

Quantum Simulators

Test and overview of the most promising simulators

Test report of the DFN WiN-Lab in Erlangen, as of 15.07.2022

Content

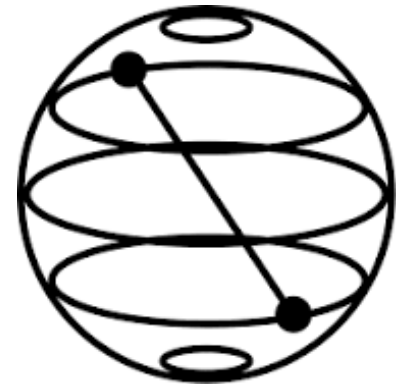
QISKIT	2
QUANTUM NETWORK EXPLORER.....	8
SIMULAQRON	11
NETSQUID	15
QUNETSIM.....	18
SILQ.....	22
GOOGLE QUANTUM AI (CIRQ).....	25
SQUANCH	27
SEQUENCE.....	30
QKDNETSIM	33
QKDSIMULATOR.....	35
AMAZON (BRAKET)	39
QUANTUM PROGRAMMING STUDIO	41
MICROSOFT AZURE QUANTUM/QKD/Q#.....	45

Foreword

This report presents various simulators that have been examined in more detail in individual tests. These are platforms, software development kits (SDKs), standalone simulators and simulation engines. For all simulators examined, their properties are described and it is explained for which application they are most suitable. In addition, there is an application example that has been tested in a concrete installation; the fastest or most practical installation method is also explained. Each test report is concluded with a conclusion.

Qiskit

Repository	https://github.com/Qiskit/qiskit
Language	Python
OS	Cross-platform
Type	Software Development Kit (SDK)
Focus	Pulse / Circuits
Feature	Web platform/IDE available ¹
License	Apache License 2.0
Website	https://qiskit.org
Registration	no



Qiskit is an SDK in Python provided by IBM for creating and executing algorithms and circuits and for generally working with noisy quantum computers. Qiskit can be installed completely independently on any **popular operating system** and can run simulations independently. It consists of several components, although only two main ones will be discussed here.

Qiskit Terra is the basic component on which all the others are built. Terra provides the ability to assemble quantum programs at the circuit and **pulse** level, and manages the execution of batches of experiments on remote devices and backend communication.

Qiskit Aer offers various simulators for quantum computers with realistic noise models hosted **locally** on the user's device or **HPC resources** available via the cloud. IBM itself calls its simulator services "advanced cloud-based classical emulators of quantum systems". At this point, it should already be mentioned that Qiskit is closely linked to IBM's "quantum cloud" and that a simulation can be run locally on the notebook as well as on "classic" hardware in the IBM cloud. The highlight here, however, is that² real quantum hardware is also available free of charge to a limited extent and you can compare simulation and real process.

Interface(s) – It should also be emphasized that you also have the choice of using a local user interface or using IBM's [Web Interface](#) both of which [Jupyter Notebooks](#) support. Advantage of the Web Composer (Figure 1) is the drag & drop editor, which transfers the designed circuit directly to Python and displays result forecasts in real time.

Backends represent either a simulator or a real quantum computer and are responsible for executing quantum circuits and returning results. As with some other software simulators³ they can be exchanged. Table 1 shows the backend simulators provided by IBM. The state vector is the default simulator here.

¹ Not mandatory

² Latency; not all processors and locations are free

³ A software simulator is the term used to describe the entire framework/SDK, which operates an interchangeable simulator in the sense of a simulation core as a backend

Qiskit

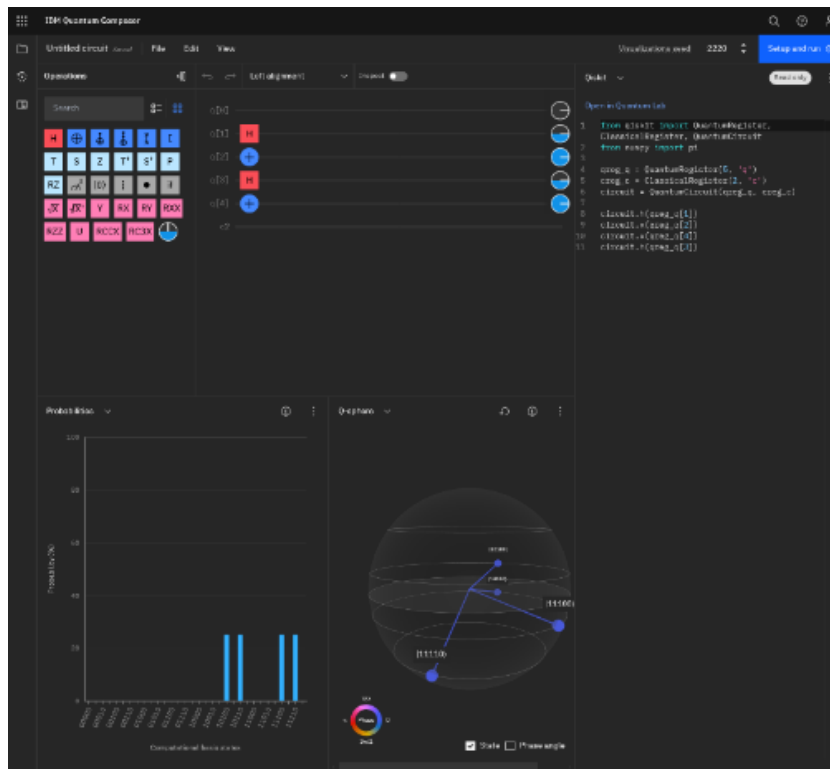


Figure 1: IBM's Web Composer

Table 1: Available Backend Simulators

<p>Statevector Type: Schrödinger wavefunction QuBits: 32 Noise modeling: Yes</p>	<p>Stabilizer Type: Clifford QuBits: 5000 Noise modeling: Yes (Clifford only)</p>	<p>Extended stabilizer Type: Extended Clifford (e.g., Clifford+T) QuBits: 63 Noise modeling: No</p>	<p>MPS Type: Matrix Product State QuBits: 100 Noise modeling: No</p>	<p>QASM Type: General, context-aware QuBits: 32 Noise modeling: Yes</p>
--	---	---	--	---

Community – A major advantage of the framework is its large community and the associated variety of media. There are already a variety of media and opportunities for interested parties to deal with the world of quantum computing. The Qiskit Foundation itself maintains a YouTube channel with many tutorials, talks and background information on the topic and regularly organizes quantum challenges, camps, hackathons and seminars. IBM also maintains its own [workspace](#) in Slack, which you can join.

Installation – If you are afraid of an installation at first, you can start directly in [IBM's Quantum](#)⁴ Lab. If you prefer to work locally, you can use `pip install qiskit` to initiate the installation. The manufacturer recommends a virtual environment with [Anaconda](#). Since we only want to demonstrate a small circuit here, we make do with the pre-installed Python distribution and create a virtual environment with `python3 -m venv qiskit-env`⁵ and install `matplotlib` in addition to `qiskit` to be able to visualize the results.

Simulation on classic hardware – We now create a relatively simple circuit with 3 qubits, which we put into superposition using a Hadamard gate and measure directly afterwards. We start Python and import the entire library for simplicity `from qiskit import *`. Then we create a circuit with 3 qubits and 3 classical bits each to store the result there `qc = QuantumCircuit(3, 3)`. The QuBits are initially in

⁴ Registration required

⁵ Activation with `source qiskit-env/bin/activate`

Qiskit

the state $|0\rangle$, which is why a measurement would also result in 0. We therefore loop through all QuBits and add the H-gate and a measurement in each case. Since all QuBits are now in the state, we can now expect a random value of 0 or 1 through measurement. With $|+\rangle^6_{qc}$.draw() the generated circuit can be drawn, see Figure 2. After selecting the backend (Table 1) the "job" is executed. Finally, the results are plotted in a histogram.

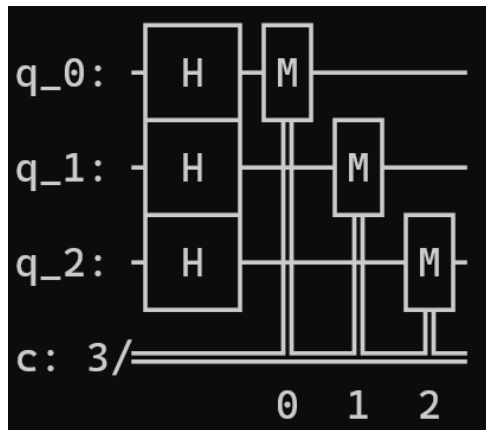


Figure 2: Example of a simple circuit

```
1 from qiskit import *
2 from qiskit.tools.visualization import plot_histogram
3 qc = QuantumCircuit(3,3)
4 for j in range(3):
5     qc.h(j)           #Hadamard Gate
6     qc.measure(j,j)  #Messung
7 qc.draw()           #Zeichnen the Figure 2
8 backend = Aer.get_backend('qasm_simulator') #Wahl of the backend job =
9 execute(qc, backend)
10     #Ausführenplot_histogram(job.result().get_counts()) #Plotten of
11 the histogram
```

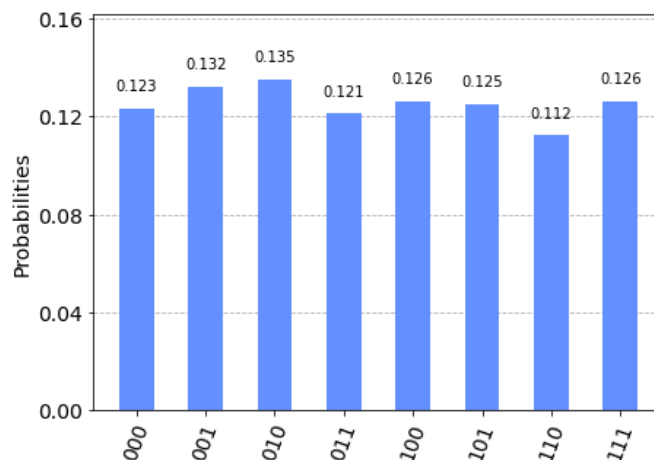


Figure 3: Result histogram of the example circuit

The histogram from our example is intended to show that by applying a Hadamard gate, a superposition state is created, in which the QuBit decays into the state 0 or 1 with a probability of 50% after the

⁶ Superposition

measurement. Since we have applied this gate equally to all qubits, we have a stochastically balanced histogram in which the probability of the occurrence of all permutations is the same.

True Quantum Hardware – IBM not only provides different simulator types, a circuit composer, and a lab for interested users⁷ but also **real quantum hardware**. This is offered as part of IBM Quantum Services and is referred to as the "System". Some of these systems are reserved for so-called "Premium Plan Clients", but there are currently 7 systems that are accessible to the public via a standard registration with IBM Quantum. There are different processor architectures, revisions, and design variants. Some of these are also subsections of a larger chip. An overview of current architectures and their specialties is available in Figure 5. In addition to the systems, programs are also offered that are intended to speed up the execution of circuits.

QV and CLOPS – In addition to the number of qubits, other attributes that describe the performance, quality, or scalability of a system are also important. IBM primarily specifies QV and CLOPS among its available quantum processors, which are supposed to play a role in the execution of programs and circuits.

QV or "Quantum Volume". Is a value that reflects the performance of gate-based quantum computers, regardless of the underlying technology. CLOPS or "Circuit Layer Operations Per Second", is a value that indicates how many layers of a QV circuit a QPU (Quantum Processing Unit) can execute per unit of time.

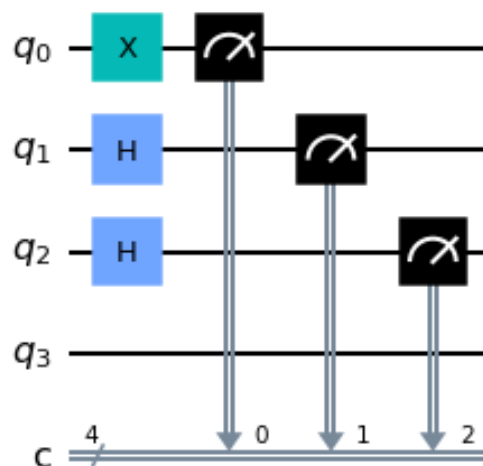


Figure 4: Example of another simple circuit

Running on real quantum hardware – As we've already seen, simple circuits with a small number of qubits can be simulated well on classical hardware. However, a simulator is an ideal quantum device that does not know true quantum noise due to decoherence⁸, faulty gates or measurements.

⁷ IBM's name for their Web IDE

⁸ Phenomenon of quantum physics, arising from unwanted interaction with environment

Qiskit

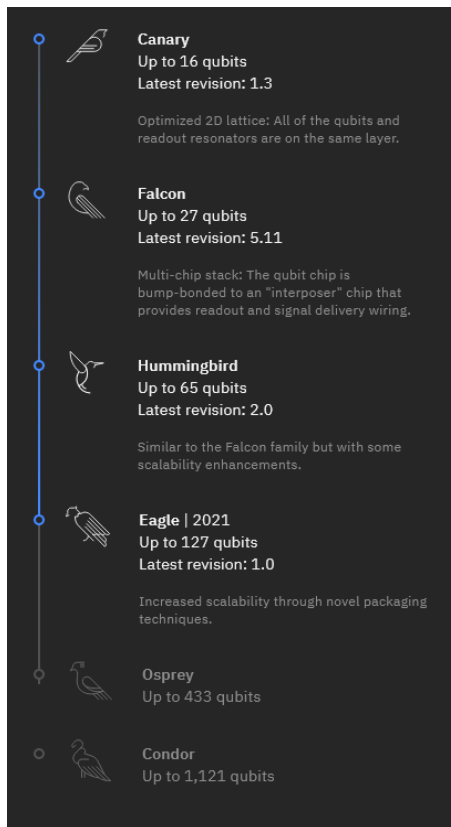


Figure 5: Different processor architectures

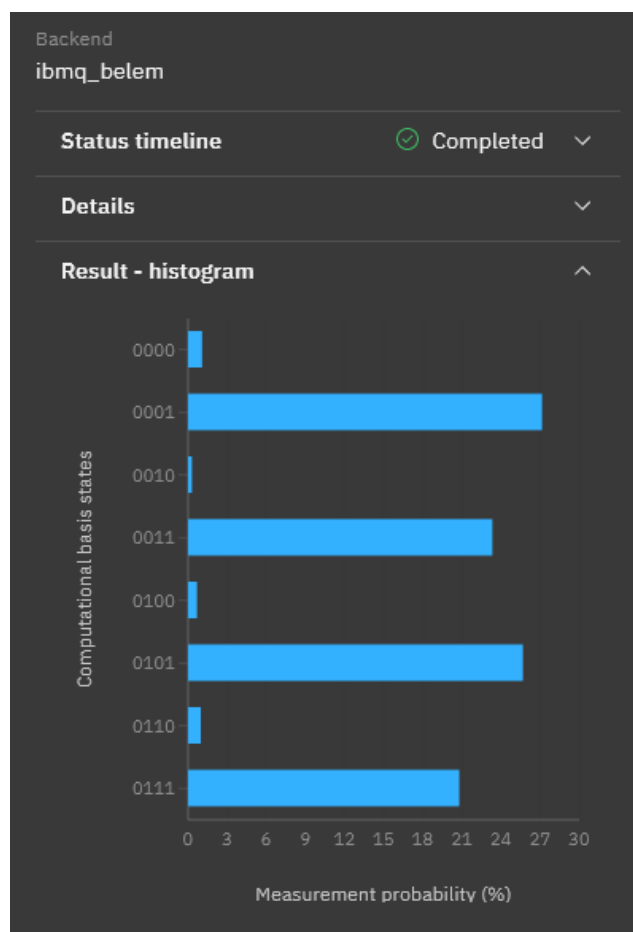


Figure 6: Completed job; Result

Qiskit

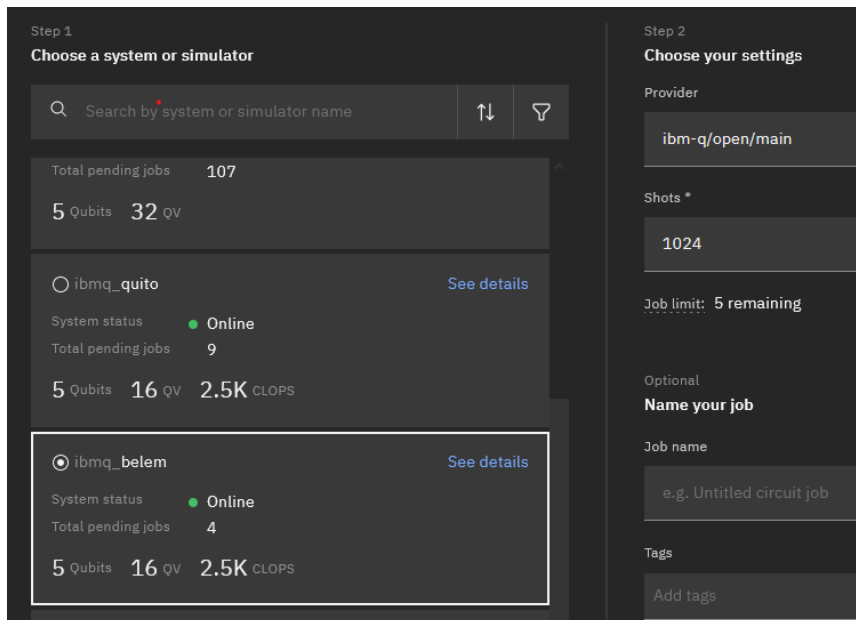


Figure 7: Selection of the system

Conclusion – Qiskit is a powerful quantum simulation software designed for pulses and circuits, with continuous development and a strong community. But network-related concepts can also be implemented on a physical level with Qiskit. Fault models are part of the software and can be validated through access to real hardware. Qiskit is very beginner-friendly among quantum simulators with a full-fledged ecosystem, not least because of the included "textbook", which explains the basics as well as provides useful examples.

Quantum Network Explorer

Repository	https://github.com/QuTech-Delft/qne-adk
Language	Drag & Drop
OS	Web browser
Type	Web Application
Focus	Network & Distributed Applications
Feature	Application Development Kit for Quantum Network Explorer (QNE-ADK)
License	MIT License (QNE-ADK)
Website	https://quantum-network.com
Registration	No



The Quantum Network Explorer, as the name suggests, is a solution provided by QuTech. [Web Application](#) with a focus on a visually clear simulation of **distributed applications** and **Network Protocols**. No registration is necessary and the simulation can be used independently of the operating system via the web browser. An extremely interesting point, especially for network operators, is the easy configurability of the so-called "**Fidelity**"⁹ for junctions and routes, see also Figure 10.

Experiments are instances of an application that are started automatically as soon as you run them. If you still want to register, it allows you to save your "experiments" and continue at a later time.

Ready to Use – Inexperienced users or users without Python programming knowledge can already use the three applications "**Distributed CNOT, State Teleportation** and **QKD**" (Illustration 8) [try](#) without having to write a single line of code. These examples are provided with detailed descriptions of the protocols and pictures and are intended to enable a quick start. The also existing [Quickstart Guide](#) describes the first steps and gives the user three different tasks to introduce him to the application.

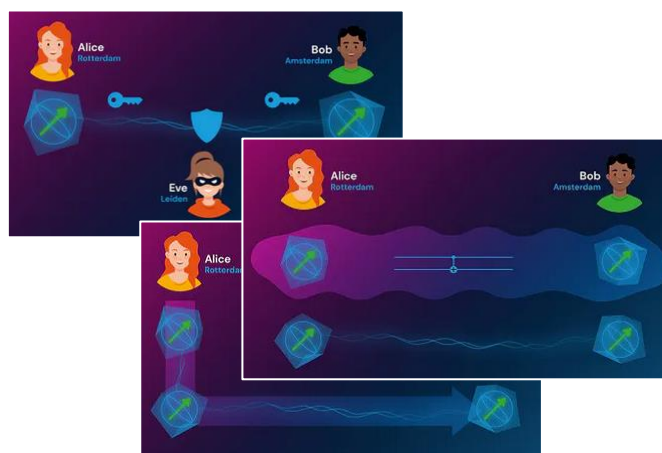


Illustration 8: Pre-built Applications in QNE (<https://www.quantum-network.com/>)

QNE-ADK is the [application development kit](#) for Quantum Network Explorer and is aimed at advanced users who want to write their own applications. QNE-ADK provides a CLI with commands to create the necessary files¹⁰ that define applications and experiments. When the experiment is configured, it can be run on the local¹¹ simulator.

⁹ **Fidelity** is a measure of the quality of quantum information processing

¹⁰ Guidelines must be followed for development.

¹¹ At the moment, QNE-ADK is only available locally

Quantum Network Explorer

Web-only application – Examples are already available, so there is no need to install or program your own circuits. The tool is initially purely web-based and has a clear visualization, which can significantly increase learning success. Therefore, this application is suitable for both the new and inexperienced user, as well as the budding expert who would like to install the associated Application Development Kit for Quantum Network Explorer (QNE-ADK). This includes everything to build your own quantum network application.

Application example CNOT – That [CNOT Gate](#) is an exclusive quantum gate that has two inputs and two outputs. The Control-QuBit x (Figure 9) indicates whether QuBit y is flipped. Like all quantum operations, the CNOT-Gate is reversible. This means that the operation can be completely reversed. This behavior (entanglement) makes the gate extremely interesting for distributed applications, as the two entangled particles can be distributed before the measurement and information is transmitted instantaneously through the "nothing" through the measurement.

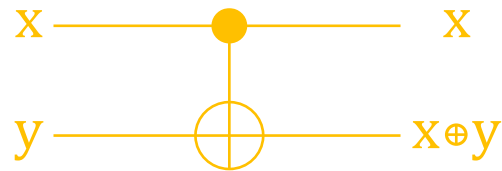


Figure 9: The CNOT gate

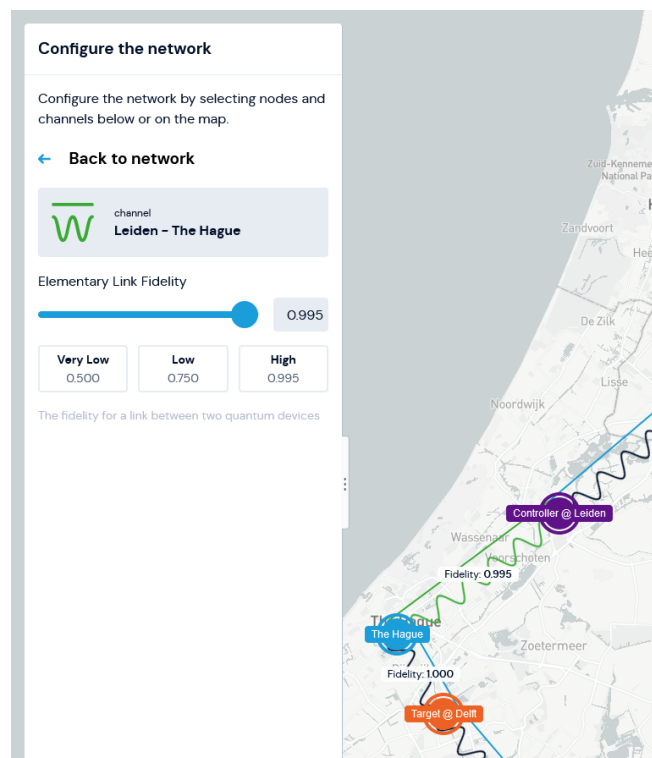


Figure 10: Configurable Fidelity for Nodes and Connections

In the following, we want to run through such an algorithm using an example in which we transfer a QuBit from a site A (controller) to a site B (target) and then observe how a measurement of another Qubit at site A determines the status of this qubit at site B without further communication.

Quantum Network Explorer

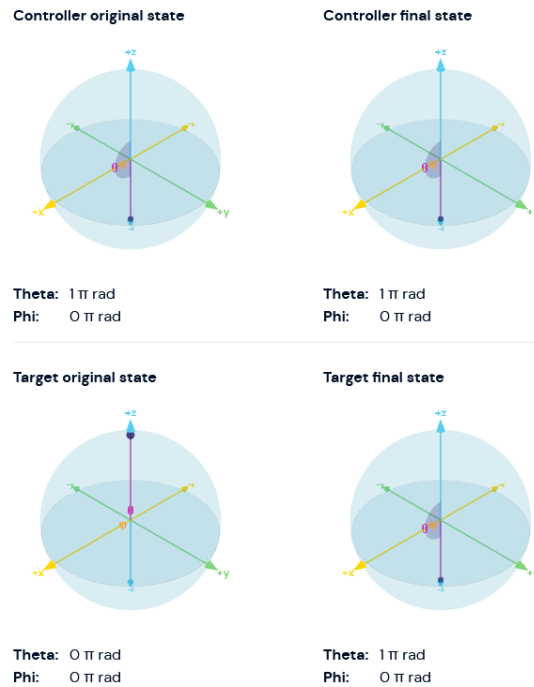


Figure 11: State of the Control and Target Qubits before and after transmission

Example Distributed CNOT – The first thing to do after starting the "Distributed CNOT" application is to select one of three possible networks. You can choose between "Randstad", "The Netherlands" and "Europe". As you might expect, these differ significantly in size and the distances to be covered. We choose Europe and now have the locations: Paris, Delft, Innsbruck, Copenhagen and Barcelona available to assign the roles "Controller" and "Target" to them, as already indicated in the CNOT section. We decide on Copenhagen as controller and Innsbruck as target, as a large part of the route runs through Germany.

Next, both endpoints are configured with a `gate_fidelity` of 0.995, for example, and the intermediate channel with a relatively low `link_fidelity` of 0.750. After that, we put the input state of the `controller` on $|1\rangle$, so that the `target` Qubit flips with the input state. $|0\rangle$

Visualization – Now, when we start the simulation, we see how an entangled Qubit pair is first created between the controller and the target. Interestingly, this is not done via the route with the lowest hops, but via an alternative route with higher fidelity. This is followed by preparations of the qubits on both sides, as well as applications of gates and measurements, followed by the transmission of results via a classic channel.

Conclusion – If you have not yet dealt with the topic of quantum algorithms relatively little, but still want to deal with distributed applications and networks without familiarizing yourself with the mathematics behind them, the Quantum Network Explorer could be a good choice. This stands out mainly with its successful visualizations and can be easily configured via the web without having to install anything.



Repository	https://github.com/SoftwareQuTech/SimulaQron
Language	Python
OS	Linux and macOS
Type	From application to network stack
Focus	Distributed application across multiple quantum processors and channels
Feature	Exploring the implementation of a network stack
License	Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: see link
Website	http://www.simulaqron.org/
Registration	-

Properties:

SimulaQron is a simulator designed for the development of platform-independent applications (mainly for the application layer) as well as for network protocols. However, it is not suitable for testing or simulating quantum repeaters, code, error correction and Qubit noise¹². The simulator can be run on one or more classical computers to simulate a network of distributed quantum processors. Libraries in Rust, Python and C are provided for the development of programs. To simulate the quantum processors (backend), which are connected to each other via a simulated communication network, SimulaQron uses the QuTip simulator. Using ProjectQ¹² is also possible. In principle, any simulator for a quantum processor that has a Python interface can be used as a backend. In addition, the simulator can be combined with NetSquid: An application written in SimulaQron can be run via the CQC (Classic Quantum Combiner) module in a Netsquid simulated network. This allows the influence of time, e.g. in the form of delays, to be taken into account in an event-driven manner.

Installation

SimulaQron can be easily installed using pip:

```
pip3 install simulaqron
```

To get the program up and running, the following entry must be changed after installation in file `simulaqron.py` in line 234 in the `simulaqron` folder (as of 04.07.2022):

```
@click.argument('value', type=click.Choice([b.value for b in
SimBackend.value]))
```

To

```
@click.argument('value', type=click.Choice([b.value for b in
SimBackend]))
```

¹² Axel Dahlberg and Stephanie Wehner, "SimulaQron – A simulator for developing quantum internet software", 2019 Quantum Sci. Technol. 4 015001

SimulaQron was tested with the OS Ubuntu 20.04.

Building SimulaQron Node – The structure of a network node is Illustration 12 depicted:

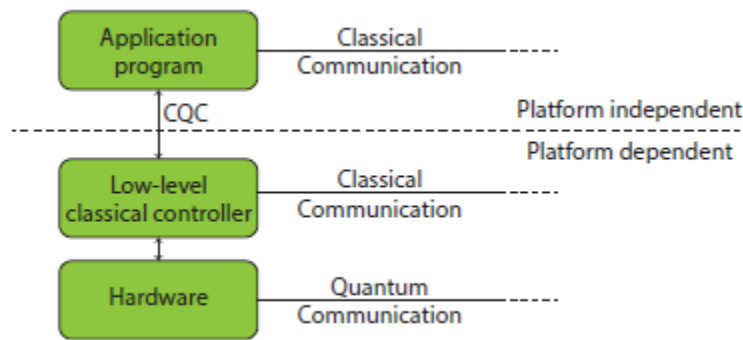


Illustration 12: Schematic Structure of a Network Node in SimulaQron¹²

The lowest level is the quantum hardware or processor, which can generate and measure qubits. It also has optical connections to other nodes. The processor has a platform-dependent controller that has, among other things, classic connections to other nodes.

The CQC (Classic Quantum Combiner) interface is provided by the platform-dependent part and contains commands e.g. for performing measurements and instructions specifically for quantum networks such as generating entanglement and transmitting QuBits.

Programming Modes – There are two different ways to access quantum hardware:

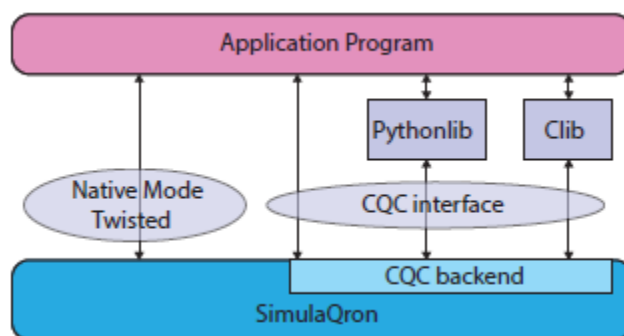


Illustration 13: Possibilities of programming SimulaQron¹²

1. *Native Mode*: In this mode, the hardware is accessed directly from the library `Twisted` in Python. This mode grants full access to the quantum hardware, but this mode is specifically designed for `Twisted` and will therefore hardly be used in future quantum networks¹².
2. *CQC Mode*: CQC is a packet format that is used to send instructions to the network node or quantum hardware. To simplify programming via CQC, a C and Python library is provided. In the meantime, a [RUST](#) library is also available.

CQC format – The CQC instruction set includes a number of different commands to control the quantum processor. The following comparison shows the command to create a QuBbit in Python and the corresponding CQC command:

Python Command	CQC Command
Creating QuBit on node1 <code>q=qubit(node1)</code>	Request to allocate a QuBit <code>CMD_NEW</code>

CQC can also be installed as a standalone Python module:

```
pip3 install cqclib
```

Example

Bell Pair Generation – A Bell Pair generation between two hosts (Alice, Bob) can be created as follows, using Python and CQC interface:

1. Start SimulaQron via the following command line command:

```
simulaqron start
```

This command starts five hosts, two of which are named Alice and Bob respectively
2. Then use the `cd` command to navigate to the following folder (folder with examples must be copied from the GitHub repository)

```
) : SimulaQron/examples/nativeMode/corrRNG/
```
3. After that, the following changes must be made in the `bobTest.py` and `aliceTest.py` files (as of 04.07.2022):
4. Replace `hostConfig` with `host_config`
5. Replace `socketsConfig` with `SocketsConfig`
6. Then start the program with `./run.sh` (file is in the examples folder).

Output of the program

```
App Alice: Measurement outcome is: 0/1
```

```
App Bob: Measurement outcome is: 0/1
```

As can be expected after measuring the Bell state, the value for Alice and Bob is either 0 or 1.

Source code `aliceTest.py`:

```
# Initialize the connection
with CQCConnection("Alice") as Alice:

    # Create an EPR pair
    q = Alice.createEPR("Bob")

    # Measure qubit
    m=q.measure()

    to_print="App {}: Measurement outcome is:
    {}".format(Alice.name,m)

    print("| " + "-"*(len(to_print)+2)+" |")
    print("| " +to_print+" |")
    print("| " + "-"*(len(to_print)+2)+" |")
```

The command `CQCConnection("Alice")` creates a host or node with the name Alice and at the same time establishes a CQC connection to it. The `Alice.createEPR("Bob")` command creates a pair of bells between the two hosts, Alice and Bob.

Result

SimulaQron enables platform-independent development of applications using CQC technology. Simulators for quantum processors (backend) are freely selectable as long as they have a Python interface. The simulator is designed for the development of applications for the application layer. For installation or Launch of the Bell pair example, it was necessary to modify the code (as of 04.07.2022). Presumably, the

SimulaQron

code or documentation has not yet been adapted accordingly. Without these adjustments, the evaluation of SimulaQron would not have been possible.

NetSquid

Repository	None available
Language	Python
OS	Platform-independent: Python interpreter required
Type	Performance study of the physical layer
Focus	Simulation of a quantum-based internet
Feature	Modular design; Quantum Computing Library
License	Free, but registration required
Website	https://netsquid.org/
Registration	Necessary, free of charge

Properties

NetSquid (NETwork Simulator for Quantum Information) is an event-oriented simulator suitable for simulating events in quantum networks and quantum computing systems from the physical to the application layer¹³.

Installation

After the required [registration](#), the NetSquid Python library can be downloaded by entering your username and password with the following command:

```
pip3 install --extra-index-url https://pypi.netsquid.org netsquid
```

Example

To illustrate the event-oriented nature of Netsquid, two hosts are supposed to send each other a QuBit, measure it and send it back again. For this purpose, Netsquid has a so-called. *discrete event simulation engine*, which arranges events, e.g. the receipt of a qubit, on a timeline and then processes them chronologically:

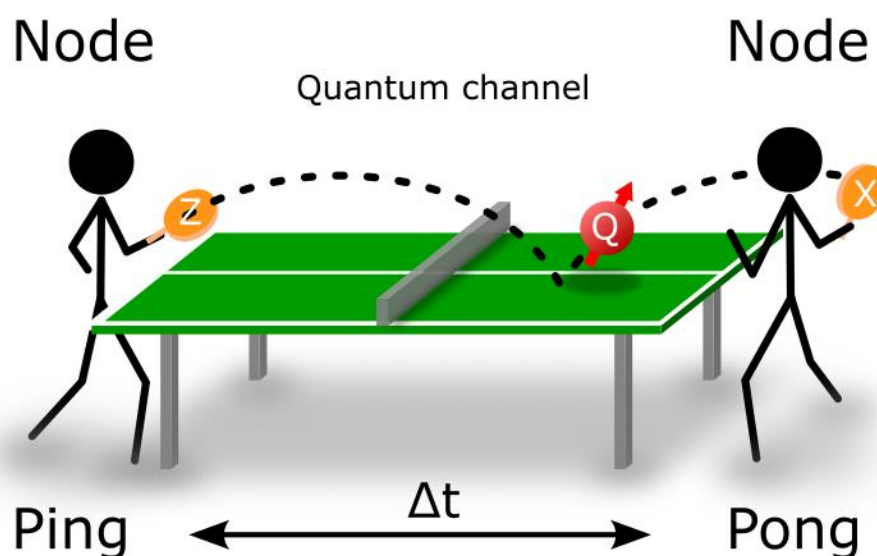
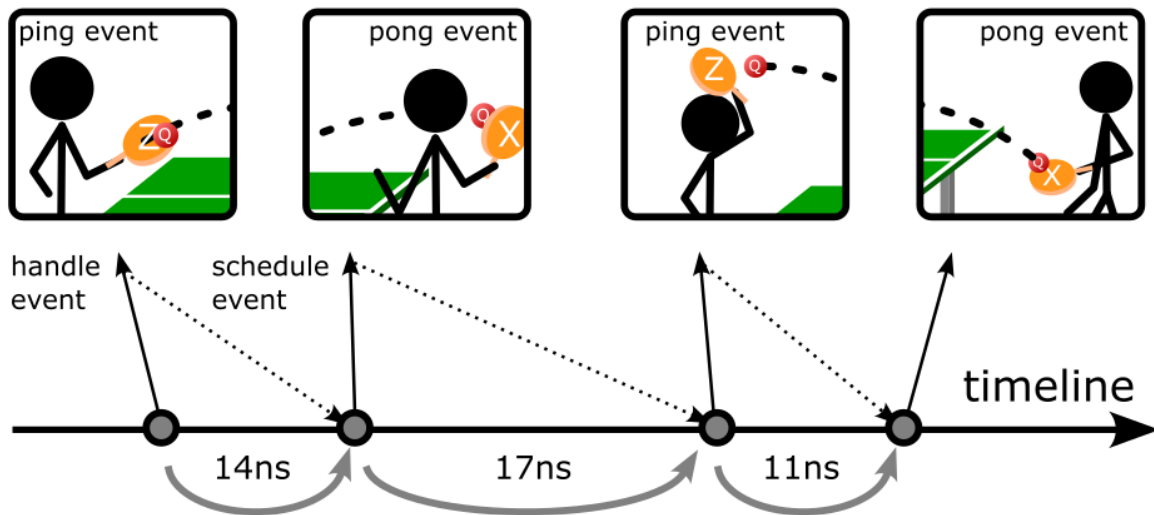


Illustration 14: Schematic representation of the program¹⁴

¹³ <https://www.nature.com/articles/s42005-021-00647-8>



Time progresses by stepping from event to event

Figure 15: Schematic representation and chronological sequence of an event-oriented simulation¹⁴

Network Components – To build the network, the following components are required:

- Two hosts or nodes, which serve as sender and receiver respectively
- Two one-way quantum channels for transmitting the QuBits

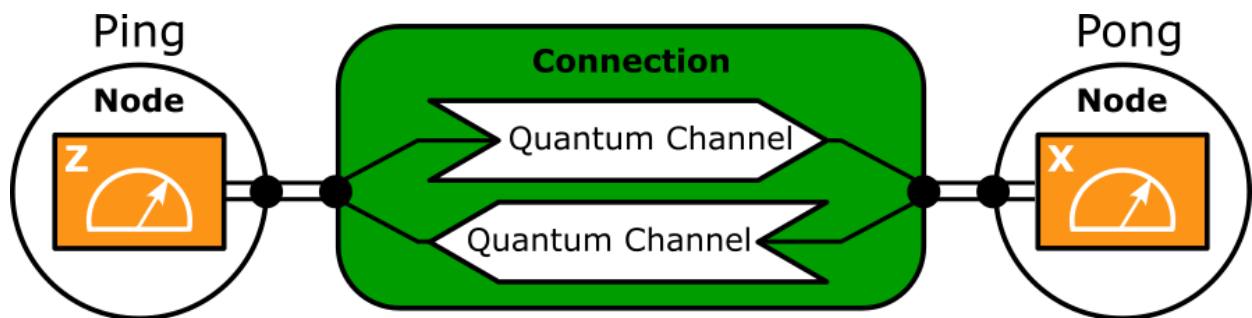


Illustration 16: Schematic representation of network topology¹⁴

The letters Z (Node Ping) and X (Node Pong) indicate the base in which the QuBits are measured. The following are the most important functions and start of the simulation with comments:

```
Create #Netzwerkknoten named Ping and Pong

node_ping = Node(name="Ping")
node_pong = Node(name="Pong")

# Create a class with a delay model for the quantum channels
class PingPongDelayModel(DelayModel)

# Connect both nodes with the quantum channels
connection = DirectConnection(name="conn[ping|pong]",
channel_AtoB=channel_1,channel_BtoA
=channel_2)

# Define protocol for sending, receiving, and measuring the QuBits
```

¹⁴ https://docs.netsquid.org/latest-release/quick_start.html

NetSquid

```
class PingPongProtocol(NodeProtocol)

# Assign log to the nodes and specify in which base to measure:

ping_protocol = PingPongProtocol(node_ping, observable=ns. Z,
qubit=qubits[0])

pong_protocol = PingPongProtocol(node_pong, observable=ns. X)

# Start logs in both nodes and set the runtime of the simulation in
ns:

ping_protocol.start()
pong_protocol.start()

run_stats = ns.sim_run(duration=300)
```

Output of the simulation:

```
17.4: Pong measured |+> with probability 0.50
33.8: Ping measured |1> with probability 0.50
51.3: Pong measured |-> with probability 0.50
69.7: Ping measured |0> with probability 0.50
87.8: Pong measured |-> with probability 0.50
```

The first entry indicates the time that has elapsed between two events. These times result from randomly generated delays (`PingPongDelayModel`) in the quantum channels. If a QuBit reaches the ping or pong node, the result of the measurement and its probability are displayed. Due to the different bases (Z-base \rightarrow ping, X-base \rightarrow pong) in which measurements are taken, the result of the measurement is random each time (probability 50%, measured states: $|0\rangle$ or $|1\rangle$: ping; $|+\rangle$ or $|-\rangle$ pong).

Result

NetSquid is suitable for simulating event-oriented processes and procedures in quantum networks. Compared to SeQUeNCE - which is also a discrete event simulator - NetSquid can simulate processes from the physical to the application layer. SeQUeNCE, on the other hand, is designed to simulate events in the lower two network layers, but is more detailed than NetSquid. Compared to SimulaQron and QuNetSim, on the other hand, NetSquid is more complex to create applications at the application layer.

QuNetSim



Repository	https://github.com/tqsd/QuNetSim
Language	Python
OS	Cross-platform
Type	framework for quantum networking simulations
Focus	Network
Feature	Optional own backend
License	MIT License
Website	https://tqsd.github.io/QuNetSim/
Registration	no

[QuNetSim](#) is a package written in Python that is freely available and is suitable for testing protocols quickly and easily. It is mainly aimed at students and teachers who are looking for a suitable demonstrator to learn and explain "high-level" protocols. The software simulates the network layer in a quantum network without the user having to worry about routing between two hosts that are (in)directly connected by the network topology. In addition, the simulator has mechanisms to control synchronization in the network. In QuNetSim, you can also run a classical and a quantum network in parallel in a simulation, as is necessary for some algorithms.

The different backends – The modular backend is worth mentioning, which works with [SimulaQron by default](#) , but can also be exchanged with other backends such as [ProjectQ](#) and [EQSN](#). If you prefer your own backend, you can integrate your own library.

The installation with pip – QuNetSim can¹⁵ be installed manually on Windows and Linux via source code, but it is advisable to perform the installation in a virtual environment in Python. A description of how to create a virtual environment can be found in the section on installing Qiskit. We do the installation with `pip install qunetsim`.

Templates are scripts that map or instantiate a network. To test these networks, all you have to do is run the corresponding template. If you want to create new templates, this is achieved via `template` . This takes the user through a wizzard, which asks what the new template should be called, how many nodes should be created, which backend should be used, and what topology the hosts should form. Among them: mesh, ring, star, linear and tree.



Hello World – After the installation is complete, we run the template script . We now find a new `.py` script in the current path, which can be executed. This already includes a kind of "Hello World" program, which in our case sends 5 qubits in the state $|1\rangle$ from host A to host B and is measured there.



The Qubit object in QuNetSim is mainly a wrapper for the Qubit located in the backend. The class is located in `qunetsim.objects.qubit` and is imported accordingly. With `q = Qubit(host, qubit=None, q_id=None, blocked=False)` a Qubit can be instantiated and assigned to a host. With the Qubit, some operations can now be performed. Among them, for example:



H()
X(),Y(),Z()
cnot(target)

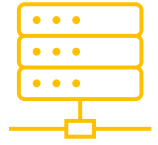
Hadamard
Pauli
CNOT

fidelity(other_qubit)
measure()
send_to(receiver_id)

Return of Fidelity
Measure
Send to Host

¹⁵ Cloning the repository and installing the requirements with pip

The *host* is analogous to a host or a node in a classic network. It can route packets, act as a relay, or follow special protocols. The class is located in `qunetsim.components.host` and a new host is created with e.g. `host_alice = Host('Alice')`. Hosts are also connected to other hosts through Connections. To do this, `host_alice.add_connection('Bob')` and `host_bob.add_connection('Alice')` are executed, which establishes two bi-directional connections, a classical connection and a quantum channel.



The *network* is a central component in any simulation. The networks must be linked to hosts that have already defined their connections. Based on the topology, different routing algorithms can now be set up for the classical and quantum channels. The shortest route is used as default.

Example – The code excerpt below is one example among many that can be found [here](#) on the documentation website. Since this is an example that mainly covers the basics, we will discuss the code excerpt in its entirety here. Some things have already been described in the previous sections.

The following example (Figure 17) is a network in which each participant is part of a linear network. Alice is connected to Bob, Bob to Eve, etc.; now Alice wants to transfer 10 QuBits to Dean and waits for a confirmation from Dean after each transmission to make sure that the QuBit has also arrived at Dean.

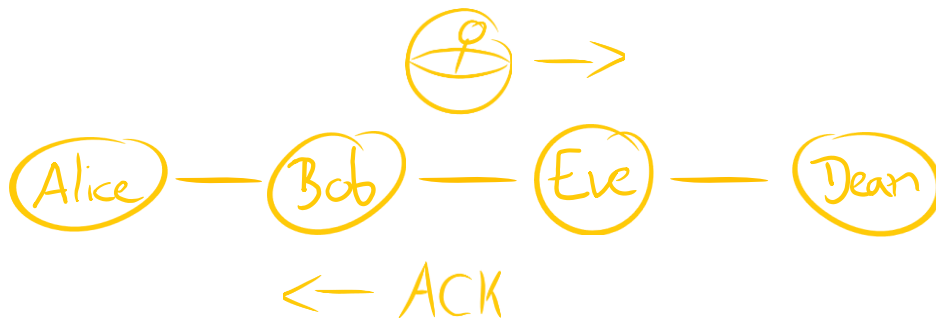


Figure 17: Example network QuNetSim

We start in lines 1–3 with the import of the required packages `Host`, `Network` and `Qubit`. In the `main()` function, a network is first instantiated (line 5), then a string array with the four participants is created (line 6) and finally started with the `network.start(nodes)` method

and with a `delay` of 0.1 (line 8). In the following (lines 9–22), the hosts are created, linked to each other and started. To complete the network configuration, the hosts are added to the network using the `network.add_host()` method.

```

1 from qunetsim.components import Host
2 from qunetsim.components import Network
3 from qunetsim.objects import Qubit
4 def main():
5     network = Network.get_instance()
6     nodes = ["Alice", "Bob", "Eve", "Dean"]
7     network.start(nodes)
8     network.delay = 0.1

```

QuNetSim

```
9     host_alice = Host('Alice')
10    host_alice.add_connection('Bob')
11    host_alice.start()

12    host_bob = Host('Bob')
13    host_bob.add_connection('Alice')
14    host_bob.add_connection('Eve')
15    host_bob.start()

16    host_eve = Host('Eve')
17    host_eve.add_connection('Bob')
18    host_eve.add_connection('Dean')
19    host_eve.start()

20    host_dean = Host('Dean')
21    host_dean.add_connection('Eve')
22    host_dean.start()

23    network.add_host(host_alice)
24    network.add_host(host_bob)
25    network.add_host(host_eve)
26    network.add_host(host_dean)

27    for _ in range(10): # Create a qubit owned by Alice
28        q = Qubit(host_alice)
29        # Put the qubit in the excited state
30        q.X()
31        # Send the qubit and await an ACK from Dean
32        q_id, _ = host_alice.send_qubit('Dean', q, await_ack=True)
33        # Get the qubit on Dean's side from Alice
34        q_rec = host_dean.get_data_qubit('Alice', q_id)
35        # Ensure the qubit arrived and then measure and print the
36 results.
37        if q_rec is not None:
38            m = q_rec.measure()
39            print("Results of the measurements for q_id are ", str(m))
40        else:
41            print('q_rec is none')

42    # Stop the network at the end of the example
43    network.stop(stop_hosts=True)

44 if __name__ == '__main__':
45     main()
```

In line 28, Host Alice will create a qubit and convert it to the excited state $|1\rangle$ with the X gate (line 30). Now everything is already done to transfer the QuBit to Dean with `host_alice.send_qubit('Dean', q, await_ack=True)` from line 32. Alice will then wait for confirmation from Dean before continuing. Since the flag `await_ack` is set

to true, `send_qubit()` returns two values: the qubit ID that was sent, and a boolean value that indicates whether the ACK has arrived or Alice has exceeded the maximum wait time.

Dean reads the QuBit in line 34 and then takes a measurement (line 37) if the transfer was successful and outputs a string. This procedure is

executed a total of 10 times (for 10 qubits) before the network is stopped.

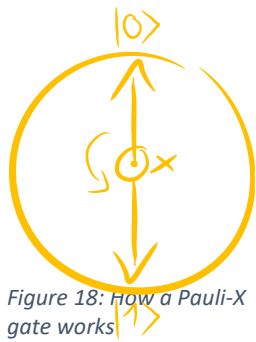


Figure 18: How a Pauli-X gate works

Explanation: In addition to the Pauli-Y and Pauli-Z gates, the Pauli-X gate is a 1-QuBit gate that inverts the input QuBit, i.e. from a state $|0\rangle$ creates a state $|1\rangle$ and vice versa. If we look at this behavior on the so-called Bloch sphere, which is the

standard for representing 1-qubit states, we will find that this change of state corresponds to a 180° rotation of the vector around the x-axis, see Figure 18. This behavior corresponds to the behavior of the classic NOT gate. The other two gates work in the same way, only around the corresponding axis.

Conclusion – QuNetSim is a simulator based on Python and thus written in a common programming language suitable for beginners. It is very suitable for students and teachers, as well as users who need a demonstrator or are looking for a suitable introduction to the topic of quantum technology. In addition, this framework is aimed at interested parties who are already looking for an entry into networks and expect that a network system has already been implemented. The documentation provided is detailed and easy to understand and contains many examples.

SILQ

Repository	https://github.com/eth-sri/silq
Language	Q#, D, Tex, Python
OS	VS Code is required to install the Silq plug-in
Type	Quantum circuits
Focus	More intuitive semantics
Feature	Uncomputation/Reset of QuBits to the Initial State
License	FreeBSD License
Website	https://silq.ethz.ch/
Registration	No restrictions: download and install without registration

Properties

Silq is a simulator that takes into account the so-called uncomputing of QuBits in simulations¹⁶: This enables the automatic reset of temporarily required QuBits - so-called Ancilla QuBits - which are then available for further operations. The simulator uses a syntax that is specially designed for programming with QuBits and has its own variable types for "classical" states and those states that are used to store quantum states (superposition). Silq is based on the perspective of application developers and not on creating algorithms for a specific type of quantum processor. Matrix operations and tensor products are not required for this: Instead, Silq uses suitable variable types and permissible operators, which significantly shortens the scope of code for creating algorithms (e.g. groover algorithm). Other simulators such as Cirq, Qiskit etc... are mostly based on Python or Matlab and are designed more for use on "classic computers". Silq, on the other hand, is specifically designed for the abstraction of low-level qubit operations.

Installation

The easiest way to install Silq is to add Silq as a plugin in VS Code (Visual Studio Code). For the following use case, the Silq plugin has been installed in VS Code on Windows 10.

Example

As a simple example, a Bell pair – including measurement – is created in Silq:

```
1 def main(){2 x0:=0:B; 3 x0:=H(x0); 4 x1:= if x0 then 1:B else
0:B;
5 return measure (x0,x1);}
```

The function returns output with a probability of 50% either (0,0) or (1,1). The formula for the Bell pair is:

$$\Psi = \frac{1}{\sqrt{2}} (|00\rangle + |11\rangle)$$

In contrast, languages such as Qiskit are based on the concept of quantum gates or status vectors for building a circuit for quantum entanglement:

```
qc = QuantumCircuit(2)
# Apply H-gate to the first:
```

¹⁶ Silq: A High-Level Quantum Language with Safe Uncomputation and Intuitive Semantics: <https://files.sri.inf.ethz.ch/website/papers/pldi20-silq.pdf>

SILQ

```
qc.h(0)
# Apply a CNOT:
qc.cx(0,1)
qobj = assemble(qc) result =
svsim.run(qobj).result().plot_histogram(result.get_counts())
```

As can be seen from the comparison of the two source codes, Qiskit requires an H and CNOT gate, while Silq only needs an H gate and performs CNOT with an if-else statement (lines of code marked CNOT in blue).

Uncomputation – Silq distinguishes between consumed and unconsumed variables, thus implementing the concept of uncomputation. To explain this concept, three variables are connected by an AND link:

$$x\&\&y\&\&z$$

On a classical computer, the results of such an operation are stored in temporary variables, e.g. the result of $x\&\&y$.

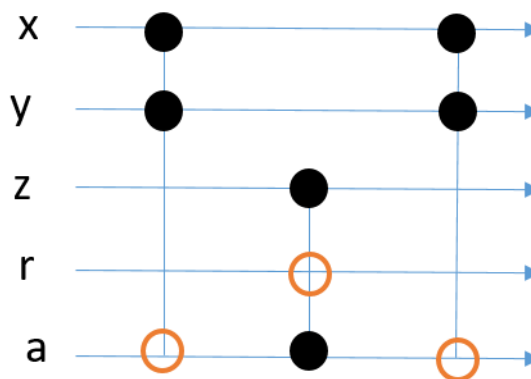


Figure 19: QuBit gate uncomputation

First, the result of the operation $x\&\&y$ is stored in the variable a . After that, the result of $a\&\&z$ is stored in r . r and a are in the initial state 0. If the fourth QuBit is to be used for further operations, it must return to state 0. This is done by so-called toffoli gates (orange circles in Figure 19). Such gates invert a QuBit if both inputs have state 1. This sets the state of QuBit a back to 0 by the third gate, because both inputs ($x\&\&y$) are true (=1). In the case of quantum computing, the uncomputation of variables is very important, as there are only limited qubits available.

Consumable and non-consumable parameters – Silq implements uncomputing in the form of consumable and non-consumable parameters. Consumed parameters are those that are utilized by a function, e.g. through a measurement. The quantum mechanical state after a function thus no longer depends on this variable, but the result of the function does.

```

bell-silq.slq
> Users > si
1 def
2
3 x0:=H(x0);
4 x1:= if x0 then 1:B else 0:B;
5 //return measure (x0,x1);
6
7 }

```

Figure 20: Error with unconsumed variable

Figure 20 shows the error output if a variable is not consumed by the function. In the example, the `return` statement with the measurement of the variable (cf. Code for the creation of the Bell state in Silq).

To define parameters that are not consumed by a function, the following annotations are available in Silq: `const`, `lifted` and `qfree`. Example: Function parameters or expressions annotated in this way do not change or destroy superpositions, i.e. `qfree` variables are automatically uncomputed after use and are available for further use:

```

def qfree_example(f : B → qfree B) qfree {
  return f(true); qfree Result
}

```

Non-consumed variables cannot store superpositions, only stable states such as $|0\rangle$ or $|1\rangle$.

Result

With its unique syntax, Silq makes it possible to simulate processes such as entanglement and teleportation largely without status vectors and matrices. The simulator is platform-independent, i.e. not designed for a specific quantum processor or computer. Quantum algorithms such as the Groover algorithm can be implemented in Silq with significantly fewer lines of code than, for example, in Qiskit (cf.^{17,18}).

The syntax used makes it possible to perform many operations with only a few lines of code, but this increases complexity. Simulations based on gates and matrices, on the other hand, are easier to understand than Silq simulations.

¹⁷ Groover algorithm in Silq: <https://silq.ethz.ch/overview>

¹⁸ Groover algorithm in Qiskit: <https://qiskit.org/textbook/ch-algorithms/grover.html>

Google Quantum AI (Cirq)

Repository	https://github.com/quantumlib/cirq
Language	Python
OS	Platform-independent: requires Python
Type	Quantum circuits
Focus	Testing algorithms (on a quantum basis)
Feature	Ecosystem
License	Apache License 2.0
Website	https://quantumai.google/cirq
Registration	Unnecessary

Description

Cirq is a Python software library for writing, manipulating, and optimizing quantum circuits, which can then be run on quantum computers and quantum simulators. Cirq provides useful abstractions for dealing with today's noisy quantum computers.

Installation

For installation on Windows, Linux or Mac OS X, Python version $\geq 3.7.0$ is required, as well as the latest version of the package manager pip.

```
python -m pip install --upgrade pip
python -m pip install cirq
```

To test whether the installation was successful, the following command is suitable:

```
python -c 'import cirq_google; print(cirq_google.Sycamore)'
```

```
(cirq-env) sascha@DESKTOP-50U42TS:~$ python -c 'import cirq_google; print(cirq_google.Sycamore)'
```

```

      (0, 5)---(0, 6)
          |
          |
      (1, 4)---(1, 5)---(1, 6)---(1, 7)
          |   |   |   |
      (2, 3)---(2, 4)---(2, 5)---(2, 6)---(2, 7)---(2, 8)
          |   |   |   |   |
      (3, 2)---(3, 3)---(3, 4)---(3, 5)---(3, 6)---(3, 7)---(3, 8)---(3, 9)
          |   |   |   |   |   |
      (4, 1)---(4, 2)---(4, 3)---(4, 4)---(4, 5)---(4, 6)---(4, 7)---(4, 8)---(4, 9)
          |   |   |   |   |   |   |
      (5, 0)---(5, 1)---(5, 2)---(5, 3)---(5, 4)---(5, 5)---(5, 6)---(5, 7)---(5, 8)
          |   |   |   |   |   |
      (6, 1)---(6, 2)---(6, 3)---(6, 4)---(6, 5)---(6, 6)---(6, 7)
          |   |   |   |
      (7, 2)---(7, 3)---(7, 4)---(7, 5)---(7, 6)
          |   |   |
      (8, 3)---(8, 4)---(8, 5)
          |
      (9, 4)
(cirq-env) sascha@DESKTOP-50U42TS:~$
```

Figure 21: Sycamore processor with 54 qubits from 2019.

Registration

No registration is required to use the framework. However, if you shy away from a local installation, you can also use Google's service called Colab, but this requires a login to a Google account. This service is basically a Jupyter Notebook clone embedded in Google Drive. Colab is compatible with Jupyter and also allows you to open and export `.ipynb` files.

Example

```
try: import cirqexcept ImportError: print("installing cirq...")!pip install --quiet cirq import cirq print("installed cirq.")

# Pick a qubit.qubit = cirq. GridQubit(0, 0)

# Create a circuitcircuit = cirq. Circuit( cirq. X(qubit)**0.5, # Square root of NOT.      cirq.measure(qubit, key='m') # Measurement.)print("Circuit:")print(circuit)

# Simulate the circuit several times.simulator = cirq. Simulator()result = simulator.run(circuit, repetitions=20)print("Results:")print(result)
```

Circuit:

```
(0, 0): —X^0.5—M('m')—
```

Results:

```
m=01110011010101001001
```

Result

With Cirq and Quantum AI (as well as the other big players IBM and Amazon), Google offers not only a framework for local installation, but an entire platform for the development and execution of algorithms. Just like IBM, Google is also a leader in the development of its own hardware and also offers it to the user, or allows remote access to globally available quantum processors and simulators, including AQT, Azure, IonQ, Pasqal and Rigetti. Researchers with approved projects can run jobs on Google's comprehensive infrastructure.

SQUANCH

Repository	https://github.com/att-innovate/squanch
Language	Python
OS	Platform-independent: Python interpreter required
Type	Simulation of multiparty networks
Focus	Simulation of quantum networks
Feature	contains classical and quantum error models
License	WITH License
Website	https://att-innovate.github.io/squanch/index.html
Registration	No restriction

SQUANCH

Description

SQUANCH is also an open-source Python framework for creating parallelized and distributed simulations of quantum information. Although SQUANCH can be used as a universal simulation library for quantum computers, it was developed specifically for simulating quantum networks. It should be possible to test ideas for quantum transmission and network protocols. The package contains several modules, including extensible quantum and classical error models, as well as a multithreaded framework for the high-performance manipulation of quantum information.

Installation

As with all other simulators, it is recommended to install them in a virtual environment (Anaconda, VENV). Under certain circumstances, a Python "distribution" should also be chosen, which already contains some scientific packages such as matplotlib, in order to be able to use all functions easily.

```
pip install squanch
```

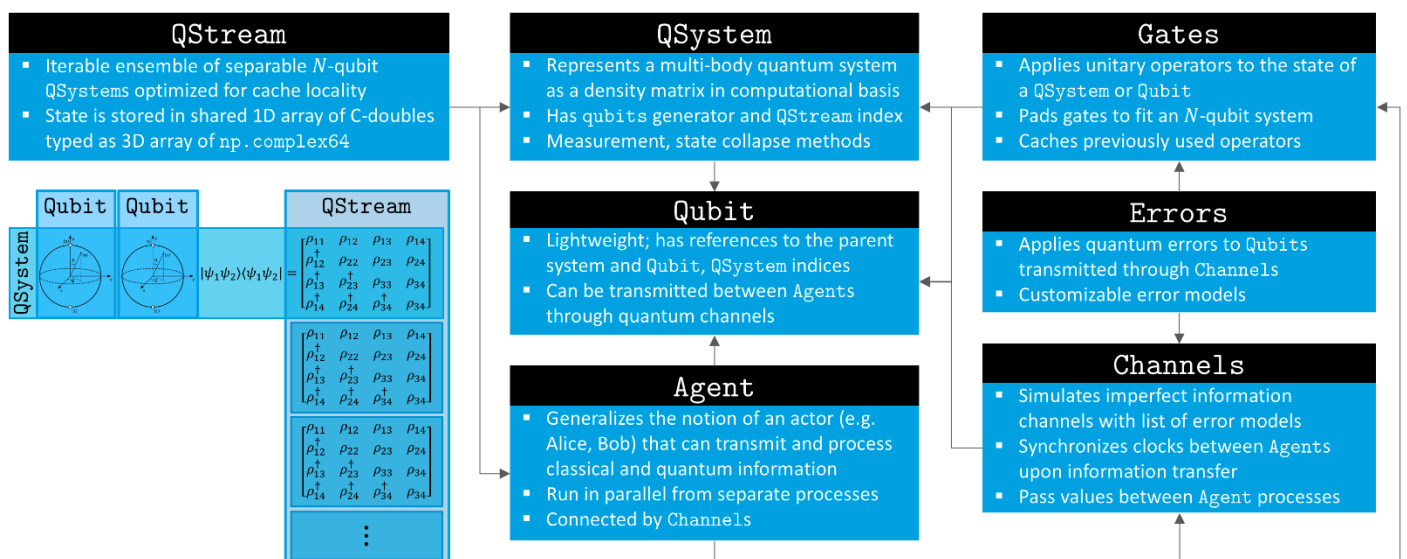
Structure and Modules

Figure 22: Source <https://att-innovate.github.io/squanch/overview.html#information-representation-and-processing>

Above is a schematic overview of the modules available in the SQUANCH framework. The QSystem is the most basic class and represents a multi-particle quantum state and is represented as a density matrix. Ensembles of quantum systems are efficiently handled by QStreams, and each QSystem has references to its qubits. Functions in the Gates module can be used to manipulate the state of a quantum system. Agents are generalized quantum mechanical "actors" that are initialized from a QStream instance and can

SQUANCH

change the state of the quantum systems in their stream object, typically by interacting directly with qubits. Agents run in parallel in separate processes and are connected by quantum and classical channels that apply customizable error models to the transmitted information and synchronize agents' clocks.¹⁹

Example

The example is intended to show how a QStream including QSystem, which contains the QuBits, works in the context of a communication between Alice and Bob. After creating a stream with two QuBits within a QSystem, one of these two QuBits is modified by the Hadamard gate. The two communication partners are children of the agent class and in this case are responsible for sending, receiving and measuring. Both partners share an output through which Bob communicates the result of his measurement. It is important here that the logic is within the agents by overriding the respective `run()` function.

```
from squanch import *

class Alice(Agent): def run(self): self.qsend(bob, a)

class Bob(Agent): def run(self): abob = (self.qrecv(alice)) abobm =
abob.measure() self.output(abobm)

stream = QStream(2,1) a, _ = stream.system(0).qubits

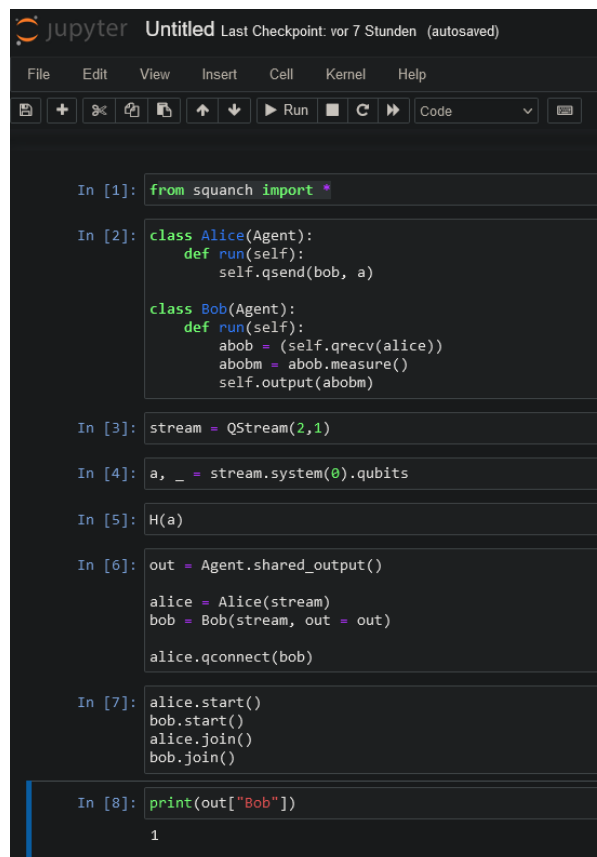
H(a)

out = Agent.shared_output()

alice = Alice(stream) bob = Bob(stream, out = out)

alice.qconnect(bob) alice.start() bob.start() alice.join() bob.join()

print(out["Bob"])
```



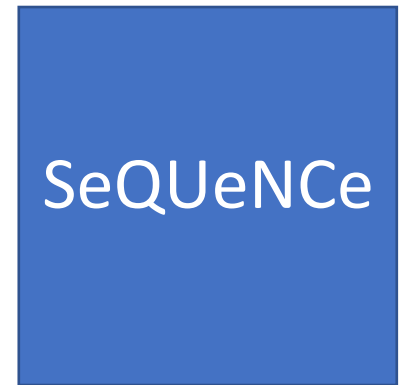
```
Jupyter Untitled Last Checkpoint: vor 7 Stunden (autosaved)
File Edit View Insert Cell Kernel Help
+ -> Run Code
In [1]: from squanch import *
In [2]: class Alice(Agent):
        def run(self):
            self.qsend(bob, a)
        class Bob(Agent):
        def run(self):
            abob = (self.qrecv(alice))
            abobm = abob.measure()
            self.output(abobm)
In [3]: stream = QStream(2,1)
In [4]: a, _ = stream.system(0).qubits
In [5]: H(a)
In [6]: out = Agent.shared_output()
        alice = Alice(stream)
        bob = Bob(stream, out = out)
        alice.qconnect(bob)
In [7]: alice.start()
        bob.start()
        alice.join()
        bob.join()
In [8]: print(out["Bob"])
        1
```

SQUANCH

Illustration 23: Jupyter Notebook IDE

Result

SQUANCH is a universal simulation library for quantum computers with a focus on mapping quantum networks. Due to the modular structure, users have various options for implementing algorithms and protocols. The developers themselves offer many technical backgrounds and examples of Quantum Teleportation, Superdense Coding, Man-In-The-Middle Attack and Quantum Error Correction in their repository.



Repository	https://github.com/sequence-toolbox/SeQUeNCe/
Language	C++, Python, Makefile
OS	Platform-independent: Python interpreter required
Type	protocols, network parameters, and topologies
Focus	Effects in quantum networks on the lower network layers
Feature	Intermediate storage of quantum states
License	Open Source License
Website	https://sequence-toolbox.github.io/index.html
Registration	No restrictions

Properties

[SeQUeNCe](#) is an event-oriented, Python-written, freely available simulator for the areas of (quantum) hardware, management for entanglement, resources, networks and applications²⁰. The simulator is particularly suitable for simulating events in the lower network layers (hardware, connection layer): three hardware components are required (quantum memory, quantum channel and detector) just to generate a quantum superposition state.

Installation

Python 3.7 or higher is required to install the simulator. SeQUeNCe can then be installed from the GitHub repository using the following commands:

```
git clone https://github.com/sequence-toolbox/SeQUeNCe.git
cd Sequence-python
pip install .
```

Example

As an application, the generation and measurement of a Bell pair in SeQUeNCe is shown:

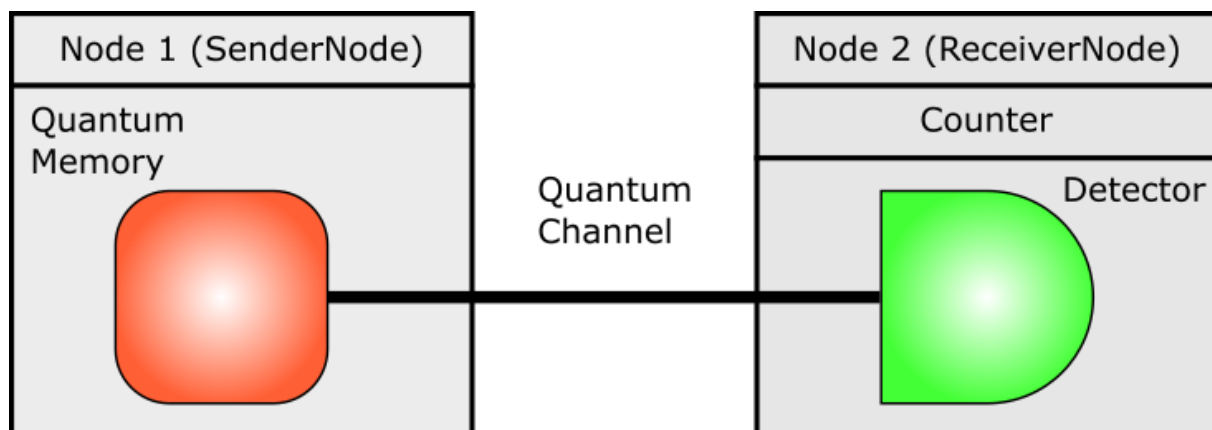


Figure 24: Hardware structure for creating entanglement²¹

The quantum memory in Figure 24 consists of an atom, which in this example is in the superposition state:

$$\Psi = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

The detector is used to measure the condition. The probability of measuring the state $|0\rangle$ or $|1\rangle$ is 50%.

²⁰ Sequence Paper: <https://doi.org/10.48550/arXiv.2009.12000>

²¹ <https://sequence-toolbox.github.io/tutorial/chapter2/hardware.html>

SeQUeNCe

First, the two nodes (quantum memory, detector) must be created:

```
# class for Quantum memory
class SenderNode(Node):
    self.memory = Memory('node1.memory', t1, fidelity=0, frequency=0,
        efficiency=1, coherence_time=0, wavelength=500)

# class Quantum receiver
class ReceiverNode(Node):
    self.detector = Detector('node2.detector', t1, efficiency=1)
```

In the case of quantum memory, it is necessary to specify further parameters, such as the coherence time and the quality of entanglement.

The protocol determines the behavior of the counter when a photon arrives. When detecting a photon, the counter's counter is increased by 1:

```
class Counter():
    def __init__(self):
        self.count = 0

    def trigger(self, detector, info):
        self.count += 1
```

The simulation time in picoseconds and the two nodes are defined below:

```
t1 = Timeline(10e12)
node1 = SenderNode("node1", t1)
node2 = ReceiverNode("node2", t1)
```

To connect the quantum memory and detector, the definition of a quantum channel – attenuation=0, length=1km- is required:

```
qc = QuantumChannel("qc", t1, attenuation=0, distance=1e3)
```

The state of the quantum memory is now set to the superposition state:

```
node1.memory.update_state([complex(0), complex(1)])
```

The quantum memory now needs to be excited and this event passed to the event handler:

```
process = Process(node1.memory, "excite", ['node2'])
event = Event(0, process)
t1.schedule(event)
```

After passing to the event handler, both nodes can be started:

```
t1.init()
t1.run()
```

The output of the program is the number of detected photons and the simulation duration:

```
detection count: 1
detection time (ps): 4999950
```

Result

Like Netsquid, SeQUeNCe is an event-oriented simulator, but it is more focused on simulating physical events. Programs in this simulator are therefore more complex, but depict processes and events more realistically. Users can use a variety of parameters such as quality of entanglement, coherence time, etc. to customize the simulations. Compared to other simulators, SeQUeNCe is more complex and requires more training time.

QKDNetSim

Repository	https://github.com/QKDNetSim/qkdnetstim-dev
Language	C/C++, Python, Perl
OS	Linux
Type	QKD Simulation Module
Focus	QKD base in overlay or TCP/IP mode
Feature	Based on the NS-3 Network Simulator
License	GPL 2.0 License
Website	https://www.qkdnetstim.info/documentation/
Registration	No restrictions



Properties

[QKDNetSim](#)²² is a module for the NS-3 network simulator written in C++ and therefore not a standalone simulator. With this expansion module, it is possible in NS-3 to simulate networks with QKD (Quantum Key Distribution) in two operating modes: `overlay` or `single TCP/IP mode`.

Installation

Note – The latest Ubuntu version running QKDNetSim is Ubuntu 18.04. This version was also used for testing the simulator. To install it, the following commands must be executed in the terminal:

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install gcc g++ python python-dev mercurial bzip2 gdb
valgrind gsl-bin doxygen graphviz imagemagick texlive texlive-
latex-extra texlive-generic-extra texlive-generic-recommended
texinfo dia texlive texlive-latex-extra texlive-extra-utils
texlive-generic-recommended texi2html python-pygraphviz python-kiwi
libboost-all-dev git flex bison tcpdump sqlite sqlite3 libsqlite3-
dev libxml2 libxml2-dev libgtk2.0-0 libgtk2.0-dev uncrustify
libgsl23 python-pygccxml libcrypto++-dev libcrypto++-doc
libcrypto++-utils -y
```

Then the QKDNetSim code must be downloaded and compiled:

```
cdgit clone https://github.com/QKDNetSim/qkdnetstim-dev.git
cd qkdnetstim-dev./waf configure./waf
```

Example

As an example, a QKD channel for transmitting a key between two hosts will be shown. The example is already included in the QKDNetSim examples and is located in the Git repository folder `qkdnetstim-dev/scratch`. From the `qkdnetstim-dev` folder, the program can be started with:

```
./waf --run scratch/qkd_channel_test
```

Output of the program:

```
Source IP address: 10.1.1.1Destination IP address: 10.1.2.2Sent
(bytes): 640 Received (bytes): 640Sent (Packets): 1 Received
(Packets): 1Ratio (bytes): 1 Ratio (packets): 1
```

²² Paper QKDNetSim: <https://link.springer.com/article/10.1007/s11128-017-1702-z>

QKDNetSim

The output shows the IP addresses of the hosts as well as the number and size of the transmitted packets. In addition to the output on the command line, the program also generates diagrams that show data such as the transfer rate or the buffer sizes of the keys on the individual hosts:

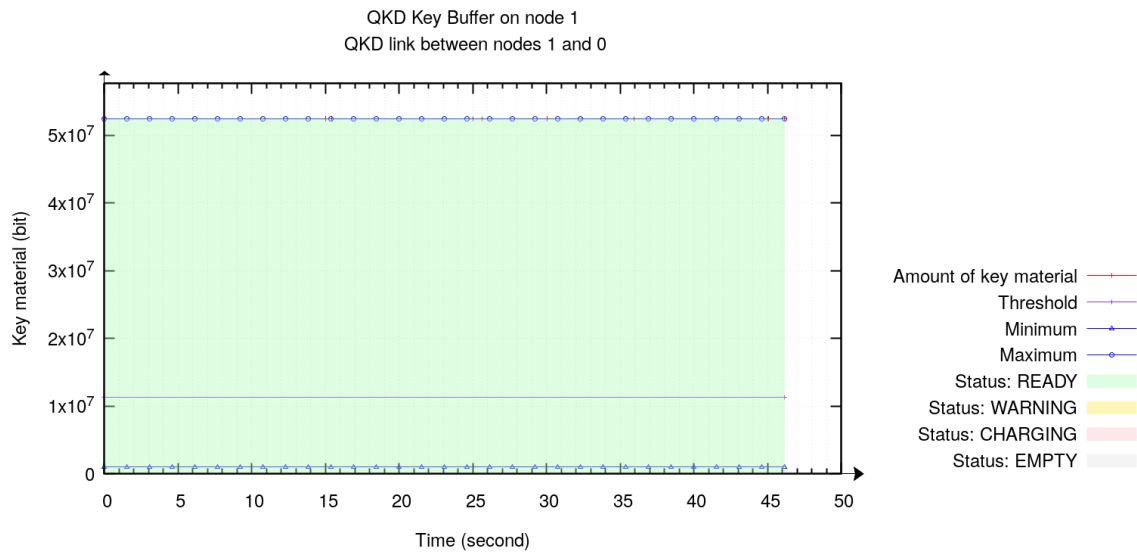


Figure 25: QKD buffer on host 1

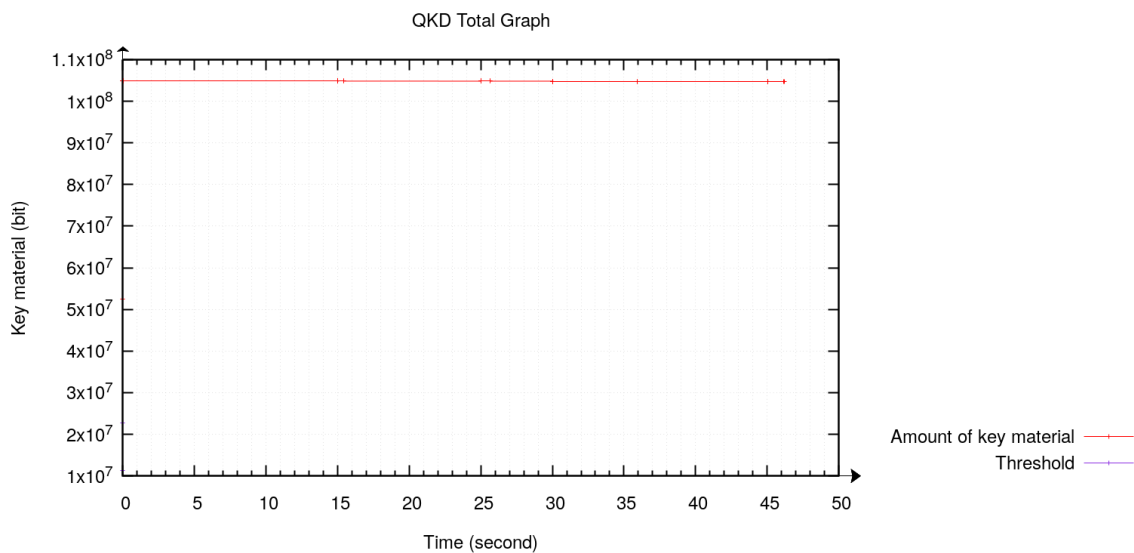


Figure 26: Key size in bits as a function of time

Result

The simulator is especially suitable for users who already have experience with the NS network simulator and have good C++ language skills. For simulations, `gnuplot` can be used to create graphics in `*.png` format from the `*.plt` files generated by the program, which can be individually adapted if necessary. So far (as of 14.07.2022) QKDNetSim can only be installed on Ubuntu distributions.

QKDSimulator

Repository	Not public (contact developer)
Language	Python
OS	Unknown/Web Browser
Type	Pure QKD Simulator
Focus	Full QKD stack or single simulation
Feature	Very detailed results
License	Not public (contact developer)
Website	https://www.qkdsimulator.com/
Registration	Unnecessary

Description

QKD-Simulator is a web application for the simulation and analysis of quantum key distribution protocols. The simulator relies on a QKD simulation toolkit that allows a wide range of parameters to be adjusted for individual components and sub-protocols in the system, e.g. quantum channel, sifting, error estimation, matching/error correction, data protection reinforcement. Each simulation provides detailed information about the intermediate and final stages of the protocol.²³

Simulation and Analysis of QKD currently supporting the BB84 variant.

Home
Simulation output example
Plots
Documentation
About

QKD simulator © is a web application aimed at simulating and analyzing quantum key distribution protocols. The simulator is powered by a QKD simulation toolkit that makes it possible to customize a wide range of parameters for individual components and sub-protocols in the system, e.g., quantum channel, sifting, error estimation, reconciliation/error correction, privacy amplification. Each simulation provides detailed information about the intermediate and final stages of the protocol.

Set the initial simulation settings and press the [Run Simulator](#) button.

Simulator type:

Parameter	Value
Initial Qubits (n)	<input type="range" value="500"/> Current value: 500
Basis choice bias delta	<input type="range" value="0.5"/> Current value: 0.5
Eve's basis choice bias	<input type="range" value="0.5"/> Current value: 0.5
Biased error estimation enabled	<input type="checkbox"/>
Error estimation sampling rate	<input type="range" value="0.20"/> Current value: 0.20
Error tolerance	<input type="range" value="0.11"/> Current value: 0.11
Channel noise enabled	<input type="checkbox"/>
Eve enabled	<input type="checkbox"/>
Eavesdropping rate	<input type="range" value="0.1"/> Current value: 0.1

[Run Simulator](#)
[?](#)

Figure 27: Web Interface of the QKDS Ivulator

²³ <https://www.qkdsimulator.com/>

Installation

Since it is a web application, no installation is necessary. However, a simulation engine is active in the background, which runs the QKDSimulator and is part of a toolkit for quantum key distribution.

Implementation

The implementation of the backend QKD toolkit that powers this website is designed to be reusable by relying on a component-based and fully modular approach, allowing each sub-protocol to be updated or replaced in an isolated manner. The current version offers an implementation of the entire QKD stack, i.e. quantum channel/transmission, key sifting, authentication with universal hashing, error estimation, matching/error correction and data protection reinforcement. The quantum channel currently only supports the [BB84 protocol](#). The QKD simulation toolkit is fully implemented in Python and uses standard scientific libraries such as Scipy, Numpy, Matplotlib, Quantum Information Toolkit (QIT) and PyCrypto²⁴ [information from developers].

²⁴ <https://www.qkdsimulator.com/about>

QKD (BB84) simulation run example:

Initial Configuration								
Property	Qubit Count	Basis choice bias delta	Eve basis choice bias delta	Eavesdropping	Eavesdropping rate	Error estimation sampling rate	Biased error estimation	Error tolerance
	500	0.5	0.5	1	0.1	0.2	0	0.11

Detailed Run Information

Phase 1: BB84 Quantum Transmission

Alice prepares a sequence of 500 qubits and send them to Bob over the quantum channel. She randomly chooses a basis for each qubit, rectilinear polarization (horizontal/0 degrees and vertical/90 degrees) or a diagonal polarization (+45 degrees and -45 degrees shifted). She then maps horizontal and vertical with the qubit states $|0\rangle$ and $|1\rangle$, and +45 degrees and -45 degrees shifted with the states $|+\rangle$ and $|-\rangle$, respectively. Further details:

- Alice sent 500 qubits to Bob with a basis selection bias of 0.5.
- Eve is eavesdropping on the quantum channel at a rate of 0.1 and with a basis selection bias of 0.5. There is an eavesdropper, Eve, listening in on the channel. She intercepts the qubits, randomly measures them in one of the two mentioned bases and thus destroys the originals, and then sends a new batch of qubits corresponding to her measurements and basis choices to Bob. Since Eve can choose the right basis only 50% of the time on average, about 1/4 of her bits differ from those of Alice.

Phase 2.1: Sifting

Bob announces on a public classical channel the qubits that he has managed to successfully measure. Alice and Bob then reveal and exchange the bases they used. They authenticate these three message exchanges. Whenever the bases happen to match - about 50% of the time on average - they both add their corresponding bit to their personal key. In the absence of channel noise, the two keys should be identical unless there has been an eavesdropper. Further details:

- The sifting phase started with 500 transmitted qubits and the resulting bit string was reduced to 257 bits.
- 0.514 of Alice's and Bob's chosen measurement bases match. 0.486 of their chosen bases do not match.
- 0.716 of the two parties measured qubits match before sifting and 0.284 of them do not.
- 0.9144 of the two parties measured qubits match after sifting and 0.0856 of them do not.

Phase 2.2: Sifting Authentication - Linear Feedback Shift Register (LFSR) Universal Hashing

Alice and Bob authenticate their basis exchange messages using the LFSR universal hashing scheme and a mutually preshared secret key for authentication. 3 messages are authenticated in the sifting phase. Further details:

- Bob informs Alice of the qubits he managed to successfully measure and he appends an authentication tag to his message. Authentication cost in terms of key material: 64
- Bob informs Alice of the bases he has chosen for measuring the qubits and he appends an authentication tag to his message. Authentication cost in terms of key material: 64
- Alice informs Bob of the bases she has chosen for preparing the qubits and she appends an authentication tag to her message. Authentication cost in terms of key material: 64

Phase 3.1: Reconciliation - Error estimation

Alice and Bob estimate the error rate in their sifted keys to determine whether they should proceed to error correction or whether they should abort the protocol based on a predefined error tolerance threshold, usually around 11%. Further details:

- Alice and Bob permute their sifted keys in order to flatten the errors across the entire bit string. They then perform the error estimation by comparing a subset of their error-flattened sifted keys.
- An error rate of 0.0784 was estimated using a sample size of 51 given a sampling ratio of 0.2

Phase 3.2: Reconciliation - Error Correction, Cascade

Alice and Bob perform an interactive error correction scheme called Cascade on the public channel in order to locate and correct the erroneous bits in their sifted bit strings. Further details:

- Cascade was run 6 rounds in order to correct the errors.
- 18 erroneous bits were detected and corrected.
- 114 bits were leaked in order to correct the errors.
- With an error probability of 0.0874, the Shannon bound for the number of leaked bits is: 89.0, compared to the actual number of leaked bits: 114.

Phase 4: Error Correction Confirmation and Authentication

Alice and Bob confirm and authenticate the error correction phase by computing the hash of their error corrected keys using their mutually preshared secret key and by comparing their respective digests. Further details:

- 64 bits of key material (preshared secret key) were used to authenticate.
- The Linear Feedback Shift Register (LFSR) universal hashing scheme was used for authentication.

Phase 5: Privacy Amplification

Alice and Bob compute the overall information leakage and run a privacy amplification protocol in order to reduce/minimize Eve's knowledge gained on the key by having eavesdropped on the channel. They do so by locally applying a universal hashing scheme based on Toeplitz matrices. The hashing function will be indexed using yet another chunk of their preshared secret keys. They can also define a security parameter to minimize Eve's knowledge to an arbitrary amount. Further details:

- 146 bits were leaked up to this point.
- The key length before running privacy amplification: 206 bits.
- The final key length is: 40 bits.
- The chosen security parameter is: 20.

Statistics and Overview

Property	Value
Initial number of qubits	500
Final key length	40
Estimated error	0.0784
Eavesdropping enabled	1
Eavesdropping rate	0.1
Alice/Bob basis selection bias	0.5
Eve basis selection bias	0.5
Raw key mismatch before error correction	0.0856
Raw key mismatch after error correction	0
Information leakage (Total number of disclosed bits)	146
Overall key cost for authentication	256
Key length before error correction	206
Bit error probability	0.0874
Bits leaked during error correction	114
Shannon bound for leakage	89
Security parameter	20

Figure 28: Example of a result

QKD Sifting Plot

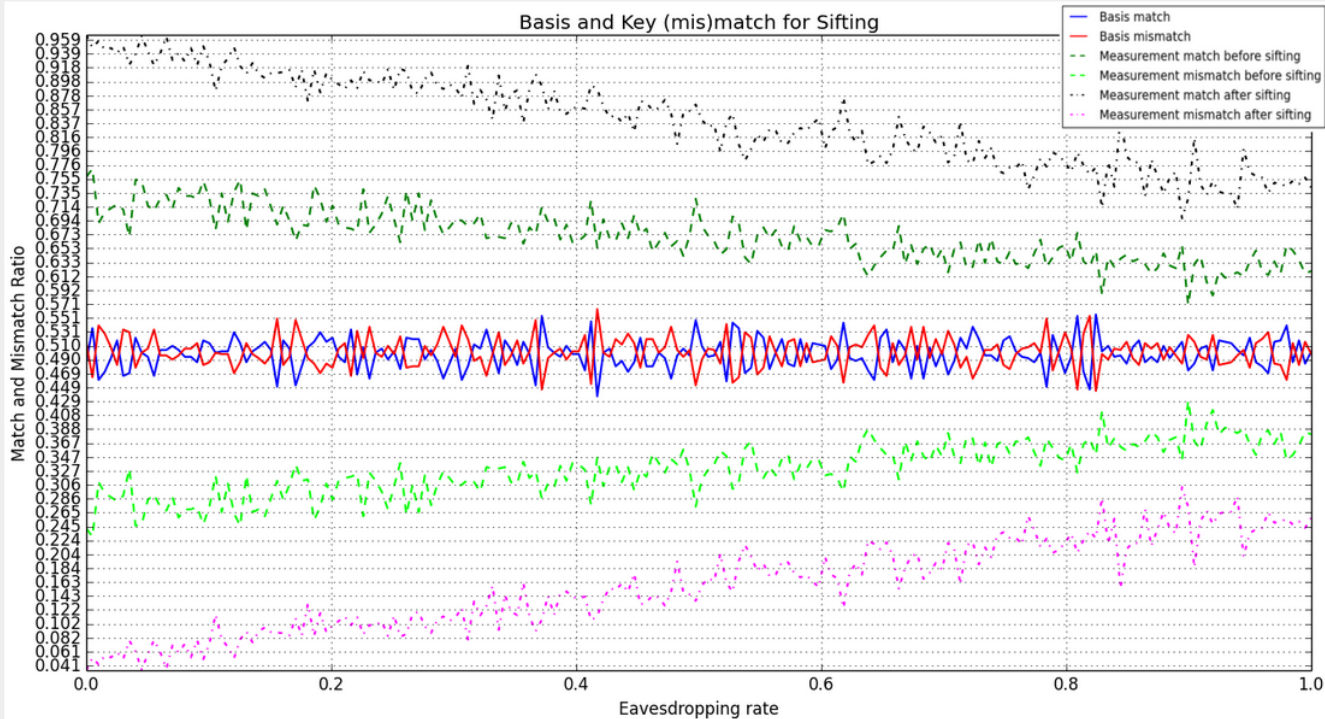


Figure 29: Elaborate and detailed plots for each result

Result

The QKDSimulator impresses especially those users who are mainly interested in QKD and want to understand this algorithm without installing any program, registering or programming themselves. The documentation is not yet available, but the results are described in detail in text passages. Also, every intermediate step is explained in detail and the plots are of high quality. The web application has only a few pages, which is both an advantage and a disadvantage. Currently, only the BB84 variant is supported, which is the most original and unmodified method of quantum key distribution.

Amazon (Braket)

Repository	https://github.com/aws/amazon-braket-sdk-python
Language	Python
OS	Python interpreter required
Type	Platform
Focus	AWS Integration
Feature	Processors from D:Wave, IonQ, Rigetti, OQC
License	Apache 2.0 License
Website	https://aws.amazon.com/de/braket/
Registration	Not for the SDK, for the platform yes (credit card)



Description

Amazon Braket is a fully managed AWS service designed to help researchers, scientists, and developers get started with quantum computing. Algorithms can be designed, tested, and run on various quantum circuit simulators and real quantum hardware in the web interface. In addition, it is also possible to design hybrid algorithms that include classical resources. Amazon also supports the use of Jupyter Notebooks with pre-installed algorithms, resources, and developer tools. Braket provides on-demand access to various types of quantum computers. Access to gate-based quantum computers from IonQ and Rigetti, as well as to a quantum annealer from D-Wave, is to be made as easy as possible for the user without having to obtain access from individual providers.

Installation

One thing in advance: A large part of the features mentioned above are only available by registering with AWS. Even if they are free plans and there are limited functionality and monthly or absolute time restrictions, credit card information is necessary to complete a registration.

For this reason, we will limit ourselves to installing the Amazon Braket Python SDK at this point. Although this also has a strong connection to Amazon's AWS, it can be easily installed on a local computer outside the cloud and has a local simulator, which is also used in the following example.

As always, it is recommended to create a virtual Python environment. The installation of the Braket SDK is done with:

```
pip install amazon-braket-sdk
```

Example

In our example, we want to create a Bell pair (EPR-pair) by applying a Hadamard gate to QuBit 0, or we want to create a CNOT relationship between QuBit 0 and 1, in which QuBit 0 acts as the control QuBit. As already mentioned, we select the local simulator included in the SDK as the simulator device and let the circuit execute 100 times and output the result in the command line (turquoise mark).

This example also shows that with Braket, little code is enough to run a simple simulation, although Amazon tries to lure the user into the cloud for their first steps by referring mainly to their WEB IDE in their documentation and to devices in the cloud for execution.

Amazon (Braket)

```
>>> from braket.devices import LocalSimulator>>> device =  
LocalSimulator("default")>>> bell = Circuit().h(0).cnot(0, 1)>>>  
task = device.run(bell, shots=100)>>>  
print(task.result().measurement_counts)  
Counter({'00': 56, '11': 44})
```

Result

As you can see, the result is only 00 and 11, which corresponds to an ideal process without noise and thus a pure simulation without external interference. With a distribution of 56% to 44% in this run, one can speak of an equal distribution for 100 shots, which would approach the 50%/50% distribution with an infinite number of repetitions.

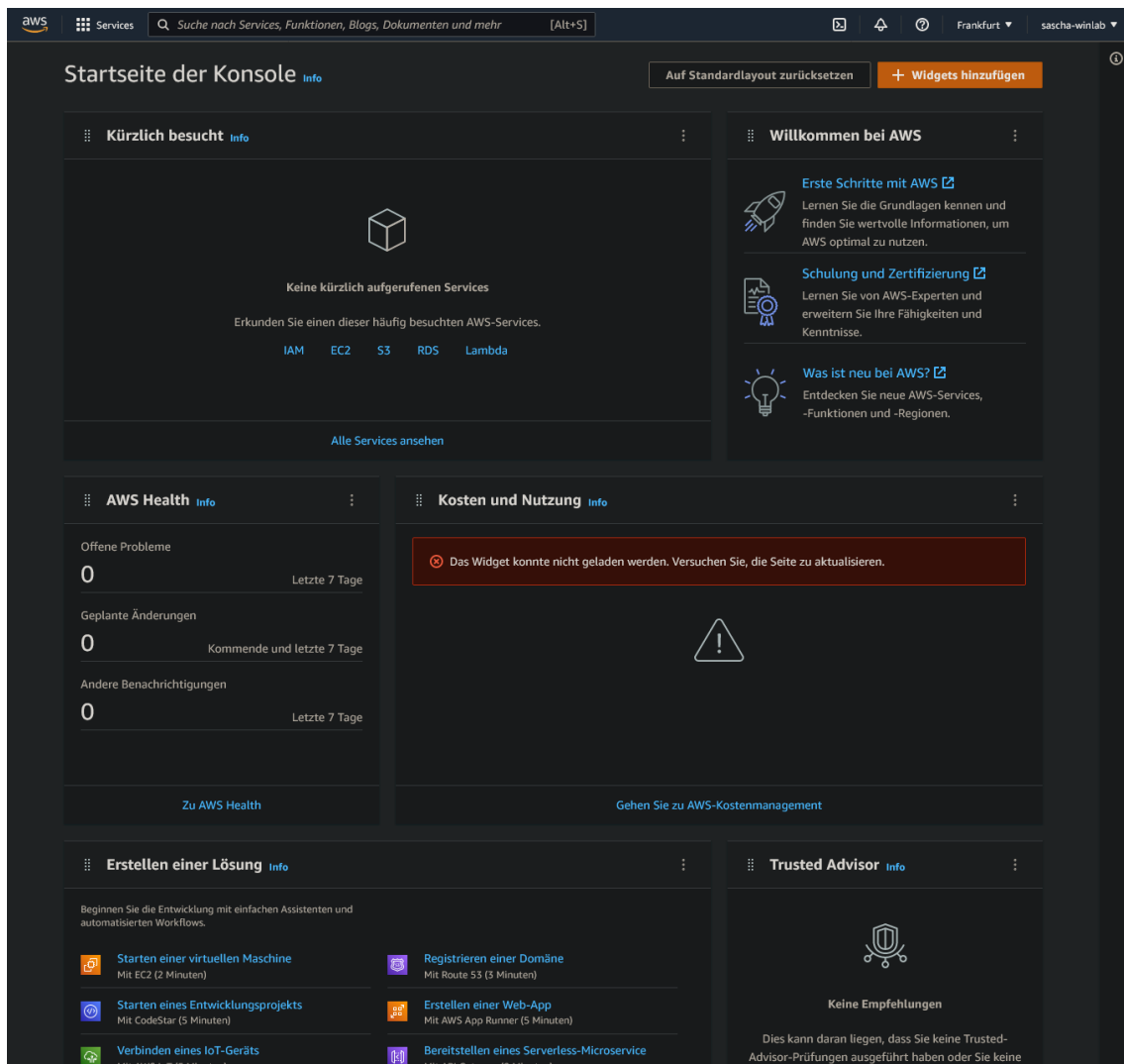


Figure 30: AWS landing page

Result

With Braket, Amazon offers convenient access to an entire environment with build, test and run tools on the topic of quantum computing. Fortunately, the core of this service, i.e. the Braket SDK, is available to the user without registration outside the cloud, without having to commit to Amazon for initial tests.

Quantum Programming Studio

Repository	https://github.com/quantastica/quantum-circuit
Language	Drag&drop / JavaScript
OS	Web browser
Type	Circuit Simulator
Focus	Extensive REST API
Feature	Conversion to many formats/languages
License	MIT License
Website	https://quantum-circuit.com
Registration	A free registration is required to use the web interface

Description

quantum-circuit is an open-source quantum circuit simulator implemented in Javascript. According to the developers, simulations with 20+ qubits in the browser or on the server are no problem.

The Quantum Programming Studio is a web-based graphical user interface that allows users to construct quantum algorithms and then simulate them directly in the browser or run them on real quantum computers. The circuit can be exported to multiple quantum programming languages/frameworks and can be run on various simulators and quantum computers.

Supported platforms include: Rigetti Forest, IBM Qiskit, Google Cirq and TensorFlow Quantum, Microsoft Quantum Development Kit, Amazon Braket.

Installation

No installation is necessary. One can use a simple drag and drop interface to create a schematic that automatically translates into code, and vice versa – you can enter the code and the plan will update accordingly.

Property	Value
Qubits	2
Columns	4
Gates	4

Gates	
Name	Count
h	1
cx	1
measure	2

Registers	
Name	Size
c	2

Figure 31: The Drag&Drop Editor (here: Circuit Bell Pair)

Quantum Programming Studio

If you are still interested in installing your own, you can implement your own instance in JavaScript with `npm install --save quantum-circuit`. Development can also be done with Jupyter Notebook, as long as a JavaScript kernel is installed.

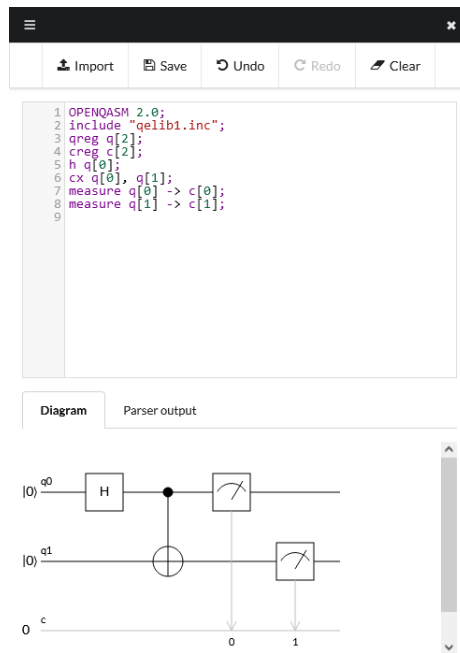


Figure 32: Automatic translation into code (code editor)

Registration

In order to use the service, a free registration under <https://quantum-circuit.com> is required. Projects can be saved there and shared with others. At the moment, 1437 projects are open to the public.

Simulation in the browser

The easiest way to simulate is to run it directly in the browser. To do this, simply open the desired project and then execute it via the "Simulation" tab. For each click on the "Simulate" button, one pass is triggered.

Quantum Programming Studio

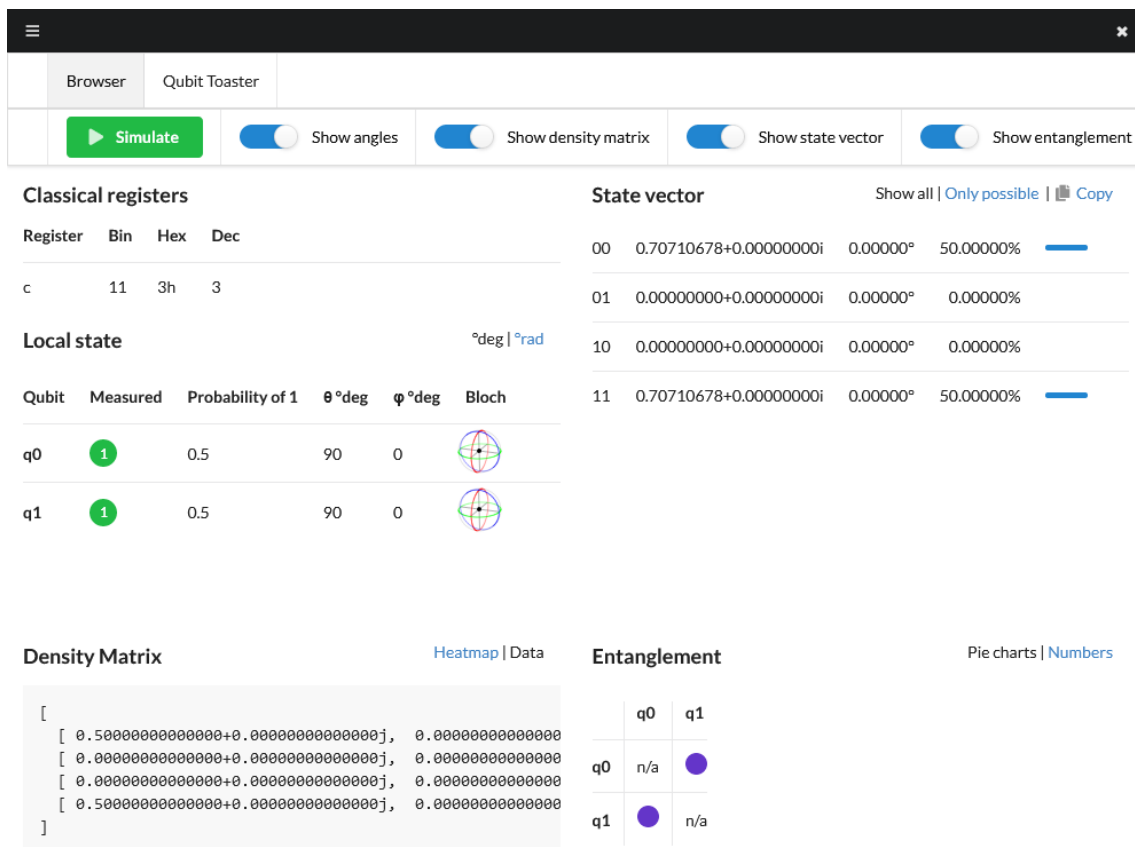


Figure 33: Results of the simulation in the browser

Simulation in the cloud and on real hardware

The "Run" tab opens a documentation with a description of which remote backends can be used and what is needed. Mainly the QPS client is needed here, which connects to the QPS server via web sockets. With this client, Quantum Programming Studio UI can be connected to Rigetti QCS, Rigetti Forrest SDK, IBM Qiskit and Quantastica Qubit Toaster. However, Rigetti's resources seem to be withheld from users who register with Rigetti as members of an organization.

Apis

An important core feature is interoperability with other languages. For example, circuits can be imported from OpenQASM and Quil and circuits can be exported to OpenQASM, pyQuil, Quil, Qiskit, Cirq, TensorFlow Quantum, QSharp and QuEST. Circuits can also be saved in SVG format.

Via the Rest API, almost any format can be served by a simple http request. The appropriate links can be found in the respective project details.

Quantum Programming Studio

REST API

You can fetch this circuit's source code from your program/script via simple HTTP call:

QUIL

```
https://quantum-circuit.com/api/get/circuit/G3woMiJoPLNwdDX53?format=quil
```

pyQuil

```
https://quantum-circuit.com/api/get/circuit/G3woMiJoPLNwdDX53?format=pyquil
```

Qiskit

```
https://quantum-circuit.com/api/get/circuit/G3woMiJoPLNwdDX53?format=qiskit
```

OpenQASM

```
https://quantum-circuit.com/api/get/circuit/G3woMiJoPLNwdDX53?format=qasm
```

Qobj

```
https://quantum-circuit.com/api/get/circuit/G3woMiJoPLNwdDX53?format=qobj
```

Braket

```
https://quantum-circuit.com/api/get/circuit/G3woMiJoPLNwdDX53?format=braket
```

Cirq

```
https://quantum-circuit.com/api/get/circuit/G3woMiJoPLNwdDX53?format=cirq
```

TensorFlow Quantum

```
https://quantum-circuit.com/api/get/circuit/G3woMiJoPLNwdDX53?format=tfq
```

Q#

```
https://quantum-circuit.com/api/get/circuit/G3woMiJoPLNwdDX53?format=qsharp
```

JavaScript

```
https://quantum-circuit.com/api/get/circuit/G3woMiJoPLNwdDX53?format=javascript
```

Toaster

```
https://quantum-circuit.com/api/get/circuit/G3woMiJoPLNwdDX53?format=toaster
```

QuEST

```
https://quantum-circuit.com/api/get/circuit/G3woMiJoPLNwdDX53?format=quest
```

SVG (stand-alone)

```
https://quantum-circuit.com/api/get/circuit/G3woMiJoPLNwdDX53?format=svg
```

SVG (inline)

```
https://quantum-circuit.com/api/get/circuit/G3woMiJoPLNwdDX53?format=svg-inline
```

Figure 34: Rest API of the test project

Result

The Quantum Programming Studio is a recommendation for all friends of a web-based, graphical user interface. It makes it possible to construct quantum algorithms and simulate them directly in the browser. If this is not enough, real quantum computers are also available. The circuits can be exported to multiple quantum programming languages/frameworks, etc., platform-independently, and run on various simulators and quantum computers. The simple user interface automatically creates code, or translates code into a circuit; Charts are updated in real-time.

Microsoft Azure Quantum/QKD/Q#



Repository	https://github.com/microsoft/QuantumLibraries
Language	Q#, Python
OS	Azure packages can be installed platform-independent
Type	QKD or platform
Focus	Simulation of quantum circuits, optimization tasks, simulation of a quantum computer with fullstack QKD
Feature	Access to other providers
License	MIT License
Website	https://azure.microsoft.com/de-de/resources/development-kit/quantum-computing/
Registration	Not necessary for QDK; You get credits for Azure (Quantum)

Description

The QDK is the development kit for Azure Quantum. Quantum applications can be created and executed with Q#, Qiskit or Cirq, both on real quantum hardware and with classical simulators, online or offline. Other tasks such as optimizations, etc. can also be formulated and carried out. As with IBM Quantum/Qiskit and Google Quantum AI/Cirq, a distinction must be made between the actual framework (SDK) and the platform built on top of it. The installation of the SDK is possible, although Microsoft also tries to lure the user to the platform for development and promises credits for various resources.

Q# is a quantum-focused high-level programming language from Microsoft and offers an approach to the development of quantum programs.

QDK

The Quantum Development Kit includes feature-rich integration with Visual Studio, Visual Studio Code, and Jupyter Notebooks. The Q# programming language can be used standalone, in notebooks and on the command line, or via the host language with Python and .NET interoperability.

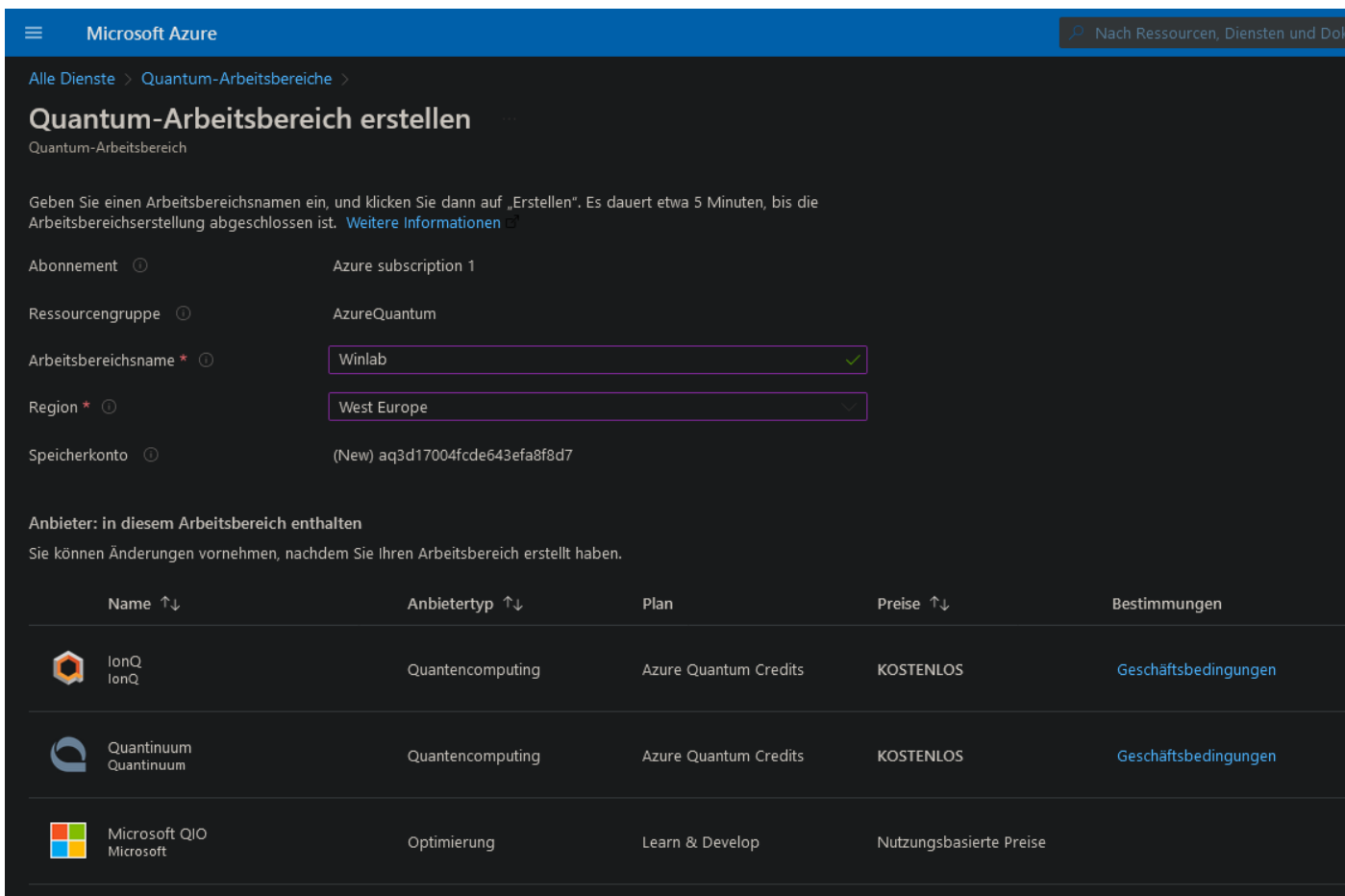


Figure 35: Creating a workspace in Azure

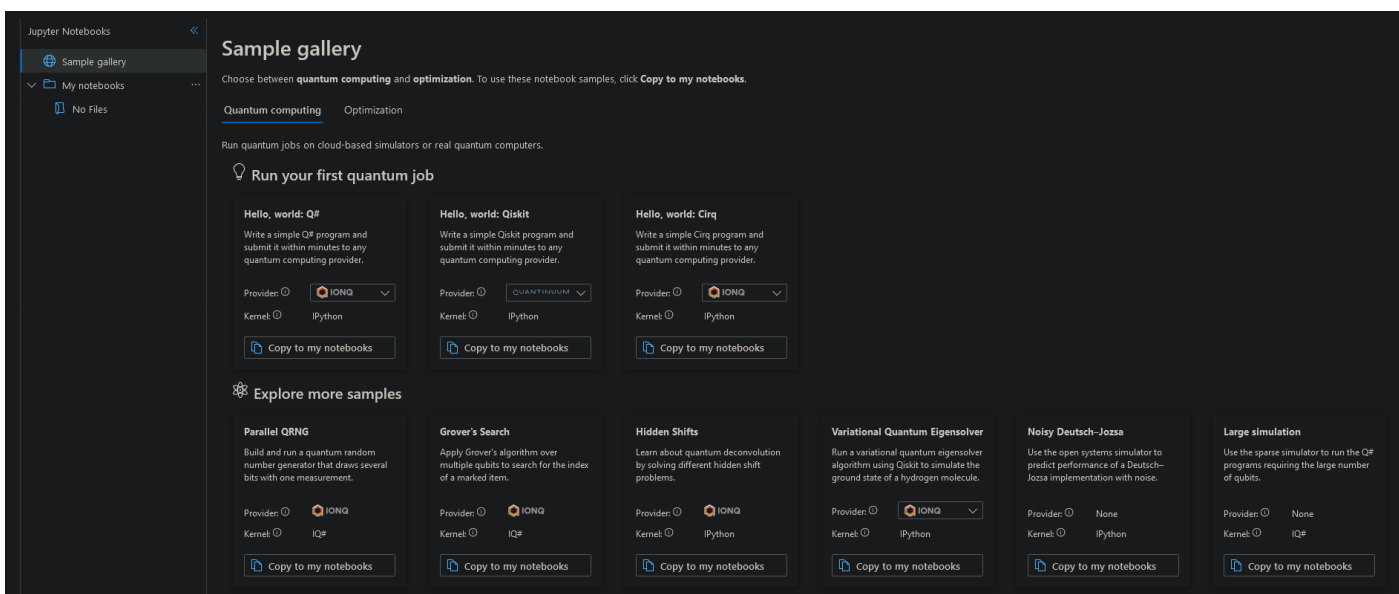


Figure 36: Azure Quantum already offers some predefined programs and circuits

Installation

There are various ways to install the QDK. This means that Q# not only runs together with Python and .NET, but other driver programs for host languages such as C# or F# can also be used. Both VS Code and Jupyter notebooks are suitable for development, or a client-server relationship between VS Code and notebooks.

For this test, we decided to use a WSL2 environment with Ubuntu 20.04. As recommended by Microsoft, Anaconda was installed as a Python distribution and an environment was created specifically for QKD.

After initialization by the qsharp package, the Jupyter server can be started and a notebook with a Q# kernel can be created.

Installation Anaconda (Download Installation Script)

```
bash Anaconda3-2022.05-Linux-x86_64.sh
```

Creation and activation of new virtual environment including required packages

```
conda create -n qsharp-env -c microsoft qsharp notebook
conda activate qsharp-env
```

Initialization

```
python -c "import qsharp"
```

Starting the Notebook Server

```
Jupyter Notebook
```

Creating a New Notebook with the Q# Kernel

```
New → Q#
```

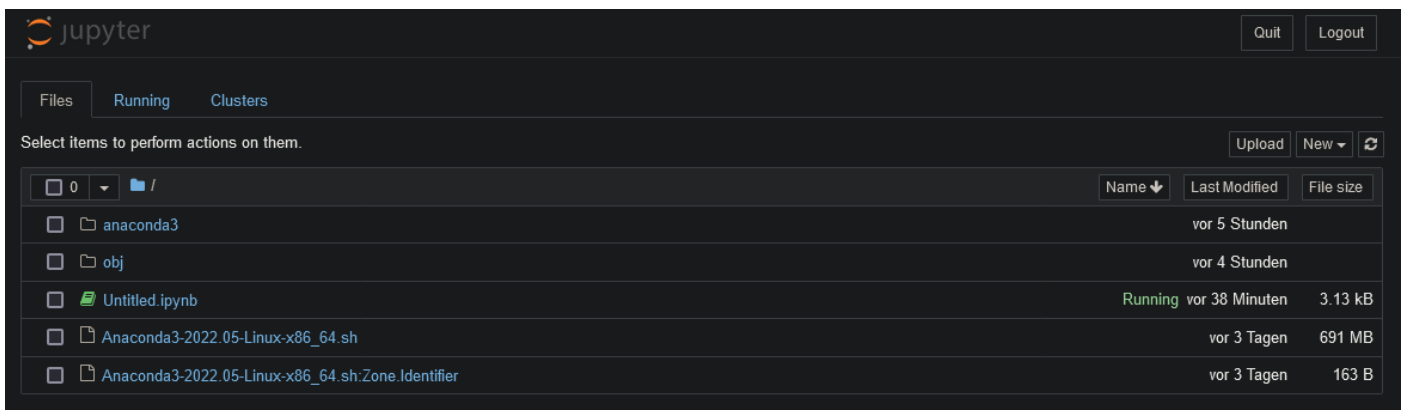


Figure 37: Jupyter Notebook Web Server

Example

In this example, a Qubit in the state $|0\rangle$ and are superposed by the function $H()$ that there is a 50% chance of either 0 or 1 in the following measurement, see Figure 38.

```
operation SampleQuantumRandomNumberGenerator() : Result {
    use q = Qubit(); // Allocate a qubit in the |0> state.
    H(q); // Put the qubit to superposition. It now has a 50% chance
of being 0 or 1.
    let r = M(q); // Measure the qubit value.
    Reset(q);
    return r;
}
```

With the *magic* function below, the result can be output directly below the cell.

```
%simulate SampleQuantumRandomNumberGenerator
```

The screenshot shows a Jupyter web interface with a dark theme. At the top, it says 'jupyter Untitled Last Checkpoint: vor 5 Stunden (autosaved)'. Below the title bar is a menu with 'File', 'Edit', 'View', 'Insert', 'Cell', 'Kernel', and 'Help'. To the right of the menu are 'Trusted' and 'Q#' buttons. Below the menu is a toolbar with icons for file operations and a 'Code' dropdown menu. The main area contains several input-output pairs:

```
In [1]: operation SampleQuantumRandomNumberGenerator() : Result {
    use q = Qubit(); // Allocate a qubit in the |0> state.
    H(q);           // Put the qubit to superposition. It now has a 50% chance of being 0 or 1.
    let r = M(q);  // Measure the qubit value.
    Reset(q);
    return r;
}
```

```
Out[1]: • SampleQuantumRandomNumberGenerator
```

```
In [2]: %simulate SampleQuantumRandomNumberGenerator
```

```
Out[2]: One
```

```
In [3]: %simulate SampleQuantumRandomNumberGenerator
```

```
Out[3]: One
```

```
In [4]: %simulate SampleQuantumRandomNumberGenerator
```

```
Out[4]: One
```

```
In [5]: %simulate SampleQuantumRandomNumberGenerator
```

```
Out[5]: Zero
```

```
In [8]: %simulate SampleQuantumRandomNumberGenerator
```

```
Out[8]: One
```

Figure 38: Example of a simple program on the web interface of the Jupyter web server

Result

The open-source Quantum Development Kit for Azure Quantum provides tools for developing quantum applications on hardware-accelerated compute resources in Azure or on the on-premises host. According to Microsoft, Q# is completely hardware-agnostic, which means that quantum computing concepts can be expressed independently of future developments. Q# programs can also be targeted to run on various quantum hardware backends in Azure Quantum. A Q# program can be compiled into a standalone application or replaced by a Q# program created in Python or in a .NET language.

All in all, Microsoft's Azure Quantum platform is a full-fledged environment like Amazon Braket, or IBM Quantum Qiskit, but with the difference that Microsoft uses its own Q# language for this.

Closing remarks

We hope that this report will make it easier to enter this extremely interesting world of quantum simulators and that testers will have a lot of fun with further application examples. The test report could only take into account current simulators and platforms; it can be assumed that these frameworks are constantly changing and that new simulators will also be used in the future.