The Journal of Open Source Software

QuantumReservoirPy: A Software Package for Time Series Prediction

Ola Tangen Kulseng [1*], **Stanley Miao** [2*], **Franz G. Fuchs** [3*], and **Alexander Stasik** [3,4*¶]

2 David R. Cheriton School of Computer Science, University of Waterloo, Canada **1** Department of Physics, Norwegian University of Science and Technology (NTNU), Norway **3** Department of Mathematics and Cybernetics, SINTEF Digital, Norway **4** Department of Data Science, Norwegian University of Life Science, Norway ¶ Corresponding author * These authors contributed equally.

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Open Journals

Reviewers:

- @openjournals

Submitted: 01 January 1970
Published: unpublished

Summary

Reservoir computing on quantum computers have recently emerged as a potential resource for time series prediction, owing to their inherent complex dynamics. To advance Quantum Reservoir Computing (QRC) research, we have developed the Python package `QuantumReservoirPy`, which fascilitates QRC using quantum circuits as reservoirs.

The package is designed to be easy to use, while staying completely customizable. The resulting interface, similar to that of reservoirpy (Trouvain et al., 2020), simplifies the development of quantum reservoirs, and provides logical methods of comparison between reservoir architectures.

Statement of need

Reservoir computing (RC) is a paradigm in machine learning for time series prediction. With recent developments, it has shown a promising advantage in efficiency due to the relative simplicity of the associated training process over conventional neural network methods (Tanaka et al., 2019). In reservoir computing, a dynamical system composed of hidden functional representations with non-linear state transitions is chosen as a reservoir. Input data from a time series is sequentially encoded and fed into the reservoir. The hidden parameters of the reservoir undergo a non-linear evolution dependent on the stored state and the encoded information fed into the system. Features are subsequently decoded from a readout of certain parameters of the reservoir, which is used to train a simple linear model for the desired time series prediction output, see Figure 1 for an illustration.

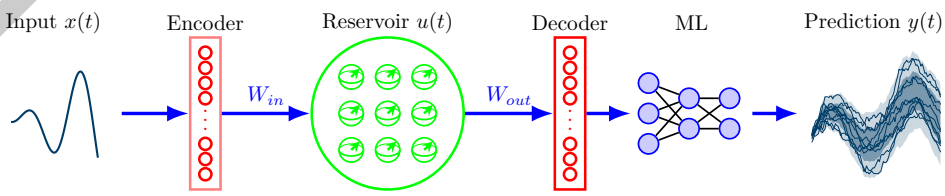


Figure 1: A quantum reservoir system consists of a learning task, an en- and de-coder (red) and the dynamic system itself (green). In standard RC the machine learning part is linear regression.

The selected reservoir must incorporate non-linear state transitions within the limitations of a fixed structure. A reservoir can be virtual, such as a sparsely-connected recurrent neural

network with random fixed weights, termed as an echo state network (Jaeger & Haas, 2004) or even physical, such as a bucket of water (Fernando & Sojakka, 2003).

QRC is a proposed method of RC utilizing quantum circuits as reservoirs. Multi-qubit systems with the capability of quantum entanglement provide compelling non-linear dynamics that match the requirements for a feasible reservoir. Furthermore, the exponentially-scaling Hilbert space of large multi-qubit systems support the efficiency and state-storage goals of RC. As a result, quantum computers have been touted as a viable dynamical system to produce the intended effects of reservoir computing.

In QRC, data is encoded by operating on one or more qubit(s) to reach a desired state. To obtain the desired complex non-linearity in a quantum system as a reservoir, entangling unitary operations are performed over the system. The readout is measured from one or more qubit(s) of the quantum state, which can be achieved through partial or full measurement over the system. Since quantum measurement results in a collapse to the measured state, repetitive measurements over identical systems are used to sample from the distribution of the unknown quantum state, which is then post-processed to obtain a decoded measurement for the subsequent linear model. Furthermore, this collapse results in a loss of retained information and entanglement in the system. Where only a partial measurement is taken (as in (Yasuda et al., 2023)), this may have a desired effect of a slow leak of information driven from earlier input. When measurement is taken over the full system, the system may instead be restored through re-preparation of the pre-existing system, achieved in (Suzuki et al., 2022) and through the restarting and rewinding protocols in (Mujal et al., 2023).

Existing implementations of QRC have used proprietary realizations on simulated and actual quantum computers. The lack of a shared structure between implementations has resulted in a disconnect with comparing reservoir architectures. In addition, individual implementations require a certain amount of redundant procedure prior to the involvement of specific concepts.

We observe that there is a need for a common framework for the implementation of QRC. As such, we have developed `QuantumReservoirPy` to solve the presented issues in current QRC research. `QuantumReservoirPy` is designed to handle every step of the reservoir computing pipeline, in addition to the pre- and post-processing necessary in the quantum case. In providing this software package, we hope to facilitate logical methods of comparison in QRC architecture and enable a simplified process of creating a custom reservoir from off-the-shelf libraries with minimal overhead requirements to begin development.

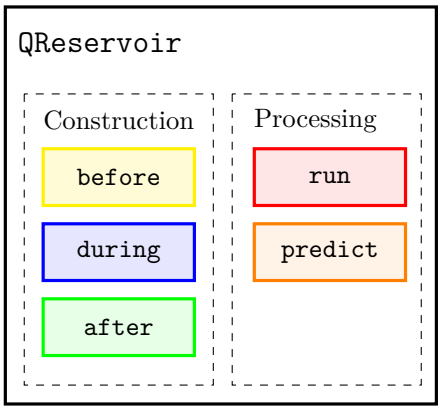


Figure 2: Quantum circuit construction may be customized through the `before`, `during`, and `after` methods and a timeseries processed with the `run` and `predict` methods.

The Journal of Open Source Software

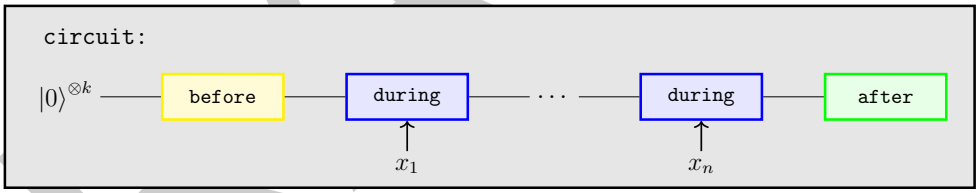
Design and implementation

We intend `QuantumReservoirPy` to provide flexibility to all possible designs of quantum reservoirs, with full control over pre-processing, input, quantum circuit operations, measurement, and post-processing. In particular, we take inspiration from the simple and flexible structure provided by the ReservoirPy software package reservoirpy (Trouvain et al., 2020).

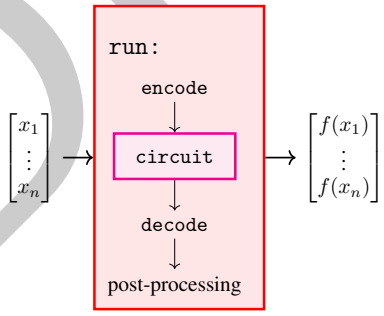
Unlike the parameterized single-class structure of ReservoirPy, `QuantumReservoirPy` uses the abstract class-based structure of Figure 2 as the former would be restrictive in the full customization of QRC. In particular, quantum circuit operations and measurement may be conducted in differing arrangements. With an abstract class, we allow the user to define the functionality of the quantum circuit, which provides full flexibility over all existing implementations of QRC. This structure also implicitly provides access to the full Qiskit circuit library.

Construction and processing

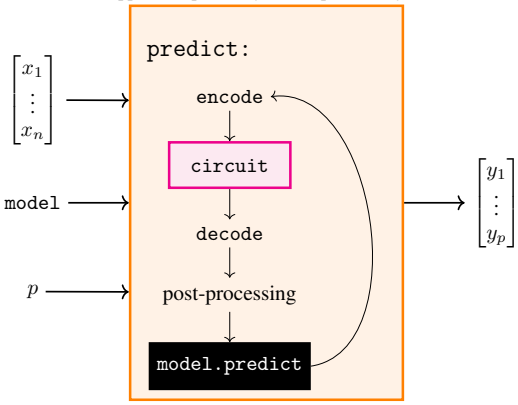
The construction methods in `QuantumReservoirPy` serve as the sequential operations performed on the quantum system. The operations in the `before` method prepares the qubits, which may include initialization to a default, initial, or previously saved state. The `during` method provides the operations that are subsequently applied for each step in the timeseries. This may include (but is not limited to) measurement, re-initialization, and entangling operations. Finally, the operations in the `predict` method are applied once following the processing of the entire timeseries, which may include the transformation of final qubit states and measurement. Figure 3a demonstrates the aforementioned arrangement, which is implemented as a hidden intermediary process in a `QuantumReservoirPy` quantum reservoir.



(a) A functional overview of the hidden quantum circuit architecture common to all quantum reservoirs, where x_t is the observed input sequence. The customized `before`, `during`, and `after` methods are applied sequentially to the quantum system.



(b) The input timeseries is encoded into the quantum circuit. After measurements are decoded from the quantum circuit, they are post-processed into the transformed feature vector.



(c) The feature vector produced by encoding, decoding, and pre-processing is used by the model to predict the next step in the timeseries. This process is repeated p times and returns the resulting prediction sequence.

Figure 3: The intended functionality of the `run` and `predict` method. The observed input sequence is $\{x_t\}$ and the target sequence $\{y_t\}$. The reservoir f performs an evolution in time.

The processing methods serve as functions acting on the quantum reservoir itself. The `run`

method is used to process training data by taking a timeseries as input and returning the transformed data after being processed by the quantum reservoir. This is done using the hidden `circuit` interface as presented in Figure 3a, where data encoding and decoding follow the implementation of the custom construction methods. Depending on the realization of QRC, such as averaging over multi-shot data, additional post-processing is included in the `run` method to achieve the desired output. The transformed data from the `run` method serves as training data for a simple machine learning model. Figure 3b provides a visualization of the `run` method.

The `predict` method functions as a complete forecasting process involving the same hidden `circuit` interface, encoding, decoding, and post-processing. Additionally, a trained simple machine learning model is used to predict the next step in the timeseries from the transformed and post-processed data. The resulting prediction is then fed in as input for the following prediction, which occurs as an iterative process until the specified number of forecasting steps is reached. At this point, the `predict` method returns the sequence of predictions from each iteration. Figure 3c provides a visualization of the `predict` method.

Reservoirs and example usage

`QuantumReservoirPy` provides two prepackaged reservoirs that implement the processing methods `run` and `predict` according to the structure presented above. These reservoirs use common QRC schemes as a basis to get started with the software package or restrict customization to circuit construction. As such, the construction of a reservoir using either of these schemes only requires user implementation of the sequential circuit methods `before`, `during`, and `after`.

Static Reservoirs

The `Static` abstract reservoir class is structured to run multi-shot processing according to a repeated-measurement process. A single circuit is created according to the circuit construction methods and the timeseries parameter. Measurement is expected in the `during` function to generate the transformed reservoir output using the single circuit. This circuit is sent to the specified Qiskit backend and is run with the remaining parameters provided to the processing methods. The resulting measurement data is post-processed by taking the average over all shots before it is returned to the user as decoded reservoir output.

An example of a `Static` quantum reservoir is the following.

```
class QuantumReservoir(Static):
    def before(self, circuit):
        circuit.h(circuit.qubits)

    def during(self, circuit, timestep):
        circuit.initialize(encoder[timestep], [0, 1])
        circuit.append(operator, circuit.qubits)
        circuit.measure([0, 1])
```

In the example, the timestep passed to the `during` method is encoded in the circuit according to an `encoder`. Measurement is also taken in the `during` method to provide sequential data using a single circuit for the entire timeseries. The `after` method is not necessary for this scheme and is left as the inherited empty method by default.

Incremental Reservoirs

The `Incremental` abstract reservoir class processes data using multiple circuits over a moving substring of the timeseries. Circuits of length at most the specified `memory` parameter are

created for step-by-step processing of the timeseries according to the circuit construction methods. Each circuit with a fixed-length memory is sent to the Qiskit backend to be run with the remaining parameters to the processing methods. Using this scheme, measurements taken in the `after` method can be post-processed to provide the user with the desired reservoir output.

An example of an `Incremental` quantum reservoir is the following.

```
class QuantumReservoir(Incremental):
    def before(self, circuit):
        circuit.h(circuit.qubits)

    def during(self, circuit, timestep):
        circuit.initialize(encoder[timestep], [0, 1])
        circuit.append(operator, circuit.qubits)

    def after(self, circuit):
        circuit.measure_all()
```

In the example, the `during` method is still used to encode the current timestep in the timeseries, but measurement is instead taken in the `after` method. Since a circuit is created for each timestep, the combined measurements produce the desired sequential data.

Custom Reservoirs

Custom reservoirs provide full flexibility over processing data in a quantum reservoir. A custom reservoir can be created by implementing the `QReservoir` abstract class. Unlike the prepackaged reservoirs, a custom reservoir must implement the `run` and `predict` methods, in addition to the circuit construction methods.

Backend

When a quantum reservoir is instantiated, it requires a Qiskit backend. If no backend is specified, then `AerSimulator` is used by default. The backend must support the circuit operations specified in the circuit construction methods. When creating a custom reservoir, the `run` and `predict` methods should run the quantum reservoir on `self.backend` attribute.

As an example, instantiation of a quantum reservoir using the simulator backend `FakeTorontoV2` is as follows.

```
from qiskit.providers.fake_provider import FakeTorontoV2

backend = FakeTorontoV2()
reservoir = QuantumReservoir(n_qubits=4, backend=backend)
```

Once a quantum reservoir has been instantiated, training data for the trainable model is produced by using the `run` method on the timeseries. The trained model can then be passed to the `predict` method to make predictions:

```
output = reservoir.run(timeseries=timeseries, shots=10000)
# ...(model training)...
predictions = reservoir.predict(num_pred=10, model=model, from_series=timeseries, shots=
```

Model training of a scikit-learn estimator between the `run` and `predict` methods is not shown. The `shots` parameter is directly passed to the Qiskit backend when using a prepackaged reservoir.

Package Details

Dependencies

The three main dependencies of `QuantumReservoirPy` are numpy, qiskit, and scikit-learn. We strive for `QuantumReservoirPy` to support compatibility with existing reservoir computing and quantum computing workflows.

`QuantumReservoirPy` uses NumPy as a core dependency. Allowed versions of NumPy ensure compatibility with classical ReservoirPy to facilitate comparison between classical and quantum reservoir architectures. An example is provided here.

Much of existing research in QRC is performed on IBM devices and simulators (see (Suzuki et al., 2022; Yasuda et al., 2023)), programmed through the Qiskit software package. To minimize disruption in current workflows, `QuantumReservoirPy` is built as a package to interact with Qiskit circuits and backends. It is expected that the user also use Qiskit in the customization of reservoir architecture when working with `QuantumReservoirPy`.

The model used by the `predict` function is expected to be a scikit-learn estimator. Likewise to our dependency on Qiskit, this allows for complete customization over the choice of model and faster adoption of `QuantumReservoirPy` from a simple package structure.

Distribution

The software package is developed and maintained through a public GitHub repository provided through the `OpenQuantumComputing` organization. The `main` branch of this repository serves as the latest stable version of `QuantumReservoirPy`. Installation through cloning this repository is intended for development purposes.

Official releases of the software package are published through the Python Package Index (PyPI) under the `quantumreservoirpy` name. Installation through PyPI is the suggested method of installation for general-purpose use.

`QuantumReservoirPy` is licensed under the GNU General Public License v3.0. `QuantumReservoirPy` also includes derivative work of Qiskit, which is licensed by IBM under the Apache License, Version 2.0.

Documentation

Documentation is mainly provided online in the form of a user guide and API reference. The user guide includes steps for installation and getting started with a simple quantum reservoir. Examples are also provided for further guidance. The API reference outlines the intended classes and functions exposed by the software package. A brief overview of the software package is also included as a `README.md` file at the top level of the public GitHub repository.

Further Development

The authors continue to support and maintain the project. Users may report package issues and desired features by opening an issue on the public GitHub repository or contacting the authors by email. Additional opportunities for further development on `QuantumReservoirPy` include supplementary built-in processing schemes, expanded features for data visualization, and reservoir evaluation methods.

Acknowledgements

Work in this project was supported by the NTNU and SINTEF Digital through the International Work-Integrated-Learning in Artificial Intelligence (IWIL AI) program, in partnership with SFI

NorwAI and the Waterloo Artificial Intelligence Institute (Waterloo.AI). IWIL AI is funded by the Norwegian Directorate for Higher Education and Skills (HK-dir).

References

Fernando, C., & Sojakka, S. (2003). Pattern recognition in a bucket. *Advances in Artificial Life*, 588–597. ISBN: 978-3-540-39432-7

Jaeger, H., & Haas, H. (2004). Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, *304*(5667), 78–80. https://doi.org/10.1126/science.1091277

Mujal, P., Martínez-Peña, R., Giorgi, G. L., Soriano, M. C., & Zambrini, R. (2023). Time-series quantum reservoir computing with weak and projective measurements. *Npj Quantum Information*, *9*(1), 16. https://doi.org/10.1038/s41534-023-00682-z

Suzuki, Y., Gao, Q., Pradel, K. C., Yasuoka, K., & Yamamoto, N. (2022). Natural quantum reservoir computing for temporal information processing. *Scientific Reports*, *12*(1), 1353. https://doi.org/10.1038/s41598-022-05061-w

Tanaka, G., Yamane, T., Héroux, J. B., Nakane, R., Kanazawa, N., Takeda, S., Numata, H., Nakano, D., & Hirose, A. (2019). Recent advances in physical reservoir computing: A review. *Neural Networks*, *115*, 100–123.

Trouvain, N., Pedrelli, L., Dinh, T. T., & Hinaut, X. (2020, September). ReservoirPy: an Efficient and User-Friendly Library to Design Echo State Networks. *ICANN 2020 - 29th International Conference on Artificial Neural Networks*. https://inria.hal.science/hal-02595026

Yasuda, T., Suzuki, Y., Kubota, T., Nakajima, K., Gao, Q., Zhang, W., Shimono, S., Nurdin, H. I., & Yamamoto, N. (2023). *Quantum reservoir computing with repeated measurements on superconducting devices*. https://arxiv.org/abs/2310.06706