2(n1, l1, m1)</td>
<td>Returns a Rabi frequency (angular, i.e. $\Omega = 2\pi \times \nu$) for resonant excitation</td>
<td>-10^-2</td>
</tr>
<tr>
<td>getStateLifetime(n, , j, [,...])</td>
<td>Returns the lifetime of the state $</td>
<td>s\rangle$</td>
</tr>
<tr>
<td>getTransitionRate(n1, l1, j1, n2, ...)</td>
<td>Transition rate due to coupling to vacuum modes (black body included) ($s^{-1}$)</td>
<td>-10^-2</td>
</tr>
<tr>
<td>getReducedMatrixElementJ_symmetric(n1, ...)</td>
<td>Reduced matrix element in $</td>
<td>s\rangle$ basis, defined in asymmetric notation (a_e)</td>
</tr>
<tr>
<td>getReducedMatrixElementJ_symmetric(n1, ...)</td>
<td>Reduced matrix element in $</td>
<td>l\rangle$ basis, defined in symmetric notation (a_e)</td>
</tr>
<tr>
<td>getReducedMatrixElementJ_symmetric(n1, ...)</td>
<td>Reduced matrix element in $</td>
<td>l\rangle$ basis, defined in symmetric notation (a_e)</td>
</tr>
<tr>
<td>getQuadrupoleMatrixElement(n1, ...)</td>
<td>Radial part of the quadrupole matrix element (a_0 e)</td>
<td>-10^-2</td>
</tr>
<tr>
<td>getPressure(temperature)</td>
<td>Vapour pressure at given temperature (Pa)</td>
<td>(1 - 5) x 10^-2</td>
</tr>
<tr>
<td>getNumberDensity(temperature)</td>
<td>Atom number density at given temperature (m^-3)</td>
<td>(1 - 5) x 10^-2</td>
</tr>
<tr>
<td>getAverageInteratomicSpacing(...)</td>
<td>Returns average interatomic spacing in atomic vapour (m)</td>
<td>(1 - 5) x 10^-2</td>
</tr>
<tr>
<td>corePotential(l, r)</td>
<td>Effective charge of</td>
<td>-10^-6</td>
</tr>
<tr>
<td>potential(l, s, j, r)</td>
<td>Potential</td>
<td>-10^-6</td>
</tr>
<tr>
<td>radialWavefunction(..., s, j, r, ...)</td>
<td>Radial part of electron wavefunction</td>
<td>-10^-6</td>
</tr>
<tr>
<td>getEnergy(n, l, j)</td>
<td>Energy of the level relative to the ionisation level (eV)</td>
<td>-10^-6</td>
</tr>
<tr>
<td>getQuantumDefect(n, l, j)</td>
<td>Quantum defect of the level.</td>
<td>-10^-6</td>
</tr>
<tr>
<td>getCTerm(n, l, j, n1, l1, j1, ...)</td>
<td>C6 interaction term for the given two pair-states ($C_6 \times Hz m^6$)</td>
<td>-10^-6</td>
</tr>
<tr>
<td>getCTerm(n1, l1, l1, j1, ...)</td>
<td>C6 interaction term for the given two pair-states ($C_6 \times Hz m^6$)</td>
<td>-10^-6</td>
</tr>
<tr>
<td>getEnergyDefect(n1, l1, j1, ...)</td>
<td>Energy defect for the given two pair-states, $E(r_{ij}) - E(r'_{ij})$ (eV)</td>
<td>-10^-6</td>
</tr>
<tr>
<td>getEnergyDefect2(n1, l1, j1, ...)</td>
<td>Energy defect for the given two pair-states, $E(r_{ij}) - E(r'_{ij})$ (eV)</td>
<td>-10^-6</td>
</tr>
<tr>
<td>updateDipoleMatrixElementsFile()</td>
<td>Updates the file with pre-calculated dipole matrix elements.</td>
<td>-10^-2</td>
</tr>
<tr>
<td>getRadialCoupling(n, l, n1, l1, j1)</td>
<td>Returns radial part of the coupling between two states (dipole, quadrupole) (a_0 e)</td>
<td>-10^-2</td>
</tr>
<tr>
<td>getAverageSpeed(temperature)</td>
<td>Average (mean) speed at a given temperature (m/s)</td>
<td>-10^-6</td>
</tr>
<tr>
<td>getLiteratureDMF(n1, n1, j1, j2, ...)</td>
<td>Returns literature information on requested transition</td>
<td>-10^-6</td>
</tr>
</tbody>
</table>

from arc import *

# ===== GENERATE DATA (ARC) =====

atom = Rubidium()

pqn = [] # principal quantum number
y = [] # rate at T = 0 K
yBB = [] # additional black-body induced transitions

# calculating decay from from 30 S_(1/2) to P_(3/2)
# where n is in range form 5 to 40

for n in range(5, 40):
    pqn.append(n)
    noBBR = atom.getTransitionRate(30, 0, 0.5, n, 1.0, 0.5, temperature=0)
    + atom.getTransitionRate(30, 0, 0.5, n, 1.5, temperature=0)
<table>
<thead>
<tr>
<th>Name (parameters)</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hydrogen([preferQuantumDefects, cpp_numerov])</td>
<td>Properties of hydrogen atoms</td>
</tr>
<tr>
<td>Lithium6([preferQuantumDefects, cpp_numerov])</td>
<td>Properties of lithium 6 atoms</td>
</tr>
<tr>
<td>Lithium7([preferQuantumDefects, cpp_numerov])</td>
<td>Properties of lithium 7 atoms</td>
</tr>
<tr>
<td>Sodium([preferQuantumDefects, cpp_numerov])</td>
<td>Properties of sodium 23 atoms</td>
</tr>
<tr>
<td>Potassium39([preferQuantumDefects, cpp_numerov])</td>
<td>Properties of potassium 39 atoms; alias Potassium(...)</td>
</tr>
<tr>
<td>Potassium40([preferQuantumDefects, cpp_numerov])</td>
<td>Properties of potassium 40 atoms</td>
</tr>
<tr>
<td>Potassium41([preferQuantumDefects, cpp_numerov])</td>
<td>Properties of potassium 41 atoms</td>
</tr>
<tr>
<td>Rubidium85([preferQuantumDefects, cpp_numerov])</td>
<td>Properties of rubidium 85 atoms; alias Rubidium(...)</td>
</tr>
<tr>
<td>Rubidium87([preferQuantumDefects, cpp_numerov])</td>
<td>Properties of rubidium 87 atoms</td>
</tr>
<tr>
<td>Caesium([preferQuantumDefects, cpp_numerov])</td>
<td>Properties of caesium 133 atoms</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name (parameters)</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>makeLevels(nFrom, nTo, lFrom, lTo)</td>
<td>Constructs energy level diagram in a given range</td>
</tr>
<tr>
<td>drawLevels()</td>
<td>Draws a level diagram plot</td>
</tr>
<tr>
<td>showPlot()</td>
<td>Shows a level diagram plot</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name (parameters)</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>defineBasis(n, l, m, nMin, ...)</td>
<td>Initialises basis of states around state of interest</td>
</tr>
<tr>
<td>diagonalise(eFieldList, [units, ...])</td>
<td>Finds atom eigenstates in a given electric field</td>
</tr>
<tr>
<td>showPlot([interactive])</td>
<td>Makes a plot of a stark map of energy levels</td>
</tr>
<tr>
<td>savePlot([filename])</td>
<td>Shows plot made by plotLevelDiagram</td>
</tr>
<tr>
<td>exportData([filename])</td>
<td>Exports StarkMap calculation data</td>
</tr>
<tr>
<td>getPolarizability([maxField, ...])</td>
<td>Returns the polarisability of the state (MHz cm²/V²)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name (parameters)</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>defineBasis(theta, ...)</td>
<td>Finds relevant states in the vicinity of the given pair-state</td>
</tr>
<tr>
<td>getLeRoyRadius()</td>
<td>Returns Le Roy radius for initial pair-state (μm)</td>
</tr>
<tr>
<td>diagonalise(rangeR, ...)</td>
<td>Finds eigenstates in atom pair basis</td>
</tr>
<tr>
<td>plotLevelDiagram()</td>
<td>Plots pair-state level diagram</td>
</tr>
<tr>
<td>showPlot([interactive])</td>
<td>Shows level diagram printed by plotLevelDiagram</td>
</tr>
<tr>
<td>exportData([filename])</td>
<td>Exports PairStateInteractions calculation data</td>
</tr>
<tr>
<td>getC6fromLevelDiagram(...)</td>
<td>Finds C₆ coefficient for original pair-state (GHz μm⁶).</td>
</tr>
<tr>
<td>getC6fromLevelDiagram(...)</td>
<td>Finds C₃ coefficient for original pair-state (GHz μm³).</td>
</tr>
<tr>
<td>getVdWfromLevelDiagram(...)</td>
<td>Finds rᵥdW coefficient for original pair-state (μm).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name (parameters)</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>findResonances(nMin, ...)</td>
<td>Finds near-resonant dipole-coupled pair-states</td>
</tr>
<tr>
<td>showPlot([interactive])</td>
<td>Plots initial state Stark map and its dipole-coupled resonances</td>
</tr>
</tbody>
</table>

```python
# same, now at T=300 K
withBBR = atom.getTransitionRate(30, 0, 0.5, \
    n=1, 0.5, temperature=300.0) + atom.getTransitionRate(30, 0, 0.5, \
    n=1, 1.5, temperature=300.0)
y.append(noBBR)
ybb.append(withBBR-noBBR)

# ====== PLOT(matplotlib) ======
pqn=np.array(pqn)
y = np.array(y)
ybb = np.array(ybb)
plt.bar(pqn-width/2.,y,width=width,color="r")
plt.bar(pqn+width/2.,ybb,width=width,color="g")
plt.xlabel("Principal quantum number, n")
plt.ylabel("Transition rate (s^{-1})")
plt.legend(("Spontaneous decays", "Black-body induced transitions"), 
    fontsize=10)
plt.xlim(4,40)
plt.set_size_inches(5.50,4)
plt.savefig("decays.pdf", bbox_inches='tight')
```
Appendix B. ARC function list

This appendix provides listings of functions, classes and methods contained within the ARC module. For detailed documentation, and more elaborate examples we refer the reader to the documentation in supplemental [19], also accessible via ARC website [21]. (See Tables B.1–B.8.)

Appendix C. Supplementary data

Supplemental material related to this article can be found online at http://dx.doi.org/10.1016/j.cpc.2017.06.015.

References


Table B.8

Function and class listing of wigner module providing support for angular element calculations.

<table>
<thead>
<tr>
<th>Name (parameters)</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wigner3j(j1,j2, ... )</td>
<td>returns Wigner 3j-coefficient</td>
</tr>
<tr>
<td>wigner6j(j1,j2, ... )</td>
<td>returns Wigner 6j-coefficient</td>
</tr>
<tr>
<td>wignerDmatrix(Theta,phi)</td>
<td>Class for obtaining Wigner D-matrix</td>
</tr>
</tbody>
</table>