The Journal of Open Source Software

qujax: Simulating quantum circuits with JAX

Samuel Duffield [1¶], **Gabriel Matos** [1,2], **and Melf Johannsen** [1]

1 Quantinuum **2** University of Leeds ¶ Corresponding author

DOI: 10.21105/joss.05504

Software

- Review
- Repository
- Archive

Editor: Lucy Whalley

Reviewers:

- @jmiszczak
- @amitkumarj441
- @meandmytram

Submitted: 29 March 2023
Published: 12 September 2023

License
Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License (CC BY 4.0).

Summary

qujax is a pure JAX (Bradbury et al., 2018), purely functional Python package for the classical simulation of quantum circuits. A JAX implementation of quantum circuits inherits benefits such as seamless automatic differentiation, support for GPUs/TPUs as well as integration with a host of other tools within the JAX ecosystem.

qujax is hosted on PyPI for easy installation and comes with detailed documentation and a suite of example notebooks.

qujax represents a quantum circuit as a collection of three equal length Python iterables:

- a series of gate identifiers specifying the sequence of quantum gates to be applied to the qubits as part of the circuit. Each element can be either
  - a string referring to a gate in `qujax.gates` e.g. `"Z"`, `"Rx"`
  - a JAX array representing a unitary matrix
  - a function that returns such an array
- a series indicating which qubits in the circuit each gate should be applied to
- a series of indices indicating which entries of a parameter vector (which is provided when later evaluating the circuit) correspond to parameters of the gate (e.g. rotation gates such as `"Rx"` take one parameter).

For example, a valid quantum circuit specification would be the following

```
qujax.print_circuit(["X", "Rz", "Rz", "CRz"],
                    [[0], [1], [0], [0, 1]],
                    [[], [0], [1], [1]])

# q0: -----X-----Rz[1]-----○---
#                          ¦
# q1: ---Rz[0]-----------CRz[1]
```

Note that for angular parameters, the default parameterised gates in `qujax.gates` assume **angles are specified in half-turns** (i.e. $\theta \in [0, 2)$) as opposed to radians.

Statetensor

In quantum mechanics, a *pure state* is fully specified by a statevector

$$|\psi\rangle = \sum_{i=1}^{2^N} \alpha_i |i\rangle \in \mathbb{C}^{2^N},$$

where N is the number of qubits and each α_i is a complex scalar number referred to as the ith *amplitude*. Quantum states are also normalised such that $\langle\psi|\psi\rangle = \sum_{i=1}^{2^N} |\alpha_i|^2 = 1$. We work in the computational basis, where $|i\rangle$ is represented as a vector of zeros with a one in the ith position (e.g. for $N = 2$, $|2\rangle$ is represented as `[0 1 0 0]`). In qujax, we represent such vectors as a *statetensor*, where a pure state is encoded in a tensor of complex numbers with

shape `(2,) * N`. The statetensor representation is convenient for quantum arithmetic (such as applying gates, tracing out qubits and sampling bitstrings). For example, the amplitude corresponding to the bitstring `[0 1 0 0]` can be accessed with `statetensor[0, 1, 0, 0]`. The statevector can always be obtained by calling `statevector = statetensor.flatten()`.

One can use `qujax.get_params_to_statetensor_func` to generate a pure JAX function encoding a parameterised quantum state

$$|\psi_\theta\rangle = U_\theta|\phi\rangle,$$

where θ is a parameter vector and $|\phi\rangle$ is an initial quantum state that can be provided via the optional argument `statetensor_in` (that defaults to $|0\rangle$ =`[1 0 ... 0]`).

Unitarytensor

Alternatively, one can call `qujax.get_params_to_unitarytensor_func` to get a function returning a tensor representation of the unitary U_θ with shape `(2,) * 2 * N`.

Densitytensor

The quantum states that can be represented as above are called pure quantum states. More general quantum states can be represented by using a *density matrix*. The density matrix representation of a pure quantum state $|\psi\rangle$ can be obtained via the outer product $\rho = |\psi\rangle\langle\psi| \in \mathbb{C}^{2^N \times 2^N}$. More generally a density matrix encodes a *mixed state*

$$\rho = \sum_k p_k |\psi_k\rangle\langle\psi_k|,$$

which can be interpreted as classical statistical mixture of pure states, with $p_k \in [0, 1]$ and $\sum_k p_k = 1$.

Density matrices are also supported in `qujax` in the form of *densitytensors* - complex tensors of shape `(2,) * 2 * N`. Similar to the statetensor simulator, parameterised evolution of a densitytensor can be implemented via general Kraus operations with `qujax.get_params_to_densitytensor_func`. Further details on density matrices and Kraus operators are available in the documentation or published literature (Nielsen & Chuang, 2010, pp. 98–105).

Expectation values

Expectation values can also be calculated conveniently with `qujax`. In simple cases, such as a combinatorial optimisation problems (e.g. MaxCut), this can be done by extracting measurement probabilities from the statetensor or densitytensor and calculating the expected value of a cost function directly. For more sophisticated bases, `qujax.get_statetensor_to_expectation_func` and `qujax.get_densitytensor_to_expectation_func` generate functions that map to the expected value of a given series of Hermitian tensors. Sampled expectation values (which replicate so-called shot noise for a given number of shots) are also supported in `qujax`.

Statement of need

JAX is emerging as a state-of-the-art library for high-perfomance scientific computation in Python due to its composability, automatic differentiation and support for GPUs/TPUs, as well as adopting the NumPy (Harris et al., 2020) API resulting in a low barrier to entry.

`qujax` is a lightweight, purely functional library written entirely in JAX, composing seamlessly with the ever-expanding JAX ecosystem. It emphasises clarity and readability, making it easy

to debug, reducing the barrier to entry, and decreasing the overhead when integrating with existing code or extending it to meet specific research needs.

These characteristics contrast with the already existing array of excellent quantum computation resources in Python, such as cirq (Cirq Developers, 2022), pytket (Sivarajah et al., 2020), qiskit (Bradbury et al., 2018), Qulacs (Suzuki et al., 2021), TensorFlow Quantum (Broughton et al., 2020), DisCoPy (Toumi et al., 2022), Pennylane (Bergholm et al., 2018) or quimb (Gray, 2018), the latter three supporting JAX as a backend. These represent complex full-fledged frameworks which supply their own abstractions, being either wider in scope or specializing in specific use-cases. The core difference is that `qujax` is designed to be purely functional.

While generic circuit simulation is within scope, `qujax` does not support tensor network simulation. There is an active area of research investigating this as a tool for classical simulation of quantum circuits with software including DisCoPy (Toumi et al., 2022), quimb (Gray, 2018) and TensorCircuit (Zhang et al., 2023). While tensor networks represent a very promising field of research, their implementation entails a more sophisticated API (in tensor networks, the representation of a quantum state can be considerably more elaborate), greatly increasing the complexity of the package. Thus, tensor network computation is currently seen as being beyond the scope of `qujax`.

`pytket-qujax`

`qujax` is accompanied by an extension package `pytket-qujax` supporting easy conversion to and from `pytket.Circuit` objects, thus providing a convenient bridge between pytket and JAX ecosystems. It possible to use the other pytket extensions to convert from and to qiskit (Bradbury et al., 2018), Qulacs (Suzuki et al., 2021), cirq (Cirq Developers, 2022) and Pennylane (Bergholm et al., 2018) as well.

Acknowledgements

We acknowledge notable support from Kirill Plekhanov as well as Gabriel Marin, Enrico Rinaldi and Richie Yeung.

References

Bergholm, V., Izaac, J., Schuld, M., Gogolin, C., Ahmed, S., Ajith, V., Alam, M. S., Alonso-Linaje, G., AkashNarayanan, B., Asadi, A., Arrazola, J. M., Azad, U., Banning, S., Blank, C., Bromley, T. R., Cordier, B. A., Ceroni, J., Delgado, A., Di Matteo, O., … Killoran, N. (2018). *PennyLane: Automatic differentiation of hybrid quantum-classical computations*. arXiv. https://doi.org/10.48550/ARXIV.1811.04968

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., & Zhang, Q. (2018). *JAX: Composable transformations of Python+NumPy programs* (Version 0.3.13). http://github.com/google/jax

Broughton, M., Verdon, G., McCourt, T., Martinez, A. J., Yoo, J. H., Isakov, S. V., Massey, P., Halavati, R., Niu, M. Y., Zlokapa, A., Peters, E., Lockwood, O., Skolik, A., Jerbi, S., Dunjko, V., Leib, M., Streif, M., Von Dollen, D., Chen, H., … Mohseni, M. (2020). *TensorFlow quantum: A software framework for quantum machine learning*. arXiv. https://doi.org/10.48550/ARXIV.2003.02989

Cirq Developers. (2022). *Cirq* (Version v1.1.0). Zenodo. https://doi.org/10.5281/zenodo.7465577

Gray, J. (2018). Quimb: A python package for quantum information and many-body calculations. *Journal of Open Source Software*, *3*(29), 819. https://doi.org/10.21105/joss.00819

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

Nielsen, M. A., & Chuang, I. L. (2010). *Quantum computation and quantum information: 10th anniversary edition*. Cambridge University Press. https://doi.org/10.1017/CBO9780511976667

Sivarajah, S., Dilkes, S., Cowtan, A., Simmons, W., Edgington, A., & Duncan, R. (2020). T|ket⟩: A retargetable compiler for NISQ devices. *Quantum Science and Technology*, *6*(1), 014003. https://doi.org/10.1088/2058-9565/ab8e92

Suzuki, Y., Kawase, Y., Masumura, Y., Hiraga, Y., Nakadai, M., Chen, J., Nakanishi, K. M., Mitarai, K., Imai, R., Tamiya, S., Yamamoto, T., Yan, T., Kawakubo, T., Nakagawa, Y. O., Ibe, Y., Zhang, Y., Yamashita, H., Yoshimura, H., Hayashi, A., & Fujii, K. (2021). Qulacs: A fast and versatile quantum circuit simulator for research purpose. *Quantum*, *5*, 559. https://doi.org/10.22331/q-2021-10-06-559

Toumi, A., Felice, G. de, & Yeung, R. (2022). *DisCoPy for the quantum computer scientist*. arXiv. https://doi.org/10.48550/ARXIV.2205.05190

Zhang, S.-X., Allcock, J., Wan, Z.-Q., Liu, S., Sun, J., Yu, H., Yang, X.-H., Qiu, J., Ye, Z., Chen, Y.-Q., Lee, C.-K., Zheng, Y.-C., Jian, S.-K., Yao, H., Hsieh, C.-Y., & Zhang, S. (2023). TensorCircuit: A Quantum Software Framework for the NISQ Era. *Quantum*, *7*, 912. https://doi.org/10.22331/q-2023-02-02-912