NetKet 3: Machine Learning Toolbox for Many-Body Quantum Systems

Filippo Vicentini[1*], Damian Hofmann[2], Attila Szabó[3,4], Dian Wu[1], Christopher Roth[5], Clemens Giuliani[1], Gabriel Pescia[1], Jannes Nys[1], Vladimir Vargas-Calderon[6], Nikita Astrakhantsev[7] and Giuseppe Carleo[1]

1 Institute of Physics, École Polytechnique Fédérale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland
2 Max Planck Institute for the Structure and Dynamics of Matter, Luruper Chaussee 149, 22761 Hamburg, Germany
3 Rudolf Peierls Centre for Theoretical Physics, University of Oxford, Oxford OX1 3PU, United Kingdom
4 ISIS Facility, Rutherford Appleton Laboratory, Harwell Campus, Didcot OX11 0QX, United Kingdom
5 Physics Department, University of Texas at Austin
6 Grupo de Superconductividad y Nanotecnología, Departamento de Física, Universidad Nacional de Colombia, Bogotá, Colombia
7 Department of Physics, University of Zurich, CH-8057 Zurich, Switzerland
* filippo.vicentini@epfl.ch

December 20, 2021

Abstract

We introduce version 3 of NetKet, the machine learning toolbox for many-body quantum physics. NetKet is built around neural-network quantum states and provides efficient algorithms for their evaluation and optimization. This new version is built on top of JAX, a differentiable programming and accelerated linear algebra framework for the Python programming language. The most significant new feature is the possibility to define arbitrary neural network ansätze in pure Python code using the concise notation of machine-learning frameworks, which allows for just-in-time compilation as well as the implicit generation of gradients thanks to automatic differentiation. NetKet 3 also comes with support for GPU and TPU accelerators, advanced support for discrete symmetry groups, chunking to scale up to thousands of degrees of freedom, drivers for quantum dynamics applications, and improved modularity, allowing users to use only parts of the toolbox as a foundation for their own code.

1 Introduction

During the last two decades, we have witnessed tremendous advances in machine learning (ML) algorithms which have been used to solve previously difficult problems such as image recognition [1, 2] or natural language processing [3]. This has only been possible thanks to sustained hardware development: the last decade alone has seen a 50-fold increase in available computing power [4]. However, unlocking the full computational potential of modern arithmetic accelerators, such as GPUs, used to require significant technical skills, hampering researchers in their efforts. The incredible pace of algorithmic advances must therefore be attributed, at least in part, to the development of frameworks allowing researchers to tap into the full potential of computer clusters while writing high-level code [5, 6].

In the last few years, researchers in quantum physics have increasingly utilized machine-learning techniques to develop novel algorithms or improve on existing approaches [7]. In the context of variational methods for many-body quantum physics in particular, the method of *neural quantum states* (NQS) has been developed [8]. NQS are based on the idea of using neural networks as an efficient parametrization of the quantum wave function. They are of particular interest because of their potential to represent highly entangled states in more than one dimension with polynomial resources [9], which is a significant challenge for more established families of variational states. NQS are also flexible: they have been successfully used to determine variational ground states of classical [10] and quantum Hamiltonians [11–18] as well as excited states [13], to approximate Hamiltonian unitary dynamics [8, 19–24], and to solve the Lindblad master equation [25–27]. In particular, NQS are currently used in the study of frustrated quantum systems [13, 16–18, 28, 29], which have so far been challenging to optimize by established numerical techniques. They have also been used to perform tomographic state reconstruction [30] and efficiently approximate quantum circuits [31].

A complication often encountered when working with NQS is, however, that standard ML frameworks like Tensorflow [32] or PyTorch [33] are not geared towards these kind of *quantum mechanical* problems, and it often takes considerable technical expertise to use them for such non-standard tasks. Alternatively, researchers sometimes avoid those frameworks and implement their routines from scratch, but this often leads to sub-optimal performance. We believe that it is possible to foster this intersection of quantum physics and ML by providing an easy-to-use interface exposing quantum mechanical objects to ML frameworks.

We therefore introduce version 3 of the NETKET framework [34].[1] NETKET 3 is an open-source Python toolbox expressing several quantum mechanical primitives in the differentiable programming framework JAX [5, 35]. NETKET provides an easy-to-use interface to high-performance variational techniques without the need to delve into the details of their implementations, but customizability is not sacrificed and advanced users can inspect, modify, and extend practically every aspect of the package. Moreover, integration of our quantum object primitives with the JAX ecosystem allows users to easily define custom neural-network architectures and compute a range of quantum mechanical quantities, as well as their gradients, which are auto-generated through JAX's tracing-based approach.

---

[1]This manuscript refers to NETKET v3.3, released on 20 December 2021.

JAX provides the ability to write numerical code in pure Python using NUMPY-like calls for array operations, while still achieving high performance through just-in-time compilation using XLA, the accelerated linear algebra compiler that underlies TensorFlow. We have also integrated JAX and MPI with the help of MPI4JAX [36] to make NETKET scale to hundreds of computing nodes.

1.1 What's new

With the release of version 3, NETKET has moved from internally relying on a custom C++ core to the JAX framework, which allows models and algorithms to be written in pure Python and just-in-time compiled for high performance on both CPU and GPU platforms[2]. By using only Python, the installation process is greatly simplified and the barrier of entry for new contributors is lowered.

From a user perspective, the most important new feature is the possibility of writing custom NQS wave functions using JAX, which allows for quick prototyping and deployment, frees users from having to manually implement gradients due to JAX's support for automatic differentiation, and makes models easily portable to GPU platforms. Other prominent new features are

- support for unitary and Markovian dissipative dynamics;

- support for continuous systems;

- efficient implementations of the quantum geometric tensor and stochastic reconfiguration, which scale to models with millions of parameters;

- group-invariant and group-equivariant layers and architectures which support arbitrary discrete symmetries.

A more advanced feature is an extension mechanism built around multiple dispatch [37], which allows users to override algorithms used internally by NETKET without editing the source itself. This can be used to make NETKET work with custom objects and algorithms to study novel problems that do not easily fit what is already available.

1.2 Outline

NETKET provides both an intuitive high-level interface with sensible defaults to welcome beginners, as well as a complete set of options and lower-level functions for flexible use by advanced users. The high-level interface is built around quantum-mechanical objects such as *Hilbert spaces* (`netket.hilbert`) and *operators* (`netket.operator`), presented in section 2.

The central object in NETKET 3 is the *variational state,* discussed in section 3, which bring together the neural-network ansatz, its variational parameters, and a Monte-Carlo sampler. In section 3.2, we give an example on how to define an arbitrary neural network using a NETKET/JAX-compatible framework, while section 3.4 presents the new API of stochastic samplers. In section 3.5, we show how to compute the quantum geometric tensor (QGT) with NETKET, and compare the different implementations.

Section 4 shows how to use the three built-in optimization drivers to perform ground-state, steady-state, and dynamics calculations. Section 5 discusses NETKET's implementation of spatial symmetries and symmetric neural quantum states, which can be exploited

---

[2]Google's Tensor Processing Units (TPUs) are also, in principle, supported. However, at the time of writing they only support half-precision float16. Some modifications would be necessary to work-around loss of precision and gradient underflow.

to lower the size of the variational manifold and to target excited states in nontrivial symmetry sectors. In section 6, we also show how to study a system with continuous degrees of freedom, such as interacting particles in one or more spatial dimensions.

The final sections present detailed workflow examples of some of the more common use cases of NETKET. In section 7, we show how to study the ground state and the excited state of a lattice Hamiltonian. Section 8 gives examples of both unitary and Lindbladian dynamical simulations.

To conclude, section 9 presents scaling benchmarks of NETKET running across multiple devices and a performance comparison with jVMC [38], another library similar in scope to NETKET.

Readers who are already familiar with the previous version of NETKET might be especially interested in the variational state interface described in section 3.1, which replaces what was called *machine* in NETKET 2 [34], the QGT interface described in section 3.5, algorithms for dynamics (section 4.3 and section 8), and symmetry-aware NQS (section 5).

1.3 Installing NetKet

NETKET is a package written in pure Python; it requires a recent Python version, currently at least version 3.7. Even though NETKET itself is platform-agnostic, JAX, its main dependency, only works on MacOS and Linux at the time of writing.[3] Installing NETKET is straightforward and can be achieved running the following line inside a python environment:

```
1  pip install --upgrade netket
```

To enable GPU support, Linux with a recent CUDA version is required and a special version of JAX must be installed. As the appropriate installation procedure can change between JAX versions, we refer the reader to the official documentation[4] for detailed instructions.

NETKET by default does not make use of multiple CPUs that might be available to the user. Exploiting multiple processors, or even running across multiple nodes, requires MPI dependencies, which can be installed using the command

```
1  pip install --upgrade "netket[mpi]"
```

These dependencies, namely `mpi4py` and `mpi4jax`, can only be installed if a working MPI distribution is already available.

Once NETKET is installed, it can be imported in a Python session or script and its version can be checked as

```
1  >>> import netket as nk
2  >>> print(nk.__version__)
3  3.3
```

We recommend that users to use an up-to-date version when starting a new project. In code listings, we will often refer to the `netket` module as `nk` for brevity.

2 Quantum-mechanical primitives

In general, when working with NETKET, the flow is the following: first, one defines the Hilbert space of the system (section 2.1) and the Hamiltonian or super-operator of interest

---

[3]In principle, JAX runs on Windows, but users must compile it themselves, which is not an easy process.
[4]https://github.com/google/jax#installation

(section 2.2). Then, one builds a variational state (section 3.1), usually combining a neural-network model and a stochastic sampler. In this section, we describe the first step in this process, namely, how to define a quantum-mechanical system to be modelled.

2.1 Hilbert spaces

Hilbert-space objects determine the state space of a quantum system and a specific choice of basis. Functionality related to Hilbert spaces is contained in the `nk.hilbert` module; for brevity, we will often leave out the prefix `nk.hilbert` in this section.

All implementations of Hilbert spaces derive from the class `AbstractHilbert` and fall into two classes:

- discrete Hilbert spaces, which inherit from the abstract class `DiscreteHilbert` and include spin (`Spin`), Fock (`Fock`), and qubit (`Qubit`) spaces. Discrete spaces are typically used to describe lattice systems. The lattice structure itself is, however, not part of the Hilbert space class and can be defined separately.

- continuous Hilbert spaces, which inherit from the abstract class `ContinuousHilbert`. Currently, the only concrete continuous space provided by NETKET is `Particle`.

Continuous Hilbert spaces are discussed in section 6. A general discrete space with N sites has the structure

$$\mathcal{H}_{\text{discrete}} = \text{span}\{|s_0\rangle \otimes \cdots \otimes |s_{N-1}\rangle \mid s_i \in \mathcal{L}_i, i \in \{0, \ldots, N-1\}\}, \tag{1}$$

where $\mathcal{L}_i$ is the set of local quantum numbers (e.g., $\mathcal{L} = \{0, 1\}$ for a qubit, $\mathcal{L} = \{\pm 1\}$ for a spin-1/2 system in the $\sigma^{(z)}$ basis, or $\mathcal{L} = \{0, 1, \ldots, N_{\max}\}$ for a Fock space with up to $N_{\max}$ particles at each site) at site i. Constraints on the allowed quantum numbers are supported, resulting in Hilbert spaces that are subspaces of Eq. (1). For example, `Spin(1/2, total_sz=0)` creates a spin-1/2 space which only includes configurations $\{s_i\}$ that satisfy $\sum_{i=1}^{N} s_i = 0$. The corresponding basis states $|s\rangle$ span the zero-magnetization subspace. Similarly, constraints on the total population in Fock spaces are also supported.

Different spaces can be composed to create coupled systems by using the exponent operator (`**`) and the multiplication operator (`*`). For example, the code below creates the Hilbert space of a bosonic cavity with a cutoff of 10 particles at each site, coupled to 6 spin$-\frac{1}{2}$ degrees of freedom.

```
>>> hi = nk.hilbert.Fock(10) * nk.hilbert.Spin(1/2)**6
>>> print("Size of the hilbert space: ", hi.n_states)
Size of the hilbert space:  704
>>> print("Size of the basis: ", hi.size)
Size of the basis:  7
>>> hi.random_state(jax.random.PRNGKey(0), (2,))
DeviceArray([[10., -1.,  1., -1.,  1., -1., -1.],
             [ 9.,  1., -1., -1.,  1., -1.,  1.]], dtype=float32)
```

All Hilbert objects can generate random basis elements through the function `random_state(rng_key, shape, dtype)`, which has the same signature as standard random number generators in JAX. The first argument is a JAX random-generator state as returned by `jax.random.PRNGKey`, while the other arguments specify the number of output states and optionally the JAX data type. In this example, an array with two state vectors has been returned. The first entry of each corresponds to the Fock space and is thus an integer in $\{0, 1, \ldots, 10\}$, while the rest contains the spin quantum numbers.

Custom Hilbert spaces can be constructed by defining a class inheriting either from `ContinuousHilbert` for continuous spaces or `DiscreteHilbert` for discrete spaces. In the rest of the paper, we will always be working with discrete Hilbert spaces unless stated otherwise.

NETKET also supports working with super-operators, such as the Liouvillian used to define open quantum systems, and variational mixed states. The density matrix is an element of the space of linear operators acting on a Hilbert space, $\mathbb{B}(\mathcal{H})$. NETKET represents this space using the Choi–Jamilkowski isomorphism [39, 40] convention $\mathbb{B}(\mathcal{H}) \sim \mathcal{H} \otimes \mathcal{H}$; this "doubled" Hilbert space is implemented as `DoubledHilbert` . Doubled Hilbert spaces behave largely similarly to standard Hilbert spaces, but their bases have double the number of degrees of freedom; for example, super-operators can be defined straightforwardly as operators acting on them.

2.2 Linear operators

NETKET is designed to allow users to work with large systems, beyond the typically small system sizes that are accessible through exact diagonalization techniques. In order to compute expectation values $\langle \hat{O} \rangle$ on such large spaces, we must be able to efficiently represent the operators $\hat{O}$ and work with their matrix elements $\langle \sigma | \hat{O} | \eta \rangle$ without storing them in memory.

NETKET provides different implementations for the operators, tailored for different use cases; they are all available in the `netket.operator` namespace. These implementations inherit from one of the abstract classes `DiscreteOperator` or `ContinuousOperator` , depending on whether they work on discrete or continuous Hilbert spaces. We defer the discussion of operators acting on a continuous space to section 6, and only discuss discrete-space operators here.

An operator acting on a discrete space can be represented as a matrix with some matrix elements $\langle \sigma | \hat{O} | \eta \rangle$. As most of those elements are zero in physical systems, a standard approach is to store the operator as sparse matrices, a format that lowers the memory cost by only storing non-zero entries. However, the number of non-zero matrix elements still scales exponentially with the number of degrees of freedom, so sparse matrices cannot scale to the thousands of lattice sites that we want to support, either. For this reason, NETKET uses one of three custom formats to represent operators:

- `LocalOperator` is an implementation that can efficiently represent sums of K-local operators, that is, operators that only act nontrivially on a set of K sites. The memory cost of this format grows linearly with the number of operator terms and the number of degrees of freedom, but it scales exponentially in K.

- `PauliStrings` is an implementation that efficiently represents a product of Pauli X, Y, Z operators acting on the whole system. This format only works with qubit-like Hilbert spaces, but it is extremely efficient and has negligible memory cost.

- Special implementations like `Ising` , which hard-code the matrix elements of the operator. Those are the most efficient, though they cannot be customized at all.

The `nk.operator` submodule also contains ready-made implementations of commonly used operators, such as Pauli matrices, bosonic ladder or projection operators, and common Hamiltonians such as the Heisenberg, or the Bose-Hubbard models.

2.2.1 Manipulating operators

Operators can be manipulated similarly to standard matrices: they can be summed, subtracted and multiplied. In the example below we show how to build up the operator

$$\hat{O} = (\hat{\sigma}_0^x + \hat{\sigma}_1^x)^2 = 2(\hat{\sigma}_0^x \hat{\sigma}_1^x + 1). \tag{2}$$

starting from the Pauli X operator acting on the i-th site, σ_i^x, given by the function `nk.operator.spin.sigmax(hi, i)` :

```
1  >>> hi = nk.hilbert.Spin(1/2)**2
2  >>> op = nk.operator.spin.sigmax(hi,0) + nk.operator.spin.sigmax(hi,1)
3  >>> op = op * op
4  >>> op
5  LocalOperator(dim=2, acting_on=[[0], [0, 1], [1]], constant=0, dtype=float64)
6  >>> op.to_dense()
7  array([[2., 0., 0., 2.],
8         [0., 2., 2., 0.],
9         [0., 2., 2., 0.],
10        [2., 0., 0., 2.]])
```

In the last step, we convert the operator into a dense matrix using the `to_dense()` method; it is also possible to convert an operator into a sparse matrix using `to_sparse()` .

While it is possible to inspect those operators and (if the Hilbert space is small enough) to convert them to dense matrices, NETKET's operators are built in order to support efficient row indexing, similar to row-sparse (CSR) matrices. Given a basis vector $|\sigma\rangle$ in a Hilbert space, one can efficiently query the list of basis states $|\eta\rangle$ and matrix elements $O(\sigma, \eta)$ such that

$$O(\sigma, \eta) = \langle\sigma| \hat{O} |\eta\rangle \neq 0 \tag{3}$$

using the function `operator.get_conn(sigma)` , which returns both the vector of non-zero matrix elements and the corresponding list of indices $|\eta\rangle$, stored as a matrix.[5]

3 Variational quantum states

In this section, we first introduce the general interface of variational states, which can be used to represent both pure states (vectors in the Hilbert space) and mixed states (positive-definite density operators). We then present how to define variational ansätze and the stochastic samplers needed that generate Monte Carlo states.

3.1 Abstract interface

A variational state describes a parametrized quantum state that depends on a (possibly large) set of variational parameters θ. The quantum state can be either pure (denoted as $|\psi_\theta\rangle$) or mixed (written as a density matrix $\hat{\rho}_\theta$). NETKET defines an abstract interface, `netket.vqs.VariationalState` , for such objects; all classes that implement this interface will automatically work with all the high-level drivers (e.g., ground-state optimization or time-dependent variational dynamics) discussed in section 4. The `VariationalState` interface is relatively simple, as it has only four requirements:

---

[5]This querying is currently performed in Python code, just-in-time compiled using NUMBA [41], which runs on the CPU. If you run your computations on a GPU with a small number of samples, this might introduce a considerable slowdown. We are aware of this issue and plan to adapt our operators to be indexed directly on the GPU.

- The parameters θ of the variational state should be exposed through the attribute `parameters`, and they should be stored as an array or a nested dictionary of arrays.

- The expectation value $\langle \hat{A} \rangle_\theta$ of an operator $\hat{A}$ can be computed or estimated through the method `expect`.

- The conjugate gradient[6] of an expectation value with respect to the variational parameters, $\partial \langle \hat{A} \rangle_\theta / \partial \theta_j^*$ is computed with the method `expect_and_grad`.

- The quantum geometric tensor (section 3.5) of a variational state can be constructed with the method `quantum_geometric_tensor`.

At the time of writing, NETKET exposes three types of variational state:

- `nk.vqs.ExactState` represents a variational pure state $|\psi_\theta\rangle$ and computes expectation values, gradients and the geometric tensor by performing exact summation over the full Hilbert space.

- `nk.vqs.MCState` (short for *Monte Carlo state*) represents a variational pure state and computes expectation values, gradients and the geometric tensor by performing Monte Carlo sampling over the Hilbert space.

- `nk.vqs.MCMixedState` represents a variational mixed state and computes expectation values by sampling diagonal entries of the density matrix.

Variational states based on Monte Carlo sampling are the main tools that we expose to users, together with a wide variety of high-performance Monte Carlo samplers. More details about stochastic estimates and Monte Carlo sampling will be discussed in section 3.3 and section 3.4.

Dispatch and algorithm selection. With three different types of variational state and several different operators supported, it is hard to write a well-performing algorithm that works with all possible combinations of types that users might require. In order not to sacrifice performance for generic algorithms, NETKET uses the approach of multiple dispatch based on the PLUM module [37]. Combined with JAX's just-in-time compilation, this solution bears a strong resemblance to the approach commonly used in the Julia language [6].

Every time the user calls `VariationalState.expect` or `.expect_and_grad`, the types of the variational state and the operator are used to select the most specific algorithm that applies to those two types. This allows NETKET to provide generic algorithms that work for all operators, but keeps it easy to supply custom algorithms for specific operator types if desired.

This mechanism is also exposed to users: it is possible to override the algorithms used by NETKET to compute expectation values and gradients without modifying the source code of NETKET but simply by defining new dispatch rules using the syntax shown below.

```
1  @nk.vqs.expect.dispatch
2  def expect(vstate: MCState, operator: Ising):
3      # more efficient implementation than default one
4      #
```

---

[6]The conjugate gradient is used because, when optimizing a real-valued function of complex-variables, the direction of largest change corresponds to the conjugate gradient instead of the gradient [42]. If variables are real, the conjugate of the gradient $(\partial \langle \hat{A} \rangle_\theta / \partial \theta_j)^*$ is computed instead.

```
5        # expectation_value = ...
6        #
7        return expectation_value
```

3.2   Defining the variational ansatz

The main feature defining a variational state is the parameter-dependent mapping of an input configuration to the corresponding probability amplitude or, in other words, the quantum wave function (for pure states)

$$(\theta, s) \mapsto \psi_\theta(s) = \langle s | \psi_\theta \rangle \tag{4}$$

or quantum density matrix (for mixed states)

$$(\theta, s, s') \mapsto \rho_\theta(s, s') = \langle s | \, \hat{\rho}_\theta \, | s' \rangle. \tag{5}$$

In NETKET, this mapping is called a *model* (of the quantum state).[7]   In the case of NQS, the model is given by a neural network. For defining models, NETKET primarily relies on FLAX [43], a JAX-based neural-network framework. Ansätze are implemented as FLAX modules that map input configurations (the structure of which is determined by the Hilbert space) to the corresponding log-probability amplitudes. For example, pure quantum states are evaluated as

$$\ln \psi_\theta(s) = \texttt{module.apply} \, (\theta, s). \tag{6}$$

The use of log-amplitudes has the benefit that the log-derivatives $\partial \ln \psi_\theta(s)/\partial \theta_j$, often needed in variational optimization algorithms, are directly available through automatic differentiation of the model. It also makes it easier for the model to learn amplitudes whose absolute values span several orders of magnitude, which is common for many types of quantum states.

The recommended way to define a custom module is to subclass `flax.linen.Module` and providing a custom implementation of the `__call__` method. For example, a simple one-layer NQS with a wave function of the form

$$\ln \psi(s) = \sum_{j=1}^{M} \tanh[Ws + b]_j \tag{7}$$

can be defined as follows:

```
1  import netket as nk
2  import jax.numpy as jnp
3  import flax
4  import flax.linen as nn
5
6  class OneLayerNQS(nn.Module):
7      # Module hyperparameter:
8      n_hidden_units: int
9
10     @nn.compact
```

---

[7]The notion of "model" in NETKET 3 is related to the "machine" classes in NETKET 2 [34]. However, while NETKET 2 machines both define the mapping (4) and store the current parameters, this has been decoupled in NETKET 3. The model only specifies the mapping, while the parameters are stored in the variational state classes.

```
11        def __call__(self, s):
12            n_visible_units = s.shape[-1]
13            # define parameters
14            # the arguments are: name, initializer, shape, dtype
15            W = self.param(
16                "weights",
17                nn.initializers.normal(),
18                (self.n_hidden_units, n_visible_units),
19                jnp.complex128,
20            )
21            b = self.param(
22                "bias",
23                nn.initializers.normal(),
24                (self.n_hidden_units,),
25                jnp.complex128,
26            )
27
28            # multiply with weight matrix over last dimension of s
29            y = jnp.einsum("ij,...j", W, s)
30            # add bias
31            y += b
32            # apply tanh activation and sum
33            y = jnp.sum(jnp.tanh(y), axis=-1)
34
35            return y
```

The decorator `flax.linen.compact` used on `__call__` (line 10) makes it possible to define the network parameters directly in the body of the call function via `self.param` as done above (lines 15 and 21). Note that the network will be just-in-time (jit) compiled to efficient machine code using `jax.jit`, which means that all code inside the `__call__` method needs to be compatible with this. In particular, users should use `jax.numpy` for NumPy calls that need to happen at runtime; explicit Python control flow, such as `for` loops and `if` statements, should also be avoided, unless one explicitly wants to have them evaluate once at compile time. We refer users to the JAX documentation for further information on how to write jit-compatible code.

For performance reasons, the input to the module is batched. This means that, instead of passing a single array of quantum numbers s, a batch of multiple state vectors is passed as a matrix of shape `(batch_size, configuration_size)`. Therefore, operations like the sum over all feature indices in the example above need to be explicitly performed over the last axis. Note that if the output of the module is a real number, the wave function, obtained by exponentiating the output of the module, will be a positive real number. Therefore, in order to describe wave functions with a sign or with a complex phase, the module should output a complex value.

The module defined above can be used by first initializing the parameters using `module.init` and then computing log-amplitudes through `module.apply`:

```
1  >>> module = OneLayerNQS(n_hidden_units=16)
2  # init takes two arguments, a PRNG key for random initialization
3  # and a dummy array used to determine the input shape
4  # (here with a batch size of one):
5  >>> params = module.init(nk.jax.PRNGKey(0), jnp.zeros((1, 8)))
6  >>> module.apply(params, jnp.array([[-1, 1, -1, 1, -1, 1, -1, 1]]))
7  DeviceArray([-0.00047843+0.07939122j], dtype=complex128)
```

Name	NETKET class	References
Jastrow ansatz	`Jastrow`	[45, 46]
Restricted Boltzmann machine (RBM)	`RBM` , `RBMMultiVal` , `RBMModPhase`	[8]
Symmetric RBM	`RBMSymm`	[8]
Group-Equivariant Convolutional Neural Network	`GCNN`	Section 5 [47]
Autoregressive Neural Network	`ARNNDense` , `ARNNConv1D` , `ARNNConv2D` , `FastARNNConv1D` , `FastARNNConv2D`	[48]
Neural Density Matrix	`NDM`	[25, 49]

Table 1: List of models included in NETKET's `nk.models` submodule, together with relevant references.

Variational parameters are stored in a so-called PyTree. "PyTree" is a JAX-specific concept, referring to a data structure consisting of nested standard Python containers (such as lists, tuples, and dictionaries).[8] Any object that is not itself a PyTree, in particular NumPy-like arrays, is referred to as a "leaf". In particular, networks defined in FLAX store their parameters in a (potentially nested) dictionary, which provides name-based access to the network parameters.[9] For the `OneLayerNQS` defined above, the parameter PyTree has the structure:[10]

```
1  >>> print(params)
2  FrozenDict({
3      params: {
4          weights: DeviceArray(..., dtype=complex128),
5          bias: DeviceArray(..., dtype=complex128),
6      },
7  })
```

The names of the entries in the parameter dictionary correspond to those given in the `param` call when defining the model. NETKET functions often work directly with both plain arrays and PyTrees of arrays. Furthermore, NETKET provides a set of utility functions to perform linear algebra operations on such PyTrees in the `nk.jax` submodule.

While FLAX modules are the main way to implement wave functions for use with NETKET, it is also possible to use modules implemented in plain JAX or using other JAX-based frameworks (such as STAX, which is included with JAX, or Haiku [44]).

3.2.1 Pre-defined ansätze included with NetKet

NETKET provides a collection of pre-defined modules under `nk.models` , which allow quick access to many commonly used NQS architectures (Table 1). Furthermore, the `nk.nn`

---

[8] See also https://jax.readthedocs.io/en/4510-2/pytrees.html.

[9] In fact, FLAX stores networks parameters in an immutable `FrozenDict` object, which otherwise has the same semantics as a standard Python dictionary and, in particular, is also a valid PyTree. The parameters can be modified by converting to a standard mutable `dict` via `flax.core.unfreeze(params)` .

[10] The actual data contained in each array has been replaced with `...` for readability.

submodule contains more generic modules such as dense and convolutional layers to be used as building blocks for custom neural networks. Compared to some of the modules directly provided by FLAX, those provided by NETKET are designed to work well with complex numbers and thus are usually more convenient to use in NQS applications.[11] As an example, a multi-layer NQS with two convolutional and one final dense layer acting as a weighted sum can be defined as follows:

```python
class MultiLayerCNN(nk.nn.Module):
    features1: int
    features2: int
    kernel_size: int

    @nn.compact
    def __call__(self, s):
        # define layers
        layer1 = nk.nn.Conv(
            features=self.features1,
            kernel_size=self.kernel_size,
        )
        layer2 = nk.nn.Conv(
            features=self.features2,
            kernel_size=self.kernel_size,
        )
        weighted_sum = nk.nn.Dense(features=1)

        # apply layers and tanh activations
        y = jnp.tanh(layer1(s))
        y = jnp.tanh(layer2(y))
        y = weighted_sum(y)
        # last axis only has one entry, so we just return that
        # but keep the batch dimension
        return y[..., 0]
```

3.3 Estimating observables

For any variational ansatz, it is crucial to also have efficient algorithms for computing quantities of interest, in particular observables and their gradients. Since evaluating the wave function on all configurations is infeasible for larger Hilbert spaces, NQS approaches rely on Monte Carlo sampling of quantum expectation values.

Pure states. The quantum expectation value of an operator $\hat{A}$ on a non-normalized pure state $|\psi\rangle$ can be written as a classical expectation value over the Born distribution $p(s) \propto |\psi(s)|^2$ using the identity

$$\langle \hat{A} \rangle = \frac{\langle \psi | \hat{A} | \psi \rangle}{\langle \psi | \psi \rangle} = \sum_s \frac{|\psi(s)|^2}{\langle \psi | \psi \rangle} \tilde{A}(s) = \sum_s p(s) \tilde{A}(s) = \mathbb{E}[\tilde{A}], \tag{8}$$

---

[11]JAX, FLAX, and OPTAX have some minor issues with complex numbers at the time of writing, behaving unexpectedly in some cases. To improve the overall ecosystem, we have contributed several fixes to those libraries and started several discussions with the developers to find the best way to properly support complex numbers. Our aim is to eventually contribute all our patches to the upstream libraries so that `nk.nn` will no longer be necessary to provide such fixes.

where $\tilde{A}$ is the *local estimator*

$$\tilde{A}(s) = \frac{\langle s|\hat{A}|\psi\rangle}{\langle s|\psi\rangle} = \sum_{s'} \frac{\psi(s')}{\psi(s)} \langle s|\hat{A}|s'\rangle, \tag{9}$$

also known as the *local energy* when $\hat{A}$ is the Hamiltonian [50]. Even though the sum in eq. (9) runs over the full Hilbert space basis, the local estimator can be efficiently computed if the operator is sufficiently sparse in the given basis, i.e., all but a tractable number of matrix elements $\langle s|\hat{A}|s'\rangle$ are zero. Thus, an efficient algorithm is required that, given s, yields all connected configurations s' together with their respective matrix elements, as described in section 2.2. Given the derivatives of the log-amplitudes

$$O_i(s) = \frac{\partial \ln \psi_\theta(s)}{\partial \theta_i}, \tag{10}$$

gradients of expectation values can also be evaluated as the covariance

$$\frac{\partial \langle \hat{A} \rangle}{\partial \theta_i} = \mathbb{E}[(\tilde{A} - \mathbb{E}[\tilde{A}])^* O_i]. \tag{11}$$

The required classical expectation values are then estimated by averaging over a sequence $\{s_i\}_{i=1}^{N_{\mathrm{s}}}$ of configurations distributed according to the Born distribution $p(s) \propto |\psi(s)|^2$; e.g., eq. (8) becomes

$$\mathbb{E}[\tilde{A}] \approx \frac{1}{N_{\mathrm{s}}} \sum_{i=1}^{N_{\mathrm{s}}} \tilde{A}(s). \tag{12}$$

For some models, in particular autoregressive neural networks [48], one can efficiently draw samples from the Born distribution directly. For a general ansatz, however, this is not possible and Markov-chain Monte Carlo (MCMC) sampling methods [50] must be used: these generate a sequence (Markov chain) of samples that asymptotically follows the Born distribution. Such a chain can be generated using the Metropolis–Hastings algorithm [51], which is implemented in NetKet's sampler interface, described in the next section.

Mixed states. When evaluating observables for mixed states, it is possible to exploit a slightly different identity,

$$\langle \hat{A} \rangle = \frac{\mathrm{Tr}\left[\hat{\rho}\hat{A}\right]}{\mathrm{Tr}[\hat{\rho}]} = \sum_s \frac{\rho(s,s)}{\mathrm{Tr}[\hat{\rho}]} \tilde{A}_\rho(s) = \mathbb{E}[\tilde{A}_\rho], \tag{13}$$

which rewrites the quantum expectation value as a classical expectation over the probability distribution defined by the diagonal of the density matrix $p(s) \propto \rho(s,s)$. Here, $\tilde{A}_\rho$ denotes the local estimator of the observable *over a mixed state,*

$$\tilde{A}_\rho(s) = \frac{\langle s|\hat{\rho}\hat{A}|s\rangle}{\langle s|\hat{\rho}|s\rangle} = \sum_{s'} \frac{\rho(s,s')}{\rho(s,s)} \langle s'|\hat{A}|s\rangle. \tag{14}$$

It is then possible to follow the same procedures detailed in the previous paragraph for pure states to compute the gradient of an expectation value of an operator over a mixed state by replacing the probability distribution over which the average is computed and the local estimator.

3.4 Monte Carlo samplers

The sampling algorithm used to obtain a sequence of configurations from the probability distribution defined by the variational ansatz is specified by sampler classes inheriting from `nk.sampler.AbstractSampler`. Following the purely functional design of JAX, we define the sampler to be a stateless collection of settings and parameters, while storing all mutable state such as the PRNG key and the statistics of acceptances in an immutable sampler state object. Both the sampler and the sampler state are stored in the variational state, but they can be used independently, as they are decoupled from the rest of NETKET.

The Metropolis–Hastings algorithm is used to generate samples from an arbitrary probability distribution. In each step, it suggests a transition from the current configuration s to a proposed configuration s'. The proposal is accepted with probability

$$P_{\mathrm{acc}}(s \to s') = \min\left(1, \frac{P(s')}{P(s)}\frac{g(s \mid s')}{g(s' \mid s)}\right), \tag{15}$$

where P is the distribution being sampled from and $g(s' \mid s)$ is the conditional probability of proposing s' given the current s. We use $L(s, s') = \log[g(s \mid s')/g(s' \mid s)]$ to denote the correcting factor to the log probability due to the transition kernel. This factor is needed for asymmetric kernels that might propose one move with higher probability than its reverse. Simple kernels, such as a local spin flip or exchange, are symmetric, therefore $L(s, s') = L(s', s) = 1$, but other proposals, such as Hamiltonian sampling, are not necessarily symmetric and need this factor.

At the time of writing, NETKET exposes four types of rules to use with the Metropolis sampler: `MetropolisLocal`, which changes one discrete local degree of freedom in each transition; `MetropolisExchange`, which exchanges two local degrees of freedom respecting a conserved quantity (e.g., total particle number or magnetization); `MetropolisHamiltonian`, which transitions the configuration according to the off-diagonal elements of the Hamiltonian; and `MetropolisGaussian`, which moves a configuration with continuous degrees of freedom according to a Gaussian distribution.

The different transition kernels in these samplers are represented by `MetropolisRule` objects. To define a Metropolis sampling algorithm with a new transition kernel, one only needs to subclass `MetropolisRule` and implement the `transition` method, which gives s' and $L(s, s')$ in each transition. For example, the following transition rule changes the local degree of freedom on two sites at a time:

```python
from netket.hilbert.random import flip_state
from netket.sampler import MetropolisRule
from netket.utils.struct import dataclass

# To be jax-compatible, it must be a dataclass
@dataclass
class TwoLocalRule(MetropolisRule):
    def transition(rule, sampler, machine, parameters, state, key, σ):
        # Deduce the number of MCMC chains from input shape
        n_chains = σ.shape[0]
        # Load the Hilbert space of the sampler
        hilb = sampler.hilbert
        # Split the rng key into 2: one for each random operation
        key_indx, key_flip = jax.random.split(key, 2)
        # Pick two random sites on every chain
        indxs = jax.random.randint(
            key_indx, shape=(n_chains, 2), minval=0, maxval=hilb.size
```

Type	Name	Usage
MCMC (Metropolis)	`MetropolisLocal`	discrete Hilbert spaces
	`MetropolisExchange`	permutations of local states, conserving total magnetization in spin systems
	`MetropolisHamiltonian`	preserving symmetries of the Hamiltonian
	`MetropolisGaussian`	continuous Hilbert spaces
Direct	`ExactSampler`	small Hilbert spaces, performs MC sampling from the exact distribution
	`ARDirectSampler`	autoregressive models

Table 2: List of samplers in NETKET with their class names and a description.

```
18          )
19          # flip those sites
20          σp, _ = flip_state(hilb, key_flip, σ, indxs)
21
22          # If this transition had a correcting factor L, it's possible
23          # to return it as a vector in the second value
24          return σp, None
```

Once a custom rule is defined, a MCMC sampler using such rule can be constructed with the command `sampler = MetropolisSampler(hilbert, TwoLocalRule())`. Besides Metropolis algorithms, more advanced Markov chain algorithms can also be implemented as NETKET samplers. Currently, parallel tempering is provided as an experimental feature.

Some models allow us to directly generate samples that are exactly distributed according to the desired probability, without the use of Markov chains and the issue of autocorrelation, which often leads to more efficient sampling. In this case, direct samplers can be implemented with an interface similar to Markov chain samplers. Currently NETKET has implemented `ARDirectSampler` to be used with ARNNs. For benchmarking purposes, NETKET also provides `ExactSampler`, which allows direct sampling from any model by computing the full Born distribution $|\psi(s)|^2$ for all s. Table 2 is a list of all the samplers.

3.5 Quantum geometric tensor

The quantum geometric tensor (QGT) [52] of a pure state is the metric tensor induced by the Fubini–Study distance [53, 54]

$$d(\psi, \phi) = \cos^{-1} \sqrt{\frac{\langle \psi | \phi \rangle \langle \phi | \psi \rangle}{\langle \psi | \psi \rangle \langle \phi | \phi \rangle}}, \tag{16}$$

which is the natural and gauge-invariant distance between two pure quantum states $|\psi\rangle$ and $|\phi\rangle$. The QGT is commonly used for time evolution (see section 4.3) and for quantum natural gradient descent [55], which was originally developed in the VMC community

under the name of *stochastic reconfiguration* (SR) [56]. Quantum natural gradient descent is directly related to the natural gradient descent developed in the machine learning community [57].

From now on, we assume that the state $|\psi_\theta\rangle$ is parametrized by a set of parameters θ. Assuming further that $|\phi\rangle = |\psi_{\theta+\delta\theta}\rangle$, the distance (16) can be expanded to second order in the infinitesimal parameter change $\delta\theta$ as $d(\psi_\theta, \psi_{\theta+\delta\theta})^2 = (\delta\theta)^\dagger G(\delta\theta)$, where G is the quantum geometric tensor. For a holomorphic mapping $\theta \mapsto |\psi_\theta\rangle$, the QGT is given by

$$G_{ij}(\theta) = \frac{\left\langle \partial_{\theta_i}\psi_\theta \middle| \partial_{\theta_j}\psi_\theta \right\rangle}{\langle\psi|\psi\rangle} - \frac{\left\langle \partial_{\theta_i}\psi_\theta \middle| \psi_\theta \right\rangle \left\langle \psi_\theta \middle| \partial_{\theta_j}\psi_\theta \right\rangle}{\langle\psi|\psi\rangle^2} \tag{17}$$

where the indices i, j label the parameters and $\langle x|\partial_{\theta_i}\psi_\theta\rangle = \partial_{\theta_i}\langle x|\psi_\theta\rangle$. Similar to expectation values and their gradients, Eq. (17) can be rewritten as a classical expectation value with respect to the Born distribution $\propto |\psi(s)|^2$:

$$G_{ij}(\theta) = \mathbb{E}\left[O_i^* O_j\right] - \mathbb{E}\left[O_i\right]^* \mathbb{E}\left[O_j\right], \tag{18}$$

where O_i are the log-derivatives (10) of the ansatz.[12] This allows the quantum geometric tensor to be estimated using the same sampling procedure used to obtain expectation values and gradients. The QGT or its stochastic estimate is also commonly known as the *S matrix* [50] or *quantum Fisher matrix* (QFM) in analogy to the classical Fisher information matrix [55, 57, 58].

For applications such as quantum natural gradient descent or time-evolution it is usually not necessary to access the full, dense matrix representation $G_{ij}(\theta)$ of the quantum geometric tensor, but only to compute its product with a vector, $\tilde{v}_i = \sum_j G_{ij}(\theta)\, v_j$. When the variational ansatz ψ_θ has millions of parameters, the QGT can indeed be too large to be stored in memory. Exploiting the Gram matrix structure of the geometric tensor [59], we can directly compute its action on a vector without ever calculating the full matrix, trading memory requirements for an increased computational cost.

Given a variational state `vs` , a QGT object can be obtained by calling:

```
1  >>> qgt = vs.quantum_geometric_tensor()
```

This `qgt` object does not store the full matrix, but can still be applied to a vector with the same shape as the parameters:

```
1  >>> vec = jax.tree_map(jnp.ones_like, vs.parameters)
2  >>> qgt_times_vec = qgt @ vec
```

It can be converted to a dense matrix by calling `to_dense` :

```
1  # get the matrix (2d array) of the qgt
2  >>> qgt_dense = qgt.to_dense()
3  # flatten vec into a 1d Array
4  >>> grad_dense, unravel = nk.jax.tree_ravel(vec)
5  >>> qgt_times_vec = unravel(qgt_dense @ vec_dense)
```

The QGT can then be used together with a direct solver, such as `jnp.linalg.eigh` , `jnp.linalg.svd` , or `jnp.linalg.qr` .

---

[12]Strictly speaking, this estimator is only correct if $\psi_\theta(s) = 0 \implies \partial_{\theta_k}\psi_\theta(s) = 0$. This is because we multiplied and divided by $\psi_\theta(s)$ in the derivation of the estimator, which is only valid if $\psi_\theta(s) \neq 0$.

Mixed states. When working with mixed states, which are encoded in a density matrix, it is necessary to pick a suitable metric to induce the QGT. Even if the most physical distances for density matrices are the spectral norm or other trace-based norms [60, 61], it is generally hard to use them to define an expression for the QGT that can be efficiently sampled and computed at polynomial cost. While this might be regarded as a *barbaric* choice, it leads to an expression equivalent to eq. (18), where the expectation value is over the joint-distribution of the of row and column labels (s, s') of the squared density matrix $\propto |\rho(s, s')|^2$. Therefore, when working with mixed states, we resort to the L_2 norm, which is equivalent to treating the density matrix as a vector ("pure state") in an enlarged Hilbert space.

3.5.1 Implementation differences

There is some freedom in the way one can calculate the QGT, each different implementation taking a different tradeoff between computational and memory cost. In the example above, we have relied on NETKET to automatically select the best implementation according to some internal heuristics, but if one wants to push variational methods to the limits, it is useful to understand the two different implementations on offer:

- `QGTJacobian`, which computes and stores the Jacobian $O_i(s)$ of the model (cf. appendix B.1) at initialization (using reverse-mode automatic differentiation) and can be applied to a vector by performing two matrix-vector multiplications;

- `QGTOnTheFly`, which lazily computes the application of the quantum geometric tensor on a vector through automatic differentiation.

QGTOnTheFly is the most flexible and can be scaled to arbitrarily large systems. It is based on the observation that the QGT is the Jacobian of the model multiplied with its conjugate, which means that its action can be calculated by combining forward and reverse-mode automatic differentiation. At initialization, it only computes the linearization (forward pass), and then it effectively recomputes the gradients every time it is applied to a vector. However, since it never has to store these gradients, it is not limited by the available memory, which also makes it perform well for shallow neural-network models like RBMs. This method works with both holomorphic and non-holomorphic ansätze with no difference in performance.

QGTJacobian. For deep networks with ill-conditioned[13] quantum geometric tensors, recomputing the gradients at every step in an iterative solver might be very costly. `QGTJacobian` can therefore achieve better performance at the cost of considerably higher memory requirements because it precomputes the Jacobian at construction and stores it. The downside is that it has to store a *matrix* of shape $N_{\text{samples}} \times N_{\text{parameters}}$, which might not fit in the memory of a GPU. We note that there are two different implementations of `QGTJacobian`: `QGTJacobianDense` and `QGTJacobianPyTree`. The difference among the two is that in the former the Jacobian is stored contiguously in memory, leading to a better throughput on GPUs, while the latter stores them in the same structure as the parameters (so each parameter block is separated from the others). Converting from the

---

[13]The number of steps required to find a solution with an iterative linear solver grows with the condition number of the matrix. Therefore, an ill-conditioned matrix requires many steps of iterative solver. Monte Carlo estimates of the QGT tend to be ill-conditioned even for a large number of samples [50]. This is especially true for NQS, which typically feature QGTs with a spectrum spanning many orders of magnitude, often making QGT-based algorithms challenging to stabilize [16, 22, 24, 58].

Implementation	Extra arguments	Use-cases	Limitations
`QGTOnTheFly`	`rescale_shift`	shallow networks, large numbers of parameters and samples, few solver steps	
`QGTJacobianDense`	`mode` `holomorphic`	deep networks with narrow layers	requires homogeneous parameter types, memory bound
`QGTJacobianPyTree`		deep networks with heterogeneous parameters	memory bound

Table 3: Overview of the three QGT implementations currently provided by NETKET with their respective options and limitations.

non-contiguous (`QGTJacobianPyTree`) to the contiguous (`QGTJacobianDense`) format has, however, a computational and memory cost which might shadow its benefit. Moreover, the dense format does not work with non-homogeneous parameter data types. The basic `QGTJacobian` algorithm supports both holomorphic and non-holomorphic NQS, but a better performing algorithm for holomorphic ansätze can be accessed instantiating it with the option `holomorphic=True` .

Key differences between the different QGT implementations are summarized in Table 3. Implementations can be selected, and options passed to them, as shown below:

```
1  >>> from netket.optimizer.qgt import (
2  ...      QGTJacobianPyTree, QGTJacobianDense, QGTOnTheFly
3  ... )
4  >>> qgt1 = vs.quantum_geometric_tensor(QGTOnTheFly)
5  >>> qgt2 = vs.quantum_geometric_tensor(QGTJacobianPyTree(holomorphic=True))
6  >>> qgt3 = vs.quantum_geometric_tensor(QGTJacobianDense)
```

Holomorphicity. When performing time evolution or natural gradient descent, one does not always need the full quantum geometric tensor: for ansätze with real parameters, as well as in the case of non-holomorphic wave functions,[14] only the real part of the QGT is used. The real and imaginary parts of the QGT are only required when working with a holomorphic ansatz. (An in-depth discussion of why this is the case can be found at [62, Table 1].) For this reason, NETKET's QGT implementations return the full geometric tensor only for holomorphic complex-parameter ansätze, and its real part in all other cases.

Linear Systems. For most applications involving the QGT, a linear system of equations of the kind

$$G_{ij}\delta_j = f_j \tag{19}$$

needs to be solved. This can be done using the standard JAX/NumPy functions:

---

[14] Non-holomorphic functions of complex parameters are internally handled by both JAX and NETKET as real-parameter functions that take the real and imaginary parts of the "complex parameters" separately.

```
1  >>> # iterative solver
2  >>> x, info = jax.scipy.sparse.linalg.cg(qgt, F)
3  >>> # direct solver, acting on the dense matrix
4  >>> x, info = jax.numpy.linalg.cholesky(qgt.to_dense(), F)
```

However, we recommend that users call the `solve` method on the QGT object, which allows some additional optimization that may improve performance:

```
1  >>> x, info = qgt.solve(jax.scipy.sparse.linalg.cg, g)
2  >>> x, info = qgt.solve(nk.optimizer.solver.cholesky, g)
```

While this works with any of the representations it is advisable to only use Jacobian based implementations (`QGTJacobianPyTree` or `QGTJacobianDense`) with direct solvers, since constructing the QGT matrix from `QGTOnTheFly` requires multiplication with all basis vectors, which is not as efficient. Finally, we highlight the fact that users can write their own functions to solve the linear system (19) using advanced regularization schemes (see for instance ref. [22]) and use them together with `qgt.solve` , as long as they respect the standard NumPy solver interface.

4 Algorithms for variational states

The main use case of NETKET is variational optimization of wave function ansätze. In the current version NETKET, three algorithms are provided out of the box via high-level driver classes: variational Monte Carlo (VMC) for finding ground states of (Hermitian) Hamiltonians, time-dependent variational Monte Carlo (t-VMC) for real- and imaginary-time evolution, and steady-state search of Liouvillian open-system dynamics.

These drivers are part of the `nk.driver` module but we also export them from the `nk` namespace. They are constructed from the relevant physical model (e.g., a Hamiltonian), a variational state, and other objects used by the optimization method. They all support the `run` method, which performs a number of optimization steps and logs their progress (e.g., variational energies and network parameters) in a variety of output formats.

We highlight that these drivers are built on top of the functionalities described in sections 2 and 3, and users are free to implement their own drivers or optimization loops, as demonstrated in section 4.4.

4.1 Ground-state search

NETKET provides the variational driver `nk.VMC` for searching for minimal-energy states using VMC [50]. In the simplest case, the `VMC` constructor takes three arguments: the Hamiltonian, an optimizer and the variational state (see section 3.1). NETKET makes use of optimizers provided by the JAX-based OPTAX library [63],[15] which can be directly passed to `VMC` , allowing the user to build complex training schedules or custom optimizers. In each optimization step, new samples of the variational state are drawn, which are used to estimate the gradient of the Hamiltonian with respect to the parameters θ of the ansatz [50],

$$f_i \equiv \frac{\partial \langle \hat{H} \rangle}{\partial \theta_i^*} = \mathbb{E}[O_i^*(\tilde{H} - \mathbb{E}[\tilde{H}])], \qquad (20)$$

---

[15]The `nk.optimizer` submodule includes a few optimizers for ease of use and backward compatibility: these are simply re-exports from OPTAX.

where $\tilde{H}$ is the local estimator (9) of the Hamiltonian, known as the *local energy*, and O_i is the log-derivative (10) of the wave function. All expectation values in eq. (20) are with respect to the Born distribution $\propto |\psi(\,\cdot\,)|^2$ and so can be estimated by averaging over the Monte Carlo samples. These gradients are then passed on to the OPTAX optimizer, which may transform them (using, e.g., Adam) further before updating the parameters θ. Using the simple stochastic gradient descent optimizer `optax.sgd` (alias `nk.optimizer.Sgd`), the update rule is

$$\theta_i \mapsto \theta_i - \eta f_i. \tag{21}$$

To improve on plain stochastic gradient descent, the `VMC` interface allows passing a keyword argument `preconditioner`. This must be a function that maps a variational state and the gradient vector f_i to the vector δ_i to be passed to the optimizer as gradients instead of f_i. An important use case is *stochastic reconfiguration* [50], where the preconditioner is the quantum geometric tensor (or its real part if the parameters are real), also known as the S-matrix:

$$G_{ij}\delta_j = f_i; \qquad\qquad \mathrm{Re}(G_{ij})\delta_j = f_i. \tag{22}$$

The corresponding preconditioner can be created from a QGT class and a JAX-compatible linear solver (the default is `jax.scipy.sparse.linalg.cg`) using `nk.optimizer.SR`:

```
1  # Construct the SR object with the chosen algorithm
2  sr = nk.optimizer.SR(
3  qgt = nk.optimizer.qgt.QGTOnTheFly,
4      solver=jax.scipy.sparse.linalg.bicgstab,
5      diag_shift=0.01,
6  )
7
8  # Construct the VMC driver
9  vmc = nk.VMC(
10     hamiltonian,                         # The Hamiltonian to optimize
11     nk.optimizer.sgd(learning_rate=0.1), # The optimizer
12     variational_state=vstate,            # The variational state
13     preconditioner=sr,                   # The preconditioner
14 )
```

4.2 Finding steady states

In order to study open quantum systems, NETKET provides the `nk.SteadyState` variational driver for determining the variational steady-state $\hat{\rho}_{ss}$ defined as the stationary point of an arbitrary super-operator $\mathcal{L}$,

$$0 = \frac{d\hat{\rho}}{dt} = \mathcal{L}\hat{\rho}. \tag{23}$$

The search is performed by minimizing the Frobenius norm of the time-derivative [25], which defines the cost function

$$\mathcal{C}(\theta) = \frac{\|\mathcal{L}\hat{\rho}\|_2^2}{\|\hat{\rho}\|_2^2} = \frac{\mathrm{Tr}\left[\hat{\rho}^\dagger \mathcal{L}^\dagger \mathcal{L}\hat{\rho}\right]}{\mathrm{Tr}\left[\hat{\rho}^\dagger \hat{\rho}\right]}, \tag{24}$$

which has a global minimum for the steady state. The stochastic gradient is estimated over the probability distribution of the entries of the vectorized density matrix according to the formula:

$$f_i \equiv \frac{\partial}{\partial \theta_i^*} \frac{\|\mathcal{L}\hat{\rho}\|_2^2}{\|\hat{\rho}\|_2^2} = \mathbb{E}\left[\tilde{\mathcal{L}}\nabla_i^* \tilde{\mathcal{L}}\right] - \mathbb{E}[O_i^* \tilde{\mathcal{L}}^2], \tag{25}$$

where $\tilde{\mathcal{L}}(s, s') = \sum_{m,m'} \mathcal{L}(s, s'; m, m')\rho(m, m')/\rho(s, s')$ is the local estimator proposed in [25], and the expectation values are taken with respect to the "Born distribution" of the vectorized density matrix, $p(s, s') \propto |\rho(s, s')|^2$. The optimization works like the ground-state optimization provided by `nk.VMC` : the gradient is passed to an OPTAX optimizer, which may transform it further before updating the parameters θ. The simplest optimizer, `optax.sgd` , would update the parameters according to the equation

$$\theta_i \to \theta_i - \eta f_i. \tag{26}$$

To improve the performance of the optimization, it is possible to pass the keyword argument `preconditioner` to specify a gradient preconditioner, such as *stochastic reconfiguration* that uses the quantum geometric tensor to transform the gradient. The geometric tensor is computed according to the L_2 norm of the vectorized density matrix (see section 3.5).

As an example, we provide a snippet to study the steady state of a transverse-field Ising chain with 10 spins and spin relaxation corresponding to the Lindblad master equation

$$\mathcal{L}\hat{\rho} = -i\left[\hat{H}, \hat{\rho}\right] + \sum_i \hat{\sigma}_i^- \hat{\rho}\hat{\sigma}_i^+ - \frac{1}{2}\left\{\hat{\sigma}_i^+ \hat{\sigma}_i^-, \hat{\rho}\right\}. \tag{27}$$

We first define the Hamiltonian and a list of jump operators, which are stored in a `operator.LocalLiovillian` object, which is a lazy representation of the super-operator $\mathcal{L}$. Next, a variational mixed state is built by defining a sampler over the doubled Hilbert space `lind.hilbert` and optionally a different sampler for the diagonal distribution $p(s) \propto \rho(s, s)$, which is used to estimate expectation values of operators. The number of samples used to estimate super-operators and operators can be specified separately, as shown in the example by specifying `n_samples` and `n_samples_diag` .

```
1   import netket as nk
2   # Define the geometry of the latticfe
3   g = nk.graph.Hypercube(length=10, n_dim=1, pbc=False)
4   # Hilbert space of spins on the graph
5   hi = nk.hilbert.Spin(s=1 / 2, N=g.n_nodes)
6
7   # Construct the Liouvillian Master Equation
8   ha = nk.operator.Ising(hi, graph=g, h=0.5)
9   j_ops = [nk.operator.spin.sigmam(hi, i) for i in range(g.n_nodes)]
10  # Create the Liouvillian with Hamiltonian and jump operators
11  lind = nk.operator.LocalLiouvillian(ha, j_ops)
12
13  # Observable
14  sz = sum([nk.operator.spin.sigmam(hi, i) for i in range(g.n_nodes)])
15
16  # Neural Density Matrix
17  sa = nk.sampler.MetropolisLocal(lind.hilbert)
18  vs = nk.vqs.MCMixedState(
19      sa, nk.models.NDM(beta=1), n_samples=2000, n_samples_diag=500
20  )
21  # Optimizer
22  op = nk.optimizer.Sgd(0.01)
23  sr = nk.optimizer.SR(diag_shift=0.01)
24
25  ss = nk.SteadyState(lind, op, variational_state=vs, preconditioner=sr)
26  out = ss.run(n_iter=300, obs={"Sz": sz})
```

4.3   Time propagation

Time propagation of variational states can be performed by incorporating the time dependence in the variational parameters and deriving an equation of motion that gives a trajectory in parameters space $\theta(t)$ that approximates the desired quantum dynamics. For real-time dynamics of pure and mixed NQS, such an equation of motion can be derived from the time-dependent variational principle (TDVP) [62, 64, 65]. When combined with VMC sampling to estimate the equation of motion (EOM), this is known as time-dependent variational Monte Carlo (t-VMC) [50, 66] and is the primary approach currently used in NQS literature [8, 19, 20, 22–24]. For complex holomorphic parametrizations, the TDVP equation of motion is

$$G_{ij}(\theta)\,\dot{\theta}_j = \gamma f_i(\theta, t), \tag{28}$$

with the QGT G and gradient f defined in sections 4.1 and 4.2. After solving eq. (28), the resulting parameter derivative $\dot{\theta}$ can be passed to an ODE solver. The factor γ determines the type of evolution:

- For $\gamma = -i$, the EOM approximates the real-time Schrödinger equation on the variational manifold, the simulation of which is the main use case for the t-VMC implementation provided by NETKET.

- For $\gamma = -1$, the EOM approximates the imaginary-time Schrödinger equation on the variational manifold. When solved using the first-order Euler scheme $\theta(t + dt) = \theta(t) + \dot{\theta}\,dt$, this EOM is equivalent to stochastic reconfiguration with learning rate dt. Imaginary-time propagation with higher-order ODE solvers can therefore also be used for ground state search as an alternative to VMC. This can result in improved convergence in some cases [18].

- For $\gamma = 1$ and with the Lindbladian super-operator taking the place of the Hamiltonian in the definition of the gradient f, this ansatz yields the dissipative real-time evolution according to the Gorini-Kossakowski-Lindblad-Sudarshan master equation [67]. Our implementation uses the QGT induced by the vector norm [26] as discussed in the last paragraph of section 3.5.

The current version of NETKET provides a set of Runge–Kutta (RK) solvers based on JAX and a driver class `TDVP` implementing the t-VMC algorithm for the three use cases listed above. At the time of writing, these features are provided as a preview version in the `netket.experimental` namespace,[16] as their API is still subject to ongoing development. Runge–Kutta solvers implement the propagation scheme [68]

$$\theta(t + dt) = \theta(t) + dt \sum_{l=1}^{L} b_l k_l \tag{29}$$

using a linear combination of slopes

$$k_l = F\left(\theta(t) + \sum_{m=1}^{l-1} a_{lm} k_m,\ t + c_l\,dt\right), \tag{30}$$

each determined by the solution $F(\theta, t) = \dot{\theta}$ of the equation of motion (28) at an intermediate time step. The coefficients $\{a_{lm}\}$, $\{b_l\}$, and $\{c_l\}$ determine the specific RK scheme and its order. NETKET further supports step size control when using adaptive RK schemes.

---

[16]The ODE solvers are located in the submodule `netket.experimental.dynamics`, the driver under `netket.experimental.TDVP`.

In this case, the step size is dynamically adjusted based on an embedded error estimate that can be computed with little overhead during the RK step (29) [68]. Step size control requires a norm on the parameters space in order to estimate the magnitude of the error. Typically, the Euclidean norm $\|\delta\| = \sqrt{\sum_i |\delta_i|^2}$ is used. However, since different directions in parameters space influence the quantum state to different degrees, it can be beneficial to use the norm $\|\delta\|_G = \sqrt{\sum_i \delta_i^* G_{ij} \delta_j}$ induced by the QGT G as suggested in Ref. [22], which takes this curvature into account and is also provided as an option with the NETKET time-evolution driver.

   An example demonstrating the use of NETKET's time evolution functionality is provided in Sec. 8.

4.4   Implementing custom algorithms using NetKet

While key algorithms for energy optimization, steady states, and time propagation are provided out of the box in the current NETKET version, there are many more applications of NQS. While we wish to provide new high-level driver classes for additional use cases, such as quantum state tomography [30] or general overlap optimization [31] it is already possible and encouraged for users to implement their own algorithms on top of NETKET. For this reason, we provide the core building blocks of NQS algorithms in a composable fashion.

   For example, it is possible to write a simple loop that solves the TDVP equation of motion (28) used in first-order Euler scheme [i.e., $\theta(t + dt) = \theta(t) + \dot{\theta}(t)\,dt$] very concisely, making use of the elementary building blocks provided by the `VariationalState` class:

```python
def custom_simple_tdvp(
    hamiltonian: AbstractOperator,   # Hamiltonian
    vstate: VariationalState,        # variational state
    t0: float,                       # initial time
    dt: float,                       # time step
    t_end: float,                    # end time
):
    t = t0
    while t < t_end:
        # compute the energy gradient f
        energy, f = vstate.expect_and_grad(hamiltonian)
        G = vstate.quantum_geometric_tensor()
        # multiply the gradient by -1.0j for unitary dynamics
        gamma_f = jax.tree_map(lambda x: -1.0j * x, f)
        # Solve the linear system using any solver, such as CG
        # (or write your own regularization scheme)
        dtheta, _ = G.solve(jax.scipy.sparse.linalg.cg, gamma_f)
        # update the parameters (theta = theta + dt * dtheta)
        vs.parameters = jax.tree_multimap(
            lambda x, y: x + dt * y, vs.parameters, dtheta
        )
        t = t + dt
```

While the included `TDVP` driver (section 4.3) provides many additional features (such as error handling, step size control, or higher-order integrators) and makes use of JAX's just-in-time compilation, this simple implementation already provides basic functionality and shows how NETKET can be used for quick prototyping.

5 Symmetry-aware neural quantum states

NETKET includes a powerful set of utilities for implementing NQS ansätze that are symmetric or transform correctly under the action of discrete symmetry groups. This is useful for modeling symmetric (e.g., lattice) Hamiltonians, whose eigenstates transform under irreducible representations of their symmetry groups. Restricting the Hilbert space to individual symmetry sectors can improve the convergence of variational optimization [69] and the accuracy of its result [15, 17, 47, 70]. Additionally, symmetry restrictions can be used to find excited states [14, 17, 71], provided they are the lowest energy level in a particular symmetry sector.

NETKET uses *group convolutional neural networks* (GCNNs) to build wave functions that are symmetric over a finite group G. GCNNs generalize convolutional neural networks, invariant under the Abelian translation group, to general symmetry groups G which may contain non-commuting elements [72]. GCNNs were originally designed to be invariant, but they can be modified to transform under an arbitrary irreducible representation (irrep) of G, using the projection operator [73]

$$|\psi_\chi\rangle = \sum_{g \in G} \chi_g^* \, g|\psi\rangle, \tag{31}$$

where g runs over all symmetry operations in G, with corresponding characters χ_g. Under the trivial irrep, where all characters are unity, the invariant model is recovered.

NETKET can infer the full space group of a lattice, defined as a set of permutations of lattice sites, starting from a geometric description of its point group. It can also generate nontrivial irrep characters [to be used in (31) for states with nonzero wave vectors or transforming nontrivially under point-group symmetries] using a convenient interface that follows standard crystallographic formalism [74]. In addition, NETKET provides powerful group-theoretic algorithms for arbitrary permutation groups of lattice sites, allowing new symmetry elements to be easily defined.

Pre-built GCNNs are then provided in the `nk.models` submodule, which can be constructed by specifying few parameters, such as the number of features in each layer, and the lattice or permutation group under which the network should transform. Symmetric RBMs [8] are also implemented as one-layer GCNNs that aggregate convolutional features using a product rather than a sum. These pre-built network architectures are made up of individual layers found in the `nk.nn` subdmodule, which can be used directly to build custom symmetric ansätze.

Section 5.1 describes the NETKET interface for constructing space groups of lattices and their irreps. Usage of GCNNs is described in section 5.2, while appendix A provides mathematical and implementation details.

5.1 Symmetry groups and representation theory

NETKET supports symmetry groups that act on a discrete Hilbert space defined on a lattice. On such a Hilbert space, space-group symmetries act by permuting sites; most generally, therefore, arbitrary subgroups of the symmetric group S_N of a lattice of N sites are supported. A symmetry group can be specified directly as a list of permutations, as in the following example, which enforces the symmetry $\psi(s_0, s_1, s_2, s_3) = \psi(s_3, s_1, s_2, s_0)$ for all four-spin configurations $s = (s_0, \ldots, s_3)$, $s_i = \pm 1$:

```
1  hi = nk.hilbert.Spin(1/2, N=4)
2  symms = [
3      [0, 1, 2, 3],   # identity element
```

```
4        [3, 1, 2, 0],  # swap first and last site
5    ]
6    model = nk.models.RBMSymm(symms, alpha=1)
```

The listed permutations are required to form a group and, in particular, the identity operation $e : s \mapsto s$ must always be included.

It is inconvenient and error-prone to specify all space-group symmetries of a large lattice by their indices. Therefore, NETKET provides support for abstract representations of permutation and point groups through the `nk.utils.group` module, complete with algorithms to compute irreducible representations [75–77]. The module also contains a library of two- and three-dimensional point groups, which can be turned into lattice-site permutation groups using the graph class `nk.graph.Lattice` (but not general `Graph` objects, which carry no geometric information about the system):

```
1    from netket.utils.group.planar import D
2    from netket.graph import Lattice
3
4    # construct a centred rectangular lattice
5    lattice = Lattice(
6        basis_vectors = [[2,0], [0,1]], # each row is a lattice vector
7        extent = (5,5),
8        site_offsets = [[0,0], [1,0.5]], # each row is the position of a site in
         the unit cell
9        point_group = D(2) # the point group of the lattice, here Z_2 x Z_2
10   )
```

NETKET contains specialized constructors for some lattices (e.g., `Square` or `Pyrochlore`) which come with a default point group; however, these can be overridden in methods like `Lattice.space_group`:

```
1    from netket.utils.group.planar import rotation, reflection_group, D
2    from netket.utils.group import PointGroup, Identity
3    from netket.graph import Honeycomb
4
5    # construct the D_6 point group of the honeycomb lattice by hand
6    cyclic_6 = PointGroup(
7        [Identity()] + [rotation(360 / n * i) for i in range(1, n)],
8        ndim=2,
9    )
10   # the @ operator returns the Cartesian product of groups
11   # but doesn't check for group structure
12   dihedral_6 = reflection_group(angle=0) @ cyclic_6
13
14   assert dihedral_6 == D(6)
15
16   lattice = Honeycomb([6,6])
17
18   # returns the full space group of 'lattice' as a PermutationGroup
19   space_group = lattice.space_group()
20   # the space group is spanned by 6^2 translations and 12 point-group
         symmetries
21   assert len(space_group) == 12 * 6 * 6
22
23   # do this if the Hamiltonian breaks reflection symmetry
24   # can also be used for generic Lattices that have no default point group
25   space_group = lattice.space_group(cyclic_6)
```

Irreducible representation (irrep) matrices can be computed for any point or permutation group object using the method `irrep_matrices()`. Characters (the traces of these matrices) are returned by the method `character_table()` as a matrix, each row of which lists the characters of all group elements. Character tables closer to the format familiar from quantum chemistry texts are produced by `character_table_readable()`. Irrep matrices and character tables are calculated using adaptations of Dixon's [75] and Burnside's [76] algorithms, respectively.

It would, however, be impractical to inspect irreps of a large space group directly to specify the symmetry sector on which to project a GCNN wave function. Exploiting the semidirect-product structure of space groups [77], space-group irreps are usually[17] described in terms of a set of symmetry-related wave vectors (known as a *star*) and an irrep of the subgroup of the point group that leaves the same invariant (known as the *little group*) [74]. Irreps can be constructed in this paradigm using `SpaceGroupBuilder` objects returned by `Lattice.space_group_builder()`:

```python
from netket.graph import Triangular

lattice = Triangular([6,6])
momentum = [0,0]
# space_group_builder() takes an optional PointGroup argument
sgb = lattice.space_group_builder()

# choosing a representation
# this one corresponds to the B_2 irrep at the Gamma point
chi = sgb.space_group_irreps(momentum)[3]
```

The irrep `chi`, generated using the little group, is equivalent to one of the irreps in `Lattice.space_group().character_table()` and can thus be used for symmetry-projecting GCNN ansätze. The order in which irreps of the little group are returned can readily be checked in an interactive session:

```python
>>> sgb.little_group(momentum).character_table_readable()
(['1xId()', '2xRot(60)', '2xRot(120)', '1xRot(180)', '3xRefl(0)',
    '3xRefl(-30)'],
array([[ 1.,  1.,  1.,  1.,  1.,  1.],      # this is irrep A1
       [ 1.,  1.,  1.,  1., -1., -1.],      # A2
       [ 1., -1.,  1., -1.,  1., -1.],      # B1
       [ 1., -1.,  1., -1., -1.,  1.],      # B2
       [ 2.,  1., -1., -2.,  0.,  0.],      # E1
       [ 2., -1., -1.,  2.,  0.,  0.]]))    # E2
```

The main caveat in using this machinery is that the point groups predefined in NETKET all leave the origin invariant (except for `cubic.Fd3m` which represents the "nonsymmorphic point group" of the diamond/pyrochlore lattice) and thus only work well with lattices in which the origin has full point-group symmetry. This behaviour can be changed (see the definition of `cubic.Fd3m` for an example), but it is generally safer to define lattices using the proper Wyckoff positions [74], of which the origin is usually maximally symmetric.

---

[17]Representation theory for wave vectors on the surface of the Brillouin zone in a nonsymmorphic space group is much more complicated [77] and is not currently implemented in NETKET.

	mode="irreps"	mode="fft"
Can be used for	any group	only space groups
Symmetries can be specified by	• `Lattice` • `PermutationGroup` • Symmetry permutations and irrep matrices	• `Lattice` • `PermutationGroup` and shape of translation group • Symmetry permutations, product table, and shape of translation group
Kernel memory footprint per layer	$O(f_{\text{in}} f_{\text{out}} |G|)$	$O(f_{\text{in}} f_{\text{out}} |G||P|)$
Evaluation time per layer per sample	$O[(f_{\text{in}} + f_{\text{out}})|G|^2 + f_{\text{in}} f_{\text{out}} \sum_a d_a^3]$	$O[(f_{\text{in}} + f_{\text{out}})|G| \log|T| + f_{\text{in}} f_{\text{out}} |G||P|]$
Preferable for	• large point groups • if expanded `"fft"` kernels don't fit in memory	• small point groups • very large batches (see App. A.2)

Table 4: Comparison of GCNN implementations. $f_{\text{in,out}}$ stands for the number of input and output features, $|G|, |P|, |T|$ for the sizes of the space group, point group, and translation group, respectively. d_a are the dimensions of irreps of G; in a large space group, $\sum_a d_a^3$ scales as $|G||P|$.

5.2 Using group convolutional neural networks (GCNNs)

NETKET uses GCNNs [47, 72] to create NQS ansätze that are symmetric under space groups of lattices. These networks consist of alternating *group convolutional layers* and pointwise nonlinearities. The former can be thought of as a generalization of convolutional layers to a generic finite group G. They are *equivariant,* that is, if their inputs are transformed by some space-group symmetry, features in all subsequent layers are transformed accordingly. As a result, the output of a GCNN can be understood as amplitudes of the wave functions $g|\psi\rangle$ for all $g \in G$, which can be combined into a symmetric wave function using the projection operator (31). Further details about equivariance and group convolutions are given in appendix A.1.

GCNNs are constructed by the function `nk.models.GCNN`. Symmetries are specified either as a `PermutationGroup` or a `Lattice`. In the latter case, the symmetry group is given by `space_group()`; an optional `point_group` argument to `GCNN` can be used to override the default point group. By default, output transforms under the trivial irrep $\chi_g \equiv 1$, that is, all output features are averaged together to obtain a wave function that is fully symmetric under the whole space group. Other irreps can be specified through the `characters` argument, which takes a vector of the same size as the space group.

```
1  from netket.graph import Triangular
2  from netket.models import GCNN
3  from netket.utils.group.planar import C
4
5  lattice = Triangular([6,6])
6  momentum = [0,0]
7  sgb = lattice.space_group_builder()
8  chi = sgb.space_group_irreps(momentum)[3]
9
```

```
10   # This transforms as the trivial irrep Gamma A_1
11   gcnn1 = GCNN(lattice, layers = 4, features=4)
12
13   # This transforms as Gamma B_2
14   gcnn2 = GCNN(lattice.space_group(), characters=chi, layers=4, features=4)
15
16   # This does not enforce reflection symmetry
17   gcnn3 = GCNN(lattice, point_group=C(6), layers=4, features=4)
```

NETKET currently supports two implementations of GCNNs, one based on group Fourier transforms (`mode="irreps"`), the other using fast Fourier transforms on each coset of the translation group (`mode="fft"`): these are discussed in more detail in appendix A.2. Their behavior is equivalent, but their performance and calling sequence is different, as explained in table 4. A default `mode="auto"` is also available. For spin models, parity symmetry (taking σ^z to $-\sigma^z$) is a useful extension of the U(1) spin symmetry group enforced by fixing magnetization along the σ^z axis. Parity-enforcing GCNNs can be constructed using the `parity` argument, which can be set to ± 1.

In addition to deep GCNNs, fully symmetric RBMs [8] are implemented in `nk.models.` `RBMSymm` as a single-layer GCNN from which the wave function is computed as

$$\psi = \prod_{i,g \in G} \cosh f_g^{(i)} \implies \log \psi = \sum_{i,g \in G} \ln \cosh f_g^{(i)}. \tag{32}$$

Due to the products (rather than sums) used, this ansatz only supports wave functions that transform under the trivial irrep. An RBM-like structure closer to that of ref. [70] can be achieved using a single-layer GCNN:

```
1   from netket.models import GCNN, RBMSymm
2   from netket.nn import logcosh
3
4   # fully symmetric RBM
5   rbm1 = RBMSymm(group, alpha=4)
6
7   # symmetrized RBM similar to (Nomura, 2021)
8   rbm2 = GCNN(group, layers=1, features=4, output_activation=logcosh)
```

6 Quantum systems with continuous degrees of freedom

In this section we will introduce the tools provided by NETKET to study systems with continuous degrees of freedom. The interface is very similar to the one introduced in section 2 for discrete degrees of freedom.

6.1 Continuous Hilbert spaces

Similar to the discrete Hilbert spaces, the bosonic Hilbert space of N particles in continuous space has the structure

$$\mathcal{H}_{\text{continuous}} = \text{span}\{|x_0\rangle \otimes \cdots \otimes |x_{N-1}\rangle : x_i \in \mathcal{L}_i, i \in \{0, \dots, N-1\}\} \tag{33}$$

where $\mathcal{L}_i$ is the space available to each individual boson: for example, $\mathcal{L}_i$ is $\mathbb{R}^d$ for a free particle in d spatial dimensions, and $[0, L]^d$ for particles confined to a d-dimensional box of side length L. In the case of finite simulation cells, the boundaries can be equipped

with periodic boundary conditions.

In the following snippet, we define the Hilbert space of five bosons in two spatial dimensions, confined to $[0, 10]^2$ with periodic boundary conditions:

```
1  >>> hilb = nk.hilbert.Particle(N=5, L=(10.0, 10.0), pbc=True)
2  >>> print("Size of the basis: ", hilb.size)
3  Size of the basis:  10
4  >>> hilb.random_state(nk.jax.PRNGKey(0), (2,))
5  [[0.02952452 0.21660899 2.836163   3.5628846  4.5622005  5.9473248
6    6.104126   8.14864    9.163713   9.263418  ]
7   [9.85617    0.4667458  2.211265   4.1587596  4.250165   6.69916
8    6.5165453  7.3764215  8.508119   0.08060825]]
```

6.2   Linear operators

Similar to the discrete-variable case, expectation values of operators can be estimated as classical averages of the local estimator

$$\tilde{O}(x) = \frac{\langle x| \hat{O} |\psi\rangle}{\langle x|\psi\rangle} \tag{34}$$

over the (continuous) Born distribution $p(x) \propto |\psi(x)|^2$. NETKET provides the base class `ContinuousOperator` to write custom (local) operators and readily implements Hamiltonians of the form ($\hbar = 1$ in our units)

$$\hat{H} = -\frac{1}{2} \sum_i \frac{1}{m_i} \nabla_i^2 + \hat{V}(\{\mathbf{x}_i\}) \tag{35}$$

using the predefined operators `KineticEnergy` and `PotentialEnergy`. For example, a harmonically confined system described by $\hat{H} = -\frac{1}{2} \sum_i \nabla_i^2 + \frac{1}{2} \sum_i \hat{\mathbf{x}}_i^2$ can be implemented as

```
1  # This function takes a single vector and returns a scalar
2  def v(x):
3      return 0.5*jnp.linalg.norm(x) ** 2
4
5  # Construct the Kinetic energy term with unit mass
6  H_kin = nk.operator.KineticEnergy(hilb, mass=1.0)
7  # Construct the Potential energy term using the potential defined above
8  H_pot = nk.operator.PotentialEnergy(hilb, v)
9
10 # Sum the two objects into a single Operator
11 H = H_kin + H_pot
```

Operators defined on continuous Hilbert spaces cannot be converted to a matrix form or used in exact diagonalization, in contrast to those defined on discrete Hilbert spaces. Continuous operators can still be used to compute expectation values and their gradients with a variational state.

6.3   Samplers

Out of the built-in samplers in the current version of NETKET (section 3.4), only the Markov chain Monte Carlo sampler `MetropolisSampler` supports continuous degrees
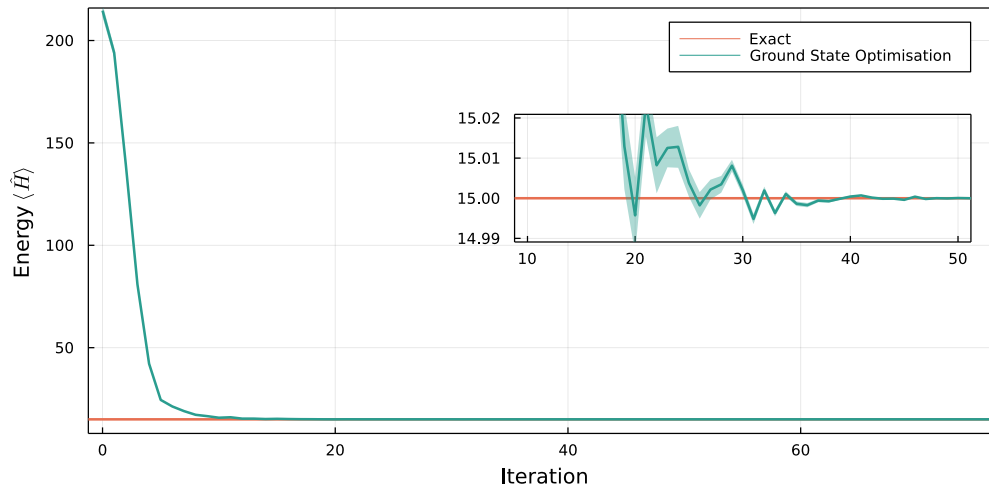
Figure 1: VMC energy estimate as a function of the optimization step for a continuous-space system of $N = 10$ particles in $d = 3$ spatial dimensions subject to a harmonic confinement.

of freedom, as both `ExactSampler` and the autoregressive `ARNNSampler` rely on the sampled basis being countable. For continuous spaces, we provide the transition rule `sampler.rules.GaussianRule` which proposes new states by adding a random shift to every degree of freedom sampled from a Gaussian of customizable width. More complex transition rules can be defined following the instructions provided in section 3.4.

6.4   Harmonic oscillators

As a complete example of how to use continuous-space Hilbert spaces, operators, variational states, and the VMC driver together, consider 10 particles in three-dimensional space, confined by a harmonic potential $V(x) = x^2/2$. The exact ground-state energy of this system is known to be $E_0 = 15$. We use the multivariate Gaussian ansatz $\log \psi(x) = -x^{\mathrm{T}} \Sigma^{-1} x$, where Σ is randomly initialized with the constraint that it be positive definite.

```
1   import netket as nk
2   import jax.numpy as jnp
3
4   def v(x):
5       return 0.5*jnp.linalg.norm(x) ** 2
6
7   hilb = nk.hilbert.Particle(N=10, L=(jnp.inf,jnp.inf,jnp.inf), pbc=False)
8   ekin = nk.operator.KineticEnergy(hilb, mass=1.0)
9   pot = nk.operator.PotentialEnergy(hilb, v)
10  ha = ekin + pot
11
12  sa = nk.sampler.MetropolisGaussian(hilb, sigma=0.1, n_chains=16, n_sweeps=1)
13  model = nk.models.Gaussian(dtype=float)
14  vs = nk.vqs.MCState(sa, model, n_samples=10 ** 4, n_discard=2000)
15
16  op = nk.optimizer.Sgd(0.1)
17  sr = nk.optimizer.SR(diag_shift=0.01)
18
19  gs = nk.VMC(ha, op, sab, variational_state=vs, preconditioner=sr)
```

```
20  gs.run(n_iter=100, out="HO_10_3d")
```

We show the training curve of above snippet in Fig. 1; exact ground-state energy is recovered to a very high accuracy.

7 Example: Finding ground and excited states of a lattice model

In this example, we define the J_1–J_2 Heisenberg Hamiltonian

$$H = J_1 \sum_{\langle ij \rangle} \vec{\sigma}_i \cdot \vec{\sigma}_j + J_2 \sum_{\langle\langle ij \rangle\rangle} \vec{\sigma}_i \cdot \vec{\sigma}_j, \tag{36}$$

on a 10×10 square lattice and use the VMC code introduced in section 4.1 to find a variational approximation of its ground state. This model gives rise to several phases of matter, including magnetically ordered states, a valence bond solid, and a quantum spin liquid. Here, we set $J_1 = 1, J_2 = 0.5$, inside the spin liquid phase [71, 78].

Our example is optimized to run on a single GPU with 16 GB of memory. We will make note of what should be changed when running the simulation on CPUs.

7.1 Defining the lattice and the Hamiltonian

We use the `Lattice` class to define the square lattice and generate its space-group symmetries. By passing `max_neighbor_order=2` to the constructor, we generate graph edges for both nearest and next-nearest neighbours. The pre-defined `Heisenberg` class supports passing different coupling constants for both types of edge.

```
1   import netket as nk
2   import numpy as np
3   import json
4   from math import pi
5
6   L = 10
7   # Build square lattice with nearest and next-nearest neighbor edges
8   lattice = nk.graph.Square(L, max_neighbor_order=2)
9   hi = nk.hilbert.Spin(s=1 / 2, total_sz=0, N=lattice.n_nodes)
10  # Heisenberg with coupling J=1.0 for nearest neighbors
11  # and J=0.5 for next-nearest neighbors
12  H = nk.operator.Heisenberg(hilbert=hi, graph=lattice, J=[1.0, 0.5])
```

7.2 Defining and training a symmetric ansatz

To enforce all spatial symmetries of (36), we use the GCNN ansatz described in section 5. By default, the GCNN projects onto the symmetric representation, which contains the ground state for this geometry. We select singlet states by only sampling basis states with $\sum_i S_i^z = 0$ and setting spin parity using `parity=1`. We use a model with four layers, each containing four feature vectors (i.e., four hidden units for each of the $8L^2$ space-group symmetries). To exploit the high degree of parallelism of GPUs, we set `sampler.n_chains`

equal to `vstate.n_samples` [18]. When using CPUs, `n_chains` should be set to a smaller value.

```python
machine = GCNN(
    symmetries=lattice,
    parity=1,
    layers=4,
    features=4,
    dtype=np.complex128,
)
sampler = nk.sampler.MetropolisExchange(
    hilbert=hi,
    n_chains=1024,
    graph=lattice,
    d_max=2,
)
opt = nk.optimizer.Sgd(learning_rate=0.02)
sr = nk.optimizer.SR(diag_shift=0.01)
vstate = nk.vqs.MCState(
    sampler=sampler,
    model=machine,
    n_samples=1024,
    n_discard_per_chain=0,
    chunk_size=4096,
)
gs = nk.VMC(H, opt, variational_state=vstate, preconditioner=sr)
gs.run(n_iter=200, out="ground_state")

data = json.load(open("ground_state.log"))
print(np.mean(data["Energy"]["Mean"]["real"][-20:])/400)
#  Output: -0.49562531096409457
print(np.std(data["Energy"]["Mean"]["real"][-20:])/400)
#  Output: 0.0002492304032505182
```

This calculation, which takes about 30 minutes on an NVIDIA A100 GPU, already delivers a fairly accurate variational energy. The evolution of variational energy during the training procedure is shown in fig. 2.

We note that a typical initialization of a GCNN gives rise to ferromagnetic correlations, which can make training an antiferromagnetic Hamiltonian unstable [16, 47]. Therefore, it is often good practice to pre-optimize the phases by restricting all amplitudes to unity by setting `equal_amplitudes=True` switch and training only the phases of the network. These parameters can then be loaded into a model with `equal_amplitudes=False`.

7.3   Finding an excited state

We can also find low-lying excited states using this procedure, by projecting the wavefunction onto a different irrep. Here, we consider the first gapless mode in the Anderson tower of states of the Néel antiferromagnet [79], a triplet at wave vector (π, π) that transforms

---

[18]Note that this results in a somewhat non-standard MC scheme where, instead of an ensemble of chains with generally non-zero internal autocorrelation, the sampler produces an ensemble of independently drawn single configurations (`n_samples_per_chain==1`). Since the sampler state of the previous VMC step is used as initial state for the next step, there can still be a residual autocorrelation with the previous samples, which is, however, alleviated by the sampler performing N_{sites} intermediate updates before yielding the single requested sample with the settings used here.
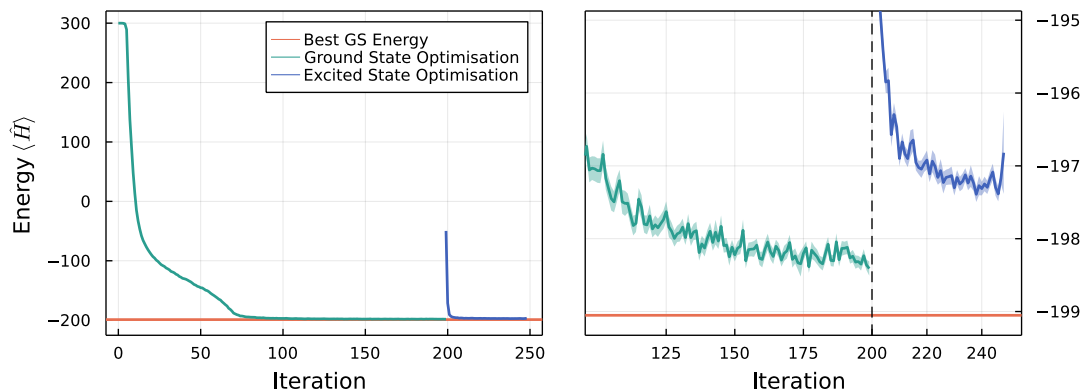
Figure 2: Energy evolution of variational ground states (green) and excited states after transfer learning (blue), compared to the lowest known variational energy for the 10×10 square-lattice J_1–J_2 model [71] (black). The variational energies can be further improved by allowing more training steps, Monte Carlo samples, etc. The plot on the right zooms in on the lowest energies.

trivially under all point-group symmetries. This mode is still gapless in the quantum spin liquid; we project out spin-singlets by focusing on parity odd states.

We expedite this calculation by using parameters optimized for the ground-state sector as an initial guess. The resulting wave function will already have a low variational energy (as shown in fig. 2) and correlations typical for low-energy eigenstates.

```python
# store the optimized ground-state parameters
saved_params = vstate.parameters
# Compute the characters of the first excited state
characters = lattice.space_group_builder().space_group_irreps(pi, pi)[0]
# Construct a model respecting the first-excited state symmetries
machine = GCNN(
    symmetries=lattice,
    characters=characters,
    parity=-1,
    layers=4,
    features=4,
    dtype=complex,
)
vstate = nk.vqs.MCState(
    sampler=sampler,
    model=machine,
    n_samples=1024,
    n_discard_per_chain=0,
    chunk_size=4096,
)
# assign the old parameters to the new variational state
vstate.parameters = saved_params
gs = nk.VMC(H, opt, variational_state=vstate, preconditioner=sr)
gs.run(n_iter=50, out='excited_state')

data = json.load(open("excited_state.log"))
print(np.mean(data["Energy"]["Mean"]["real"][-10:])/400)
#  Output: -0.49301426054097885
print(np.std(data["Energy"]["Mean"]["real"][-10:])/400)
#  Output: 0.0003802670752071611
```

8 Example: Real-time dynamics

We demonstrate the simulation of NQS dynamics in the transverse-field Ising model (TFIM) on an L site chain with periodic boundaries, using a restricted Boltzmann machine (RBM) as the NQS ansatz. The Hamiltonian reads

$$\hat{H}_{\text{Ising}} = J \sum_{\langle ij \rangle} \hat{\sigma}_i^z \hat{\sigma}_j^z - h \sum_i \hat{\sigma}_i^x \tag{37}$$

with $J = 1$ and $h = 1$ and periodic boundary conditions. We will estimate the expectation value of the transverse magnetization $\hat{S}^x = \sum_i \hat{\sigma}_i^x$ along the way.

We simulate the dynamics starting from an initial state $|\psi(t_0)\rangle$ that is the ground state for the TFIM Hamiltonian with $h = 1/2$. The random weight initialization of a neural network yields a random initial state. Therefore, we determine the weights corresponding to this initial state by performing a ground-state optimization. Even though the TFIM ground state can be parametrized using an RBM ansatz with real-valued weights, we need to use complex-valued weights here, in order to describe the complex-phase of the wave function that arises during the time evolution[19]. In this example, we work with a chain of $L = 20$ sites, which can be easily simulated on a typical laptop with the parameters below.

```python
import netket as nk
import netket.experimental as nkx

# Spin Hilbert space on 20-site chain with PBC
L = 20
chain = nk.graph.Chain(L)
hi = nk.hilbert.Spin(s=1/2) ** L

# Define RBM ansatz and variational state
rbm = nk.models.RBM(alpha=1, dtype=complex)
sampler = nk.sampler.MetropolisLocal(hi)
vstate = nk.vqs.MCState(sampler, rbm, n_samples=4096)

# Hamiltonian at J=1 (default) and external field h=1/2
ha0 = nk.operator.Ising(hi, chain, h=0.5)
# Observable (transverse magnetization)
obs = {"sx": sum(nk.operator.spin.sigmax(hi, i) for i in range(L))}

# First, find the ground state of ha0 to use it as initial state
optimizer = nk.optimizer.Sgd(0.01)
sr = nk.optimizer.SR()
vmc = nk.VMC(ha0, optimizer, variational_state=vstate, preconditioner=sr)
# We run VMC with SR for 300 steps
vmc.run(300, out="ising1d_groundstate_log", obs=obs)
```

8.1 Unitary dynamics

Starting from the ground state, we can compute the dynamics after a quench of the transferse field strength to $h = 1$. We use the second-order Heun scheme for time stepping, with a step size of $dt = 0.01$, and explicitly specify a QGT implementation (compare Sec. 3.5) in order to make use of the more efficient code path for holomorphic models.

---

[19]It is possible to first use a real-weight RBM for the initial state preparation and then switch to complex-valued weights for the dynamics. For the sake of simplicity, we leave out this extra step in the present example.
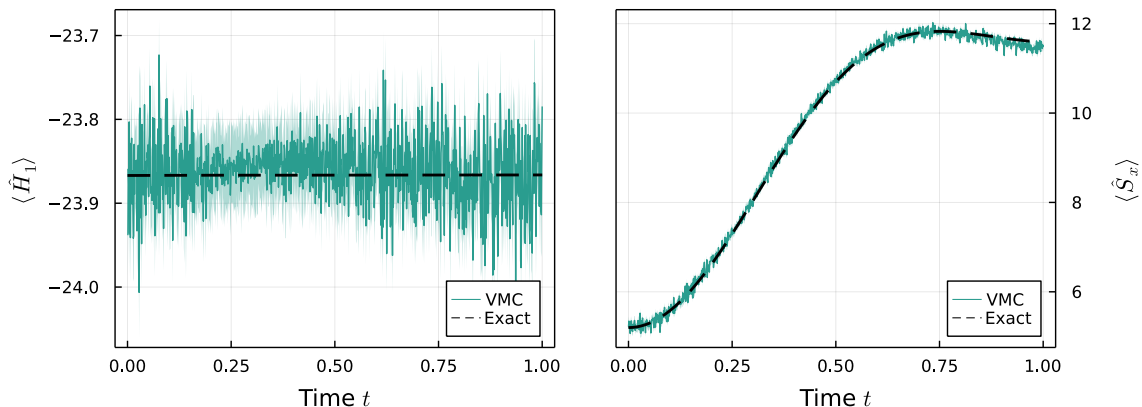
Figure 3: Comparison between the exact (dashed line) and variational dynamics of a quench on the Transverse Field Ising model. **(Left):** expectation value of the quenched Hamiltonian, which is conserved by the unitary dynamics. The shaded area represents the uncertainty due to Monte-Carlo sampling. **(Right):** expectation value of the total magnetization along the $\hat{x}$ axis during the evolution.

```
1  # Quenched Hamiltonian
2  ha1 = nk.operator.Ising(hi, chain, h=1.0)
3  # Heun integrator configuration
4  integrator = nkx.dynamics.Heun(dt=0.001)
5  # QGT options
6  qgt = nk.optimizer.qgt.QGTJacobianDense(holomorphic=True)
7  # Creating the time-evolution driver
8  te = nkx.TDVP(ha1, vstate, integrator, qgt=qgt)
9  # Run the t-VMC solver until time T=1.0
10 te.run(T=1.0, out="ising1d_quench_log", obs=obs)
```

In fig. 3 we show the results of this calculation, comparing against an exact solution computed using QuTiP [80, 81].

8.2 Dissipative dynamics (Lindblad master equation)

In section 4.3 we have shown that the time-dependent variational principle can also be used to study the dissipative dynamics of an open quantum system. In this section we give a concrete example, studying the transverse-field Ising model coupled to a zero-temperature bath. The coupling is modeled through the spin depolarization operators $\hat{\sigma}_i^-$ acting on every site i.

As the dissipative dynamics converges to the steady state, which is also an attractor of the non-unitary dynamics, we will use a weak-simulation of the dynamics[20] to determine the steady-state more efficiently than using the natural gradient descent scheme proposed in ref. [25]. This scheme is similar to what was proposed in ref. [26].

We employ the positive-definite RBM ansatz proposed in ref. [49]. A version of that network with complex-valued parameters is provided in NETKET with the name `nk.models.NDM` .

```
1  # The graph of the Hamiltonian
```

---

[20]We use weak and strong simulation in the sense of the theory of numerical SDE schemes. This means that weak integration is an integration which is not accurate at finite times but which converges to the right state at long times. A strong integration yields the correct state at every time t.

```
2  g = nk.graph.Chain(L, pbc=False)
3  # Hilbert space
4  hi = nk.hilbert.Spin(0.5)**g.n_nodes
5  # The Hamiltonian
6  ha = nk.operator.Ising(hi, graph=g, h=-1.3, J=0.5)
7  # Define the list of jump operators
8  j_ops = [nk.operator.spin.sigmam(hi, i) for i in range(g.n_nodes)]
9  # Construct the Liouvillian
10 lind = nk.operator.LocalLiouvillian(ha, j_ops)
11
12 # observable
13 Sx = sum([nk.operator.spin.sigmax(hi, i) for i in
       range(g.n_nodes)])/g.n_nodes
14
15 # Positive-definite RBM-like ansatz (Torlai et al.)
16 ma = nk.models.NDM(alpha=1, beta=2)
17 # MetropolisLocal sampling on the Choi's doubled space.
18 sa = nk.sampler.MetropolisLocal(lind.hilbert, n_chains=16)
19 # Mixed Variational State. Use less samples for the observables.
20 vs = nk.vqs.MCMixedState(
21     sa, ma, n_samples=1600, n_samples_diag=800, n_discard_per_chain=60
22 )
23 # Setup the ODE integrator and QGT.
24 integrator = nkx.dynamics.Euler(dt=0.01)
25 # The NDM ansatz is not holomorphic because it uses conjugation
26 qgt = nk.optimizer.qgt.QGTJacobianPyTree(holomorphic=False, diag_shift=0.05)
27 te = nkx.TDVP(lind, variational_state=vs, integrator=integrator, qgt=qgt)
28
29 # run the simulation and compute observables
30 te.run(T=6.0, obs={"Sx": Sx, "LdagL": lind.H @ lind})
```

In the listing above, we first construct the Liouvillian by assembling the Hamiltonian and the jump operators, then we construct the variational mixed state. We chose a different number of samples for the diagonal, used when sampling the observables, as that happens on a smaller space with respect to the full system. For the geometric tensor, we choose the `QGTJacobianPyTree` and specify that the ansatz is non holomorphic (that is the default, but a warning would be printed otherwise asking the user to be explicit). The choice is motivated by the fact that the TDVP driver by default uses an SVD-based solver, which works best `QGTJacobian`-based implementations. However, `NDM` uses a mix of complex and real parameters which is not supported by `QGTJacobianDense`, and would throw an error. Normally, to simulate a meaningful dynamics you'd want to keep the diagonal shift small, but since we are striving for a weak simulation a large value helps stabilize the dynamics.

9 Benchmarks

9.1 Variational Monte Carlo

We benchmark NETKET by measuring its performance on the 1D/2D transverse-field Ising model defined as in eq. (37) with $J = 1$ and $h = 1$ and periodic boundary conditions.

We first monitor the scaling behavior of NETKET's VMC implementation by running an energy optimization consisting of 100 steps. In order to carry out a meaningful benchmark, we run a first VMC step to trigger the JIT-compiler on the relevant JAX and NUMBA
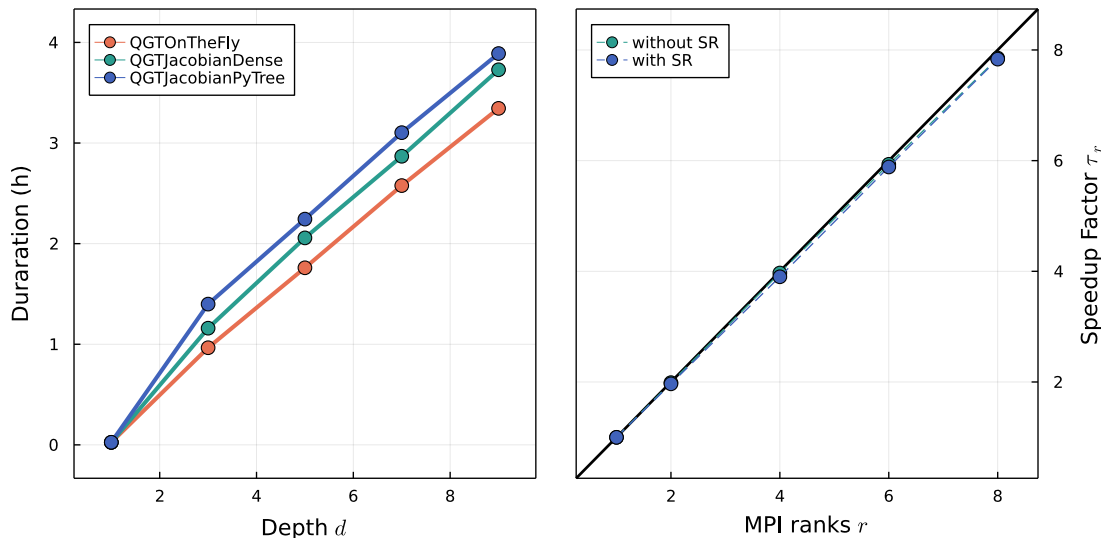
Figure 4: **(Left):** Benchmark of NETKET's VMC implementation. Each data point shows the minimum time spent (out of 5 repetitions) to evolve a DNN with depth d layers and complex weights over 100 VMC steps for the 1D transverse-field Ising model with $L = 256$ and $N_{\text{samples}} = 2^{14} = 16384$. **(Right):** Scaling behavior of the required computational time as a function of the number of MPI ranks on a single node. We repeat 5 VMC optimizations and report the minimum time required for 100 steps for the 1D Ising model with $L = 256$ and an RBM with $\alpha = 1$ with complex weights. The black line represents ideal scaling behavior.

functions, while all reported timings are for the evaluation of JIT compiled functions only. The left panel of fig. 4 depicts the scaling behavior of the computational time as a function of the complexity of the NQS model, using three implementations of the QGT. Hereby, we increase the complexity of the NQS by optimizing a DNN with an increasing amount of layers, where depth d represents the number of dense layers (with $\alpha = 1$), each followed by a sigmoid activation function. Such a DNN with d layers has $\mathcal{O}((d-1)L^2 + L)$ free parameters.

9.2 MPI for NetKet

We benchmark the scaling behavior of NETKET as a function of the computational resources available to perform parallel computations. Therefore, NETKET uses MPI for JAX through MPI4JAX [36]. The effectiveness of the MPI implementation is illustrated in fig. 4 for a VMC optimization with and without Stochastic Reconfiguration (SR). Throughout our analyses, we provide each MPI rank with 2 CPU cores. We introduce the speedup factor $\tau_r = \Delta t_1 / \Delta t_r$ where Δt_r refers to the time required to perform the computations on r MPI ranks. Similarly, we define τ_n as the speedup factor when using n nodes. The right panel of fig. 5 demonstrates that even on a single node, our MPI implementation can introduce significant speedups by running multiple Markov Chain samplers in parallel. This is consistent with the fact that JAX is not able to make use of multiple CPU cores unless working on very large matrices.

The performance of NETKET on challenging Hamiltonians, as well as its scalability with both system size and model complexity depends on the implementation details of the quantum geometric tensor [see eq. (17)] and its matrix multiplication with the gradient vector, as discussed in section 3.5. We therefore isolate these operations and benchmark
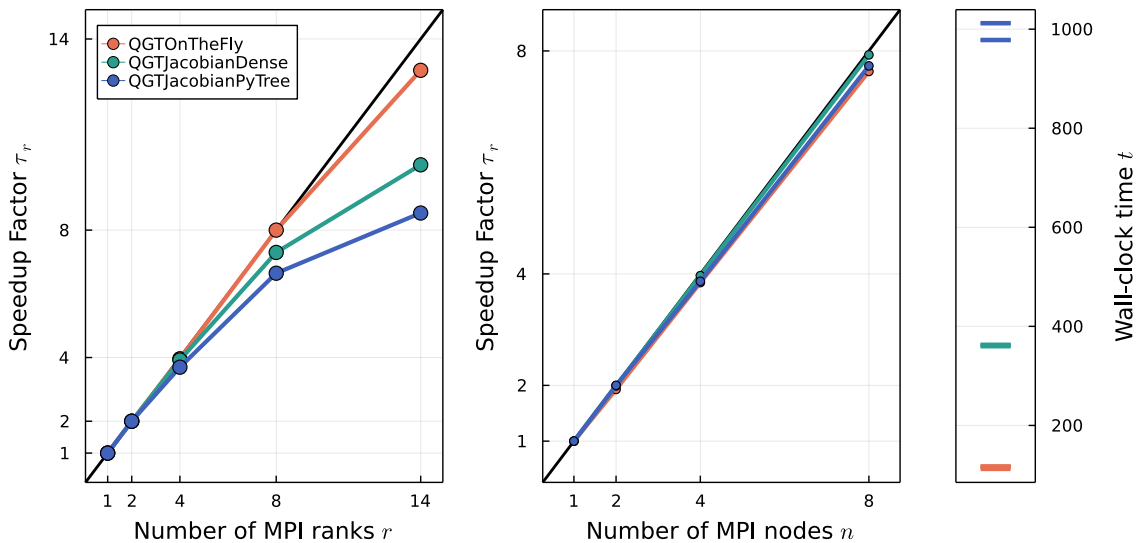
Figure 5: **(Left):** Speedup factor τ observed by increasing the number of MPI ranks r on a single node while keeping the problem size constant (strong scaling).**(Center):** Speedup factor observed by scaling across multiple nodes n, and scaling the number of samples accordingly (weak scaling). **(Right):** Absolute wall clock time in seconds for the runs reported in the other plots. We repeat 5 iterations of constructing the QGT and 1000 matrix multiplications with the gradient vector for the 2D Ising model with $L = 8$ and a DNN with 9 layers and complex weights. In the left panel, we keep the number of samples $N_{\mathrm{samples}} = 2^{14}$ constant, while in the right panel, we increase the number of samples to $2^{14} \times n$, while we correct the timing by the number of nodes n to show the speedup factor for a constant number of 2^{14} samples. We remark that while the speedup factor is resistant to changes in the network architecture, the absolute timing might favor one or another implementation depending on several details and can change depending on the architecture and problem at hand.

the QGT constructor, combined with 1000 matrix multiplications with the gradient vector (where the latter imitates many steps in the iterative solver). In fig. 5, we show the scaling behavior of these operations with respect to both the number of ranks (on a single node), and its scaling behavior with respect the number of nodes (thereby including communication over the Infiniband network). One can observe that the scaling behavior is close to optimal, especially when the number of samples per rank is sufficiently large.

9.3 Comparison with jVMC

We compare NETKET with jVMC [38], another open-source Python package supporting VMC optimization of NQS. Results are shown in fig. 4. We remark that since both NETKET and jVMC are JAX-based, performance on sampling, expectation values and gradients is roughly equivalent when using the same hyperparameters [38]. However, a performance difference arises in algorithms requiring the use of the QGT, such as TDVP or natural gradient (SR). Such difference will vanish in cases where the cost of solving the QGT linear system is small with respect to the cost of computing the energy and its gradient.

At the time of writing, jVMC only implements a singular-value decomposition (SVD) solver to invert the QGT matrix. The same type of solver can be used also in NETKET

		NetKet	**jVMC**
SVD solver	CPU (32 cores)	48	337
	GPU (×1)	24	44
	GPU (×2)	20	36
CG solver	CPU (32 cores)	86	N/A
	CPU (32 cores, MPIx16)	7.5	N/A
	GPU (×1)	3.9	N/A
	GPU (×2)	1.7	N/A

Table 5: Comparison of performance between NETKET and jVMC. All times are indicated in seconds and have been taken on a workstation with an `AMD Ryzen Threadripper 3970X 32-Core` processor and `2xNvidia RTX 3090` GPUs. Timings are for one VMC step using a complex-valued RBM model with hidden unit density of $\alpha = N_{\text{hidden}}/L = 1$ on the 1-dimensional TFIM model (37) with $L = 64$ sites. Other parameters are: 2^{14} total samples, 2^{10} independent Markov chains per GPU (or across all CPUs). Calculations for NETKET where performed using `QGTJacobianDense(holomorphic=True)`. NETKET multi-GPU calculations use CUDA-enabled MPI for inter-process communication, while jVMC uses JAX built-in mechanism. The run labeled with (MPI×16) is run on the same workstation but 16 MPI processes are used to better take advantage of the multiple cores of the processors.

(for a detailed discussion, see section 3.5). We limit the computations to models with less than 7000 parameters in order for the QGT to have less than $49 \cdot 10^3$ elements, which is approximately the maximum matrix size that can be diagonalized in reasonable time on the GPUs we have access to[21]. For that reason we chose the size of $L = 64$ spins.

As shown in table 5, NETKET outperforms by almost an order of magnitude jVMC on a 32-core CPU using SVD-based solvers. On GPU, jVMC requires significantly less computational time than on CPU, yet, NETKET outperforms jVMC by about 50% in a full VMC iteration. We remark that in this benchmark both packages scale poorly when going from a single GPU to two. This is because the diagonalization of the QGT, in this case the bottleneck, cannot be parallelized.

In order to scale efficiently to many GPUs, our QGT implementations can be combined with iterative solvers to scale up to potentially millions of parameters, as well as significantly larger system sizes. As expected from fig. 5, increasing the number of MPI ranks reduces the total time by a factor nearing the number of ranks (with eventually a saturation in the speedup). Notice also that the CG-solver becomes significantly more efficient on GPU.

---

[21]Using distributed linear-algebra libraries such as ScaLaPack [82], ELPA [83] or the recent [84] would allow us to avoid this barrier, however we are not aware of any Python binding for those libraries. Regardless, if those libraries exposed a distributed linear solver to Python, using it with NETKET would be as simple as using it as the linear solver for the QGT.

10   Discussion and conclusion

We have presented NETKET 3, a modular Python toolbox to study complex quantum-mechanical problems with machine learning-inspired tools. Compared to version 2 [34], the major new feature is the ability to define arbitrary neural-network ansätze for either wave functions or density matrices using the flexible JAX framework; we believe this makes NETKET much more useful to non-technical users. Another significant improvement is that NETKET is now completely modular. Users of NETKET 3 can decide to use only the neural-network architectures, the stochastic samplers, the quantum geometric tensor, or the operators without necessarily requiring the `VariationalState` interface, which is convenient but geared towards the most common applications of variational NQS. Care has been taken to ensure that the algorithms implemented can scale to very large systems and models with millions of parameters thanks to more efficient implementations of the geometric tensor and other algorithmic bottlenecks. Thanks to its JAX foundations, NETKET 3 can now also make effective use of GPU hardware, without any need for manual low-level programming for these platforms.

Even with all the new features that have been introduced with this version, there are many things that we would like to see integrated into NETKET in the future. To name a few: native support for fermionic systems; support for more general geometries in continuous systems; improvements to the dynamics submodule in order to support a wider variety of ODE solvers and more advanced regularization and diagnostic schemes [22,24]; new drivers for quantum state reconstruction [30] and overlap optimization [31]; a more general way to define arbitrary cost functions to be optimized. However, we think that our new JAX-based core is very welcoming to contributors, and we believe that this constitutes a solid foundation upon which to build in the future. Moreover, we are now explicitly committed to stability of the user-facing API, in order to make sure that code written today will keep working for a reasonable time, while we iterate and refine NETKET.

Since a project is only as big as its community, the most important developments are probably those related to documentation, learning material, and developing a community where users answer each other's questions in the spirit of open, shared science. We are taking steps to make all of this happen.

Acknowledgments

We are grateful to the authors and maintainers of JAX [5], FLAX [43], and OPTAX [63] for guidance and constructive discussions on how to best use their tools. F.V. thanks W. Bruinsma for adapting plum-dispatch to our needs and making life more Julian in the Python world. We also thank all the users who engaged with us on GitHub reporting bugs, sharing ideas, and ultimately helping us to produce a better framework.

Funding information.   This work is supported by the Swiss National Science Foundation under Grant No. 200021_200336. A.Sz. acknowledges support from the Oxford–ShanghaiTech collaboration project.

A   Details of group convolutions

A.1   Group convolutions and equivariance

As explained in section 5.2, GCNNs generate the action of every element g of the symmetry group G on a wave function $|\psi\rangle$, written as $|\psi_g\rangle = g|\psi\rangle$, which are then combined into a symmetric wave function using the projection operator (31). Amplitudes of the computational basis states $|\boldsymbol{\sigma}\rangle$ are related to one another in these wave functions as

$$\psi_g(\boldsymbol{\sigma}) = \langle\boldsymbol{\sigma}|g|\psi\rangle = \psi(g^{-1}\boldsymbol{\sigma}). \tag{38}$$

Therefore, feature maps inside the GCNN are indexed by group elements rather than lattice sites, and all layers must be *equivariant:* that is, if their input is transformed by a space-group symmetry, their output must be transformed the same way, so that (38) would always hold. Pointwise nonlinearities clearly fit this bill [72]; we now consider what linear transformations are allowed.

First, input feature maps naturally defined on lattice sites must be embedded into group-valued features:

$$\mathbf{f}_g = \sum_{\vec{r}} \mathbf{W}_{g^{-1}\vec{r}}\sigma_{\vec{r}} = \sum_{\vec{r}} \mathbf{W}_r\sigma_{g\vec{r}} = \sum_{\vec{r}} \mathbf{W}_{\vec{r}}(g^{-1}\boldsymbol{\sigma})_{\vec{r}}, \tag{39}$$

consistent with (38). We also see that the embedding (39) is equivariant: if the input is transformed by some symmetry operation u, the output transforms as

$$\sum_{\vec{r}} \mathbf{W}_{g^{-1}\vec{r}}\sigma_{u\vec{r}} = \sum_{\vec{r}} \mathbf{W}_{g^{-1}u^{-1}\vec{r}}\sigma_{\vec{r}} = \mathbf{f}_{ug}. \tag{40}$$

This layer is implemented in NETKET as `nk.nn.DenseSymm`.

To build deeper GCNNs, we also need to map group-valued features onto one another in an equivariant fashion. This is achieved by *group convolutional* layers, which transform input features as[22]

$$\phi_g = \sum_{h \in G} \mathbf{W}_{h^{-1}g}\mathbf{f}_h. \tag{41}$$

This layer is implemented in NETKET as `nk.nn.DenseEquivariant`. It is equivariant under multiplying with a group element u from the left:

$$\sum_{h \in G} \mathbf{W}_{h^{-1}g}\mathbf{f}_{uh} = \sum_{h \in G} \mathbf{W}_{h^{-1}ug}\mathbf{f}_h = \phi_{ug}, \tag{42}$$

which is consistent with how the embedding layer (39) is equivariant, cf. (40). Indeed, it can be composed with (39):

$$\phi_g = \sum_{h} \mathbf{W}_{h^{-1}g} \sum_{\vec{r}} \mathbf{W}'_{h^{-1}\vec{r}}\sigma_{\vec{r}} = \sum_{\vec{r}} \left(\sum_{h} \mathbf{W}_{h^{-1}} \mathbf{W}'_{h^{-1}g^{-1}\vec{r}} \right)\sigma_{\vec{r}} \equiv \sum_{\vec{r}} \mathbf{W}''_{g^{-1}\vec{r}}\sigma_{\vec{r}} \tag{43}$$

as the expression in brackets only depends on $g^{-1}\vec{r}$.

Finally, the output features of the last layer of the GCNN are turned into the wave functions $\psi_g(\boldsymbol{\sigma}) = \sum_i \exp\left(f_g^{(i)}\right)$ and projected on an irrep using (31) (we drop the irrelevant constant prefactor):

$$\psi(\boldsymbol{\sigma}) = \sum_{i,g} \chi_g^* \exp\left(f_g^{(i)}\right), \tag{44}$$

---

[22]Our convention differs from that of Ref. [72], which in fact implements group correlation rather than convolution. The two conventions are equivalent (the indexing of the kernels differs by taking the inverse of each element); we use convolutions to simplify the Fourier transform-based implementations of Sec. A.2.

where χ_g are the characters of the irrep. In addition to allowing nontrivial symmetries, our choice of summing a large number of terms in the ansatz appears to improve the stability of variational optimization for sign-problematic Hamiltonians [16, 47, 70, 71].

A.2 Fast group convolutions using Fourier transforms

The simplest implementation of a group convolutional layer is expanding each of the $f_{\text{in}}f_{\text{out}}$ kernels, containing $|G|$ entries, to a $|G| \times |G|$ matrix defined as

$$\tilde{W}_{g,h}^{(a,b)} = W_{g^{-1}h}^{(a,b)}, \tag{45}$$

where a, b index input and output features, respectively. The resulting tensor of size $f_{\text{in}}f_{\text{out}}|G|^2$ can then be contracted straightforwardly with the input features:

$$\phi_h^{(b)} = \sum_{a,g} \tilde{W}_{g,h}^{(a,b)} f_g^{(a)} \tag{46}$$

is equivalent to (41). Embedding layers (39) can be constructed analogously. This method is the easiest to interpret and code, serving as a useful check on other methods; however, the enlarged kernels require substantial amounts of memory, which already becomes a serious problem on modestly sized lattices and networks. Furthermore, evaluating a convolution using this method takes $O(f_{\text{in}}f_{\text{out}}|G|^2)$ time. NETKET implements two approaches to improve on this scaling.

The first approach uses *group Fourier transforms,* which generalize the usual discrete Fourier transform for arbitrary finite groups. The forward and backward transformations are defined by

$$\hat{f}(\rho) = \sum_{g \in G} f(g)\rho(g); \qquad\qquad f(g) = \frac{1}{|G|} \sum_{\rho} d_\rho \operatorname{Tr}\left[\hat{f}(\rho)\rho(g^{-1})\right]. \tag{47}$$

In the forward transformation, ρ is a representation of the group G; $f(g)$ is a function defined on group elements, while $\hat{f}(\rho)$ is a matrix of the same shape as the representatives $\rho(g)$. The sum in the backward transformation runs over all inequivalent irreps ρ, of dimension d_ρ, of the group. Since $\sum_\rho d_\rho^2 = |G|$, this transformation does not increase the amount of memory needed to store either inputs, outputs, or kernels.[23] Group convolutions can readily be implemented by multiplying the Fourier transform matrices (we drop feature indices for brevity):

$$\hat{\phi}(\rho) = \sum_g \phi_g \rho(g) = \sum_{g,h} f_h W_{h^{-1}g}\rho(h)\rho(h^{-1}g) = \hat{f}(\rho)\hat{W}(\rho). \tag{48}$$

To calculate a convolution using this approach, the input features are Fourier transformed $[O(f_{\text{in}}|G|^2)$ as there is no generic fast Fourier transform algorithm for group Fourier transforms], multiplied with the kernel Fourier transform for each irrep $[O(f_{\text{in}}f_{\text{out}}d_\rho^3)$ for an irrep of dimension $d_\rho]$, and the output is transformed back $[O(f_{\text{out}}|G|^2)]$, yielding the total runtime $O[(f_{\text{in}} + f_{\text{out}})|G|^2 + f_{\text{in}}f_{\text{out}}\sum_\rho d_\rho^3]$. In a large space group, most irreps are defined on a star of $|P|$ wave vectors (P is the point group) and thus have dimension $|P|$; accordingly, $\sum_\rho d_\rho^3 \approx |G||P|$.

The second approach, based on Ref. [72], exploits the fact that the translation group T is a normal subgroup of the space group G, so each $g \in G$ can be written as $t_g p_g$, where t_g is a translation and p_g is a fixed coset representative (in symmorphic groups, we can

---

[23]If some irreps cannot be expressed as matrices with real entries, the Fourier transform of real inputs/outputs/kernels is complex too, temporarily doubling the amount of memory used.

choose these to be point-group symmetries). Now, we can define the expanded kernels (we drop feature indices again to reduce clutter)

$$\tilde{W}_t^{(p_g,p_h)} \equiv W_{p_g^{-1}tp_h} \tag{49}$$

such that

$$\phi_h = \sum_g f_g W_{g^{-1}h} = \sum_{t_g,p_g} f_{t_g p_g} W_{p_g t_g^{-1} t_h p_h} \equiv \sum_{t_g,p_g} \tilde{f}_{t_g}^{(p_g)} \tilde{W}_{t_h-t_g}^{(p_g,p_h)}. \tag{50}$$

In the last form, we split the space-group feature map f into cosets of the translation group and observe that the latter is Abelian. In fact, the translation group is equivalent to the set of valid lattice vectors, so the sum over t_g in (50) is a standard convolution. Ref. [72] proposes to perform this convolution using standard cuDNN routines. However, we are usually interested in convolutions that span the entire lattice in periodic boundary conditions: these can be performed more efficiently using fast Fourier transforms (FFTs) as the Fourier transform of a convolution is the product of Fourier transforms. Therefore, we FFT the kernels $\tilde{W}$ and features $\tilde{f}$, contract them as appropriate, and FFT the result back:

$$\tilde{\phi}^{(b,p_h)} = \mathcal{F}^{-1}\left[\sum_{a,p_g} \mathcal{F}\left(\tilde{f}^{(a,p_g)} \right) \mathcal{F}\left(\tilde{W}^{(a,b;p_g,p_h)} \right) \right], \tag{51}$$

where the Fourier transform is understood to act on the omitted translation-group indices, and the Fourier transforms are multiplied pointwise.

Calculating a convolution in this approach involves $f_{\text{in}}|P|$ forward FFTs $[O(|T|\log|T|)$ each], $|T|$ tensor inner products $[O(f_{\text{in}}f_{\text{out}}|P|^2$ each], and $f_{\text{out}}|P|$ backward FFTs; as $|G| = |T||P|$, this yields a total of $O[(f_{\text{in}} + f_{\text{out}})|G|\log|T| + f_{\text{in}}f_{\text{out}}|G||P|]$. For large lattices, which bring out the better asymptotic scaling of FFTs, this improves significantly on the runtime of the group Fourier transform-based approach, especially in the pre- and postprocessing stages. By contrast, the group Fourier transform approach is better for large point groups, as it avoids constructing the $|P|^2$ reshaped kernels $\tilde{W}^{p_g p_h}$, which can be prohibitive for large lattices.

In practice, as both FFTs and group Fourier transforms involve steps more complicated than simple tensor multiplication, their performance is hard to assess beyond asymptotes, especially on a GPU. On CPUs, the FFT-based approach tends to be faster. On GPUs, computation time tends to scale sub-linearly with the number of operations so long as the process is efficiently parallelized. As all operations of the group Fourier transform implementation involve multiplications of large matrices, it can fully exploit the large GPU registers even with relatively few samples. By contrast, FFTs cannot be fully vectorized, meaning that larger batches are required to make full use of the computing power of the GPU. In practice therefore, the FFT-based approach may not perform better until most of the GPU memory becomes involved in evaluating a batch.

B   Implementation details of the quantum geometric tensor

In the following, we discuss our implementations of the quantum geometric tensor, introduced in section 3.5, in more detail. In particular, we show how the action of the quantum geometric tensor on a vector can be computed efficiently without storing the full matrix. Appendix B.1 introduces relevant automatic-differentiation concepts in general terms; the concrete algorithms used by `QGTJacobian` and `QGTOnTheFly` are discussed in appendices B.2 and B.3, respectively.

B.1   Jacobians and their products

We assume that our NQS is modeled by the scalar parametric function $f(s) = \ln \psi_\theta(s)$, where θ is a vector of variational parameters and s is a basis vector of the Hilbert space. Consistent with the notation of the main text, $O_j(s) = \partial_{\theta_j} \ln \psi_\theta(s)$ are the log-derivatives (10) of the NQS.

We also assume that f can be vectorized and evaluated for a batch of inputs $\{s_k\}_{k=1...N_s}$, yielding the vector $f_k = \ln \psi_\theta(s_k)$. The Jacobian of this function is therefore the matrix

$$J_{kl} = O_l(s_k) = \frac{\partial \ln \psi_\theta(s_k)}{\partial \theta_l}; \tag{52}$$

each row corresponds to the gradient of f evaluated at a different input s_k, so $k = 1 \ldots N_s$ and $l = 1 \ldots N_{\text{parameters}}$.

The Jacobian matrix can be computed in JAX with `jax.jacrev(log_wavefunction)(s)`, which returns a matrix.[24] However, it is often not needed to have access to the full Jacobian: for example, when computing the gradient (20) of the variational energy, we only need the product of the Jacobian with a vector, namely $\Delta E_{\text{loc}}(s_k) = \tilde{H}(s_k) - \mathbb{E}[\tilde{H}]$.

A vector can be contracted with the Jacobian along its dimension corresponding to either parameters or outputs:

- *Jacobian–vector products* (Jvp), $\tilde{\mathbf{v}} = J\mathbf{v}$, can be computed using forward-mode automatic differentiation;

- *vector–Jacobian products* (vJp), $\tilde{\mathbf{v}} = \mathbf{v}^T J$, can be computed through backward-mode automatic differentiation (backward propagation).

Modern automatic-differentiation frameworks like that of JAX implement primitives that evaluate Jvp and vJp, and construct higher-level functions such as `jax.grad` or `jax.jacrev` on top of those functions; that is, one can extract the best performance from JAX by making use of vJp and Jvp as much as possible [85].

B.2   QGTJacobian

Writing the estimator (18) of the quantum geometric tensor explicitly in terms a finite number of samples s_k, we obtain

$$\begin{aligned}
G_{ij} &= \mathbb{E}\left[O_i^* O_j\right] - \mathbb{E}\left[O_i\right]^* \mathbb{E}\left[O_j\right] \\
&\approx \frac{1}{N_s} \sum_{k=1}^{N_s} O_i(s_k)^* O_j(s_k) - \frac{1}{N_s^2} \left(\sum_{k=1}^{N_s} O_i(s_k)\right)^* \left(\sum_{k=1}^{N_s} O_j(s_k)\right) \\
&= \frac{1}{N_s} \sum_{k=1}^{N_s} \left(O_i(s_k) - \sum_{k=1}^{N_s} \frac{O_i(s_k)}{N_s}\right)^* \left(O_j(s_k) - \sum_{k=1}^{N_s} \frac{O_j(s_k)}{N_s}\right) \\
&= \frac{1}{N_s} \sum_{k=1}^{N_s} \left(J_{ki} - \sum_{k=1}^{N_s} \frac{J_{ki}}{N_s}\right)^* \left(J_{kj} - \sum_{k=1}^{N_s} \frac{J_{kj}}{N_s}\right) \\
&= \frac{1}{N_s} \sum_{k=1}^{N_s} \left(\Delta J_{ki}\right)^* \left(\Delta J_{kj}\right),
\end{aligned} \tag{53}$$

---

[24]More precisely, it returns a PyTree with a structure similar to the PyTree that stores the parameters; each leaf gains an additional dimension of length N_s.

where we have defined the centered Jacobian $\Delta J_{ki} \equiv J_{ki} - \sum_{k=1}^{N_s} \frac{J_{ki}}{N_s}$. In matrix notation, this is equivalent to

$$G = \frac{\Delta J^\dagger}{\sqrt{N_s}} \frac{\Delta J}{\sqrt{N_s}}. \tag{54}$$

The Jacobian-based implementation of the quantum geometric tensor computes[25] and stores[26] the full Jacobian matrix J_{kl} for the given samples upon construction. Then, QGT–vector products $\tilde{\mathbf{v}} = G\mathbf{v}$ are computed without finding the full matrix G, in two steps:

$$\Delta\mathbf{w} = \frac{\Delta J}{\sqrt{N_s}}\mathbf{v}; \qquad\qquad \tilde{\mathbf{v}} = \frac{\Delta J^\dagger}{\sqrt{N_s}}\Delta\mathbf{w} = \left(\frac{\Delta J}{\sqrt{N_s}}\Delta\mathbf{w}^\dagger\right)^\dagger; \tag{55}$$

the final form has the advantage that the Hermitian transpose of a vector is simply its conjugate. Evaluating eq. (55) is usually less computationally expensive than constructing the full quantum geometric tensor.

B.3 QGTOnTheFly

In some cases one might have so many parameters or samples that it is impossible to store the full Jacobian matrix in memory. In that case, we still evaluate a set of equations similar to eq. (55), but without pre-computing the full Jacobian, only using vector–Jacobian and Jacobian–vector products.

It would be impractical to perform a vJp using the centered Jacobian; however, eq. (18) can be rewritten as $G_{ij} = \mathbb{E}\left[O_i^*\left(O_j - \mathbb{E}\left[O_j\right]\right)\right]$, which yields

$$G = \frac{1}{N_s}J^\dagger\Delta J, \tag{56}$$

where we have substituted one of the two centered Jacobians with a plain Jacobian. Then, we note that the centered-Jacobian–vector product can be expressed as

$$\Delta\mathbf{w} \equiv \Delta J\,\mathbf{v} = \left(J - \frac{\mathbf{1}^T J}{N_s}\right)\mathbf{v} = \mathbf{w} - \frac{\mathbf{1}^T\mathbf{w}}{N_s}, \tag{57}$$

where $\mathbf{1}^T$ is a row vector all entries of which are 1, used to express averaging the columns of the Jacobian in the matrix formalism. Therefore, `QGTOnTheFly` performs the following calculations:

$$\mathbf{w} = \frac{1}{N_s}J\mathbf{v}; \qquad\qquad \Delta\mathbf{w} = \mathbf{w} - \langle\mathbf{w}\rangle; \qquad\qquad \tilde{\mathbf{v}} = J^\dagger\,\Delta\mathbf{w}, \tag{58}$$

where the first and the last step are implemented with `jax.vjp` and `jax.jvp`, respectively.

References

[1] A. Krizhevsky, I. Sutskever and G. E. Hinton, *Imagenet classification with deep convolutional neural networks*, In F. Pereira, C. J. C. Burges, L. Bottou and K. Q. Weinberger, eds., *Advances in Neural Information Processing Systems*, vol. 25. Curran Associates, Inc. (2012).

---

[25]The full Jacobian can be computed row by row, performing vJps with vectors that have a single nonzero entry. In practice, as the network is evaluated independently for all samples in a batch, each of these products requires us to back-propagate the network for only one sample at a time.

[26]To speed up the evaluation of (55), we actually store $\Delta J/\sqrt{N_s}$.

[2] C. Chen, O. Li, D. Tao, A. Barnett, C. Rudin and J. K. Su, *This looks like that: Deep learning for interpretable image recognition*, In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox and R. Garnett, eds., *Advances in Neural Information Processing Systems*, vol. 32, pp. 8928–8939. Curran Associates, Inc. (2019).

[3] D. W. Otter, J. R. Medina and J. K. Kalita, *A survey of the usages of deep learning in natural language processing*, arXiv preprint arXiv:1807.10854 (2018).

[4] Y. Sun, N. B. Agostini, S. Dong and D. Kaeli, *Summarizing cpu and gpu design trends with product data* (2019), arXiv:1911.11313.

[5] R. Frostig, M. J. Johnson and C. Leary, *Compiling machine learning programs via high-level tracing*, Systems for Machine Learning (2018).

[6] J. Bezanson, A. Edelman, S. Karpinski and V. B. Shah, *Julia: A fresh approach to numerical computing*, SIAM review **59**(1), 65 (2017).

[7] G. Carleo, I. Cirac, K. Cranmer, L. Daudet, M. Schuld, N. Tishby, L. Vogt-Maranto and L. Zdeborová, *Machine learning and the physical sciences*, Reviews of Modern Physics **91**(4) (2019), doi:10.1103/revmodphys.91.045002.

[8] G. Carleo and M. Troyer, *Solving the quantum many-body problem with artificial neural networks*, Science **355**(6325), 602 (2017), doi:10.1126/science.aag2302.

[9] D.-L. Deng, X. Li and S. Das Sarma, *Quantum entanglement in neural network states*, Phys. Rev. X **7**, 021021 (2017), doi:10.1103/PhysRevX.7.021021.

[10] D. Wu, L. Wang and P. Zhang, *Solving statistical mechanics using variational autoregressive networks*, Phys. Rev. Lett. **122**, 080602 (2019), doi:10.1103/PhysRevLett.122.080602.

[11] R. Kaubruegger, L. Pastori and J. C. Budich, *Chiral topological phases from artificial neural networks*, Physical Review B **97**(19) (2018), doi:10.1103/physrevb.97.195136.

[12] I. Glasser, N. Pancotti, M. August, I. D. Rodriguez and J. I. Cirac, *Neural-network quantum states, string-bond states, and chiral topological states*, Phys. Rev. X **8**, 011006 (2018), doi:10.1103/PhysRevX.8.011006.

[13] K. Choo, G. Carleo, N. Regnault and T. Neupert, *Symmetries and many-body excitations with neural-network quantum states*, Phys. Rev. Lett. **121**, 167204 (2018), doi:10.1103/PhysRevLett.121.167204.

[14] K. Choo, G. Carleo, N. Regnault and T. Neupert, *Symmetries and many-body excitations with neural-network quantum states*, Physical review letters **121**(16), 167204 (2018).

[15] T. Vieijra, C. Casert, J. Nys, W. De Neve, J. Haegeman, J. Ryckebusch and F. Verstraete, *Restricted boltzmann machines for quantum states with non-abelian or anyonic symmetries*, Physical review letters **124**(9), 097201 (2020).

[16] A. Szabó and C. Castelnovo, *Neural network wave functions and the sign problem*, Phys. Rev. Research **2**, 033075 (2020), doi:10.1103/PhysRevResearch.2.033075.

[17] T. Vieijra and J. Nys, *Many-body quantum states with exact conservation of non-abelian and lattice symmetries through variational monte carlo*, Phys. Rev. B **104**, 045123 (2021), doi:10.1103/PhysRevB.104.045123.

[18] M. Bukov, M. Schmitt and M. Dupont, *Learning the ground state of a non-stoquastic quantum Hamiltonian in a rugged neural network landscape*, SciPost Phys. **10**, 147 (2021), doi:10.21468/SciPostPhys.10.6.147.

[19] S. Czischek, M. Gärttner and T. Gasenzer, *Quenches near ising quantum criticality as a challenge for artificial neural networks*, Physical Review B **98**(2) (2018), doi:10.1103/physrevb.98.024311.

[20] G. Fabiani and J. Mentink, *Investigating ultrafast quantum magnetism with machine learning*, SciPost Physics **7**(1) (2019), doi:10.21468/scipostphys.7.1.004.

[21] I. L. Gutiérrez and C. B. Mendl, *Real time evolution with neural-network quantum states* (2021), 1912.08831.

[22] M. Schmitt and M. Heyl, *Quantum many-body dynamics in two dimensions with artificial neural networks*, Phys. Rev. Lett. **125**, 100503 (2020), doi:10.1103/PhysRevLett.125.100503.

[23] G. Fabiani, M. Bouman and J. Mentink, *Supermagnonic propagation in two-dimensional antiferromagnets*, Physical Review Letters **127**(9) (2021), doi:10.1103/physrevlett.127.097202.

[24] D. Hofmann, G. Fabiani, J. H. Mentink, G. Carleo and M. A. Sentef, *Role of stochastic noise and generalization error in the time propagation of neural-network quantum states* (2021), arXiv:2105.01054.

[25] F. Vicentini, A. Biella, N. Regnault and C. Ciuti, *Variational neural-network ansatz for steady states in open quantum systems*, Phys. Rev. Lett. **122**, 250503 (2019), doi:10.1103/PhysRevLett.122.250503.

[26] A. Nagy and V. Savona, *Variational quantum monte carlo method with a neural-network ansatz for open quantum systems*, Phys. Rev. Lett. **122**, 250501 (2019), doi:10.1103/PhysRevLett.122.250501.

[27] M. J. Hartmann and G. Carleo, *Neural-network approach to dissipative quantum many-body dynamics*, Phys. Rev. Lett. **122**, 250502 (2019), doi:10.1103/PhysRevLett.122.250502.

[28] C.-Y. Park and M. J. Kastoryano, *Expressive power of complex-valued restricted boltzmann machines for solving non-stoquastic hamiltonians* (2021), 2012.08889.

[29] N. Astrakhantsev, T. Westerhout, A. Tiwari, K. Choo, A. Chen, M. H. Fischer, G. Carleo and T. Neupert, *Broken-symmetry ground states of the heisenberg model on the pyrochlore lattice*, Physical Review X **11**(4) (2021), doi:10.1103/physrevx.11.041021.

[30] G. Torlai, G. Mazzola, J. Carrasquilla, M. Troyer, R. Melko and G. Carleo, *Neural-network quantum state tomography*, Nature Physics **14**(5), 447 (2018), doi:10.1038/s41567-018-0048-5.

[31] B. Jónsson, B. Bauer and G. Carleo, *Neural-network states for the classical simulation of quantum computing* (2018), 1808.05232.

[32] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org (2015).

[33] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf *et al.*, *Pytorch: An imperative style, high-performance deep learning library*, In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox and R. Garnett, eds., *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc. (2019).

[34] G. Carleo, K. Choo, D. Hofmann, J. E. Smith, T. Westerhout, F. Alet, E. J. Davis, S. Efthymiou, I. Glasser, S.-H. Lin, M. Mauri, G. Mazzola *et al.*, *NetKet: A machine learning toolkit for many-body quantum systems*, SoftwareX **10**, 100311 (2019), doi:10.1016/j.softx.2019.100311.

[35] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne and Q. Zhang, *JAX: composable transformations of Python+NumPy programs*, URL http://github.com/google/jax (2018).

[36] D. Häfner and F. Vicentini, *mpi4jax: Zero-copy MPI communication of JAX arrays*, Journal of Open Source Software **6**(65) (2021), doi:10.21105/joss.03419.

[37] W. Bruinsma, *Plum: Multiple dispatch in python*, doi:10.5281/zenodo.5774909, URL https://github.com/wesselb/plum (2021).

[38] M. Schmitt and M. Reh, *Jvmc: Versatile and performant variational monte carlo leveraging automated differentiation and gpu acceleration* (2021), arXiv:2108.03409.

[39] M.-D. Choi, *Completely positive linear maps on complex matrices*, Linear Algebra and its Applications **10**(3), 285 (1975), doi:10.1016/0024-3795(75)90075-0.

[40] A. Jamiołkowski, *Linear transformations which preserve trace and positive semidefiniteness of operators*, Reports on Mathematical Physics **3**(4), 275 (1972), doi:10.1016/0034-4877(72)90011-0.

[41] S. K. Lam, A. Pitrou and S. Seibert, *Numba: A llvm-based python jit compiler*, In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pp. 1–6 (2015).

[42] K. Kreutz-Delgado, *The complex gradient operator and the cr-calculus* (2009), arXiv:0906.4835.

[43] 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*, URL http://github.com/google/flax (2020).

[44] T. Hennigan, T. Cai, T. Norman and I. Babuschkin, *Haiku: Sonnet for JAX*, URL http://github.com/deepmind/dm-haiku (2020).

[45] R. Jastrow, *Many-body problem with strong forces*, Phys. Rev. **98**, 1479 (1955), doi:10.1103/PhysRev.98.1479.

[46] D. A. Huse and V. Elser, *Simple variational wave functions for two-dimensional heisenberg spin-½ antiferromagnets*, Phys. Rev. Lett. **60**, 2531 (1988), doi:10.1103/PhysRevLett.60.2531.

[47] C. Roth and A. H. MacDonald, *Group convolutional neural networks improve quantum state accuracy*, arXiv preprint arXiv:2104.05085 (2021).

[48] O. Sharir, Y. Levine, N. Wies, G. Carleo and A. Shashua, *Deep autoregressive models for the efficient variational simulation of many-body quantum systems*, Phys. Rev. Lett. **124**, 020503 (2020), doi:10.1103/PhysRevLett.124.020503.

[49] G. Torlai and R. G. Melko, *Latent space purification via neural density operators*, Phys. Rev. Lett. **120**, 240503 (2018), doi:10.1103/PhysRevLett.120.240503.

[50] F. Becca and S. Sorella, *Quantum Monte Carlo Approaches for Correlated Systems*, Cambridge University Press, doi:10.1017/9781316417041 (2017).

[51] W. K. Hastings, *Monte carlo sampling methods using markov chains and their applications* **57**(1), 97 (1970), doi:10.1093/biomet/57.1.97.

[52] M. V. Berry, *The quantum phase, five years after*, Geometric phases in physics pp. 7–28 (1989).

[53] G. Fubini, *Sulle metriche definite da una forme hermitiana*, Atti R. Ist. Veneto Sci. Lett. Arti **63**, 502 (1904).

[54] E. Study, *Kürzeste Wege im komplexen Gebiet*, Mathematische Annalen **60**(3), 321 (1905), doi:10.1007/BF01457616.

[55] J. Stokes, J. Izaac, N. Killoran and G. Carleo, *Quantum Natural Gradient*, Quantum **4**, 269 (2020), doi:10.22331/q-2020-05-25-269.

[56] S. Sorella, *Green function monte carlo with stochastic reconfiguration*, Physical review letters **80**(20), 4558 (1998).

[57] S.-I. Amari, *Natural gradient works efficiently in learning*, Neural computation **10**(2), 251 (1998).

[58] C.-Y. Park and M. J. Kastoryano, *Geometry of learning neural quantum states*, Physical Review Research **2**(2) (2020), doi:10.1103/physrevresearch.2.023232.

[59] R. Horn, *Matrix analysis*, Cambridge University Press, Cambridge New York, ISBN 978-0-521-54823-6 (2012).

[60] I. Contreras, E. Ercolessi and M. Schiavina, *On the geometry of mixed states and the fisher information tensor*, Journal of Mathematical Physics **57**(6), 062209 (2016), doi:10.1063/1.4954328.

[61] H. Weimer, *Variational principle for steady states of dissipative quantum many-body systems*, Phys. Rev. Lett. **114**, 040402 (2015), doi:10.1103/PhysRevLett.114.040402.

[62] X. Yuan, S. Endo, Q. Zhao, Y. Li and S. C. Benjamin, *Theory of variational quantum simulation*, Quantum **3**, 191 (2019), doi:10.22331/q-2019-10-07-191.

[63] M. Hessel, D. Budden, F. Viola, M. Rosca, E. Sezener and T. Hennigan, *Optax: composable gradient transformation and optimisation, in jax!*, URL http://github.com/deepmind/optax (2020).

[64] P. Kramer and M. Saraceno, eds., *Geometry of the Time-Dependent Variational Principle in Quantum Mechanics*, Springer Berlin Heidelberg, doi:10.1007/3-540-10579-4 (1981).

[65] J. Haegeman, J. I. Cirac, T. J. Osborne, I. Pižorn, H. Verschelde and F. Verstraete, *Time-dependent variational principle for quantum lattices*, Physical Review Letters **107**(7) (2011), doi:10.1103/physrevlett.107.070601.

[66] G. Carleo, F. Becca, M. Schiró and M. Fabrizio, *Localization and glassy dynamics of many-body quantum systems*, Scientific Reports **2**(1) (2012), doi:10.1038/srep00243.

[67] H. Breuer and F. Petruccione, *The Theory of Open Quantum Systems*, OUP Oxford, ISBN 9780199213900, doi:10.1093/acprof:oso/9780199213900.001.0001 (2007).

[68] J. Stoer and R. Bulirsch, *Introduction to Numerical Analysis*, Springer New York, doi:10.1007/978-0-387-21738-3 (2002).

[69] S. d'Ascoli, L. Sagun, J. Bruna and G. Biroli, *Finding the Needle in the Haystack with Convolutions: on the benefits of architectural bias*, arXiv:1906.06766 (2019).

[70] Y. Nomura, *Helping restricted boltzmann machines with quantum-state representation by restoring symmetry*, Journal of Physics: Condensed Matter **33**(17), 174003 (2021).

[71] Y. Nomura and M. Imada, *Dirac-type nodal spin liquid revealed by machine learning*, arXiv preprint arXiv:2005.14142 (2020).

[72] T. Cohen and M. Welling, *Group equivariant convolutional networks*, In *International conference on machine learning*, pp. 2990–2999. PMLR (2016).

[73] V. Heine, *Group Theory in Quantum Mechanics: An Introduction to Its Present Usage*, vol. 91 of *International Series in Natural Philosophy*, Pergamon, ISBN 9781483152004, doi:10.1016/C2013-0-01646-5 (1960).

[74] M. I. Aroyo, ed., *Space-group symmetry*, vol. A of *International Tables for Crystallography*, Wiley, New York, 2 edn., ISBN 978-0-470-97423-0, doi:10.1107/97809553602060000114 (2016).

[75] J. Dixon, *Computing irreducible representations of groups*, Math. Comp. **24**, 707 (1970), doi:10.1090/S0025-5718-1970-0280611-6.

[76] W. Burnside, *The Theory of Groups*, Cambridge University Press (1911).

[77] C. J. Bradley and A. P. Cracknell, *The Mathematical Theory of Symmetry in Solids: Representation Theory for Point Groups and Space Groups*, Clarendon Press, ISBN 9780198519201 (1972).

[78] F. Ferrari and F. Becca, *Gapless spin liquid and valence-bond solid in the j 1-j 2 heisenberg model on the square lattice: Insights from singlet and triplet excitations*, Physical Review B **102**(1), 014417 (2020).

[79] P. W. Anderson, *An approximate quantum theory of the antiferromagnetic ground state*, Phys. Rev. **86**, 694 (1952), doi:10.1103/PhysRev.86.694.

[80] J. Johansson, P. Nation and F. Nori, *QuTiP: An open-source python framework for the dynamics of open quantum systems*, Computer Physics Communications **183**(8), 1760 (2012), doi:10.1016/j.cpc.2012.02.021.

[81] J. Johansson, P. Nation and F. Nori, *QuTiP 2: A python framework for the dynamics of open quantum systems*, Computer Physics Communications **184**(4), 1234 (2013), doi:10.1016/j.cpc.2012.11.019.

[82] J. Choi, J. Dongarra, R. Pozo and D. Walker, *ScaLAPACK: a scalable linear algebra library for distributed memory concurrent computers*, In *[Proceedings 1992] The Fourth Symposium on the Frontiers of Massively Parallel Computation*. IEEE Comput. Soc. Press, doi:10.1109/fmpc.1992.234898 (1992).

[83] T. Auckenthaler, V. Blum, H.-J. Bungartz, T. Huckle, R. Johanni, L. Krämer, B. Lang, H. Lederer and P. Willems, *Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations*, Parallel Computing **37**(12), 783 (2011), doi:10.1016/j.parco.2011.05.002.

[84] M. Gates, J. Kurzak, A. Charara, A. YarKhan and J. Dongarra, *SLATE*, In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, doi:10.1145/3295500.3356223 (2019).

[85] A. Griewank and A. Walther, *Evaluating derivatives : principles and techniques of algorithmic differentiation*, Society for Industrial and Applied Mathematics, Philadelphia, PA, ISBN 978-0898716597 (2008).