SOLAX: A Python solver for fermionic quantum systems with neural network support

Louis Thirion[1], Philipp Hansmann[1] and Pavlo Bilous[2⋆]

1 Department of Physics, Friedrich-Alexander-Universität Erlangen-Nürnberg, 91058 Erlangen, Germany
2 Max Planck Institute for the Science of Light, Staudtstraße 2, 91058 Erlangen, Germany

⋆ pavlo.bilous@mpl.mpg.de

Abstract

Numerical modeling of fermionic many-body quantum systems presents similar challenges across various research domains, necessitating universal tools, including state-of-the-art machine learning techniques. Here, we introduce SOLAX, a Python library designed to compute and analyze fermionic quantum systems using the formalism of second quantization. SOLAX provides a modular framework for constructing and manipulating basis sets, quantum states, and operators, facilitating the simulation of electronic structures and determining many-body quantum states in finite-size Hilbert spaces. The library integrates machine learning capabilities to mitigate the exponential growth of Hilbert space dimensions in large quantum clusters. The core low-level functionalities are implemented using the recently developed Python library JAX. Demonstrated through its application to the Single Impurity Anderson Model, SOLAX offers a flexible and powerful tool for researchers addressing the challenges of many-body quantum systems across a broad spectrum of fields, including atomic physics, quantum chemistry, and condensed matter physics.

Received 2024-09-02
Accepted 2025-02-03
Published 2025-02-20
doi:10.21468/SciPostPhysCodeb.51

Check for updates

This publication is part of a bundle: Please cite both the article and the release you used.

DOI	Type
doi:10.21468/SciPostPhysCodeb.51	Article
doi:10.21468/SciPostPhysCodeb.51-r1.0	Codebase release

Contents

1 Introduction

Accurate numerical modeling of fermionic quantum many-body systems presents an essential challenge across many research domains. In atomic physics, for example, precise knowledge of electronic energy levels is indispensable for the development of atomic frequency standards, the understanding of astrophysical spectra, and the search for phenomena beyond the Standard Model [1]. In particular, the promising yet scarcely explored domain of highly charged ions lacks experimental data and requires extensive computational support [2]. In quantum chemistry, the pursuit of highly accurate electronic structure calculations, such as those achieved through full configuration interaction (full CI) methods [3–7], is crucial for the accurate determination of molecular properties and the prediction of chemical reactivity. In condensed matter physics, quantitative research on low-energy effective Hamiltonians such as the Hubbard model [8] and its derivatives, supports the qualitative understanding of microscopic mechanisms underlying phenomena like unconventional superconductivity in cuprates [9, 10], iron pnictides [11], and nickelates [12, 13]. While the core motivations and goals in all these diverse research areas are often completely different, the computational challenges are similar and the most challenging tasks are often identical from the technical point of view. Alongside methodological developments, advanced simulation codes for quantum many-body systems are therefore essential for scientific progress across a broad spectrum of fields.

Despite these advances, many challenges still demand computational efforts that exceed the capabilities of even the most efficient codes and/or the available computational resources. In such cases, machine learning techniques can be applied to reduce the complexity of the calculation without compromising the accuracy. Possible approaches based on a neural network (NN) were demonstrated in Refs. [14, 15] for problems in computational atomic physics, and in Ref. [16] in a more general context of fermionic systems requiring large expansions of the wave function in the basis of Slater determinants. In the latter work, the machine learning functionality based on the TensorFlow library [17] was interfaced with the Quanty CI code [18]. Following the successful "proof of principle" in Ref. [16], we saw the need to implement an integrated Python library rather than interfacing existing codes. Here we present the resulting SOLAX package.

SOLAX is a comprehensive NN-boosted Python library designed for the study of fermionic quantum many-body systems. Within the standard quantum many-body formalism of second quantization, SOLAX provides a framework for constructing and solving quantum cluster problems. The SOLAX package offers a versatile set of tools to efficiently encode and manipulate basis sets, quantum states, and operators, enabling users to simulate and explore the structure of quantum many-body systems. The library supports the accurate determination of many-body quantum states in finite-size Hilbert spaces. Beyond its core functionalities, it includes built-in machine learning tools. Specifically, SOLAX addresses the exponential growth of Hilbert space dimensions in large quantum clusters: When full diagonalization becomes computationally infeasible, a NN classifier can be employed to approximate the solution through efficient ba-

sis optimization. The SOLAX library has already been successfully applied to the study of molecules such as N_2 [19]. The NN algorithm presented in this article was first demonstrated in Ref. [16] in its application to the Single Impurity Anderson Model (SIAM).

Typically, selection methods for basis states were developed in Fortran or similar languages, see e.g. Ref. [20]. This makes it difficult to include NNs in the code, since the standard libraries for NNs are primarily provided within the Python ecosystem. We implemented SOLAX directly in Python with the core low-level functionalities based on the JAX library recently developed at Google [21]. JAX offers highly efficient GPU-accelerated mechanisms to manipulate data organized as arrays, similar to those from the well-known NumPy library [22]. This allows for a seamless transition of data to and from the NumPy format, which is the main method for storing data in SOLAX. Importantly, JAX was initially designed for high-performance machine learning research and offers powerful capabilities to leverage NNs. We would like to highlight the recently developed JAX-based NetKet package [23, 24], built around the concept of neural quantum states, i.e., NNs that approximately encode states of quantum many-body systems [25]. Here, we follow a different approach from Ref. [16], using a NN classifier to perform selection of important basis states. To the best of our knowledge, SOLAX is the first integrated JAX-based implementation of such a NN-supported approach.

We assume the reader of this article to be familiar with the Python programming language and the libraries NumPy [22] and SciPy [26] which are both extensively used in scientific programming. The necessary information on the tools from the JAX ecosystem will be provided as they are used. We do not assume that the reader has experience with machine learning using NNs and provide an introduction to the basic NN concepts relevant for this work. For more information on NNs we refer to the classical work [27]. Machine learning from the general (probabilistic) perspective is discussed in depth in the comprehensive source [28, 29]. For a practical introduction to machine learning (including neural networks using TensorFlow), we recommend the book [30].

The article is structured as follows. In Section 2, we showcase the core functionality of SOLAX for solving the fermionic many-body problem and provide an exemplary computation for SIAM. Section 3 presents the built-in SOLAX tools for NN-assisted computations and their application to SIAM along with computational time benchmarks. In Section 4, we describe the mechanisms for saving and loading SOLAX objects, as well as reproducing SOLAX computations. The article closes with the conclusions and outlook in Section 5.

1.1 Code availability and dependencies

The SOLAX code can be cloned directly from the GitHub repository [31], where we also provide Jupyter notebooks with the code snippets shown in this article. Apart from packages from the standard Python library available without separate installation, SOLAX employs the following third-party libraries: NumPy [22], Pandas [32], SciPy [26], JAX [21], FLAX [33], and Orbax [34] (the latter 3 libraries belong to the JAX ecosystem). The versions of Python and the listed packages used in SOLAX at the moment of the present publication are summarized in Table 1. The user is required to perform the necessary installations before using SOLAX. For installing JAX, we suggest to follow the instructions on the webpage [35], where different installation aspects are addressed. For leveraging GPU acceleration, a GPU-capable version of JAX must be installed. The current version of SOLAX includes a possibility to perform some computations in parallel on multiple GPUs. Note, however, that this functionality is still under development, and is considered here as an experimental feature. We also note that compatibility of the presented SOLAX version with newer versions of the listed packages cannot be guaranteed, especially for the libraries from the JAX ecosystem which are still under extensive development. In further SOLAX versions we plan to take into account the development progress for the dependencies.

Table 1: The versions of Python and the third-party libraries used in SOLAX at the moment of the present publication.

Python	NumPy	Pandas	SciPy	JAX	FLAX	Orbax
3.10.9	1.26.1	1.5.3	1.10.0	0.4.30	0.8.5	0.1.9

2 Solver for fermionic quantum systems

The core functionality of the SOLAX package consists of encoding and solving eigenvalue equations for fermionic quantum many-body systems. Fully antisymmetric Slater determinants serve as the basis for many-body Hilbert spaces of fermionic wave functions. The occupation number representation of a Slater determinant is a binary string of zeroes "0" and ones "1" (adhering to the Pauli exclusion principle). As the full set of Slater determinants forms a complete basis on a given Hilbert space, any many-body quantum state in this space can be expanded using this basis. SOLAX follows this paradigm in the representation of quantum states. Operators within the SOLAX package are expressed in terms of creation ($\hat{a}_i^\dagger$) and annihilation ($\hat{a}_i$) operators which act on basis Slater determinants or quantum states represented as linear combinations of Slater determinants. These ladder operators are indexed by single-particle quantum numbers i which indicate the positions in the occupation-number string on which they act. In this formalism, a standard Hamilton operator can be decomposed into single-particle, two-particle, and more generally, n-particle operators. Each of these terms in the Hamiltonian is expressed as a sum of ladder-operator products containing the corresponding number of creation and annihilation operators.

In the following, we describe how Slater determinant bases, many-body quantum states, operators, and their matrix representation can be defined and manipulated in SOLAX. We first introduce the main components implemented in SOLAX as Python classes: `Basis`, `State`, `OperatorTerm`, `Operator`, and `OperatorMatrix`. Subsequently, we demonstrate these fundamental tools with the help of an example by finding the ground state of the paradigmatic Single Impurity Anderson Model (SIAM). The data storage and processing for the presented classes are primarily based on NumPy arrays [22], which also facilitate convenient interaction with the user. Internally, SOLAX enhances operations on 2D NumPy arrays using the Pandas library [32]. This powerful data analysis tool allows us to leverage hash maps and significantly speed up operations in which search in 2D NumPy arrays is needed. At the same time, the central quantum solver operations like e.g. acting with operators on quantum states are implemented in JAX [21] and support automatic GPU acceleration.

2.1 Basis

We begin by introducing the `Basis` class in SOLAX, which facilitates efficient manipulation of basis sets composed of Slater determinants. To utilize SOLAX and NumPy in a Python environment, the libraries are imported as follows:

```
import solax as sx
import numpy as np
```

2.1.1 Object construction

A `Basis` instance is created from a collection of strings, each representing a Slater determinant in the occupation number representation (note that when initializing a `Basis` with a single Slater Determinant, the latter still must be included in a Python collection, e.g. a `list`).

All determinants (strings) must have the same length corresponding to the number of the underlying single-particle degrees of freedom which we refer to from here on as "spin-orbitals". For example, a `Basis` object `basis` with all possible Slater determinants for two electrons occupying four spin-orbitals is constructed as follows:

```
basis = sx.Basis(["1100", "1010", "1001", "0110", "0101", "0011"])
```

SOLAX treats the received strings as binary representations of 8-bit integer numbers (in general a few for each determinant) and stores these efficiently in a 2D NumPy array. The length of each Slater determinant (i.e. the total number of 0s and 1s) in a `Basis` object can be accessed using the read-only `bitlen` property:

```
print(basis.bitlen)
```

```
4
```

As will be seen in the following, `Basis` objects behave similarly to Python sets in many contexts. In particular, if repeated determinants are passed to the class constructor, these are automatically discarded.

2.1.2 Conversion to a Python string and printing

The user can convert a `Basis` to a Python string and print it in order to see the stored determinants in the usual format:

```
print(basis)
```

```
1100
1010
1001
0110
0101
...
```

Note that only the first five (the default value of the print limit) determinants are reflected in the printed string. The value of this limit can be changed using the context manager `dets_printing_limit`:

```
with sx.dets_printing_limit(10):
    print(basis)
```

```
1100
1010
1001
0110
0101
0011
```

The user provides here the maximal number of printed determinants (in this case 10) or `None` if no limit is to be used.

2.1.3 Length, indexing, and slicing

A. Basis as a Python sequence. The `Basis` class implements the Python sequence protocol, i.e. its objects have length and can be indexed and sliced in the standard way. The length is the number of the contained Slater determinants, and can be obtained with the Python built-in `len` function:

```
print(len(basis))
```

```
6
```

Indexing and slicing picks the Slater determinants at the corresponding positions and returns a new `Basis` object:

```
print(basis[0])
```

```
1100
```

```
print(basis[0:3])
```

```
1100
1010
1001
```

Note that in contrast to the typical behavior of sequences, `basis[0]` is again of type `Basis`. However, this is natural for this particular class and still complies with the Python sequence interface.

B. "Fancy" and boolean indexing. On top of the standard Python sequence functionality, the `Basis` class supports the NumPy-style "fancy" and boolean indexing [36]. That is, `Basis` objects can be indexed in two additional ways: using a list on indices and with a boolean mask, respectively:

```
print(basis[0, 1, 4])
```

```
1100
1010
0101
```

```
print(basis[True, True, False, False, True, False])
```

```
1100
1010
0101
```

As for NumPy arrays, the boolean mask here must have the same length as the sequence itself.

Important note! There is a crucial difference between the indexing mechanisms shown in (A) and (B), which is inherited from NumPy. In NumPy, standard Python indexing and slicing do not create a new array but instead reference a sub-array of the original array. In contrast, fancy or boolean indexing does create a new array in memory. This behavior extends to the `Basis` class, as it is based on NumPy (like most classes in SOLAX). While a new `Basis` object is created regardless, this does not necessarily hold for the underlying NumPy arrays. Therefore, for efficient memory usage, the indexing method described in (A) should be preferred.

2.1.4 Set operations

From the operational perspective, `Basis` objects behave similarly to Python sets. However, they do not fully implement the standard Python set interface. The reason for this is twofold: (1) not all set operations are necessary for SOLAX applications, and (2) the standard Python set nomenclature can be somewhat confusing in this context. Below, we describe the operations with `Basis` objects as implemented in SOLAX.

A. Equality relation. As for Python sets, the equality relation == disregards the order of the Slater determinants in the compared `Basis` instances. For example, we compare the created `basis` object with its reverse:

```
print(basis == basis[::-1])
```

```
True
```

B. Addition (set union). Objects of the `Basis` class can be added (unified like sets) using the + operator. Note that the analogous operator for Python sets is |. As an example, consider two `Basis` objects obtained from `basis` by selecting only Slater determinants at even and odd positions, respectively:

```
basis_even = basis[::2]
basis_odd = basis[1::2]
```

As expected, their addition equals the initial `basis`:

```
print(basis_even + basis_odd == basis)
```

```
True
```

Repeated determinants are automatically excluded from the resulting `Basis`. For example, unification of `basis` with itself gives again `basis` since no new determinants are added:

```
print(basis + basis == basis)
```

```
True
```

C. Set difference. `Basis` objects can be subtracted like sets. This operation is bound to the operator `%` instead of `-` as for Python sets. For instance, for the objects `basis`, `basis_even` and `basis_odd` introduced above, we have:

```
print(basis % basis_even == basis_odd)
```

```
True
```

Note that both for the `+` and the `%` operator, the Slater determinants in both operands must have the same length in the sense of the total number of 0s and 1s (which can be accessed using the `bitlen` property).

2.2 State

We proceed with the `State` class in SOLAX. While the `Basis` class incorporates sets of Slater determinants, the `State` class also includes real or complex coefficients assigned to each determinant. As such, a `State` object represents a quantum state expanded in the basis of Slater determinants. SOLAX supports linear algebra operations on `State` objects including their scalar product, thereby implementing the structure of a Hilbert space.

2.2.1 Fundamentals

Here we demonstrate the basic usage of the `State` class. As will be seen, many aspects are similar to those of the `Basis` class. A `State` instance is created from a `Basis` instance and a NumPy array of associated coefficients:

```
state = sx.State(basis, np.ones(6))
```

Here we used the `basis` object created in the previous section and a NumPy array with real numbers all equal 1. Note that states as represented by the SOLAX `State` class can be unnormalized, e.g. as the just created `state` object. Objects of the `State` class can be converted to a Python string and printed:

```
print(state)
```

```
|1100>  *  1.0
|1010>  *  1.0
|1001>  *  1.0
|0110>  *  1.0
|0101>  *  1.0
...
```

As for the `Basis` class, the number of shown determinants can be changed using the `dets_printing_limit` context manager. The underlying `Basis` object and the array of coefficients can be accessed directly as the attributes `basis` and `coeffs`:

```
print(state.basis)
```

```
1100
1010
1001
0110
0101
...
```

```
print(state.coeffs)
```

```
[1. 1. 1. 1. 1. 1.]
```

Analogously to the `Basis` class demonstrated in the previous section, the `State` class implements the Python sequence protocol and supports NumPy-style "fancy" and boolean indexing [36]. From the quantum mechanical perspective, this functionality corresponds to projecting the state onto the Hilbert subspace spanned by the selected determinants.

Additionally, SOLAX supports operations of the form `State % Basis`, which remove from the `State` object all determinants present in the `Basis` together with their associated coefficients. Quantum mechanically, this operation corresponds to projecting the state along the Hilbert subspace spanned by the deleted determinants. We demonstrate this operation using the `basis_even` and `basis_odd` objects from the previous section. The following code line retains in the resulting `State` only the determinants present in `basis_odd`:

```
state_odd = state % basis_even
print(state_odd)
```

```
|1010>  *  1.0
|0110>  *  1.0
|0011>  *  1.0
```

```
print(state_odd.basis == basis_odd)
```

```
True
```

The equality operator `==` is not directly implemented for the `State` class because the latter involves real or complex coefficients that are represented up to machine precision. Treating these accurately is important to avoid unexpected behavior. Later in this section, we will demonstrate a method for comparing `State` objects with user-defined accuracy.

2.2.2 Hilbert space operations

The `State` class represents quantum many-body states and supports corresponding operations in the Hilbert space.

A. Multiplication by a scalar. A `State` object can be multiplied by a real or complex number resulting in a new `State` instance:

```
print(2 * state)
```

```
  |1100>  *  2.0
  |1010>  *  2.0
  |1001>  *  2.0
  |0110>  *  2.0
  |0101>  *  2.0
  ...
```

The resulting `State` object contains a newly created NumPy array of `coeffs`, while the underlying `basis` is shared between the new and original `State` objects. The scalar can be used as both the left and right operand. In addition to multiplication by a scalar, SOLAX also supports division by a scalar and the unary - operator.

B. Addition. Two `State` objects can be added using the + operator, which also applies the + operator to their underlying `Basis` objects, as described in the previous section. The coefficients associated with the same Slater determinants in the operands are summed. For instance, consider two `State` objects obtained from `state` via "fancy" indexing:

```
state1 = state[0, 1, 4]
print(state1)
```

```
  |1100>  *  1.0
  |1010>  *  1.0
  |0101>  *  1.0
```

```
state2 = state[0, 3]
print(state2)
```

```
  |1100>  *  1.0
  |0110>  *  1.0
```

The addition operation gives:

```
print(state1 + state2)
```

```
  |1100>  *  2.0
  |1010>  *  1.0
  |0101>  *  1.0
  |0110>  *  1.0
```

The subtraction operation based on addition and unary negation are also supported:

```
print(state1 - state2)
```

```
|1100>  *  0.0
|1010>  *  1.0
|0101>  *  1.0
|0110>  *  -1.0
```

Important note! As demonstrated in the previous example, Slater determinants with zero coefficients are not automatically removed in SOLAX. Instead, determinants with exact zero coefficients, as well as those with very small coefficients, must be manually removed using a user-defined cutoff. This approach allows us to treat equivalently the following examples of "zeros" (where only the first is exactly zero as represented in the machine):

```
print(0.1 + 0.1 - 0.2)
```

```
0.0
```

```
print(0.1 + 0.2 - 0.3)
```

```
5.551115123125783e-17
```

We show in the following how to "chop off" such zeros using SOLAX tools.

C. Scalar product and normalization. SOLAX supports Hermitian scalar product of `State` objects:

```
print(state1 * state2)
```

```
1.0
```

which allows to compute the norm as

```
print(state * state)
```

```
6.0
```

Once the norm is known, a `State` can be normalized using division by a scalar. SOLAX offers a shortcut for this operation implemented as the `normalize` method:

```
state_normalized = state.normalize()
print(state_normalized * state_normalized)
```

```
1.0000000000000002
```

The `normalize` method does not transform the initial `State` object but returns a new one. The underlying `basis` is shared.

2.2.3 Chopping and equality of states

Chopping a `State` object to a specified threshold is a useful operation implemented in SOLAX. As demonstrated below, the chop operation also enables the comparison of two `State` objects with a user-defined error margin.

A. The chop method. The `chop` method removes all Slater determinants from a `State` object whose coefficients have absolute values smaller than a specified threshold. This operation creates a new `State` instance, leaving the original one unmodified. For demonstration, we consider the following `State`:

```
state3 = state2[-1]
state123 = state1 - state2 - state3
print(state123)
```

```
|1100>  *  0.0
|1010>  *  1.0
|0101>  *  1.0
|0110>  *  -2.0
```

Below we show 3 `State` objects obtained by chopping `state123` with respect to different thresholds:

```
state_chopped1 = state123.chop(1e-14)
print(state_chopped1)
```

```
|1010>  *  1.0
|0101>  *  1.0
|0110>  *  -2.0
```

```
state_chopped2 = state123.chop(1.5)
print(state_chopped2)
```

```
|0110>  *  -2.0
```

```
state_chopped3 = state123.chop(2.5)
print(state_chopped3)
print(len(state_chopped3))
```

```
0
```

As shown by the first example, the `chop` method allows the user to manually delete determinants with (numerically) zero coefficients. We stress again that in SOLAX this is not done automatically.

B. Equality of two State objects. As mentioned above, the direct equality relation == is not implemented for the `State` class due to the finite machine precision of the involved real or complex coefficients. Instead, we can construct the difference of the `State` objects and chop the result with respect to a small cutoff. If the obtained `State` is empty, then the compared `State` objects were close within the precision determined by the threshold. For demonstration, we use the following `State` objects `state_a` and `state_b` which are not exactly equal due to the machine error of the involved coefficients:

```
state_mini = state[:2]
state_a = 0.1 * state_mini + 0.2 * state_mini
print(state_a)
```

```
|1100>  *  0.30000000000000004
|1010>  *  0.30000000000000004
```

```
state_b = 0.3 * state_mini
print(state_b)
```

```
|1100>  *  0.3
|1010>  *  0.3
```

Following the described procedure, we obtain:

```
state_diff = state_a - state_b
print(state_diff)
```

```
|1100>  *  5.551115123125783e-17
|1010>  *  5.551115123125783e-17
```

Now, chopping `state_diff` with respect to a small threshold gives an empty state:

```
state_zero = state_diff.chop(1e-14)
print(len(state_zero))
```

```
0
```

meaning that `state_a` and `state_b` are indeed equal within the error of `1e-14`. Practice shows that proper control here can be crucial to avoid unexpected behavior.

2.3 OperatorTerm

We now introduce quantum mechanical operators represented in SOLAX by two classes: `OperatorTerm` and `Operator`. The `OperatorTerm` class efficiently represents products of ladder operators with the same structure, while the `Operator` class encapsulates multiple `OperatorTerm` objects with different structures in a single entity. We will further discuss quantum operators as implemented in SOLAX through a concrete example, beginning with the `OperatorTerm` class.

Once more we consider the case of two electrons occupying four spin-orbitals. In this example, the four slots with indices 0, 1, 2, and 3 correspond to the electronic states $\uparrow^{(1)}$, $\downarrow^{(1)}$, $\uparrow^{(2)}$, and $\downarrow^{(2)}$, respectively. Here, the arrow and superscript indicate the spin and orbital quantum numbers, respectively. We examine the following hopping operator:

$$\hat{V} = v\left(\hat{a}_0^\dagger \hat{a}_2 + \hat{a}_1^\dagger \hat{a}_3\right) + \text{h.c.} \equiv v\left(\hat{a}_{1\uparrow}^\dagger \hat{a}_{2\uparrow} + \hat{a}_{1\downarrow}^\dagger \hat{a}_{2\downarrow}\right) + \text{h.c.}\,, \tag{1}$$

and assume $v = 1$. For clarity, we additionally provided the formula in the format typically used in quantum mechanics via "≡". Note that $\hat{V}$ consists of ladder operator products of the same structure $\hat{a}_i^\dagger \hat{a}_j$ and, therefore, can be represented by one `OperatorTerm` object. We start with the non-Hermitian operator

$$\hat{V}_0 = \hat{a}_0^\dagger \hat{a}_2 + \hat{a}_1^\dagger \hat{a}_3 \equiv \hat{a}_{1\uparrow}^\dagger \hat{a}_{2\uparrow} + \hat{a}_{1\downarrow}^\dagger \hat{a}_{2\downarrow}\,, \tag{2}$$

and gradually build up $\hat{V}$ using the functionality provided in SOLAX.

2.3.1 Object construction

Here we show how `OperatorTerm` objects are instantiated by considering the example of the introduced $\hat{V}_0$ operator. However, to keep the explanation generic, we denote the number of multipliers in each ladder operator product as L, and the number of the summed products as K (specifically for $\hat{V}_0$ we have $L = 2$ and $K = 2$).

Each `OperatorTerm` has 3 ingredients:

- `daggers` is a tuple of 0s and 1s of length L showing which ladder operators in the products are annihilation (0s) and which are creation (1s) operators;
- `posits` is a 2D NumPy array of shape $K \times L$ and integer type indicating at which positions (as counted from 0) the ladder operators in each product act;
- `coeffs` is a 1D NumPy array of length K containing real or complex coefficients for each product.

For $\hat{V}_0$:

```
daggers = (1, 0)

posits = np.array([
    [0, 2],
    [1, 3]
])

coeffs = np.array([
    1.0,
    1.0
])
```

An `OperatorTerm` instance is then created as

```
V0 = sx.OperatorTerm(daggers, posits, coeffs)
print(V0)
```

```
OperatorTerm(
    daggers=(1, 0),
    posits=array([[0, 2],
           [1, 3]]),
    coeffs=array([1., 1.])
)
```

We stress that in SOLAX, the ordering of ladder operators is fully the choice of the user which is controlled by the argument `daggers` indicating the positions of the creation and annihilation operators.

Unlike the `Basis` and `State` classes, which contain NumPy arrays with encoded Slater determinants that are not directly readable by the user, the `OperatorTerm` class displays its underlying NumPy arrays without any special formatting when converted to a Python string and printed. Additionally, if the `posits` array passed to `OperatorTerm` contains repeated rows, these duplicates are automatically removed, and the corresponding coefficients in `coeffs` are summed.

2.3.2 Similarities with the State class

The `OperatorTerm` class has strong similarities with the `State` class. Conceptually, they both represent expansions over some basis elements (ladder operator products and Slater determinants, respectively). At the technical level, both classes incorporate a 2D NumPy array with an accompanying 1D coefficient array. We list here the analogous features without going into comprehensive details.

- `OperatorTerm` implements the Python sequence protocol and supports the NumPy-style "fancy" and boolean indexing [36] (see also the section on the `Basis` class).
- Objects of `OperatorTerm` can be added using the + operator. If the `daggers` tuples of the summands are equal (i.e. the quantum operators have the same structure), the result is of the `OperatorTerm` type and otherwise of the `Operator` type (the latter class is considered in the next section).
- The `OperatorTerm` class supports multiplication with scalars.
- `OperatorTerm` objects can be "chopped" with respect to a real threshold using the `chop` method.
- Equality relation == is not implemented for the `OperatorTerm` class. As for the `State` class, the equality up to a user-determined precision can be checked with the binary - operator and subsequent chopping.

2.3.3 Hermitian conjugate

The `OperatorTerm` class supports the operation of Hermitian conjugation via the `hconj` property returning a new `OperatorTerm` object:

```
print(V0.hconj)
```

```
OperatorTerm(
    daggers=(1, 0),
    posits=array([[2, 0],
           [3, 1]]),
    coeffs=array([1., 1.])
)
```

Using the introduced functionality, we can construct now the full operator $\hat{V}$ as

```
V = V0 + V0.hconj
print(V)
```

```
OperatorTerm(
    daggers=(1, 0),
    posits=array([[0, 2],
           [1, 3],
           [2, 0],
           [3, 1]]),
    coeffs=array([1., 1., 1., 1.])
)
```

Note that the Hermitian conjugate $\hat{V}_0^\dagger$ has the same operator structure as $\hat{V}_0$, and hence the sum is represented by an `OperatorTerm` object.

2.3.4 Acting on states and bases

`OperatorTerm` objects represent quantum mechanical operators, meaning they can act on quantum states. To illustrate this, we consider the singlet state $|\Psi\rangle$, which is a normalized, anti-symmetric combination of a spin-up and spin-down electron, ensuring that the total spin $S_z = 0$:

```
basis = sx.Basis(["1001", "0110"])
cfs = np.array([1.0, -1.0])

psi = sx.State(basis, cfs)
psi = psi.normalize()

print(psi)
```

```
|1001>  *  0.7071067811865475
|0110>  *  -0.7071067811865475
```

The operator action $\hat{V}|\Psi\rangle$ can be now performed as a call

```
result_psi = V(psi)
print(result_psi)
```

```
|1100>  *  1.414213562373095
|0011>  *  1.414213562373095
```

Additionally, it is possible to act with an `OperatorTerm` directly on `Basis` objects. The result is then also of type `Basis` and contains the same Slater determinants as when acting on a `State`:

```
result_basis = V(psi.basis)
print(result_basis)
```

```
1100
0011
```

As will be demonstrated later in this work, the operation of acting directly on `Basis` objects is useful for iterative basis extension procedures via acting with operators.

2.3.5 GPU acceleration and batches

Acting with `OperatorTerm` is implemented using the JAX library [21] which supports computations on an NVIDIA GPU. Therefore, if such GPU is available on the machine and a GPU-capable version of JAX is installed, it will be automatically used for this operation.

Batching. Since GPU memory is often scarce, we provide the user with the possibility to perform the operator action in batches by using the call arguments `det_batch_size` and `op_batch_size`. They are responsible for batching the `State` (or `Basis`) object and the `OperatorTerm` object, respectively. Note that these are keyword-only arguments and have to be provided with the argument names explicitly. For example:

```
result_psi_batches = V(psi, det_batch_size=1, op_batch_size=2)
```

We can now use the standard procedure to ensure that the `State` objects obtained with and without batching are equal within a very small error:

```
s = result_psi_batches - result_psi
s = s.chop(1e-14)
print(len(s))
```

```
0
```

Note, however, that the internal ordering of the Slater determinants in the resulting objects may be different as computed with and without batching. We note also that in this demonstration example, very small batch sizes are chosen, but in general, the user should aim at maximally exhausting the GPU memory.

Multiple GPUs. Parallelization across multiple GPUs is currently under development and is made partially available in the current SOLAX version as an experimental feature. The action of an `OperatorTerm` can be enabled for multi-GPU mode by setting the keyword-only argument `multiple_devices=True`. In this mode, the batches of the `State` (or `Basis`) object are automatically distributed across all available GPUs on the machine. Note that for multi-GPU processing, the argument `det_batch_size` must be specified for creation of multiple batches. Otherwise, only one batch will be present and therefore only one GPU will be used. We stress again that this functionality is at the moment under development and is partially accessible as an experimental feature.

2.4 Operator

Thus far, we have considered quantum operators consisting of ladder operator products with the same structure, differing only in the positions on which the ladder operators act. These are represented in SOLAX using the `OperatorTerm` class. As the next step, we introduce the `Operator` class, which encapsulates `OperatorTerm` objects of different structures along with a scalar term. An `Operator` object can encode any quantum operator expressed in the second quantization formalism using annihilation and creation operators.

While computations can be performed using only the `OperatorTerm` class introduced in the previous section, we recommend the users to follow these guidelines:

- prefer the `Operator` class for basic usage like acting on quantum states or basis sets;
- access the underlying `OperatorTerm` objects to fine-tune the operator.

In this section we will demonstrate this approach in practice.

2.4.1 Construction of simple operators

The basic way to instantiate `Operator` objects is based on the same ingredients as for the `OperatorTerm` class, i.e. `daggers`, `posits` and `coeffs` (see Section 2.3.1). These arguments can be passed directly to the `Operator` constructor:

```
op = sx.Operator(daggers, posits, coeffs)
print(op)
```

```
Operator({
    (1, 0): OperatorTerm(
        daggers=(1, 0),
        posits=array([[0, 2],
                      [1, 3]]),
        coeffs=array([1., 1.])
    )
})
```

Explanation of the printing output. From the technical perspective, the `Operator` class implements the Python mapping protocol. In practice, this means that its objects behave similarly to Python dictionaries. Specifically, `Operator` objects store their underlying `OperatorTerm` objects as values in key-value pairs, with the keys being the corresponding `daggers` tuples. It is also possible to include a scalar term, which is associated with the string key `"scalar"`. This structure is reflected when `Operator` objects are converted to a Python string and printed, as demonstrated in the example above.

Once `daggers`, `posits` and `coeffs` are received, SOLAX creates automatically an `OperatorTerm` from the provided arguments and wraps it in an `Operator` object for convenient usage and further extension. In particular, this can be seen from the printed output for the op object above. Alternatively, the same `Operator` can be instantiated directly from the `OperatorTerm`:

```
print(sx.Operator(V0))
```

```
Operator({
    (1, 0): OperatorTerm(
        daggers=(1, 0),
        posits=array([[0, 2],
               [1, 3]]),
        coeffs=array([1., 1.])
    )
})
```

Here we reused the `OperatorTerm` object `V0` created in the previous section.

2.4.2 Addition as a way to build operators

The `Operator` created above is still trivial in the sense that it hosts only one `OperatorTerm`. More advanced `Operator` objects can be constructed from more basic ones using addition. Also addition of an `Operator` with an `OperatorTerm` or a scalar leads to another `Operator`. Moreover, as mentioned in the previous section, addition of two incompatible `OperatorTerm` objects does not lead to an error but creation of an `Operator` hosting the summands. For illustration, we introduce the following onsite energy operator:

$$\hat{U} = u_{01}\,\hat{a}_0^\dagger\hat{a}_0\hat{a}_1^\dagger\hat{a}_1 + u_{23}\,\hat{a}_2^\dagger\hat{a}_2\hat{a}_3^\dagger\hat{a}_3 \equiv u_1\,\hat{a}_{1\uparrow}^\dagger\hat{a}_{1\uparrow}\hat{a}_{1\downarrow}^\dagger\hat{a}_{1\downarrow} + u_2\,\hat{a}_{2\uparrow}^\dagger\hat{a}_{2\uparrow}\hat{a}_{2\downarrow}^\dagger\hat{a}_{2\downarrow}\,, \tag{3}$$

with $u_{01} = 0.25 \equiv u_1$ and $u_{23} = 0.75 \equiv u_2$ as an example. As before, we additionally provided the formula in the format typically used in quantum mechanics via "≡". We encode $\hat{U}$ as an `Operator` object using the described standard way:

```
daggers_u = (1, 0, 1, 0)
posits_u = np.array([
    [0, 0, 1, 1],
    [2, 2, 3, 3]
])
coeffs_u = np.array([0.25, 0.75])

U = sx.Operator(daggers_u, posits_u, coeffs_u)
print(U)
```

```
Operator({
    (1, 0, 1, 0): OperatorTerm(
        daggers=(1, 0, 1, 0),
        posits=array([[0, 0, 1, 1],
               [2, 2, 3, 3]]),
        coeffs=array([0.25, 0.75])
    )
})
```

Now we can construct e.g. the compound operator

$$\hat{H} = \mathbb{1} + \hat{V} + \hat{U} = \mathbb{1} + \hat{V}_0 + \hat{V}_0^\dagger + \hat{U}\,, \tag{4}$$

directly as

```
H = 1 + V0 + V0.hconj + U
print(H)
```

```
Operator({
    (1, 0): OperatorTerm(
        daggers=(1, 0),
        posits=array([[0, 2],
                      [1, 3],
                      [2, 0],
                      [3, 1]]),
        coeffs=array([1., 1., 1., 1.])
    ),
    scalar: 1,
    (1, 0, 1, 0): OperatorTerm(
        daggers=(1, 0, 1, 0),
        posits=array([[0, 0, 1, 1],
                      [2, 2, 3, 3]]),
        coeffs=array([0.25, 0.75])
    )
})
```

As seen from the printed output, the resulting `Operator` consists of the scalar 1 and two `OperatorTerm` objects.

2.4.3 Similarities with the OperatorTerm class

The `Operator` class contains operations which are similar to the `OperatorTerm` class.

- Linear operations of addition and multiplication with scalars.
- Hermitian conjugation using the `hconj` property.
- Action on `State` and `Basis` objects — is delegated to the underlying `OperatorTerm` components and multiplication with the scalar with subsequent addition of the partial results. Note that in case of action on a `Basis`, the scalar, if present, acts effectively as the unity operator (even if it is equal to zero).
- The keyword-only batching arguments `det_batch_size` and `op_batch_size` are available for the `Operator` action, and control batching for the `OperatorTerm` components.
- We remind the user here, that the underlying `OperatorTerm` objects automatically support computations on an NVIDIA GPU.
- For leveraging the multi-GPU parallelization (experimental!), the `Operator` action call can receive the keyword-only argument `multiple_devices=True` which propagates to the underlying `OperatorTerm` objects. In this case, batches corresponding to `det_batch_size` will be distributed over a few GPUs, if available.

2.4.4 Manipulations with Operator objects

A. Accessing underlying components. A key difference between the `Operator` and `OperatorTerm` classes is the container type they implement. `OperatorTerm` objects are sequences and can be indexed using integer positions and slices, as well as NumPy's "fancy" and boolean indexing mechanisms. In contrast, the `Operator` class implements the Python mapping protocol, making it similar to Python dictionaries, which are indexed via their keys. For `Operator` objects, the keys are either the `daggers` tuples or the `"scalar"` string. For example:

```
print(H[1, 0, 1, 0])
```

```
OperatorTerm(
    daggers=(1, 0, 1, 0),
    posits=array([[0, 0, 1, 1],
           [2, 2, 3, 3]]),
    coeffs=array([0.25, 0.75])
)
```

```
print(H["scalar"])
```

```
1
```

Note that for indexing, the "tuple" parenthesis for `daggers` can be omitted as in the example above. As usual Python dictionaries, `Operator` instances support views `keys`, `values` and `items` for iteration over their entries.

B. Operator length. `Operator` objects as mappings have length, which however only reflects the number of the stored components and is not related to the length of the underlying `OperatorTerm` objects. Indeed:

```
print(len(H))
```

```
3
```

whereas for the contained `OperatorTerm` components the lengths are:

```
for key, term in H.items():
    if key != "scalar":
        print(f"Length of the OperatorTerm {key} is {len(term)}")
```

```
Length of the OperatorTerm (1, 0) is 4
Length of the OperatorTerm (1, 0, 1, 0) is 2
```

C. Dropping components. We have seen how `Operator` instances can be enriched with new components using addition. Conversely, if there is a need to remove an `OperatorTerm` or the scalar, this can be done using the `drop` method, which takes the key of the component to be removed and returns a new `Operator` object without it. The original `Operator` remains unmodified. For example:

```
H_without_V = H.drop(1, 0)
print((1, 0) in H_without_V)
```

```
False
```

```
H_without_scalar = H.drop("scalar")
print("scalar" in H_without_scalar)
```

```
False
```

Here we used the Python `in` keyword for membership checks. This becomes automatically possible since the `Operator` class implements the mapping interface.

D. Chopping OperatorTerm components. The chopping operation is implemented for the `Operator` class as the `chop` method, but can be applied only to a particular `OperatorTerm` via its `daggers` key. The returned `Operator` object contains the chopped version of the corresponding `OperatorTerm` or does not contain it at all if it has become empty after chopping. The initial `Operator` stays unchanged. Chopping for the scalar term is not supported. For instance, chopping `U` with respect to the cutoff 0.5 is performed as

```
H_chopped1 = H.chop((1, 0, 1, 0), 0.5)
print(len(H_chopped1[1, 0, 1, 0]))
```

```
1
```

leaving only one entry out of the two in the corresponding `OperatorTerm`. Chopping with respect to 1.0 leads to chopping it off completely:

```
H_chopped2 = H.chop((1, 0, 1, 0), 1.0)
print((1, 0, 1, 0) in H_chopped2)
```

```
False
```

2.5 OperatorMatrix

To solve a quantum many-body eigenvalue problem, it is often necessary to construct the matrix representation of a quantum mechanical operator (e.g. Hamiltonian) on a given basis set. We address this requirement with the SOLAX class `OperatorMatrix`, which provides tools for efficient matrix construction. It is important to note that the `OperatorMatrix` class is designed solely for the efficient construction of the operator matrix, while subsequent diagonalization is performed by the user with the help of the SciPy library [26].

2.5.1 Obtaining an OperatorMatrix

For the demonstration, we reconsider again the case of two electrons in four spin-orbitals. We use the `Operator` object `H` constructed in the last section and create here also a `Basis` of Slater determinants with the spin projection $S_z = 0$:

```
basis = sx.Basis(["1001", "1100", "0110", "0011"])
```

The matrix of the operator $\hat{H}$ on this basis can be built directly using the `build_matrix` method of the corresponding `Operator` object (this method is available also for the `OperatorTerm` class):

```
matrix = H.build_matrix(basis)
```

The result is an object is of the `OperatorMatrix` class, which stores matrix elements in a coordinate sparse format, meaning only non-zero matrix elements are stored. Here, we highlight the main features of the matrix construction operation:

- It is possible to construct non-square matrices using two distinct `Basis` objects for rows and columns by passing them as arguments to the `build_matrix` method.
- The `build_matrix` function supports the keyword-only arguments `det_batch_size` and `op_batch_size` which are used in evaluation of the matrix elements via action of the `OperatorTerm` objects (see the section on the `OperatorTerm` class).
- Matrix evaluation inherits from the `OperatorTerm` class the possibility to perform computations automatically on an NVIDIA GPU, if available.
- Computations on multiple GPUs (experimental!) can be switched on by providing `multiple_devices=True` to the `build_matrix` method. In this case, batches corresponding to `det_batch_size` will be distributed over a few GPUs, if available (see the section on the `OperatorTerm` class).

The dimensions of the constructed matrix can be accessed using the read-only `size` property:

```
print(matrix.size)
```

```
(4, 4)
```

The number of the non-zero matrix elements (i.e. the total number of the stored matrix elements) can be obtained as

```
print(matrix.num_nonzero)
```

```
12
```

The content of an `OperatorMatrix` object can be viewed after conversion to SciPy and NumPy which we discuss in the following.

2.5.2 Conversion to SciPy and NumPy

The built `OperatorMatrix` can be now converted to the SciPy format using the `to_scipy` method:

```
coo_matrix = matrix.to_scipy()
```

The returned object is of type `scipy.sparse.coo_matrix`, and stores the matrix in the coordinate sparse format similarly to the `OperatorMatrix` class. If necessary, the user can now convert it to a different sparse format using the SciPy means. Here, we convert the matrix to the usual NumPy dense format and print it:

```
dense_matrix = coo_matrix.todense()
print(dense_matrix)
```

```
[[ 1.    1.    0.    1.  ]
 [ 1.    1.25 -1.    0.  ]
 [ 0.   -1.    1.   -1.  ]
 [ 1.    0.   -1.    1.75]]
```

Note that the NumPy dense representation is feasible only for small matrices, e.g. for demonstration or testing purposes. In this section we will often perform such conversion of `OperatorMatrix` objects in order to print their content in a usual matrix format. Therefore, we define the shortcut function:

```
def print_matrix(m):
    print(m.to_scipy().todense())
```

2.5.3 Manipulations with OperatorMatrix objects

Once obtained from an `Operator` or `OperatorTerm`, the `OperatorMatrix` object allows for further useful manipulations, which we present here. In all examples considered below, a new transformed `OperatorMatrix` instance is created while the original remains unmodified. Here, we always utilize the `matrix` object constructed and shown above.

A. Displace. The `displace` method allows to shift the matrix content along the row and the column axes with the corresponding change of the matrix dimensions. The two method arguments are the number of positions the matrix is displaced by vertically (row axis) and horizontally (column axis), respectively. The shifts can be positive and negative. For example:

```
print_matrix(
    matrix.displace(2, 1)
)
```

```
[[ 0.    0.    0.    0.    0.  ]
 [ 0.    0.    0.    0.    0.  ]
 [ 0.    1.    1.    0.    1.  ]
 [ 0.    1.    1.25 -1.    0.  ]
 [ 0.    0.   -1.    1.   -1.  ]
 [ 0.    1.    0.   -1.    1.75]]
```

```
print_matrix(
    matrix.displace(-1, -1)
)
```

```
[[ 1.25 -1.    0.  ]
 [-1.    1.   -1.  ]
 [ 0.   -1.    1.75]]
```

As seen from these examples, the newly created positions are effectively filled with zeros, whereas the entries with resulting negative positions are dropped.

B. Window. The `window` method implements a rectangular filter, which sets all elements outside this rectangle to zero without changing the matrix shape. Technically, the filtered out matrix elements are directly discarded, since only non-zero matrix elements are stored in `OperatorMatrix` objects. For instance:

```
print_matrix(
    matrix.window((1, 1), (3, 4))
)
```

```
[[ 0.    0.    0.    0.  ]
 [ 0.    1.25 -1.    0.  ]
 [ 0.   -1.    1.   -1.  ]
 [ 0.    0.    0.    0.  ]]
```

The tuples passed to the `window` method correspond to the positions of the left upper (inclusive) and the right lower (exclusive) corners of the filter. If tuples (a, b) and (c, d) are passed, then the matrix values at the intersection of rows i: $a \leq i < c$ and columns j: $b \leq j < d$ survive, whereas the other matrix elements become zero. If any of the arguments a, b, c, d is `None`, it will be replaced by a position leading to the largest possible filter size.

C. Shrink basis. After the matrix on a particular basis is constructed, it is straightforward to obtain the matrix on any sub-basis by extracting the corresponding matrix elements. This can be done using the `shrink_basis` method as we show in the following. We choose the sub-basis of our `basis` object corresponding to the electrons occupying different spatial orbitals:

```
sub_basis = basis[0, 2]
print(sub_basis)
```

```
1001
0110
```

Then the matrix of the same operator $\hat{H}$ on this sub-basis can be obtained as

```
print_matrix(
    matrix.shrink_basis(basis, sub_basis)
)
```

```
[[1. 0.]
 [0. 1.]]
```

By default, this operation shrinks the basis along both axes. It is also possible to shrink the basis only along the row or the column axis by passing an additional argument `axis=0` or `axis=1` to the `shrink_basis` method, respectively.

Important note! Both the sub-basis and the initial basis have to be passed to the `shrink_basis` method. Therefore, it is important to perform the sub-matrix extraction prior to any matrix displacements, since displaced matrices are not related to the initial basis anymore.

D. Chopping. The `chop` method provides the possibility to discard matrix elements with absolute values less than a user-defined threshold. These entries are effectively set to zero (we remind that zeros are not stored in `OperatorMatrix` objects). As an example, we chop our matrix with respect to the threshold `1.1`:

```
print_matrix(
    matrix.chop(1.1)
)
```

```
[[0.   0.   0.   0.  ]
 [0.   1.25 0.   0.  ]
 [0.   0.   0.   0.  ]
 [0.   0.   0.   1.75]]
```

2.5.4 Linear operations and Hermitian conjugate

Like other SOLAX classes related to quantum mechanical operators, the `OperatorMatrix` class supports addition, multiplication by real or complex scalars, and Hermitian conjugation via the `hconj` property. However, it is important to note that the addition operation deviates from the standard linear algebra convention. Specifically, `OperatorMatrix` objects can be added regardless of their shape. If the dimensions do not match, the matrices are effectively padded with zeros to form a minimal rectangle that encompasses both matrices before being added. This behavior is inherited from and natural for the `OperatorMatrix` implementation in the sparse format. To demonstrate this feature, we create two matrices of size 3×3 and 5×2 from our initial 4×4 matrix using displacements and then add them to obtain a matrix of size 5×3:

```
matrix3_3 = matrix.displace(-1, -1)
print_matrix(matrix3_3)
```

```
[[ 1.25 -1.    0.  ]
 [-1.    1.   -1.  ]
 [ 0.   -1.    1.75]]
```

```
matrix5_2 = matrix.displace(1, -2)
print_matrix(matrix5_2)
```

```
[[ 0.    0.  ]
 [ 0.    1.  ]
 [-1.    0.  ]
 [ 1.   -1.  ]
 [-1.    1.75]]
```

```
print_matrix(
    matrix3_3 + matrix5_2
)
```

```
[[ 1.25 -1.    0.  ]
 [-1.    2.   -1.  ]
 [-1.   -1.    1.75]
 [ 1.   -1.    0.  ]
 [-1.    1.75  0.  ]]
```

Later in this work we will use the introduced manipulations for efficient computations with operator matrices.

2.5.5 Equality of OperatorMatrix objects

As also the other SOLAX classes containing real or complex numbers, the `OperatorMatrix` class does not directly support the equality relation and the == operator. The user can check closeness of non-zero elements in two `OperatorMatrix` objects by

a) subtracting them;

b) using the `chop` method with respect to a small threshold;

c) ensuring that the `num_nonzeros` property returns zero.

Note that due to the specific `OperatorMatrix` implementation, this procedure disregards completely the matrix dimensions and only checks closeness of non-zero elements having the same position in the matrices. For example, consider the following matrices `m1` and `m2`:

```
m1 = matrix.window((0, 0), (2, 2))
print_matrix(m1)
```

```
[[1.   1.   0.   0.  ]
 [1.   1.25 0.   0.  ]
 [0.   0.   0.   0.  ]
 [0.   0.   0.   0.  ]]
```

```
m2 = matrix.shrink_basis(basis, basis[:2])
print_matrix(m2)
```

```
[[1.   1.  ]
 [1.   1.25]]
```

The outlined procedure gives

```
print(
    (m1 - m2).chop(1e-14).num_nonzero == 0
)
```

```
True
```

That is, all non-zero matrix elements having the same position are equal (within the chosen accuracy), but still the dimensions may differ, as in the considered case. The latter can be additionally compared as

```
print(m1.size == m2.size)
```

```
False
```

2.6 Demonstration computation for SIAM

After the introduction of the foundational components of the SOLAX package, we now demonstrate its application to the Single Impurity Anderson Model (SIAM). This example illustrates how SOLAX can be used to construct and analyze finite size quantum systems by efficiently handling basis sets, states, and operators, specifically to find the ground state of a complex quantum many-body model.

2.6.1 Introduction to SIAM

The SIAM is a cornerstone model in the study of strongly correlated electron systems. Initially introduced by Anderson [37], the model describes a single localized level (the "impurity") with onsite interaction U coupled to a continuum of noninteracting conduction electrons (the "bath"). It was proposed to capture essential physics relevant to magnetic impurities in metals, such as those found in dilute alloys like gold doped with iron. Notably, it describes the Kondo effect, a phenomenon in which the impurity spin is screened by the surrounding conduction electrons at low temperatures, resulting in a highly correlated many-body ground state. In addition to being a paradigm in its own right, the SIAM also serves as an auxiliary model for the dynamical mean-field (DMFT) solution of the Hubbard and related models [38–40].

Today, the most frequently used SIAM solvers are Quantum Monte Carlo methods which usually work on the imaginary Matsubara time domain, see e.g. Refs. [41, 42]. While they allow for continuous baths, they often struggle with the fermionic sign problem at low temperatures [43]. Complementary diagonalization solvers like our approach do not suffer from any sign problem but require a discretized bath, which is represented by a finite number of single particle energy levels, each coupled to the impurity through hybridization terms. This introduces a computational challenge: the number of bath sites N_{bath} controls the resolution of the bath representation, and pushing N_{bath} to larger values is crucial to accurately capture

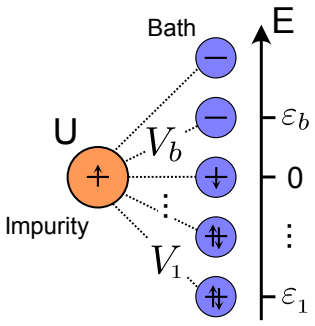

Figure 1: Schematic illustration of SIAM with $N_{\text{bath}} = 5$ bath sites. The model is considered in the "star geometry", in which the correlated impurity (orange circle) hybridizes with the non-interacting and uncorrelated bath sites (blue circles). The impurity onsite energy is set to zero. The onsite impurity interaction is given by U. The bath site energies are ε_b and the hybridization strengths are V_b.

the continuous nature of the bath, especially at low temperatures. Our SOLAX package is well-suited to this task, as it can efficiently handle large fermionic clusters. In turn, the possibility of incrementally enlarging the discretized SIAM model makes it an ideal and flexible test case for our code.

2.6.2 Model description

In Fig. 1 we sketch the SIAM which consists of an impurity with an effective onsite interaction U, coupled to a set of N_{bath} non-interacting bath sites with energies ε_b and hybridization strengths V_b. The SIAM Hamiltonian reads

$$\hat{H}_{\text{SIAM}} = \hat{H}_{\text{imp}} + \hat{H}_{\text{bath}} + \hat{H}_{\text{hyb}}\,, \tag{5}$$

$$\hat{H}_{\text{imp}} = U\left(\hat{n}_{\text{imp}\uparrow} - \frac{1}{2}\right)\left(\hat{n}_{\text{imp}\downarrow} - \frac{1}{2}\right), \tag{6}$$

$$\hat{H}_{\text{bath}} = \sum_{\sigma\in\{\uparrow,\downarrow\}}\sum_{b=1}^{N_{\text{bath}}} \varepsilon_b \hat{n}_{b\sigma}\,, \tag{7}$$

$$\hat{H}_{\text{hyb}} = \sum_{\sigma\in\{\uparrow,\downarrow\}}\sum_{b=1}^{N_{\text{bath}}} V_b\left(\hat{a}^\dagger_{\text{imp}\,\sigma}\hat{a}_{b\sigma} + \text{h.c.}\right), \tag{8}$$

where $\hat{a}^\dagger_{\alpha\sigma}$ and $\hat{a}_{\alpha\sigma}$ are fermionic creation and annihilation operators with α labeling the respective (impurity or bath) site, and $\hat{n}_{\alpha\sigma} \equiv \hat{a}^\dagger_{\alpha\sigma}\hat{a}_{\alpha\sigma}$ are the corresponding occupation operators. Note that we assumed here the onsite impurity energy to be zero. The parameters of the model are the number of non-interacting bath sites N_{bath}, the onsite energies of the bath sites ε_{b}, the hybridization amplitudes V_b, and the particle-hole symmetric interaction on the impurity site U.

We follow Ref. [16] and choose the parameters ε_b and V_b such that our star geometry maps directly to a 1D chain i.e., an impurity site coupled to the first site of a 1D bath chain with hybridization V with constant nearest neighbor hopping t [44]. To this end, we set

$$\begin{aligned}\varepsilon_b &= -2t\cos\left(\frac{b\pi}{N_{\text{bath}}+1}\right),\\ V_b &= V\sqrt{\frac{2}{N_{\text{bath}}+1}}\sqrt{1-\left(\frac{\varepsilon_b}{2t}\right)^2},\end{aligned} \tag{9}$$

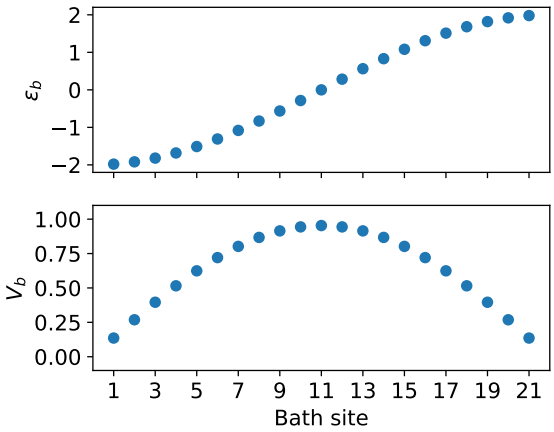

Figure 2: The bath site energies ε_b and the hybridization strengths V_b for $N_{\text{bath}} = 21$.

with the bath site index b running from 1 to N_{bath}. In the present work, we choose $V = \sqrt{10}$ eV and $t = 1.0$ eV, and give all energies in units of t. Moreover, we restrict our calculations to an odd number of bath sites (such that there is always a bath site at $\varepsilon_b = 0$) and half-filling, i.e. $N_e = N_{\text{bath}} + 1$.

To set up the parameters for our bath we use the `build_bath` function:

```
def build_bath(N_bath):
    ii = np.arange(N_bath) + 1
    xx = ii * np.pi / (N_bath + 1)
    e_bath = -2 * np.cos(xx)

    V0 = np.sqrt(20 / (N_bath + 1))
    V_bath = V0 * np.sqrt(1 - (e_bath / 2)**2)

    return e_bath, V_bath
```

In Fig. 2 we show ε_b and V_b for the case of $N_{\text{bath}} = 21$ as constructed using this function.

2.6.3 Eigenvalue problem and solution procedure

We aim to compute the energy of the ground state, which is known to belong to the $S_z = 0$ sector. An exact solution would require the construction and partial diagonalization of the Hamiltonian matrix over the basis set of all possible Slater determinants with $S_z = 0$. However, even with a few tens of bath sites, the complete basis becomes combinatorially large and numerically intractable. Therefore, it is common to perform these computations iteratively on a growing partial basis. Starting with an initial set of Slater determinants, the following iterations are performed:

- construction and partial diagonalization of the Hamiltonian matrix on the current basis set;
- extension of the basis set by acting on it with an extension operator $\hat{O}$ (here we choose $\hat{O} = \hat{H}$).

The energies obtained in each iteration at the diagonalization stage are monitored in order to stop the computation once convergence is achieved.

We mention in passing, that the choice of the initial set of Slater determinants, as well as the choice of the extension operator, can affect the convergence of the procedure [16]. A natural choice for the initial set is the mean-field solution which consists of a single or a few degenerate Slater determinants. Below, we chose the twofold spin degenerate $S_z = 0$ solution of the zero-hybridization limit (in the particle-hole symmetric case considered here) as our initial set of Slater determinants.

2.6.4 Representation of Slater determinants

In order to represent Slater determinants as strings of 0s and 1s, we group together the spin-orbital pairs corresponding to the same orbital but having the opposite spins ↑↓. The leftmost pair in the string is attributed to the impurity and the further pairs correspond to the bath sites with growing energies from left to right. The content of a determinantal string is illustrated as

$$\underbrace{\uparrow\downarrow}_{\text{imp.}} \overbrace{\underbrace{\uparrow\downarrow \ldots \uparrow\downarrow}_{\varepsilon_b<0} \; \underbrace{\uparrow\downarrow}_{\varepsilon_b=0} \; \underbrace{\uparrow\downarrow \ldots \uparrow\downarrow}_{\varepsilon_b>0}}^{\text{bath}} \tag{10}$$

As mentioned above, we take the two degenerate determinants with $S_z = 0$ and the lowest net one-particle energy in the zero-hybridization limit as our initial basis:

$$10\ 11\ldots11\ 01\ 00\ldots00, \tag{11}$$
$$01\ 11\ldots11\ 10\ 00\ldots00. \tag{12}$$

For given N_{bath}, Python strings for these determinants can be build using the function

```
def build_start_dets(N_bath):
    det1 = "01"  + "1" * (N_bath - 1) + "10" + "0" * (N_bath - 1)
    det2 = "10"  + "1" * (N_bath - 1) + "01" + "0" * (N_bath - 1)
    return det1, det2
```

2.6.5 Starting basis object

The impurity onsite interaction strength is in all examples $U = 10$. For the start, we stick to the simplest non-degenerate case of $N_{\text{bath}} = 3$ bath sites and increase later N_{bath} for more advanced demonstrations.

```
U = 10
N_bath = 3
e_bath, V_bath = build_bath(N_bath)
start_dets = build_start_dets(N_bath)
```

We follow the standard procedures described in the previous sections to construct necessary objects of the SOLAX classes. In particular, a `Basis` object containing the two starting Slater determinants (11, 12) is created as

```
basis_start = sx.Basis(start_dets)
print(basis_start)
```

```
01111000
10110100
```

2.6.6 Operator object for Hamiltonian

Here we encode the Hamiltonian in parts represented by Eqs. (5)—(8). Note that for demonstration purposes, we switch the NumPy module to the regime in which up to 3 digits after the decimal point are printed (this does not influence the computational precision).

Impurity term. We represent the impurity onsite interaction operator as

$$\hat{H}_{\text{imp}} = \underbrace{U\,\hat{a}^\dagger_{\text{imp}\uparrow}\,\hat{a}_{\text{imp}\uparrow}\hat{a}^\dagger_{\text{imp}\downarrow}\,\hat{a}_{\text{imp}\downarrow}}_{\hat{H}^{(2)}_{\text{imp}}} - \underbrace{\frac{U}{2}\left(\hat{a}^\dagger_{\text{imp}\uparrow}\,\hat{a}_{\text{imp}\uparrow} + \hat{a}^\dagger_{\text{imp}\downarrow}\,\hat{a}_{\text{imp}\downarrow}\right)}_{\hat{H}^{(1)}_{\text{imp}}} + \frac{U}{4}\,, \tag{13}$$

by expanding Eq. (6) and build up our `Operator` object term by term:

```
H_imp2 = sx.Operator(
    (1, 0, 1, 0),
    np.array([
        [0, 0, 1, 1]
    ]),
    np.array([U])
)

H_imp1 = sx.Operator(
    (1, 0),
    np.array([
        [0, 0],
        [1, 1]
    ]),
    np.array([-U / 2, -U / 2])
)
```

```
H_imp = H_imp2 + H_imp1 + U / 4
print(H_imp)
```

```
Operator({
    (1, 0): OperatorTerm(
        daggers=(1, 0),
        posits=array([[0, 0],
                      [1, 1]]),
        coeffs=array([-5., -5.])
    ),
    (1, 0, 1, 0): OperatorTerm(
        daggers=(1, 0, 1, 0),
        posits=array([[0, 0, 1, 1]]),
        coeffs=array([10.])
    ),
    scalar: 2.5
})
```

Bath term. Taking into account that in each bath site both spin-orbitals have the same energy, the bath Hamiltonian term $\hat{H}_{\text{bath}}$ is encoded as

```
H_bath = sx.Operator(
    (1, 0),
    np.arange(2, 2 * N_bath + 2).repeat(2).reshape(-1, 2),
    e_bath.repeat(2)
)
print(H_bath)
```

```
Operator({
    (1, 0): OperatorTerm(
        daggers=(1, 0),
        posits=array([[2, 2],
                [3, 3],
                [4, 4],
                [5, 5],
                [6, 6],
                [7, 7]]),
        coeffs=array([-1.414e+0, -1.414e+00, -1.225e-16,
                -1.225e-16, 1.414e+00, 1.414e+00])
    )
})
```

Hybridization term. Hybridization as described by the $\hat{H}_{\text{hyb}}$ term takes place between spin-orbitals with the same spin. An `Operator` object for $\hat{H}_{\text{hyb}}$ without h.c. is then built as

```
H_hyb_posits = np.vstack([
    np.array([0, 1] * N_bath),
    np.arange(2, 2 * N_bath + 2)
]).T

H_hyb_nohc = sx.Operator(
    (1, 0),
    H_hyb_posits,
    V_bath.repeat(2)
)
print(H_hyb_nohc)
```

```
Operator({
    (1, 0): OperatorTerm(
        daggers=(1, 0),
        posits=array([[0, 2],
                [1, 3],
                [0, 4],
                [1, 5],
                [0, 6],
                [1, 7]]),
        coeffs=array([1.581, 1.581, 2.236, 2.236, 1.581, 1.581])
    )
})
```

Full Hamiltonian. Finally we obtain an `Operator` object for the full SIAM Hamiltonian (which we don't print here):

```
H = H_imp + H_bath + H_hyb_nohc + H_hyb_nohc.hconj
```

2.6.7 Hamiltonian matrix and state energy

We can now obtain the Hamiltonian matrix on the basis of the 2 "starting" Slater determinants as

```
matrix_start = H.build_matrix(basis_start)
```

For this demonstration example, the obtained `OperatorMatrix` object can be converted to the NumPy dense format and printed:

```
matrix_dense_start = matrix_start.to_scipy().todense()
print(matrix_dense_start)
```

```
[[-5.328  0.   ]
 [ 0.    -5.328]]
```

This matrix is diagonal, and contains directly the state energy corresponding to the Hartree-Fock approximation:

```
energy_start = matrix_dense_start[0, 0]
print(energy_start)
```

```
-5.32842712474619
```

2.6.8 Basis extension

The state energy obtained above is the roughest approximation and must be refined by extending the basis to span a larger subspace of the Hilbert space of the considered many-body system. This can be achieved by generating new determinants via acting with an extension operator on the initial basis. Usually, extension operators are chosen which promote electrons from occupied to unoccupied orbitals via single or double excitation, see e.g. Ref. [45]. In our example, we use the Hamilton operator itself (which in this case includes only single excitations). The advantage of this choice is that the excitation procedure automatically respects the symmetry of the problem. Specifically, extension with the spin-conserving Hamiltonian $\hat{H}_{\mathrm{SIAM}}$ from the two initial determinants with the total spin projection $S_z = 0$, yields only determinants with $S_z = 0$. In this way, we automatically generate only determinants with non-zero contribution to the ground state.

To this end, we extend the basis by acting on `basis_start` with the extension operator (i.e. the Hamiltonian) as

```
basis = H(basis_start)
print(len(basis))
```

```
8
```

The matrix built on this basis is not diagonal anymore:

```
matrix = H.build_matrix(basis)

matrix_dense = matrix.to_scipy().todense()
print(matrix_dense)
```

```
[[-5.328 -1.581 -2.236 -2.236 -1.581  0.     0.     0.   ]
 [-1.581  1.086  0.     0.     0.     0.     0.     0.   ]
 [-2.236  0.    -0.328  0.     0.     2.236  0.     0.   ]
 [-2.236  0.     0.    -0.328  0.     2.236  0.     0.   ]
 [-1.581  0.     0.     0.     1.086  0.     0.     0.   ]
 [ 0.     0.     2.236  2.236  0.    -5.328 -1.581 -1.581]
 [ 0.     0.     0.     0.     0.    -1.581  1.086  0.   ]
 [ 0.     0.     0.     0.     0.    -1.581  0.     1.086]]
```

Therefore, in order to obtain the state energy, the lowest eigenvalue has to be computed using the SciPy means (we use NumPy in this demonstration example):

```
energy = np.linalg.eigvals(matrix_dense).min()

basis_size = len(basis)
print(f"Basis size = {basis_size}\tEnergy = {energy}")
```

```
Basis size = 8  Energy = -8.351171437060554
```

The iterations of basis extension and Hamiltonian matrix evaluation should be now repeated until the state energy converges. We switch now to a more advanced example with larger N_{bath} for demonstration of these iterations.

2.6.9 An example of full computation

We now consider the SIAM with $N_{\text{bath}} = 21$ bath sites and iteratively evaluate the state energy using the described approach. The reconstruction of the `Basis` and `Operator` objects can be performed directly by rerunning the code above after assigning the new value to the `N_bath` variable. We omit this part and show the loop with the iterations directly.

```
import scipy as sp

num_iterations = 4

basis = basis_start

for i in range(num_iterations):
    matrix = H.build_matrix(basis)
    energy = sp.sparse.linalg.eigsh(
        matrix.to_scipy(), k=1, which="SA"
    )[0][0]

    basis_size = len(basis)
    print(
        f"Iteration: {i+1:<8d}"
        f"Basis size = {basis_size:<12d}"
        f"Energy = {energy}"
    )

    if i < num_iterations - 1:
        basis = H(basis)
```

```
Iteration: 1       Basis size = 2           Energy = -28.463653910211487
Iteration: 2       Basis size = 44          Energy = -30.19530217404953
Iteration: 3       Basis size = 684         Energy = -31.242891311317756
Iteration: 4       Basis size = 7084        Energy = -31.70729257122757
```

To find the ground state in each iteration, we used the SciPy diagonalization routine `sp.sparse.linalg.eigsh` for Hermitian sparse matrices. We pass the Hamiltonian matrix and request the first smallest (`k=1`, `which="SA"`) eigenvalue. We note that the basis is not extended in the last iteration since the computation terminates immediately thereafter. It is seen that the energy is converging with the iterations, which should be stopped once the desired precision is achieved.

2.6.10 Optimization of matrix construction

In the iterations above we rebuilt the Hamiltonian matrix each time. Using the `OperatorMatrix` tools demonstrated in the previous section, it is possible to avoid re-evaluation of the matrix elements which have already been calculated in the previous iteration.

To demonstrate this, we start from the `OperatorMatrix` object constructed in the last performed iteration. In Python, variables remain available after the loop is finished. Therefore, we access `basis` and `matrix` directly, and use different variable names corresponding to the analytical notations below:

```
basis_small = basis
M_small = matrix
print(M_small.size)
```

```
(7084, 7084)
```

In the following, we will use the analytical notation M_{small} corresponding to the `OperatorMatrix` object `M_small`. Now we extend the basis another time and build the Hamiltonian matrix $M_{\text{big}}^{\text{direct}}$ directly on the resulting basis (thus computing all matrix elements from scratch as before):

```
basis_big = H(basis_small)
M_big_direct = H.build_matrix(basis_big)
print(M_big_direct.size)
```

```
(58984, 58984)
```

However, the set of Slater determinants `basis_small` is a subset of `basis_big`; this can be checked e.g. as follows:

```
print(len(basis_small % basis_big) == 0)
```

```
True
```

Therefore, all matrix elements of M_{small} enter also $M_{\text{big}}^{\text{direct}}$ allowing to avoid unnecessary re-computations. We implement here a possible scenario of such optimized matrix construction. Note, however, that M_{small} is in general not a rectangular submatrix in $M_{\text{big}}^{\text{direct}}$. Instead, the matrix elements of M_{small} are spread in $M_{\text{big}}^{\text{direct}}$ according to the positions of the Slater determinants from `basis_small` in `basis_big`. In the following we construct a matrix M_{big} which does contain M_{small} as a true submatrix. Though M_{big} and $M_{\text{big}}^{\text{direct}}$ are in general not exactly equal, they are equivalent up to permutation of the basis determinants irrelevant for our applications.

Evaluation of the missing submatrix. The target matrix M_{big} is a Hermitian block matrix

$$M_{\text{big}} = \begin{pmatrix} M_{\text{small}} & A \\ A^{\dagger} & B \end{pmatrix}, \tag{14}$$

with unknown blocks A and B, where B is Hermitian. Given M_{small} is known, we need to additionally evaluate only the block matrix

$$C = \begin{pmatrix} A \\ B \end{pmatrix} \tag{15}$$

in order to construct M_{big}. The rectangular matrix C is built on the following `Basis` objects:

```
basis_cols = basis_big % basis_small
basis_rows = basis_small + basis_cols
```

As mentioned in the section on the `OperatorMatrix` class, this can be achieved by passing both `Basis` objects to the `build_matrix` method:

```
C = H.build_matrix(basis_rows, basis_cols)
print(C.size)
```

```
(58984, 51900)
```

Constructing the matrix from its parts. We use now the methods supported by the `OperatorMatrix` class to build up M_{big} from M_{small} and C. First of all, we bring `C` to its right place in M_{big} by displacing it along the column axis, and obtain the matrix

$$C_{\text{displ}} = \begin{pmatrix} 0 & A \\ 0 & B \end{pmatrix}. \tag{16}$$

```
C_displ = C.displace(0, len(basis_small))
```

Then the following sum corresponds to the block matrix

$$M_{\text{with2B}} = \begin{pmatrix} M_{\text{small}} & A \\ A^\dagger & 2B \end{pmatrix}. \tag{17}$$

```
M_with2B = M_small + C_displ + C_displ.hconj
```

Now we need to subtract B displaced to the proper position:

$$B_{\text{displ}} = \begin{pmatrix} 0 & 0 \\ 0 & B \end{pmatrix}. \tag{18}$$

This matrix can be obtained directly from C_{displ} using the `window` method:

```
left_top = (len(basis_small), len(basis_small))
right_bottom = (None, None)

B_displ = C_displ.window(left_top, right_bottom)
```

Then finally the targeted M_{big} matrix is obtained as

```
M_big = M_with2B - B_displ
```

Comparison of the results. We ensure now the equality of the state energies obtained using the direct and the optimized approaches to the matrix construction:

```
energy_big_direct = sp.sparse.linalg.eigsh(
    M_big_direct.to_scipy(), k=1, which="SA"
)[0][0]
print(energy_big_direct)
```

```
-31.819018639483936
```

```
energy_big = sp.sparse.linalg.eigsh(
    M_big.to_scipy(), k=1, which="SA"
)[0][0]
print(energy_big)
```

```
-31.819018639483833
```

It is evident that the obtained results agree within a very small error, which can be attributed to randomization in the SciPy eigensolver.

In applications involving particularly large Hermitian operators, such as the Hamiltonian in the computations for the N_2 molecule performed in Ref. [19], working directly with full operators is disadvantageous. Instead, it is possible to represent a Hermitian operator $\hat{A}$ as the sum $\hat{A} = \hat{B} + \hat{B}^\dagger$ and use only the part $\hat{B}$ for computations. In particular, this greatly simplifies the construction of the operator matrix. Once an `OperatorMatrix` is built for $\hat{B}$, adding its Hermitian conjugate creates an `OperatorMatrix` for the full operator $\hat{A}$.

3 Neural network support for tackling big basis sets

We have presented the basic functionality of SOLAX as a solver for fermionic quantum systems and now switch to the built-in neural network (NN) support for managing large sets of Slater determinants. While the iterative solution procedure demonstrated for the SIAM in the previous section allows energy refinement to arbitrary precision, the practical implementation of the basis extension approach leads to exponential growth in the basis size, making computations increasingly infeasible. In this section, we demonstrate the NN-based tools available in SOLAX to control basis growth and converge the results with reduced computational resources.

To this end, we follow the algorithm developed in Ref. [16] for managing exponentially growing bases as exemplified for SIAM. We stress that the NN support in SOLAX is not a simple function containing the entire algorithm from Ref. [16]. Instead, we provide modular building blocks that users can easily customize and adapt to their specific research needs. We also demonstrate how these building blocks can be used to reconstruct the algorithm from Ref. [16]. To assist users with little or no prior experience in NNs, the chapter begins with an introduction into the basic NN concepts relevant to understand the code examples.

3.1 Introduction to neural networks

3.1.1 Regression with dense neural networks

Although in the present work a NN is used for solution of the classification task, we start this introduction from considering the regression problem. This path is typically taken in literature since it allows to introduce many important concepts in a clear and intuitive way. On the other hand, as will be discussed below, NN-supported regression can be turned into NN-supported classification by a few modifications.

The regression task consists in approximating a multidimensional function $f : \mathbb{R}^N \to \mathbb{R}^M$ given its values $\tilde{y}_1, \ldots, \tilde{y}_P$ on P points $x_1, \ldots, x_P$. In this dataset, some noise may be present and each $\tilde{y}$ may slightly differ from the corresponding value $f(x)$. We start from NNs of the feedforward dense architecture which are the most general and conceptually simplest approximators for continuous functions. Such a NN consists of L layers $i = 1, \ldots, L$ each representing the most general linear transformation $z^{(i)} = W^{(i)} x^{(i)} + b^{(i)}$ of the input vector $x^{(i)}$ with some matrix $W^{(i)}$ (kernel) and vector $b^{(i)}$ (bias). The output $z^{(i)}$ of each intermediate layer $i < L$ is additionally transformed with a non-linear function $h^{(i)}$ (e.g. the so called rectified linear unit, ReLU) and passed as input to the next layer: $x^{(i+1)} = h^{(i)}\left(z^{(i)}\right)$. In the case of regression, we take $y^{(L)} = z^{(L)}$ for the last layer. With an appropriate selection of kernels and biases for the layers of the NN, the entire network serves as a transformation $y(x)$, mapping the input of the first layer $x = x^{(1)}$ to the output of the last layer $y = y^{(L)}$, thereby approximating the continuous functional relationship $f(x)$ based on the provided dataset.

It is convenient to view the NN layers as consisting of nodes (neurons) each corresponding to one component of the output vector $z^{(i)}$. In Fig. 3(a) we show a schematic illustration of

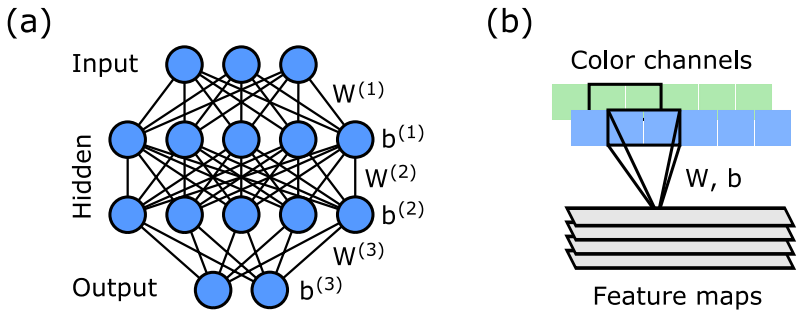

Figure 3: (a) An example of a dense feedforward NN with $L = 3$ kernel-bias pairs, and thus 2 hidden layers. (b) An example of a convolutional layer with $C = 2$ color input channels, kernel length $D = 2$, and $T = 4$ output feature maps.

a dense feedforward NN with $L = 3$. The kernel matrix $W^{(i)}$ is represented as connections between the layers i and $i-1$, whereas the bias vector $b^{(i)}$ is associated with the neurons in each layer. In NNs of the usual dense architecture, each neuron is connected with all neurons of the previous layer. Note that the input $x = x^{(1)}$ to the whole NN is usually called "input layer" which, however, is not associated with any data transformation. The $L-1$ layers enclosed between the input and output layers are referred to as hidden layers.

3.1.2 Neural network training

Training of a NN is a process of iterative improvement of the unknown parameters $W^{(i)}$ and $b^{(i)}$ also referred to as trainable parameters. The NN performance is characterized by a loss function $l(y, \tilde{y})$ which for a given x from the training dataset quantifies the mismatch between the NN-predicted value y and the corresponding "correct" value $\tilde{y}$. Training is usually performed not on individual data entries $(x, \tilde{y})$, but on batches $(x_1, \tilde{y}_1), \ldots, (x_B, \tilde{y}_B)$ of a fixed size B. The loss on a batch is then the average over its entries:

$$L = \frac{1}{B} \sum_{j=1}^{B} l(y_j, \tilde{y}_j) . \tag{19}$$

In the case of regression, the typically used loss function is the square deviation:

$$l_2(y, \tilde{y}) = \frac{1}{2}(y - \tilde{y})^2 . \tag{20}$$

A training iteration on a data batch can be performed using the gradient descent approach as follows. First, the values x_j with $j = 1, \ldots, B$ are sent to the NN which predicts the values y_j. The loss L on the batch is then computed as described above. In order to improve each NN trainable parameter which we denote here generically θ, the gradients of the loss $\frac{\partial L}{\partial \theta}$ are evaluated. Each parameter θ is then improved towards the loss minimization as

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \frac{\partial L}{\partial \theta} , \tag{21}$$

where η is the so called learning rate common for all trainable parameters. Note that η is a hyperparameter, i.e. a parameter fixed by the user and not following from the training procedure. In practice, more advanced gradient-based update rules are often applied in order to stabilize and speed up convergence to the optimal values for the trainable parameters, e.g. the adaptive moment estimation (Adam) algorithm [46] used also in the present work.

One key feature of this algorithm is that it uses adaptive learning rates for each parameter, based on the first moment (mean) and the second moment (variance) of the gradients.

Usually, NN training requires going batch by batch through the whole dataset multiple times (epochs). As convergence is achieved and no further improvement is observed, the training can be stopped. Such "early stopping" does not only speedup the training process, but also helps to avoid overfitting, which occurs when the NN learns the training data "by heart", leading to significantly worse performance on new, unseen data compared to the training data. To prevent overfitting, it is crucial to monitor the loss after each epoch not on the training set, but on a separate validation set held out from the dataset before training.

3.1.3 Neural network as a classifier

In the recent decades, the classification problem has been successfully tackled with NNs, in particular, in the domain of image recognition [47]. In the present work we demonstrate an algorithm which uses a NN classifier in an analogous way to iteratively select the most important Slater determinants. Here we show the modifications which turn the introduced NN regressor into a classifier used in this work.

In the classification problem, the training dataset must contain for each x a discrete value $\tilde{y}$ indicating the correct class for x. We use here the one-hot encoding approach: If there are K distinct classes, the k-th class is encoded by a vector $(0,\dots,1,\dots,0)$ of length K where only the k-th component is non-zero and equal to 1. Each vector $\tilde{y}$ in one-hot encoding can be also interpreted as a set of probabilities for the corresponding x to belong to each class. Since we have the full confidence about the correct class for each x from the dataset, the probability distribution $\tilde{y}$ is degenerate. The corresponding approximative distribution $y(x)$ as predicted by the NN is, however, generally speaking non-degenerate. In this way, the inherently discrete-valued classification problem reduces to a problem with continuous values enabling us to use NNs similarly to the case of regression.

Whereas the hidden layers for a NN classifier can be constructed in the same way as in the case of a NN regressor, the output layer needs modification in order to ensure that the NN prediction $y(x)$ is a valid probability distribution. For the softmax-classifier considered here, the output vector $z = z^{(L)}$ of the last NN layer (see Section 3.1.1) is sent through the softmax nonlinearity performed in two steps. Firstly, the NN output is made positive by element-wise exponentiation:

$$u = \left(e^{z[1]},\dots,e^{z[K]}\right), \tag{22}$$

where $z[k]$ denotes the k-th component of the vector z. Secondly, the resulting vector is normalized ensuring additionally that the NN output sums to 1:

$$y = \frac{u}{\sum_{k=1}^{K} u[k]}. \tag{23}$$

Note that in contrast to e.g. ReLU which acts on the components of the vector argument independently (locally), the softmax operation is non-local due to the normalization step.

The natural measure quantifying the difference between the NN output probability distribution y and the corresponding "correct" degenerate distribution $\tilde{y}$ is cross-entropy

$$s(y,\tilde{y}) = -\sum_{k=1}^{K} \tilde{y}[k]\log y[k], \tag{24}$$

which is always non-negative and becomes zero for the targeted case $y = \tilde{y}$. Training for minimization of the cross-entropy loss is performed in a similar manner as in the case of regression with the square loss. After each training epoch, the classification accuracy (i.e. the fraction of the dataset entries classified by the NN correctly) can be additionally evaluated on a validation set and used for early stopping.

3.1.4 Convolutional neural networks

Among other classification tasks, the image classification problem possesses a special property of translational invariance. For example the type of a vehicle shown in an image does not depend on where exactly it is placed with respect to the image frame. This symmetry can be accounted for directly in the NN architecture by using convolutional layers prior to a dense feedforward classifier. We formulate here at the general level the concepts from convolutional NNs relevant to our work.

Whereas image recognition deals usually with 2D images, we concentrate here on 1D "images" (which in the present work encode Slater determinants). A convolutional layer [see Fig. 3(b)] applies a linear transformation to small "windows" of the input pixels $x[d],\dots,x[d+D-1]$ of a fixed length D and different starting positions d. For each d, the transformation returns a scalar value

$$z[d] = \sum_{j=1}^{D} W[j]\cdot x[j+d-1] + b\,, \tag{25}$$

where the kernel vector W of length D and the bias scalar b are trainable parameters. As d is varied and the window is moved within the image, the transformation results in a converted image typically referred to as a feature map. Note that for $D>1$ the obtained feature map is smaller than the original image. In the image recognition domain, the same size for the feature map and the original image is often enforced by padding the latter with corresponding number of zeros beyond its frame. In the present work, only the actual data are used and no padding is performed.

If a few color channels indexed by $c=1,\dots,C$ are present, a feature map is constructed as

$$z[d] = \sum_{c=1}^{C}\sum_{j=1}^{D} W[c,j]\cdot x[c,j+d-1] + b\,. \tag{26}$$

Usually dozens of feature maps are constructed at the same time as

$$z[t,d] = \sum_{c=1}^{C}\sum_{j=1}^{D} W[t,c,j]\cdot x[c,j+d-1] + b[t]\,, \tag{27}$$

where the new index $t=1,\dots,T$ labels the feature maps. The layer output $y[t,d]$, which is obtained from $z[t,d]$ by additionally applying a non-linear transformation (e.g. ReLU), can serve as input with T "color" channels to the next convolutional layer, or can be flattened an forwarded to a dense NN classifier.

We stress that convolutional NNs actually used for image recognition usually contain further transformations (e.g. pooling and upsampling) and additional variations of the architecture, see e.g. Ref. [27]. We only provided here the necessary information for demonstration of the algorithm from Ref. [16] for managing large sets of Slater determinants.

3.2 Algorithm description

Following this introductory section on machine learning, we will now outline the core concept of the algorithm introduced in Ref. [16], designed to handle large basis sets employing a NN classifier. A schematic illustration of the algorithm is provided in Fig. 4. When a basis extension generates an excessive number of new Slater determinants, not all of these new "candidates" are included in the calculation, but only a fraction of *the most important* ones. The importance of a determinant is quantified by its weight, which corresponds to the magnitude of its expansion coefficient in the quantum state. The fraction α of determinants to be included is a parameter chosen by the user.

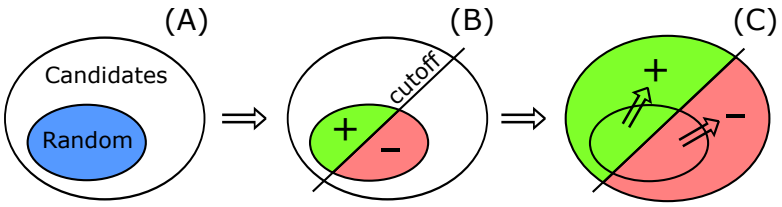

Figure 4: Schematic illustration of the method developed in Ref. [16]. Only the core part of the algorithm is shown. See text for details.

To fulfill the user's request, the following steps are performed. First, as illustrated in panel (A) of Fig. 4, a random selection is drawn from the set of candidate determinants and added to the existing basis used in the computation (the existing basis is not shown in Fig. 4). A diagonalization is then performed, providing the expansion coefficients for the randomly added determinants. Note that the size of the random selection is determined by the user.

Next, as illustrated in panel (B), a cutoff is chosen to divide the random selection into two classes: determinants with weights above the cutoff ("important") and those below it ("unimportant"). Based on the known weights of the randomly selected determinants, the cutoff is automatically adjusted such that the important class comprises a fraction α of the random selection. If the algorithm is well applicable to the case at hand, this cutoff divides also the full set of candidates in a similar proportion (note that the algorithm performance should be investigated for each specific case e.g. as we demonstrate in Section 3.4.4). It remains unknown *which* determinants outside the random selection belong to each class, and this is where a NN is useful.

At this stage, as shown in panel (C), a NN classifier is employed to categorize the remaining determinants into the importance classes. The random selection serves as the training data. For each determinant in the training set, the spin-orbital occupations are used as input features for the NN, while the determinant class serves as the "correct answer". Once trained, the NN is applied to the rest of the candidates to predict their importance class. This process sorts out the entire set of candidates, allowing only the important ones to be retained for diagonalization. We focus here on demonstrating this core procedure. Using the presented SOLAX tools, the algorithm can be further modified (e.g., in Ref. [16], some determinants outside the set of candidates were also used for NN training).

In SOLAX, we implemented a custom machine learning package based on the FLAX library [33] from the JAX ecosystem. This package is primarily intended for development purposes and is not exposed to the user at the general SOLAX interface level. We employed these machine learning tools to provide the necessary functionality for implementing algorithms such as the one described. This functionality is encapsulated in two classes available at the SOLAX interface level: `BasisClassifier` and `BigBasisManager`. The former encapsulates a NN setting, while the latter contains methods that reflect parts of the described algorithm. For demonstration, we continue with the SIAM example computation of Section 2.6.

3.3 BasisClassifier

We now turn to a discussion of the `BasisClassifier` class, which integrates a NN and an optimizer to facilitate necessary machine learning operations. The user is expected to create a NN architecture using FLAX building blocks, while an optimizer is selected from the Optax package available alongside JAX. First, we make the necessary imports:

```
from flax import linen as nn
import optax
```

Here, the `flax.linen` API is imported via the conventional name `nn`. A NN architecture can be now defined using the standard FLAX approach, i.e. via writing a Python function describing how a single data entry propagates towards the classification output. In this function, the FLAX building blocks as well as general JAX transformations can be used. We implement here the convolutional architecture employed in the work [16], which was at the time implemented based on the TensorFlow library [17].

```
def nn_call_on_bits(x):
    x = x.reshape(-1, 2)
    x = nn.Conv(features=64, kernel_size=(2,), padding="valid")(x)
    x = nn.relu(x)
    x = nn.Conv(features=4, kernel_size=(1,), padding="valid")(x)
    x = nn.relu(x)
    x = x.reshape(-1)

    x = nn.Dense(features=dense_size)(x)
    x = nn.relu(x)
    x = nn.Dense(features=dense_size//2)(x)
    x = nn.relu(x)
    x = nn.Dense(features=dense_size//4)(x)
    x = nn.relu(x)
    x = nn.Dense(features=2)(x)
    return x
```

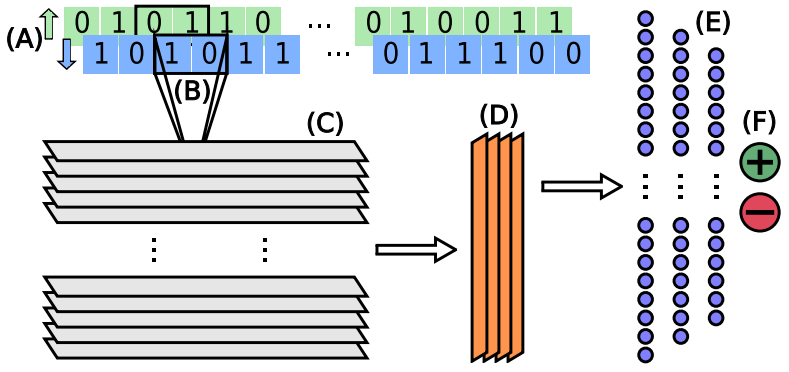

Figure 5: Architecture of the convolutional NN as implemented in the function `nn_call_on_bits`. A candidate Slater determinant is encoded as spin-orbital populations distributed into two spin-channels (A). This input is processed by a convolutional kernel (B) of size 2 into 64 feature maps (C), and then by a convolutional kernel of size 1 (now shown in the plot) into 4 feature maps (D). Both convolutional layers have ReLU activation. The output is flattened and sent to the dense block (E)—(F) which ends with two classification logits (F) classifying the determinant as important/unimportant. Note that `nn_call_on_bits` does not contain the softmax activation, since SOLAX automatically applies softmax to the output logits (F).

We depict this function schematically in Fig. 5 labeling the relevant parts using capital letters (A)—(F). The input `x` is an array of bits (0 and 1) representing the spin-orbital populations of the incoming Slater determinant. The input data are distributed into the spin-up (↑) and spin-down (↓) channels (A) using `reshape`. This representation is processed by a convolutional kernel (B) of size 2 into 64 feature maps (C), and then by a convolutional kernel of size 1 (now shown in the plot) into 4 feature maps (D). Both convolutional layers have ReLU activation. The argument `padding="valid"` indicates that there is no additional padding with zeros, see Section 3.1.4. The output of the convolutional block (A)—(D) is flattened using `reshape(-1)` and passed to the dense block (E)—(F), which ends with two neurons (F) containing the classification logits. Note that in the code calling this function, the output logits are automatically passed through the softmax activation function to convert them into probabilities summing to 1. Hence, there is no need to explicitly apply the softmax function within the NN definition.

In the function body above, the size of the layers in the dense block is determined by the global variable `dense_size` initialized as follows:

```
dense_size = int(7 * np.sqrt(2 * N_bath + 2))
print(dense_size)
```

```
46
```

Here, we provided the generic expression for arbitrary N_{bath} used in Ref. [16]. Empirically, this turned out to be a good choice in the case of SIAM. Also for other models it could serve as a reasonable starting point for further search and fine-tuning. An object of `BasisClassifier` is now instantiated as

```
classifier = sx.BasisClassifier(nn_call_on_bits)
```

We note that so far no NN has been actually initialized in memory. The latter needs to be performed explicitly using the `initialize` method which requires the following additional arguments: (1) an input `Basis` prototype, (2) an optimizer, and (3) a JAX random key.

The input `Basis` prototype required by the `BasisClassifier` `initialize` method is needed to find out the size of the input `x` of the function defining the NN. This determines also the concrete structure of the NN as created in memory. We use here the variable `basis_start` from the SIAM example in Section 2.6. We choose the standard Adam optimizer [46] which can be created using Optax as

```
optimizer = optax.adam(learning_rate=0.005)
```

3.3.1 RandomKeys

In JAX, randomization is performed in a deterministic and reproducible manner. Each function generating random numbers receives a JAX random key which determines the randomization. These keys can be created from each other, whereas the very first one is created from an integer seed. We wrapped this mechanism in a convenience class `RandomKeys` which implements the Python iterator interface. A `RandomKeys` instance is created as

```
rand_keys = sx.RandomKeys(seed=1234)
```

where `seed` is a keyword-only argument. Now, each time when a new JAX random key needs to be generated, the Python `next` keyword can be used on `rand_keys`:

```
key_for_nn = next(rand_keys)
```

The obtained key is passed to the `initialize` method in order to randomly initialize the NN weights.

Using the needed components, we initialize the NN in memory as

```
classifier.initialize(key_for_nn, basis_start, optimizer)
```

Note that once initialized, the NN summary can be printed out using the `print_summary` method (we skip this step here due to the output size). This exhausts the knowledge about the `BasisClassifier` class needed to proceed with implementation of the sketched algorithm using `BigBasisManager`.

3.4 BigBasisManager

The central SOLAX tool for implementing algorithms as the one described above, is the `BigBasisManager` class. Its objects are created from:

- a set of candidate Slater determinants represented as an object of the SOLAX class `Basis`;
- a NN setting represented as an object of the just shown SOLAX class `BasisClassifier`.

In order to demonstrate the functions provided with `BigBasisManager`, we turn back to the SIAM example with $N_{\text{bath}} = 21$ bath sites considered in Section 2.6.9. In the iterative solution shown there, we achieved the basis size of 7084 Slater determinants. Another extension yields

```
basis_small = basis

basis_big = H(basis_small)
print(len(basis_big))
```

```
58984
```

Here we stick to the notations adopted also in Section 2.6.10. The obtained basis poses no computational challenge and was tackled directly in Section 2.6.10. Still, for demonstration, we sort it out using the NN-supported algorithm and compare the resulting state energy with the one obtained directly. The set of new "candidates" generated at the last extension step is obtained as

```
candidates = basis_big % basis_small
print(len(candidates))
```

```
51900
```

Using also the `classifier` object built above, we create now a `BigBasisManager` instance:

```
bbm = sx.BigBasisManager(candidates, classifier)
```

Instead of including all 51900 candidates in the computation, we follow Ref. [16] and target inclusion of the following number of *the most important* determinants:

```
target_num = int(np.sqrt(len(basis_big)) * 50)
print(target_num)
```

```
12143
```

We provided here the empirical generic expression used in Ref. [16] for SIAM, which for other systems can be used as a reasonable starting point for further search. In the following we implement the sketched NN-based approach with the help of the created `bbm` object.

3.4.1 Random selection

For the random selection size, we follow again the empirical choice from Ref. [16] for SIAM, and use the following expression for the size of the random selection shown in Fig. 4(A):

```
random_num = int(target_num / 1.5)
print(random_num)
```

```
8095
```

The selection can be drawn from `candidates` using the `bbm` object via the `sample_subbasis` method:

```
random_sel = bbm.sample_subbasis(next(rand_keys), random_num)
```

Here we generated another JAX random key from the `rand_keys` object and passed it directly to the `sample_subbasis` function. The returned `random_sel` object is of type `Basis`. In order to obtain the weights for the picked determinants as required by the algorithm, we add the random selection on top of the old basis:

```
basis_diag = basis_small + random_sel
print(len(basis_diag))
```

```
15179
```

and perform diagonalization as shown in Section 2.6.9:

```
matrix = H.build_matrix(basis_diag)
result = sp.sparse.linalg.eigsh(matrix.to_scipy(), k=1, which="SA")

energy = result[0][0]
print(f"Intermediate energy:\t{energy}")
```

```
Intermediate energy: -31.720920015599123
```

Finally, we use the obtained eigenvector to create a `State`:

```
eigenvec = result[1][:, 0]
state_diag = sx.State(basis_diag, eigenvec)
```

from which we strip `basis_small` obtaining the randomly selected determinants together with their coefficients encapsulated in a `State` object:

```
state_train = state_diag % basis_small
```

Note that the latter object serves only for convenient storing the random selection coefficient and does not actually represent the searched quantum state. The variable name `state_train` reflects that this `State` instance contains the data which will be used for the NN training.

As expected, the intermediate energy obtained above lies between the energy on `basis_small` computed in Section 2.6.9 and the energy on `basis_big` from Section 2.6.10. The corresponding separations are approx. 0.0136 and 0.0981. In this way, inclusion of the random selection in the computation does not considerably promote the energy towards the value on the larger basis. In contrast, as we will see in the following sections, inclusion of NN-selected determinants pushes the energy close to the value on `basis_big`.

3.4.2 Deriving the cutoff

We switch now to splitting the random selection with a cutoff as discussed in Section 3.2 and illustrated in Fig. 4(B). The cutoff can be obtained using the `BigBasisManager` method `derive_abs_coeff_cut`:

```
abs_coeff_cut = bbm.derive_abs_coeff_cut(target_num, state_train)
print(f"Cutoff:\t{abs_coeff_cut}")
```

```
Cutoff: 0.00020762079204639022
```

This cutoff splits the random selection into two importance classes of determinants with larger and smaller weights. We point out again that under weight we understand here the absolute value of the determinant coefficient (not its square). The “important” class as represented by a `State` object is obtained by chopping:

```
state_train_impt = state_train.chop(abs_coeff_cut)
print(len(state_train_impt))
```

```
1893
```

The cutoff is automatically chosen such that the fraction of the important class in the random selection

```
print(len(state_train_impt) / len(state_train))
```

```
0.23384805435453984
```

equals the ratio of the targeted number of the most important determinants to be included to the full number of the candidates:

```
print(target_num / len(candidates))
```

```
0.23396917148362234
```

3.4.3 Using the neural network

In the following, we employ the NN in order to distribute the determinants outside the random selection into the importance classes as illustrated in Fig. 4(C). The NN is already created, initialized and incorporated in the `bbm` object of the `BigBasisManager` class. The latter provides high-level functions for the NN usage in the considered context. We train the NN similarly to Ref. [16] as follows:

```
early_stopped = bbm.train_classifier(
    next(rand_keys),
    state_train,
    abs_coeff_cut,
    batch_size=256,
    epochs=200,
    early_stop=True,
    early_stop_params={"patience": 3}
)
```

```
Started: accuracy=2.472703e-01
Epoch 0: accuracy=8.015612e-01
Epoch 1: accuracy=8.385798e-01
Epoch 2: accuracy=8.525778e-01
Epoch 3: accuracy=8.595095e-01
Epoch 4: accuracy=8.705761e-01
Epoch 5: accuracy=8.619702e-01
Epoch 6: accuracy=9.055037e-01
Epoch 7: accuracy=8.985720e-01
Epoch 8: accuracy=9.237778e-01
Epoch 9: accuracy=9.326122e-01
Epoch 10: accuracy=9.412651e-01
Epoch 11: accuracy=9.312204e-01
Epoch 12: accuracy=9.491246e-01
Epoch 13: accuracy=9.461933e-01
Epoch 14: accuracy=9.596332e-01
Epoch 15: accuracy=9.524729e-01
Epoch 16: accuracy=9.450301e-01
Epoch 17: accuracy=9.506576e-01
Epoch 18: accuracy=9.534478e-01
```

Here, a new JAX random key is generated from `rand_keys` and passed directly to the `train_classifier` call. It is needed for reshuffling to avoid any ordering bias in the training data. The latter are provided as the `state_train` object of the `State` class. Whereas `state_train` contains only the Slater determinants with their coefficients, the

`train_classifier` method takes care of converting this to classification data based on the cutoff `abs_coeff_cut` also provided to the method.

The rest of the parameters are keyword-only arguments controlling the NN training, which is performed in batches of size 256 until the best performance is achieved and then early-stopped. After the epoch in which the NN failed to improve, we give it a chance to try 3 more times by indicating `patience=3`. The early stopping parameters passed as the `early_stop_params` dictionary are forwarded directly to the underlying `EarlyStopping` FLAX class and can be looked up in the FLAX documentation [33]. Note that the NN is reset to its best state achieved in the training process. The output value `early_stopped` is `True` if the training was early-stopped, and `False` if it went through all 200 epochs (provided with the `epochs` argument) without reaching the best performance.

The printed NN training information contains the NN classification accuracy evaluated before the training and after each epoch on a data part held out from the training set. The fraction of the data used for the performance evaluation can be controlled via the keyword-only argument `val_frac` of the `train_classifier` method (the default value is `val_frac=0.2`). We note that if an NVIDIA GPU is available and a GPU-capable version of JAX is installed on the machine, it will be automatically used for the NN training usually leading to a significant speedup.

The trained NN can be now used for classification of all candidates:

```
nn_selected = bbm.predict_impt_subbasis(batch_size=256)
nn_selected = nn_selected % state_train.basis
print(len(nn_selected))
```

```
9834
```

The method `predict_impt_subbasis` returns a `Basis` object `nn_selected` containing the Slater determinants classified by the NN as important. Note that the NN is applied here to the full set of candidates including the training subset present in the `State` object `state_train`. Therefore, we strip the latter from `nn_selected`. The NN prediction operation runs automatically on a GPU if available. In this case, the keyword-only `batch_size` argument can be passed to the `predict_impt_subbasis` method for controlling the GPU memory usage.

The final subset of candidates to be included in the computation as the result of the NN-supported procedure is build as

```
basis_impt = nn_selected + state_train_impt.basis
print(len(basis_impt))
print(abs(len(basis_impt) - target_num) / target_num)
```

```
11727
0.034258420489170716
```

Here we printed the size of the obtained important subset and its relative deviation from the targeted size `target_num`. We see that the user's request has been satisfied in this demonstration example.

3.4.4 Checking and processing of the results

We construct now the full basis and evaluate the state energy:

```
basis = basis_small + basis_impt

matrix = H.build_matrix(basis)
result = sp.sparse.linalg.eigsh(matrix.to_scipy(), k=1, which="SA")

energy = result[0][0]
print(f"Basis:\t{len(basis)}")
print(f"Energy:\t{energy}")
```

```
Basis:  18811
Energy:  -31.817043901573747
```

The obtained energy is separated from the energy computed on `basis_small` in Section 2.6.9 and the energy on `basis_big` from Section 2.6.10 by 0.1100 and 0.0017, respectively. In this way, by using the NN support provided in SOLAX we were able to almost reach the same state energy on 19462 Slater determinants instead of 58984.

We use now the obtained eigenvector to check how many determinants out of those suggested by the NN indeed possess weights higher that `abs_coeff_cut`. First we construct a `State` object corresponding to the NN suggestion:

```
eigenvec = result[1][:, 0]
state = sx.State(basis, eigenvec)

nn_selected_state = state % basis_small % state_train.basis
print(nn_selected_state.basis == nn_selected)
```

```
True
```

The `State` instance `nn_selected_state` contains a basis set equal to `nn_selected` and additionally the evaluated coefficients. Now we chop off the misclassified determinants:

```
nn_selected_right = nn_selected_state.chop(abs_coeff_cut).basis
print(len(nn_selected_right))
print(len(nn_selected_right) / len(nn_selected))
```

```
8648
0.8793980069147854
```

The printed fraction of the Slater determinants classified correctly is smaller that the NN accuracy achieved in the training procedure. We attribute this to the drift of the determinant coefficients upon the inclusion in the computation of other determinants. We stress again that the NN performance should be investigated individually for each application case e.g. following the outlined procedure.

If the obtained basis is involved in further computations (e.g. as the starting point of the next NN-supported iteration), it is advantageous to exclude from it the misclassified determinants:

```
nn_selected_wrong = nn_selected % nn_selected_right
basis_final = basis % nn_selected_wrong
print(len(basis_final))
```

```
17625
```

If needed, the energy and the coefficients can be evaluated on `basis_final` via the corresponding Hamiltonian matrix. Note, however, that it is inefficient to compute the latter from scratch using the `H.build_matrix` call. Instead, as demonstrated in the section on the `OperatorMatrix` class, the `shrink_basis` method of the `matrix` object can be used to extract the matrix on `basis_final` directly from the `OperatorMatrix` instance `matrix` built on `basis`. This shortcut is possible since `basis_final` represents a subset of `basis`, and therefore all the needed matrix elements have been already evaluated.

We point out that each `BigBasisManager` instance is bound to a particular `Basis` object with candidates to be sorted out. Therefore, it is necessary to create a new `BigBasisManager` object for each new big basis to be optimized (e.g. if further NN-supported iterations follow). At the same time, the same `BasisClassifier` object can be reused as a component in different `BigBasisManager` instances transferring in this way the NN experience from case to case.

To summarize, using the NN support tools provided in SOLAX, we could reach the same accuracy level for the SIAM ground state energy on a much smaller basis. Further iterations of the described algorithm would profit strongly from the reduced basis size as the starting point. In Ref. [16], this algorithm was implemented using the TensorFlow library [17] and the Quanty code [18] for working with fermionic quantum systems. It was applied to SIAM with up to $N_{\text{bath}} = 299$ bath sites to sort out many millions of Slater determinants and helped to reduce the necessary basis set sizes by orders of magnitude. As implemented in SOLAX, this approach was applied for the first time in Ref. [19] for computations of the ground state of the N_2 molecule. Note that whereas we prefer NNs due to their scalability, flexibility and availability of powerful frameworks like JAX and TensorFlow, also other classifier types could be potentially employed in the described algorithm, as demonstrated in Refs [48, 49] for similar computations. The limitation of the presented method can be considered in a two-fold way. From the conceptual perspective, its applicability to a particular problem is itself a research question, which can be addressed with the SOLAX tools we provide. From the technical perspective, i.e. in the context of the needed computational resources, it is important to address the computational time needed by different parts of the algorithm. In the following section we provide such benchmarks for larger NN-supported SIAM computations with GPU acceleration and purely on CPU.

3.4.5 Computation time benchmarks

In this section, we show comparisons of computation times spent in different parts of the described algorithm. These comparisons cannot be performed in a representative way for such small examples as presented above. Therefore, we turn here to more extensive computations for the SIAM with 149 bath sites and perform five NN-supported iterations. One iteration of the NN-supported algorithm can be divided into five main components listed below. Note that we follow here exactly Ref. [16] where a slightly modified algorithm was employed. Whereas the first NN-supported iteration is identical with the algorithm version described above, in further iterations modifications are included which are specified below for the individual algorithm components.

A) **Basis extension:** This step involves applying an extension operator to `basis_small` to generate new candidate Slater determinants. Starting from the second iteration, we form the new set of candidates not by directly acting with the extension operator on the set of determinants currently present in the computation, but by acting only on the determinants included in the last iteration, and combining the result with the set of candidates from the last iteration. Whereas the obtained set is the same, the latter procedure is less costly, see Ref. [16].

B) **Training data generation:** This process selects `random_num` determinants randomly, performs a partial diagonalization of the Hamiltonian represented on `basis_small + random_sel`, and computes the cutoff coefficient to label the determinants in `random_sel`. Beyond the first iteration, the Slater determinants included in the previous NN-supported iterations are additionally used for the NN training, see Ref. [16].

C) **Neural network training:** The NN classifier is trained using the generated training data.

D) **Neural network prediction:** The trained NN predicts which of the candidate determinants are important, resulting in the basis set `nn_selected`.

E) **Selection correction:** To complete the iteration, a second partial diagonalization is performed on `basis_small + nn_selected` along with the training determinants labeled as important.

In Fig. 6, we present the time spent in these five algorithm parts with single-GPU acceleration and purely on CPU. It is seen that the strongest GPU speedup is achieved in parts C) and D), i.e. the NN training and prediction. The full computation on a cluster with a GPU took 0.7 hours, and on a CPU cluster 1.7 hours. The same computation performed in Ref. [16] on a CPU cluster took about 35 hours dominated by the inefficient communication between the Quanty CI code and the Python code with a NN.

We note that for more complex Hamiltonians, e.g. for atoms and molecules including inter-orbital interactions, the ratio between the execution times for the algorithm parts can further shift towards the non-NN parts. For example, in computations performed for the N_2 molecule in Ref. [19], we observed that steps B) and E) involving the construction of an `OperatorMatrix` were significantly more expensive than the rest of the algorithm. At the same time, however, the observed GPU acceleration was here more pronounced than for the considered SIAM example.

4 Saving/loading SOLAX objects and reproducing computations

In this section, we focus on the crucial functionality of saving and loading objects of the SOLAX classes. This feature enables checkpointing computations and restoring them in a flexible and efficient way. SOLAX provides a convenient and unified mechanism for this purpose through the global functions `save` and `load`, which are compatible with objects of the `Basis`, `State`, `OperatorTerm`, `Operator`, `OperatorMatrix`, and `RandomKeys` classes. It is important to note that `RandomKeys` objects are saved in their current iteration state, rather than JAX random keys generated by them. The `BasisClassifier` class employs its own approach, which is based on the Orbax library [34] from the JAX ecosystem.

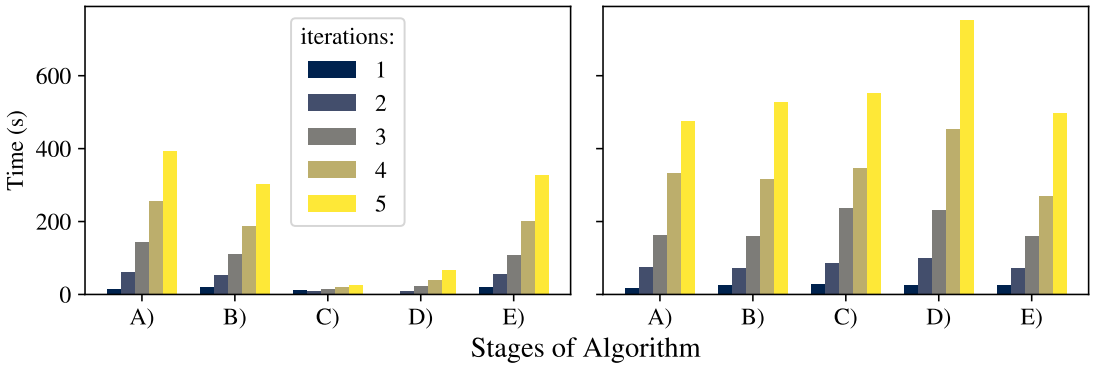

Figure 6: Comparison of time spent on different parts of our algorithm for SIAM with $N_{\text{bath}} = 149$ as performed with single-GPU acceleration (left panel) and purely on CPU (right panel). The categories A)—E) are explained in the text.

4.1 Standard mechanism

We start from the standard mechanism based on the `save` and `load` functions. Using this approach, it is possible to save and load objects of the SOLAX classes listed above. For instance, we can save the `basis_final` object containing the basis set of Slater determinants built using the NN-supported algorithm in the previous sections:

```
sx.save(basis_final, "solax_basis_")
```

and load it again as

```
basis_loaded = sx.load("solax_basis_")
print(basis_loaded == basis_final)
```

```
True
```

Here we ensured that the loaded `Basis` object is indeed equal to the saved one. The string `"solax_basis_"` indicates the name of the directory we save to and load from.

Important note! The directory is created prior to saving. If the directory already exists, it will be first erased together with its current content. Note that this applies also to the `BasisClassifier` class which has its own saving/loading mechanism (see below).

Instead of saving and loading standalone SOLAX objects, it is possible to first bundle many objects in one Python dictionary. The user provides here a string label to each object as the key in a key-value pair, whereas the object itself is the value. For instance, for the objects `basis_final` and `H` from the previous section we use the string labels `"basis_from_nn"` and `"hamiltonian"`, respectively:

```
dict_to_save = dict(
    basis_from_nn=basis_final,
    hamiltonian=H
)

sx.save(dict_to_save, "solax_basis_ham_")
```

```
loaded_dict = sx.load("solax_basis_ham_")

for key, value in loaded_dict.items():
    print(f"{key} has type {type(value).__name__}")
```

```
basis_from_nn has type Basis
hamiltonian has type Operator
```

We printed here only the types of the loaded objects. The used key strings are employed in internal addressing of the saved objects and must satisfy the following rule: A key string is valid if it *could be* used as a Python variable name. Therefore, we recommend to create dictionaries for saving using the `dict` constructor rather than the literal `{...}`. Then, the formulated rule is satisfied automatically, since the key strings actually *are* used as variable names.

It is possible to nest such dictionaries as above, and add variables of many standard Python types or NumPy arrays. This allows to perform a unified saving in a comprehensive way. For example, the following dictionary can be saved using directly the `save` function:

```
dict_to_save = dict(
    info="This computation is a demonstration of SOLAX",
    params=dict(N_bath=N_bath, U_impurity=U),
    basis_from_nn=basis_final,
    last_epochs=dict(
        epochs=np.array([14, 15, 16, 17, 18]),
        accuracies=np.array([9.596332e-01, 9.524729e-01,
                9.450301e-01, 9.506576e-01, 9.534478e-01])
    ),
    random_keys_after=rand_keys
)

sx.save(dict_to_save, "solax_big_save_")
```

At the technical level, the `save` and `load` functions process SOLAX objects in the following way:

- the standard Python types are converted to the JSON format [50] which is saved and loaded using the standard Python `json` module;
- the underlying NumPy arrays are saved and loaded using the NumPy means (the `RandomKeys` class needs additionally conversion of JAX arrays to and from NumPy).

We deliberately refrained from using the well known `pickle` module from the Python standard library, since it has been shown to have safety flaws [51].

4.2 Saving/loading BasisClassifier objects

The `BasisClassifier` class does not follow the standard saving/loading mechanism described above. Instead, it implements its own methods `save_state` and `load_state` based on the Orbax library [34]. Saving is performed in a straightforward way:

```
classifier.save_state("solax_nn_")
```

Important note! As in the case of the global `save` function, the shown method creates the directory before the `BasisClassifier` object is saved there. If the directory already exists, it will be first erased together with its current content.

In order to reconstruct the saved `BasisClassifier` object, the user needs first to create and initialize a new `BasisClassifier` instance following the procedure described in Section 3.3:

```
loaded_nn = sx.BasisClassifier(nn_call_on_bits)

fake_key = sx.RandomKeys.fake_key()
loaded_nn.initialize(fake_key, basis_start, optimizer)
```

Here we reused the `nn_call_on_bits` function defining the NN architecture, `basis_start` as a prototype `Basis` instance, and the `optimizer` object. Instead of using a properly constructed JAX random key, we obtained a "fake" key directly from the `RandomKeys` class (i.e. without creating an instance and making an iteration step). Such fake keys should not be applied in true randomization, but can be used e.g. to initialize a NN whose neuron weights and biases will be anyway overwritten by loading. The saved NN state can be now restored as

```
loaded_nn.load_state("solax_nn_")
```

The presented tools are complete to provide the user with the possibility to conveniently and efficiently save and load SOLAX computations.

4.3 A note on randomization under GPU acceleration

The JAX library and its derivatives like FLAX used here for the NN implementation, treat randomness in a deterministic way based on random keys. However, in GPU-accelerated computations some non-deterministic effects have been observed, see e.g. the discussion [52]. In our computations, we observed deviations of the NN training accuracies when rerunning the script from scratch. This could be avoided by adding the following lines immediately after the imports:

```
import os
os.environ['XLA_FLAGS']='--xla_gpu_deterministic_ops=true'
```

In subsequent JAX versions a different approach might be needed or the problem might be completely resolved.

5 Conclusions and outlook

In this work, we have demonstrated the foundational capabilities of the SOLAX library for tackling complex fermionic many-body quantum systems. The core components of SOLAX implemented as the `Basis`, `State`, `OperatorTerm`, `Operator` and `OperatorMatrix` classes, provide a robust framework for encoding and manipulating bases of Slater determinants, constructing quantum states, and handling operators within the second quantization formalism. Through a detailed application to the Single Impurity Anderson Model (SIAM), we illustrated how the iterative extension of the basis set combined with diagonalization can be efficiently implemented using SOLAX.

When the basis set grows too large to handle using available computational resources, the SOLAX built-in neural network (NN) support offers a practical solution. This approach allows for an efficient approximation of quantum states by identifying and selecting the most significant Slater determinants, thereby reducing the computational burden. The presented SIAM results, together with the SOLAX-based computations for the paradigmatic N_2 molecule performed in Ref. [19], underscore the flexibility and power of SOLAX in dealing with large basis sets. The modular integration of state-of-the-art machine learning techniques into the SOLAX framework opens up new avenues for addressing larger and more complex quantum systems that were previously beyond reach with alternative methods.

Looking forward, the development of additional toolboxes within the SOLAX package will be closely aligned with advancing research projects in quantum many-body physics and quantum chemistry. This requires enabling seamless interfacing between SOLAX and existing computational codes for, e.g., density functional theory (DFT) and Hartree-Fock computations. For instance, in our N_2 study [19], SOLAX was successfully interfaced with the Python-based GPAW [53] code, which demonstrates the potential for straightforward integration with other Python-compatible packages in future research.

Furthermore, a planned enhancement to SOLAX is the inclusion of a comprehensive library of predefined operators to facilitate the construction of effective Hamiltonians. Many quantum many-body problems require models built from standard elements, such as hopping and approximative interaction operators. The former will be defined based on lattice geometry, hopping range, and the symmetry of the involved orbital degrees of freedom, while the (typically onsite) interaction operator will be parametrized using physical quantities like the Hubbard U and Hund's coupling J. By incorporating such fundamental building blocks into an operator library, SOLAX aims to simplify and standardize Hamiltonian construction for a wide range of applications in condensed matter, quantum chemistry, and even for pedagogical purposes in lectures on these advanced topics.

A major feature which is currently under development is the computation of spectral functions. Spectral functions are not only essential for studying the excitation spectra of cluster models, but will also enable SOLAX to serve as an impurity solver within the framework of dynamical mean-field theory (DMFT) [38–40]. Conceptually closely related to DMFT applications, we plan to implement predefined functionalities tailored for so called *embedding schemes*. These schemes aim to partition the full problem into a subset of orbitals which will be treated with full many-body rigor, while the rest is maintained at a static mean-field level.

All of the proposed features will be implemented in a modular way within Python, ensuring flexibility and ease of use. Through these developments, we hope to create a vibrant user/developer community which brings together expertise from a wide range of fields and reflecting the wide spectrum of potential applications for SOLAX.

Acknowledgments

PB thanks Fred Baptiste for his valuable Python lessons. We gratefully acknowledge the group of Hannes Jónsson at the University of Iceland for their collaboration on the N_2 molecule [19] which was the first application of SOLAX. We further thank Henri Menke, Paul Fadler and Max Kroesbergen for useful discussions.

Funding information The authors gratefully acknowledge the scientific support and HPC resources provided by the Erlangen National High Performance Computing Center (NHR@FAU) of the Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU). PB gratefully acknowledges the ARTEMIS funding via the QuantERA program of the European Union.

References

[1] C. F. Fischer, M. Godefroid, T. Brage, P. Jönsson and G. Gaigalas, *Advanced multiconfiguration methods for complex atoms: I. Energies and wave functions*, J. Phys. B: At. Mol. Opt. Phys. **49**, 182004 (2016), doi:10.1088/0953-4075/49/18/182004.

[2] M. G. Kozlov, M. S. Safronova, J. R. C. López-Urrutia and P. O. Schmidt, *Highly charged ions: Optical clocks and applications in fundamental physics*, Rev. Mod. Phys. **90**, 045005 (2018), doi:10.1103/RevModPhys.90.045005.

[3] E. Rossi, G. L. Bendazzoli, S. Evangelisti and D. Maynau, *A full-configuration benchmark for the N2 molecule*, Chem. Phys. Lett. **310**, 530 (1999), doi:10.1016/S0009-2614(99)00791-5.

[4] B. Huron, J. P. Malrieu and P. Rancurel, *Iterative perturbation calculations of ground and excited state energies from multiconfigurational zeroth-order wavefunctions*, J. Chem. Phys. **58**, 5745 (1973), doi:10.1063/1.1679199.

[5] J. C. Greer, *Monte Carlo configuration interaction*, J. Comput. Phys. **146**, 181 (1998), doi:10.1006/jcph.1998.5953.

[6] Y. Garniron, A. Scemama, E. Giner, M. Caffarel and P.-F. Loos, *Selected configuration interaction dressed by perturbation*, J. Chem. Phys. **149**, 064103 (2018), doi:10.1063/1.5044503.

[7] N. M. Tubman, C. D. Freeman, D. S. Levine, D. Hait, M. Head-Gordon and K. B. Whaley, *Modern approaches to exact diagonalization and selected configuration interaction with the adaptive sampling CI method*, J. Chem. Theory Comput. **16**, 2139 (2020), doi:10.1021/acs.jctc.8b00536.

[8] T. Schäfer et al., *Tracking the footprints of spin fluctuations: A multimethod, multimessenger study of the two-dimensional Hubbard model*, Phys. Rev. X **11**, 011058 (2021), doi:10.1103/PhysRevX.11.011058.

[9] P. A. Lee, N. Nagaosa and X.-G. Wen, *Doping a Mott insulator: Physics of high-temperature superconductivity*, Rev. Mod. Phys. **78**, 17 (2006), doi:10.1103/RevModPhys.78.17.

[10] B. Keimer, S. A. Kivelson, M. R. Norman, S. Uchida and J. Zaanen, *From quantum matter to high-temperature superconductivity in copper oxides*, Nature **518**, 179 (2015), doi:10.1038/nature14165.

[11] D. C. Johnston, *The puzzle of high temperature superconductivity in layered iron pnictides and chalcogenides*, Adv. Phys. **59**, 803 (2010), doi:10.1080/00018732.2010.513480.

[12] D. Li, K. Lee, B. Y. Wang, M. Osada, S. Crossley, H. R. Lee, Y. Cui, Y. Hikita and H. Y. Hwang, *Superconductivity in an infinite-layer nickelate*, Nature **572**, 624 (2019), doi:10.1038/s41586-019-1496-5.

[13] M. Hepting et al., *Electronic structure of the parent compound of superconducting infinite-layer nickelates*, Nat. Mater. **19**, 381 (2020), doi:10.1038/s41563-019-0585-z.

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

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

[16] P. Bilous, L. Thirion, H. Menke, M. W. Haverkort, A. Pálffy and P. Hansmann, *Neural-network-supported basis optimizer for the configuration interaction problem in quantum many-body clusters: Feasibility study and numerical proof*, Phys. Rev. B **111**, 035124 (2024), doi:10.1103/PhysRevB.111.035124.

[17] M. Abadi et al., *TensorFlow: Large-scale machine learning on heterogeneous systems* (2015), https://www.tensorflow.org/.

[18] Y. Lu, M. Höppner, O. Gunnarsson and M. W. Haverkort, *Efficient real-frequency solver for dynamical mean-field theory*, Phys. Rev. B **90**, 085102 (2014), doi:10.1103/PhysRevB.90.085102.

[19] Y. L. A. Schmerwitz, L. Thirion, G. Levi, E. Ö. Jónsson, P. Bilous, H. Jónsson and P. Hansmann, *Revisiting N_2 with neural-network-supported CI*, (arXiv preprint) doi:10.48550/arXiv.2406.08154.

[20] Y. Garniron et al., *Quantum package 2.0: An open-source determinant-driven suite of programs*, J. Chem. Theory Comput. **15**, 3591 (2019), doi:10.1021/acs.jctc.9b00176.

[21] J. Bradbury et al., *JAX: Composable transformations of Python+NumPy programs* (2018), http://github.com/jax-ml/jax.

[22] C. R. Harris et al., *Array programming with NumPy*, Nature **585**, 357 (2020), doi:10.1038/s41586-020-2649-2.

[23] F. Vicentini et al., *NetKet 3: Machine learning toolbox for many-body quantum systems*, SciPost Phys. Codebases 7 (2022), doi:10.21468/SciPostPhysCodeb.7.

[24] F. Vicentini et al., *Codebase release 3.4 for NetKet*, SciPost Phys. Codebases 7-r3.4 (2022), doi:10.21468/SciPostPhysCodeb.7-r3.4.

[25] G. Carleo and M. Troyer, *Solving the quantum many-body problem with artificial neural networks*, Science **355**, 602 (2017), doi:10.1126/science.aag2302.

[26] P. Virtanen et al., *SciPy 1.0: Fundamental algorithms for scientific computing in Python*, Nat. Methods **17**, 261 (2020), doi:10.1038/s41592-019-0686-2.

[27] I. Goodfellow, Y. Bengio and A. Courville, *Deep learning*, MIT Press, Cambridge, USA (2016).

[28] K. P. Murphy, *Probabilistic machine learning: An introduction*, MIT Press, Cambridge, USA (2022).

[29] K. P. Murphy, *Probabilistic machine learning: Advanced topics*, MIT Press, Cambridge, USA (2023).

[30] A. Geron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*, O'Reilly Media, Sebastopol, USA, ISBN 9781492032649 (2019).

[31] *SOLAX*, https://github.com/pavlobilous/SOLAX.

[32] Pandas development team, *Pandas-dev/pandas: Pandas*, Zenodo (2020), doi:10.5281/zenodo.3509134.

[33] J. Heek, A. Levskaya, A. Oliver, M. Ritter, B. Rondepierre, A. Steiner and M. van Zee, *Flax: A neural network library and ecosystem for JAX* (2023), https://github.com/google/flax.

[34] C. Gaffney, D. Li, J. Zhang, R. Sang, A. Jain and H. Hu, *Orbax* (2024), https://orbax.readthedocs.io/en/latest/.

[35] *Jax installation* (2024), https://jax.readthedocs.io/en/latest/installation.html.

[36] W. McKinney, *Python for data analysis: Data wrangling with Pandas, Numpy, and Jupyter*, O'Reilly Media, Sebastopol, USA, ISBN 9781098104030 (2022).

[37] P. W. Anderson, *Localized magnetic states in metals*, Phys. Rev. **124**, 41 (1961), doi:10.1103/PhysRev.124.41.

[38] W. Metzner and D. Vollhardt, *Correlated lattice fermions in $d = \infty$ dimensions*, Phys. Rev. Lett. **62**, 324 (1989), doi:10.1103/PhysRevLett.62.324.

[39] A. Georges, G. Kotliar, W. Krauth and M. J. Rozenberg, *Dynamical mean-field theory of strongly correlated fermion systems and the limit of infinite dimensions*, Rev. Mod. Phys. **68**, 13 (1996), doi:10.1103/RevModPhys.68.13.

[40] G. Kotliar, S. Y. Savrasov, K. Haule, V. S. Oudovenko, O. Parcollet and C. A. Marianetti, *Electronic structure calculations with dynamical mean-field theory*, Rev. Mod. Phys. **78**, 865 (2006), doi:10.1103/RevModPhys.78.865.

[41] J. E. Hirsch and R. M. Fye, *Monte Carlo method for magnetic impurities in metals*, Phys. Rev. Lett. **56**, 2521 (1986), doi:10.1103/PhysRevLett.56.2521.

[42] E. Gull, A. J. Millis, A. I. Lichtenstein, A. N. Rubtsov, M. Troyer and P. Werner, *Continuous-time Monte Carlo methods for quantum impurity models*, Rev. Mod. Phys. **83**, 349 (2011), doi:10.1103/RevModPhys.83.349.

[43] G. Pan and Z. Y. Meng, *The sign problem in quantum Monte Carlo simulations*, in *Encyclopedia of condensed matter physics*, Elsevier, Amsterdam, Netherlands, ISBN 9780323914086 (2024), doi:10.1016/B978-0-323-90800-9.00095-0.

[44] M. Nuss, E. Arrigoni, M. Aichhorn and W. von der Linden, *Variational cluster approach to the single-impurity Anderson model*, Phys. Rev. B **85**, 235107 (2012), doi:10.1103/PhysRevB.85.235107.

[45] T. Helgaker, P. Jørgensen and J. Olsen, *Molecular electronic-structure theory*, Wiley, Chichester, UK, ISBN 9780471967552 (2000), doi:10.1002/9781119019572.

[46] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, (arXiv preprint) doi:10.48550/arXiv.1412.6980.

[47] A. Krizhevsky, I. Sutskever and G. E. Hinton, *Imagenet classification with deep convolutional neural networks*, Curran Associates, Red Hook, USA, ISBN 9781627480031 (2012).

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

[49] S. D. P. Flores, *Chembot: A machine learning approach to selective configuration interaction*, J. Chem. Theory Comput. **17**, 4028 (2021), doi:10.1021/acs.jctc.1c00196.

[50] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte and D. Vrgoč, *Foundations of JSON schema*, in *Proceedings of the 25th international conference on world wide web*, Geneva, Switzerland, ISBN 9781450341431 (2016), doi:10.1145/2872427.2883029.

[51] *Pickle – Python object serialization*, https://docs.python.org/3/library/pickle.html.

[52] S. Ainsworth, *Question #10674: "Write your model in jax they said. It'll be deterministic they said."* (2022), https://github.com/jax-ml/jax/discussions/10674.

[53] J. J. Mortensen et al., *GPAW: An open Python package for electronic structure calculations*, J. Chem. Phys. **160**, 092503 (2024), doi:10.1063/5.0182685.