Computer Physics Communications 240 (2019) 1–14

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

CPC 50th anniversary article

A fresh computational approach to atomic structures, processes and cascades

Stephan Fritzsche *

Helmholtz Institute Jena, Fröbelstieg 3, 07743 Jena, Germany
Theoretisch-Physikalisches Institut, Friedrich-Schiller-Universität Jena, 07743 Jena, Germany

ARTICLE INFO

Article history:
Received 30 October 2018
Received in revised form 3 January 2019
Accepted 15 January 2019
Available online 28 January 2019

This work is dedicated to Walter R. Johnson for his achievements in atomic physics on the occasion of his 90th birthday

Keywords:
Atomic structure calculations
Atomic properties
Atomic processes
Atomic cascade
Relativistic, statistical tensor

ABSTRACT

Electronic structure computations of atoms and ions have a long tradition in physics with applications in basic research, spectroscopy, life sciences and technology. Various theoretical methods (and codes) have therefore been developed to account for the many-particle structure of atoms, from simple semi-empirical estimates to accurate predictions of selected data, and up to highly advanced time-independent and time-dependent numerical techniques. — Here, I present a fresh concept and implementation of (relativistic) atomic structure theory that supports the computation of interaction amplitudes, properties as well as a large number of excitation and decay processes for open-shell atoms and ions across the whole periodic table. This implementation will facilitate also studies on atomic cascades, responses as well as the time-evolution of atoms and ions. It is based on Julia, a new programming language for scientific computing, and provides an easy-to-use but powerful platform to extent atomic theory towards new applications.

1. Introduction

Atomic computations are nowadays required in many different research fields in physics, science and elsewhere, from plasma physics to spectroscopy and metrology, to VUV and x-ray lithography, and up to material science, to mention just a few of them. Apart from developing new theoretical techniques for describing quantum many-electron systems, these computations are needed, for example, in order to interpret spectroscopic data, to make use of astrophysical observations, or simply to enroll the structure of biomolecules and proteins. Indeed, computations on the electronic structure and properties of free atoms and ions have been found an ubiquitous tool to explore the interaction of matter with light fields and particles of various kinds as well as in different physical and chemical environments.

For different applications, of course, quite different requirements arise with respect to the accuracy and complexity of the predicted data, and also regarding the shell structures of the atoms and ions of interest. The quite distinct requirements from different fields gave rise to a large number of methods and codes that have been developed over the last decades, and which are (often less) available to the community. Although some of these methods are highly advanced, more often than not, they were developed by individual researchers and groups and just applied for selected case studies. These methods can therefore neither be easily extended nor combined with each other in order to improve the underlying physics or to support more complex applications. Moreover, many of the so developed programs are (still) implemented in some traditional language, such as Fortran or C, making further extensions to these codes, rapid proto-typing or the use of graphical interfaces rather cumbersome.

Therefore, to further simplify the application of atomic data, I here present a *fresh* computational approach to (relativistic) structure theory that supports atomic calculations of different complexity and sophistication. In particular, I shall provide and explain below the Jena Atomic Calculator (Jac), a first implementation of this concept within the framework of Julia [1]. This toolbox enables one not only to calculate the level structures and properties of free atoms and ions, such as hyperfine or isotope-shift parameters, but also an extensive list of atomic excitation and decay processes as well as multi-step atomic cascades. Jac is based on the multi-configuration Dirac–Hartree–Fock (MCDHF) method [2,3] and has been designed as a high-level Julia implementation in order to deal with quite different requests to atomic structure theory.

In this first version of Jac, emphasis was placed especially on establishing a *language* (for the implementation) close to the underlying formalism. From the very beginning, moreover, I had in mind a much wider range of applications than known from

* Correspondence to: Helmholtz Institute Jena, Fröbelstieg 3, 07743 Jena, Germany.
E-mail address: s.fritzsche@gsi.de.

https://doi.org/10.1016/j.cpc.2019.01.012

other, presently available codes, although no attempt has been undertaken to incorporate all of the previously implemented features. The design of Jac also includes the formulation of a few general guidelines which I followed in the implementation below and which makes the code applicable to all elements of the periodic table and to quite different models, and with the goal to incorporate the dominant relativistic and correlation effects on equal footings into most, if not all, computations.

Indeed, Jac aims to provide a general toolbox for computing relativistic atomic structures, properties and a good deal of different atomic processes. It also helps simulate atomic cascades and the time-evolution of density matrices (statistical tensors). In the next section, I shall first recall some of the present-day requirements to atomic theory, together with a brief account of the relativistic representation of the atomic states within the MCDHF method. This includes the concept of *many-electron* amplitudes and how they can easily be utilized in order to support different kinds of computations. Section 3 then presents further details on the present implementation, starting with a short overview about the program and a few of its features. Here, I shall summarize especially the presently implemented (or partly implemented) many-electron amplitudes, level properties and atomic processes as well as the first steps that were undertaken to incorporate atomic cascades, the time-evolution of statistical tensors and semi-empirical estimates. Finally, a short summary and outlook is given in Section 4.

2. A fresh computational approach to predicting the properties and behavior of atoms

2.1. Present-day requirements to atomic structure theory

The need for (good) atomic data can hardly be overestimated. New requirements arise today not only in the traditional fields, like astrophysics and fusion research, but also in several other, emerging areas. The advancements in developing new light sources, research with free-electron lasers (FEL) at different wavelengths or the recent successes with atomic clocks are just a few examples where further progress will critically depend on atomic theory and our abilities to predict accurate data. Often, moreover, different theoretical data need to be combined with each other and with further information about the given environment.

At FEL and synchrotrons, for example, experiments with inner-shell excited atoms make it necessary to simulate complex (atomic) cascades, including an initial and subsequent excitation of the atoms and ions, in order to understand their interaction with light of variable frequency (ranges), polarization and pulse duration [4–6]. Such cascade processes also frequently affect the diagnostics of plasma [7] or the imaging of complex (bio-) molecules in intense FEL radiation [8].

To address these requirements, a number of sizeable atomic codes have been developed during the last decades, including Civ3 [9], Cowan's code [10], Atsp [11], Grasp [12,13], Ratip [14,15], Fac [16], and several others. In Grasp, for instance, the focus has recently been placed upon the low-lying level structure of atoms and ions and their systematic improvement. These developments nowadays support quite accurate predictions on hyperfine and isotope-shift parameters or transition probabilities of many atoms, while the (accurate) description of atomic processes with one or several electrons in the continuum still remains a great challenge for atomic theory. Apart from excluding *free* electrons from detailed computations, moreover, also the transfer of alignment and polarization in atomic excitation and decay processes, the decay of inner-shell excited atoms, or the response of atoms and ions to incident light pulses can unfortunately not be described (so readily) by the currently available codes.

With the present concept, I wish to overcome many, if not all, of these limitations. In particular, I shall explain and discuss below the Jac program that is able to handle a large number of atomic properties and processes as well as atomic cascades. Based on the MCDHF method and the Dirac–Coulomb–Breit Hamiltonian, this implementation aims to deal quite equally with relativity and electronic correlations as required for heavy atoms and multiply-charged ions, though this first version is restricted to the frequency-independent form of the Breit interaction. Also, as of now, less attention has been placed upon restricted active-space computations, for which only some basic features are presently prepared in Jac. When compared with other codes, however, a much larger flexibility has been achieved for atomic computations of first- and higher-order processes, including double ionization and the scattering of light, on multiple-step atomic cascades or the treatment of polarization and correlation phenomena of atoms in collision with light and electrons. An essential part of the present approach is to set up proper building blocks and a language for complex atomic computations that closely reflects the theory behind the observed behavior of atoms.

2.2. Representation of many-electron states based on Dirac's equation

In the *ab-initio* theory of free atoms and ions, two central keys are (i) the choice of the (many-electron) Hamiltonian and (ii) the representation of the atomic states. If we start from Dirac's theory for the motion of each atomic electron, we readily arrive at the (so-called) *Dirac–Coulomb* Hamiltonian [3]

$$H_{\rm DC} = \sum_i h_{\rm D}(\mathbf{r}_i) \; + \; \sum_{i<j} \frac{1}{r_{ij}}, \tag{1}$$

in which the one-electron Dirac operator

$$h_{\rm D}(\mathbf{r}) = c\,\boldsymbol{\alpha}\cdot\mathbf{p} \; + \; \beta c^2 \; + \; V_{\rm nuc}(r)$$

describes the kinetic energy of the electron and its interaction with the (external) nuclear potential $V_{\rm nuc}(r)$, and where the second term refers to the static Coulomb repulsion between each pair of electrons. Although this Hamiltonian by itself is not bound from below and, hence, requires special care in order to prevent the electrons from falling into the "Dirac sea", it has been shown [17] that a rigorous lower bound to the upper part of its spectrum exists (and similar for the upper bound of the lower part). Here, I shall not provide much further detail about the relativistic theory but refer the reader instead to the literature [2,18].

Instead of the Coulomb repulsion, however, the pairwise interaction between the electrons is for heavy elements often better described by

$$\sum_{i<j} v_{ij} = \sum_{i<j} \left(\frac{1}{r_{ij}} \; + \; b_{ij} \right), \tag{2}$$

the sum of the Coulomb term and the (so-called) *transverse* Breit interaction b_{ij} in order to correct for the magnetic and retardation contributions due to the *relativistic* motion of the electrons. In the long-wavelength approximation for the virtually exchanged photons, $\omega \;\rightarrow\; 0$, the Breit interaction coincides with the *frequency-independent* (and original) Breit operator [19]

$$b_{ij} \quad \longrightarrow \quad b_{ij}^{(0)} = -\frac{1}{2\,r_{ij}} \left[\boldsymbol{\alpha}_i \cdot \boldsymbol{\alpha}_j \; + \; \frac{(\boldsymbol{\alpha}_i \cdot \mathbf{r}_{ij})\,(\boldsymbol{\alpha}_j \cdot \mathbf{r}_{ij})}{r_{ij}^2} \right].$$

Although this zero-frequency approximation to the full transverse interaction neglects contributions $\sim \alpha^4 Z^3$ (as well as those of higher order in αZ), it has been found appropriate for most practical computations of many-electron atoms, since the explicit frequency-dependence gives usually rise to very tiny corrections

only. Together with the Breit operator , one then obtains the *Dirac–Coulomb–Breit* Hamiltonian

$$H_{\rm DCB} = \sum_i h_{\rm D}(\mathbf{r}_i) + \sum_{i<j}\left(\frac{1}{r_{ij}} + b_{ij}\right) \tag{3}$$

which is often applied in first-order perturbation theory to incorporate the relativistic contributions to the electron–electron interaction. The decision about the particular form of the Hamiltonian operator, i.e. of applying either the operator (1) or (3), or even some other approximation to the electron–electron interaction, is usually based upon physical arguments, such as the nuclear charge, the charge state of the ion, the shell structure of the atomic states of interest, etc. [20]. Moreover, the same form of the electron–electron interaction operator (second term) occurs in the computation of all electron emission and capture processes, the electron-impact excitation (or ionization) of atoms and ions as well as for describing several other processes. In the Jac program presented below, the particular form of the Hamiltonian can be specified quite flexibly to account for all necessary relativistic contributions.

Apart from the choice of the Hamiltonian, the details and accuracy of any (atomic) computation critically depend also on the representation of the atomic states and the explicit definition of the many-electron basis. Like in the symmetry-adapted nonrelativistic Hartree–Fock (HF) approximation, an atomic state function (ASF) is written within the MCDHF model as a linear combination of configuration state functions (CSF) with well-defined parity P, total angular momentum J, and its projection M [2],

$$\psi_\alpha(PJM) \equiv |\alpha\, \mathbb{J} M\rangle = \sum_{r=1}^{n_c} c_r(\alpha)\, |\gamma_r PJM\rangle\,, \tag{4}$$

and where γ_r refers to all additional quantum numbers that are needed in order to specify the (N-electron) CSF uniquely. Here, I have introduced also the notation $\alpha\, \mathbb{J} \equiv \alpha J^P$ to just specify an individual level (or ASF) with well-defined total angular momentum and parity as well as all additional quantum numbers α; this particular notation has also been frequently applied within the implementation and the manual of Jac, cf. Ref. [21].

In ansatz (4), n_c denotes the number of CSF and $\{c_r(\alpha)\}$ the representation of the atomic state within the (given) CSF basis. In most standard computations, the set $\{|\gamma_r PJM\rangle\}$ of CSF is constructed as antisymmetrized products of a common set of *orthonormal* orbitals, making use of *jj*-coupling and a proper seniority scheme for the classification of the (antisymmetrized) subshell states [3]. Within the MCDHF method, moreover, both the radial (one-electron) orbital functions and the expansion coefficients $\{c_r(\alpha)\}$ are optimized simultaneously. In Jac, we make use of a finite relativistic B-spline basis in order to represent all one-electron orbitals, and which then automatically provides a spectrum of eigenvalues that can be readily separated into a positive and negative part. Indeed, the use of B-splines replaces the usual non-linear Hartree–Dirac–Fock equations by a set of generalized eigenvalue problems and helps implement also the (so-called) *no-pair* Hamiltonian without the need of explicit projections [22].

While the self-consistent field (SCF) itself is often based upon the Dirac–Coulomb Hamiltonian, the Breit interaction is, if at all, added perturbatively. In practice, the computation of atomic states is performed in a few individual steps that can be summarized as follows [3,23]:

- specification of the nuclear parameters and the angular structure (and extent) of the CSF basis $\{\psi_\alpha(PJM)\}$ that is to be utilized in ansatz (4);
- computations of the spin-angular integrals (so-called angular coefficients) by means of algebraic techniques;
- generation of initial estimates and a SCF for all radial orbitals functions, based on the Dirac–Coulomb Hamiltonian;
- configuration interaction (CI) calculations in order to incorporate further relativistic contributions due to the Breit interaction and the polarization of the quantum electro-dynamical (QED) vacuum into the Hamiltonian matrix and/or to enlarge the CSF basis beyond the given SCF model.

These steps are typically carried out for each atomic multiplet of interest, i.e. set of simultaneously determined ASF, or sometimes also for individual ASF, and are thus required to generate the wave functions that are needed to form all the physically relevant amplitudes below.

2.3. Concept of atomic amplitudes

When compared with other advanced many-electron techniques, such as many-body perturbation theory or the coupled-cluster approach [24], the MCDHF method has the great advantage that it can be readily applied also to (highly) excited and open-shell structures across the whole periodic table. Owing to ansatz (4), moreover, this method is naturally formulated by means of many-electron *amplitudes* (matrix elements) that can be easily combined in order to describe (almost) all properties and behavior of atoms, such as energy shifts, rates, and cross sections for a large number of processes and for quite different experimental scenarios.

In Jac, I make consequent use of these many-electron amplitudes that are obtained by combining atomic bound states of different levels and, often also, of different charge states but then with electrons in the continuum. In fact, these amplitudes help and enable us to use a language that remains very close to the formal theory. This is quite in contrast to most other atomic structure codes, that are usually built from the beginning upon different decompositions of these (many-electron) amplitudes into one- and two-particle (reduced) matrix elements or even into radial integrals, and this is often done *well before* any implementation or coding is carried out. In fact, the different decompositions and large number of notations in atomic physics has in the past impeded a simple comparison of different codes or the (re-) use of distinct program components. In Jac, in contrast, (reduced) amplitudes like $\langle \alpha \mathbb{J} M \,|\mathbb{T}^{(KQ)}|\, \beta \mathbb{J}' M'\rangle$ and $\langle \alpha \mathbb{J} \,\|\mathbb{T}^{(K)}\|\, \beta \mathbb{J}'\rangle$ for some operator $\mathbb{T}$ of rank K, which describe the interaction among the electrons or with external particles and fields, are *central* to most components and help facilitate further extensions of the code. The use of these amplitudes also enables one to even model second- and higher-order processes, once an appropriate (intermediate) basis $\{\alpha_\nu\, \mathbb{J}_\nu,\ \ \nu = 1, \ldots, n_\nu\}$ has been constructed for some given process, or to apply this approach to the time-independent or time-dependent density matrix theory [25,26].

In practice, there are five types of amplitudes that frequently occur in the computation of the electronic structure, properties and processes of atoms and that basically refer to different inner-atomic interactions. All these (reduced) amplitudes can be written in a quite compact form and can be readily accessed within the Jac program, cf. Table 1 in Section 3.1:

1. The *electron–electron interaction amplitudes* $\langle \alpha\mathbb{J} \,\|\, \mathbb{V} \,\|\, \beta\mathbb{J}'\rangle = \langle \alpha\mathbb{J} \,\|\, \mathbb{V} \,\|\, \beta\mathbb{J}\rangle\ \ \delta_{\mathbb{J}\mathbb{J}'}$ refer to the scalar interaction operator [cf. Eq. (2)]

$$\mathbb{V} = \mathbb{V}^{(\rm Coulomb)} + \mathbb{V}^{(\rm Breit)} = \sum_{i<j}\left(\frac{1}{r_{ij}} + b_{ij}\right) \tag{5}$$

that occurs in the Dirac–Coulomb–Breit Hamiltonian (3). These amplitudes are needed very frequently for the computation of the level structure, (Auger) emission and capture processes, electron impact and scattering processes, and in many places elsewhere.

Table 1
Selected data types (`struct`) to represent important building blocks from atomic structure theory. Some of these types occur in different submodules of Jac and, then, often with a slightly different specification. A more detailed description of these data types can be obtained interactively, for instance by `? Level`, recalling the purpose of this struct and the definition of all fields [cf. Appendix].

Struct	Brief explanation
`Atomic.CasComputation`	An individual or a series of systematically enlarged SCF computations.
`Atomic.CasStep`	Single-step of an (systematically enlarged) SCF calculation.
`Atomic.Computation`	An atomic computation of one or several multiplets, including the SCF and CI calculations, as well as of properties or processes.
`Basis`	(Relativistic) atomic basis, including the specification of the configuration space and radial orbitals.
`Cascade.Computation`	Specifies an atomic excitation/decay cascade, including the initial state, allowed processes and the depths of the cascade.
`Cascade.Simulation`	Specifies how a simulation of some cascade (data) has to be done.
`Cascade.Step`	An individual step of a Cascade.Computation that typically combines two ionization states of ions.
`Configuration`	(Non-relativistic) electron configuration as specified by its shell occupation.
`ConfigurationR`	(Relativistic) electron configuration as specified by its subshell occupation.
`EmMultipole`	A multipole (component) of the electro-magnetic field, specified by its parity and multipolarity.
`Level`	Atomic level in terms of its quantum numbers, symmetry, energy and its (possibly full) representation.
`Multiplet`	An ordered list of atomic levels.
`Nuclear.Model`	A nuclear model of an atom to keep all nuclear parameters together.
`Orbital`	(Relativistic) radial orbital function that appears as 'building block' in order to define the many-electron CSF; it is typically given on a (radial) grid and comprises a large and small component.
`Radial.Grid`	Radial grid to represent the (radial) orbitals and to perform all radial integrations.
`Radial.Potential`	Radial potential function.
`Radiative.Channel`	Radiative channel of well-defined multipolarity and gauge.
`Radiative.Line`	Radiative line between two given (initial- and final-state) levels, and along with all of its multipole channels.
`Radiative.Settings`	Settings for computing radiative lines that can be specified (overwritten) by the user.
`Shell`	Non-relativistic shell, such as $1s$, $2s$, $2p$, ...
`Subshell`	Relativistic subshell, such as $1s_{1/2}$, $2s_{1/2}$, $2p_{1/2}$, $2p_{3/2}$, . . .
`Statistical.Tensor`	Statistical tensor $\rho_{kq}(\alpha\mathbb{J},\ \beta\mathbb{J}')$ of given rank k, projection q, and which typically depends on two atomic levels (resonances).

2. The *electron–photon interaction amplitudes* $\left\langle \alpha_f \mathbb{J}_f \left\| \mathbb{O}^{(\mathbb{M})} \right\| \alpha_i \mathbb{J}_i \right\rangle$ are usually handled separately for the different multipole components $\mathbb{M} = \{\mathrm{E1}, \mathrm{M1}, \mathrm{E2}, \ldots\}$ of the radiation field. Here, one needs to carefully distinguish between absorption and emission processes,

$$\left\langle \alpha_f \mathbb{J}_f \left\| \mathbb{O}^{(\mathbb{M},\mathrm{absorption})} \right\| \alpha_i \mathbb{J}_i \right\rangle \equiv \left\langle \alpha_f \mathbb{J}_f \left\| \sum_{k=1}^{N} a^{\lambda}_{k,L} \right\| \alpha_i \mathbb{J}_i \right\rangle = \left\langle \alpha_i \mathbb{J}_i \left\| \mathbb{O}^{(\mathbb{M},\mathrm{emission})} \right\| \alpha_f \mathbb{J}_f \right\rangle^* ,$$

as well as between *bound–bound*, *bound–free* and *free–free* amplitudes, if electron(s) appear in the continuum, in order to ensure the correct phase relation between different multipole components. In addition, $a^{\lambda}_{k,L}$ denotes the electron–photon interaction operator for the annihilation of a photon with multipolarity L and helicity λ. Such electron–photon interaction amplitudes frequently arise in all photoexcitation, ionization and capture processes as well as in the computation of transition probabilities, dielectronic recombination strengths, multi-photon processes, cascades, etc.

3. The *electron–nucleus (hyperfine) interaction amplitudes* $\left\langle \alpha\mathbb{J}\ F \left\| \mathbb{W}^{(K)} \cdot \mathbb{T}^{(K)} \right\| \beta\mathbb{J}'\ F' \right\rangle$ with $K = 1, 2$ and especially the (electronic) $\left\langle \alpha\mathbb{J} \left\| \mathbb{T}^{(1)} \right\| \beta\mathbb{J}' \right\rangle$ and $\left\langle \alpha\mathbb{J} \left\| \mathbb{T}^{(2)} \right\| \beta\mathbb{J}' \right\rangle$ amplitudes refer to the hyperfine interaction of the bound electron density with the magnetic-dipole ($K = 1$) and electric-quadrupole field ($K = 2$) of the nucleus, respectively. Apart from the calculation of atomic hyperfine A and B parameters, these (electronic) amplitudes are required, for instance, to obtain a representation of hyperfine levels within a *IJF*-coupled basis or to explore hyperfine-induced processes [27].

4. The *nuclear recoil amplitudes* $\left\langle \alpha\mathbb{J} \left\| \mathbb{H}^{(\mathrm{NMS})} \right\| \beta\mathbb{J}' \right\rangle$ for the normal mass-shift and $\left\langle \alpha\mathbb{J} \left\| \mathbb{H}^{(\mathrm{SMS})} \right\| \beta\mathbb{J}' \right\rangle$ for the specific mass-shift arise from the (recoil) Hamiltonian

$$\mathbb{H}^{(\mathrm{recoil})}(M) = \mathbb{H}^{(\mathrm{NMS})} + \mathbb{H}^{(\mathrm{SMS})} = \frac{1}{2M} \sum_{ij} \left[\mathbf{p}_i \cdot \mathbf{p}_j - \frac{\alpha Z}{r_i} \left(\boldsymbol{\alpha}_i + \frac{(\boldsymbol{\alpha}_i \cdot \mathbf{r}_i)\mathbf{r}_i}{r_i^2} \right) \cdot \mathbf{p}_j \right],$$

i.e., in lowest-order relativistic approximation ($\sim v^2/c^2$) and in first order of m/M [28,29]. These amplitudes are utilized to calculate the isotope shifts $\delta\nu^{M,M'}$ for individual level energies or the mass-shifts parameters $K^{(\mathrm{MS})} = K^{(\mathrm{NMS})} + K^{(\mathrm{SMS})}$ for isotopic chains in order to help extract, for example, changes in the nuclear charge radii of rare and radioactive isotopes [30,31].

5. *Coulomb excitation (or ionization) amplitudes* $\mathcal{K}_{fi}^{(\mathrm{Coulex})}(\mathbf{q}; \alpha_f\ \mathbb{J}_f M_f,\ \alpha_i\ \mathbb{J}_i M_i)$ arise within the semi-classical approximation from the Lienard–Wiechert potential of the target (or projectile) atoms with nuclear charge Z_t. These amplitudes are given by

$$A^{\mu,+}(r'_k) = \gamma\ \frac{\alpha\, Z_t}{r'_k(t)}\ (1, 0, 0, +\beta_p)\,,$$

as *seen* by the kth projectile (target) electron, with $\gamma = 1/\sqrt{1-\beta_p^2}$ and $r'_k(t) = \left[(x_k - b)^2 + y_k^2 + \gamma_p^2 (z_k - v_p\, t)^2\right]^{1/2}$. These amplitudes can be evaluated most conveniently in momentum space, in which the complicated time dependence $r'(t)$ of the potential can be replaced by an explicit wave number (momentum transfer) dependence. Coulomb (excitation) amplitudes are frequently required for studying the excitation and alignment transfer of ions at storage rings [32].

The central role of these many-electron amplitudes will become more obvious even in Sections 3.3–3.5, where I shall summarize the (partly-) implemented properties and processes in Jac. Moreover, a simple and fast access to these amplitudes is very crucial for the computation of second-order and multi-step processes, the simulation of atomic cascades as well as the time-evolution of density matrices and related observables.

2.4. Design and goals of the present approach. Limitations

The present-day requirements to atomic (structure) theory from above make it first of all desirable to develop a domain-specific high-level *language*, here built on top of Julia, that reflects the underlying formalism as close as possible and which avoids all unnecessary technical slang. Such a language should also facilitate a simple decomposition of a given computational task into coarse-grained but well-designed steps, like pseudocode is often used to provide a high-level description of some program or algorithm.

Here, I aim to design and provide such a *language concept* for performing rather general atomic computations, together with a first implementation of some basic parts of this concept. This goal also includes further guidance how the different entities from atomic theory can be accessed and how the implementation can be extended towards (more) complex processes.

Such a high-level language should first of all be based on properly defined data types which help express the general terms and 'building blocks' of atomic structure and collision theory, *are* utilized for the implementation and which facilitate the communication *with* and *within* the program. Furthermore, the (notion of these) data types should be readily understandable to any atomic physicist without much additional training. A few selected examples for such building blocks (or `struct`'s within the JAC program, see below) refer to

- the nuclear model for some given atom (`Nuclear.Model`), describing the nuclear charge Z, shape, spin I, nuclear moments and all further information, if needed;
- a (sub-) shell (`Shell` or `Subshell`) to simplify communication with the program at input/output time;
- the radial grid (`Radial.Grid`) on which the amplitudes (matrix elements) are evaluated;
- a complete many-electron basis (`Basis`), including the full definition of all relativistic CSF (`CsfR`), their symmetry and radial orbitals (`Orbital`);
- an atomic level $|\alpha\mathbb{J}\rangle$ (`Level`) in terms of its full representation $\{c_r(\alpha)\}$ and the corresponding basis (`Basis`);
- an atomic multiplet (`Multiplet`) as a group of levels that are defined with regard to the same basis;
- an atomic transition (`Line`) or a sequence of such well-specified transitions (`Pathway`).

In practice, there are many more of these structs defined in JAC that help simplify the implementation and will be further discussed in Section 3.2. For example, the constructor `Nuclear.Model(Z)` just specifies a Fermi-distributed nucleus with charge Z, the chosen standard within the JAC program, while `Nuclear.Model(Z, model, mass, radius, spinI, mu, Q)` enables the user to specify all the nuclear parameters in much greater detail. Further constructors are available or can be defined as the need arises at the user's side. Moreover, data types such as `Line` and `Pathway` are typical specific to some particular process and, hence, occur in different modules and with a slightly different specification and constructors.

Apart from a concise and *descriptive* language, further goals of this concept refer to the development of a computational framework that

(a) supports a simple communication with the program;

(b) is easy to use and sufficiently general for a wide range of open-shell structures and applications in atomic physics and related areas;

(c) enables the user to (re-) define the physical units (at input and output time) and constants as well as all (default) settings that are specific to some particular process or application;

(d) helps with the visualization of large data sets as required for the analysis of results and for debugging;

(e) and may thus allow for a (much) larger functionality than previously developed codes.

Below, I shall explain in detail how some of these features are realized within JAC. Although the concept and its implementation has been tried to keep rather general, JAC will be restricted to (atomic) systems and processes, and especially to those for which the atomic states can be expanded properly in a *(finite) basis of time-independent CSF* that are built upon a radial-spherical representation of all atomic orbitals. Thus, the concept explicitly *includes* the computation of atoms in weak and time-dependent fields as long as the level structure of the atoms and ions remains basically intact, while atoms in (very) strong or short-pulse fields are beyond the scope of this concept. It will enable us also to deal in further detail with atomic excitation and decay cascades as well as the time evolution of density matrices.

Of course, the goals and high-level language above cannot be realized without a proper computational framework and language that is fast enough to allow for quantum (many-electron) computations, and which easily supports the implementation of all the necessary building blocks. Here, I apply Julia that has been developed recently and that was designed from the beginning for high performance [1]. Julia is a rich language for providing descriptive data types, and which supports also their interactive use. Indeed, Julia stands for the *combination of productivity and performance* through a deliberate language design and carefully chosen technologies [33]. These technologies include (i) an expressive *type system* that allows optional type annotations to support an offensive code specialization against run-time types; (ii) *multiple dispatch* to dynamically select the most suitable procedure for running the right code at the right time and to structure the programs close to the underlying science; (iii) *Just-In-Time* compilation; (iv) features for *parallelization* like 'remote calls'; (v) *built-in documentation* as well as several others. In contrast to other, object-oriented and 'noun'-based programming languages, Julia appears as a 'verb'-based language in which the generic functions play a more prominent role than the data types. Indeed, Julia has received great popularity during recent years and does not enforce the user to resort to FORTRAN or C for fast computations; cf. [1]. Moreover, Julia's design allows for a gradual learning and improvement of modern scientific computing [33].

Like most modern languages, Julia supports a modular structure of the program that enables one to generate re-usable code, and without enforcing the user to take care about the namespace and the argument lists (of all subsequent calls). In the implementation below, for example, each atomic property and process is associated with a particular module; cf. Sections 3.3–3.5. Especially the modular structure of the code will enable us (or the user) to easily enlarge the number of atomic processes in the future.

2.5. Merging different requests. Kinds of computations

Different fields in physics require atomic computations of different size, sophistication and complexity, from (very) simple estimates to highly correlated computations of individual frequency shifts or cross sections, and up to large sets of interdependent atomic data. Such large data sets are often required for modeling, for example, cascades or plasma processes. In order to deal with and support these requests, we here distinguish and wish to support (partly still in the future) seven *kinds* of computations which can be summarized as follows:

(1) *Atomic computations, based on explicitly specified electron configurations:* This kind refers to the computation of level energies, atomic states and to either one *or* several atomic properties for the levels of a given multiplet. It also helps compute one selected process at a time, if atomic levels from two or more multiplets are involved in atomic transitions.

(2) *Restricted active-space computations (RAS):* This kind concerns systematically-enlarged calculations of atomic states and level energies due to a specified active space of orbitals as well as due to the (number and/or kind of the) virtual excitations that are taken to be into account [23]. Such RAS computations are normally performed *stepwise* by utilizing the orbitals from some prior step.

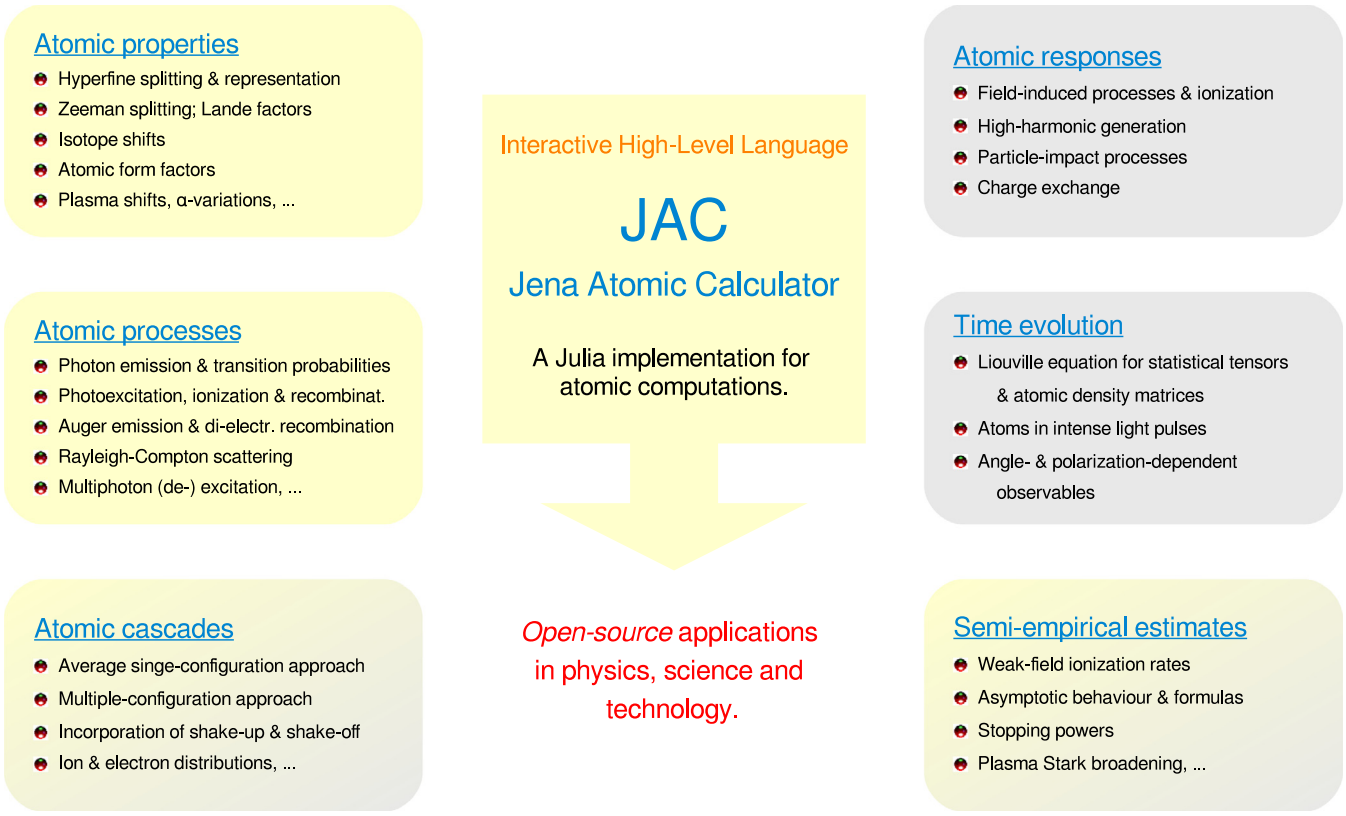

Fig. 1. Overview of the Jena Atomic Calculator (JAC) that supports atomic computations of different kind and complexity. JAC provides an interactive, high-level language to describe and compute a good number of atomic properties and processes as well as for the simulation of atomic cascades (yellow background). Less attention has been paid so far to the evaluation of atomic responses and the time-evolution of density matrices and statistical tensors (gray background). JAC has been set up as open-source project and will soon be made available on Github [34] to the community.

(3) *Interactive computations:* Here, the (large set of) methods of the JAC program are applied interactively, either directly (from the REPL [35]) or by using some short Julia script in order to compute and evaluate the desired observables (parameters), such as energies, expansion coefficients, transition matrices, rates, cross sections, etc. An interactive computation typically first prepares and applies (instances of) JAC's data types, such as orbitals, (configuration-state) bases, multiplets, and others. And like Julia is built on many (high-level) functions and methods, JAC then provides the language elements for performing specific atomic computations at various degree of sophistication.

(4) *Atomic cascade computations:* A cascade typically includes atoms in three or more charge states that are connected to each other by different atomic processes, such as photoionization, dielectronic recombination, Auger decay, radiative transitions, and due to the set-up and geometry of the given experiment. Cascade computations are usually based on some predefined (cascade) approach that enables one to select automatically the atomic processes to be considered for the various steps of the cascade and to specify additional restrictions in order to keep the computations feasible.

(5) *Atomic responses:* With this kind, I wish to support computations that help analyze the response of atoms to incident light pulses and particles, such as field-induced ionization processes, high-harmonic generation and others. For these responses, the detailed atomic structure was often not considered until the present but will become relevant as more elaborate and accurate measurements are carried out.

(6) *Atomic time-evolution of statistical tensors:* We here wish to simulate the population and coherences of (atomic) levels using the Liouville equation, when atoms and ions are irradiated by (intense) light pulses. For these computations, we always assume that the level structure of the atoms is kept intact; further (decay) processes can be taken into account by some *loss* rate but without that the atoms leave the pre-specified space of sublevels. In particular, I here plan to consider (the interplay of) pulses of different shape, polarization strength and duration.

(7) *Semi-empirical estimates of atomic properties,* such as cross sections, stopping powers, asymptotic behavior, etc. These 'estimates' often refer to simple model computations or the use of fit functions. They are only implemented when data are needed but no *ab-initio* computations are feasible otherwise.

2.6. Shaping the language for atomic computations. A first example

It is quite obvious from this short discussion above that, in a first implementation of this concept, not all of these *kinds* of computations can be equally supported right from the beginning. Up to the present, the main focus was therefore placed on (atomic) computations for the level structure and a good number of excitation and decay processes for general open-shell atoms. These computations can (all) be carried out by means of the data type `Atomic.Computation`, a single and quite general data type in the implementation below that enables one to *specify* all the requested details about a particular computation, although the (internally) chosen defaults are often sufficient for many purposes. In addition, I also prepared (and implemented) many other notions and data types in JAC that are required for the simulation of atomic cascades

and the time-evolution of statistical tensors, cf. Fig. 1. These features will be further developed in the future, though schedule and depths may depend also on the interest of the user community.

In Julia, like in other modern programming languages, all instances of individual data types can be readily defined by *constructors*, either the standard constructor or some deliberately designed ones. Such constructors may include even graphical user interfaces (GUI) which save the user from knowing the details of the implementation. While I plan to support such GUI already in the near future, a first example will here be explained by means of a specific but simple constructor of an `Atomic.Computation()` that enables one to (re-) define the details of a computation by calling some other constructors. Owing to the complexity of atomic computations, however, a major goal should be to perform many steps automatically, based on well-chosen default values. This refers especially to the computation of self-consistent fields, the incorporation of relativistic corrections or to the construction of approximate scattering states, if free electrons are involved in some process. Once an `Atomic.Computation` has been specified with just the information that is necessary from a physics viewpoint, it should be simply *performed* and should return all requested results in a nicely tabulated or graphical form.

In practice, many researchers perform atomic computations as part of their, either theoretical or experimental, work. Especially, these users expect a simple communication and control of the program. Apart from proper data types, this can be achieved by using task-dependent default values and by an *active* (verb-based) description of the computation. In Jac, the selection of the level property or process to be calculated is simply made by choosing one (or several) identifiers in `Atomic.Computation()` and by providing the associated `Settings`.

Let us here consider and explain a first example to make the suggested language (and design of Jac) more explicit to the reader. In this example, we wish to calculate the low-lying level structure and Einstein coefficients of the $3s3p^6 + 3s^23p^43d \rightarrow 3s^23p^5$ transition array for Fe^{9+} ions, also known as the spectrum Fe X. This and similar spectra have attracted recent interest as they are observed not only in the Sun but also in a number of other astronomical objects [36]. To perform such a computation within the framework of Jac, one needs to specify the initial- and final-state configurations in an instance of an `Atomic.Computation`, together with the specifier `process=RadiativeX`. We here also provide a title (line), the multipoles (default `E1`) and the gauge forms for the coupling of the radiation field that are to be applied in these calculations; cf. the input at the bottom of this page.

In this input, we could also specify that the computations are made only for transitions with energies in a given interval or for some individual lines that are designated by pairs of initial- and final-state indices (with regard to the level sequence in the corresponding multiplets). This latter feature has been found helpful if configurations with complex shell structure or a large sets of configurations need to be considered. Formally, any number of electron configurations can be provided here to `Atomic.Computation()`, if listed explicitly. Once the details of a computation have been selected, one just needs to `perform(comp::Atomic.Computation)`, and that is already all at user's side.

Jac then performs the necessary steps automatically, including the (independent) self-consistent-field computations for the initial- and final-state multiplets, their diagonalization on the basis of the Dirac–Coulomb–Breit Hamiltonian and, eventually, the calculation of all requested transition amplitudes and probabilities. In addition to the Einstein A and B coefficients, the program also determines the (emission) oscillator strengths and the radiative lifetimes of all considered levels. All these results are tabulated in a neat format both at screen and in a summary file. In these tables, the atomic levels and transitions are usually specified in terms of the level numbers as they arise from the diagonalization of the corresponding Hamiltonian matrix within the Jac program. Although these level numbers are not unique from the viewpoint of physics, especially if highly-excited or inner-shell hole states are considered, they are very helpful to select individual levels and transitions and to simplify the communication of data between different components of the Jac program. Moreover, the (complete) representation of the initial- and final-state multiplets and all the computed radiative lines can be obtained from the function `perform()` if called with the optional argument `output=true`.

Of course, this is a rather simple, though quite typical example of the present concept and its implementation. This example is also part of the tutorials which will be provided together with the code. Computations of other level properties and atomic processes follow very similar lines by just using other identifiers and `Settings` as well as a consistent set of (electron) configurations. For many processes, moreover, quite distinct physical parameters (observables), such as angular distribution and polarization parameters, or angle- and energy-differential cross sections, can be computed in addition to some standard output, but this needs again to be specified through the corresponding settings.

A similar input and communication with the program is planned (and partly implemented already) for all other kinds of computations introduced above, and this simple but common appearance of most input data will make this concept even more powerful.

2.7. Reuse of existing and third-party codes

No implementation of such an elaborate concept can likely be realized successfully without taking advantage of existing and previously established codes (packages). Like other modern languages, Julia is built on modules as separate workspaces to allow for top-level definitions and without the risk of name conflicts, whenever third-party code is used by others, both for *imported* and *exported* functionality. Julia also has a built-in package manager for installing other functionality, written in Julia, and which helps install and maintain external libraries [1]. In the implementation of Jac, I am currently `using` three such registered packages: (i) `GSL.jl`, a Julia interface to the Gnu Scientific Library; (ii) `SpecialFunctions.jl` to compute a few special functions from mathematical physics; and (iii) `Interact.jl` to work with interactive widgets, respectively. For test purposes, moreover, the packages `FortranFiles.jl` for reading unformatted Fortran files from Grasp [13] and `QuadGK` for integration have been applied occasionally.

Besides these Julia packages, the present design (and realization) of Jac has benefited a lot from the experience and implementation of previously developed codes, such as Cowan's code [10], Grasp [12,13], Ratip [15], Fac [16] and various others. Indeed, all these codes help develop (relativistic) atomic structure theory during the last decades and have broadened its range of applications, even if the number and complexity of atomic processes, that can be modeled by these codes, is still quite limited. Therefore, any (new)

```
comp = Atomic.Computation("Energies and Einstein coefficients for the spectrum Fe X",  Nuclear.Model(26.);
                          initialConfigs = [Configuration("[Ne] 3s 3p^6"), Configuration("[Ne] 3s^2 3p^4 3d")],
                          finalConfigs   = [Configuration("[Ne] 3s^2 3p^5")],
                          process = RadiativeX,
                          processSettings = Radiative.Settings([E1, M1, E2, M2], [UseCoulomb, UseBabushkin] )
perform(comp)
```

atomic structure code *should* exploit such existing and specialized code, for instance, in order to deal with the theories of angular momentum or QED. In Jac, we especially need to compute the angular coefficients for the symmetry-adapted CSF [cf. Section 2.2]. To calculate these angular integrals for any pair of CSF and for operators of different rank, the corresponding functions from the Anco library [37] are called by means of the `ccall()` syntax within Julia. Anco has been part of the Ratip code and, with some modifications, also of Grasp. At present, this is the only external (Fortran) library used in Jac, although an analogue interface to the angular library of Grasp-2018 [38] would be desirable in order to benefit from further improvements of this code. Further interest in such *specialized* code(s) will likely arise from the radial integration for particular atomic processes as well as if Jac needs to be parallelized beyond the features provided by Julia.

Indeed, many advanced numerical algorithms and libraries have first been developed in atomic theory during the last decades and, more often than not, implemented in Fortran. The careful implementation of such algorithms, as for the generation of accurate SCF or the diagonalization of large matrices, will make it highly desirable to apply these libraries also within the framework of Jac. However, even if such Fortran and C routines can be directly called by applying the `ccall()` syntax from above, Julia code can be maintained more readily if a well-defined Julia interface will be established by the developers. We shall therefore observe to which extent such interfaces will be *built up*, either by us or the community.

2.8. A open approach for the community

Several of the previously developed atomic structure codes, such as Grasp [13], and Fac [16] have recently been moved towards an object-oriented design by using, for instance, features from Fortran 95 or some script languages. Grasp, for example, is now also available from Github [38]. For the implementation of the present concept, I therefore (also) wish to define and make consequent use of data *objects* close to the underlying theory as well as of graphical user interfaces, tools for visualization of intermediate and final results, and of further concepts from modern scientific computing.

In practice, even the present implementation became already quite sizeable owing to the large number of atomic properties and processes that *are* or are planned to be included into Jac. Further steps concern the simulation of atomic cascades or the time-evolution of statistical tensors, whose implementation is still in its infancy. These (intended) goals make the coding not practicable for a single developer and suggest already from the very beginning to design an *open* community platform instead. This refers (i) to an *open-source* code that is made available on, e.g., Github [34] as a common platform and (ii) to an *open-ended* project whose medium- and long-term development will surely depend on the requests of its users. On Github, I shall therefore provide and develop both the source code of Jac as well as an extensive manual and compendium on the underlying theory [21]. Since it is difficult to predict in detail, whether a particular application will receive enough attention by the community (if at all), some flexibility in the design and specification of the code remains crucial for its overall success.

By making Jac *available* to the public right from the beginning, I like to encourage also contributions from the atomic physics community. In particular, further tests and help with the implementation are very welcome, if the overall style of the program is maintained and if consensus exists how to add new features to the code. Here, the goal *should* be to avoid duplication and inhomogeneity across the package as well as to implement (too) specific features that may cause issues in the future. Such an external support by developers may include incremental improvements as well as multiple approaches for algorithms and modules in order to provide well-tested alternatives, for instance, if some particular approach does not work properly. Moreover, emphasis will be placed first on all those applications that receive enough attention by the community.

3. The Jena Atomic Calculator

3.1. Overview of JAC

The Jac program provides a first implementation of the concept above and helps perform (relativistic) atomic structure calculations of different kind and complexity. In particular, this program has been worked out to compute not only atomic state functions and properties but also the (transition) *amplitudes* for a large number of atomic processes and, hence, the cross sections, rates, angular distributions and other parameters associated with these processes. Fig. 1 displays an overview of the Jac program together with a few details about six of the deliberated (kinds of) computations, even if not all of them are yet equally supported. The first and present focus in developing Jac has been placed upon the (automatic) generation of self-consistent fields, atomic properties and processes as well as on simple simulations of atomic cascades. With some further elaboration, however, Jac will be ready also to support interactively controlled computations of different kind, the time-evolution of statistical tensors as well as a few semi-empirical estimates of required atomic properties.

As mentioned above, Jac implements the MCDHF method based on the Dirac–Coulomb (-Breit) Hamiltonian. It therefore enables one to incorporate both the dominant relativistic and correlation contributions to the electronic structure of free atoms and ions in equal manner. Within the MCDHF method, as usual, the computations are mainly restricted by the size of the wave function expansion and the complexity of the processes under consideration, and these limitations concern first of all the given computer resources. No other restrictions occur with regard to the number of open shells, nor the grid size or the number of atomic transitions or pathways. As in Grasp [13], antisymmetric subshell states with more than two equivalent electrons are supported only for $j \leq 9/2$. Overall, however, Jac is well suited for all elements of the periodic table as well as for multiply and highly charged ions.

Jac has been implemented as a Julia package and can be installed like other packages within this environment. Once installed, it can be loaded simply by `using JAC` and without the need to take care about any compilation, linkage, nor that special libraries need to be available. In Julia, all the required dependencies are installed automatically and dependent on which Julia version is in use. This feature makes it very easy to port the code to different platforms.

Like other modern software projects, Jac is internally built upon a large suite of (sub-) modules and methods that help communicate data with and within the program. A separate module is typically designed for each atomic property (e.g., kind of interatomic interaction) or excitation and decay process. For example, the module `JAC.Hfs` contains the code for the computation of (all) hyperfine-related properties, such as hyperfine A and B parameters of selected levels, their energy splitting, the computation of non-diagonal matrix elements or the generation of atomic hyperfine levels. These hyperfine (-splitted) levels could then be applied to analyze hyperfine-resolved spectra, e.g. high-resolution dielectronic resonance spectra or other hyperfine-induced decay processes. At present, Jac is based on about 45 modules which contain the source code and which, in total, comprise about 900 functions (methods) and 30,000 lines of code. Approximately 80 % of the code will (soon) be made public to the user. In the next section, I shall outline some of the guidelines and features that make Jac appropriate for a wide range of atomic computations.

3.2. Implementation of JAC. Selected details

Jac will certainly grow further in its size and functionality as other kinds of computations and features are added to the project,

cf. Fig. 1. This (expected) size makes it advisable, of course, to follow a common style through most if not all parts of the program in order to ensure a long *life-cycle* of the code. Here, I shall explain some of these guidelines that are realized in Jac and how they (hopefully) help to keep the underlying physics transparent throughout the implementation.

As pointed out above, well-defined data types are very crucial for setting-up a strong and powerful (high-level) language that reflects the theoretical formalism within the program. Table 1 shows a number of selected data types and briefly explains their – more or less – obvious role in the code. Several of these data types (`structs`) are defined in different submodules of Jac and with a slightly different specification then. For example, a `Line` or `Settings` occur in the implementation of various atomic processes, and not only for describing radiative transitions as explained in the example in Section 2.6. In total, there are more than 150 of such `structs` defined within Jac, although the user needs to know only a few of them.

For example, the data type `Level` occurs in (almost) all current applications of Jac and specifies an atomic level $|\alpha\mathbb{J}\rangle$ in terms of its energy, total angular momentum and parity, $\mathbb{J} \equiv J^P$, as well as all of its additional quantum numbers that are needed for a unique specification of the level, cf. Appendix. If necessary, moreover, a particular magnetic sublevel and/or its relative (integer) position within a given multiplet can be specified as well, starting from the lowest level of the multiplet. Any (instance of a) `Level` typically knows also everything about its underlying many-electron basis, e.g., the exact definition of the CSF basis as well as the representation of the radial orbitals and, hence, specifies an atomic state (function) in full detail. Therefore, two selected (instances of) levels, together with some given operator known to Jac, are generally sufficient in order to form and evaluate any many-electron amplitude, without the need to provide any additional information to the program. This internal representation and simple handling of all many-electron matrix elements has been consequently implemented in order to support a large number of atomic properties and processes, and this will enable us to easily extend the code towards new or more elaborate processes.

An `Atomic.Computation` is another important data type that enables the user to *specify* the computation of atomic levels, of one or several atomic properties or just *one* process at a given time. In such an atomic computation, all the required wave functions are then generated automatically by a call to `perform (comp::Atomic.Computation)`, and without requiring further input from the user. The restriction to just *one* process (at a given time) arises very naturally from the fact that most atomic processes, such as the photoionization or Auger emission of atoms, combine different charge states and, hence, require to generate initial- and final-state wave functions, and sometimes even a representation of intermediate-state expansions. Atomic processes which connect more than three atomic charge states are generally handled as (atomic) *cascades*, cf. Section 3.6. For these cascades, however, less observables can typically be calculated by Jac. Here, I shall not display the detailed (and internally rather sizeable) specification of an `Atomic.Computation` (type) that allows the user to set up the properties or process of interest, together with the nuclear model and all the required electron configurations. All details about an atomic computation can be specified explicitly by the user beyond its *defaults* by setting up the proper constructors for the fields or subfields of an `Atomic.Computation`. In practice, however, the defaults are often enough for most applications. A first (and very simple) example of such an `Atomic.Computation` was briefly explained in Section 2.6. In the future, moreover, I intend to provide also a graphical user or web-based interface to an `Atomic.Computation` in order to facilitate its use and to overwrite, if desired, the given default values.

For the calculation of a given atomic property or process, the *flow* of the program follows certain settings that specify all details about the requested computation. Therefore, an individually adapted data type `<module>.Settings` is also defined for all properties and processes. These settings usually enable one to select individual levels, lines or pathways (if not all need to be computed) as well as various physical and technical parameters, such as the multipoles, gauges, etc. All information about the nucleus and its parameters are in contrast contained in objects of `Nuclear.Model`, including the nuclear charge and the shape of the charge distribution, mass number as well as the spin and moments of the nucleus. The use of individually adapted `Settings` for each property and process facilitates to easily implement further details, for instance about the control of some numerical algorithm, to one part of the program, without affecting other parts. Again, the use of graphical interfaces will simplify the application of Jac and will be addressed in the future.

In Jac, atomic units are used throughout for all internal computations. This predefinition facilitates the implementation of the underlying theory as well as the internal data flow in the program, although atomic units are often less suitable for the user. To simplify the communication with and the use of the program, the *units* of time, energy, cross section, rates and others can be predefined by the user. The current (default) settings of the units can be shown interactively by calling `JAC.display("units")`, and these settings can be overwritten by the method `JAC.define()`, if the default settings are not appropriate. For example, a call `JAC.define("unit: energy", "Kayser")` or `JAC.define ("unit: energy", "Hartree")` tells Jac that all energies are subsequently handled either in Kaysers or Hartree (atomic units), and that this selection applies to all input and output of the Jac program, i.e. whenever the user interacts with the code. In addition, `JAC.define()` can be utilized also to globally specify the numerical methods that are applied internally. `JAC.define("method: continuum, spherical Bessel")` declares that all continuum orbitals are to be generated as (simple) spherical Bessel waves.

While the input to Jac can often be *minimized* to just specifying a single `Computation`, the output depends of course on the given input and may be utilized further as input for other, subsequent calculations. This use of output data helps the user to built up complexity. If an `Atomic.Computation` is carried out, all major results are printed (by default) only to screen, while `nothing` is returned otherwise. However, an internal representation of the final results can be readily obtained from a dictionary by calling `dict = perform(comp::Atomic.Computation; output=true)`; here the default has been set (so far) to `output=false`. We expect that use is made of this (dictionary) output in order to combine the results from different atomic computations and to model processes with higher complexity. Until now, however, there has been little need to define (external) file formats in order to communicate data between different applications of the program. Such data files might become useful, if also atomic cascades or time evolutions of density matrices will be supported by the program. A similar request might arise from systematically enlarged computations (the RAS computations from above) for which some proper data types are already prepared but are not (yet) fully supported by the present version of Jac. In general, the input and output of all function or methods of Jac can be obtained interactively by `? <function>`, and this might be further improved by using some proper documentation generator in the future.

Once the installation of Jac is finished or if the code is to be used at a new platform, proper and simple-to-use *tests* are highly desirable. In Julia, this is solved by (so-called) *unit* testing, a simple way to see that the code works as expected. At present, a test suite for Jac helps verifying that all major modules of the code work correctly. The call `> test JAC` prints either `true` or `false`,

Table 2
Selected many-electron (reduced) amplitudes that are accessible within the Jac program. Further details about the call of these amplitudes can be found in the manual [21] or interactively by `?<module>.amplitude`.

Amplitude	Call within Jac	Brief explanation
$\langle \alpha \mathbb{J} \Vert \mathbb{T}^{(1)} \Vert \beta \mathbb{J}' \rangle$, $\langle \alpha \mathbb{J} \Vert \mathbb{T}^{(2)} \Vert \beta \mathbb{J}' \rangle$	Hfs.amplitude	Amplitude for the hyperfine interaction with the magnetic-dipole and electric-quadrupole field of the nucleus.
$\langle \alpha \mathbb{J} \Vert \mathbb{N}^{(1)} \Vert \beta \mathbb{J}' \rangle$	LandeZeeman.amplitude	Amplitude for the interaction with an external magnetic field.
$\langle \alpha_f \mathbb{J}_f \Vert \mathbb{O}^{(\mathbb{M},\text{emission})} \Vert \alpha_i \mathbb{J}_i \rangle$	Radiative.amplitude	Transition amplitude for the emission of a multipole ($\mathbb{M}$) photon.
$\langle \alpha_f \mathbb{J}_f \Vert \mathbb{O}^{(\mathbb{M},\text{absorption})} \Vert \alpha_i \mathbb{J}_i \rangle$	Radiative.amplitude	Transition amplitude for the absorption of a multipole ($\mathbb{M}$) photon.
$\langle (\alpha_f \mathbb{J}_f, \varepsilon\, \kappa)\, \mathbb{J}_t \Vert \mathbb{O}^{(\mathbb{M},\text{ photoionization})} \Vert \alpha_i \mathbb{J}_i \rangle$	PhotoIonization.amplitude	Photoionization amplitude for the absorption of a multipole ($\mathbb{M}$) photon and the release of an electron in the partial wave $\vert \varepsilon\, \kappa \rangle$.
$\langle \alpha_f \mathbb{J}_f \Vert \mathbb{O}^{(\mathbb{M},\text{recombination})} \Vert (\alpha_i \mathbb{J}_i, \varepsilon\kappa) \mathbb{J}_t \rangle$	PhotoRecombination.amplitude	Photorecombination amplitude for the emission of a multipole ($\mathbb{M}$) photon and the capture of an electron in the partial wave $\vert \varepsilon\, \kappa \rangle$.
$\langle (\alpha_f \mathbb{J}_f,\ \varepsilon\, \kappa)\, \mathbb{J}_t \Vert \mathbb{V}^{(\text{Auger})} \Vert \alpha_i\, \mathbb{J}_i \rangle$	Auger.amplitude	Auger transition amplitude due to the electron–electron interaction and the release of an electron in the partial wave $\vert \varepsilon\, \kappa \rangle$.
$\langle \alpha_f \mathbb{J}_f \Vert \sum \exp i\, \mathbf{q} \cdot \mathbf{r}_i \Vert \alpha_i \mathbb{J}_i \rangle$	FormFactor.amplitude	Amplitude for a momentum transfer $\mathbf{q}$.
$\langle \alpha_f \mathbb{J}_f \Vert \mathbb{O}^{(\text{PNC})} \Vert \alpha_i \mathbb{J}_i \rangle$	PNC.amplitude	Parity-nonconservation amplitude.

if all tests were successful or not, and along with a short report (`jac-test.report`) that summarizes the differences with regard to the expected (and internally stored) outcomes of all the tested modules.

Of course, much more could be said about the implementation of Jac; many of these details can be found in the manual to the program that is distributed along with the code [21]. The manual provides the basic notations and the theoretical background of Jac, including the formalism and further technical details for modeling a large number of atomic properties and processes. It also summarizes and explains all the physical parameters (observables) that can be computed by the program and how individual parts of the code can be controlled and utilized by the user. The manual will be updated regularly and, *vice versa*, it will hopefully be enhanced by help of the community by making use of the known features of Github. In the following, I shall briefly summarize some of the applications that are implemented or partly implemented within the Jac program. Emphasis will be placed especially on the those amplitudes, properties and processes that can already be applied, while a good number of further processes and the simulation of cascades and time evolutions are outlined but not yet fully worked out in the manual.

3.3. Computation of atomic amplitudes

As mentioned above, the many-electron amplitudes are central to any implementation in atomic (structure) theory, even if we are interested (only) in transition rates, cross sections or angular parameters. Of course, all these amplitudes can be directly invoked also by the users if the atomic states (levels) and operators are properly specified. Here, we shall not explain the detailed definition and decomposition of these amplitudes into some angular and radial parts, which can be found in the manual [21]. Table 2 shows a few selected (reduced) amplitudes that can be readily accessed within the Jac program. Further interactive help can be obtained in Jac by typing `? <module>.amplitude`.

Apart from those transition amplitudes, that can be evaluated for any given two atomic states (or with just two levels in the case of reduced matrix elements), many atomic amplitudes also include a *summation* over a complete intermediate spectrum or over some essential part of such a spectrum. These second- or higher-order amplitudes often occur in processes that cannot be described in first-order perturbation theory, such as Rayleigh and Compton scattering processes, multi-photon or double-Auger processes. Internally, these amplitudes are computed by performing an explicit summation over the corresponding (many-electron) matrix elements and are then utilized to evaluate cross sections, angular parameters and other atomic data. This explicit summation over a (finite) spectrum underlines again the role of the many-electron amplitudes as central building blocks for both the theory and implementation.

Similar, though formally different many-electron amplitudes are important intermediate results for studying parity- and time-violating (P-odd and T-odd) interactions in atoms. For example, the observation of an electric-dipole moment (EDM) of an atom or ion would unambiguously indicate *new* physics beyond the standard model, since this model predicts only a tiny EDM, about eight orders of magnitude smaller than the present (upper) limit [39]. An atomic EDM could be induced however by different P- and T-odd mechanisms, such as an intrinsic EDM of an electron that interacts and is enhanced by the atomic field, a P-odd and/or T-odd electron–nucleon interaction or an intrinsic EDM of the nucleons [40]. Indeed, accurate predictions of various parity- and time-violating interactions and their comparison with precision measurements make atomic table-top experiments competitive with high-energy physics measurements in probing *CP* violation for hadronic matter. In Jac, we wish to support the computation of a few of these P-odd and T-odd amplitudes, although they are less well tested so far.

3.4. Computation of atomic properties

In most atomic structure codes like Grasp [12,13], emphasis has been placed on the computation of (level) properties, such as level energies, hyperfine A and B or isotope-shift parameters. In Jac, we also support a good number of these and further properties. Table 3 lists the presently (partly-) implemented properties as well as a few others that are planned to be incorporated into Jac in the near future. One or several of these properties can be treated together in a single `Atomic.Computation` for all (or selected) levels from a common CSF basis. To specify the bound states of interest, Jac makes use of the level numbers within a given basis that is generated by using an explicitly given list of electron configurations at input time. At present, no systematically enlarged computations are yet supported for these properties, although such computation can be performed interactively step-by-step or by applying a corresponding Julia script.

3.5. Computation of atomic processes

Atomic processes are typically associated with two or more atomic bound states (levels) and, more often than not, also with different charge states of the atom. Moreover, many of these processes are accompanied by the absorption or emission of photons and/or electrons. Indeed, a large number of such processes are known in atomic physics, including excitation, ionization, recombination and scattering processes. In this first implementation of Jac, I have focused on the computation of different atomic processes in order to support users and spectroscopic

Table 3
In Jac implemented or partly-implemented atomic properties. For these properties, different parameters (observables) can be obtained by performing an `Atomic.Computation(..., properties=[id1, id2, ...])`, if one or more of the given identifiers are specified. For each of these properties, moreover, the corresponding (default) `Settings` can be overwritten by the user to control the computations.

Property	id	Brief explanation.
$\|\alpha\mathbb{J}\rangle \longrightarrow \|\alpha(J)\mathbb{F}\rangle$	HFS	Hyperfine splitting of an atomic level into hyperfine (sub-) levels with $F = \|I-J\|, \ldots, I+J-1, I+J$; hyperfine A and B coefficients; hyperfine energies and interaction constants; representation of atomic hyperfine levels in a IJF-coupled basis.
$\|\alpha\mathbb{J}\rangle \longrightarrow \|\alpha\,\mathbb{J}M\rangle$	LandeJ	Zeeman splitting of an atomic level into Zeeman (sub-) levels; Lande $g_J \equiv g(\alpha\mathbb{J})$ and $g_F \equiv g(\alpha\mathbb{F})$ factors for the atomic and hyperfine levels.
$K^{(\mathrm{MS})},\ F$	Isotope	Isotope shift of an atomic level for two isotopes with masses A, A': $\Delta E^{AA'} = E(\alpha\mathbb{J};\ A) - E(\alpha\mathbb{J};\ A')$; mass-shift parameter $K^{(\mathrm{MS})}$ and field-shift parameter F.
α-variations	AlphaX	Differential sensitivity parameter $\Delta q(\delta\alpha)$ of an atomic level; $\Delta E(\delta\alpha;\ \beta\mathbb{J}),\ \Delta q(\delta\alpha;\ \beta\mathbb{J}), K(\beta\mathbb{J})$.
$F(q;\ \alpha\,\mathbb{J})$	FormF	Standard and modified atomic form factor of an atomic level $\|\alpha\mathbb{J}\rangle$ with a spherical-symmetric charge distribution.
$\omega(\alpha\mathbb{J}) + a(\alpha\mathbb{J}) = 1$	Yields	Fluorescence & Auger decay yields of an atomic level.
$\alpha^{(\mathbb{M})}(L, \omega)$		Static and dynamic (ac, multipolar) polarizibilities.
$E(\alpha\mathbb{J};\ \mathrm{plasmamodel})$	Plasma	Plasma shift of an atomic level as obtained for different but still simple plasma models.
$\|\alpha_i\mathbb{J}_i\rangle \longrightarrow \|\alpha_f\mathbb{J}_f\rangle + \hbar\omega$	EinsteinX[a]	Photon emission from an atom or ion; Einstein A and B coefficients and oscillator strength between levels $\|\alpha_i\mathbb{J}_i\rangle \rightarrow \|\alpha_f\mathbb{J}_f\rangle$ that belong to a single multiplet (representation).

[a] Although the Einstein coefficients are not the property of a single level, we here still support a quick computation of these coefficients by means of the `Einstein` module for (two) levels that are represented by a single CSF basis.

Table 4
In Jac implemented or partly-implemented atomic processes. For *one* process at a time, different parameters (observables) can be obtained by performing an `Atomic.Computation(..., process=id)`, if the corresponding identifier is specified. For this particular property, moreover, the corresponding (default) `Settings` can be overwritten by the user to control the computations.

Process	id	Brief explanation
$A^* \longrightarrow A^{(*)} + \hbar\,\omega$	RadiativeX	Photon emission from an atom or ion; transition probabilities; oscillator strengths; angular distributions.
$A + \hbar\,\omega \longrightarrow A^*$	PhotoExc	Photoexcitation of an atom or ion; alignment parameters; statistical tensors.
$A + \hbar\,\omega \longrightarrow A^{+*} + e_p^-$	PhotoIon	Photoionization of an atom or ion; cross sections; angular parameters; statistical tensors.
$A^{q+} + e^- \longrightarrow A^{(q-1)+} + \hbar\,\omega$	Rec	Photorecombination of an atom or ion; recombination cross sections; angular parameters.
$A^{q+*} \longrightarrow A^{(q+1)+(*)} + e_a^-$	AugerX	Auger emission (autoionization) of an atom or ion; rates; angular and polarization parameters.
$A^{q+} + e^- \rightarrow A^{(q-1)+*} \rightarrow A^{(q-1)+(*)} + \hbar\,\omega$	Dierec	Dielectronic recombination (DR) of an atom or ion; resonance strengths.
$A + \hbar\,\omega_i \longrightarrow A^* \longrightarrow A^{(*)} + \hbar\,\omega_f$	PhotoExcFluor	Photoexcitation of an atom or ion with subsequent fluorescence emission.
$A + \hbar\,\omega \longrightarrow A^* \longrightarrow A^{(*)} + e_a^-$	PhotoExcAuto	Photoexcitation & autoionization of an atom or ion.
$A + \hbar\,\omega_i \longrightarrow A^{(*)} + \hbar\,\omega_f$	Compton	Rayleigh or Compton scattering of photons at an atom or ion; angle-differential and total cross sections.
$A + n\hbar\,\omega \longrightarrow A^*$ or $A^* \longrightarrow A^* + n\hbar\,\omega$	MultiPhoton	Multi-photon (de-) excitation of an atom or ion, including two-photon decay, etc.
$A + Z_p \longrightarrow A^* + Z_p$	CoulExc	Coulomb excitation of an atom or ion by fast, heavy ions; energy-differential, partial and total Coulomb excitation cross sections.
$A + \hbar\,\omega \rightarrow A^* + e_p^- \rightarrow A^{(*)} + e_p^- + \hbar\,\omega'$	PhotoIonFluor	Photoionization of an atom or ion with subsequent fluorescence emission.
$A + \hbar\,\omega \rightarrow A^* + e_p^- \rightarrow A^{(*)} + e_p^- + e_a^-$	PhotoIonAuto	Photoionization of an atom or ion with subsequent autoionization.
$A^{q+} + e^- \rightarrow A^{(q-1)+*} \rightarrow A^{(q-1)+(*)} + \hbar\,\omega \rightarrow A^{(q-1)+} + \hbar\,\omega + \hbar\,\omega'$	DierecFluor	Dielectronic recombination of an atom or ion with subsequent fluorescence.
$e_s^- + A \longrightarrow A^* + e_s^{-'}$	Eimex	Electron-impact excitation of an atom or ion; collision strength.
$A + e_s^- \rightarrow A^* + e_s^{-'} \rightarrow A^{+(*)} + e_s^{-'} + e_a^-$	EimexAuto	Electron-impact excitation and subsequent autoionization of an atom or ion.
$A^{q+*} \longrightarrow A^{(q+1)+(*)} + (e_a^- + \hbar\,\omega)$	RadAuger	Radiative-Auger (autoionization) of an atom or ion.
$A + n\hbar\,\omega \longrightarrow A^{(*)} + e_p^-$	MultiIon	Multi-photon ionization of an atom or ion.
$A + n\hbar\,\omega \longrightarrow A^{(*)} + e_{p_1}^- + e_{p_1}^-$	MultiDoubleIon	Multi-photon double ionization of an atom or ion.
$A^{q+}[\mathrm{nucleus}^*] \longrightarrow A^{(q+1)+*} + e_c^-$	Conversion	Internal conversion, i.e. electron emission due to nuclear de-excitation.

investigations from quite different fields in physics. Table 4 lists the presently (partly-) implemented processes together with a few related ones that might be of direct interest to the community. For each of these processes, various cross sections and parameters can often be determined numerically by performing (again) an `Atomic.Computation` with the corresponding process identifier and `Settings` being specified by the user.

If free electrons are involved in an atomic process, the atomic bound state (of the remaining ion) needs to be combined with continuum orbitals of some given energy ε and (partial-wave) symmetry κ in order to construct so-called scattering states with well-defined total angular momentum and parity within the MCDHF model. In contrast to the bound orbitals, which are generated self-consistently, the continuum orbitals are typically solved within a static potential (of the corresponding ionic core) and for fixed energies as obtained from energy-conservation arguments [41]. At present, however, no attempt has been undertaken to incorporate the continuum (interchannel) interactions in the construction of scattering states [41,42] if one (or more) free electrons occur in some process.

Any rearrangement of the electron density in course of the emission and absorption of particles also results in single-electron orbital functions for the representation of the initial and final states that are not quite orthogonal to each other [43]. This *non-orthogonality* can be accounted for in the evaluation of many-electron matrix elements either by a biorthonormal transformation of orbital sets [44] or by an expansion of the atomic states into Slater determinants [45]. At present, however, *orthogonality*

is assumed in the program for the evaluation of all many-electron amplitudes, apart from the radial integration, although we plan to account for the electron relaxation contributions more carefully in future versions of Jac.

3.6. Computation of atomic cascades

Atomic cascades typically arise from the excitation or ionization of inner-shell electrons and are associated with the subsequent emission of two or more electrons and/or photons. Owing to the advent of free-electron lasers and fourth-generation synchrotrons, such cascade processes have received much attention as they enable one to investigate the electron dynamics of extreme matter states [46]. We here refer to an atomic cascade as a *stepwise* excitation and/or de-excitation of atoms, including photon and electron (emission) processes, but where all the initial, intermediate and final bound states can be well described by stationary theory, e.g. the MCDHF method within Jac.

Any of these stepwise cascades can be understood also by means of (a set of) *pathways*, that is by sequences of atomic bound states that relate the initial level(s) of the cascade with the possible intermediate and final states of the ion. In Jac, any description of a cascade therefore begins from its decomposition into such pathways. In general, however, the number and complexity of these pathways can indeed be huge and, hence, unfeasible to incorporate all of them for an arbitrary atom or ion. In order to deal with such cascades, we therefore distinguish between different approaches that are clearly discernible with regard to their complexity and computational costs. These approaches include:

(a) *Averaged single-configuration approach:* This approach makes use of just one "common set of orbitals" and of configuration-averaged data for all ionization stages of the atom and throughout all simulations. These data are obtained by simply averaging over all fine-structure levels, rates and cross sections. In practice, this averaged approach should be feasible for (almost) all atoms and ions across the periodic table, although little is yet known about its reliability.

(b) *Single-configuration approach:* This approach applies a common set of orbitals for each ionization stage of the atom but calculates, separately for each configuration, an explicit representation of all fine-structure levels. The transition rates and cross sections for all specified processes are then calculated with these (fine-structure) states and are combined to extract all distributions (observables) of interest. Since the number of fine-structure levels increases very rapidly with the number of open shells, this (or a similar) approach has been realized so far, and partly in Jac also, only for the decay of closed-shell atoms with initially a single inner-shell hole, and just for pathways of length two or three. In medium and heavy atoms, in contrast, much longer pathways may occur, and this makes detailed simulation of such cascades still very challenging.

(c) *Multiple-configuration & shake approach:* With this approach, I intend to incorporate the dominant electron–electron correlations by means of configuration mixing between closeby-lying configurations. To this end, all possible configurations in a cascade are *grouped* together for each ionization stage of the atom, and by incorporating additional *shake-up* and *shake-off* configurations. Here, the complexity of almost every cascade will literally 'explode', and some further physical insight into the (de-) selection of configurations will be necessary for successful computations. For each group of configurations, the excitation and decay processes are then handled again for all fine-structure levels like in approach (b) [47]. Until now, very little is known how well and to which extent this approach can be realized for atoms with complex shell structure.

For all of these cascade approaches (except a), the rearrangement of the electron density in course of the cascade as well as the multipole components of the radiation field should be treated later in Jac like for the computation of the individual atomic processes in Section 3.5. Because of the predefinition of these cascade approaches, however, electronic correlations can be included only in a restrictive manner.

In practice, any cascade computation starts from an automatically generated list of electron configurations that is ordered by their energies and will be made available at the beginning of some simulation. It will be desirable here to have also further graphical tools available to display and manually select the configurations of interest, since this is often the starting point to group the configurations together and to avoid both, double counting or the (unintended) omission of relevant levels.

For all cascade approaches above, I shall support first of all the *propagation* of the (occupation) probability from the initial level to energetically lower-lying levels due to photon and electron emission, and eventually up to the final-state ions. The final ion distribution is reached for such a cascade, if this probability distribution becomes constant after a finite number of decay steps and no further autoionization is possible. A graphical representation of the 'probability flux' along the considered pathways of a cascade could be controlled by some resolution parameter and may help experimentalists in the future to recognize the major decay paths of a cascade.

Indeed, one of the foci in developing Jac has been placed upon the simulation of such cascades and the question of how to deal with the associated complexity. Although the overall computational effort is difficult to estimate in general, the three approaches above should be (each) realized such, that their costs are negligible when compared with the *last and most sophisticated* approximation that is still feasible to be carried out for a particular atom. The planned treatment of such cascades will bring Jac to the forefront in studying extreme matter states since it enables one to proceed from simple to quite well-correlated computational models.

3.7. Computation of atomic responses and time evolution of statistical tensors. Interactive use of JAC

Recent advancements in short- and strong-field physics make it possible today to explore the electron dynamics of atoms under extreme conditions. Apart from a remarkable increase in the intensity of light pulses (up to a factor of $10^{8..9}$), their time structure has been improved significantly. Indeed, pulse durations of 0.1–100 femtoseconds are feasible today and help studying nonlinear photoprocesses, which were previously known only from the optical region, now also with soft X-rays [48]. When combined with advanced detector techniques, such as reaction microscopes or velocity map imaging detectors, these novel light sources enable one also to analyze the time evolution of atomic processes [5]. In practice, however, very little is still known how the light pulses affect the level population of atoms or, *vice versa*, how the (distribution of) atomic levels determine shape, phase and the duration of subsequently emitted radiation. Here, Jac might facilitate further investigations owing to its simple access to the wave functions of atoms and ions, and their interaction with the radiation field. This can help, for instance, to manipulate the ionization of atoms or to tailor the emission of high harmonics.

In addition, several free-electron lasers in the ultraviolet and X-ray region, such as Flash, Lcls or Fermi, allow to explore the excitation and ionization mechanisms of atoms at the (sub-) femtosecond scale. Theoretically, such short-pulse ionization phenomena can be described most naturally by means of time-dependent statistical tensors, from which all the relevant observables can then be derived. In Jac, I therefore wish to support the explicit time-evolution of these tensors, based on Liouvillle's equation. This approach is *nonperturbative* and applicable for quite arbitrary geometries and pulses. It is equivalent also, though much more effi-

cient, to solving the time-dependent (many-electron) Schrödinger equation as long as the level structure of the atoms remains fairly undisturbed, and this appears to be especially fulfilled for typical VUV and XFEL pulses (with Keldysh parameters $\gamma \lesssim 1$). Unlike other explicitly time-dependent methods, the use of statistical tensors helps incorporate electron–electron correlation contributions in a systematic fashion into the time-evolution of complex targets. In JAC, so far, I have defined data types for statistical tensors and light pulses of different shapes and polarization, while their time-evolution and relation to useful physical observables has not been implemented yet.

First of all and as pointed out above, however, JAC provides a high-level language that helps decompose atomic computations into well-defined steps, reflecting the underlying theory in a neat format. For such a language, the defined data types and functions are essential elements which facilitate the communication *with* and *within* the program. Of course, all these types and functions can be used also interactively to support applications that are specific to the needs of the user. It is this *interactive use* of JAC's various data types and functions that will help the community to proceed towards computations of higher complexity. Like for most other commands in Julia, an inline documentation of all methods can be obtained by `? function` or `? <module>.function`.

3.8. Semi-empirical atomic estimates

For many applications, and especially for atoms in external fields and environments, *ab-initio* theory needs to be combined with semi-empirical estimates in order to model the experimental setup and observations. Indeed, such an *empirical* treatment has often been found helpful in atomic physics for incorporating either additional effects (not considered within the given theoretical framework) or to make the computations faster or feasible at all. Several empirical models are known from the literature, and they may come along with quite different formal and computational sophistication. Examples refer to the (weak) field ionization of atoms in external electromagnetic fields, the formation of the charge-state distribution in (dilute) plasmas under the influence of strong fields, the Stark broadening of spectral lines in plasmas (i.e. their widths and shift), or the stopping power of electrons and protons in radiation physics and medicine, to recall just a few.

In JAC, we shall support such semi-empirical estimates if this is *desired* by the community. However, no attempt will be made to support many different empirical models from the literature. To support selected estimates in JAC, I defined the data type `Estimation` that currently supports just a very few estimates, for instance, for electron-impact ionization cross sections. Most of these semi-empirical models are based on specialized codes and, hence, are placed in the module `Semiempirical`. For example, the call `estimate("ionization potential", shell, Z)` enables the user to quickly access the binding energies of electron shells for most elements from the periodic table.

4. Summary and outlook

A modern concept for atomic computations of different kind and complexity has been presented here, cf. Fig. 1. Apart from standard (relativistic) calculations of the electronic structure and properties of free atoms and ions, the focus of this concept has been placed on simulations of a large number of atomic processes, cascades or even the time-evolution of atomic density matrices in order to describe atomic behavior within different environments. With the JAC program, built on Julia, I also provide a first implementation of this concept. In particular, all the (currently) implemented level properties and processes can be readily computed by simply performing an `Atomic.Computation`, cf. Section 2.6 and Tables 2 and 3. Moreover, the program JAC will soon be made public and will then hopefully support further developments in computational atomic physics.

Since JAC's very first design, the number of properties and processes, that this code can handle, has steadily grown and will make it powerful for applications in atomic photoionization and electron spectroscopy, the computation of heavy and superheavy elements, or the generation of atomic data for astro and plasma physics. Already at present, about 20 different atomic processes have been (partly) implemented in JAC, including various two-step electron capture and decay mechanisms as well as *excitation-with-subsequent-decay* processes, cf. Table 4. A good number of further processes are planned to be incorporated into JAC in the forthcoming years to account for either two electrons in the continuum as well as the creation and annihilation of electron–positron pairs. For many-electron atoms and ions, any accurate prediction for these mechanisms are still a great challenge for atomic theory as they require the evaluation of *free–free* transitions amplitudes.

Of course, I shall continue the effort to further increase the range of applications of JAC. In particular, I presently work on some graphical interface in order to make the `Atomic.Computation`'s easier to apply, to support systematically enlarged wave function expansions and to simulate atomic cascades. An improved treatment of cascade processes, in particular, will pave the way for the direct analysis of ongoing experiments at synchrotrons or XFEL. Moreover, a simple and reliable treatment of enlarged wave function expansions for the computation of hyperfine structures and isotope shifts of medium and heavy elements will help derive nuclear parameters, such as nuclear moments or changes of the nuclear charge radii. In the future, for sure, also other types of computations will be supported by JAC following our own needs and the interests of the community.

Obviously, however, the scope of this concept is much larger than what I can (and plan to) implement myself here in Jena. Therefore, JAC will soon be made available on Github [34], and I will there provide both the source code of JAC as well as an extensive manual and compendium on the underlying theory [21]. With its upload to Github, I wish also to encourage the users to fork the code and to report improvements, failures, bugs, etc. Using this platform, non-trivial changes to the code can be made via pull requests, i.e. by submitting code for review by other users prior their merger with the master code. Although a good number of tests have been made on JAC, this is still a *first* implementation, and no code is error free. Furthermore, in order to ensure a good stability of the code, we aim for *continuous integration*, a process that stands for running tests regularly and on different platforms. I shall thus appreciate reports from the users if problems are encountered or, more helpful, if solutions are provided. One of the simplest way to start contributing to JAC is writing a tutorial, in addition to those provided by JAC, to navigate others to the task of a new user. Also, new plotting features on different outcomes of atomic computations will be very helpful for the community. Moreover, further suggestions can be found by calling `JAC.todo()`.

While extensions to an existing code might first be made often in response to some given experiment and for some particular atom or ion of interest, JAC's framework is flexible enough to later generalize these computations also to other atoms and shell structures. — With this concept and implementation, I therefore overall hope to enlarge the number of atomic (structure) application in physics, science and technology.

Acknowledgments

In designing and implementing JAC, I have benefited from and like to acknowledge the support and discussion with Randolf Beerwerth, Sebastian Stock and Andrey Surzhykov.

Appendix. The `Level` struct

Proper data types are a key to any modern implementation of atomic theory. In Jac, the `strut Level` has been found important to comprise all information about a level, such as the total angular momenta and parity, its energy and representation in terms of a given many-electron basis and further information. Like for all other data types, its internal representation can be obtained from calling `? Level`:

```
‘struct  Level‘  ... defines a type for an atomic level in terms of its quantum numbers, energy and with regard to a
                     specified relativistic basis.

  + J             ::AngularJ64       ... Total angular momentum J.
  + M             ::AngularM64       ... Total projection M, only used if a particular sublevel is referred to.
  + parity        ::Parity           ... Parity of the level.
  + index         ::Int64            ... Index of this level in its original multiplet, or 0.
  + energy        ::Float64          ... energy
  + relativeOcc   ::Float64          ... Relative occupation of this level, if involved in the evolution of a cascade.
  + hasStateRep   ::Bool             ... Determines whether this level 'points' to a physical representation of an state
                                         (i.e. an basis and corresponding mixing coefficient vector), or to just an
                                         empty instances of a basis, otherwise.
  + basis         ::Basis            ... relativistic atomic basis
  + mc            ::Vector{Float64}  ... Vector of mixing coefficients w.r.t basis.
```

References

[1] https://docs.julialang.org/ (13/2/2019).
[2] I.P. Grant, in: S. Wilson (Ed.), Relativistic Effects in Atoms and Molecules, in: Methods in Computational Chemistry, vol. 2, Plenum New York, 1988, p. 1.
[3] I.P. Grant, Relativistic Quantum Theory of Atoms and Molecules: Theory and Computation, Springer, 2007.
[4] A. Karamatskou, et al., Phys. Rev. A 89 (2014) 033415.
[5] M. Kurka, et al., J. Phys. B 42 (2009) 141002.
[6] S. Schippers, et al., J. Phys. B 48 (2015) 144003.
[7] L. Godbert, et al., Phys. Rev. E 49 (1994) 5644.
[8] K. Ayyer, et al., Struct. Dyn. 2 (2015) 041702.
[9] A. Hibbert, Comput. Phys. Comm. 9 (1975) 141.
[10] R.D. Cowan, The Theory of Atomic Structure and Spectra, University of California Press, Berkeley, 1981.
[11] C. Froese Fischer, G. Tachiev, G. Gaigalas, M. Godefroid, Comp. Phys. Commun. 176 (2007) 559.
[12] I.P. Grant, et al., Comput. Phys. Comm. 21 (1980) 207.
[13] P. Jönsson, et al., Comput. Phys. Comm. 184 (2013) 2197.
[14] S. Fritzsche, J. Elec. Spec. Rel. Phenom. 114–116 (2001) 1155.
[15] S. Fritzsche, Comput. Phys. Comm. 183 (2012) 1525.
[16] M.F. Gu, Astrophys. J. 582 (2003).
[17] J. Sucher, Rep. Progr. Phys. 41 (1978) 1781.
[18] W.R. Johnson, Atomic Structure Theory: Lectures on Atomic Physics, Springer, 2007.
[19] G. Breit, Phys. Rev. 39 (1932) 616.
[20] S. Fritzsche, Phys. Scr. T 100 (2002) 37.
[21] S. Fritzsche, Jac: Manual, Compendium & Theoretical Background, unpublished, 2018.
[22] O. Zatsarinny, C. Froese Fischer, Comput. Phys. Comm. 202 (2016) 287.
[23] C. Froese Fischer, M. Godefroid, T. Brage, P. Jönsson, G. Gaigalas, J. Phys. B 49 (2016) 182004.
[24] I. Lindgren, Relativistic Many-Body Theory, in: Springer Series on Atomic, Optical, and Plasma Physics, vol. 63, New York a. o., 2011.
[25] V.V. Balashov, A.N. Grum-Grzhimailo, N.M. Kabachnik, Polarization and Correlation Phenomena in Atomic Collisions, Kluwer Academic Plenum Publishers, New York, 2000.
[26] N.M. Kabachnik, et al., Phys. Rep. 451 (2007) 155.
[27] Z.W. Wu, et al., Phys. Rev. A 90 (2014) 052515;
Z.W. Wu, et al., Phys. Rev. A 91 (2015) 056502.
[28] V.M. Shabaev, A.N. Artemyev, J. Phys. B 27 (1994) 1307.
[29] E. Gaidamauskas, et al., J. Phys. B 44 (2011) 175003.
[30] F.C. Charlwood, et al., Phys. Lett. B 690 (2010) 346.
[31] B. Cheal, T.E. Cocolios, S. Fritzsche, Phys. Rev. A 86 (2012) 042501.
[32] A. Surzhykov, et al., Phys. Rev. A 73 (2006) 032716;
A. Surzhykov, et al., Phys. Rev. A 77 (2008) 042722.
[33] J. Bezanson, et al., SIAM Rev. 59 (2017) 65.
[34] https://www.github.com/OpenJAC/JAC.jl (13/2/2019).
[35] Julia comes with a full-featured interactive command-line REPL (read-eval-print loop) that is built into the executable of the language.
[36] C.Z. Dong, et al., Mon. Notes R. Astr. Soc. 307 (1999) 809;
C.Z. Dong, et al., Mon. Notes R. Astr. Soc. 318 (2000) 263.
[37] G. Gaigalas, S. Fritzsche, I.P. Grant, Comput. Phys. Comm. 139 (2001) 263.
[38] C. Froese Fischer, G. Gaigalas, P. Jönsson, J. Bieron, Comput. Phys. Comm. 237 (2019) 184, https://github.com/compas/grasp2018 (13/2/2019).
[39] B. Graner, et al., Phys. Rev. Lett. 116 (2016) 161601.
[40] J.S.M. Ginges, V.V. Flambaum, Phys. Rep. 397 (2003) 63.
[41] S. Fritzsche, B. Fricke, W.-D. Sepp, Phys. Rev. A 45 (1992) 1465.
[42] T. Å.berg, G. Howat, W. Mehlhorn (Ed.), Corpuscles and Radiation in Matter I, Encyclopedia of Physics, XXXI, Springer, Berlin, p. 469.
[43] S. Fritzsche, C. Froese Fischer, Comput. Phys. Comm. 99 (1997) 323.
[44] J. Olsen, et al., Phys. Rev. E 52 (1995) 4499.
[45] S. Fritzsche, I.P. Grant, Comput. Phys. Comm. 92 (1995) 111.
[46] J. Andersson, et al., Phys. Rev. A 92 (2015) 023414;
J. Andersson, et al., Phys. Rev. A 96 (2017) 012505.
[47] S. Stock, R. Beerwerth, S. Fritzsche, Phys. Rev. A 95 (2017) 053407.
[48] M. Meyer, et al., Phys. Rev. Lett. 104 (2010) 213001.