**Abstract**

We introduce *tqix.pis*, a library of *tqix* for executing various algorithms in large-scale quantum simulation platforms. The program emulates basic functions of a quantum circuit, including initialization qubits, quantum gates, and measurements. It utilizes the collective processes in ensembles of two-level systems to reduce the dimension, and facilitates the simulation time with multi-core processors and Graphics Processing Units. The library is thus programmable for different large-scale quantum simulation platforms, such as trapped ions, ultracold atoms in optical lattices, Rydberg atom arrays in optical tweezers, and nitrogen-vacancy centers. It is applicable for examining spin squeezing, variational quantum squeezing, quantum phase transition, many-body quantum dynamics, and other quantum algorithms.

**Keywords**: quantum toolbox; quantum simulation platform; spins squeezing, variational squeezing, quantum phase transition.

**PROGRAM SUMMARY/NEW VERSION PROGRAM SUMMARY**

*Program Title*: *tqix.pis*: A toolbox for large-scale quantum simulation platforms.

*Licensing provisions*: GPLv3

*Programming language*: Python

*External routines*: NumPy, SciPy, Sympy, Matplotlib, Sklearn, pyTorch (for GPUs users).

*Nature of problems*: *tqix.pis* is a library in *tqix*, an open-source program providing some convenient tools for quantum measurement, quantum tomography, quantum metrology, and others. The library allows executing quantum circuits with a large number of qubits assisting from multi-core processors and graphics processing units (GPUs).

*Solution method*: Execute quantum circuits with a large number of qubits. It generates a quantum register (state) by an ensemble of spin-1/2 particles under the collective process, evolves the state under applied quantum gates, and provides measurement results on the Dicke basis.

*Reasons for new version*: In this version, we added a new library *tqix.pis* for large-scale quantum simulation platforms. All functions from the previous version are intact.

*Additional comments including Restrictions and Unusual features*: Official website: https://vqisinfo.wixsite.com/tqix

1. Introduction

The toolbox *tqix* is a Python-based simulation package for quantum measurement and applications in quantum metrology and quantum tomography [1]. The package allows for designing quantum systems (state and operators) and performing quantum measurements. It embeds various conventional measurement operators, including Pauli, Stoke, MUB-POVM, and SIC-POVM, with two backends to simulate the measurement results. The applications in quantum metrology and quantum tomography were reported so far.

In this version, we introduce a new library called *tqix.pis* for programmable large-scale quantum simulation platforms. The program executes basic functions of a quantum circuit with a sequence of initialization qubits, the action quantum gates, and measurements. It utilizes the collective processes in ensembles of spin-1/2 particles, and represents an N-particle state in the Dicke basis with $O(N^2)$-dimension, smaller than the product $2^N$-dimension [2]. This is the backbone for large-scale simulation with low computation cost compared to other existing libraries. The library supports parallelizing multi-core processors and Graphics Processing Units (GPUs) to facilitate the running time. In terms of application, it is suitable for studying spin squeezing [3], variational quantum squeezing [4, 5], quantum phase transition [6], many-body quantum dynamics [7], and other quantum algorithms. The library is widely adaptable for various quantum simulation platforms, including trapped ions [8], ultracold atoms in optical lattices [9], Rydberg atom arrays in optical tweezers [10], and nitrogen-vacancy (NV) centers [10].

This paper is organized as follows. Section 2 introduces quantum computing with collective processes in an ensemble of two-level systems, the cornerstone for large-scale computing. Section 3 details the structure of *tqix.pis* library and its benchmark results are given in Section 4. Section 5 is devoted to applications on spin squeezing and quantum phase transition.
A brief conclusion is given in Section 3. The full code for generating figures in the paper can be found in the Appendices.

2. Quantum computing with collective processes in ensembles of two-level systems

2.1. Collective processes in ensembles of two-level systems

Consider an ensemble of $N$ two-level particles characterized by the single and collective angular momentum operators $J_{\alpha}^{(s)} = \frac{1}{2} \sigma_\alpha$, and $J_{\alpha} = \sum_{\alpha} \sigma_\alpha$, respectively, where $\sigma_\alpha$, $\alpha = \{ x, y, z \}$, are Pauli matrices. The joint Hilbert space of the ensemble is a composite $\mathcal{H}_E = \mathcal{H}^{(s)} \otimes \cdots \otimes \mathcal{H}^{(s)}$ with $\dim(\mathcal{H}_E) = 2^N$. A generic mixed state is given by a $(2^N \times 2^N)$-matrix as

$$\rho = \sum_{m_1, m_2, \ldots, m_N} \rho_{m_1, m_2, \ldots, m_N} |m_1, m_2, \ldots, m_N\rangle \langle m_1, m_2, \ldots, m_N| ,$$

where the product basis is $|m_1, m_2, \ldots, m_N\rangle = |m_1\rangle \otimes |m_2\rangle \otimes \cdots \otimes |m_N\rangle$, with $m_\alpha = \pm \frac{1}{2}$ are eigenvalues of $J_{\alpha}^{(s)}$.

Under the collective processes [11, 12], the product basis is represented by an irreducible representation (irrep) basis $|j,m, i\rangle$, where $j \leq N/2$ is the total angular momentum with $J_{\min} = \text{mod}(N/2)/2$, $J_{\max} = N/2$, and $|m| \leq j$. The quantum number $i = 1, \ldots, d_j$ distinguishes $d_j$ degenerate irreps, where

$$d_j = \frac{N!(2j+1)}{(N-2j)!(N/2+j+1)!}$$

is the number of ways to combine $N$ particles that gets the total angular momentum $j$ [13]. The quantum state is recast in the irrep basis as

$$\rho = \sum_{j,m,i} \rho_{j,m,i} |j,m,i\rangle \langle j,m,i| .$$

The state $\rho$ is permutation invariant (PI) if it does not change under a permutation operator $P_\pi$ of a permutation $\pi$, i.e., $P_\pi \rho P_\pi^\dagger = \rho \forall \pi$ [14]. Decompose $P_\pi$ into the multiplicity subspaces $\mathcal{H}_j \otimes \mathcal{H}_j$ following the Schur-Weyl duality [15] as $P_\pi = \sum_{j=2}^{N/2} I_{\mathcal{H}_j} \otimes P_\pi$, where $\sum_{j=2}^{N/2} I_{\mathcal{H}_j} \otimes \mathcal{H}_j = \mathcal{H}_E$, $\dim(\mathcal{H}_j) = 2j + 1$ and $\dim(\mathcal{H}_j') = d_j$, that $P_\pi$ only acts on the irrep space $\mathcal{H}_j$, which causes a general permutation symmetry, i.e., $\rho_{j,m,i} = \rho_{j,m,i'} \forall i, i' \in [1, d_j]$, then a permutation-invariant state is given by

$$\rho_{\pi} = \sum_{j=2}^{N/2} \rho_{j} \otimes I_{\mathcal{H}_j} = \sum_{j,m,i} \rho_{j,m,i} |j,m,i\rangle \langle j,m,i| .$$

Hence, we can ignore the identical irrep label $i$ and recast the state in a collective form as [11]

$$\rho_{\pi} = \sum_{j=2}^{N/2} \rho_{j} \otimes I_{\mathcal{H}_j} = \sum_{j,m,i} \rho_{j,m,i} |j,m,i\rangle \langle j,m,i| ,$$

where the effective amplitude $\rho_{j,m,i}$ and the effective density matrix elements $|j,m,i\rangle \langle j,m'|$ are given by

$$\rho_{j,m,i} = \frac{1}{d_j} \sum_{i'} |j,m,i\rangle \langle j,m',i'| .$$

The effective basis $|j,m\rangle$ is known as the Dicke basis [2], i.e., $|j,m\rangle = \frac{1}{\sqrt{d_j}} \sum_{i=1}^{d_j} |j,m,i\rangle$, that is the eigenstates of the collective angular momentum operators

$$J_j^2 |j,m\rangle = j(j+1) |j,m\rangle ,$$

$$J_j |j,m\rangle = m |j,m\rangle ,$$

$$J_j \pm |j,m\rangle = \sqrt{j(j+1)\pm m^2} |j,m\rangle ,$$

where $J_j = J_x^2 + J_y^2 + J_z^2$ and $J_x = J_x \pm i J_y$. The collective angular momentum operators obey the commuting relation

$$[J_\alpha, J_\beta] = i e_{\alpha\beta\gamma} J_\gamma ,$$

where $e_{\alpha\beta\gamma} \in \{ x, y, z \}$, and $e_{\alpha\beta\gamma}$ is the Levi-Civita symbol.

Under the collective processes, the Hilbert space $\mathcal{H}_E$ reduces to a collective subspace $\mathcal{H}_C \sim O(N^2) \subset \mathcal{H}_E$ as [11, 12]

$$\dim(\mathcal{H}_C) = \sum_j \dim(\mathcal{H}_j) = \left\{ \begin{array}{ll} \frac{(N+3)(N+1)}{4}, & \text{for odd } N, \\ \frac{(N+2)^2}{4}, & \text{for even } N. \end{array} \right.$$
where around the R wise by a polar angle \( \theta \).

2.3. Collective operators

An important collective quantum state is the ground state where all spins are down, i.e., \( |g\rangle^N = |\downarrow \cdots \downarrow \rangle \). In the Dicke basis, it gives \( |g\rangle^N = |N/2 \rangle \) and its density matrix locates in the bottom-right position of the first irrep block. Another one is the excited state, \( |e\rangle^N = | \uparrow \cdots \uparrow \rangle \), which is given by \( |e\rangle^N = \sqrt{N \over 2} \{|N/2, N/2\rangle \) and the density matrix locates on the top-left of the first irrep block. An entangle GHZ state is a superposition of the ground and excited states as

\[
|\text{GHZ}\rangle = \frac{1}{\sqrt{2}} \left( |N/2, N/2\rangle + |N/2, -N/2\rangle \right).
\]

Besides, a literature well-known coherent spin state (CSS) is a product state that resembles a coherent state of a classical harmonic oscillator \([24, 27]\). It explicitly expends in terms of a linear combination of \( |N/2, m\rangle \) elements as

\[
|\theta, \phi\rangle_{\text{CSS}} = \sum_{m=-N/2}^{N/2} \sqrt{ \binom{N/2 + m}{m} } \left( \cos \frac{\theta}{2} e^{i m \phi} \right) \left( \sin \frac{\theta}{2} e^{-i m \phi} \right) |N/2, m\rangle,
\]

where \( 0 \leq \theta \leq \pi \) is the polar angle, and \( 0 \leq \phi \leq 2\pi \) denotes the azimuth angle in the spherical coordinates.

We emphasize that all these states obey the symmetry and are thus represented in the subspace \( \mathcal{H}_{N/2} \). However, under noise, such as decoherence, symmetry states decay from the maximum \( j_{\max} \)-irrep block and transfer to the lower blocks, resulting in the full collective subspace \( \mathcal{H}_C \). (See Ref. [13] for an example.)

2.3. Collective operators

Collective operators are unitary transformations over all particles of the ensemble. We first introduce the rotation operator denoted by \( R_{\theta}(\theta, \phi) \). It rotates the original state counterclockwise by a polar angle \( \theta \) around the \( xy \) plane and then by an angle \( \phi \) around the \( z \) axis \([3]\)

\[
R_{\theta}(\theta, \phi) = e^{-i J_{x} \theta},
\]

where \( J_{x} = (J_{x}, J_{y}, J_{z}) \), \( n = (\sin \phi, \cos \phi, 0) \), and \( J_{x} = J \cdot n \). For example, a CSS state can be prepared from the rotation \( R_{\theta}(\theta, \phi) \) of the ground state, i.e., \( |\theta, \phi\rangle_{\text{CSS}} = R_{\theta}(\theta, \phi) |N/2, -N/2\rangle \) \([27]\). Specific cases of the rotation operator are rotation around the \( \alpha \) axis, \( \alpha \in \{x, y, z\} \) as

\[
R_{\alpha}(\theta) = e^{-i J_{\alpha} \theta}.
\]

These rotations allow us to transform quantum states everywhere in the spherical representation but not deform the states.

Hereafter, let us introduce other classes of transformations based on the nonlinear interactions that can deform the quantum states. They include one-axis twisting (OAT) \([19, 28, 34]\), two-axis countertwisting (TAT) \([19, 55, 58]\), twist-and-turn (TNT) \([39, 42]\), and an important global Mølmer-Sørensen operator (GMS) \([43, 47]\).

An OAT transformation along the \( \alpha \) axis is defined by

\[
U_{\text{OAT}} = e^{-i \alpha J_{\alpha}^2},
\]

where through the paper, we use \( h = 1 \) is the nonlinearity interparticle interaction, and \( \tilde{\tau} \) is the time. It leads to a rotation proportional to \( J_{\alpha} \), \( \alpha \in \{x, y, z\} \) and twist the quantum fluctuations of the ensemble. Similarly, a TAT transformation is defined by

\[
U_{\text{TAT}} = e^{-i \alpha (J_{\alpha}^2 - J_{\alpha}^z)},
\]

where \( \{\alpha, \beta\} \in \{x, y, z, +, -\} \). Next, for the TNT transformation, we have

\[
U_{\text{TNT}} = e^{-i \alpha (J_{\alpha}^2 - \Omega J_{\alpha}^z)},
\]

where \( \Omega \) is the linearly coupling strength. Set \( \Lambda = N \chi / \Omega \), then for \( \chi \ll \Omega \), the TNT transformation reduces to the OAT one. Finally, the definition of a global Mølmer-Sørensen transformation is an entangling gate that operates on all particles of the ensemble \([48]\)

\[
U_{\text{GMS}} = e^{-i \alpha (J_{x} \cos \phi + J_{z} \sin \phi)}.
\]

It encompasses the pairwise and is given by two angles \( \theta \) and \( \phi \). For \( \theta = \pi/2 \), the GMS gate is maximally entangling, i.e., GHZ state \([43]\).

These transformations deform the collective states and result in the entangled or squeezed effect in the ensemble systems and thus exhibit various applications from quantum-enhanced metrology \([49, 52]\) to quantum sensors, and atomic clocks \([5, 53]\).

Besides, other transformations for creating highly quantum-enhanced states include light-to-atoms transformation \([54, 59]\), quantum nondemolition measurement (QND) \([49, 60, 65]\), and adiabatically quantum phase transition \([64, 68]\), to name a few.

2.4. Depolarizing noise in the collective systems

Let \( \rho = \sum_{j, j', m, m'} \rho_{j_{\alpha} j_{\alpha'} m m'} |j, m, j', m'\rangle \langle j, m, j', m'| \) is the initial quantum state. Through a depolarizing channel applied to all particles in the ensemble, the state transforms to \([69]\)

\[
\rho_1 = (1 - \epsilon) \rho + \frac{4 \epsilon}{3 N} \sum_{n=1}^{N} \left( J_{n}^{(0)} \rho J_{n}^{(0)} + J_{n}^{(0)} \rho J_{n}^{(0)} + J_{n}^{(0)} \rho J_{n}^{(0)} \right),
\]

where \( 0 \leq \epsilon \leq 1 \) is the noisy probability. Substituting

\[
J_{n}^{(0)} = \frac{1}{2} \left( J_{n}^{+} + J_{n}^{-} \right), \quad J_{n}^{(0)} = \frac{1}{2} \left( J_{n}^{+} - J_{n}^{-} \right)
\]

into Eq. \((20)\) and set

\[
\rho' = \frac{4}{3 N} \sum_{n=1}^{N} \left[ \frac{1}{2} \left( J_{n}^{(0)} \rho J_{n}^{(0)} + J_{n}^{(0)} \rho J_{n}^{(0)} \right) + J_{n}^{(0)} \rho J_{n}^{(0)} \right].
\]
that normalized by $\rho^\text{norm} = \rho^{-1} \rho / \text{tr}(\rho^{-1} \rho')$. Finally, we rewrite Eq. (20) as
\[ \rho_1 = (1 - \epsilon) \rho + \epsilon \rho^\text{norm} \]  
(23)

Following Refs. [11, 12, 70, 71], we derive the terms $\sum_{n=1}^N p'(j, m) \langle j, m' | j, m \rangle^{(n)}$ for all $[k, l] \in \{+,-,z\}$, and compute $\rho_1$ straightforwardly.

Under the noise, the symmetry is broken, resulting in a deviation from $|j, m\rangle$ to $|j \pm 1, m \pm 1\rangle$. Thus, the final state spans in the collective Hilbert space $\mathcal{H}_C$ with all $j$-irrep blocks for all $J_{\text{min}} \leq j \leq J_{\text{max}}$.

In Sec. 3, we introduce the tqix.pis library for large-scale simulation with the collective processes in the ensemble systems based on the theoretical framework described here.

3. Large-scale simulation with collective processes in ensembles of two-level systems

The simulation is based on the tqix package, a quantum toolbox for quantum measurement, metrology, and tomography [1]. Here, we develop the tqix.pis library for simulating various functions in large-scale quantum simulation platforms.

3.1. Features of the library
- Assist a large number of qubits with the collective effective of the ensemble.
- Assist fast simulation with parallelizing multi-core processors and Graphics Processing Units (GPUs).

3.2. Structure of the program
A basic quantum circuit includes a quantum register (quantum state), quantum gates, and measurements. In tqix.pis, the computational basis is the Dicke basis $|j, m\rangle$, represented by a sparse matrix. The quantum register is a density matrix in the Dicke basis, initially prepared in the ground state, i.e., $\rho_0 = |\frac{1}{2}, -\frac{1}{2}\rangle \langle \frac{1}{2}, -\frac{1}{2}|$. This is a symmetry collective state spans in the subspace $\mathcal{H}_N/2$, and thus it has the dimension of $(N + 1) \times (N + 1) \propto 2^N \times 2^N$. This is a highlighted feature of the library that allows us to compute a large number of qubits. Similarly, quantum gates are collective operators, e.g., $K$, that apply to the quantum register and transform it into the evolved state
\[ \rho = K \rho_0 K^\dagger \]  
(24)

A list of build-in quantum gates and their syntax is given in Tab. 1. These gates are represented by sparse matrices with the same dimension as the quantum register. Finally, the measurement on the Dicke basis gives the probability
\[ P(j, m) = \text{tr}(\rho \cdot |j, m\rangle \langle j, m|) = \langle j, m| \rho |j, m\rangle. \]  
(25)

To mimic the experiential probability, we apply the cdf backend in tqix [1].

The following example code shows how to execute a simple quantum circuit with a rotation gate $R_\theta$ using the circuit, RN, and measure functions in the library tqix.pis:

```python
from tqix import *
from tqix.pis import *
import numpy as np

N = 50  #qubits
qc = circuit(N)  #create circuit
qc.RN(-np.pi/2,np.pi/4)  #apply RN
prob = qc.measure(num_shots=1000)  #measure

to get state information
psi = qc.state  #sparse matrix
psi = qc.state.toarray()  #full matrix

The results are given in figure 2. The left figure is the Husimi visualization for the circuit’s state initially prepared in the ground state (the South pole). Under the rotation gate $R_N(-\pi/2, \pi/4)$, the state rotates around the $y$ axis at an angle $\theta = -\pi/2$ and then rotates around the $z$ axis at an angle $\phi = \pi/4$. The final state becomes a CSS state, where its probabilities measured in the Dicke basis are exhibited in the right figure. (See the full code in Appendix A)

To compute the expectation value of a given observable, such as $J_z$, we execute the expval function as
\[ \text{exp}J_z = qc.expval(\{J_z\}) \]

We emphasize that the observable must obey the collective process, such as the collective angular momentum operators $|J_\alpha\rangle$, for $\alpha \in \{x, y, z, +, -\}$. The program supports these observables $|J_x\rangle$ include $\{J_x\}$, $|J_y\rangle$, $|J_z\rangle$, $|J_{\text{plus}}\rangle$, $|J_{\text{minus}}\rangle$, and $|J_{\text{plus2}}\rangle$, $|J_{\text{minus2}}\rangle$. For an arbitrary self-defined observable, such as a linear combination of these listed observables, the users can compute manually from the quantum state and the self-defined observable. For example, the following script computes the expectation value of $J_x + J_y + J_z$

\[
\begin{align*}
\text{Jx} &= qc.Jx() \\
\text{Jy} &= qc.Jy() \\
\text{Jz} &= qc.Jz() \\
\text{Js} &= \text{Jx} + \text{Jy} + \text{Jz} \\
\text{psi} &= qc.state \\
\text{expectJ} &= \text{np.trace(psi@Js)}
\end{align*}
\]

Finally, the noise induces wherever we apply a quantum gate onto the system. To include the noise, we use the noise option
in the quantum gate function, which is the noise parameter $\epsilon$. For example, we add the noise with $\epsilon = 0.01$ into the RN gate as following

```python
qc.RN(-np.pi/2, np.pi/4, noise = 0.01)
```

### 3.3. Additional options for running multi-core processors and GPUs

By default, the program executes on CPUs with one processor. However, it also allows for running parallel multi-core processors, which assists the running time. To execute the program in multi-core processors, we add the option `num_process` into the circuit. This is an integer number stands for the number of processors. For example, one can create a quantum register with 50 qubits and run it with 25 processors by using the following command

```python
qc = circuit(N = 50, num_process = 25)
```

Similarly, the library supports running on GPU devices, which can outperform the CPUs in some cases. To execute the program with GPUs, we add the option `use_gpu` into the circuit. For example,

```python
qc = circuit(N = 50, use_gpu = True)
```

Notable that to run with GPUs, the users must turn off the option `num_process` or leave it as one. In GPUs, the sparse-matrix structure will convert into the tensor structure using `pyTorch` library.

### 4. Benchmark results

For benchmarking the library, we consider the execution time versus the numbers of qubit $N$. The benchmarking circuit is shown in Fig. 3(a), which consists of a set of rotation gates $\{R_x, R_y, R_z\}$ with $L$ times and the measurement of the final state. The running time versus $N$ for $L = 3$ is shown in Fig. 3(b). For the noiseless case (filled diamond), the circuit executes up to $N = 200$, consuming around 3 seconds with 1 processor CPU (see the inset Fig. 3(b)). The complexity is $O(N^3)$. Under the presence of noise, the running time rapidly increases with $N$. The complexity is $O(N^7)$ for CPU regardless of the number of processors. Nevertheless, by using multi-processing on CPUs, we can reduce the running time of adding noise by $p$ times, with $p$ being the computer’s maximum processors. Hence, the running time for 25 processors (open diamond) is faster than the 1 processor (open star).

We next evaluate the performance of the GPU (NVIDIA V100) (dashed-dot curve) and compare it with the 25-processor CPU. The complexity performing on the GPU is $O(N^7)$. For $N < 25$, it offers a better running time, while increasing $N$ will slow down the running time. This result relies on the different matrix structures in these two devices, i.e., the CPU uses sparse matrices, whereas the GPU uses tensor matrices. For small $N$, the tensor dimension is small, the GPU thus displays its advantage overcome the CPU, and for larger $N$, the tensor dimension increases while the sparse matrices are unaffected. As a result, the performance of GPU gradually overcomes the GPU. The full code for producing Fig. 3 is given in Appendix B.

The time complexities of a quantum circuit with various cases are listed in Tab. 2. We evaluate the complexity by analyzing the number of operations performed on the code’s statements. Details are given in Appendix C.

Finally, we compare `tqix.pis` library with other libraries including qsim [72], cirq [73], qulacs [74], yao [75], qsun [76], quest [77], pennylane [78], qiskit [79], projectQ [80].

| Name   | Syntax                        | Description                                                                 |
|--------|-------------------------------|-----------------------------------------------------------------------------|
| $R_x$  | $RX(\theta, *\text{args})$   | Rotate the quantum state at an angle $\theta$ around the $x$ axis, i.e., $RX(\theta) = e^{-iJ_x\theta}$  |
| $R_y$  | $RY(\theta, *\text{args})$   | Rotate the quantum state at an angle $\theta$ around the $y$ axis, i.e., $RY(\theta) = e^{-iJ_y\theta}$  |
| $R_z$  | $RZ(\theta, *\text{args})$   | Rotate the quantum state at an angle $\theta$ around the $z$ axis, i.e., $RZ(\theta) = e^{-iJ_z\theta}$  |
| $R_{\phi}$ | $RN(\theta, \phi, *\text{args})$ | Rotate the quantum state at an angle $\theta$ and $\phi$ around the $z$ axis, i.e., $RN(\theta, \phi) = e^{-iJ_z\theta \sin \phi - iJ_z\phi \cos \phi}$. |
| OAT    | OAT($\theta, \text{gate\_type}, *\text{args}$) | One-axis twisting gate, c.g., OAT($\theta$, $x$) = $e^{-iJ_x\theta}$ |
| TAT    | TAT($\theta, \text{gate\_type}, *\text{args}$) | Two-axis twisting gate, c.g., TAT($\theta$, $\alpha\beta$) = $e^{-iJ_{\alpha\beta}\theta}$ |
| TNT    | TNT($\theta, \Lambda, \text{gate\_type}, *\text{args}$) | Twist and Turn gate, c.g., TNT($\theta, \Lambda$, $\alpha\beta$) = $e^{-iJ_{\alpha\beta}\theta}$ |
| GMS    | GMS($\theta, \phi, *\text{args}$) | Global Mølmer-Sørensen gate, c.g., GMS($\theta, \phi$) = $e^{-iJ_{\alpha\beta}\theta \sin \phi - iJ_{\alpha\beta}\phi \cos \phi}$. |

Table 1: List of basic quantum gates built in the library `tqix.pis`. |


The complexities of a quantum circuit for different cases as shown in Table 2.

| Symmetry state | Noiseless $O(N^n)$ | Noisy $O(N^n)$ |
|----------------|---------------------|----------------|
| Collective state | $O(N^n)$ | $O(N^n)$ |

and qylu [81]. The results are shown in Fig. 4. We consider a quantum circuit that generates the maximum entanglement GHZ state. In tqix.pis, a GHZ state is created by applying the GMS($\pi/2,0$) gate onto the initial ground state. In other libraries, the circuit encompasses one Hadamard gate and $N - 1$ CNOT gates. For $N < 30$ as shown in the figure, tqix.pis remains the running time lower than $10^2$ seconds while the time for others increases exponentially versus $N$. See the full code in [Appendix D]. The tqix.pis represents quantum states and operators by sparse matrices; thus, it consumes a certain running time and less depend on (small) $N$. The other libraries represent quantum states by state vectors, where the running time depends on $2^N$. Our method thus work well for large-scale quantum simulation.

5. Applications

5.1. Spin squeezing

Definition. There are several definitions for the squeezing parameter in the literature [8, 15, 29, 82]. The two well-known spin squeezing parameters are given by Kitagawa and Ueda [15] and Wineland [82]. According to Kitagawa and Ueda, the spin-squeezing state (SSS) redistributes quantum fluctuations between two noncommuting observables while preserving the minimum uncertainty product [11, 12]. The corresponding squeezing parameter is given by

$$\xi_S^2 = \frac{2}{N} \left( J_n^2 + J_n^2 \right) \pm \sqrt{(J_n^2 - J_n^2)^2 + 4 \text{cov}^2(J_n, J_n)},$$

(26)

where

$$n_2 = (-\sin \phi, \cos \phi, 0);$$
$$n_1 = (\cos \theta \cos \phi, \cos \theta \sin \phi, -\sin \theta),$$

and
$$\theta = \arccos \left( \frac{J_y}{|J|} \right);$$

with $|J| = \sqrt{(J_x^2 + (J_y)^2 + (J_z)^2)}$. These notations are given in the spherical coordinate. The covariant $\text{cov}(J_n, J_n)$ is given by

$$\text{cov}(J_n, J_n) = \frac{1}{2}(|J_n|) \delta^2.$$  
(27)

Similarly, the Wineland squeezing parameter is defined by [82]

$$\xi_W^2 = \left( \frac{N}{2|J|} \right) \xi_S^2.$$  
(28)

When $\xi_S^2 < 1$, the system state is squeezed.

In tqix.pis, to compute the squeezing parameters, we call get_xi_2_s and get_xi_2_r functions for $\xi_S^2$ and $\xi_R^2$, respectively. For example, we generate a quantum circuit with $N = 50$ qubits and calculate its squeezing parameters as follows:

```python
qc = circuit(N = 50)
xiiS2 = get_xi_2_S(qc)
xiiR2 = get_xi_2_R(qc)
```
Spin squeezing by nonlinear gates. Spin squeezing state (SSS) can be created from the CSS by applying a nonlinear gate such as OAT, TNT, and TAT. Here, we illustrate the spin squeezing for these three cases using tqi.pis as follows. First, we apply the rotation gate \( R\theta(\pi/2, 0) \) onto the initial quantum register to transform the ground state into the CSS state. Then, we apply different nonlinear gates, including OAT(\( \theta \)), TNT(\( \theta \)) and TAT(\( \theta, N/2 \)). We calculate the squeezing parameters of the final states in these cases and show the result in figure 5. In the left figure, we plot \( \xi_3^2(\text{dB}) \) and \( \xi_4^2(\text{dB}) \). There exhibit squeezing for all cases, i.e., \( \xi_3^2(\text{dB}) < 0 \), and there exists a lower bound for each squeezing parameter at a certain \( \theta \). The \( \xi_3^2 \)'s curves are always higher than \( \xi_4^2 \) since \( \frac{\xi_3^2}{\xi_4^2} \geq |J| \) in Eq. (28). The corresponding Husimi visualizations are shown in the right figure for several values of \( \theta \), in which displace different types of squeezing due to different types of nonlinear gate (OAT, TNT, and TAT). We plot the results for \( N = 100 \) qubits. See the full code in Appendix E.

Spin squeezing by variational algorithms. It can be seen from the previous results, these squeezing parameters reflect the presence of an optimal SSS when tuning to the right \( \theta \), so it is reasonable to use optimization algorithms to find optimal circuit’s parameters corresponding to the minimum of squeezing function. Here, we introduce a quantum-classical hybrid variational scheme. In the quantum part, we sequentially apply a set of OAT(\( \theta_1 \)), TNT(\( \theta_2 \)), and TAT(\( \theta_3, N/2 \)), into a CSS circuit. We then measure the final state on the Dicke basis and send the results into the classical part to compute the cost function. For generating SSS, we apply the cost function to be the squeezing parameter

\[
C(\theta) = \xi_3^2(\theta),
\]

where \( \theta = (\theta_1, \theta_2, \theta_3) \). We solve the following optimization problem

\[
\theta^* = \arg \min_{\theta} C(\theta),
\]

by using gradient-based optimizers including gradient descent (GD), Adam, and quantum natural gradient descent (QNG) (see Appendix F for the details.) The gradient of \( C \) with respect to \( \theta \) is derived by a finite difference (findiff) approximation for sparse matrix type (use in CPUS)

\[
\frac{dC}{d\theta_i} \approx \frac{C(\theta_i + \epsilon) - C(\theta_i - \epsilon)}{2\epsilon},
\]

which small \( \epsilon \), and by an automatic differential (autodiff) mechanism for tensor type (use in GPUs). Different from findiff, the autodiff works on the chain rule of the differentials with two modes, forward accumulation and reverse accumulation [83, 84].

We choose the initial \( \theta \) randomly and iteratively optimize the cost function as shown in the results in the main Fig. 6. Among the three optimizes, the QNG converges fastest while the Adam results in a less stable near the optimal value, and the GD fluctuates in the first several iterations then towards the optimal value without fluctuating. This observation reveals the natural features of these optimizers [85, 87]. The optimal values \( \theta^* \) and the optimal cost function are given in Tab. 3. The results are plotted at \( N = 100 \), and the custom learning rate \( \eta \) for each optimize are shown in the main Fig. 6. See the full code in Appendix G.

In addition, we compare the running time through the iteration for two types of devices: CPU and GPU. It can be seen from the inset Fig. 6 that the GPU runs faster than the CPU. The reason is that the autodiff in GPUs is more optimized than the findiff approximation in CPUs. The autodiff only calculates the cost function once and uses the reverse accumulation for calculating the gradient. Whereas the findiff needs to calculate the partial derivative for every \( \theta_i \), where for each \( \theta_i \), the cost function is computed twice: once for \( \theta_i + \epsilon \) and once for \( \theta_i - \epsilon \) (see Eq. (31).) Moreover, the findiff is difficult to convert, leading to inefficient code and intractable to perform higher order deriva-

**Table 3: Optimal parameters \( \theta \) and its respective cost values for each optimizer.**

| Algorithm | \( \theta_1 \) | \( \theta_2 \) | \( \theta_3 \) | \( C(\theta) \) |
|-----------|---------------|---------------|---------------|----------------|
| GD        | -0.06292      | 0.07942       | -0.02435      | 0.02273        |
| ADAM      | -0.03632      | 0.10609       | 0.00115       | 0.02622        |
| QNG       | -0.08166      | 0.11887       | 0.01525       | 0.03895        |
5.2. Quantum phase transition

Quantum phase transition (QPT) is a transition between different quantum phases, such as the change in the ground state phases under the variation of magnetic fields (see [6] for reference.) Let us consider the Lipkin-Meshkov-Glick (LMG) model for an ensemble spins system interacting through a spin-spin infinite-range exchange coupling $\lambda$ and exposing under an effective transverse field $h$. The interaction Hamiltonian is given by [88]

$$H_{\text{LMG}} = -2hJ_z - \frac{1}{N}(J_x^2 - J_y^2).$$

Due to the spin-spin interaction, the system may exhibit the quantum phase transition under the adiabatic dynamics [89–91].

We illustrate the QPT using $\text{tqix.pis}$ library. Let us set the ratio $r = h/\lambda$ for a fixed $\lambda$, and recast Eq. (32) as

$$H_{\text{LMG}} = -2\lambda[J_z + \frac{1}{2}(J_x^2 - J_y^2)].$$

Its unitary evolution is given in terms of quantum gates as

$$U_{\text{LMG}} = e^{-iH_{\text{LMG}}} \approx e^{-i\lambda \rho J_z} e^{-i\lambda \rho (J_x^2 - J_y^2)} = \text{RZ}(\Lambda r) \text{TAT}(\Lambda/N, \rho, \lambda),$$

where we ignored higher order terms in $t$, i.e., the interaction time is short, and used $\Lambda = -2\lambda t$.

The results are shown in Fig. 7 for the first- and second-order of the expectation values, i.e., $\langle J_z \rangle$, $\langle J_x^2 \rangle$, and $\langle J_y^2 \rangle$. We fixed $\Lambda = -0.2$. In all cases, we observe the phase transition as the change in expectation values oscillate after the critical point. Detailed code is given in Appendix H.

6. Conclusion

We developed $\text{tqix.pis}$, an object-orientation library for large-scale quantum simulation platforms. We applied the collective process in the ensemble of qubits and thus reduced the dimension of the whole system. The library allows for simulating quantum circuits with collective states and collective gates suitable for some quantum simulation platforms such as trapped ions, ultracold atoms in optical lattices, Rydberg atom arrays in optical tweezers, and nitrogen-vacancy centers. It integrates parallelizing multi-core processors and Graphics Processing Units, making it run faster than other libraries for many-qubit systems. We finally showed two representative applications on quantum squeezing and quantum phase transition. Our library is a practical tool for simulation collective phenomena at a large scale with low computational cost, and we look for near future applications in many-body quantum dynamics.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have ap-
Acknowledgment

This work is supported by the Ensemble Grants for Early Career Researches in Tohoku University.

Author contributions

L.B.H. proposed the theoretical framework, designed the model, and scrutinized the results. N.T.V. wrote and implemented the code and performed the numerical calculations. N.T.C. benchmarked and compared various libraries. V.T.N.H. proposed and analyzed the application on quantum phase transition. All authors discussed and contributed to the final manuscript.

Appendix A. An example code for employing a quantum circuit in qutip.pis

In tqix.pis to create a quantum circuit, we call the circuit function, where circuit.state is a sparse matrix representation for the initial quantum state. The action of quantum gates on the state is given by circuit.gate.name, while the measurement is executed by circuit.measure(num_shots).

```python
from tqix import *
from tqix.pis import *
import numpy as np

numq = 50

#call the initial circuit
circuit = circuit(numq)
psi = circuit.state

#print psi
print(psi) #sparse matrix
print(psi.toarray()) #full matrix

#apply the rotation gate RN on the circuit
circuit.RN(-np.pi/2, np.pi/4)
psi2 = circuit.state

#visualize state
THETA = [0, np.pi]
PHI = [0, 2*np.pi]
husimi_spin_3d(psi.toarray()+psi2.toarray(), THETA, PHI, cmap = cmindex(1), dirname="./FIG", fname = "husimi_sphere.eps", view=(0,0))

#get probability
prob = circuit.measure(num_shots=1000)

#plot figure
from matplotlib import pyplot as plt
x = np.arange(0, numq+1, 1)
plt.bar(x, prob)
plt.savefig("./FIG/Pjm.eps")
```

Appendix B. Full code for producing Fig. 3 in the main text

```python
from tqix.pis import *
from tqix import *
import time
import matplotlib.pyplot as plt

N_max = 200 #max number of qubits
L = 3 #number of layers

#run time benchmark on noiseless qubit system
noiseless_time_qubits = []
for N in range(1, N_max):
    qc = circuit(N)
    start = time.time()
    for _ in range(L):
        qc.RX(np.pi/3)
        qc.RY(np.pi/3)
        qc.RZ(np.pi/3)
    noiseless_time = time.time()-start
    noiseless_time_qubits.append(noiseless_time)

#run time benchmark on noise qubit system with #1 process for simulating noise
noise_time_qubits = []
for N in range(1, N_max):
    qc = circuit(N)
    start = time.time()
    for _ in range(L):
        qc.RX(np.pi/3, noise=0.05)
        qc.RY(np.pi/3, noise=0.05)
        qc.RZ(np.pi/3, noise=0.05)
    noise_time = time.time()-start
    noise_time_qubits.append(noise_time)

#run time benchmark on noise qubit system with #multi-processes for simulating noise
mp_noise_time_qubits = []
for N in range(1, N_max):
    qc = circuit(N, num_process=25)
    start = time.time()
    for _ in range(L):
        qc.RX(np.pi/3, noise=0.05)
        qc.RY(np.pi/3, noise=0.05)
        qc.RZ(np.pi/3, noise=0.05)
    mp_noise_time = time.time()-start
    mp_noise_time_qubits.append(mp_noise_time)

#run time benchmark on noise qubit system with #gpu for simulating noise
gpu_noise_time_qubits = []
```
for N in range(1,N_max):
    qc = circuit(N,use_gpu=True)
    start = time.time()
for _ in range(L):
    qc.RX(np.pi/3,noise=0.05)
    qc.RY(np.pi/3,noise=0.05)
    qc.RZ(np.pi/3,noise=0.05)
gpu_noise_time = time.time()-start
gpu_noise_time_qubits.append(gpu_noise_time)

plt.plot(range(1, len(gpu_noise_time_qubits)+1), gpu_noise_time_qubits, label = "GPU: NVIDIA V100")
plt.savefig("./lib_benchmark.eps")
plt.show()
if os.path.isdir('./TNT '):
    pass
else:
    os.mkdir('./TNT ')

if os.path.isdir('./GMS '):
    pass
else:
    os.mkdir('./GMS ')

if os.path.isdir('./TAT '):
    pass
else:
    os.mkdir('./TAT ')

# OAT
OAT_xi_2_S = []
OAT_xi_2_R = []
for theta in angles:
    qc = circuit(N)
    qc.RN(np.pi/2,0)
    qc.OAT(theta,"Z")
    OAT_xi_2_S.append(10*np.log10(np.real(get_xi_2_S(qc))))
    OAT_xi_2_R.append(10*np.log10(np.real(get_xi_2_R(qc))))

# plot sphere
for theta in [(0.0, 0.02, 0.04, 0.06, 0.08, 0.1)]:
    qc = circuit(N)
    qc.RN(np.pi/2,0)
    qc.OAT(theta,"Z")
    husimi_spin_3d(qc.state.toarray(), THETA ,PHI , cmap = cmindex(1),dirname="./OAT",
    fname = str(theta)+"husimi_sphere.eps",view =(-90,0))

#GMS
GMS_xi_2_S = []
GMS_xi_2_R = []
for theta in angles:
    qc = circuit(N)
    qc.GMS(theta,phi)
    GMS_xi_2_S.append(10*np.log10(np.real(get_xi_2_S(qc))))
    GMS_xi_2_R.append(10*np.log10(np.real(get_xi_2_R(qc))))

# plot sphere
for theta in [(0.0, 0.02, 0.04, 0.06, 0.08, 0.1)]:
    qc = circuit(N)
    qc.GMS(theta,phi)
    husimi_spin_3d(qc.state.toarray(), THETA ,PHI , cmap = cmindex(1),dirname="./GMS",
    fname = str(theta)+"husimi_sphere.eps",view =(-90,0))

# TAT
TAT_xi_2_S = []
TAT_xi_2_R = []
for theta in angles:
    qc = circuit(N)
    qc.RN(np.pi/2,0)
    qc.TAT(theta,"ZY")
    TAT_xi_2_S.append(10*np.log10(np.real(get_xi_2_S(qc))))
    TAT_xi_2_R.append(10*np.log10(np.real(get_xi_2_R(qc))))

# plot sphere
for theta in [(0.0, 0.02, 0.04, 0.06, 0.08, 0.1)]:
    qc = circuit(N)
    qc.RN(np.pi/2,0)
    qc.TAT(theta,"ZY")
    husimi_spin_3d(qc.state.toarray(), THETA ,PHI , cmap = cmindex(1),dirname="./TAT",
    fname = str(theta)+"husimi_sphere.eps",view =(-90,0))

ax = plt.gca()
ax.plot(angles, OAT_xi_2_S,'c-o',label=r'$10\log_{10}(\xi^{2}_{S})-OAT$')
ax.plot(angles, TNT_xi_2_S,'r-s',label=r'$10\log_{10}(\xi^{2}_{S})-TNT$')
ax.plot(angles, TAT_xi_2_S,'g-*',label=r'$10\log_{10}(\xi^{2}_{S})-TAT$')
ax.plot(angles, GMS_xi_2_S,'y-o',label=r'$10\log_{10}(\xi^{2}_{S})-GMS$')
ax.plot(angles, OAT_xi_2_R,'c--o',label=r'$10\log_{10}(\xi^{2}_{R})-OAT$')
ax.plot(angles, TNT_xi_2_R,'r--s',label=r'$10\log_{10}(\xi^{2}_{R})-TNT$')
ax.plot(angles, TAT_xi_2_R,'g--*',label=r'$10\log_{10}(\xi^{2}_{R})-TAT$')
ax.plot(angles, GMS_xi_2_R,'y--',label=r'$10\log_{10}(\xi^{2}_{R})-GMS$')
Appendix F. Optimizers used in variation quantum squeezing circuit

In a variational circuit, the quantum part is parameterized by $\theta$ that will be iteratively updated by, such as gradient-based optimizers. Here, we consider the gradient descent (GD), Adam [94], and Quantum natural gradient (QNG) [85] optimizers.

The GD computes new parameters via

$$\theta^{t+1} = \theta^t - \eta \nabla C(\theta), \quad (F.1)$$

where $\nabla C(\theta)$ is the gradient of $C(\theta)$, and $\eta$ is the learning rate. It is simple, but the coverage is low, and one must choose a proper learning rate to have the best result. Whereas, the Adam computes new parameters by

$$\theta^{t+1} = \theta^t - \eta \frac{m_t}{v_t} \epsilon, \quad (F.2)$$

where $m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla \theta C(\theta), v_t = \beta_2 v_{t-1} + (1 - \beta_2) \nabla^2 \theta C(\theta), m_t = \hat{m}_t / (1 - \beta_1), v_t = \hat{v}_t / (1 - \beta_2)$, with $\eta = 0.01, \beta_1 = 0.8, \beta_2 = 0.999$ and $\epsilon = 10^{-10}$. The Adam optimizer automatically adapts the learning rate and fast coverage, but it is noisy near the optimal point. Finally, the QNG is better than other optimizers but requires more computational cost regards to quantum circuits. It is given by

$$\theta^{t+1} = \theta^t - \eta g^* \nabla C(\theta), \quad (F.3)$$

where $g^*$ is the pseudo-inverse of a Fubini-Study metric tensor $g$ [85]. Detailed for deriving the Fubini-Study metric tensor can be seen from Ref. [87].

Appendix G. Full code for producing Fig. 6 in the main text

```python
from tqix.pis import *
from tqix import *
import numpy as np
from matplotlib import pyplot as plt
import torch
import numpy as np

def cost_function(theta, use_gpu=False):
    qc = circuit(N, use_gpu=use_gpu)
    qc.RN(np.pi/2, 0)
    qc.OAT(theta[0], "Z")
    return loss_dict[mode] = loss_hist

if use_gpu:
    loss = torch.real(get_xi_2_S(qc))
else:
    loss = np.real(get_xi_2_S(qc))
    return loss

N=100 #number of qubits
route = ((("RN2"), ("OAT","Z"), ("TNT","Z"), ("TAT","Z"))) #define layers for QNG

#function to optimize circuit of sparse array
def sparse(optimzer, loss_dict, mode):
    objective_function = lambda params:
    cost_function(params)
    init_params = [0.0195902, 0.14166777, 0.01656466] #random init parameters
    _, _, _, loss_hist, time_iters = fit(
        objective_function, optimizer, init_params,
        return_loss_hist=True, return_time_iters=True)
    loss_dict[mode] = loss_hist
    return loss_dict, time_iters

#function to optimize circuit of tensor
def tensor(optimzer, loss_dict, mode):
    objective_function = lambda params:
    cost_function(params)
    init_params = [0.0195902, 0.14166777, 0.01656466] #random init parameters
    init_params = torch.tensor(init_params).to('cuda').requires_grad_()
    _, _, _, loss_hist, time_iters = fit(
        objective_function, optimizer, init_params,
        return_loss_hist=True, return_time_iters=True)
    loss_dict[mode] = loss_hist
    return loss_dict, time_iters

#QNG
optimizer = GD(lr=0.03, eps=1e-10, maxiter=200, use_qng=True, route=route, tol=1e-19, N=N)
loss_dict, _ = tensor(optimizer, loss_dict, "tensor_qng")

#GD
optimizer = GD(lr=0.0001, eps=1e-10, maxiter=200, tol=1e-19, N=N)
loss_dict, _ = tensor(optimizer, loss_dict, "tensor_gd")

#ADAM
optimizer = ADAM(lr=0.01, eps=1e-10, amsgrad=False, maxiter=200)
loss_dict, sparse_times = sparse(optimizer, loss_dict, "sparse_adam")

optimizer = ADAM(lr=0.001, eps=1e-10, amsgrad=False, maxiter=200)
```
```python
loss_dict, tensor_times = tensor(optimizer, loss_dict, "tensor_adam")

# plot loss values with respect to iterations
ax = plt.gca()
ax.plot(range(len(loss_dict["tensor_qng"])), loss_dict["tensor_qng"], 'c-*', label=r'$QNG; \eta=0.1$')
ax.plot(range(len(loss_dict["tensor_gd"])), loss_dict["tensor_gd"], 'y-^', label=r'$GD; \eta=0.03$')
ax.plot(range(len(loss_dict["tensor_adam"])), loss_dict["tensor_adam"], 'g-o', label=r'$Adam; \eta=0.01$')

ax.set_xlabel("iterations")
ax.set_ylabel(r"$C(\theta)$")
ax.set_xlim(0, 130)
ax.set_ylim(0, 15)
lgd = ax.legend(loc='center left', bbox_to_anchor=(1, 0.25))
plt.savefig("./loss_bm.eps", bbox_extra_artists=(lgd,), bbox_inches='tight')

# for compare running time between using tensor and sparse array structure, we plot ADAM as an example

cumsum_vqa_time_res_sparse = np.cumsum(loss_dict["sparse_adam"])
cumsum_vqa_time_res_tensor = np.cumsum(loss_dict["tensor_adam"])
ax = plt.gca()
ax.plot(range(len(cumsum_vqa_time_res_sparse)), cumsum_vqa_time_res_sparse, 'g--', label='CPU')
ax.plot(range(len(cumsum_vqa_time_res_tensor)), cumsum_vqa_time_res_tensor, 'r-', label='GPU')
ax.set_xlabel("iterations")
ax.set_ylabel("time(s)")
ax.set_ylim(0, 100)
lgd = ax.legend(loc='upper right')
plt.savefig("./timetensorsparse.eps", bbox_extra_artists=(lgd,), bbox_inches='tight')

Appendix H. Full code for producing Fig. 7 in the main text

```python
from tqli import *
from tqli.pis import *
import numpy as np
from matplotlib import pyplot as plt

N = [20, 50, 100]
figure, axis = plt.subplots(3, 1)
for i in N:

c = circuit(i, use_gpu = False)
aveJz = []  # average J_z
aveJx2 = []  # average J_x^2
aveJy2 = []  # average J_y^2
r_min = -5
r_max = 5
r_iter = 357
r_dr = (r_min - r_max) / r_iter 
0.028
x = np.linspace(r_min, r_max, r_iter)
lambda_p = -0.2
for j in x:
c.RZ(lambda_p * j)
c.TAT(lambda_p / i, 'xy')
tr0 = c.expval('Jz') / i * 2
tr1 = c.expval('Jx2') / i ** 2 * 4
tr2 = c.expval('Jy2') / i ** 2 * 4
aveJz.append(np.real(tr0))
aveJx2.append(np.real(tr1))
aveJy2.append(np.real(tr2))
axis[0].plot(x, aveJz, '--')
axis[1].plot(x, aveJx2, '--')
axis[2].plot(x, aveJy2, '--')
plt.savefig("figQPT.eps")

As mentioned in the main text, we can observe the QPT by choosing the approximate r_step, from which resolve to the adiabatic annealing evolution.
```
Appendix I. Guide for running multi-core and GPU-Acceleration on chip Apple M1 Max

It requires MacOS 12.3+ and an ARM Python installation. We can check them by:

```python
import platform
platform.platform()
```

```
>> macOS-12.4-arm64 (OK)
```

Switch to the terminal and create a new ARM environment using Anaconda:

```
CONDA_SUBDIR=osx-arm64 conda create -n ml
```

```
>> macOS-11.8-x86_64-i386-64bit (NOT)
>> macOS-12.4-arm64-arm-64bit (OK)
```

Modify the CONDA_SUBDIR variable to permanently use osx-arm64 for future use:

```
conda env config vars set CONDA_SUBDIR=osx-arm64
```

From here, we can switch between the two environments (base) and (ml):

```
conda activate
conda activate ml
```

In the (ml) environment, let’s install the required libraries (including tqix and pyTorch) and run the codes above normally. For more detail, see: https://towardsdatascience.com/gpu-acceleration-comes-to-pytorch-on-m1-macs-195c399efcc

We can confirm if the MPS is working in Python by:

```
import torch
torch.has_mps
```

```
>> True
```

In Fig 1.8 we compare the running time on chip Apple M1 Max with #process = 10 and GPU. The system configuration is the same as in Fig. 3

---

**References**

[1] L. B. Ho, K. Q. Tuan, H. Q. Nguyen, tqix: A toolbox for quantum in x: X: Quantum measurement, quantum tomography, a Computer Physics Communications 263 (2021) 107902. doi:https://doi.org/10.1016/j.cpc.2021.107902

[2] R. H. Dicke, Coherence in spontaneous radiation processes Phys. Rev. 93 (1954) 99–110. doi:10.1103/PhysRev.93.99

[3] P. Ma, X. Wang, C. Sun, F. Nori, Quantum spin squeezing Physics Reports 509 (2) (2011) 89–165. doi:10.1016/j.physrep.2011.08.003

[4] R. Kaubruegger, P. Silvi, C. Kokail, R. van Bijnen, A. M. Rey, J. Ye, A. M. Kaufman, P. Zoller, Variational spin-squeezing algorithms on programmable quantum sensors, Phys. Rev. Lett. 123 (2019) 260505. doi:10.1103/PhysRevLett.123.260505

[5] R. Kaubruegger, D. V. Vasilyev, M. Schütte, K. Hammerer, P. Zoller, Quantum variational optimization of Ramsey interferometry and atomic clocks, Phys Rev. X11(2021)041045. doi:10.1103/PhysRevX.11.041045

[6] S. Sachdev, Quantum Phase Transitions, 2nd Edition, Cambridge University Press, 2011. doi:10.1017/CBO9780511973765

[7] A. Smith, M. S. Kim, F. Pollmann, J. Knolle, Simulating quantum many-body dynamics on a current digital quantum computer, Quantum Information and Computing 1 (1) (2019) 106. doi:10.1023/s14534-019-0217-0

[8] P. Schindler, D. Nigg, T. Monz, J. T. Barreiro, E. Martínez, S. Hild, D. Schmiedmayer, C. Zinkernagel, J. Fröhlich, M. Hennrich, R. Blatt, A quantum information processor with trapped ions, New Journal of Physics 15 (12) (2013) 123012. doi:10.1088/1367-2630/15/12/123012

[9] T. Hernández Yanes, M. Płodzień, M. Macko, Sinkevičienė, G. Zlabys, G. Juzeliūnas, E. Witkowska, Phonom-induced spin-spin interactions in diamond nanostructures: Application to spin squeezing, Phys. Rev. Lett. 129 (2022) 090403. doi:10.1103/PhysRevLett.129.090403

[10] S. D. Bennett, N. Y. Yao, J. Otterbach, P. Zoller, P. Rabl, M. D. Lukin, Collective uncertainty in partially polarized and partially decohered spin-4 systems, Phys. Rev. A 78 (2008) 052101. doi:10.1103/PhysRevA.78.052101

[11] B. A. Chase, J. M. Geremia, Collective processes of an ensemble of spin-1/2 particles, Phys. Rev. A 81 (2010) 052104. doi:10.1103/PhysRevA.81.052104

[12] S. Sachdev, Quantum Phase Transitions, 2nd Edition, Cambridge University Press, 2011. doi:10.1017/CBO9780511973765

[13] V. V. Mikhailov, Multiplicities of angular momenta in a system of n-dimensional oscillators and the reduction SU(n)⊃0(3), Journal of Physics A: Mathematical and General 11 (3) (1978) 443–456. doi:10.1088/0305-4470/11/3/006

[14] L. Novo, T. Moroder, O. Gühne, Genuine multiparticle entanglement of permutationally invariant states, Phys. Rev. A 88 (2013) 012305. doi:10.1103/PhysRevA.88.012305

[15] L. Arnaud, All possible permutation symmetries of a quantum system, Phys. Rev. A 93 (2016) 012320. doi:10.1103/PhysRevA.93.012320

[16] J. K. Stockton, J. M. Geremia, A. C. Doherty, H. Mabuchi, Characterizing the entanglement of symmetric many-particle spin-4 systems, Phys. Rev. A 97 (2013) 032112. doi:10.1103/PhysRevA.97.032112

[17] L. M. Duan, M. Zoller, Lukin, J. I. Cirac, P. Zoller, Long-distance quantum communication with atomic ensembles and linear optics, URL

---

Figure 1.8: Benchmarking on chip Apple M1 Max.
[84] R. D. Neidinger, Introduction to automatic differentiation and matlab object-oriented programming, SIAM Review 52 (3) (2010) 545–566.
https://doi.org/10.1137/080743627
https://www.sciencedirect.com/science/article/pii/S0377042700004222

[85] J. Stokes, J. Isaac, N. Killoran, G. Carleo, Quantum Natural Gradient, Quantum 4 (2020) 269, doi:10.22331/q-2020-05-25-269
https://doi.org/10.22331/q-2020-05-25-269

[86] T. Haug, M. S. Kim, Natural parameterized quantum circuit, https://arxiv.org/abs/2107.14063
https://arxiv.org/abs/2107.14063

[87] V. T. Hai, L. B. Ho, Universal compilation for quantum state preparation and tomography, https://arxiv.org/abs/2204.11635

[88] H. Lipkin, N. Meshkov, A. Glick, Validity of many-body approximation methods for a solvable model: (i). exact solutions and perturbation theory, Nuclear Physics 62 (2) (1965) 188–198.
https://doi.org/10.1016/0029-5582(65)90862-X
https://www.sciencedirect.com/science/article/pii/002955826590862X

[89] S. Dusuel, J. Vidal, Continuous unitary transformations and finite-size scaling exponents in the lipkin-meshkov-glick model, Phys. Rev. B 71 (2005) 224420, doi:10.1103/PhysRevB.71.224420
https://link.aps.org/doi/10.1103/PhysRevB.71.224420

[90] T. Caneva, R. Fazio, G. E. Santoro, Adiabatic quantum dynamics of the lipkin-meshkov-glick model, Phys. Rev. B 78 (2008) 104426, doi:10.1103/PhysRevB.78.104426
https://link.aps.org/doi/10.1103/PhysRevB.78.104426

[91] O. L. Acevedo, L. Quiroga, F. J. Rodríguez, N. F. Johnson, New dynamical scaling universality for quantum networks across adiabatic quantum phase transitions, Phys. Rev. Lett. 112 (2014) 030403, doi:10.1103/PhysRevLett.112.030403
https://link.aps.org/doi/10.1103/PhysRevLett.112.030403

[92] E. Farhi, J. Goldstone, S. Gutmann, J. Lapan, A. Lundgren, D. Preda, A quantum adiabatic evolution algorithm applied to random instances of an np-complete problem, Science 292 (5516) (2001) 472–475
arXiv:https://www.science.org/doi/pdf/10.1126/science.1057726
https://www.science.org/doi/abs/10.1126/science.1057726

[93] D. Aharonov, W. van Dam, J. Kempe, Z. Landau, S. Lloyd, O. Regev, Adiabatic quantum computation is equivalent to standard quantum computation, Science 297 (5584) (2002) 479–484
https://doi.org/10.1126/science.1057726
https://www.science.org/doi/10.1126/science.1057726

[94] D. P. Kingma, J. Ba, Adam: A method for stochastic optimization, arXiv:https://arxiv.org/abs/1412.6980
https://arxiv.org/abs/1412.6980

17