




VeriQuEST.jl: Emulating verification of quantum computations with QuEST

Jonathan Miller^{1*}, Dominik Leichtle^{2*}, Elham Kashefi², and Cica Gustiani²

¹ School of Informatics, University of Edinburgh, 10 Crichton Street, Edinburgh EH8 9AB, United Kingdom ² Laboratoire d'Informatique de Paris 6, CNRS, Sorbonne Université, 4 Place Jussieu, Paris 75005, France * These authors contributed equally.

DOI: [10.xxxxxx/draft](https://doi.org/10.xxxxxx/draft)

Software

- [Review](#) 
- [Repository](#) 
- [Archive](#) 

Editor: [Open Journals](#) 

Reviewers:

- [@openjournals](#)

Submitted: 01 January 1970

Published: unpublished

License

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

Summary

Within the practical context of quantum computing, the *client-server* framework is anticipated to become predominant in the future due to the high costs, intensive maintenance, and operational complexity of quantum computers. In this framework, a client with a very limited quantum power, so-called Alice, delegates her quantum computation to a powerful quantum server, so-called Bob, who then provides Alice her computation results. While this scenario appears practical, it raises significant security concern: how do we ensure Bob follows Alice's instructions and confirm the reliability of his quantum computer? These questions undeniably pose significant challenges and remain under active investigation; *verifiable quantum computations* is a subfield of quantum computing that aims to address these concerns.

This paper introduces VeriQuEST.jl, a Julia package equipped with functionalities to emulate verification protocols for quantum computation ([Gheorghiu et al., 2019](#)) within the framework of measurement-based quantum computations ([Raussendorf & Briegel, 2001](#)). The package utilizes QuEST ([Jones et al., 2019](#)) as its backend, a versatile quantum computer emulator (including noise simulation) written in C, while supporting high-performance computations. VeriQuEST.jl is designed to aid researchers in testing their verification protocols or concepts on emulated quantum systems. This approach allows them to assess performance estimates without the need for actual hardware, which, at the time of writing this paper, is significantly limited.

Statement of need

Our package VeriQuEST.jl is aimed at assisting researchers in exploring and designing their verification protocols within the paradigm of *measurement-based quantum computation* (MBQC). The conventional way to express quantum algorithms is through a series of unitary operations and measurements, so-called *gate-based* computations ([Nielsen & Chuang, 2011](#)). On the other hand, MBQC uses a *graph state* as the resource and a series of adaptive single-qubit measurements to realise a quantum algorithm. While gate-based emulators – such as QuEST – are predominant, MBQC emulators remain scarce. Our package, VeriQuEST.jl, offers an interface that enables users to express their verification protocols and quantum algorithms in the native MBQC language, supporting interactive protocols, such as verifications, that require a quantum internet, powered by a performant backend QuEST ([Jones et al., 2019](#)). Moreover, to this date, there is no emulator that is specifically aimed at simulating verification protocols.

Core features and functionality

VeriQuEST.jl is equipped with fundamental functionalities to support the testing of various verification protocols of quantum computations based on MBQC. The package VeriQuEST.jl contains three fundamental elements to support emulation of verification protocols: MBQC computations, multi-round interactions between client and server, and Hilbert space partition to realize the explicit client-server separation. The package allows for (noiseless) state vector and (noisy) density matrix simulations. In the current release, we provide several well-known MBQCs and verification protocols ready for the users to use. If one wishes to become acquainted with the details, see the public GitHub repository for [VeriQuEST](#).

Software architecture

VeriQuEST.jl is a package written entirely in Julia, however to emulate the quantum computation, the C wrapper QuEST.jl which utilises the library QuEST is used. QuEST.jl calls the C through a precompiled versions (v3.7) of QuEST, with a number of features on the Julia level to prevent run-time crashes. There are some experimental features, but basic usage and specifically calls relevant to VeriQuEST.jl are sound. The user takes advantage of the Julia just-in-time compilation and can run scripts or functions per standard Julia usage. There are many functions not automatically exported to the user but available in the normal way (e.g., calling VeriQuEST.<function/struct/type-call>). QuEST can be compiled to run on multi-core, multi-threads, distributed and GPU clusters, however here QuEST.jl uses pre-compiled binaries suited for multi-threading only.

DRAFT

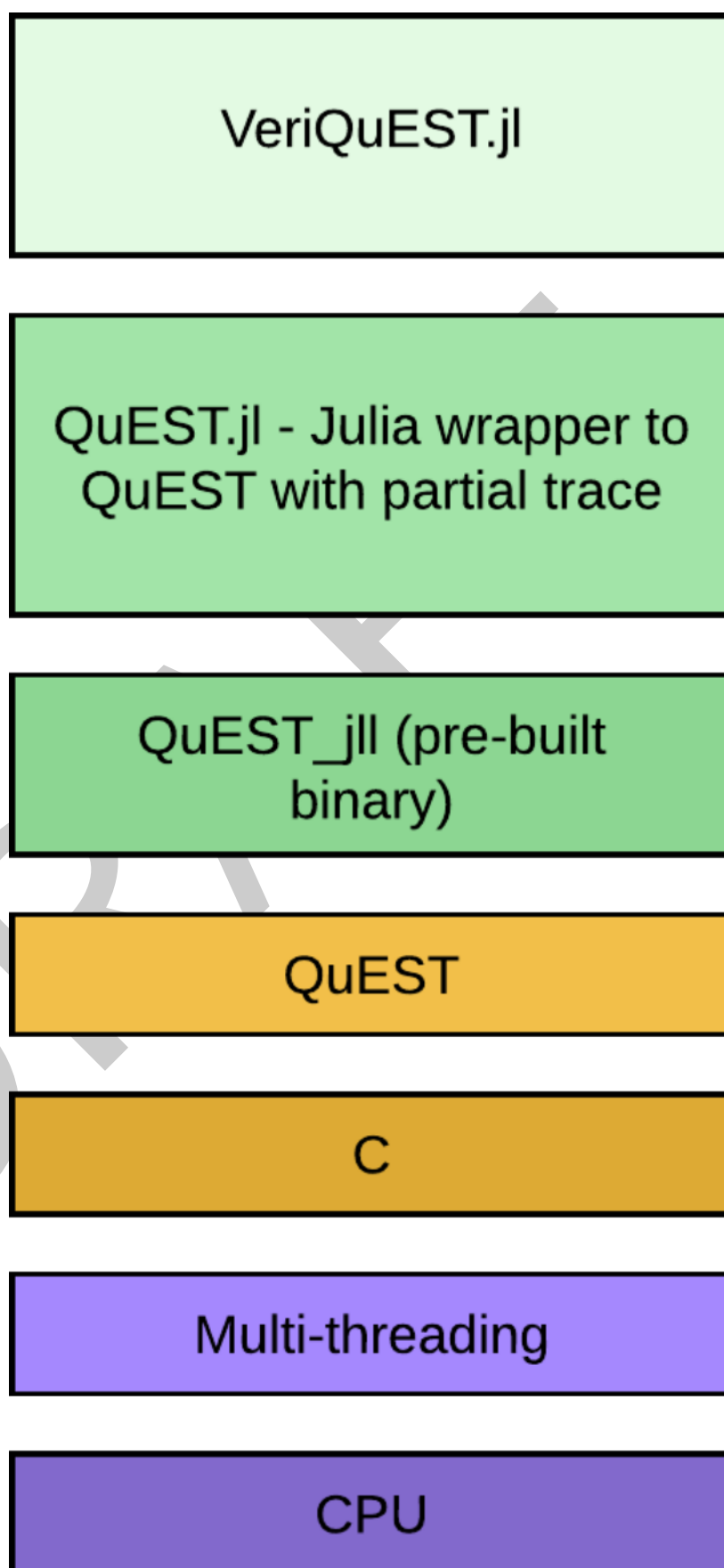


Figure 1: Software architecture

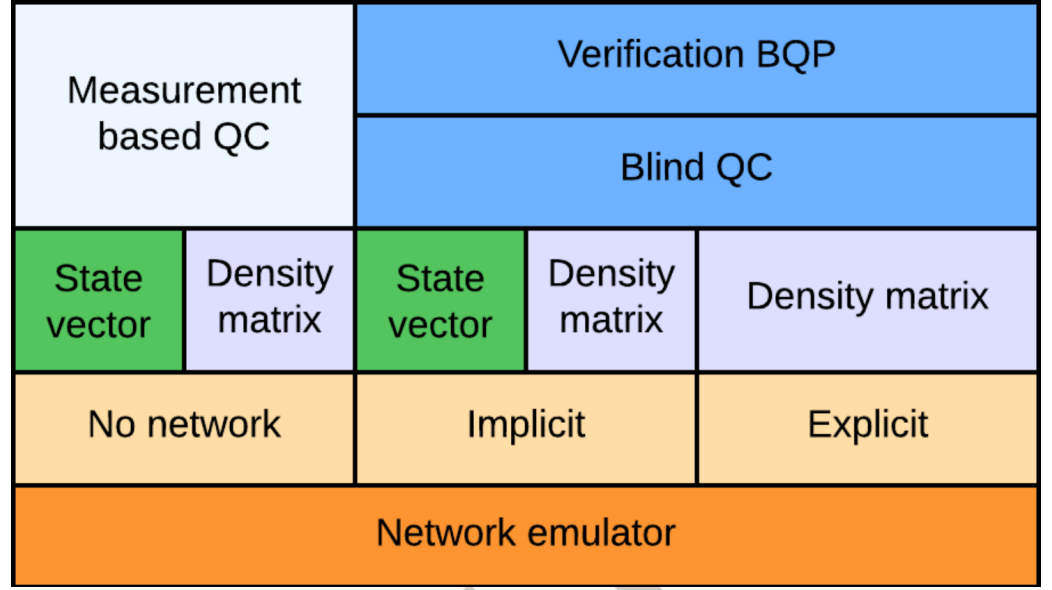


Figure 2: Network architecture

Measurement-Based Quantum Computation (MBQC)

In the MBQC framework, a quantum algorithm can be represented as a set $\{(G, I, O), \vec{\phi}\}$, where (G, I, O) denotes the quantum resource and $\vec{\phi}$ is a set of measurement angles. The triplet (G, I, O) signifies an *open graph*, i.e., graphs characterised by the presence of flow (Danos & Kashefi, 2006). Single-qubit adaptive measurements are performed sequentially on the vertices with measurement operator $\{|+\theta_j\rangle\langle+\theta_j|, |-\theta_j\rangle\langle-\theta_j|\}$, measured on vertex j , where $|\pm\theta\rangle := (|0\rangle \pm e^{i\theta}|1\rangle)/\sqrt{2}$ and $\theta_j = \phi_j + \delta$ for a correction δ that is calculated by our tool. For an illustration, see Figure 3.

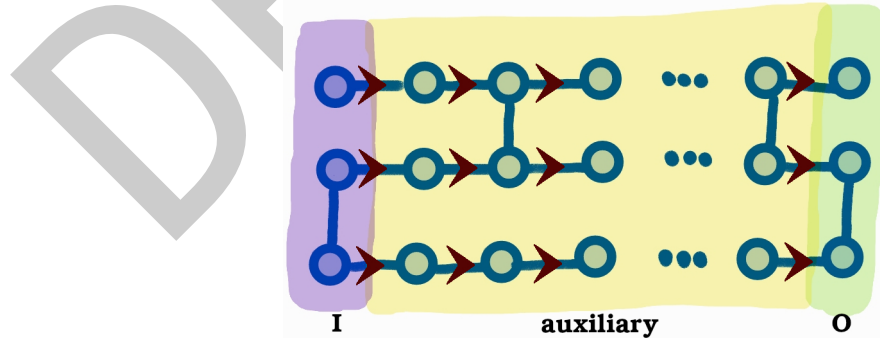


Figure 3: The triplet (G, I, O) representing graph state for the quantum resource, where $G = (V, E)$ is the graph, I is a set of input nodes, and O is a set of output nodes. Each node represents qubit with state $(|0\rangle + |1\rangle)/\sqrt{2}$ and each edge represent controlled- Z operation operated to the corresponding nodes. Measurements in the XY -plane of Bloch sphere are performed from the left to the right. The arrows indicate the flow $f : O^c \rightarrow I^c$ which induces partial ordering of the measurements. Input nodes I may be initialised to an arbitrary quantum state ρ and the output nodes O is the final output that can be classical or quantum – if left unmeasured.

Configuration

In this example a simple path graph is implemented.

```

using Pkg
Pkg.activate(".")
using VeriQuEST
using Graphs
# Set up input values
graph = Graph(2)
add_edge!(graph,1,2)

io = InputOutput(Inputs(),Outputs(2))
qgraph = QuantumGraph(graph,io)
function forward_flow(vertex)
    v_str = string(vertex)
    forward = Dict{
        "1" =>2,
        "2"=>0)
    forward[v_str]
end
flow = Flow(forward_flow)
measurement_angles = Angles([ $\pi/2$ , $\pi/2$ ])

68 To run the MBQC pattern

mbqc_comp_type = MeasurementBasedQuantumComputation(qgraph,flow,measurement_angles)
no_network = NoNetworkEmulation()
dm = DensityMatrix()
sv = StateVector()
ch = NoisyChannel(NoNoise(NoQubits()))
cr = MBQCRound()

69 Run as state vector

mg_dm = compute!(mbqc_comp_type,no_network,sv,ch,cr)

70 and as density matrix

mg_sv = compute!(mbqc_comp_type,no_network,dm,ch,cr)

```

71 Blind Quantum Computation (BQC)

72 Blind Quantum Computation (BQC) is a type of cryptographic protocol that enables a client,
 73 referred to as Alice, to securely delegate her quantum computing tasks to a powerful quantum
 74 server, called Bob, while ensuring privacy. Within this setup, Bob cannot infer Alice's algorithm
 75 nor her measurement outcomes. The most notable BQC protocol is the Universal Blind
 76 Quantum Computation (UBQC)([Broadbent et al., 2009](#)). This framework allows Alice to
 77 privately delegate her MBQC algorithm to Bob through a series of quantum and classical
 78 interactions between them.

79 Remarkably, the UBQC promises composable security that does not rely on traditional computa-
 80 tional assumptions ([Dunjko et al., 2014](#)). Instead, the privacy stems directly from the intrinsic
 81 properties of quantum measurement. In this model, Alice requires only limited quantum
 82 resources: the ability to prepare specific quantum states, $|+\theta\rangle$, and to send them to Bob,
 83 underscoring the necessity for a robust quantum network. The essence of maintaining Alice's
 84 privacy lies in obfuscating the measurement angles ϕ and the associated outcomes through
 85 the strategic use of randomness, as illustrated in [Figure 4](#).

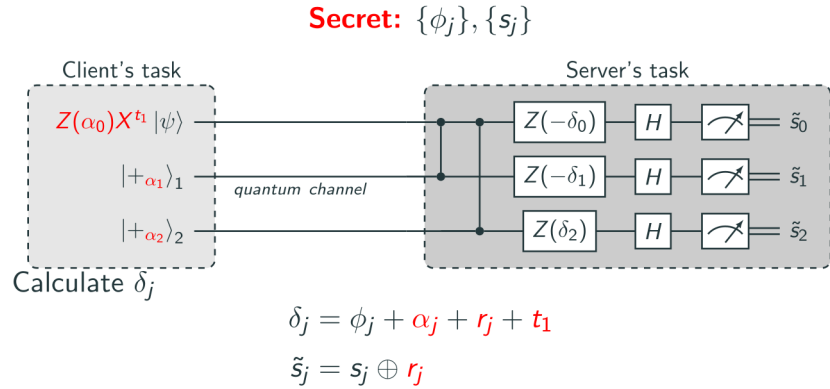


Figure 4: This circuit shows task separation between the client and the hiding strategy, where measurement angles $\{\phi_j\}$ and initial outcomes $\{s_j\}$ are obscured. Additional randomness (highlighted in red) is introduced for hiding the secrets, which later neutralised by adjustments in δ_j to hide the measurement angles ϕ_j . Notice that the actual measurement outcomes s_j remain exclusively accessible to Alice due to random binary r_j . Note that the first state $|\psi\rangle$ indicates an arbitrary input state that is encrypted by random phase α_0 and random bit flip t_1 ; thus, $\delta_0 = (-1)^{t_1} + \alpha_0 + r_0$. In the MBQC language, this circuit is equivalent with the three nodes path graph with angles $\{\alpha_0, \alpha_1, \alpha_2\}$.

In BQC, delegating quantum tasks privately highlights the importance of client state preparation, state transfer, accessibility, correctness, and security. To address these critical aspects efficiently, end users are presented with two approaches for emulating BQC algorithms:

- **Implicit network emulation.** In this option, we do not explicitly emulate the quantum state of the client or the quantum network. Instead, the states prepared on the server side already take into account the state transfer. This approach is useful for studying the noise effect on the computation. For an example, see the code below.

```
ubqc_comp_type = BlindQuantumComputation(qgraph,flow,measurement_angles)
dm = DensityMatrix()
implicit_network = ImplicitNetworkEmulation()
cr = ComputationRound()
mg_imp = compute!(ubqc_comp_type,implicit_network,dm,ch,cr)
```

- **Explicit network emulation.** In this option, the quantum state of the client and the state transfer are explicitly emulated. The quantum network is simulated using remote entanglement operators, which can also be specified by the end user. This is made possible by operators operating between the client and the server in the joint Hilbert space of the client and the server, denoted as $\mathcal{H}_c \otimes \mathcal{H}_s$. This approach is useful for studying the effects of noise on the protocol as well as examining security in greater detail. Additionally, users can access the state of each party: the client's state in \mathcal{H}_c and the server's state in \mathcal{H}_s , enabled by the partial trace operation. For an example, see the code below. [JON:CODE SNIPPET with explicit client]

```
bell_pair_explicit_network = BellPairExplicitNetwork()
mg_bp = compute!(ubqc_comp_type,bell_pair_explicit_network,dm,ch,cr)
```

Verification protocols [DOM]

The verification protocol can be added on top of BQC, certifying the executed quantum algorithm.

Introduce verification and.

Our tool provides several verification protocols that are ready for users to use:

```

107     ■ Dominik's BQP verification (Leichtle et al., 2021)
108     ■ Maybe Fitsimons verification with hidden trap?
109 To run the verification on a noisless server
    flow = Flow(forward_flow)
    measurement_angles = Angles([ $\pi/2$ , $\pi/2$ ])
    total_rounds = 10
    computation_rounds = 1
    trapification_strategy = TestRoundTrapAndDummycolouring()
    ct = LeichtleVerification(
        total_rounds,
        computation_rounds,
        trapification_strategy,
        qgraph,flow,measurement_angles)
    nt_bp = BellPairExplicitNetwork()
    nt_im = ImplicitNetworkEmulation()
    st = DensityMatrix()
    ch = NoisyChannel(NoNoise(NoQubits()))
110 Run on an implicit network
    ver_res1 = run_verification_simulator(ct,nt_bp,st,ch)
111 and an explicit network
    ver_res2 = run_verification_simulator(ct,nt_im,st,ch)
112 To get results
    get_tests(ver_res2)
    get_computations(ver_res2)
    get_tests_verbose(ver_res2)
    get_computations_verbose(ver_res2)
    get_computations_mode(ver_res2)
113 ## Noiseless and noisy simulations
114
115 In the study of quantum verification protocols, it is crucial to be able to simulate bot
116 qubit system has dimension  $2^n$ , making it easier to check if the protocols are correct
117 qubit system has dimension  $2^n \times 2^n$ , however, they are excellent for understandin
118
119 ### Implicit noise channels
120
121 Here are examples of bit flip noise and angles added to measurement basis
122
123 ```julia
124 # Post Angle Update
125 angle = 0.1
126 model = PostAngleUpdate(SingleQubit(),angle)
127 ch = NoisyChannel(model)
128 vr = run_verification_simulator(ct,nt_im,st,ch)
129 vr = run_verification_simulator(ct,nt_bp,st,ch)
130 st = StateVector()
131 vr = run_verification_simulator(ct,nt_im,st,ch)
132
133
134
135 # Add bit flip

```

```

136 bit_flip_prob = 0.5
137 model = AddBitFlip(SingleQubit(),bit_flip_prob)
138 ch = NoisyChannel(model)
139 vr = run_verification_simulator(ct,nt_im,st,ch)
140 vr = run_verification_simulator(ct,nt_bp,st,ch)
141 st = StateVector()
142 vr = run_verification_simulator(ct,nt_im,st,ch)

143 Explicit noise channels
144 Here are examples of damping, dephasing and depolarising noise.

# Prob scaling
p_scale = 0.1
p = [p_scale*rand() for i in vertices(graph)]

# Damping
model = Damping(SingleQubit(),p)
ch = NoisyChannel(model)
vr = run_verification_simulator(ct,nt_im,st,ch)
vr = run_verification_simulator(ct,nt_bp,st,ch)

# Dephasing
model = Dephasing(SingleQubit(),p)
ch = NoisyChannel(model)
vr = run_verification_simulator(ct,nt_im,st,ch)
vr = run_verification_simulator(ct,nt_bp,st,ch)

# Depolarising
model = Depolarising(SingleQubit(),p)
ch = NoisyChannel(model)
vr = run_verification_simulator(ct,nt_im,st,ch)
vr = run_verification_simulator(ct,nt_bp,st,ch)

# Pauli
p_xyz(p_scale) = p_scale/10 .* [rand(),rand(),rand()]
p = [p_xyz(p_scale) for i in vertices(graph)]
model = Pauli(SingleQubit(),p)
ch = NoisyChannel(model)
vr = run_verification_simulator(ct,nt_im,st,ch)
vr = run_verification_simulator(ct,nt_bp,st,ch)

# Vector of noise models
model_vec = [Damping,Dephasing,Depolarising,Pauli]
p_damp = [p_scale*rand() for i in vertices(graph)]
p_deph = [p_scale*rand() for i in vertices(graph)]
p_depo = [p_scale*rand() for i in vertices(graph)]
p_pauli = [p_xyz(p_scale) for i in vertices(graph)]
prob_vec = [p_damp,p_deph,p_depo,p_pauli]

models = Vector{AbstractNoiseModels}()
for m in eachindex(model_vec)
    push!(models,model_vec[m](SingleQubit(),prob_vec[m]))
end
ch = NoisyChannel(models)

```



```
vr = run_verification_simulator(ct,nt_im,st,ch)
vr = run_verification_simulator(ct,nt_bp,st,ch)
```

Future plans

Adding the upcoming verification protocols in the library. Multi-parties computation as per. Graph with gflow, more choice on the measurement plane. Minimal resource of MBQC (Lazy 1WQC). Integrating realistic noise. Mathematica integration.

Acknowledgements

We acknowledge contributions from QSL, NQCC,...???

References

- Broadbent, A., Fitzsimons, J., & Kashefi, E. (2009). Universal blind quantum computation. *2009 50th Annual IEEE Symposium on Foundations of Computer Science*, 517–526. <https://doi.org/10.1109/FOCS.2009.36>
- Danos, V., & Kashefi, E. (2006). Determinism in the one-way model. *Physical Review A*, 74(5), 052310. <https://doi.org/10.1103/PhysRevA.74.052310>
- Dunjko, V., Fitzsimons, J. F., Portmann, C., & Renner, R. (2014). Composable security of delegated quantum computation. *International Conference on the Theory and Application of Cryptology and Information Security*, 406–425. https://doi.org/10.1007/978-3-662-45608-8_22
- Gheorghiu, A., Kapourniotis, T., & Kashefi, E. (2019). Verification of quantum computation: An overview of existing approaches. *Theory of Computing Systems*, 63, 715–808. <https://doi.org/10.1007/s00224-018-9872-3>
- Jones, T., Brown, A., Bush, I., & Benjamin, S. C. (2019). QuEST and high performance simulation of quantum computers. *Scientific Reports*, 9(1), 10736. <https://doi.org/10.1038/s41598-019-47174-9>
- Leichtle, D., Music, L., Kashefi, E., & Ollivier, H. (2021). Verifying BQP computations on noisy devices with minimal overhead. *PRX Quantum*, 2, 040302. <https://doi.org/10.1103/PRXQuantum.2.040302>
- Nielsen, M. A., & Chuang, I. L. (2011). *Quantum computation and quantum information: 10th anniversary edition*. Cambridge University Press. ISBN: 9781107002173
- Raussendorf, R., & Briegel, H. J. (2001). A one-way quantum computer. *Physical Review Letters*, 86(22), 5188. <https://doi.org/10.1103/PhysRevLett.86.5188>