Computer Programs in Physics

A neural-network-based Python package for performing large-scale atomic CI using pCI and other high-performance atomic codes

Pavlo Bilous [a], [iD],[*], Charles Cheung [b], Marianna S. Safronova [b]

[a] *Max Planck Institute for the Science of Light, Staudtstr. 2, 91058 Erlangen, Germany*
[b] *Department of Physics and Astronomy, University of Delaware, DE 19716, USA*

ARTICLE INFO

ABSTRACT

Modern atomic physics applications in science and technology pose ever higher demands on the precision of computations of properties of atoms and ions. Especially challenging is the modeling of electronic correlations within the configuration interaction (CI) framework, which often requires expansions of the atomic state in huge bases of Slater determinants or configuration state functions. This can easily render the problem intractable even for highly efficient atomic codes running on distributed supercomputer systems. Recently, we have successfully addressed this problem using a neural-network (NN) approach [1]. In this work, we present our Python code for performing NN-supported large-scale atomic CI using pCI [2] and other high-performance atomic codes.

Program summary
Program Title: nn_for_pci
CPC Library link to program files: https://doi.org/10.17632/yy29nhwkbw.1
Developer's repository link: https://github.com/pavlobilous/nn_for_pci
Licensing provisions: GPLv3
Programming language: Python
Nature of problem: Exponential scaling of the basis size in the atomic CI approach
Solution method: Iterative NN-based selection of the relevant basis elements out of a large CI basis

References
[1] P. Bilous, C. Cheung, M. Safronova, Phys. Rev. A 110 (2024) 042818.
[2] C. Cheung, M.G. Kozlov, S.G. Porsev, M.S. Safronova, I.I. Tupitsyn, A.I. Bondarev, Comput. Phys. Commun. 308 (2025) 109463.

1. Introduction

Advanced applications of computational atomic physics, such as the development of novel atomic clocks or studies of astrophysical spectra, require ever greater precision and, therefore, highly accurate modeling of electronic correlations. The high-precision treatment of electronic correlations can be performed within the configuration interaction (CI) framework [1], in which the electronic many-body wave function is expanded in a given basis, usually Slater determinants or configuration state functions (CSFs). Unfortunately, in this approach, higher precision implies an exponential increase of the number of needed basis elements,

quickly leading to a numerically intractable eigenvalue problem, even with highly parallelized codes running on advanced supercomputers. This requires the development of approximation methods, in particular, based on *a priori* selection of the most important basis states.

In our recent work [2], we presented a neural network (NN) approach to running large-scale atomic CI computations. Following the developments in Refs. [3–6], we introduced important algorithmic improvements that made the method suitable for large computations of atomic spectra without considerable overhead for NN operations. Coupled with the recent release of the pCI atomic structure codes [7], our NN-based approach proved to be a powerful tool to tackle the expo-

---

* Corresponding author.
  E-mail address: pavlo.bilous@mpl.mpg.de (P. Bilous).

Fig. 1. NN-supported computation iteration.

Table 1
The versions of Python and the third-party libraries as used in the present work.

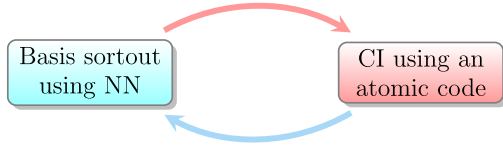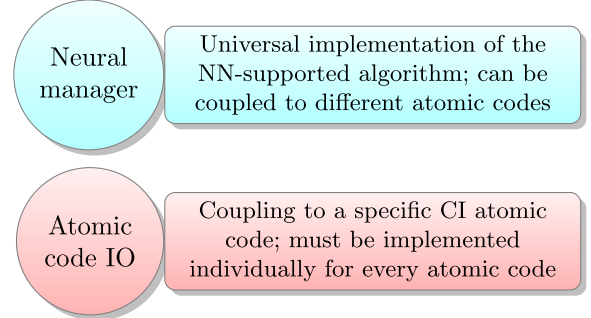| Python	NumPy	Pandas	TensorFlow
3.10.9	2.0.2	2.2.3	2.18.0

Fig. 2. The subpackages of the presented Python package.

nential growth of the CI basis, which is undoubtedly one of the most challenging problems in high-precision atomic computations. In this work, we aim to enable such NN-supported CI calculations as presented in Refs. [2,3] to the atomic physics community. We present our Python package implementing the NN-based selection algorithm, along with a demonstration example of a NN-supported CI computation.

The computation consists of two-block iterations shown in Fig. 1: (1) the Python code selects the relevant subbasis from the CI basis using the NN, and (2) an atomic code performs CI with the NN-selected subbasis. While the Python codes are in principle suitable for supporting any CI atomic code, we demonstrate the NN support for CI computations performed by the recently published pCI atomic code package [7,2]. The pCI codes are suitable for extremely large parallelized CI calculations [8] and have demonstrated exceptional accuracy for a wide variety of many-electron systems, including neutral atoms, highly charged ions, and negative ions [9–15]. We also present how the NN algorithm can be implemented with any other atomic code performing CI.

The article is structured as follows. In Sec. 2, we describe the NN-supported algorithm. Sec. 3 is devoted to the installation and description of the corresponding Python codes. In Sec. 4, we provide an example of a direct CI computation with the pCI codes. Then in Sec. 5, we demonstrate how the same example is solved using our implementation of the NN algorithm. In Sec. 6, we demonstrate how the presented NN support can be coupled with any atomic code performing CI. The article is concluded with a summary in Sec. 7.

2. NN-supported CI algorithm

In the CI approach [1], the wave function is represented as an expansion $|\Psi\rangle = \sum_\alpha c_\alpha |\alpha\rangle$ in a fixed basis $\mathcal{B} = \{|\alpha\rangle\}$, where $|\alpha\rangle$ are typically Slater determinants or CSFs. The unknown expansion coefficients c_α are then obtained by forming the Hamiltonian with matrix elements $H_{\alpha\beta} = \langle\alpha|\hat{H}|\beta\rangle$, and then solving the eigenvalue problem on the basis $\mathcal{B}$. Obtaining high-precision results for complex atomic systems typically requires a huge basis that may render the CI computation intractable. In this case, our NN approach can be used to replace the CI computation by a number of smaller ones on an iteratively growing subbasis $\mathcal{B}_i \subset \mathcal{B}$ managed by the NN.

In general, the NN first trains on the data obtained in iteration $i-1$, and then classifies the candidate basis elements from $\mathcal{B} \setminus \mathcal{B}_{i-1}$ as "important" or "unimportant". A basis element $|\alpha\rangle$ is considered "important" if its weight in the CI expansion $w_\alpha = |c_\alpha|^2$ exceeds a cut-off x_i, set by the user in each iteration. The basis states classified by the NN as important are included in the subbasis $\mathcal{B}_i$, on which a CI calculation with an atomic code is then performed. The latter yields the true CI expansion coefficients, which are subsequently used to give the NN feedback on its performance and retrain it in the next iteration. The basis elements which turned out to be unimportant are excluded from $\mathcal{B}_i$. The cutoff for the next iteration is chosen as $x_{i+1} < x_i$. These iterations are repeated until the corresponding energies obtained at the CI calculation step are converged or until the available computational resources are exhausted.

Crucial improvements were introduced in Ref. [2] to the described procedure, which enabled the use of the NN-supported approach in large-scale CI for atomic spectra without considerable overhead for the NN operation. We highlight these here:

- We treat basis states (Slater determinants or CSFs) not separately but in groups of relativistic configurations. The weight of a relativistic configuration Γ is the total weight of its determinants: $w_\Gamma^{(k)} = \sum_{|\alpha\rangle \in \Gamma} w_\alpha^{(k)}$, where k labels the energy levels.
- Computations are performed not for one energy level, but for a number of energy levels at once. The "compound" weight of a relativistic configuration w_Γ used in the NN-supported algorithm is given as $w_\Gamma = \max_k w_\Gamma^{(k)}$.
- We assume a smaller-scale (direct or NN-supported) computation to have been performed prior, which is used in the first iteration of the presented algorithm, in which the NN is not yet trained.

3. Installation and structure of the package

The Python package presented in this work can be cloned directly from the GitHub repository https://github.com/pavlobilous/nn_for_pci. It employs NumPy [16] and Pandas [17] for efficient processing of data arrays, as well as TensorFlow [18] for leveraging the NN functionalities. The version of Python and those of the respective libraries used in this work are summarized in Table 1. We also tested the package with the recently released TensorFlow version 2.19.0 and the NumPy version 2.1.3.

The Python package consists of two subpackages illustrated in Fig. 2. The "Neural manager" subpackage contains the tools implementing the NN-supported algorithm from Ref. [2]. While it is universal and independent of an atomic code, it relies on the coupling functionalities implemented in the second subpackage "Atomic code IO." The purpose of the latter is writing the input for the atomic code and reading its output. We stress that the "Atomic code IO" does not run the atomic code. Instead, this step has to be performed by the user explicitly. This subpackage must be implemented individually for every atomic code by following an abstract interface, as discussed in Sec. 6. We provide here an "Atomic code IO" implementation for the pCI atomic codes which can be used out-of-the-box and serves as an example for other implementations.

The installation guide for the pCI package can be found in Ref. [7]. The base installation includes all programs required to perform the CI computations for the desired number of low-lying energy levels. In summary, the `hfd` and `bass` programs are responsible for the construction of the set of one-electron basis orbitals following a recurrent procedure described in Refs. [19,20], and the `add`, `pbasc`, and `pconf` programs realize the CI algorithm. The Python scripts included in the distribution of the pCI package automate these steps. We provide only the necessary

information in the context of the present work, and refer to Ref. [7] for further details on the pCI atomic code package.

4. Example of direct CI using pCI codes

Here, we introduce an example of a direct CI computation using the pCI codes. We will use then the same example to demonstrate the NN-based approach. We include the input and output files in the directory `example` of our GitHub repository. All file paths in this and the following section will be given relative to this directory.

While our previous work [2] proves feasibility of the NN method for very large CI computations, here we stick to a medium-size example for demonstration purposes. In contrast to the cases considered in Ref. [2], the CI computations we demonstrate here can be performed directly. Note that in the following sections presenting NN-supported computations, the pCI codes are utilized in the same manner as discussed here.

We consider the Fe^{16+} highly charged ion with all electrons active and allow single and double (SD) excitations from the configurations $1s^2 2s^2 2p^6$, $1s^2 2s^2 2p^5 3p$, and $1s^2 2s^2 2p^6 3s$ to the virtual orbitals with the principal quantum number $n \leq 15$ and the orbital quantum number $l \leq 4$. Our goal is to obtain the 5 lowest even-parity energy levels of this ion. The full even-parity CI basis consists of 151 422 relativistic configurations, corresponding to 9 266 197 Slater determinants.

In the first stage, we modify the `config_Fe16+.yml` configuration file, encoding the example. We include this file in the directory `full`. We execute the `basis.py` Python script, which conveniently creates a `basis` directory and runs the respective codes to form the orbital basis set (not to be confused with the CI basis) stored in the `HFD.DAT` file. Next, we run the `ci.py` script to prepare the input for CI calculations in the generated `even0` directory. The script generates the `CONF.INP` input file, which contains the list of relativistic configurations spanning the CI space.

The direct CI computation can now be done in the `even0` directory. It is performed using the programs `pbasc` (computation of the radial integrals) and `pconf` (evaluation and partial diagonalization of the Hamiltonian matrix). After the CI computation is complete, the outputs (energies and weights of configurations) are written to the file `CONF.RES`. In the directory `full`, we include the input file `CONF.INP` with the relativistic configurations, as well as the output file `CONF.RES` containing the evaluated CI expansion weights and the level energies.

5. NN-supported CI with pCI codes

An alternate route to the described direct CI computation is our NN-supported approach, which we demonstrate here. First of all, as mentioned in Sec. 2, it is assumed that there has been a prior CI computation performed on a subbasis of the full basis. We construct this subbasis by restricting the original basis with principal quantum number $n \leq 15$ to $n \leq 8$, resulting in 28 348 relativistic configurations. The CI computation is performed directly for the $n \leq 8$ subbasis. We provide the corresponding input and output files in the `prior` directory.

Next, we discuss the Python code for NN-supported computations. We assume that the code snippets are executed in a Python shell. Alternatively, Python scripts from the GitHub repository can be utilized.

5.1. Establishing communication with pCI

Each of the two subpackages discussed in Sec. 3 and shown in Fig. 2 contains a Python class implementing the main functionalities. The "Atomic code IO" subpackage (which is now a specific implementation for pCI) provides the `PciIO` class used for establishing the communication between the "Neural manager" and the pCI package. Note that all needed imports can be performed from the top-level package `nn_for_pci` directly, for instance the `PciIO` class is imported as

```
>>> from nn_for_pci import PciIO
```

The user creates an instance of `PciIO` by providing the paths to the pCI input (`CONF.INP`) and output (`CONF.RES`) files:

- `CONF.INP` file containing the full (large) set of relativistic configurations;
- `CONF.INP` file containing the set of relativistic configurations in the "prior" computation;
- `CONF.RES` file containing the CI expansion weights resulting from the "prior" computation;
- "Current" `CONF.INP` file;
- "Current" `CONF.RES` file.

The former three files initialize the computation, while the latter two files serve as data communication with pCI in each iteration. For convenience, we provide a named tuple class `PciIOFiles` with the fields indicating the listed files. The user creates a `PciIOFiles` object with the corresponding file paths and passes it to the `PciIO` constructor:

```
>>> from nn_for_pci import PciIOFiles
>>>
>>> pci_io_files = PciIOFiles(
>>>     conf_inp_full="full/CONF.INP",
>>>     conf_inp_prior="prior/CONF.INP",
>>>     conf_res_prior="prior/CONF.RES",
>>>     conf_inp_current="CONF.INP",
>>>     conf_res_current="CONF.RES"
>>> )
>>>
>>> pci_io = PciIO(pci_io_files)
```

The created `pci_io` object encapsulates the coupling of the "Neural manager" subpackage to the pCI atomic codes, and is used in an automated manner.

5.2. Starting NN-supported computation

In order to start a NN-supported CI computation, we first import the `NeuralManager` class, which is the main class of the "Neural manager" subpackage:

```
>>> from nn_for_pci import NeuralManager
```

and instantiate it using the `pci_io` object created above:

```
>>> mng = NeuralManager(pci_io)

************************
Creating a NeuralManager
==> Loading full basis...
    Full basis size: 151422
    Features: 115
Done.
```

At this moment, the full set of relativistic orbitals is loaded into a 2D NumPy array with the height equal to the basis size and the width equal to the number of features, that is the parameters characterizing each configuration (here, the populations of the relativistic orbitals). The next step is to call the `NeuralManager` method `start_new_comp`:

```
>>> mng.start_new_comp(0.05)

Starting a new neural-network-supported computation
==> Loading prior basis...
    Prior basis size: 28348
Done.
==> Adding randoms on top...
```

```
    Fraction of randoms: 0.05
    Number of randoms: 6153
Done.
==> Writing input for the atomic code...
    Number of written: 34501
Done.
```

As input, we provided the fraction 0.05 of the non-prior configurations to be randomly included. Note that in large computations, a smaller fraction may be more appropriate. First, it loads the data from the "prior" computation. Then, it adds on top a number of configurations chosen randomly from the rest of the full basis, which are needed for the NN to explore beyond the prior set of configurations. Since the CI coefficients for the randomly added configurations are unknown, we need to evaluate them by running pCI.

For large computations, the basis set constructed here may be too big for running pCI directly. In this case, the user can choose to include the "prior" basis in the resulting `CONF.INP` file not completely, but up to a given weight cutoff y. This can be achieved by passing $\log_{10} y$ to the `start_new_comp` method as an additional keyword-only argument `cutlog`. This allows the weights for the randomly added configurations to be evaluated with a smaller pCI run. At the same time, the full data available from the "prior" run are used for the subsequent NN training.

The pCI run is performed not within the Python session, but by submitting a separate (usually strongly parallelized) job. Before exiting the current Python computation, we save it by executing the `save_comp` method:

```
>>> mng.save_comp("_saved_comp")

Saving computation
    path='_saved_comp'
Done.
```

This saves the 1D NumPy arrays which track the basis selection process. The code up to this point is contained in the Python script file `start.py`, and the printed output in the `start.log` file in our GitHub repository. The resulting `CONF.RES` file after the pCI run is contained in the `start` directory. Note that since the described procedure contains randomization, the user will obtain different `CONF.INP` and `CONF.RES` files.

5.3. First iteration with NN

After the pCI code execution finishes and the outputs are written to the corresponding files, we switch to the first iteration with the NN.

5.3.1. Preparation

The way to create the `PciIO` and `NeuralManager` objects demonstrated so far leads to loading the full set of the relativistic configurations as a 2D NumPy array. However, the columns of this dataset are of different scale, since the corresponding orbitals have different maximal populations. Our package implements two ways to tackle this issue, which is unfavorable for the NN training. The `PciIO` class can be instantiated with an additional argument `normalize=True` or `digitize=True` (which are both by default `False` and cannot be `True` at the same time). It triggers a corresponding transformation of the dataset with the relativistic configurations immediately after loading. In case `normalize=True` is provided, each column of the dataset is normalized with respect to the maximal value in this column. The argument `digitize=True` leads to the transformation of integer orbital populations in the relativistic configurations to binary format, with each digit treated as a separate feature. Note that the "trivial" features, such as those having a constant value ("0" or "1") across the whole dataset, are deleted.

In this example we choose the option `digitize=True`:

```
>>> pci_io = PciIO(pci_io_files, digitize=True)
```

Note that this transformation typically increases the number of features (i.e. the number of the 2D NumPy array columns) as seen from the output when instantiating the `NeuralManager` class:

```
>>> mng = NeuralManager(pci_io)


************************
Creating a NeuralManager
==> Loading full basis...
    Full basis size: 151422
    Features: 231
Done.
```

Here, the digitization and subsequent deletion of the trivial features increased the width of the data from 115 to 231. For more explanation we refer to Ref. [2] and, in particular, Table I therein showing a demonstration example.

Now we load the previous Python computation as

```
>>> mng.load_comp("_saved_comp")

Loading computation
    path='_saved_comp'
Done.
```

5.3.2. NN setting

Our implementation of the NN-supported algorithm assumes that the user encodes the NN setting as a TensorFlow model, which is then automatically used within our package. In this way, there remains flexibility in the NN structure, which only has to satisfy the NN input and output format. Also other elements of the NN setting (optimizer, loss function, hyperparameter values etc.) are provided by the user together with the TensorFlow model. We construct here an NN model based on Ref. [2] which can be used as a starting point for further NN setting optimization. One NN input entry corresponds to one relativistic configuration and consists of its orbital populations in the digitized format. The NN output consists always of 2 neurons which encode the probabilities of the given relativistic configuration to be important or unimportant, that is, their values must lie in the range $(0, 1)$ and sum to 1. In the following, we provide the code for the NN setting from Ref. [2]. We refer to the classical book [21] for a general introduction to NNs and to the book [22] for a practical NN introduction using TensorFlow.

We employ in this work the Keras package [23], which enables usage of the TensorFlow NN functionalities in an easy and high-level way. First, we import the necessary Keras classes:

```
>>> from tensorflow.keras.models \
>>>     import Sequential
>>> from tensorflow.keras.layers \
>>>     import InputLayer, Dense
>>> from tensorflow.keras.callbacks \
>>>     import EarlyStopping
```

The NN architecture is encoded as

```
>>> inpdim = mng.features_num
>>>
>>> model = Sequential()
>>> model.add(InputLayer((inpdim,)))
>>> model.add(Dense(inpdim, activation='relu'))
>>> model.add(Dense(inpdim, activation='relu'))
>>> model.add(Dense(inpdim//2, activation='relu'))
>>> model.add(Dense(inpdim//4, activation='relu'))
>>> model.add(Dense(2, activation='softmax'))
```

Here, we first obtained the size of one NN input entry via the `Neural-Manager` property `features_num` and used it to determine the sizes of the hidden NN layers. In all hidden layers, rectified linear unit (reLU) is used as the activation function. In order to guarantee the interpretability of the NN output as a probability distribution, we applied the softmax activation function in the output NN layer.

We "compile" the created Tensorflow model as follows

```
>>> model.compile(
>>>     optimizer='adam',
>>>     loss='categorical_crossentropy',
>>>     metrics=['accuracy']
>>> )
```

We chose here the standard "adam" optimization algorithm [24] for the NN training. The discrepancy of the NN output and the "correct" answer is measured by categorical cross-entropy, which is a standard choice for softmax-classifiers. The monitored metric is the accuracy, that is the ratio of the entries classified by the NN correctly. Note that the accuracy is monitored on a "validation set" held out from the training data in advance and never exposed to the NN. It is used to stop the NN training once no progress in the classification quality is achieved (so called "early stopping").

Apart from the TensorFlow model, our package requires three Python dictionaries containing parameters for (1) the initial NN evaluation, (2) the NN training, and (3) the prediction using the NN. The initial NN evaluation is performed on the full training set and is controlled by the dictionary

```
>>> init_params = {
>>>     'batch_size': 32 * 32 * 32,
>>>     'verbose': 2
>>> }
```

The provided `batch_size` entry determines the size of the batches when the NN is applied to the training set. Note that the Keras default is 32, which can unnecessarily slow down the performance if this parameter is not set explicitly to a larger value. For the NN training, we use the following dictionary:

```
>>> es = EarlyStopping(
>>>     monitor='val_accuracy',
>>>     restore_best_weights=True,
>>>     patience=5
>>> )
>>>
>>> train_params = {
>>>     'epochs': 200,
>>>     'validation_split': 0.2,
>>>     'verbose': 2,
>>>     'callbacks': [es]
>>> }
```

Here, we first created an instance of the Keras `EarlyStopping` class for managing early stopping based on the monitored accuracy. The entry `patience=5` tells the NN training not to stop immediately after a lower NN performance is observed, but to attempt another 5 training epochs. The model with the best achieved performance is then restored as controlled by `restore_best_weights=True`. The `epochs` parameter determines the maximum number of training epochs, whereas `validation_split` encodes the size of the validation set relative to the full training set.

The NN prediction is controlled by the following dictionary:

```
>>> apply_params = {
>>>     'batch_size' : 32 * 32 * 32,
```

```
>>>     'verbose': 0
>>> }
```

Note that since the NN is applied to sort out a potentially large number of relativistic configurations, it is especially important here to specify `batch_size`. Otherwise, the Keras default value of 32 is used, potentially resulting in a strong slowdown of this step.

5.3.3. Basis sortout

As the NN-setting is prepared, we can switch to sorting out the large CI basis using the NN. In each NN-supported iteration, this requires the user to set the following two parameters. Firstly, as described in Sec. 2, the "importance" of relativistic configurations is determined by the cut-off x_i. In this first iteration, we choose $\log_{10} x_1 = -8.5$. Secondly, the NN needs feedback not only on the basis elements it classified as important, but also on those classified as unimportant. Therefore, we include in the current CI basis $\mathcal{B}_i$ a number of randomly chosen configurations discarded by the NN. We refer to this step as "balancing" and choose the "balancing fraction" of 0.5 following Ref. [2]. In this case, if the NN suggests K important configurations, we add on top $K/2$ configurations considered by the NN as unimportant.

We now have all necessary ingredients to call the `NeuralManager` method `neural_sortout`, which is the central component of our Python package, encapsulating the NN part of one NN-supported iteration (see Fig. 1):

```
>>> cutlog = -8.5
>>> bal_frac = 0.5
>>>
>>> mng.neural_sortout(cutlog, bal_frac, model,
>>>     init_params, train_params, apply_params)

Sorting out the basis using a neural network
cutlog: -8.5
==> Reading and preparing weights...
    Prior weights were present;
    they are taken into account and deleted.
Done.
==> Training the neural network...
    Training set size: 34501
    ...of which important: 9362
    <NN EVALUATION OMITTED>
Done.
==> Applying the neural network...
    Application to: 116921
    Classified as important: 18615
Done.
==> Balancing the next training set...
    Balancing fraction: 0.5
    Balance set size: 9307
Done.
==> Writing input for the atomic code...
    Number of written: 37284
Done.
```

For brevity, we omit here the printed output from the evaluation of the NN classification accuracy before the training and after each training epoch. The full output for this and the subsequent NN-supported iterations is provided in the `*.log` files, together with the corresponding Python scripts.

This completes the NN part of the current iteration. The computation (i.e., the internal 1D NumPy arrays tracking the selection process) can be now saved along with the NN model:

```
>>> model.save("_saved_model1.keras")
>>> mng.save_comp("_saved_comp1")
```

Table 2
The number of relativistic configurations and the computational times for the pCI stage in the NN-supported iterations, the full computation, and the "prior" computation are presented. The entry "start" represents the stage described in Sec. 5.2.

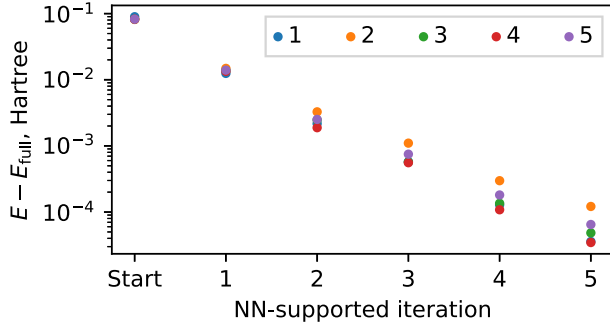|	Full	Prior	Start	Iter. 1	Iter. 2	Iter. 3	Iter. 4	Iter. 5
Rel. configs	151 422	28 348	34 501	37 284	35 859	49 617	60 629	60 070
Time, min	137	3	5	12	13	24	32	36

Fig. 3. Convergence of the energy levels to the corresponding energies on the full CI set with NN-supported iterations. "Start" labels the starting stage described in Sec. 5.2.

```
Saving computation
    path='_saved_comp1'
Done.
```

Here we used the `save` method available for Keras models. Now the Python session can be closed, and a CI computation can be performed using the resulting NN-selected set of configurations.

5.4. Further iterations

After the CI results are obtained, the computation described in Sec. 5.3 is performed again on a smaller importance cutoff. In each further iteration, the NN is not created from scratch, but loaded using the Keras function `tensorflow.keras.models.load`. Note also that in the very last iteration, balancing of the training set discussed above should be omitted by setting `bal_frac=0`, since no NN training follows. This is especially important for large computations, since the last iterations are the most numerically demanding.

For convenience, we provide the Python scripts `iter*.py` for five NN-supported iterations performed in this work. The outputs printed by the Python scripts are contained in the `iter*.log` files. The `CONF.RES` output files are provided in the `iter*` directories. Since the NN-supported iterations contain randomization, the user will obtain their own versions of these outputs.

5.5. Discussion of results

After performing the five NN-supported iterations for the considered example, we show in Fig. 3 how the energy levels converge to the corresponding energies on the full CI set. Here, "Start" labels the starting stage described in Sec. 5.2. Note that for larger computations, the full CI energy might not be possible to directly evaluate. We refer to Ref. [2] for a discussion of an extrapolation procedure for the energies.

In Table 2, we present the CI basis sizes at different NN-supported iterations, as well as for the full CI computation and the "prior" computation. The listed times refer only to how long it takes to run the `pconf` program, which accounts for the majority of the whole CI computation. The program `pbasc` was run only once for the full CI and took negligible time. While the pCI codes were run on 10 computational nodes,

each with 32 CPU cores, the Python scripts with NN were executed interactively on a single CPU core and took negligible time, due to crucial algorithmic improvements introduced in Ref. [2].

In this demonstration, the total time of 122 min taken by all iterations of the NN algorithm almost reaches the time of 137 min needed for direct CI. However, for larger cases the NN algorithm offers a clear advantage. In Ref. [2], we considered examples where direct CI computations lied far beyond the computational possibilities. There, we demonstrated that the NN-based computations were much more efficient with respect to an alternative "basis upscaling" method. The aggregate execution time for the direct CI computations was about 284 hours, compared to 71 hours of execution time using the NN-supported CI approach [2]. We emphasize that in addition to the approximate 5x reduced computational time, less "human" time was required to set up various CI computations. This concludes the demonstration of our NN method with the atomic pCI package, and we switch to a discussion of possible extensions to other atomic codes.

6. Coupling to other atomic codes

While the previous discussion specifically involves the pCI package, the demonstrated NN support can also be coupled with other atomic codes. The block "Neutral manager" (see Fig. 2) is universal and can be used without any changes with a different atomic code, while the coupling part "Atomic code IO" must be implemented from scratch following a particular interface. This interface is encoded and provided with our Python package as an abstract class `AtomicCodeIO`.

A custom coupling is created by the user as a class inheriting from `AtomicCodeIO`:

```
>>> from nn_for_pci import AtomicCodeIO
>>>
>>> class MyAtomicCodeIO(AtomicCodeIO):
>>>
>>>     def read_full_basis(self):
>>>         ... # your code here
>>>
>>>     def read_prior_basis(self):
>>>         ... # your code here
>>>
>>>     def read_prior_weights(self):
>>>         ... # your code here
>>>
>>>     def read_current_weights(self):
>>>         ... # your code here
>>>
>>>     def write_current_basis(self, which_write):
>>>         ... # your code here
```

The inheritance enforces implementation of the above five methods, which are supposed to perform writing the input for the atomic code and reading its output. We stress that running the atomic code for CI computations does not belong to the responsibilities of the presented Python codes and is performed by the user separately. Once implemented in this way, the `MyAtomicCodeIO` class becomes compatible with `NeuralManager`.

The methods to be implemented must adhere to particular input and output formats summarized by the following.

- `read_full_basis` takes no input and returns a 2D NumPy array for the full CI basis set with rows corresponding to relativistic configurations and columns representing relativistic orbital populations.
- `read_prior_basis` is the same as `read_full_basis`, but for the "prior" CI basis set.
- `read_prior_weights` takes no input and returns a 1D NumPy array with weights obtained in the "prior" computation.
- `read_current_weights` takes no input and returns a 1D NumPy array with weights obtained in the "current" CI run.
- `write_current_basis` writes input for the next CI run. The argument `which_write` is a 1D boolean NumPy array with `True` corresponding to the relativistic configurations to be written.

In this way, the `NeuralManager` class encapsulating the NN method operates with NumPy arrays communicated with the coupling class.

The user is free to introduce further functionalities to the coupling class beyond the five obligatory methods. For instance, in our computations with the pCI codes, the NN received as input not populations of the relativistic orbitals directly, but their digitized versions. This feature was implemented within the `PciIO` coupling class and was controlled by its constructor argument `digitize`. This behavior can be reintroduced also for other atomic codes. Note, however, that at the first stage of the computation described in Sec. 5.2, `NeuralManager` identifies positions of the "prior" basis in the full basis as represented by the 2D NumPy arrays returned by the functions `read_start_basis` and `read_full_basis`, respectively. Due to this implementation detail, it is important to ensure that at that stage, both NumPy arrays contain relativistic orbitals populations in the same format. In particular, in Sec. 5.2, both arrays are in the original (not digitized) format.

7. Summary

We presented our Python package for performing NN-supported atomic CI computations. While our previous work [2] demonstrated this approach for large-scale CI, here we employed a medium-size example for presenting the code and the computational workflow. Our Python codes implementing the NN are used not as a standalone tool, but in coupling with an atomic code performing CI computations. We share our package coupled out-of-the-box to the recently published parallelized pCI atomic codes [7]. We also demonstrate how a coupling to a different atomic code can be established by following an interface included in our package. This makes the presented Python package a universal tool for NN-supported large-scale atomic CI.

CRediT authorship contribution statement

Pavlo Bilous: Writing – original draft, Validation, Resources, Methodology, Formal analysis, Conceptualization, Writing – review & editing, Visualization, Software, Project administration, Investigation, Data curation. **Charles Cheung:** Writing – review & editing, Visualization, Software, Project administration, Investigation, Formal analysis, Conceptualization, Writing – original draft, Validation, Resources, Methodology, Funding acquisition, Data curation. **Marianna S. Safronova:** Writing – original draft, Validation, Software, Project administration, Investigation, Formal analysis, Writing – review & editing, Visualization, Supervision, Resources, Methodology, Funding acquisition, Data curation, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

[1] I.P. Grant, Relativistic Quantum Theory of Atoms and Molecules: Theory and Computation, Springer New York, New York, NY, 2007.

[2] P. Bilous, C. Cheung, M. Safronova, Neural-network approach to running high-precision atomic computations, Phys. Rev. A 110 (2024) 042818, https://doi.org/10.1103/PhysRevA.110.042818.

[3] P. Bilous, A. Pálffy, F. Marquardt, Deep-learning approach for the atomic configuration interaction problem on large basis sets, Phys. Rev. Lett. 131 (2023) 133002, https://doi.org/10.1103/PhysRevLett.131.133002.

[4] J.P. Coe, Machine learning configuration interaction, J. Chem. Theory Comput. 14 (11) (2018) 5739–5749, https://doi.org/10.1021/acs.jctc.8b00849, pMID: 30285426 arXiv:https://doi.org/10.1021/acs.jctc.8b00849.

[5] W. Jeong, C.A. Gaggioli, L. Gagliardi, Active learning configuration interaction for excited-state calculations of polycyclic aromatic hydrocarbons, J. Chem. Theory Comput. 17 (12) (2021) 7518–7530, https://doi.org/10.1021/acs.jctc.1c00769.

[6] S.D. Pineda Flores, Chembot: a machine learning approach to selective configuration interaction, J. Chem. Theory Comput. 17 (7) (2021) 4028–4038, https://doi.org/10.1021/acs.jctc.1c00196, pMID: 34125549.

[7] C. Cheung, M.G. Kozlov, S.G. Porsev, M.S. Safronova, I.I. Tupitsyn, A.I. Bondarev, pci: a parallel configuration interaction software package for high-precision atomic structure calculations, Comput. Phys. Commun. 308 (2025) 109463, https://doi.org/10.1016/j.cpc.2024.109463.

[8] C. Cheung, M. Safronova, S. Porsev, Scalable codes for precision calculations of properties of complex atomic systems, Symmetry 13 (4) (2021), https://doi.org/10.3390/sym13040621.

[9] C. Shah, M. Togawa, M. Botz, J. Danisch, J.J. Goes, S. Bernitt, M. Maxton, K. Köbnick, J. Buck, J. Seltmann, M. Hoesch, M.F. Gu, F.S. Porter, T. Pfeifer, M.A. Leutenegger, C. Cheung, M.S. Safronova, J.R. Crespo López-Urrutia, High-precision transition energy measurements of neon-like fe xvii ions, Astrophys. J. 969 (1) (2024) 52, https://doi.org/10.3847/1538-4357/ad454b.

[10] C. Shah, S. Kühn, S. Bernitt, R. Steinbrügge, M. Togawa, L. Berger, J. Buck, M. Hoesch, J. Seltmann, M.G. Kozlov, S.G. Porsev, M.F. Gu, F.S. Porter, T. Pfeifer, M.A. Leutenegger, C. Cheung, M.S. Safronova, J.R. Crespo López-Urrutia, Natural-linewidth measurements of the 3c and 3d soft-x-ray transitions in ni xix, Phys. Rev. A 109 (2024) 063108, https://doi.org/10.1103/PhysRevA.109.063108.

[11] S. Eustice, D. Filin, J. Schrott, S. Porsev, C. Cheung, D. Novoa, D.M. Stamper-Kurn, M.S. Safronova, Optical telecommunications-band clock based on neutral titanium atoms, Phys. Rev. A 107 (5) (May 2023), https://doi.org/10.1103/physreva.107.l051102.

[12] C.C.M.S.S. David, R. Leibrandt, Sergey G. Porsev, Prospects of a thousand-ion sn²⁺ Coulomb-crystal clock with sub-10⁻¹⁹ inaccuracy, Nat. Commun. 15 (2024) 5663.

[13] S. Kühn, C. Cheung, N.S. Oreshkina, R. Steinbrügge, M. Togawa, S. Bernitt, L. Berger, J. Buck, M. Hoesch, J. Seltmann, F. Trinter, C.H. Keitel, M.G. Kozlov, S.G. Porsev, M.F. Gu, F.S. Porter, T. Pfeifer, M.A. Leutenegger, Z. Harman, M.S. Safronova, J.R.C. López-Urrutia, C. Shah, New measurement resolves key astrophysical fe xvii oscillator strength problem, Biophys. Rev. Lett. 129 (24) (Dec. 2022), https://doi.org/10.1103/physrevlett.129.245001.

[14] E.B. Norrgard, D.S. Barker, S.P. Eckel, S.G. Porsev, C. Cheung, M.G. Kozlov, I.I. Tupitsyn, M.S. Safronova, Laser spectroscopy of the y $^7p_J^o$ states of cr i, Phys. Rev. A 105 (2022) 032812, https://doi.org/10.1103/PhysRevA.105.032812.

[15] C. Walter, S. Spielman, R. Ponce, N. Gibson, J. Yukich, C. Cheung, M. Safronova, Observation of an electric quadrupole transition in a negative ion: experiment and theory, Biophys. Rev. Lett. 126 (8) (Feb. 2021), https://doi.org/10.1103/physrevlett.126.083001.

[16] C.R. Harris, K.J. Millman, S.J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N.J. Smith, R. Kern, M. Picus, S. Hoyer, M.H. van Kerkwijk, M. Brett, A. Haldane, J.F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, T.E. Oliphant, Array programming with NumPy, Nature 585 (7825) (2020) 357–362, https://doi.org/10.1038/s41586-020-2649-2.

[17] T. pandas development team, pandas-dev/pandas: pandas, https://doi.org/10.5281/zenodo.3509134, Feb. 2020.

[18] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G.S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, TensorFlow: large-scale machine learning on heterogeneous systems, software available from tensorflow.org, https://www.tensorflow.org/, 2015.

[19] M.G. Kozlov, S.G. Porsev, V.V. Flambaum, Manifestation of the nuclear anapole moment in the m1 transitions in bismuth, J. Phys. B 29 (4) (1996) 689–697.

[20] M.G. Kozlov, S.G. Porsev, M.S. Safronova, I.I. Tupitsyn, Ci-mbpt: a package of programs for relativistic atomic calculations based on a method combining configuration interaction and many-body perturbation theory, Comput. Phys. Commun. 195 (2015) 199–213.

[21] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016, http://www.deeplearningbook.org.

[22] A. Geron, Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems, 2nd edition, O'Reilly Media, Inc., 2019.

[23] F. Chollet, et al., Keras, https://keras.io, 2015.

[24] D.P. Kingma, J. Ba, Adam: a method for stochastic optimization, arXiv:1412.6980, 2017.