



# Summary

We present here the Julia package, `RobustBlindVerification.jl` (RBV) to emulate quantum verification protocols in ideal and noisy settings. The RBV package is a quantum computing emulator, which uses the `QuEST.jl` package (from QuEST, a C library wrapped packaged using `BinaryBuilder.jl` for reproducible 3rd-party binaries) to perform the quantum operations. RBV is based on the measurement based quantum computing (MBQC) paradigm, and uses universal blind quantum computing to hide computations from detection. Verification is implemented via trapification strategies and multiple rounds of computation. The verification algorithm is implemented to account for noise and naive malicious behaviour or uncorrelated noise.

Refine and proof ... [[@PRXQuantum.2.040302](#)]

## Statement of need

The rise of quantum computers gives rise to a variety of path dependent access points to such computers. Delegated quantum computing may likely be the dominant means most can access a quantum computer. One party's access may require secrecy in their computation. Another party, or the same, may need to verify results or computations are trustworthy. Formal methods in quantum verification have been developed in theory. Many of these methods rely on quantum networks, mid-circuit measurement and qubit capabilities beyond the current near-term offering. In preparation for the aforementioned to become reality, quantum emulators offer cheap computational access in many toy problems, along with an ability to test theoretical results.

RBV is to-date (and at the authors undersatnding) the only emulator which implements a simulated blind quantum computation, let alone the verification protocol [cite]. Many quantum computing emulators also only focus on the gate-based model, whilst some implement MBQC [Cite] many do not allow for noise models beyond uncorrelated models, which do not utilise density matrix backends. Many emulators are also focused on python-based libraries [cite]. RBV is based in Julia and calls on a C library. QuEST is a remarkable library that is capable of use agnostically to the machinery accessing the library.

What else ....

Cite .....

## State of the field

Verification is becoming a core component in the quantum computing stack, in addition to confirming computation is done as expected, verification can act to determine the noisiness of a given machine.

Add more here ...

## Core features and functionality

Core features:

1. Run MBQC
2. Run UBQC
3. Run Verification noiseless
4. Run Verification with server that changes the measurement angle (malicious)
5. Run Verification with suite of standard decoherence models with Kraus maps and density matrix mixing
6. Run test rounds on random graphs for general analysis
7. Should be useful in multithreading/cores
8. Could be used with GPU with certain algorithms
9. ...
10. Uses Julia's multiple dispatch to implement different noise models efficiently
11. Should have a generic idea of noise if user wants to implement their own

## Examples

1. Present examples using simple graphs
2. Present Grover MBQC as example

3. Present Rgover in verification
4. Present Random graphs in noise expression

Activate the Julia project of the local directory location

```
using Pkg; Pkg.activate(".")
```

Add `RobustBlindVerification.jl`

```
] add RobustBlindVerification
```

To run, first decide the computational backend, either a state vector or density matrix. These are structs named `StateVector` and `DensityMatrix`.

```
state_type = DensityMatrix()
```

Choose the number of rounds and the number of computational rounds (the number of test rounds is the difference of computation from total).

```
total_rounds,computation_rounds = 100,50
```

RBV uses `Graphs.jl` and `MetaGraphs.jl`, so defining a graph uses that syntax. In this example we use the MBQC version of the 2 qubit Gover oracle algorithm.

```
num_vertices = 8
graph = Graph(num_vertices)
add_edge!(graph,1,2)
add_edge!(graph,2,3)
add_edge!(graph,3,6)
add_edge!(graph,6,7)
add_edge!(graph,1,4)
add_edge!(graph,4,5)
add_edge!(graph,5,8)
add_edge!(graph,7,8)
```

We can specify any classical inputs to be loaded into the algorithm as well the output qubits.

```
input = (indices = (), values = ())
output = (7,8)
```

There are algorithms for automatic flow detection, these have not been implemented, so the user defines a function for the so-called *forward flow*. The function is used to determine the *backward flow*.

```
function forward_flow(vertex)
  v_str = string(vertex)
  forward = Dict{
    "1" =>4,
    "2" =>3,
    "3" =>6,
    "4" =>5,
    "5" =>8,
    "6" =>7,
    "7" =>0,
    "8" =>0)
  forward[v_str]
end
```

Note: the internal structure of the `forward_flow` does not matter, as long as vertex input returns vertex output ( `Int` type) and these values correspond to the graph, then the user can define its choice of I/O for that function.

The secret angles are defined next. For the Grover algorithm, the angles are crucial in its operation.

```
function generate_grover_secret_angles(search::String)

  Dict{"00"=>(1.0*π,1.0*π),"01"=>(1.0*π,0),"10"=>(0,1.0*π),"11"=>(0,0)) |>
  x -> x[search] |>
  x -> [0,0,1.0*x[1],1.0*x[2],0,0,1.0*π,1.0*π] |>
  x -> Float64.(x)
end
```

The function `generate_grover_secret_angles` defines the angles associated to each vertex

in the `graph` which is dependent on the `search` input. Here we will define search as `11`

```
search = "11"
secret_angles = generate_grover_secret_angles(search)
```

Names of inputs can vary, but they are all stored in a `NamedTuple` as

```
para= (
    graph=graph,
    forward_flow = forward_flow,
    input = input,
    output = output,
    secret_angles=secret_angles,
    state_type = state_type,
    total_rounds = total_rounds,
    computation_rounds = computation_rounds)
```

To run the MBQC and UBQC we only need `para`

```
mbqc_outcome = run_mbqc(para)
ubqc_outcome = run_ubqc(para)
```

For the verification protocol we specify the type of *Server*

```
vbqc_outcome = run_verification_simulator(TrustworthyServer(),Verbose(),para)
vbqc_outcome = run_verification_simulator(TrustworthyServer(),Terse(),para)
```

Here the server is `TrustworthyServer` , so all results are done in a noiseless state and the server agent (simulated server not real one) is not interfering with the computation. The `Verbose` flag dictates the output to include extra details of the results [add content]. The `Terse` flag returns only whether the computation was `Ok` or `Abort` . For a `MaliciousServer` , meaning one that adds an additional angle to the updated angle basis for measurement.

```
malicious_angles =  $\pi/2$ 
malicious_vbqc_outcome = run_verification_simulator(MaliciousServer(),Verbose(),para,r
```

The `malicious_angles` can be a single angle, applied uniformly to all qubits, or a vector of angles applied specifically to each qubit according to order from `1` to `num_vertices`.

To add noise to the verification, there are some ready-made models to choose from, damping, dephasing, depolarising, pauli, (need to finalise these: Kraus, two qubit, density matrix mixing, N qubit kruas).

Depending on the noise, there are constraints based on the theory of the noise, so lets scale noise `p_scale = 0.05`. Lets do a damping noise model.

```
p = [p_scale*rand() for i in vertices(para[:graph])]
model = Damping(Quest(),SingleQubit(),p)
server = NoisyServer(model)
vbqc_outcome = run_verification_simulator(server,Verbose(),para)
```

Here we generate `n` random probabilities, which will be applied to each qubit in the graph. A noise model is defined by the `struct` representing the noise, in this case, `Damping()`. Each noise model only takes three arguments. In future plans different quantum computing backends will be used (BYOQ - Bring Your Own Quantum), so the QC emulator is specified, `Quest` in this case. The noise model is applied to individual qubits, the second argument is `SingleQubit`, as opposed to `TwoQubit` or `NQubit`, the latter are not in use yet. Finally, the probability, like before this can be a scalar or a vector, the scalar being applied uniformly as before. The `model` is a container, which is stored inside the type of server, here `NoisyServer`. Then the same simulator is called `run_verification_simulator` to run the protocol, but with the specified noise model. The following show a similar implementation.

```

# Dephasing
p = [p_scale*rand() for i in vertices(para[:graph])]
model = Dephasing(Quest(),SingleQubit(),p)
server = NoisyServer(model)
vbqc_outcome = run_verification_simulator(server,Verbose(),para)

# Depolarising
p = [p_scale*rand() for i in vertices(para[:graph])]
model = Depolarising(Quest(),SingleQubit(),p)
server = NoisyServer(model)
vbqc_outcome = run_verification_simulator(server,Verbose(),para)

# Pauli
p_xyz(p_scale) = p_scale .* [rand(),rand(),rand()]
p = [p_xyz(p_scale) for i in vertices(para[:graph])]
model = Pauli(Quest(),SingleQubit(),p)
server = NoisyServer(model)
vbqc_outcome = run_verification_simulator(server,Verbose(),para)

```

The Pauli noise acts on the x,y and z axis so there are three probabilities for each qubit. In addition to each simulator being able to run a single noise model, we can run a list of noise models.

```

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

models = Vector{NoiseModels}()
for m in eachindex(model_vec)
    push!(models,model_vec[m](Quest(),SingleQubit(),prob_vec[m]))
end
server = NoisyServer(models)
vbqc_outcome = run_verification_simulator(server,Verbose(),para)

```

If we want to run general Kraus maps, we can do so on single, double and `n` qubit levels -

**NEED TO DO.** If we have a density matrix with a specified noise model, we can mix that with the algorithm as well - **NEED TO DO.**

## Inside the verification simulator

Let us look closer at the `TrustWorthy` `Verbose` simulator function.

```
function run_verification_simulator(::TrustworthyServer, ::Verbose, para)
...
end
```

Currently, the trapification strategy is to generate a random colouring of the presented graph, based on a greedy heuristic [Cite graphs package?]. Naively the choice is made such that out of `100` repetitions, the best coloring is picked. The computation round is simply one colour for all vertices.

```
computation_colours = ones(nv(para[:graph]))
```

For each coloured vertex, a 2-colour graph is formed, that of the colored vertex and a colour for the remaining vertices. Hence, a 4-colour graph, can be split into 4 distinct graphs according to the colouring.

```
test_colours = get_vector_graph_colors(para[:graph]; reps=reps)
```

To determine the number of acceptable failed rounds a value is computed dependent on the BQP error, the number of colours and the distribution of test rounds to total rounds.

```
chroma_number = length(test_colours)
bqp = InherentBoundedError(1/3)
test_rounds_theshold = compute_trap_round_fail_threshold(para[:total_rounds], para[:con
```

Recall that the `forward_flow` is user defined, then the `backward_flow` is numerically computed./



```
backward_flow(vertex) = compute_backward_flow(para[:graph], para[:forward_flow], vertex)
```

These computations along with the remaining values from `para` are stored in a `NamedTuple` which is used to create a `struct` wrapping all variables in a defined type.

```
p = (  
    input_indices = para[:input][:indices],  
    input_values = para[:input][:values],  
    output_indices = para[:output],  
    graph = para[:graph],  
    computation_colours = computation_colours,  
    test_colours = test_colours,  
    secret_angles = para[:secret_angles],  
    forward_flow = para[:forward_flow],  
    backward_flow = backward_flow)  
  
client_resource = create_graph_resource(p)
```

The `round_types` is a random permutation of `structs` `ComputationRound` and `TestRound` on the exact number of each.

```
round_types = draw_random_rounds(para[:total_rounds], para[:computation_rounds])
```

The type held by `round_types` is a `Vector`, which is iterated through to execute rounds determined by element of the iterator. The function `run_verification` executes the protocol for each round and return a vector of `MetaGraphs` for each round.

```
rounds_as_graphs = run_verification(  
    Client(), Server(),  
    round_types, client_resource,  
    para[:state_type])
```

The `MetaGraph` is the core data structure for any given MBQC computation. Due to its inherent graph and vertex properties, multiple dispatch was used to perform appropriate computation across vertices in a graph. Results for the `Verbose` flag are the outcomes with the results for acceptable round or failed round.

```

test_verification = verify_rounds(Client(),TestRound(),Terse(),rounds_as_graphs,t
computation_verification = verify_rounds(Client(),ComputationRound(),Terse(),round
test_verification_verb = verify_rounds(Client(),TestRound(),Verbose(),rounds_as_g
computation_verification_verb = verify_rounds(Client(),ComputationRound(),Verbose
mode_outcome = get_mode_output(Client(),ComputationRound(),rounds_as_graphs)

```

These results are returned as a `NamedTuple`

```

return (
    test_verification = test_verification,
    test_verification_verb = test_verification_verb,
    computation_verification = computation_verification,
    computation_verification_verb = computation_verification_verb,
    mode_outcome = mode_outcome)

```

Let's look closer at the verification function, `run_verification`.

```

function run_verification(::Client,::Server,
    round_types,client_resource,state_type)
    ...
    round_graphs
end

```

As before, we take a `Client` and a `Server` type, the server can be replaced with a different type given a desired computation. We run the verification protocol over a vector of rounds.

```

round_graphs = []
for round_type in round_types
    ...
    client_meta_graph = ...
    push!(round_graphs,client_meta_graph)
end

```

For each `round_type` in the vector `round_types`, note `round_type` is either `TestRound` or `ComputationRound` and Julia's multiple dispatch will perform the appropriate functions based on said type, the computation is performed, details are mutated onto the property graph, `client_meta_graph` and stored in the vector `round_graphs`. Within each round several

computations take place. First,

```
client_meta_graph = generate_property_graph!(  
    Client(),  
    round_type,  
    client_resource,  
    state_type)
```

a `client_meta_graph` is generated with the function `generate_property_graph!` which takes the `Client`, `round_type`, `client_resource` and `state_type` as inputs. This function will be discussed further below. The `client_meta_graph` contains round specific values, the vertex angles, the forward and backward vertices according to the flow, allocation variables for measurement outcomes, the initialised state quantum register backed by QuEST. Note that no vertex is entangled, only initialised according the UBQC methodologies of client and server separation. To effect this distinction the client quantum register and simple graph are extracted.

```
client_graph = produce_initialised_graph(Client(),client_meta_graph)  
client_qureg = produce_initialised_qureg(Client(),client_meta_graph)
```

When the server resource is created, `create_resource`, using the `Server`, `client_graph` and `client_qureg` as inputs. The server side resource is where noise is added if that is the case and where the circuit is entangled according to the edge map of the underlying MBQC graph structure.

```
server_resource = create_resource(Server(),client_graph,client_qureg)
```

From here the server quantum state, or register is extracted from the server container, `server_resource`, thus, separation of the client and the server is effected. We even extract the number of qubits based on the server register.

```
server_quantum_state = server_resource["quantum_state"]  
num_qubits_from_server = server_quantum_state.numQubitsRepresented
```

Now for the given round, which has entirely driven the specific values of the property graph,

the UBQC computation can be performed.

```
run_computation(Client(),Server(),client_meta_graph,num_qubits_from_server,server_quantum_state)
```

The underlying register is still in effect, so to save loading and memory usage, we clear the state of the server register, ready to take on the next round of client side computation.

```
initialise_blank_quantum_state!(server_quantum_state)
```

Now we come back to pushing the `client_meta_graph`, which now holds all of the measurement outcomes, to the `round_graphs` vector. This is then returned at the end of the function call.

```
push!(round_graphs,client_meta_graph)
```

## Citations

Leave here for reference

Citations to entries in paper.bib should be in

[rMarkdown](#)

format.

If you want to cite a software repository URL (e.g. something on GitHub without a preferred citation) then you can do it with the example BibTeX entry below for @fidgit.

For a quick reference, the following citation commands can be used:

- `@author:2001` -> "Author et al. (2001)"
- `[@author:2001]` -> "(Author et al., 2001)"
- `[@author1:2001; @author2:2001]` -> "(Author1 et al., 2001; Author2 et al., 2002)"

# Figures

Figures can be included like this:

Caption for example figure. `figure.abel{fig:example}`

and referenced from text using `\autoref{fig:example}`.

Figure sizes can be customized by adding an optional second parameter:

Caption for  
example figure.

# Acknowledgements

We acknowledge contributions from Brigitta Sipocz, Syrtis Major, and Semyeong Oh, and support from Kathryn Johnston during the genesis of this project.

# References