# Quantensimulatoren

Test und Übersicht der vielversprechensten Simulatoren

Testbericht des DFN WiN-Labors in Erlangen, Stand 15.07.2022

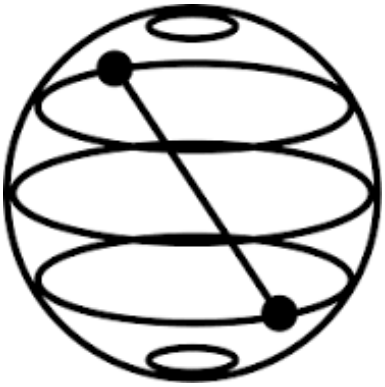
## Inhalt



Vorwort

Im vorliegenden Bericht werden verschiedene Simulatoren vorgestellt, die in Einzeltests näher untersucht wurden. Es handelt sich dabei um Plattformen, Software Development Kits (SDKs), Standalone Simulatoren und Simulationsengines. Für alle untersuchten Simulatoren werden deren Eigenschaften beschrieben und es wird erklärt für welche Anwendung sie am passendsten sind. Darüber hinaus gibt es ein Anwendungsbeispiel, dass in einer konkreten Installation getestet wurde; die schnellste bzw. praktischste Installtionsmethode wird auch erläutert. Jeder Testbericht wird mit einem Fazit abgeschlossen.

# Qiskit



| | |
|---|---|
| **Repository** | https://github.com/Qiskit/qiskit |
| **Sprache** | Python |
| **OS** | Cross-platform |
| **Typ** | Software Development Kit (SDK) |
| **Fokus** | Impuls / Schaltungen |
| **Besonderheit** | Web-Plattform/IDE vorhanden[1] |
| **Lizenz** | Apache License 2.0 |
| **Website** | https://qiskit.org |
| **Registrierung** | nein |

Bei Qiskit handelt es um ein von IBM bereitgestelltes SDK in Python zum Erstellen und Ausführen von Algorithmen und Schaltungen und zur generellen Arbeit mit rauschbehafteten Quantencomputern. Qiskit kann völlig unabhängig auf jedem **gängigen Betriebssystem** installiert werden und kann selbstständig Simulationen durchführen. Es besteht aus mehreren Komponenten, wobei hier nur auf zwei wesentliche eingegangen wird.

*Qiskit Terra* ist die Basiskomponente auf der alle anderen aufbauen. Terra bietet die Möglichkeit für die Zusammenstellung von Quantenprogrammen auf der Ebene von **Schaltkreisen** und **Impulsen** und verwaltet die Ausführung von Stapeln von Experimenten auf entfernten Geräten und die Backend-Kommunikation.

*Qiskit Aer* bietet verschiedene Simulatoren für Quantencomputer mit realistischen Rauschmodellen, die **lokal** auf dem Gerät des Nutzers gehostet werden, oder **HPC-Ressourcen**, die über die Cloud verfügbar sind. IBM selbst nennt seine Simulatordienste „fortgeschrittene cloudbasierte klassische Emulatoren von Quantensystemen“. An dieser Stelle sei bereits erwähnt, dass Qiskit eng mit der „Quanten-Cloud“ von IBM verbandelt ist und dass eine Simulation sowohl lokal auf dem Notebook, als auch auf „klassischer“ Hardware in der IBM-Cloud ausgeführt werden kann. Der Clou hierbei ist jedoch, dass auch beschränkt[2] echte Quantenhardware kostenlos zur Verfügung steht und man Simulation und echten Prozess vergleichen kann.

*Interface(s)* – Ebenfalls hervorzuheben ist, dass man darüber hinaus die Wahl hat, ein lokales Userinterface zu verwenden oder auf IBMs Webinterface zurückzugreifen, die beide Jupyter Notbooks unterstützen. Vorteil des Web-Composers (Abbildung 1) ist hierbei der drag & drop Editor, welche die gestaltete Schaltung direkt nach Python überführt und in Echtzeit Ergebnisprognosen anzeigt.

*Backends* stellen entweder einen Simulator oder einen echten Quantencomputer dar und sind für die Ausführung von Quantenschaltungen und die Rückgabe von Ergebnissen zuständig. Wie bei einigen anderen Software-Simulatoren[3] lassen sich diese austauschen. Tabelle 1 zeigt die von IBM zur Verfügung gestellten Backend-Simulatoren. Der Statevector ist hier der default Simulator.

[1] Nicht obligatorisch
[2] Wartezeit; nicht alle Prozessoren und Standorte kostenlos
[3] Als Software-Simulator wird hier das gesamte Framework/SDK bezeichnet, welches als Backend einen austauschbaren Simulator im Sinne eines Simulationskerns betreibt

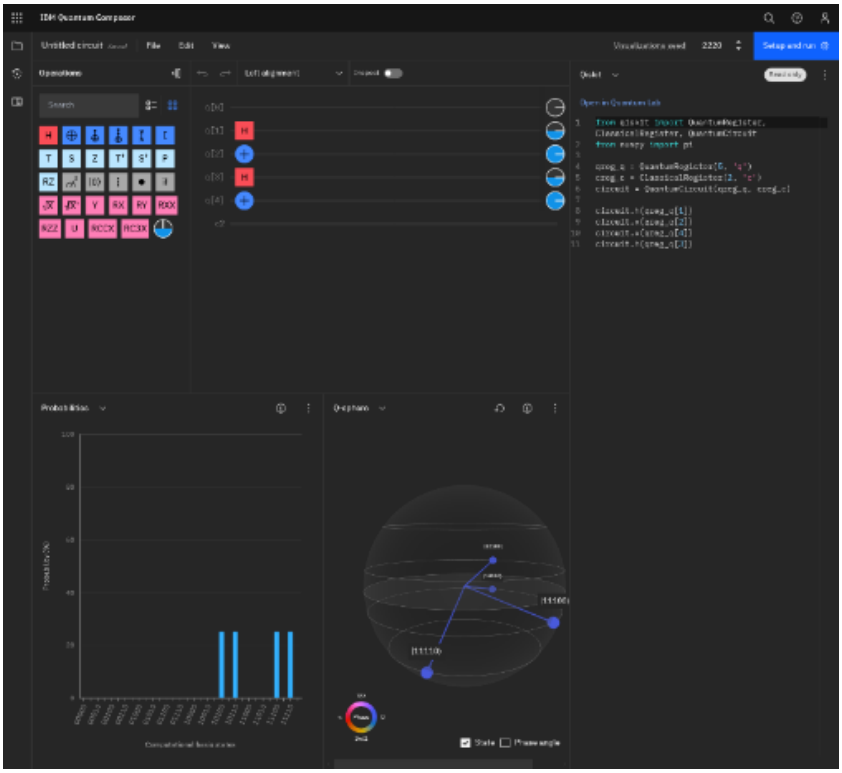


*Abbildung 1: IBMs Web-Composer*

*Tabelle 1: Verfügbare Backend-Simulatoren*

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

*Community* – Ein großer Vorteil des Frameworks liegt in seiner großen Community und damit verbunden eine Vielzahl an Medien. Es existieren bereits jetzt schon eine Vielzahl an Medien und Möglichkeiten sich als Interessierter mit der Welt des Quantum-Computing zu beschäftigen. Die Qiskit Foundation selbst unterhält einen YouTube-Kanal mit vielen Tutorials, Talks und Hintergründen zum Thema und veranstaltet immer wieder Quantum Challenges, Camps, Hackathons und Seminare und Meetups. Außerdem unterhält IBM einen eigenen Workspace in Slack, dem man beitreten kann.

*Installation* – Wer zunächst eine Installation scheut, kann direkt im in IBMs Quantum[4] Lab beginnen. Wer lieber lokal arbeiten möchte, kann mit `pip install qiskit` die Installation anstoßen. Seitens des Herstellers wird eine virtuelle Umgebung mit Anaconda empfohlen. Da wir hier nur eine kleine Schaltung demonstrieren wollen, begnügen wir uns mit der vorinstallierten Python-Distribution und erzeugen eine virtuelle Umgebung mit `python3 -m venv qiskit-env/`[5] und installieren neben `qiskit` noch `matplotlib`, um die Ergebnisse visualisieren zu können.

*Simulation auf klassischer Hardware* – Wir erzeugen nun eine relativ simple Schaltung mit 3 QuBits, welche wir mittels Hadamard-Gatter in Superposition versetzen und direkt danach messen. Wir starten Python

[4] Registrierung notwendig
[5] Aktivierung mit `source qiskit-env/bin/activate`

und importieren aus Gründen der Einfachheit die gesamte Bibliothek `from qiskit import *`. Danach erzeugen wir eine Schaltung mit jeweils 3 QuBits und 3 klassischen Bits, um dort das Ergebnis zu speichern `qc = QuantumCircuit(3,3)`. Die QuBits befinden sich anfangs im Zustand $|0\rangle$, weshalb eine Messung ebenfalls 0 ergeben würde. Wir loopen deshalb durch alle QuBits und fügen jeweils das H-Gatter und eine Messung hinzu. Da sich nun alle QuBits im Zustand $|+\rangle$[6] befinden können wir durch Messung nun einen zufälligen Wert 0 oder 1 erwarten. Mit `qc.draw()` kann die erzeugte Schaltung gezeichnet werden, siehe Abbildung 2. Nach Auswahl des Backends (Tabelle 1) wird der „Job“ ausgeführt. Zuletzt werden die Ergebnisse in einem Histogramm geplottet.

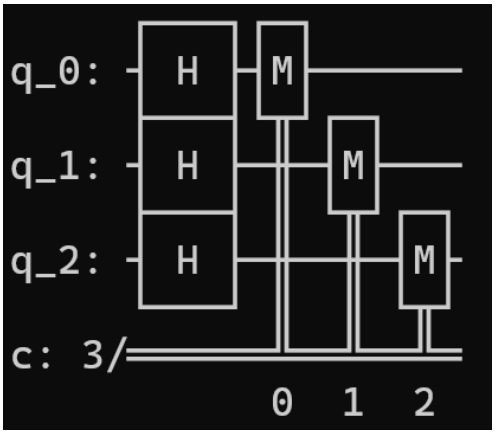



*Abbildung 2: Beispiel einer einfachen Schaltung*

```
from qiskit import *
from qiskit.tools.visualization import plot_histogram
qc = QuantumCircuit(3,3)
for j in range(3):
      qc.h(j)              #Hadamard-Gatter
      qc.measure(j,j)      #Messung
qc.draw()                  #Zeichnen der Abbildung 2
backend = Aer.get_backend('qasm_simulator')  #Wahl des Backends
job = execute(qc, backend)                   #Ausführen
plot_histogram(job.result().get_counts())    #Plotten des Histograms
```

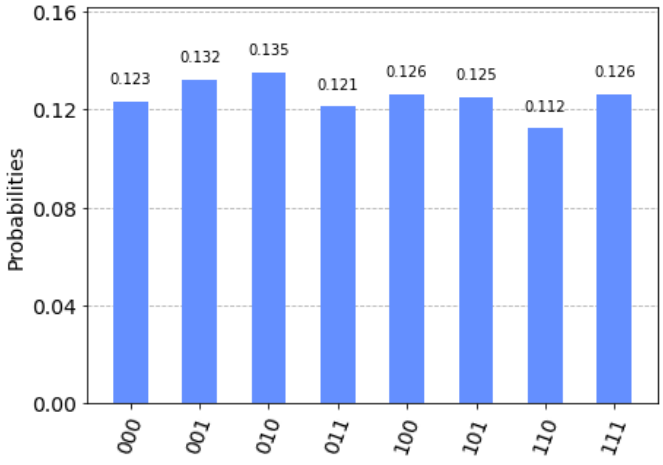



*Abbildung 3: Ergebnis-Histogramm der Beispielschaltung*

[6] Superposition

Das Histogramm aus unserem Beispiel soll zeigen, dass durch Anwendung eines Hadamard-Gatters ein Superpositionszustand entsteht, in welchem das QuBit nach der Messung jeweils mit einer Wahrscheinlichkeit von 50% in den Zustand 0 oder 1 zerfällt. Da wir dieses Gatter auf alle QuBits gleichermaßen angewendet haben, ergibt sich ein stochastisch ausgewogenes Histogramm, in welchem die Wahrscheinlichkeit des Auftretens aller Permutationen gleich ist.

*Echte Quantenhardware* – IBM stellt für interessierte Benutzer nicht nur verschiedene Simulator Typen, einen Circuit-Composer, sowie ein Lab[7] zur Verfügung, sondern ebenfalls **reale Quantenhardware**. Diese wird im Rahmen der IBM Quantum Services angeboten und wird als „System“ bezeichnet. Einige dieser Systeme sind sogenannten „Premium Plan Clients“ vorbehalten, jedoch gibt es zum aktuellen Zeitpunkt 7 Systeme, welche über eine Standardregistrierung bei IBM Quantum für die Öffentlichkeit erreichbar sind. Es gibt verschiedene Prozessorarchitekturen, Revisionen und Designvarianten. Einige davon sind auch Untersektionen eines größeren Chips. Ein Übersicht aktueller Architekturen und ihrer Spezialitäten ist in Abbildung 5 zu finden. Neben den Systemen werden ebenfalls Programme angeboten, welche die Ausführung von Schaltungen beschleunigen soll.

*QV und CLOPS* – Neben der Anzahl an Qubits sind auch andere Attribute, die die Leistung, Qualität oder Skalierbarkeit eines Systems beschreiben, wichtig. IBM gibt bei seinen verfügbaren Quantenprozessoren in erster Linie QV und CLOPS an, welche eine Rolle bei der Ausführung von Programmen und Schaltungen spielen sollen.

QV oder „Quantum Volume“. Ist ein Wert, der die Performance von Gatter basierten Quantencomputer wiedergibt, unabhängig von der zugrundeliegenden Technologie.
CLOPS oder “Circuit Layer Operations Per Second”, ist ein Wert, der angibt wie viele Schichten einer QV Schaltung eine QPU (Quantum Processing Unit) pro Zeiteinheit ausführen kann.

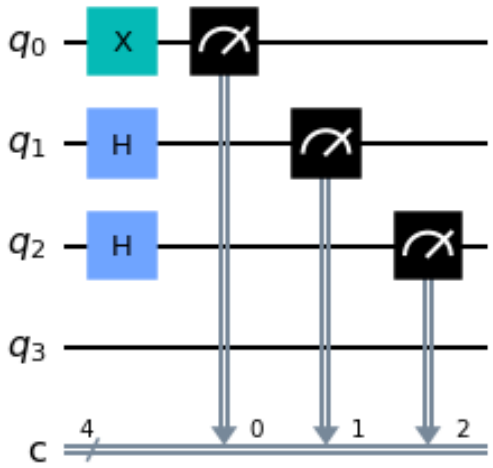



*Abbildung 4: Beispiel einer weiteren einfachen Schaltung*

*Ausführen auf realer Quantenhardware* – Wie wir bereits gesehen haben, können einfache Schaltungen mit einer geringen Anzahl an QuBits auf klassischer Hardware gut simuliert werden. Jedoch stellt ein Simulator ein ideales Quantengerät dar, welches kein echtes Quantenrauschen aufgrund von Dekohärenz[8], bzw. fehlerhafte Gatter oder Messungen kennt.

[7] IBMs Bezeichnung für deren Web-IDE
[8] Phänomen der Quantenphysik, entstehend durch ungewollte Wechselwirkung mit Umgebung

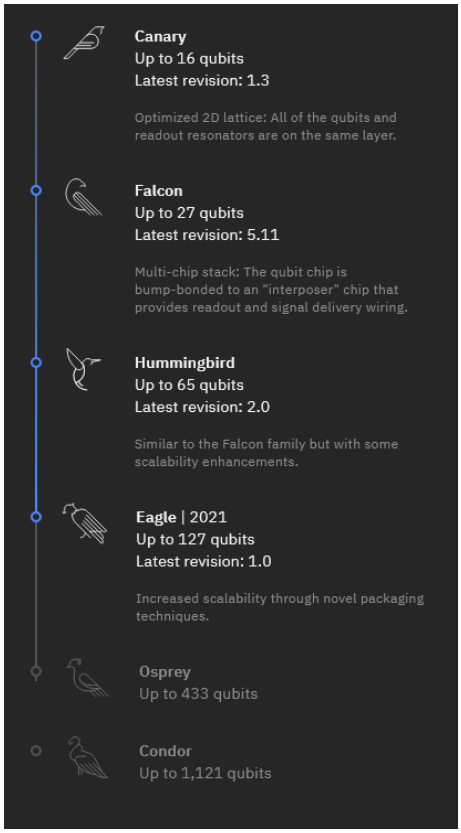



*Abbildung 5: verschiedene Prozessorarchitekturen*

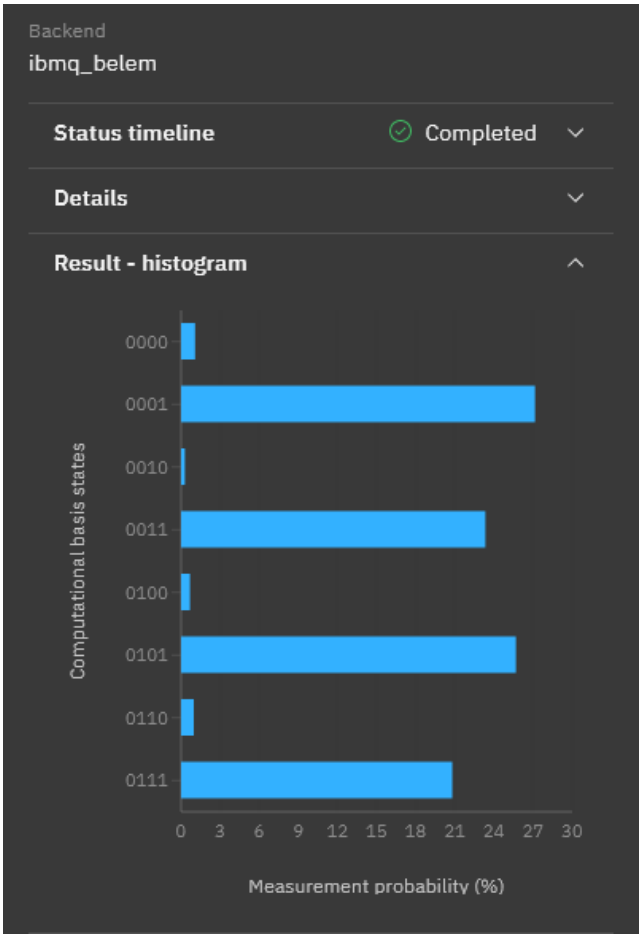



*Abbildung 6: Abgeschlossener Job; Ergebnis*

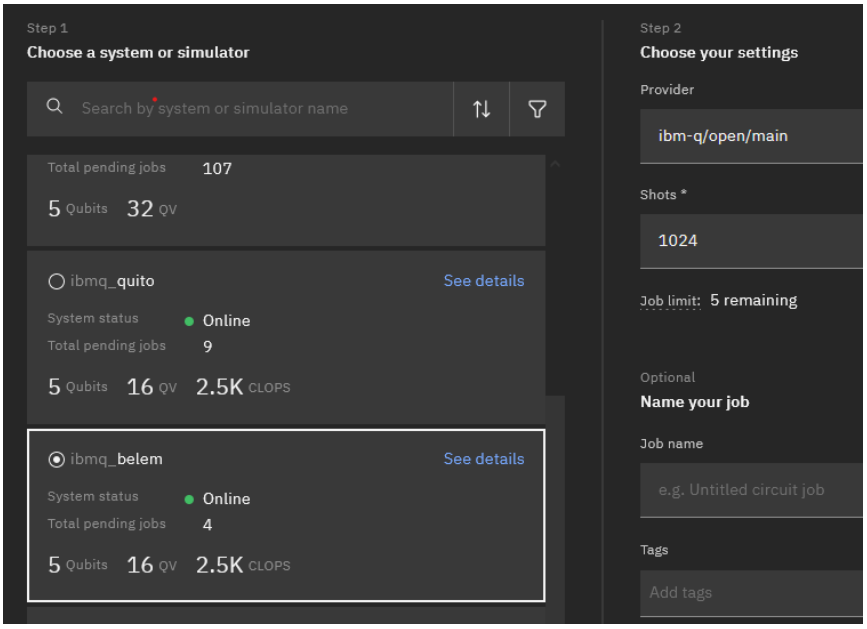



*Abbildung 7: Auswahl des Systems*

*Fazit* – Qiskit ist eine mächtige auf Impulsen und Schaltungen ausgelegte Quantensimulationssoftware, mit kontinuierlicher Entwicklung und starker Community. Aber auch netzwerknahe Konzepte lassen sich mit Qiskit auf physikalischer Ebene umsetzen. Fehlermodelle sind Teil der Software und lassen sich durch den Zugang zu realer Hardware validieren. Qiskit ist unter den Quantensimulatoren mit einem ausgewachsenen Ökosystem sehr einsteigerfreundlich, nicht zuletzt wegen dem enthaltenen „Textbook“, welches gleichermaßen Grundlagen erklärt wie auch sinnvolle Beispiele liefert.

# Quantum Network Explorer

| | |
|---|---|
| **Repository** | https://github.com/QuTech-Delft/qne-adk |
| **Sprache** | drag & drop |
| **OS** | Webbrowser |
| **Typ** | Web-Anwendung |
| **Fokus** | Netzwerk & verteilte Anwendungen |
| **Besonderheit** | Application Development Kit for Quantum Network Explorer (QNE-ADK) |
| **Lizenz** | MIT License (QNE-ADK) |
| **Website** | https://quantum-network.com |
| **Registrierung** | Nein |

Der Quantum Network Explorer ist, wie der Name bereits verrät, eine von QuTech bereitgestellte Web-Anwendung mit Fokus auf einer visuell übersichtlichen Simulation von **verteilten Anwendungen** und **Netzwerkprotokollen**. Es ist keine Registrierung notwendig und die Simulation kann betriebssystemunabhängig über den Webbrowser verwendet werden. Ein äußerst interessanter Punkt, vor allem für Netzbetreiber, ist die einfache Konfigurierbarkeit der sogenannten „**Fidelity**[9]“ für Knotenpunkte und Strecken, siehe auch Abbildung 10.

*Experimente* sind Instanzen einer Applikation, welche automatisch gestartet werden, sobald man diese ausführt. Sofern man sich dennoch registrieren möchte, ermöglicht es einem seine „Experimente“ zu speichern und zu einem späteren Zeitpunkt fortzufahren.

*Ready to Use* – Unerfahrene oder Nutzer ohne Python Programmierkenntnisse können bereits die drei Anwendungen „**Distributed CNOT**, **State Teleportation** und **QKD**“ (Abbildung 8) ausprobieren, ohne auch nur eine Zeile Code schreiben zu müssen. Diese Beispiele sind mit ausführlichen Beschreibungen der Protokolle und Bildern versehen und sollen einen schnellen Einstieg ermöglichen. Der ebenfalls vorhandene Quickstart-Guide beschreibt die ersten Schritte und gibt dem Benutzer drei verschiedene Aufgaben, um diesen an die Anwendung heranzuführen.

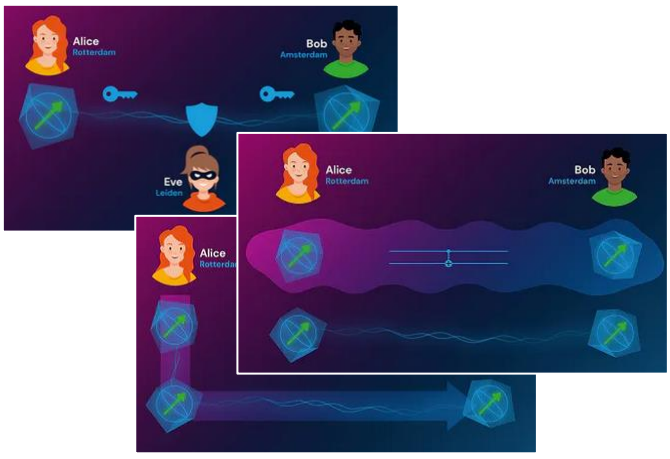



*Abbildung 8: Pre-built Applications im QNE (https://www.quantum-network.com/)*

*QNE-ADK* ist das Application Development Kit für den Quantum Network Explorer und ist an fortgeschrittene Benutzer gerichtet, die ihre eigenen Anwendungen schreiben möchten. QNE-ADK bietet

[9] **Fidelity** ist ein Maß der Güte der Quanteninformationsverarbeitung

eine CLI mit Befehlen zum Erstellen der notwendigen Dateien[10], die Anwendungen und Experimente definieren. Wenn das Experiment konfiguriert ist, kann es auf dem lokalen[11] Simulator ausgeführt werden.

*reine Webanwendung* – Es sind bereits Beispiele vorhanden, sodass weder eine Installation noch das Programmieren eigener Circuits notwendig ist. Das Tool ist zunächst rein webbasiert und verfügt über eine übersichtliche Visualisierung, was den Lernerfolg beträchtlich steigern kann. Deshalb eignet sich diese Anwendung sowohl für den neuen und unerfahrenen Anwender, also auch für den angehenden Experten, der gerne das zugehörige Application Development Kit for Quantum Network Explorer (QNE-ADK) installieren möchte. Dies beinhaltet alles, um eine eigene Quantennetzwerkanwendung zu bauen.

*Anwendungsbeispiel CNOT* – Das CNOT-Gatter ist ein exklusives Quantengatter, welches zwei Eingänge und zwei Ausgänge besitzt. Das Control-QuBit *x* (Abbildung 9) gibt an, ob QuBit *y* geflippt wird. Wie alle Quantenoperationen, ist auch das CNOT-Gate reversibel. Das heißt die Operation kann vollständig rückgängig gemacht werden. Dieses Verhalten (Entanglement) macht das Gatter für verteilte Anwendungen äußerst interessant, da die beiden verschränkten Teilchen vor der Messung verteilt werden können und durch die Messung eine Information instantan durch das „Nichts“ übertragen wird.

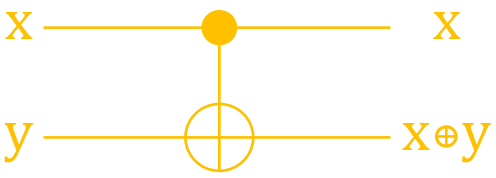



*Abbildung 9: Das CNOT-Gatter*

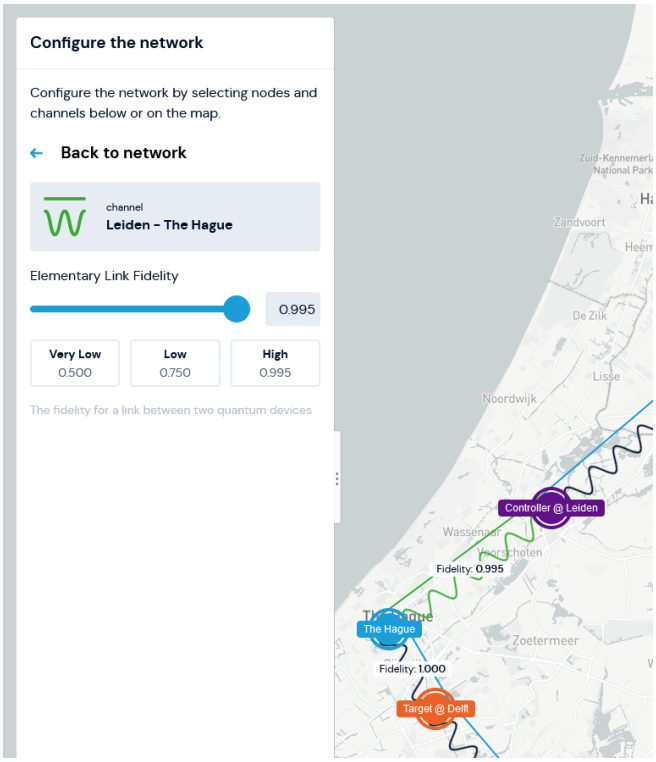



*Abbildung 10: Konfigurierbare Fidelity/Güte für Knoten und Verbindungen*

Im Folgenden wollen wir einen derartigen Algorithmus anhand eines Beispiels durchspielen, in welchem wir ein QuBit von einem Standort A (Controller) zu einem Standort B (Target) transferieren und dann

[10] Für die Entwicklung müssen Richtlinien eingehalten werden.
[11] Im Moment ist QNE-ADK nur lokal verfügbar

beobachten, wie eine Messung eines anderen QuBits am Standort A ohne weitere Kommunikation den Status eben dieses QuBits am Standort B vorgibt.

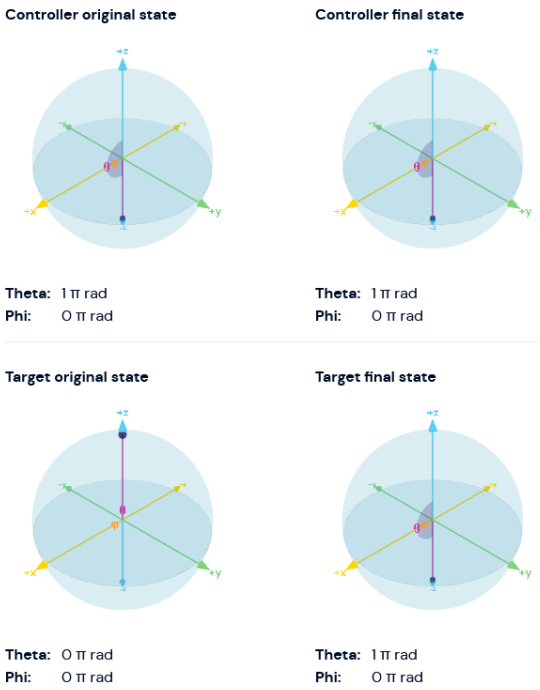



*Abbildung 11: Zustand des Control- und Target-QuBits vor und nach der Übertragung*

*Beispiel Distributed CNOT* – Als erstes nach dem Start der Applikation „Distributed CNOT“ muss eines von drei möglichen Netzwerken ausgewählt werden. Zur Auswahl stehen „Randstad“, „The Netherlands“ und „Europe“. Wie man schon vermutet, unterscheiden diese sich in der Größe und den zurückzulegenden Strecken deutlich. Wir wählen `Europe` und haben nun die Standorte: Paris, Delft, Innsbruck, Copenhagen und Barcelona zur Verfügung, um diese mit den Rollen „Controller“ und „Target“ zu versehen, wie es im Abschnitt CNOT bereits angedeutet wurde. Wir entscheiden uns für `Copenhagen` als `Controller` und `Innsbruck` als `Target`, da somit ein Großteil der Strecke durch die BRD verläuft.

Als nächstes werden beide Endpunkte z.B. mit einer `Gate Fidelity` von `0.995` konfiguriert und der dazwischenliegende Kanal mit einer relativ niedrigen `Link Fidelity` von `0.750`. Danach legen wir den Input state des `Controllers` auf $|1\rangle$, damit das `Target` QuBit mit dem Input state $|0\rangle$ flippt.

*Visualisierung* – Wenn wir nun die Simulation starten, sehen wir, wie zunächst ein verschränktes QuBit-Paar zwischen dem Controller und dem Target hergestellt wird. Interessanterweise geschieht dies nicht über die Strecke mit den geringsten Hops, sondern über eine Alternativroute mit höherer Fidelity. Danach erfolgen Präparationen der QuBits auf beiden Seiten, sowie Anwendungen von Gattern und Messungen, gefolgt von der Übertragung von Ergebnissen über einen klassischen Kanal.

*Fazit* – Wer sich bisher noch relativ wenig mit dem Thema Quantenalgorithmen auseinandergesetzt hat, sich aber trotzdem mit verteilten Anwendungen und Netzwerken beschäftigen möchte ohne sich mit der Mathematik dahinter vertraut zu machen, könnte mit dem Quantum Network Explorer gut bedient sein. Dieser sticht hauptsächlich mit seinen gelungenen Visualisierungen heraus und lässt sich über das Web gut konfigurieren, ohne etwas installieren zu müssen.

# SimulaQron

| | |
|---|---|
| **Repository** | https://github.com/SoftwareQuTech/SimulaQron |
| **Sprache** | Python |
| **OS** | Linux und macOS |
| **Typ** | Von Anwendung bis Netzwerkstack |
| **Fokus** | Verteilte Anwendung über mehrere Quantenprozessoren und Channels |
| **Besonderheit** | Erforschung der Implementierung eines Netzwerk-Stacks |
| **Lizenz** | Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: siehe Link |
| **Website** | http://www.simulaqron.org/ |
| **Registrierung** | - |


Eigenschaften:

SimulaQron ist ein Simulator, der für die Entwicklung von plattformunabhängigen Applikationen (überwiegend für die Anwendungsschicht) sowie für Netzwerkprotokolle ausgelegt ist. Jedoch ist er nicht zum Testen bzw. Simulieren von Quantenrepeatern, Codes, zur Fehlerkorrektur und QuBit Rauschen geeignet[12]. Der Simulator kann auf einem oder mehreren klassischen Rechnern ausgeführt werden, um ein Netzwerk aus verteilten Quantenprozessoren zu simulieren. Für die Entwicklung von Programmen werden Bibliotheken in Rust, Python und C bereitgestellt. Zur Simulation der Quantenprozessoren (Backend), die über ein simuliertes Kommunikationsnetzwerk miteinander verbunden sind, nutzt SimulaQron den Simulator QuTip. Die Nutzung von ProjectQ[12] ist ebenfalls möglich. Als Backend kann grundsätzlich jeder Simulator für einen Quantenprozessor verwendet werden, der eine Python- Schnittstelle besitzt. Zudem kann der Simulator mit NetSquid kombiniert werden: Eine in SimulaQron geschriebene Anwendung kann über das CQC (Classic Quantum Combiner) Modul in einem Netsquid simulierten Netzwerk ausgeführt werden. Dadurch kann der Einfluss von Zeit z.B. in Form von Verzögerungen ereignisgesteuert berücksichtigt werden.

Installation

SimulaQron kann einfach mittels pip installiert werden:

```
pip3 install simulaqron
```

Um das Programm zum Laufen zu bringen, muss folgender Eintrag nach Installation in der Datei `simulaqron.py` in Zeile 234 im Ordner `simulaqron` geändert werden (Stand 04.07.2022):

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

Zu

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

SimulaQron wurde mit dem OS Ubuntu 20.04 getestet.

*Aufbau SimulaQron-Knoten –* Der Aufbau eines Netzwerkknotens ist in Abbildung 12 dargestellt:

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

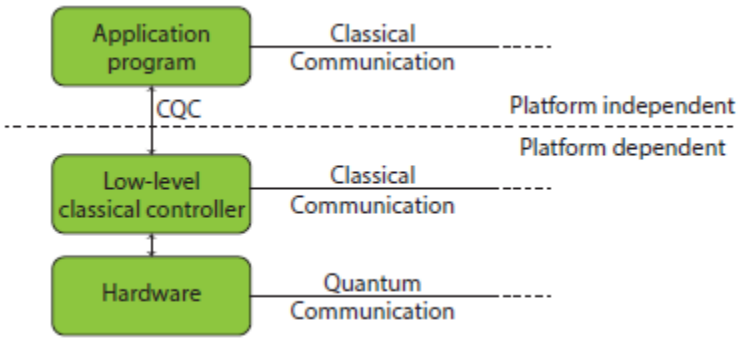



*Abbildung 12: Schematischer Aufbau eines Netzwerkknotens in SimulaQron*[12]

Die unterste Ebene ist die Quantenhardware bzw. -Prozessor, der QuBits erzeugen und messen kann. Zudem besitzt er optische Verbindungen zu anderen Knoten. Der Prozessor verfügt über einen plattformabhängigen Kontroller, der u.a. über klassische Verbindungen zu anderen Knoten verfügt.

Das CQC (Classic Quantum Combiner) Interface wird von dem plattformabhängigen Teil bereitgestellt und enthält Befehle z.B. zum Durchführen von Messungen und Anweisungen speziell für Quantennetze wie dem Erzeugen von Verschränkung und Übertragung von QuBits.

*Programmiermodi –* Für den Zugang zur Quantenhardware gibt es zwei unterschiedliche Wege:

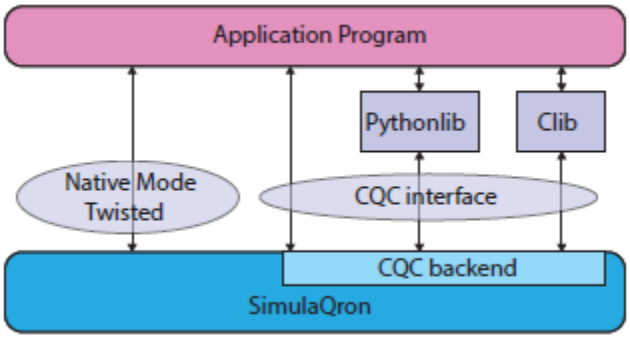



*Abbildung 13: Möglichkeiten der Programmierung von SimulaQron*[12]

1. *Native Mode*: In diesem Modus wird die Hardware direkt über die Bibliothek `Twisted` in Python angesteuert. Dieser Modus gewährt den vollen Zugriff auf die Quantenhardware, jedoch ist dieser Modus speziell für `Twisted` ausgelegt und wird daher in zukünftigen Quantennetzen kaum Verwendung finden[12].
2. *CQC Mode*: CQC ist ein Paketformat, über das Anweisungen an den Netzwerkknoten bzw. die Quantenhardware gesendet werden. Zur Vereinfachung der Programmierung über CQC werden eine C- und Python-Bibliothek bereitgestellt. Inzwischen ist auch eine RUST-Bibliothek verfügbar.

*CQC-Format –* Der CQC Befehlssatz umfasst eine Reihe unterschiedlicher Kommandos zur Steuerung des Quantenprozessors. Die folgende Gegenüberstellung zeigt den Befehl zum Anlegen eines QuBits in Python und den entsprechenden CQC-Befehl:

| Python Befehl | CQC Befehl |
|---|---|
| `//QuBit auf node1 anlegen`<br>`q=qubit(node1)` | `//Anfrage zum Allokieren eines QuBits`<br>`CMD_NEW` |

CQC kann auch als Standalone Python Modul installiert werden:

```
pip3 install cqc
```

## Anwendungsbeispiel

*Bell Paar Erzeugung –* Eine Bell Paar Erzeugung zwischen zwei Hosts (Alice, Bob) kann wie folgt -unter Nutzung von Python und CQC Interface- erzeugt werden:

1. Starte SimulaQron über folgenden Command Line Befehl:
   `simulaqron start`
   Dieser Befehl startet fünf Hosts, von denen zwei die Namen Alice bzw. Bob haben
2. Anschließend mit dem Befehl `cd` in folgenden Ordner navigieren (Ordner mit Beispielen muss vom GitHub Repository kopiert werden):
   `SimulaQron/examples/nativeMode/corrRNG/`
3. Danach müssen in den Dateien `bobTest.py` und `aliceTest.py` folgende Änderungen gemacht werden (Stand 04.07.2022):
4. `hostConfig` durch `host_config` ersetzen
5. `socketsConfig` durch `SocketsConfig` ersetzen
6. Anschließend Programm mit `./run.sh` (Datei liegt im examples Ordner) starten.

### *Ausgabe des Programms*

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

Wie nach der Messung des Bell Zustandes zu erwarten ist, ist der Wert bei Alice und Bob entweder 0 oder 1.

Quellcode `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)+"|")
```

Mit dem Befehl `CQCConnection("Alice")` wird ein Host bzw. Knoten mit dem Namen Alice angelegt und gleichzeitig eine CQC-Verbindung zum diesem aufgebaut. Mit dem Befehl `Alice.createEPR("Bob")` wird ein Bell-Paar zwischen den beiden Hosts `Alice` und `Bob` erzeugt.

## Fazit

SimulaQron ermöglicht plattformunabhängige Entwicklung von Anwendungen mittels CQC-Technik. Simulatoren für Quantenprozessoren (Backend) sind frei wählbar, solange sie eine Python Schnittstelle haben. Der Simulator ist auf die Entwicklung von Applikationen für die Anwendungsschicht ausgelegt. Zur Installation bzw. Start des Bell-Paar Beispiels war es nötig, den Code zu modifizieren (Stand 04.07.2022). Vermutlich wurde der Code oder die Dokumentation bisher noch nicht entsprechend angepasst. Ohne diese Anpassungen wäre die Evaluation von SimulaQron nicht möglich gewesen.

# NetSquid

| | |
|---|---|
| **Repository** | Keines vorhanden |
| **Sprache** | Python |
| **OS** | Plattformunabhängig: Python Interpreter erforderlich |
| **Typ** | Performanceuntersuchung der physikalischen Schicht |
| **Fokus** | Simulation eines quantenbasierten Internets |
| **Besonderheit** | Modularer Aufbau; Quanten Computing Library |
| **Lizenz** | Kostenlos, aber Registrierung erforderlich |
| **Website** | https://netsquid.org/ |
| **Registrierung** | Notwendig, kostenlos |

## Eigenschaften

NetSquid (NETwork Simulator for Quantum Information) ist ein ereignisorientierter Simulator, der sich zum Simulieren von Ereignissen in Quantennetzen und Quantencomputing Systemen von der physikalischen bis hin zur Anwendungsschicht eignet[13].

## Installation

Nach der erforderlichen Registrierung kann die NetSquid Python Bibliothek unter Angabe von Benutzernamen und Passwort mit folgendem Befehl heruntergeladen werden:

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

## Anwendungsbeispiel

Um den ereignisorientierten Charakter von Netsquid zu verdeutlichen, sollen zwei Hosts sich gegenseitig ein QuBit zuschicken, messen und wieder zurückschicken. Dafür besitzt Netsquid einen sog. *discrete event simulation engine*, welcher Ereignisse z.B. das Empfangen eines QuBits auf einer Zeitlinie anordnet und diese dann chronologisch abarbeitet:

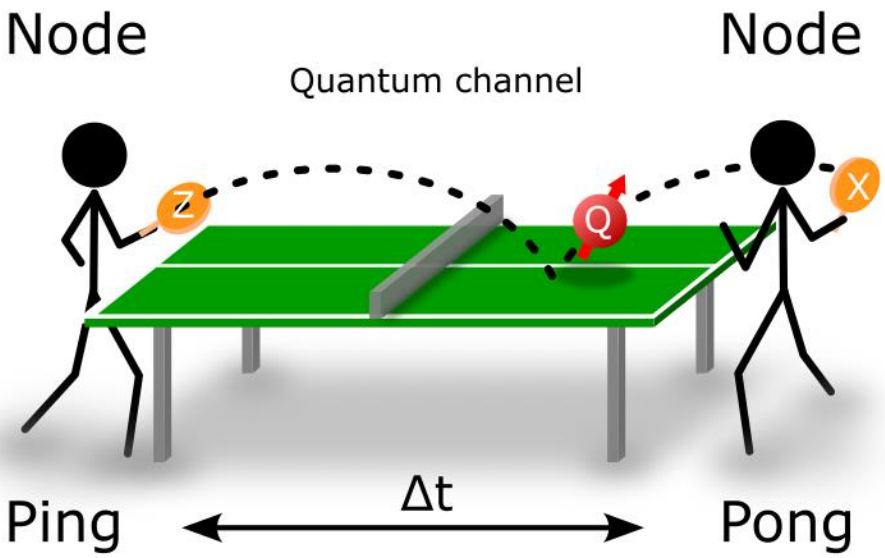



*Abbildung 14: Schematische Darstellung des Programms[14]*

[13] https://www.nature.com/articles/s42005-021-00647-8

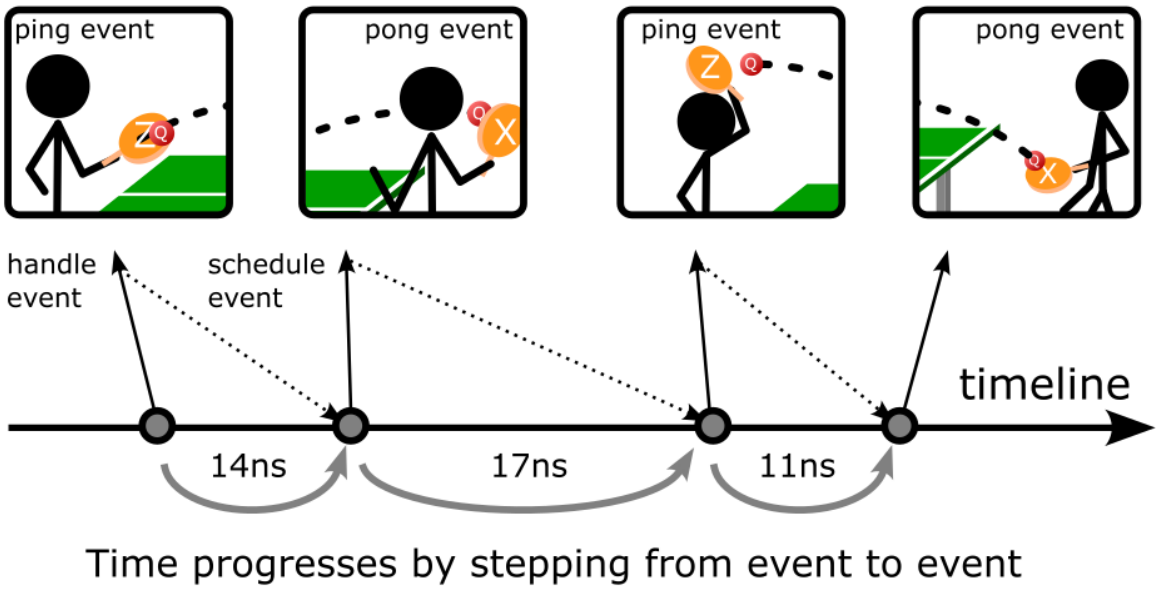



*Abbildung 15: schematische Darstellung und chronologischer Ablauf einer ereignisorientierten Simulation*[14]

*Netzwerkkomponenten –* Zum Aufbau des Netzwerks sind folgende Komponenten erforderlich:

- Zwei Hosts bzw. Knoten, welcher als Sender bzw. Empfänger dienen
- Zwei Einweg Quantenkanäle zur Übertragung der QuBits

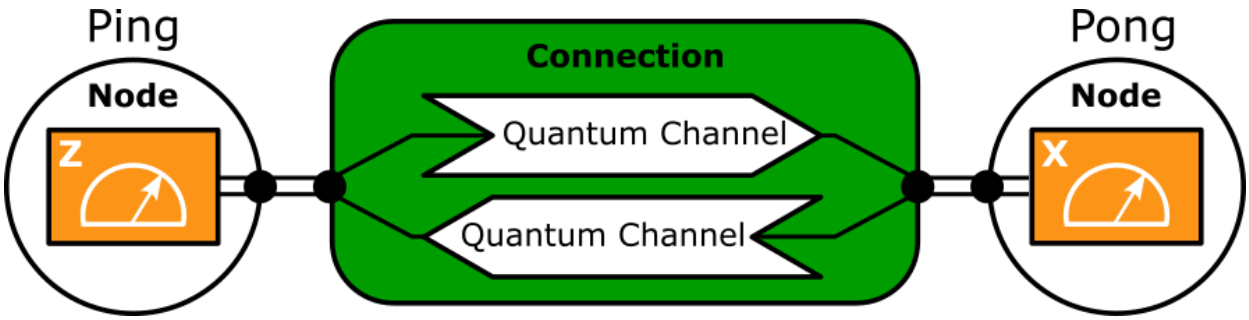



*Abbildung 16: Schematische Darstellung Netzwerktopologie*[14]

Die Buchstaben Z (`Node Ping`) und X (`Node Pong`) deuten an, in welcher Basis die QuBits gemessen werden. Folgend sind die wichtigsten Funktionen und Start der Simulation mit Kommentaren aufgeführt:

```
#Netzwerkknoten mit den Namen Ping und Pong erstellen

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

# Klasse mit Verzögerungsmodell für die Quantenkanäle anlegen

class PingPongDelayModel(DelayModel)

# Beide Knoten mit den Quantenkanälen verbinden

connection = DirectConnection(name="conn[ping|pong]",
channel_AtoB=channel_1,
channel_BtoA=channel_2)

# Protokoll zum Senden, Empfangen und Messen der QuBits definieren
```

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

```
class PingPongProtocol(NodeProtocol)

# Protokoll den Knoten zuweisen und festlegen, in welcher Basis
gemessen werden soll:

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

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

# Protokolle in beiden Knoten starten und Laufzeit der Simulation
in ns festlegen:

ping_protocol.start()
pong_protocol.start()

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

Ausgabe der 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
```

Der erste Eintrag gibt die Zeit an, die zwischen zwei Events vergangen ist. Diese Zeiten ergeben sich durch zufällig generierte Verzögerungen (`PingPongDelayModel`) in den Quantenkanälen. Erreicht ein QuBit den Ping bzw. Pong Knoten, wird das Ergebnis der Messung und dessen Wahrscheinlichkeit angezeigt. Bedingt durch die unterschiedlichen Basen (Z-Basis → Ping, X-Basis → Pong) in denen gemessen wird, ist das Ergebnis der Messung jedes Mal zufällig (Wahrscheinlichkeit 50%, gemessene Zustände: |0> oder |1>: Ping; |+> oder |-> Pong).

Fazit

NetSquid ist dafür geeignet, ereignisorientierte Prozesse und Abläufe in Quantennetzen zu simulieren. Im Vergleich zu SeQUeNCE- der ebenfalls ein ereignisdiskreter Simulator ist- kann NetSquid Prozesse von der physikalischen bis hin zur Anwendungsschicht simulieren. SeQUeNCe ist dagegen auf die Simulation auf Ereignisse in den unteren beiden Netzwerkschichten ausgelegt, dafür detaillierter als NetSquid. Gegenüber SimulaQron und QuNetSim hingegen ist es mit NetSquid aufwendiger Applikationen auf der Anwendungsschicht zu erstellen.

# QuNetSim

| | |
|---|---|
| **Repository** | https://github.com/tqsd/QuNetSim |
| **Sprache** | Python |
| **OS** | Cross-platform |
| **Typ** | framework for quantum networking simulations |
| **Fokus** | Netzwerk |
| **Besonderheit** | Optional eigenes Backend |
| **Lizenz** | MIT License |
| **Website** | https://tqsd.github.io/QuNetSim/ |
| **Registrierung** | nein |

QuNetSim ist ein in Python geschriebenes Paket, das frei erhältlich ist und sich zum schnellen und einfachen testen von Protokollen eignet. Es richtet sich hauptsächlich an Studierende und Lehrer, welche einen geeigneten Demonstrator suchen, um „high-level“ Protokolle zu erlernen und erklären zu können. Die Software simuliert die Netzwerkschicht in einem Quantennetz, ohne dass sich der Benutzer um das Routing zwischen zwei Hosts, die (in-) direkt durch die Netzwerktopologie verbunden sind, kümmern muss. Zudem verfügt der Simulator über Mechanismen zur Kontrolle der Synchronisation im Netz. In QuNetSim kann man auch ein klassisches und ein Quantennetz parallel in einer Simulation betreiben, wie es für manche Algorithmen notwendig ist.

*Die verschiedenen Backends* – Hervorzuheben ist das modulare Backend, welches standardmäßig mit SimulaQron arbeitet, jedoch auch mit anderen Backends wie ProjectQ and EQSN ausgetauscht werden kann. Wem ein eigenes Backend lieber ist, der kann seine eigene Library einbinden.

*Die Installation mit pip* – QuNetSim kann zwar über Sourcecode[15] manuell auf Windows und Linux installiert werden, allerdings ist es ratsam die Installation in einer Virtual Environment in Python durchzuführen. Eine Beschreibung zur Erzeugung einer virtuellen Umgebung ist im Abschnitt zur Installation von Qiskit zu finden. Wir führen die Installation mit `pip install qunetsim` durch.

*Templates* sind Skripte, welche ein Netzwerk abbilden, bzw. instanziieren. Um diese Netzwerke zu testen, muss lediglich das entsprechende Template ausgeführt werden. Möchte man neue Templates erstellen, wird das über `template` erreicht. Dies führt den Benutzer durch einen Wizzard, welcher abfragt, wie das neue Template heißen soll, wie viele Nodes erstellt werden sollen, welches Backend verwendet werden soll und welche Topologie die Hosts bilden sollen. Darunter: mesh, ring, star, linear und tree.

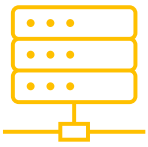
*Hello World* – Nach abgeschlossener Installation führen wir das Skript `template` aus. Wir finden nun ein neues `.py`-Script im aktuellen Pfad, welches sich ausführen lässt. Diese beinhaltet netterweise schon eine Art „Hello World“ Programm, das in unserem Fall 5 QuBits im Zustand $|1\rangle$ von Host A nach Host B versendet und dort gemessen wird.

[15] Klonen des Repository und Installation der Requirements mit pip

*Das QuBit-Objekt* in QuNetSim ist hauptsächlich ein Wrapper für das im Backend befindliche QuBit. Die Klasse befindet sich in `qunetsim.objects.qubit` und wird auch dementsprechend importiert. Mit `q = Qubit(host, qubit=None, q_id=None, blocked=False)` lässt sich ein QuBit instanziieren und einem Host zuweisen. Mit dem QuBit können nun einige Operationen durchgeführt werden. Darunter z.B.:

| | | | |
|---|---|---|---|
| H() | **Hadamard** | fidelity(other_qubit) | **Rückgabe der Fidelity** |
| X(),Y(),Z() | **Pauli** | measure() | **Messen** |
| cnot(target) | **CNOT** | send_to(receiver_id) | **Senden an Host** |
| density_operator() | **Rückgabe Dichtmatrix** | | |

*Der Host* ist analog zu einem Host oder einem Knoten in einem klassischen Netzwerk zu betrachten. Er kann Pakete routen, als Relay agieren, oder spezielle Protokolle befolgen. Die Klasse befindet sich in `qunetsim.components.host` und ein neuer Host wird z.B. mit `host_alice = Host('Alice')` erzeugt. Hosts werden außerdem mit anderen Hosts über Connections verbunden. Dafür wird `host_alice.add_connection('Bob')` und `host_bob.add_connection('Alice')` ausgeführt, was zwei bi-direktionale Verbindungen etabliert, eine klassische Verbindung und einen Quantenkanal.



*Das Netzwerk* ist eine zentrale Komponente in jeder Simulation. Dabei müssen die Netzwerke mit Hosts verknüpft sein, die ihre Verbindungen bereits definiert haben. Auf der Topologie aufbauend können nun für den klassischen und den Quantenkanal verschiedene Routingalgorithmen aufgesetzt werden. Als default wird die kürzeste Route verwendet.

*Beispiel* – Der Codeauszug unten ist ein Beispiel unter vielen, welche sich hier auf der Dokumentations-Webseite auffinden lassen. Da es sich hier um ein Beispiel handelt, welches hauptsächlich die Grundlagen aufgreift, werden wir an dieser Stelle den Codeauszug komplett besprechen. Einige Dinge wurden auch bereits in den vergangenen Abschnitten beschrieben.

Das folgende Beispiel (Abbildung 17) ist ein Netzwerk, in welchem jeder Teilnehmer Teil eines linearen Netzwerkes ist. Alice ist mit Bob verbunden, Bob mit Eve etc.; nun möchte Alice 10 QuBits an Dean übertragen und wartet nach jedem Senden auf eine Bestätigung von Dean, um sicher zu gehen, dass das QuBit auch bei Dean angekommen ist.

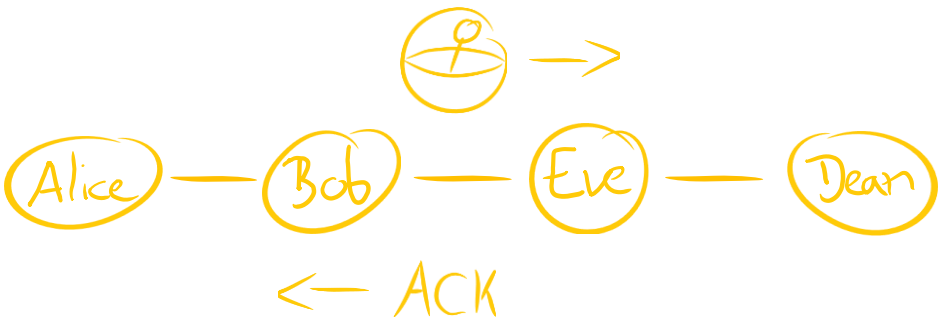



*Abbildung 17: Beispielnetzwerk QuNetSim*

Wir beginnen in Zeile `1-3` mit dem Import der benötigten Pakete `Host`, `Network` und `Qubit`. In der `main()` Funktion wird ein zunächst Netzwerk instanziiert (Zeile `5`), dann ein String-Array mit den vier Teilnehmern erzeugt (Zeile `6`) und schließlich mit der Methode `network.start(nodes)` gestartet

und mit einem `delay` von 0.1 beaufschlagt (Zeile 8). Folgend (Zeile 9–22) werden die Hosts angelegt, miteinander verknüpft und gestartet. Um die Netzwerkkonfiguration abzuschließen, werden die Hosts dem Netzwerk mit der `network.add_host()` Methode hinzugefügt.

```
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
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 Zeile 28 wird Host Alice ein Qubit erzeugen und diese mit dem X-Gatter (Zeile 30) in den angeregten Zustand |1⟩ überführen. Nun ist alles bereits um das QuBit mit `host_alice.send_qubit('Dean', q, await_ack=True)` aus Zeile 32 an Dean zu übertragen. Alice wird dann auf eine Bestätigung von Dean warten, bevor sie fortfährt. Da die Flag `await_ack` auf `true` gesetzt ist, gibt `send_qubit()` zwei Werte zurück: die Qubit-ID, die gesendet wurde, und einen booleschen Wert, der angibt, ob das ACK angekommen ist oder Alice die maximale Wartezeit überschritten hat.

Dean liest in Zeile 34 das QuBit aus und nimmt dann (Zeile 37) eine Messung vor, falls die Übertragung erfolgreich war und gibt einen String aus. Diese Prozedur wir insgesamt 10-mal (für 10 QuBits) ausgeführt, bevor das Netzwerk gestoppt wird.

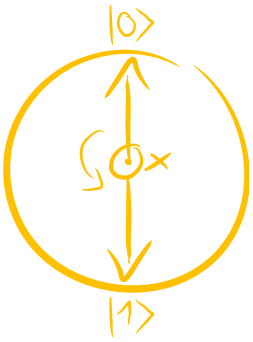



*Abbildung 18: Funktionsweise eines Pauli-X-Gatters*

*Erläuterung:* Das Pauli-X-Gatter ist neben dem Pauli-Y- und dem Pauli-Z-Gatter ein 1-QuBit-Gatter, welches das Eingangs-QuBit invertiert, sprich aus einem Zustand |0⟩ einen Zustand |1⟩ erzeugt und umgekehrt. Betrachtet man dieses Verhalten auf der sogenannten Bloch-Kugel, welche sozusagen der Standard zur Darstellung von 1-QuBit-Zuständen ist, so wird man feststellen, dass diese Zustandsänderung einer 180°-Drehung des Vektors um die x-Achse entspricht, siehe Abbildung 18. Dieses Verhalten entspricht dem Verhalten des klassischen NOT-Gatters. Gleichermaßen funktionieren die beiden anderen Gatter, nur jeweils um die entsprechende Achse.

*Fazit* – QuNetSim ist ein auf Python aufbauender Simulator und damit in einer gängigen und für Anfänger geeigneten Programmiersprache geschrieben. Er eignet sich sehr gut für Studierende und Lehrende, sowie Benutzer, welche einen Demonstrator benötigen oder auf der Suche nach einem geeigneten Einstieg in das Thema Quantentechnologie sind. Zusätzlich richtet sich dieses Framework an Interessierte, die bereits einen Einstieg in Netzwerke suchen und erwarten, dass eine Netzwerksystem bereits implementiert ist. Die bereitgestellte Dokumentation ist ausführlich und gut verständlich und beinhaltet viele Beispiele.

# SILQ

| | |
|---|---|
| **Repository** | https://github.com/eth-sri/silq |
| **Sprache** | Q#, D, Tex, Python |
| **OS** | Es wird VS Code benötigt um das Silq PlugIn zu installieren |
| **Typ** | Quanten-Schaltungen |
| **Fokus** | More intuitive semantics |
| **Besonderheit** | Uncomputation/Rücksetzung von QuBits auf den Ausgangszustand |
| **Lizenz** | FreeBSD Lizenz |
| **Website** | https://silq.ethz.ch/ |
| **Registrierung** | Keine Beschränkungen: Download und Installation ohne Registrierung |


## Eigenschaften

Silq ist ein Simulator, der bei Simulationen das sog. Uncomputing von QuBits berücksichtigt[16]: Dies ermöglicht die automatische Rücksetzung temporär benötigter QuBits -sog. Ancilla QuBits- die dann für weitere Operationen zur Verfügung stehen. Der Simulator benutzt eine Syntax, die speziell auf die Programmierung mit QuBits ausgelegt ist und verfügt über eigene Variablentypen für „klassische“ und solche Zustände, die zur Speicherung von Quantenzuständen (Superposition) dienen. Silq ist an die Sichtweise von Anwendungsentwicklern angelehnt und nicht auf das Erstellen von Algorithmen für einen spezifischen Typ von Quantenprozessor. Matrixoperationen und Tensor Produkte werden dafür nicht benötigt: Silq nutzt stattdessen passende Variablentypen und dafür zulässige Operatoren, was den Umfang des Codes zum Erstellen von Algorithmen (z.B. Groover Algorithmus) wesentlich verkürzt. Andere Simulatoren wie Cirq, Qiskit etc… basieren meist auf Python oder Matlab und sind eher für den Gebrauch auf „klassischen Rechnern“ ausgelegt. Silq hingegen ist speziell auf die Abstraktion von Low Level Qubit Operationen ausgerichtet.

## Installation

Der einfachste Weg, Silq zu installieren, ist Silq als Plugin in VS Code (Visual Studio Code) hinzuzufügen. Für das folgende Anwendungsbeispiel wurde das Silq Plugin in VS Code unter Windows 10 installiert.

## Anwendungsbeispiel

Als einfaches Beispiel soll ein Bell-Paar-inklusive Messung- in Silq erzeugt werden:

```
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);
}
```

Die Funktion liefert als Ausgabe mit einer Wahrscheinlichkeit von 50% entweder (0,0) oder (1,1). Die Formel für das Bell-Paar lautet:

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

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

Im Gegensatz dazu orientieren sich Sprachen wie Qiskit am Konzept von Quantengattern bzw. Statusvektoren zum Aufbau einer Schaltung für die Quantenverschränkung:

```
qc = QuantumCircuit(2)

# Apply H-gate to the first:

qc.h(0)

# Apply a CNOT:

qc.cx(0,1)
qobj = assemble(qc)
result = svsim.run(qobj).result()
plot_histogram(result.get_counts()
```

Wie aus dem Vergleich der beiden Quellcodes hervorgeht, benötigt Qiskit ein H und CNOT Gatter, während Silq nur ein H Gatter benötigt und CNOT mit einer if-else Anweisung durchführt (Codezeilen CNOT blau markiert).

*Uncomputation –* Silq unterscheidet zwischen konsumierten und nicht konsumierten Variablen, wodurch das Konzept der Uncomputation umgesetzt wird. Um dieses Konzept zu erläutern, werden drei Variablen durch eine UND Verknüpfung miteinander verbunden:

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

Auf einem klassischen Computer werden Ergebnisse einer solchen Operation in temporären Variablen gespeichert z.B. das Ergebnis von x&&y.

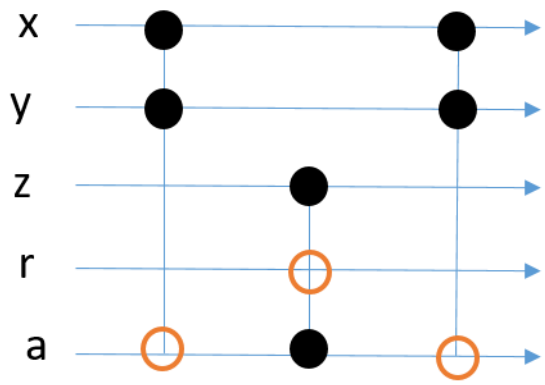



*Abbildung 19: QuBit-Gatter Uncomputation*

Als erstes wird das Ergebnis der Operation x&&y in der Variablen a gespeichert. Danach wird das Ergebnis von a&&z in r gespeichert. r und a befinden sich am Anfang im Ausgangszustand 0. Soll das vierte QuBit für weitere Operationen verwendet werden, muss es wieder den Zustand 0 annehmen. Dies wird durch sog. Toffoli-Gatter (orangene Kreise in Abbildung 19) gelöst. Solche Gatter invertieren ein QuBit, wenn beide Eingänge den Zustand 1 haben. Damit wird der Zustand von QuBit a durch das dritte Gatter wieder auf 0 gesetzt, weil beide Eingänge (x&&y) wahr (=1) sind. In Falle von Quantencomputing ist die Uncomputation von Variablen sehr wichtig, da nur begrenzt QuBits zur Verfügung stehen.

*Konsumierbare und nicht konsumierbare Parameter –* Silq setzt das Uncomputing in Form konsumierbarer und nicht konsumierbarer Parameter um. Konsumierte Parameter sind solche, die von einer Funktion verwertet werden z.B. durch eine Messung. Der quantenmechanische Zustand nach einer Funktion hängt somit nicht mehr von dieser Variablen ab, wohl aber das Ergebnis der Funktion.

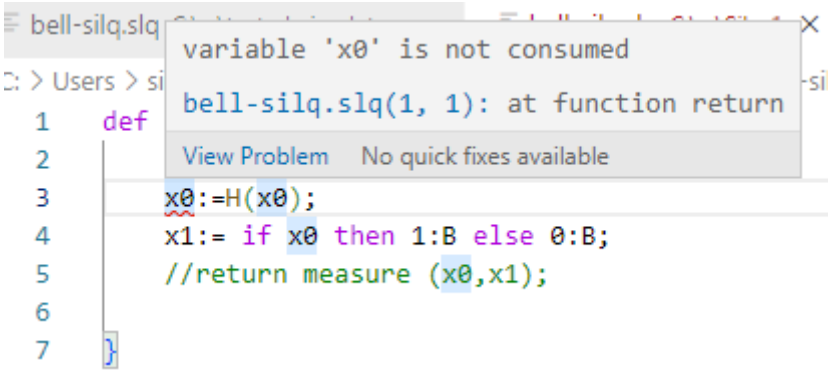



*Abbildung 20: Fehler bei nicht konsumierter Variable*

Abbildung 20 zeigt die Fehlerausgabe, falls eine Variable nicht von der Funktion konsumiert wird. Im Beispiel wurde die `return` Anweisung mit der Messung der Variablen auskommentiert (vgl. Code zur Erzeugung des Bell-Zustandes in Silq).

Um Parameter zu definieren, die von einer Funktion nicht konsumiert werden, stehen in Silq folgende Annotationen zur Verfügung: `const`, `lifted` und `qfree`. Beispiel `qfree`: Derart annotierte Funktionsparameter oder Ausdrücke verändern oder zerstören keine Superpositionen d.h. qfree Variablen werden nach Benutzung automatisch uncomputet und stehen für eine weitere Benutzung zur Verfügung:

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

Nicht konsumierte Variablen können keine Superpositionen speichern, sondern nur stabile Zustände wie |0> oder |1>.

Fazit

Silq ermöglicht durch seine einzigartige Syntax, Prozesse wie Verschränkung, Teleportation weitgehend ohne Statusvektoren und Matrizen zu simulieren. Der Simulator ist plattformunabhängig, d.h. nicht auf einen bestimmten Quantenprozessor oder Computer ausgelegt. Quantenalgorithmen wie der Groover Algorithmus lassen sich in Silq mit deutlich weniger Codezeilen umsetzen als z.B. in Qiskit (vgl.[17,18]).

Durch die verwendete Syntax ist es möglich, viele Operationen mit nur wenigen Codezeilen auszuführen, jedoch erhöht sich dadurch die Komplexität. Simulationen, die auf Basis von Gattern und Matrizen arbeiten sind hingegen leichter nachvollziehbar als Silq-Simulationen.

[17] Groover Algorithmus in Silq: https://silq.ethz.ch/overview
[18] Groover Algorithmus in Qiskit: https://qiskit.org/textbook/ch-algorithms/grover.html

# Google Quantum AI (Cirq)

| | |
|---|---|
| **Repository** | https://github.com/quantumlib/cirq |
| **Sprache** | Python |
| **OS** | Plattformunabhängig: benötigt Python |
| **Typ** | Quantenschaltungen |
| **Fokus** | Testen von Algorithmen (auf Quantenbasis) |
| **Besonderheit** | Ökosystem |
| **Lizenz** | Apache License 2.0 |
| **Website** | https://quantumai.google/cirq |
| **Registrierung** | Nicht notwendig |

## Beschreibung

Cirq ist eine Python-Softwarebibliothek zum Schreiben, Manipulieren und Optimieren von Quantenschaltungen, die dann auf Quantencomputern und Quantensimulatoren ausgeführt werden können. Cirq bietet nützliche Abstraktionen für den Umgang mit den heutigen verrauschten Quantencomputern.

## Installation

Für die Installation unter Windows, Linux oder Mac OS X wird Python in Version ≥ 3.7.0 vorausgesetzt, sowie die aktuellste Version des Paketmanagers pip.

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

Zum Testen ob die Installation erfolgreich war eignet sich folgender Befehl:

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

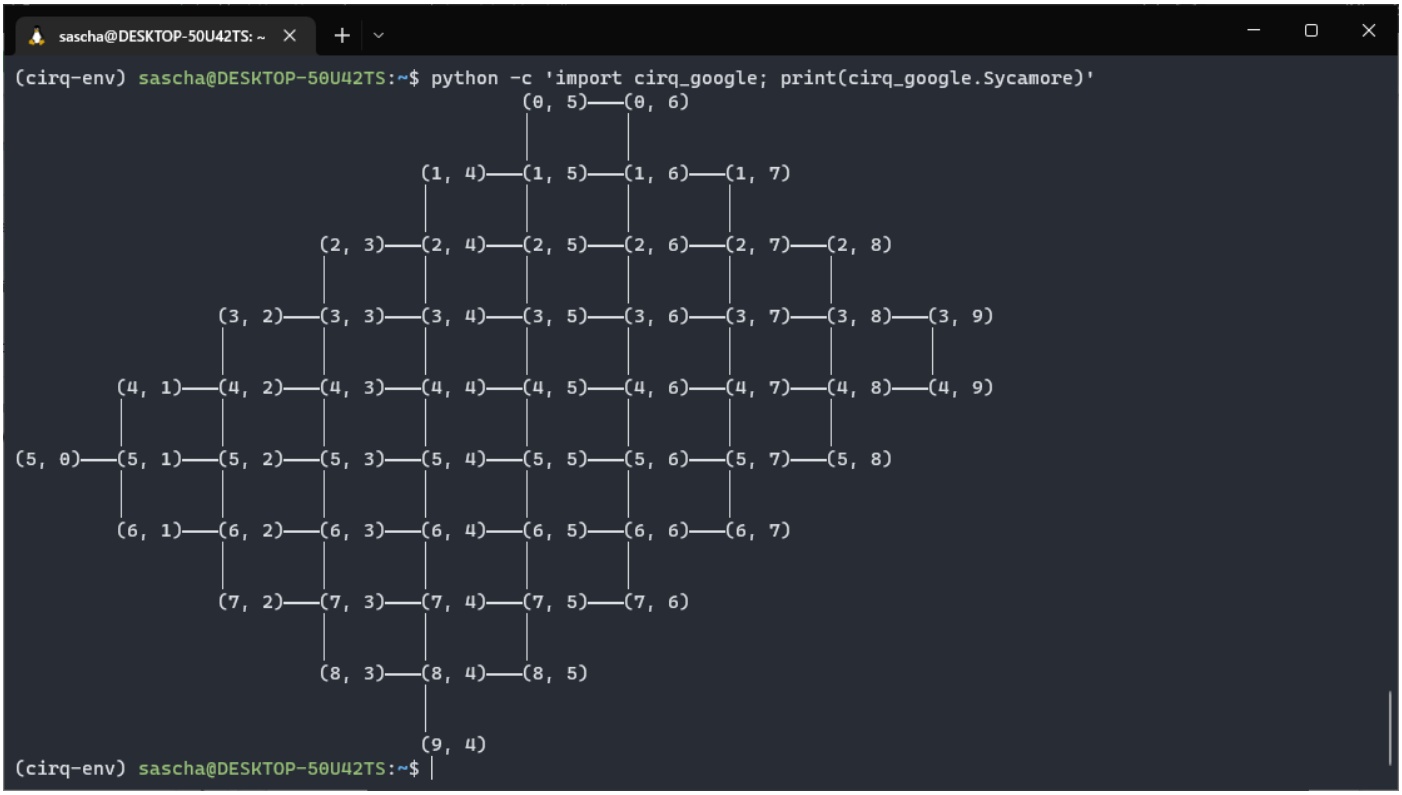



*Abbildung 21: Sycamore Prozessor mit 54 Qubits aus 2019.*

Registrierung

Zur Nutzung des Frameworks ist keine Registrierung notwendig. Wenn man jedoch eine lokale Installation scheut, kann man ebenfalls Googles Service namens Colab nutzen, was jedoch eine Anmeldung in einem Google-Account voraussetzt. Dieser Service ist im Grunde ein Jupyter Notebook Klon, welcher in Google Drive eingebettet ist. Colab ist kompatibel mit Jupyter und erlaubt ebenfalls das Öffnen und Exportieren von `.ipynb`-Dateien.

Beispiel

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


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

# Create a circuit
circuit = 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
```

Fazit

Google bietet mit Cirq und Quantum AI (wie auch die anderen Big-Player IBM und Amazon) nicht nur ein Framework für die lokale Installation, sondern eine ganze Plattform für die Entwicklung und Ausführung von Algorithmen. Genauso wie IBM, ist auch Google führend bei der Entwicklung eigener Hardware und bietet diese dem Benutzer auch an, bzw. ermöglicht Fernzugriff auf weltweit verfügbare Quantenprozessoren und -simulatoren, darunter AQT, Azure, IonQ, Pasqal und Rigetti. Forscher mit genehmigten Projekten können Aufträge auf der umfassenden Google-Infrastruktur ausführen.

# SQUANCH

| | |
|---|---|
| **Repository** | https://github.com/att-innovate/squanch |
| **Sprache** | Python |
| **OS** | Plattformunabhängig: Python Interpreter erforderlich |
| **Typ** | Simulation von Multiparty Netzwerken |
| **Fokus** | Simulation von Quantennetzwerken |
| **Besonderheit** | enthält klassische und Quanten Fehlermodelle |
| **Lizenz** | MIT Lizenz |
| **Website** | https://att-innovate.github.io/squanch/index.html |
| **Registrierung** | Keine Beschränkung |

## Beschreibung

SQUANCH ist ebenfalls ein quelloffenes Python-Framework zur Erstellung parallelisierter und verteilter Simulationen von Quanteninformationen. Obwohl SQUANCH als universelle Simulationsbibliothek für Quantencomputer verwendet werden kann, wurde es speziell für die Simulation von Quantennetzwerken entwickelt. Es sollen Ideen für Quantenübertragungs- und Netzwerkprotokolle getestet werden können. Das Paket enthält mehrere Module, darunter erweiterbare Quanten- und klassische Fehlermodelle, sowie ein Multithreading-Framework zur performanten Manipulation von Quanteninformationen.

## Installation

Wie bei allen anderen Simulatoren wird die Installation in einer virtuellen Umgebung (Anaconda, VENV) empfohlen. Unter Umständen sollte ebenfalls eine Python „Distribution“ gewählt werden, welche bereits einige wissenschaftliche Pakete wie z.B. matplotlib enthält, um unkompliziert alle Funktionen nutzen zu können.

```
pip install squanch
```

## Aufbau und Module

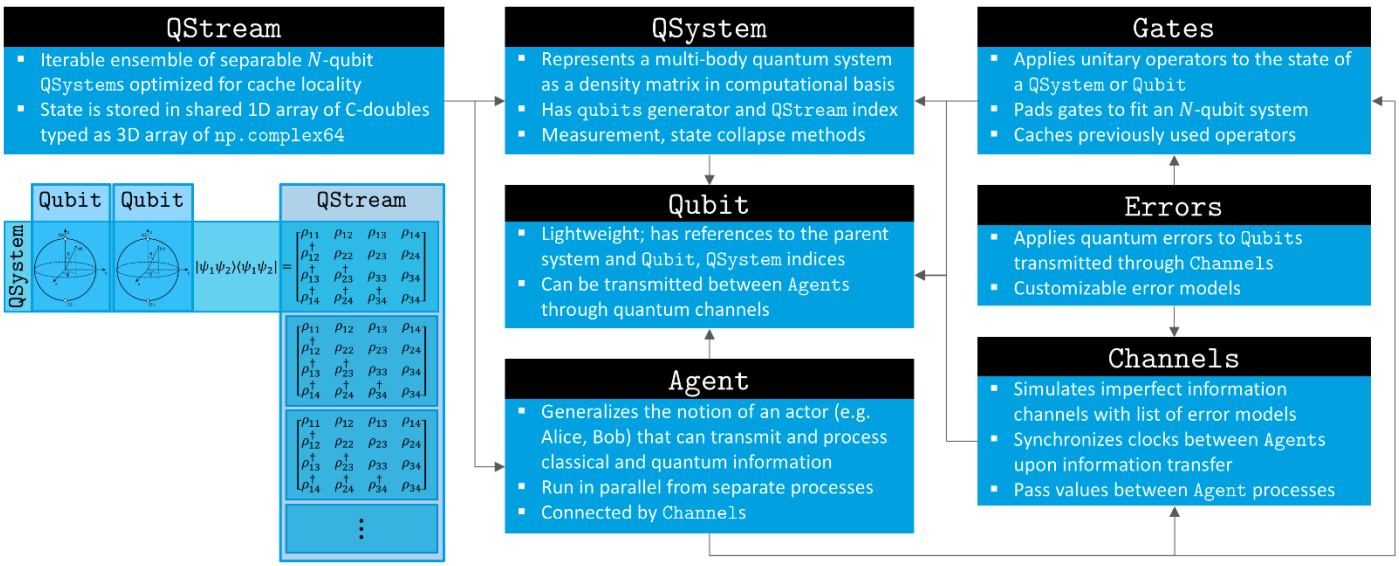



*Abbildung 22: Quelle https://att-innovate.github.io/squanch/overview.html#information-representation-and-processing*

Oben ist ein schematischer Überblick über die im SQUANCH-Framework verfügbaren Module. Das QSystem ist die grundlegendste Klasse und repräsentiert einen Mehrteilchen-Quantenzustand und wird als Dichtematrix dargestellt. Ensembles von Quantensystemen werden effizient durch QStreams behandelt, und jedes QSystem hat Verweise auf die ihm zugehörigen Qubits. Funktionen im Modul Gates können

verwendet werden, um den Zustand eines Quantensystems zu manipulieren. Agenten sind verallgemeinerte quantenmechanische "Akteure", die aus einer QStream-Instanz initialisiert werden und den Zustand der Quantensysteme in ihrem Stream-Objekt verändern können, typischerweise durch direkte Interaktion mit Qubits. Agenten laufen parallel in separaten Prozessen und sind durch Quanten- und klassische Kanäle, die anpassbare Fehlermodelle auf die übertragenen Informationen anwenden und die Uhren der Agenten synchronisieren, verbunden.[19]

Beispiel

Das Beispiel soll zeigen, wie ein QStream inklusive QSystem, welches die QuBits beinhaltet, im Kontext einer Kommunikation zwischen Alice und Bob funktioniert. Nach dem Erzeugen eines Streams mit zwei QuBits innerhalb eines QSystems, wird eines dieser beiden QuBits durch das Hadamard-Gatter modifiziert. Die beiden Kommunikationspartner sind Kinder der Agenten-Klasse und sind in diesem Fall für das Senden, Empfangen und Messen zuständig. Beide Partner teilen sich einen Output, über welchen Bob das Ergebnis seiner Messung mitteilt. Wichtig ist hier, dass sich die Logik durch das Überschreiben der jeweiligen `run()`-Funktion innerhalb der Agenten befindet.

```
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"])
```

[19] arXiv:1808.07047v1 [quant-ph] 21 Aug 2018 https://arxiv.org/pdf/1808.07047.pdf


*Abbildung 23: Jupyter-Notebook IDE*

## Fazit

SQUANCH ist eine universelle Simulationsbibliothek für Quantencomputer mit dem Fokus auf der Abbildung von Quantennetzwerken. Die Benutzer haben aufgrund des modularen Aufbaus verschiedene Möglichkeiten Algorithmen und Protokolle umzusetzen. Die Entwickler selbst bieten in ihrem Repository viele technische Hintergründe und Beispiele zu Quantum Teleportation, Superdense Coding, Man-In-The-Middle Attack and Quantum Error Correction an.

# SeQUeNCe

| | |
|---|---|
| **Repository** | https://github.com/sequence-toolbox/SeQUeNCe/ |
| **Sprache** | C++, Python, Makefile |
| **OS** | Plattformunabhängig: Python Interpreter erforderlich |
| **Typ** | Protokollen, Netzwerkparametern und Topologien |
| **Fokus** | Effekte in Quantennetzen auf den unteren Netzwerkschichten |
| **Besonderheit** | Zwischenspeicherung von Quantenzuständen |
| **Lizenz** | Open Source Lizenz |
| **Website** | https://sequence-toolbox.github.io/index.html |
| **Registrierung** | Keine Beschränkungen |

## Eigenschaften

SeQUeNCe ist ein ereignisorientierter, in Python geschriebener, frei erhältlicher Simulator für die Bereiche (Quanten-) Hardware, Management für Verschränkung, Ressourcen, Netzwerke und Anwendungen[20]. Der Simulator eignet sich vor allem für die Simulation von Ereignissen in den unteren Netzwerkschichten (Hardware-, Verbindungsschicht): alleine zur Erzeugung eines Quanten-Überlagerungszustandes sind drei Hardwarekomponenten erforderlich (Quantenspeicher, Quantenkanal und Detektor).

## Installation

Zur Installation des Simulators ist Python 3.7 oder höher erforderlich. SeQUeNCe kann dann aus dem GitHub Repository heraus durch folgende Kommandos installiert werden:

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

## Anwendungsbeispiel

Als Anwendung wird die Erzeugung und Messung eines Bell-Paares in SeQUeNCe gezeigt:

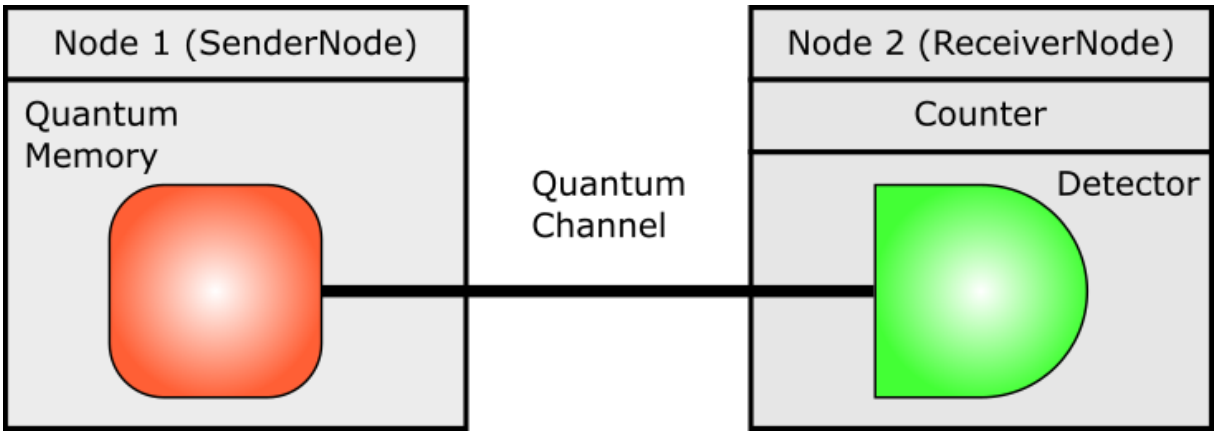



*Abbildung 24: Hardwareaufbau zur Erzeugung der Verschränkung[21]*

Der Quantenspeicher in Abbildung 24 besteht aus einem Atom, welches sich in diesem Beispiel im Überlagerungszustand befindet:

[20] Sequence Paper: https://doi.org/10.48550/arXiv.2009.12000
[21] https://sequence-toolbox.github.io/tutorial/chapter2/hardware.html

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

Der Detektor dient zur Messung des Zustandes. Die Wahrscheinlichkeit den Zustand |0> oder |1> zu messen liegt bei 50%.

Zuerst müssen die beiden Knoten (Quantenspeicher, Detektor) angelegt werden:

```
# class for Quantum memory

class SenderNode(Node):
self.memory = Memory('node1.memory', tl, fidelity=0, frequency=0,
efficiency=1, coherence_time=0, wavelength=500)

# class Quantum receiver

class ReceiverNode(Node):
self.detector = Detector('node2.detector', tl, efficiency=1)
```

Bei dem Quantenspeicher ist die Angabe von weiteren Parametern erforderlich wie z.B. die Kohärenzzeit und Güte der Verschränkung.

Das Protokoll legt das Verhalten des Counters bei Eintreffen eines Photons fest. Bei Detektion eines Photons wird der Zähler des Counters um 1 erhöht:

```
class Counter():

    def __init__(self):
        self.count = 0

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

Folgend werden die Simulationszeit in Picosekunden und die beiden Knoten definiert:

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

Um den Quantenspeicher und Detektor zu verbinden, ist die Definition eines Quantenkanals- Dämpfung=0, Länge=1km- erforderlich:

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

Der Status des Quantenspeichers wird jetzt in den Überlagerungszustand versetzt:

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

Der Quantenspeicher muss jetzt angeregt und dieses Ereignis dem Eventhandler übergeben werden:

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

Nach Übergabe an den Eventhandler können beide Nodes gestartet werden:

```
tl.init()
tl.run()
```

Als Ausgabe des Programms erhält man die Anzahl detektierter Photonen sowie die Simulationsdauer:

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

Fazit

SeQUeNCe ist wie Netsquid ein ereignisorientierter Simulator, der jedoch mehr auf die Simulation von physikalischen Ereignissen ausgerichtet ist. Programme in diesem Simulator sind deswegen komplexer, bilden aber Prozesse und Ereignisse realitätsnäher ab. Benutzer können über eine Vielzahl von Parametern wie zum Beispiel Güte der Verschränkung, Kohärenzzeit etc… die Simulationen individuell gestalten. Im Vergleich zu anderen Simulatoren ist SeQUeNCe komplexer und erfordert mehr Einarbeitungszeit.

# QKDNetSim

| | |
|---|---|
| **Repository** | https://github.com/QKDNetSim/qkdnetsim-dev |
| **Sprache** | C/C++, Python, Perl |
| **OS** | Linux |
| **Typ** | QKD-Simulationsmodul |
| **Fokus** | QKD-Basis im Overlay oder TCP/IP Modus |
| **Besonderheit** | Basiert auf dem NS-3 Network Simulator |
| **Lizenz** | GPL 2.0 Lizenz |
| **Website** | https://www.qkdnetsim.info/documentation/ |
| **Registrierung** | Keine Beschränkungen |

## Eigenschaften

QKDNetSim[22] ist ein in C++ geschriebenes Modul für den Netzwerksimulator NS-3 und daher kein Standalone Simulator. Mit diesem Erweiterungsmodul ist es in NS-3 möglich, Netze mit QKD (Quantum Key Distribution) in zwei Betriebsmodi zu simulieren: `Overlay` oder `single` TCP/IP Modus.

## Installation

*Hinweis –* Die aktuellste Ubuntu Version, unter der QKDNetSim läuft, ist Ubuntu 18.04. Diese Version wurde auch für den Test des Simulators verwendet. Zur Installation sind im Terminal folgende Kommandos auszuführen:

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install gcc g++ python python-dev mercurial bzr 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
```

Danach muss der QKDNetSim Code heruntergeladen und kompiliert werden:

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

## Anwendungsbeispiel

Als Beispiel soll ein QKD-Kanal zur Übertragung eines Schlüssels zwischen zwei Hosts gezeigt werden. Das Beispiel ist bereits in den QKDNetSim Beispielen enthalten und befindet sich im Git Repository Ordner `qkdnetsim-dev/scratch`. Aus dem Ordner `qkdnetsim-dev` heraus lässt sich das Programm starten mit:

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

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

Ausgabe des Programms:

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

In der Ausgabe werden die IP Adressen der Hosts sowie Anzahl und Größe der übermittelten Pakete angezeigt. Neben der Ausgabe auf der Kommandozeile werden vom Programm auch Diagramme erzeugt, die Daten wie die Übertragungsrate oder die Puffergrößen der Schlüssel auf den einzelnen Hosts zeigen:

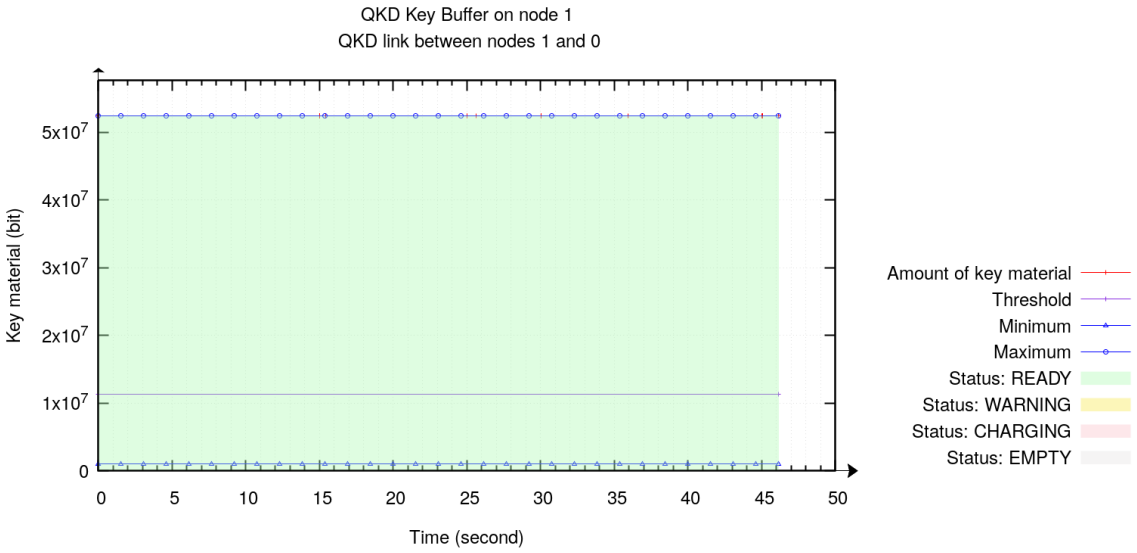



*Abbildung 25: QKD-Puffer auf Host 1*

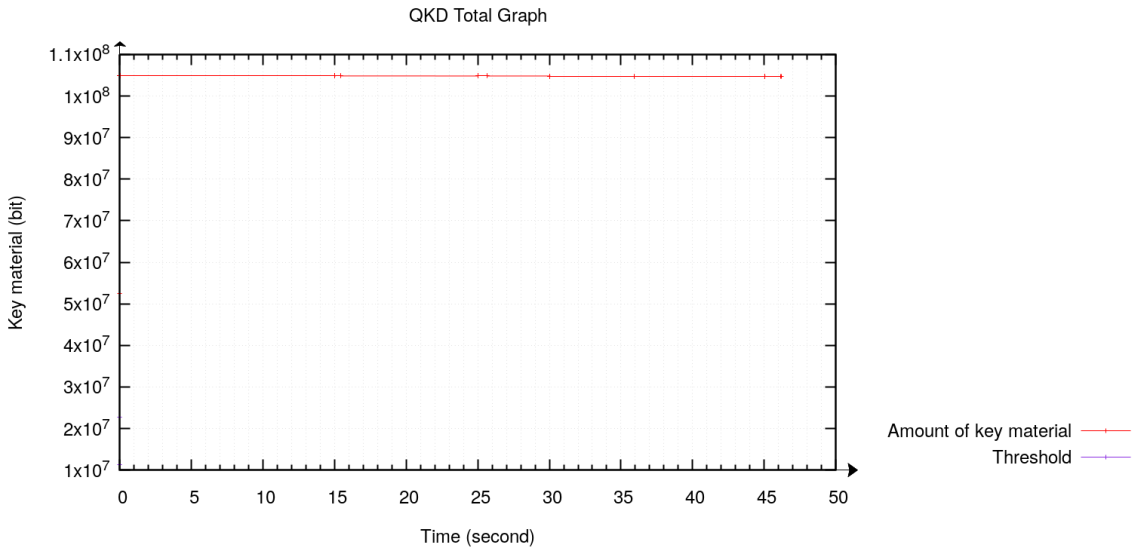



*Abbildung 26: Schlüsselgröße in Bit in Abhängigkeit der Zeit*

## Fazit

Der Simulator ist vor allem für Benutzer geeignet, die bereits Erfahrung mit dem NS-Netzwerksimulator haben und über gute C++ Sprachkenntnisse verfügen. Für Simulationen lassen sich mittels `gnuplot` aus den vom Programm erzeugten `*.plt` Dateien z.B. Grafiken im `*.png`- Format erzeugen, die bei Bedarf individuell angepasst werden können. Bisher (Stand 14.07.2022) kann QKDNetSim nur unter Ubuntu Distributionen installiert werden.

# QKDSimulator

QKDSimulator

| | |
|---|---|
| **Repository** | Nicht öffentlich (Kontakt zu Developer) |
| **Sprache** | Python |
| **OS** | Unbekannt/Webbrowser |
| **Typ** | Reiner QKD Simulator |
| **Fokus** | Voller QKD Stack bzw. Einzelsimulation |
| **Besonderheit** | Sehr ausführliche Ergebnisse |
| **Lizenz** | Nicht öffentlich (Kontakt zu Developer) |
| **Website** | https://www.qkdsimulator.com/ |
| **Registrierung** | Nicht notwendig |

## Beschreibung

QKD-Simulator ist eine Webanwendung zur Simulation und Analyse von Quantenschlüsselverteilungs-protokollen. Der Simulator stützt sich auf ein QKD-Simulations-Toolkit, das es ermöglicht, eine breite Palette von Parametern für einzelne Komponenten und Teilprotokolle im System anzupassen, z. B. Quantenkanal, Sifting, Fehlerabschätzung, Abgleich/Fehlerkorrektur, Datenschutzverstärkung. Jede Simulation liefert detaillierte Informationen über die Zwischen- und Endstufen des Protokolls.[23]

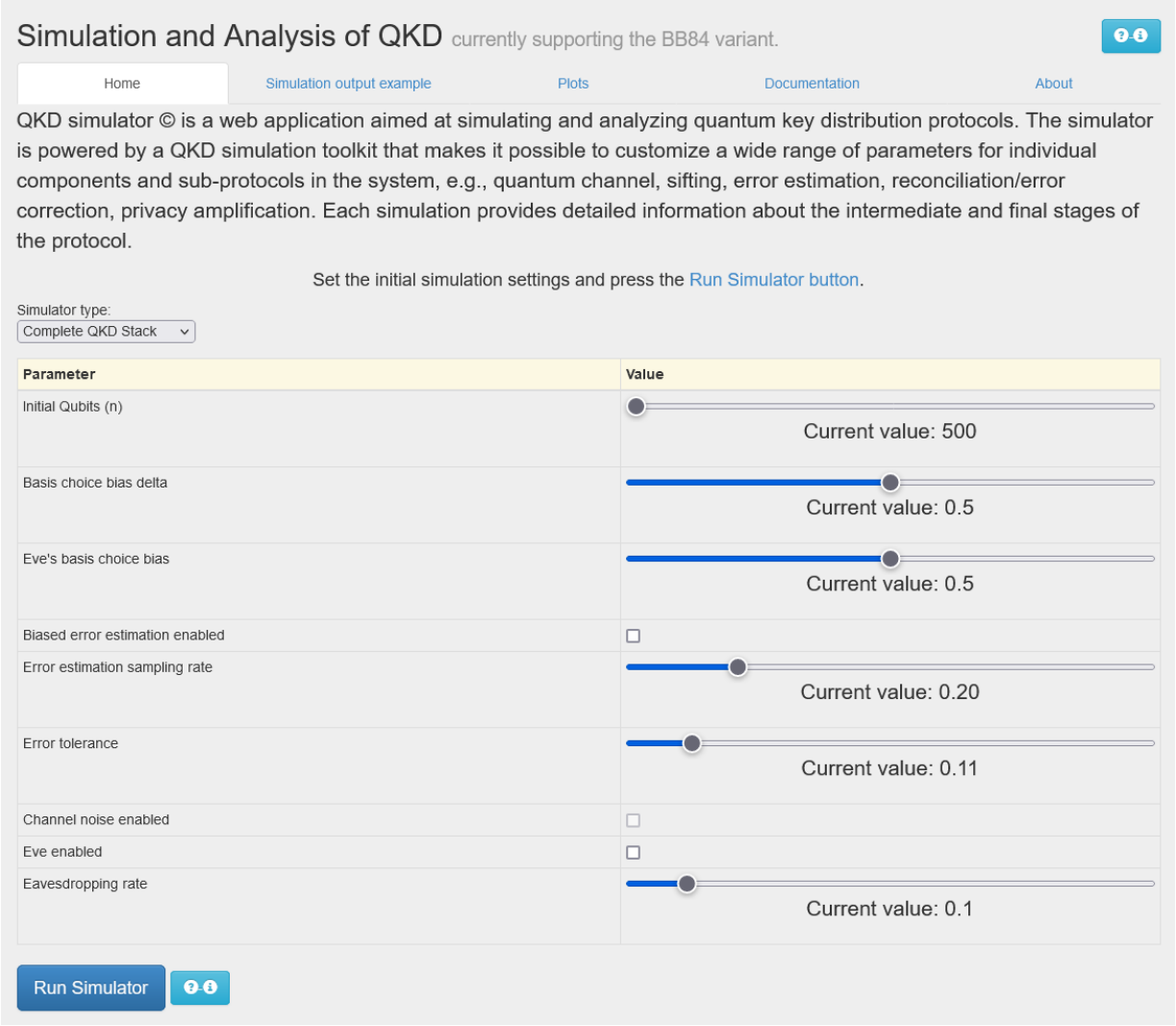



*Abbildung 27: Weboberfläche des QKDSimulators*

[23] https://www.qkdsimulator.com/

## Installation

Da es sich um eine Webanwendung handelt, ist keine Installation notwendig. Im Hintergrund ist jedoch ein Simulationsengine aktiv, der den QKDSimulator betreibt und Teil eines Toolkits für Quantenschlüssel-verteilung ist.

## Implementation

Die Implementierung des Backend-QKD-Toolkits, das diese Website betreibt, ist so konzipiert, dass sie wiederverwendbar ist, indem sie auf einem komponentenbasierten und vollständig modularen Ansatz beruht, so dass jedes Teilprotokoll auf isolierte Weise aktualisiert oder ersetzt werden kann. Die aktuelle Version bietet eine Implementierung des gesamten QKD-Stacks, d. h. Quantenkanal/Übertragung, Schlüsselsichtung, Authentifizierung mit universellem Hashing, Fehlerabschätzung, Abgleich/Fehlerkorrektur und Datenschutzverstärkung. Der Quantenkanal unterstützt derzeit nur das BB84-Protokoll. Das QKD-Simulations-Toolkit ist vollständig in Python implementiert und nutzt wissenschaftliche Standardbibliotheken wie Scipy, Numpy, Matplotlib, Quantum Information Toolkit (QIT) und PyCrypto[24] [Angaben der Developer].

[24] https://www.qkdsimulator.com/about

# Simulation and Analysis of QKD currently supporting the BB84 variant.

Home | Simulation output example | Plots | Documentation | 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> and |1>, and +45 degrees and -45 degrees shifted with the states |+> and |->, 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 averate, 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 paramter 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 |

*Abbildung 28: Beispiel eines Ergebnisses*


*Abbildung 29: Aufwendige und ausführliche Plots zu jedem Ergebnis*

Fazit

Der QKDSimulator besticht vor allem diejenigen der Benutzer, die sich hauptsächlich für QKD interessieren und diesen Algorithmus nachvollziehen wollen, ohne irgendein Programm zu installieren, sich registrieren oder selbst programmieren möchten. Die Dokumentation ist noch nicht vorhanden, aber die Ergebnisse sind ausführlich in Textpassagen beschrieben. Auch ist jeder Zwischenschritt ausführlich erklärt und die Plots haben eine hohe Qualität. Die Web-Application hat nur wenige Seiten, was vor und Nachteil zu gleich ist. Aktuell wird nur die BB84 Variante unterstützt, die die ursprünglichste und unmodifizierte Methode der Quantum-Key-Distribution darstellt.

# Amazon (Braket)


| | |
|---|---|
| **Repository** | https://github.com/aws/amazon-braket-sdk-python |
| **Sprache** | Python |
| **OS** | Python Interpreter erforderlich |
| **Typ** | Plattform |
| **Fokus** | AWS-Integration |
| **Besonderheit** | Prozessoren von D:Wave, IonQ, Rigetti, OQC |
| **Lizenz** | Apache 2.0 Lizenz |
| **Website** | https://aws.amazon.com/de/braket/ |
| **Registrierung** | Für das SDK nicht, für die Platform ja (Kreditkarte) |

## Beschreibung

Amazon Braket ist ein vollständig verwalteter AWS-Service, der Forschern, Wissenschaftlern und Entwicklern den Einstieg in das Quantencomputing erleichtern soll. Algorithmen können im Webinterface entworfen, getestet und auf verschiedenen Quantenschaltungssimulatoren und echter Quantenhardware ausgeführt werden. Darüber hinaus ist ebenfalls das Designen von hybriden Algorithmen möglich, welche klassische Ressourcen mit einbeziehen. Amazon unterstützt ebenfalls die Verwendung von Jupyter Notebooks mit vorinstallierten Algorithmen, Ressourcen und Developer Tools. Braket bietet On-Demand-Zugang zu verschiedenen Arten von Quantencomputern. Der Zugang zu Gatter-basierten Quantencomputern von IonQ und Rigetti, sowie zu einem Quanten-Annealer von D-Wave soll für den Benutzer so einfach wie möglich gestaltet werden, ohne dass dieser sich den Zugang bei einzelnen Anbietern beschaffen muss.

## Installation

Eines vorweg: Ein Großteil der oben erwähnten Features ist nur durch eine Registrierung bei AWS verfügbar. Auch wenn es Free-Plans sind und es damit eingeschränkte Funktionalität und monatliche oder absolute Zeitbeschränkungen gibt, ist für den Abschluss einer Registrierung das Hinterlegen von Kreditkarteninformationen notwendig.

Aus diesem Grund beschränken wir uns an dieser Stelle auf die Installation des Amazon Braket Python SDK. Dieses besitzt zwar auch eine starke Anbindung an Amazons AWS, jedoch lässt es sich problemlos auf einem lokalen Computer außerhalb der Cloud installieren und besitzt einen lokalen Simulator, welcher im kommenden Beispiel auch benutzt wird.

Wie immer empfiehlt es sich eine virtuelle Python Umgebung zu erzeugen. Die Installation des Braket SDK erfolgt mit:

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

## Beispiel

In unserem Beispiel wollen wir ein Bell-Paar (EPR-pair) erzeugen, indem wir auf QuBit 0 ein Hadamard-Gatter anwenden, bzw. wir zwischen QuBit 0 und 1 eine CNOT-Beziehung schaffen, in welcher QuBit 0 als Control-QuBit fungiert. Wie bereits erwähnt wählen wir als Simulator Device den im SDK enthaltenen lokalen Simulator und lassen die Schaltung 100-mal ausführen und geben das Ergebnis in der Kommandozeile (türkise Markierung) aus.

Anhand dieses Beispiels ist ebenfalls zu sehen, das mit Braket wenig Code ausreicht, um eine einfache Simulation auszuführen, wenn auch Amazon versucht den Benutzer für seine ersten Schritte in die Cloud zu

locken, indem sie in ihrer Dokumentation hauptsächlich auf ihre WEB-IDE und zur Ausführung auf Geräte in der Cloud verweisen.

```
>>> 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})
```

## Ergebnis

Wie man sieht, bekommt man als Ergebnis ausschließlich 00 und 11, was einem idealen Prozess ohne Rauschen und damit einer reinen Simulation ohne äußere Störung entspricht. Mit einer Verteilung von 56% zu 44% bei diesem Durchlauf kann man bei 100 Shots von einer Gleichverteilung sprechen, welche sich bei einer unendlichen Anzahl an Wiederholungen der 50%/50% Verteilung annähern würde.

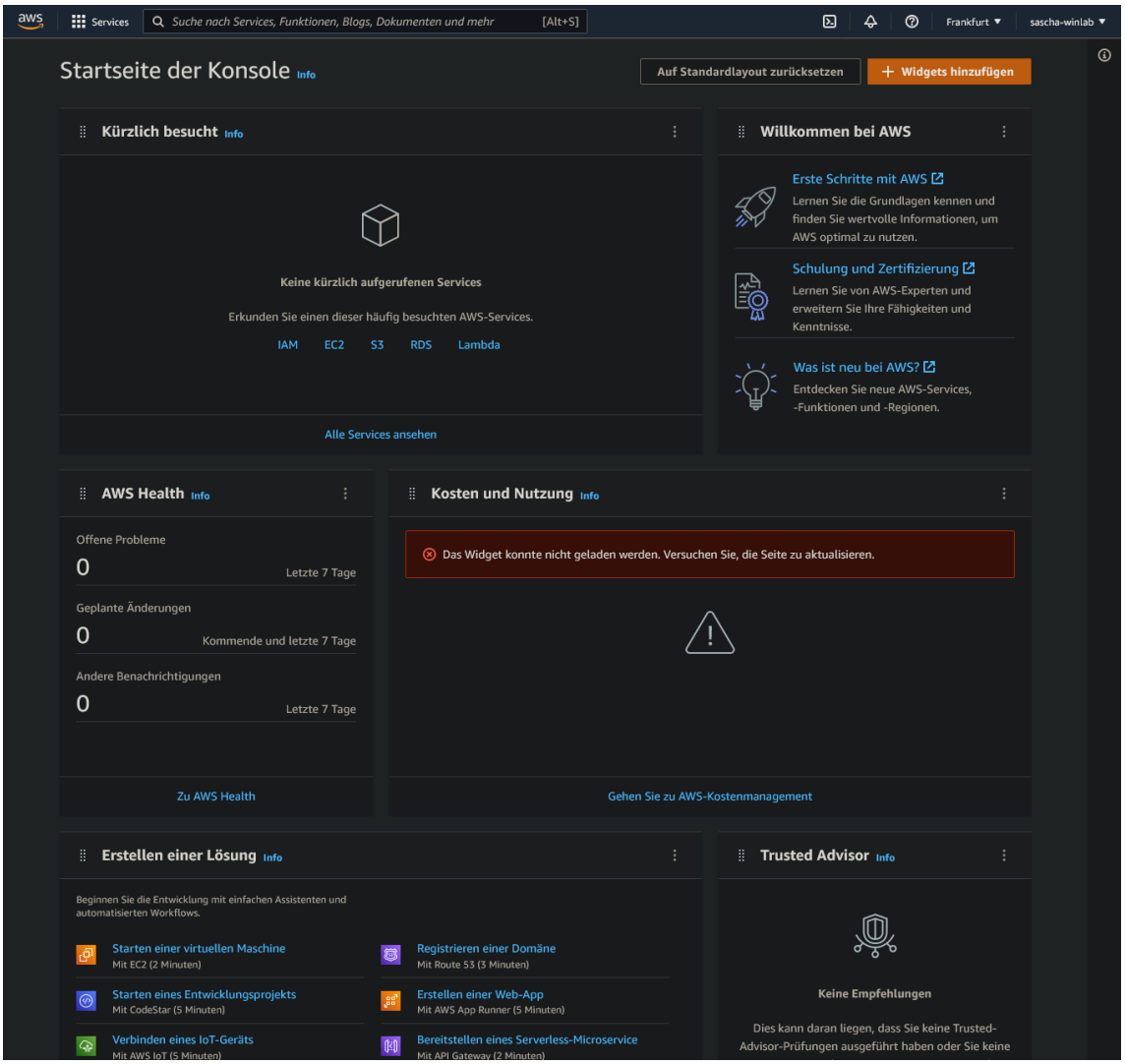



*Abbildung 30: Einstiegsseite AWS*

## Fazit

Amazon bietet mit Braket einen komfortablen Zugang zu einer ganzen Umgebung mit Build, Test und Run Tools zum Thema Quantumcomputing. Der Kern dieses Services, sprich das Braket SDK, steht dem Benutzer glücklicherweise auch ohne eine Registrierung außerhalb der Cloud zur Verfügung, ohne dass dieser sich für erste Tests an Amazon binden muss.

# Quantum Programming Studio

| | |
|---|---|
| **Repository** | https://github.com/quantastica/quantum-circuit |
| **Sprache** | Drag&drop / JavaScript |
| **OS** | Webbrowser |
| **Typ** | Schaltungssimulator |
| **Fokus** | Umfangreiche REST-API |
| **Besonderheit** | Konvertierung in viele Formate/Sprachen |
| **Lizenz** | MIT-Lizenz |
| **Website** | https://quantum-circuit.com |
| **Registrierung** | Für Benutzung des Webinterface ist eine kostenlose Registrierung erforderlich |

## Beschreibung

quantum-circuit ist ein Open-Source-Quantenschaltungssimulator, der in Javascript implementiert ist. Laut den Entwicklern sind Simulationen mit 20+ Qubits im Browser oder auf dem Server kein Problem.

Das Quantum Programming Studio ist eine webbasierte grafische Benutzeroberfläche, die es Benutzern ermöglicht, Quantenalgorithmen zu konstruieren und diese dann direkt im Browser zu simulieren oder auf echten Quantencomputern auszuführen. Der Schaltkreis kann in mehrere Quanten-Programmiersprachen/ Frameworks exportiert werden und kann auf verschiedenen Simulatoren und Quantencomputern ausgeführt werden.

Unterstützte Plattformen sind unter anderem: Rigetti Forest, IBM Qiskit, Google Cirq und TensorFlow Quantum, Microsoft Quantum Development Kit, Amazon Braket.

## Installation

Es ist keine Installation nötig. Man kann eine einfache Drag & Drop-Benutzeroberfläche verwenden, um einen Schaltplan zu erstellen, der automatisch in Code übersetzt wird, und umgekehrt - Sie können den Code eingeben und der Plan wird entsprechend aktualisiert.

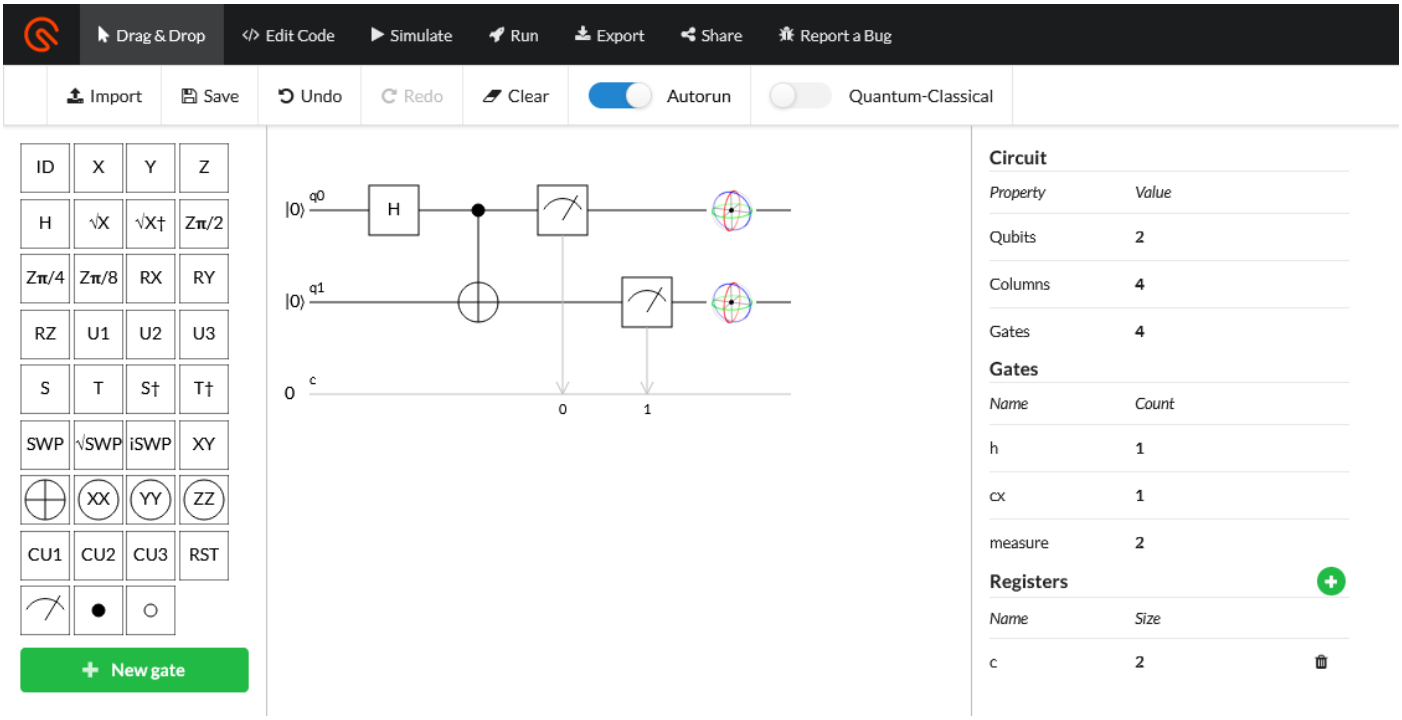



*Abbildung 31: Der Drag&Drop Editor (hier: Schaltung Bell-Paar)*

Wer dennoch an einer eigenen Installation interessiert ist, kann mit `npm install --save quantum-circuit` seine eigene Instanz in JavaScript implementieren. Die Entwicklung kann ebenfalls mit Jupyter Notebook erfolgen, sofern ein JavaScript Kernel installiert ist.

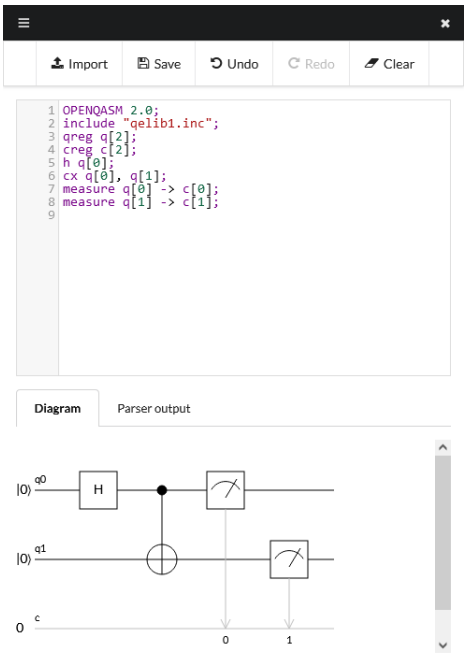



*Abbildung 32: Automatische Übersetzung in Code (Code-Editor)*

## Registrierung

Um den Service nutzen zu können ist eine kostenlose Registrierung unter https://quantum-circuit.com nötig. Dort können Projekte gespeichert und mit anderen geteilt werden. Zum aktuellen Zeitpunkt sind 1437 Projekte öffentlich zugänglich.

## Simulation im Browser

Die einfachste Möglichkeit der Simulation ist diese direkt im Browser laufen zu lassen. Hierfür muss einfach das gewünschte Projekt geöffnet werden und dann über den Reiter „Simulation“ ausgeführt werden. Pro Klick auf den Button „Simulate“ wird ein Durchgang ausgelöst.

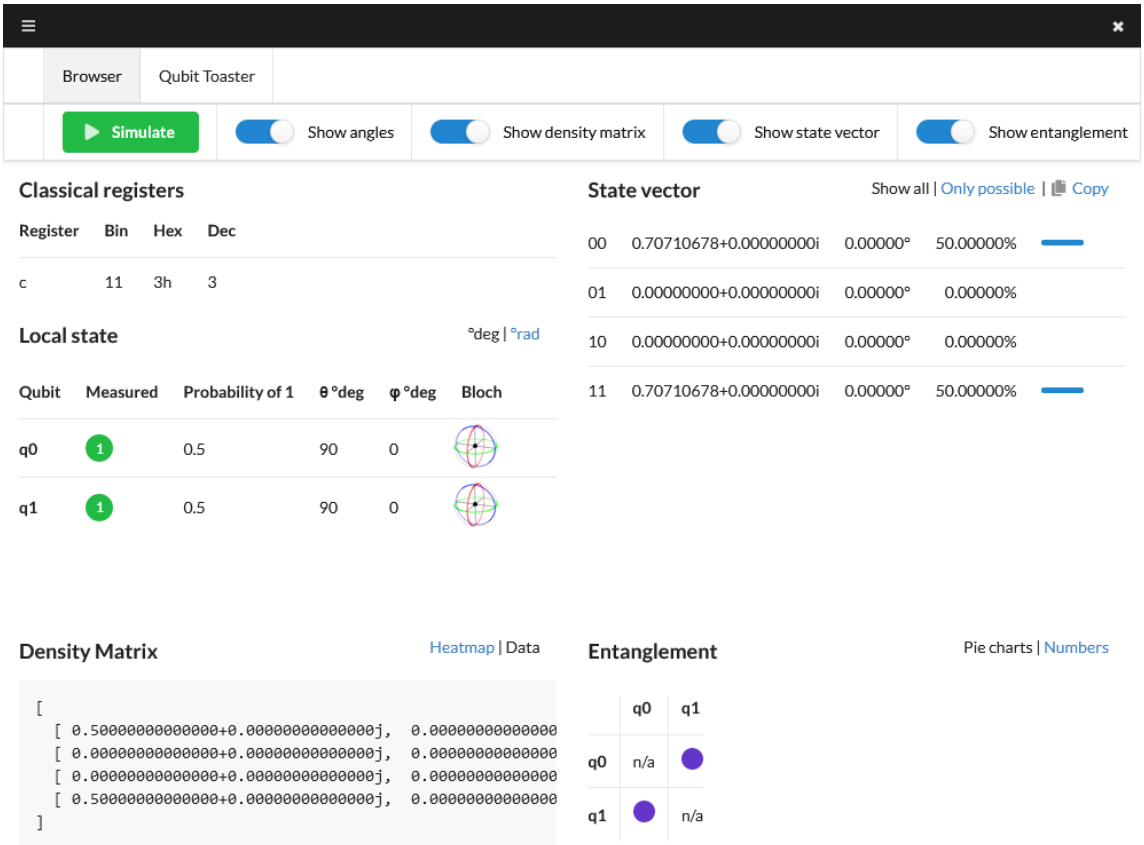



*Abbildung 33: Ergebnisse der Simulation im Browser*

## Simulation in der Cloud und auf echter Hardware

Über den Reiter „Run“ öffnet sich eine Dokumentation mit einer Beschreibung welche Remote Backends verwendet werden können und was dazu benötigt wird. Hauptsächlich wird hier der QPS Client benötigt, welcher über web sockets den QPS-Server anbindet. Mit diesem Client kann Quantum Programming Studio UI mit Rigetti QCS, Rigetti Forrest SDK, IBM Qiskit und Quantastica Qubit Toaster verbunden werden. Die Ressourcen von Rigetti scheinen jedoch Benutzern vorenthalten zu sein, die sich als Mitglieder einer Organisation bei Rigetti registrieren.

## APIs

Ein wichtiges Kernfeature ist die Interoperabilität mit anderen Sprachen. So können Schaltungen von OpenQASM und Quil importiert und Schaltungen nach OpenQASM, pyQuil, Quil, Qiskit, Cirq, TensorFlow Quantum, QSharp und QuEST exportiert werden. Schaltungen können ebenfalls im SVG-Format gespeichert werden.

Über die Rest API kann durch einen einfachen http-Request nahezu jedes beliebige Format bedient werden. Die passenden Links finden sich in den jeweiligen Projekt-Details.

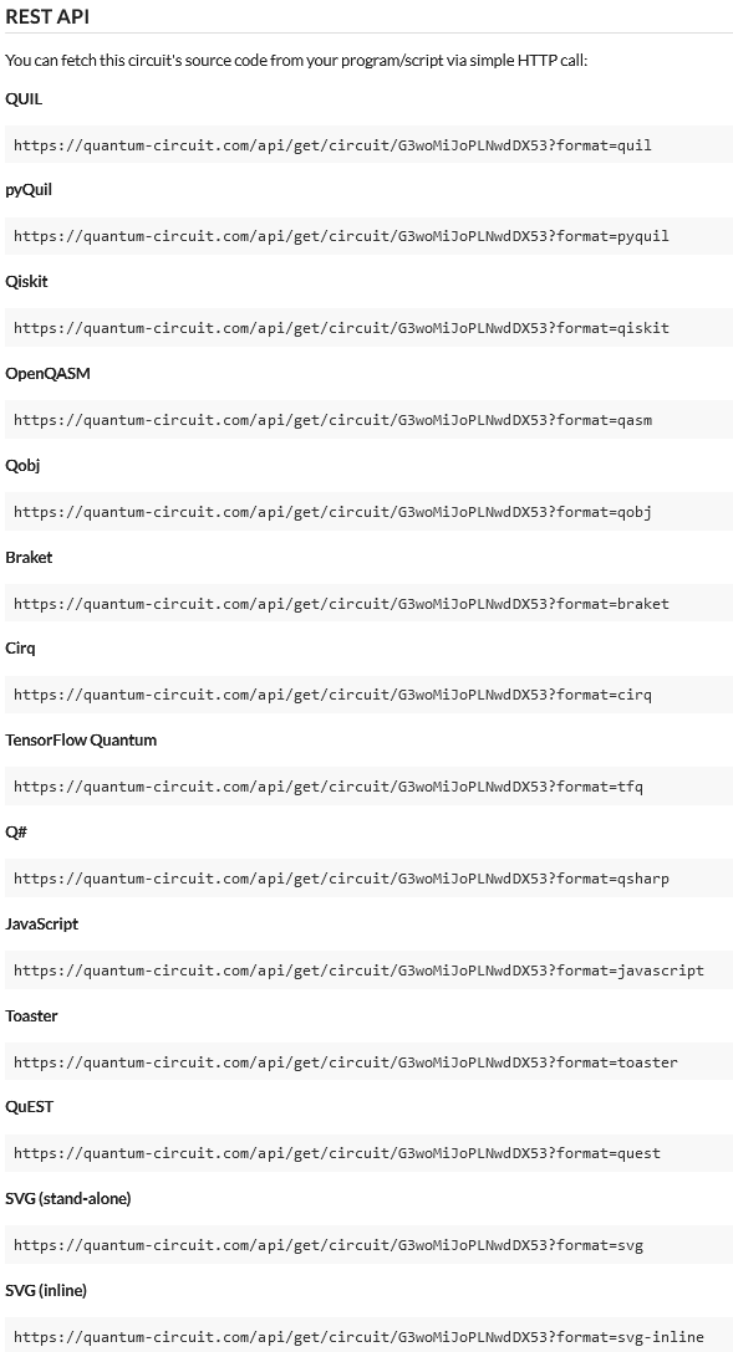



*Abbildung 34: Rest API des Testprojekts*

## Fazit

Das Quantum Programming Studio ist eine Empfehlung für alle Freunde einer webbasierten, grafischen Benutzeroberfläche. Sie ermöglicht es, Quantenalgorithmen zu konstruieren und direkt im Browser zu simulieren. Reicht dies nicht aus, stehen auch echte Quantencomputer zur Verfügung. Die Schaltkreise können plattformunabhängig in mehrere Quantenprogrammiersprachen/Frameworks etc. exportiert und auf verschiedenen Simulatoren und Quantencomputern ausgeführt werden. Die einfache Benutzeroberfläche erstellt automatisch Code, oder übersetzt Code in eine Schaltung; Diagramme werden in Echtzeit aktualisiert.

# Microsoft Azure Quantum/QKD/Q#

| | |
|---|---|
| **Repository** | https://github.com/microsoft/QuantumLibraries |
| **Sprache** | Q#, Python |
| **OS** | Azure Pakete lassen sich plattformunabhängig installieren |
| **Typ** | QKD bzw. Plattform |
| **Fokus** | Simulation von Quantenschaltungen, Optimierungsaufgaben, Simulation eines Quantencomputers mit Fullstack QKD |
| **Besonderheit** | Zugang zu anderen Anbietern |
| **Lizenz** | MIT License |
| **Website** | https://azure.microsoft.com/de-de/resources/development-kit/quantum-computing/ |
| **Registrierung** | Für QDK nicht nötig; Für Azure (Quantum) erhält man Guthaben |

Beschreibung

Das QDK ist das Development Kit für Azure Quantum. Es können Quantenanwendungen mit Q#, Qiskit oder Cirq erstellt und ausgeführt werden, sowohl auf echter Quantenhardware als auch mit klassischen Simulatoren, on- oder offline. Es können auch andere Aufgaben wie Optimierungen etc. formuliert und durchgeführt werden. Wie auch bei IBM Quantum/Qiskit und Google Quantum AI/Cirq muss zwischen dem eigentlich Framework (SDK) und der drauf aufgebauten Plattform unterschieden werden. Die Installation des SDKs ist möglich, obwohl auch Microsoft versucht den Benutzer zur Entwicklung auf die Plattform zu locken und Guthaben für diverse Ressourcen verspricht.

Das im Mittelpunkt stehende Q#, ist eine auf Quanten fokussierte High-Level-Programmiersprache von Microsoft und bietet einen Ansatz für die Entwicklung von Quantenprogrammen.

QDK

Das Quantum Development Kit umfasst eine funktionsreiche Integration mit Visual Studio, Visual Studio Code und Jupyter Notebooks. Die Q#-Programmiersprache kann eigenständig, in Notebooks und in der Befehlszeile verwendet werden, oder via Hostsprache mit Python- und .NET-Interoperabilität.

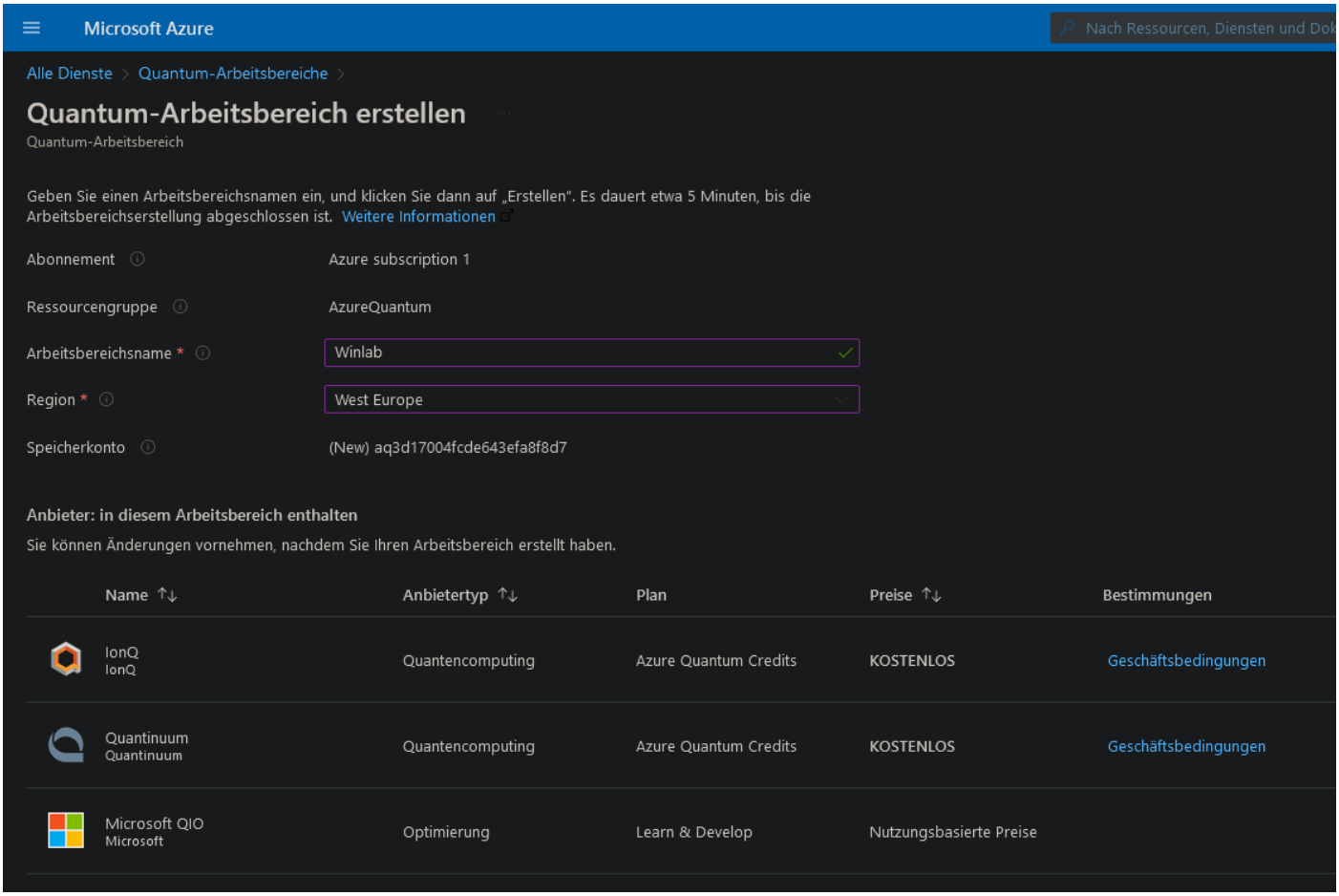



*Abbildung 35: Erzeugung eines Arbeitsbereichs in Azure*

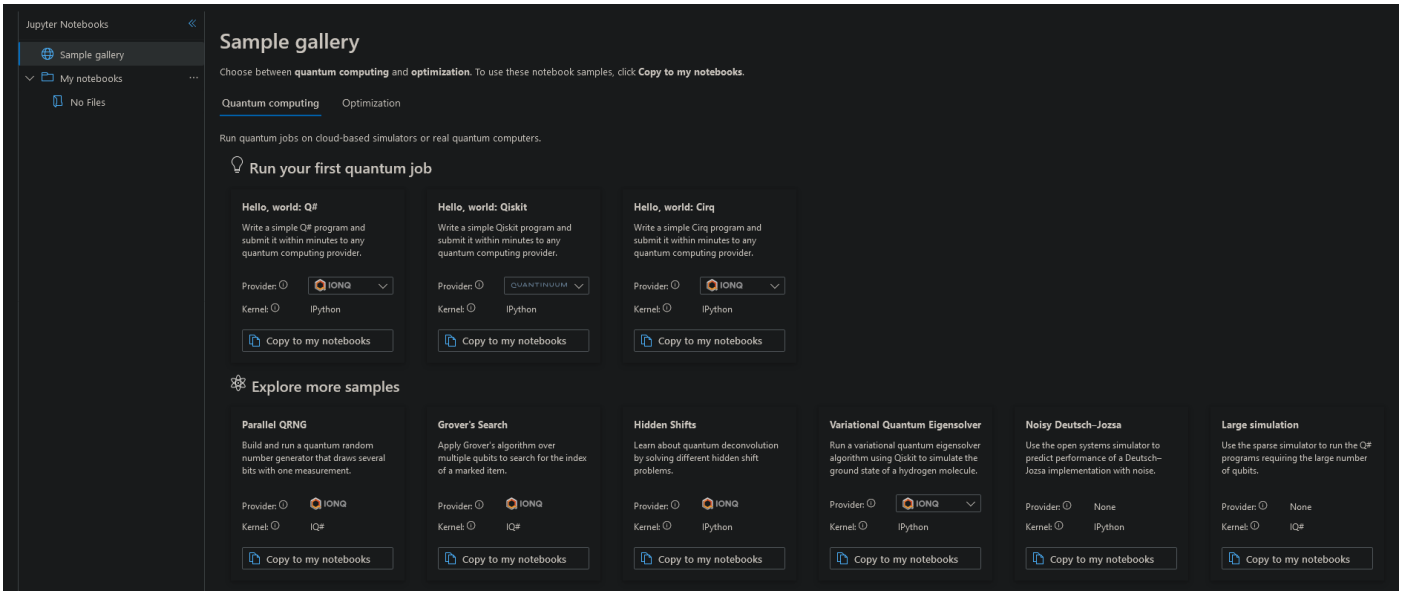



*Abbildung 36: Azure Quantum bietet bereits einige vordefinierte Programme und Schaltungen*

## Installation

Für die Nutzung des QDK gibt es verschiedenste Möglichkeiten zur Installation. So läuft Q# nicht nur zusammen mit Python und .NET, sondern es können ebenfalls auch noch andere Treiberprogramme für Hostsprachen wie C# oder F# genutzt werden. Für die Entwicklung bieten sich sowohl VS Code, als auch Jupyter Notebooks an, bzw. eine Client-Server-Beziehung zwischen VS Code und Notebooks.

Für diesen Test haben wir uns entschieden eine WSL2 Umgebung mit Ubuntu 20.04 zu nutzen. Wie von Microsoft empfohlen, wurde Anaconda als Python-Distribution installiert und ein Environment speziell für

das QKD erzeugt. Nach der Initialisierung durch das qsharp-Paket, kann der Jupyter Server gestartet und ein Notebook mit Q#-Kernel erzeugt werden.

# Installation Anaconda (Download Installation Script)

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

# Erzeugen und Aktivierung neuer virtueller Umgebung inklusive benötigter Pakete

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

```
conda activate qsharp-env
```

# Initialisierung

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

# Starten des Notebook Servers

```
jupyter notebook
```

Erstellen eines neuen Notebooks mit Q#-Kernel

```
New → Q#
```

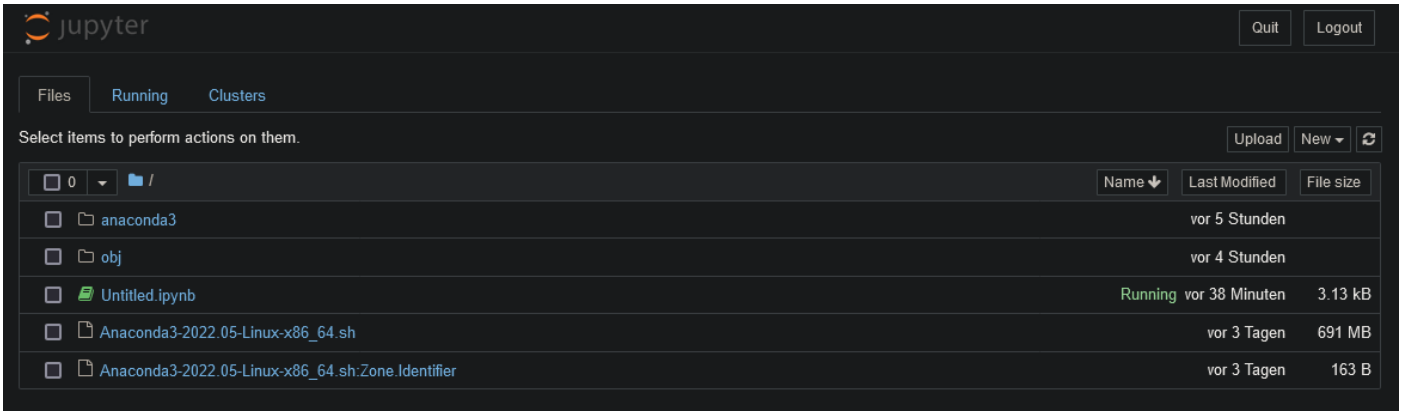



*Abbildung 37: Webserver Jupyter Notebook*

## Beispiel

In diesem Beispiel soll ein QuBit im Zustand |0⟩ initialisiert werden und durch die Funktion H() in Superposition versetzt werden, dass bei der darauffolgenden Messung ein 50%ige Chance auf entweder 0 oder 1 besteht, siehe Abbildung 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;
}
```

Mit der untenstehenden *magic*-Funktion kann das Ergebnis direkt unter der Zelle ausgegeben werden.

```
%simulate SampleQuantumRandomNumberGenerator
```

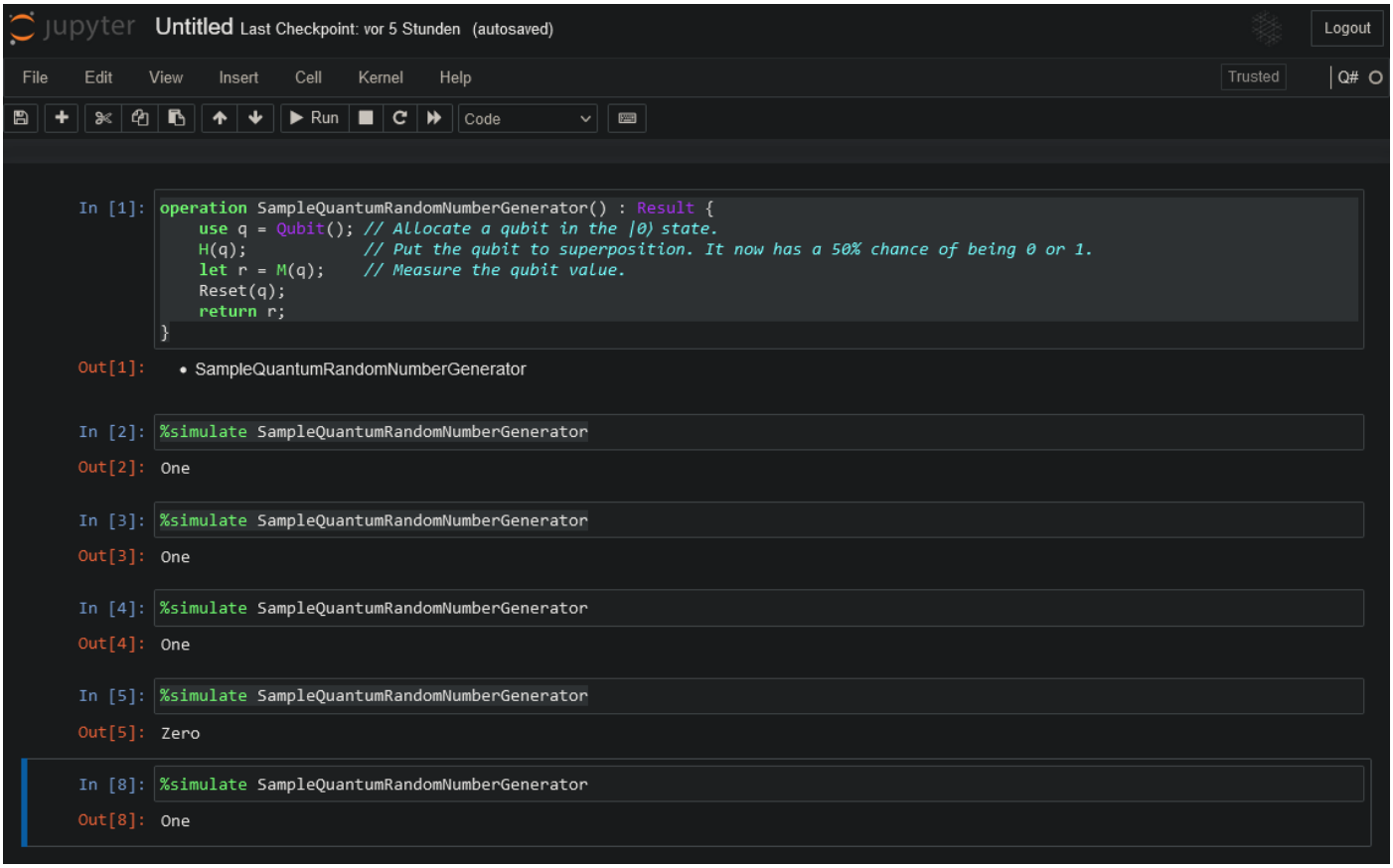



*Abbildung 38: Beispiel eines einfachen Programmes auf dem Webinterface des Jupyter Web-Servers*

## Fazit

Das Open-Source Quantum-Development Kit für Azure Quantum bietet Tools für die Entwicklung von Quantenanwendungen auf hardwarebeschleunigten Computeressourcen in Azure oder auf dem lokalen Host an. Nach Angaben von Microsoft ist Q# vollständig hardwareunabhängig, was bedeuten soll, dass Quantencomputerkonzepte unabhängig von zukünftigen Entwicklungen ausgedrückt werden können. Q#-Programme können auch gezielt auf verschiedenen Quantenhardware-Back-Ends in Azure Quantum ausgeführt werden. Ein Q#-Programm kann zu einer eigenständigen Anwendung kompiliert oder durch ein in Python oder in einer .NET-Sprache geschriebenes Hostprogramm aufgerufen werden.

Alles in allem ist Microsofts Azure Quantum Plattform eine vollwertige Umgebung wie Amazon Braket, oder IBM Quantum Qiskit, jedoch mit dem Unterschied, dass Microsoft hierfür seine eigene Sprache Q# verwendet.

## Schlusswort

Wir hoffen, dass dieser Bericht den Einstieg in diese äußerst intersannte Welt der Quanten-simulatoren erleichtert und den Testern viel Spaß mit weiteren Anwendungsbeispielen bietet. Der Testbericht konnte nur derzeitige Simulatoren und Plattformen berücksichtigen; es ist davon auszugehen, dass diese Frameworks sich laufend ändern und in Zukunft auch neue Simulatoren zur Anwendung kommen werden.