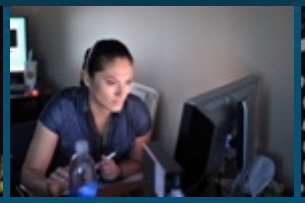
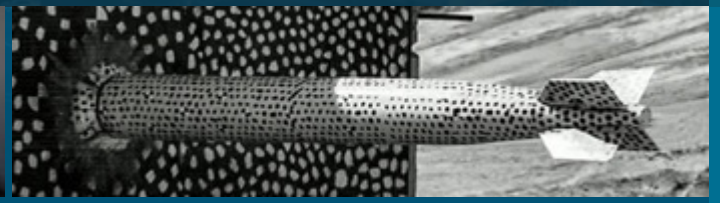
QSCOUT Webinar: Intro & Jaqal Overview





Presented by

Andrew J. Landahl

September 13, 2021

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Welcome & Introductions

Our round-two users:

1. **IBM:** Petar Jurcevic
   - *QV on QSCOUT plus ion transpiling*
2. **Indiana University:** Phil Richerme, Srinivansan Iyengar, Jeremy Smith, Amr Sabry
   - *Simulating Quantum Chemical Nuclear Dynamics problems on the QSCOUT*
3. **Johns Hopkins University:** Colin Trout, Robert Barr, Leigh Norris, Dave Clader
   - *Characterization and Optimal Control of Time-Correlated Amplitude Control Noise*
4. **Lawrence Livermore National Laboratory:** Kristin Beck, Stephen Libby, Jonathan DuBois, Ilon Joseph, Frank Graziani
   - *Using control pulse engineering to improve the effective fidelity of ion trap quantum computers*
5. **London Centre of Nanotechnology:** Andrew Green
   - *Simulation Quantum Evolution of Infinite Systems Using Tensor Networks on NISQ Devices*
6. **Super.tech:** Pranav Gokhale, Teague Tomesh, Victory Omole, and Frederic Chong
   - *Low-Level Control and Performance Benchmarking*

Other attendees

- **Tutorial speakers:** Susan Clark, Andrew Landahl, Antonio Russo, Jay Van Der Wall, Kenny Rudinger, Benjamin Morrison, Oliver Maupin, Dan Lobser, Matt Chow
- **Other Sandians:**
- **Other non-Sandians:**

Logistics

- Sandia's MS Teams does not include all possible functionality.
  - Can call your (US) phone if your audio doesn't work, though.
- Feel free to interrupt with questions. Just say "Question" or similar to jump in.
- If possible, please show your face when asking/answering questions.
- Please mute when not asking/answering a question to cut down on background noise.
- Can also pose questions in chat.
- Talks are being recorded; we will strive to distribute them afterwards.
- Stay hydrated—it's a long day! Scheduled breaks are good times for "bio breaks."

Other logistical issues

- Feel free to interrupt with questions. If it becomes problematic, we will consider alternatives. We decided not to squelch with “Zoom Webinar” format.
- If possible, please show your face when asking/answering questions.
- Please mute when not asking/answering a question to cut down on background.
- No guarantee that questions posed in chat will be answered.
- Talks are being recorded; we will strive to distribute them afterwards.
- If folks want to use the 1.5-hour long break for chatting via Zoom they can (we will leave it on), but we aren’t supporting breakout Zoom rooms.
- Stay hydrated—it’s a long day!

Any logistical questions before we begin?

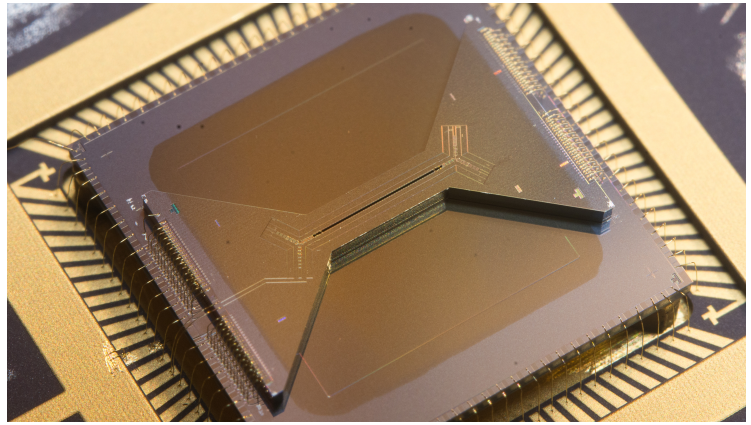
Part 1: Overview

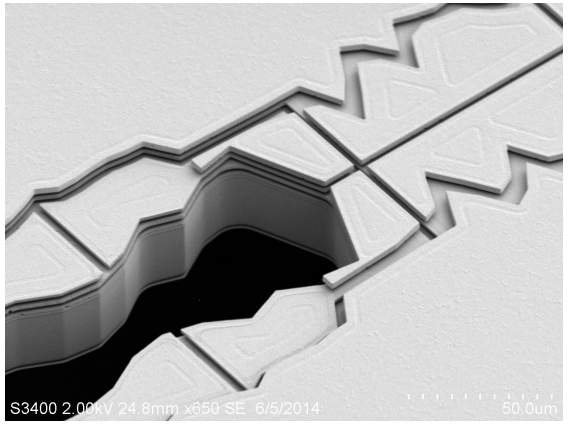
“Computer science is no more about computers than astronomy is about telescopes.”
—(Mis?)attributed to E. Dijkstra

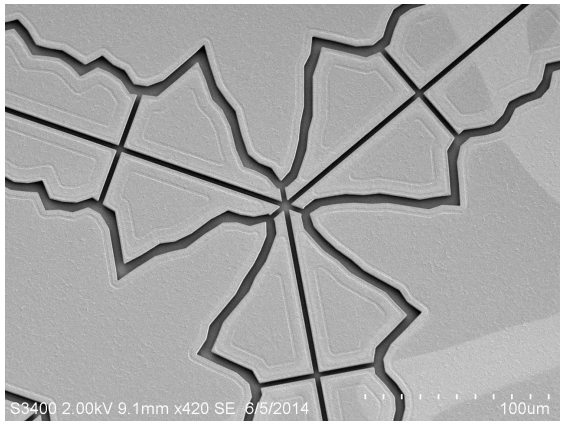
Image: Wikimedia Commons, Blue computer icon, Inter Oficial (2013)

QSCOUT

The quantum computer *is* the object of study







Wanted: A programming language that forces the quantum computer to do *exactly* what I want and *exactly* when I want it.

"A language for micro-managing control freaks."

(Tagline failed in focus groups.)

Existing languages

Cirq

OpenQASM

Q#

Quil

Designed for performance, not control

Many behind-the-scenes "optimizations" and/or lacking connection to hardware primitives.

Images: Company websites.

Enter the Jaqal

JAQAL

Just another quantum assembly language

"A language for quantum computer scientists, by quantum computer scientists."

(Tagline fared better in focus groups.)

Key features of Jaqal

1. Precise addressing of qubits in the qubit register.
2. Precise control of parallel and serial gate scheduling.
3. Explicit connection of gates to pulse programs.
4. Support for gate and qubit macros and aliases.
5. A purely quantum language (except fixed-length loops)*

*Future Jaqal versions will allow classical arithmetic in defined constants and fast conditional “branch” statements on measurement outcomes.

A purely quantum language?

Verboten

- If-then
- Do-while
- For-until
- 1+1
- 3.14 / 2.0
- rand()

Why this madness?

- Ions are slow: Metaprogramming works!
- QSCOUT 1.0 has no mid-circuit measurements
- Why reinvent classical programming?

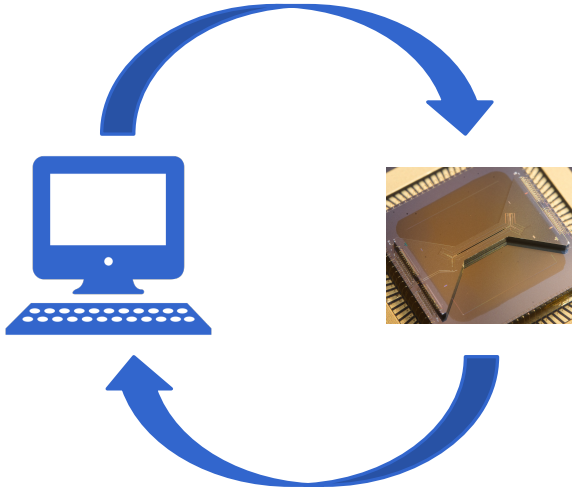


Image: Wikimedia Commons, Blue computer icon, Inter Oficial (2013)

Jaqal's connection to pulse programs

```
from <gate pulse file> usepulses *
```

- The first line of any Jaqal program.
- Defines the native gates; allows multi-platform support.

Examples

```
from qscout.v1.std usepulses *

from qscout.v1.stretched usepulses *

from qscout.v2.std usepulses *

from my_favorite_tech.v1.std usepulses *
```

Unsupported, but planned feature: Jaqal importing

```
import <jaqal file>
```

- Will allow one to split jaqal code across multiple files.
- Currently unsupported, but `import` reserved as a keyword.

(Non-)Examples

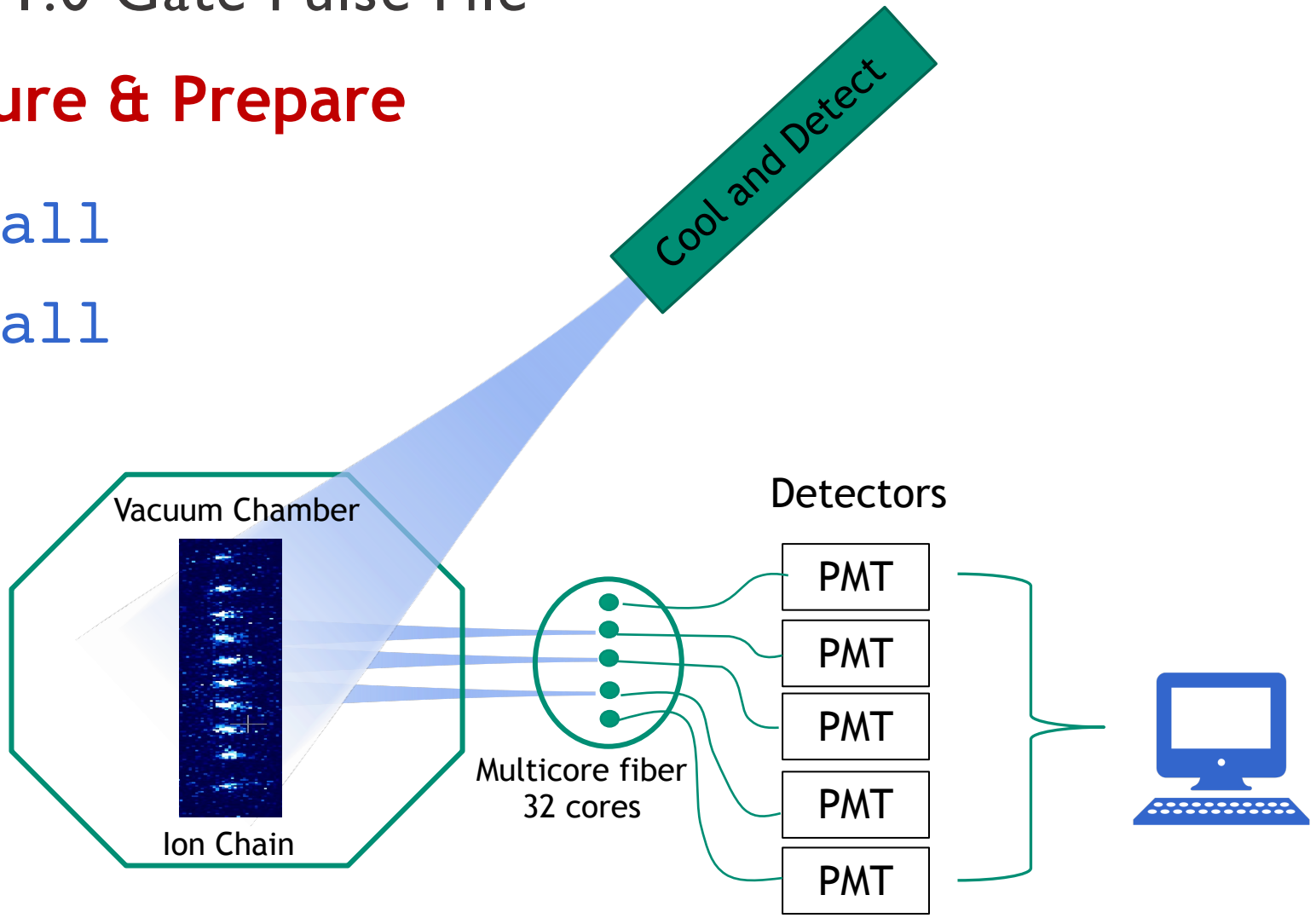
```
import standard_gate_macros.jql

import named_constants.jql
```

Aside: “.jql or .jaqal?” See “.jpg or .jpeg?”

QSCOUT 1.0 Gate Pulse File

Gates: Measure & Prepare

```
prepare_all
measure_all
```



Can only prepare & measure all qubits simultaneously in the *Z* basis.

Image: Wikimedia Commons, Blue computer icon, Inter Oficial (2013)

QSCOUT 1.0 Gate Pulse File

Gates: Continuously parameterized

```
Rx <qubit> <rotation angle>
Ry <qubit> <rotation angle>
Rz <qubit> <rotation angle>
MS <qubit> <qubit> <axis angle> <rotation angle>
```

$$\mathrm{MS}(q_1, q_2, \varphi, \theta) = \exp\left[-i\left(\frac{\theta}{2}\right)(X\cos\varphi + Y\sin\varphi)^{\otimes 2}\right]$$

Notes

- All angles are for counter-clockwise rotations in base-10 radians.
- Angles only have meaning to 40 bits of precision for QSCOUT 1.0.

QSCOUT 1.0 Gate Pulse File

Gates: Counter-clockwise π rotations

```
Px <qubit>        Py <qubit>        Pz <qubit>
```

Gates: Counter-clockwise $\pi/2$ rotations

```
Sx <qubit>        Sy <qubit>        Sz <qubit>
```

Gates: Clockwise $\pi/2$ rotations

```
Sxd <qubit>       Syd <qubit>       Szd <qubit>
```

Gates: "Standard" Mølmer-Sørensen gate

```
Sxx <qubit> <qubit>
```

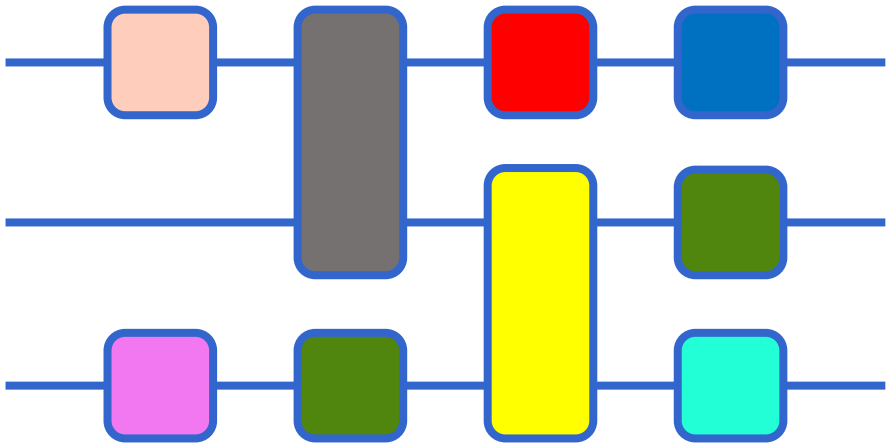
$$\mathrm{S}_{xx}(q) = \exp\left[-i\left(\frac{\pi}{4}\right)(X \otimes X)\right]$$

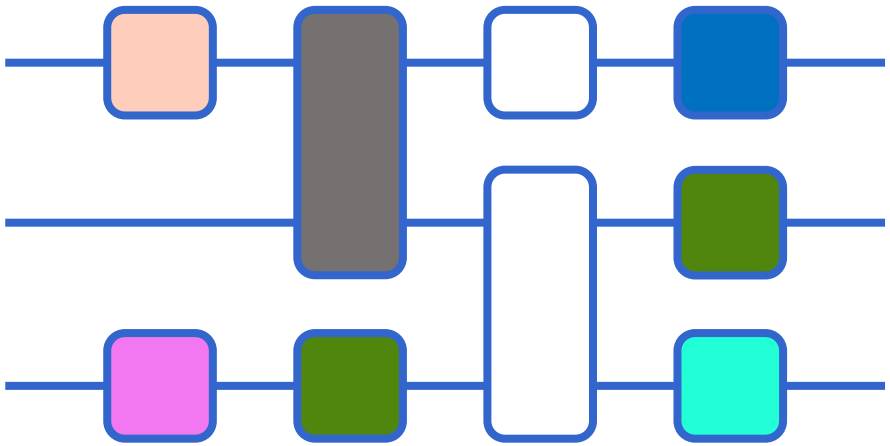
QSCOUT 1.0 Gate Pulse File

Gates: Idles with known timing (14 new gates)

```
I_<gate> <args>
```

Why?





Sneak Peek: JaqalPaw

```python
    def gate_prepare_all(self, qubit_num=8):
        return [PulseData(ch, 3e-7, waittrig=False) for ch in range(qubit_num)] + \
               [PulseData(ch, 3e-7, waittrig=True) for ch in range(qubit_num)]

    def gate_measure_all(self, qubit_num=8):
        return [PulseData(ch, 3e-7, waittrig=False) for ch in range(qubit_num)]*2

    def gate_R_copropagating_square(self, qubit, theta, phi):
        phase = (phi < 0)*180 + theta/math.pi*180
        return [PulseData(qubit,
                          self.duration_from_rabi_angle(phi, 1.0, self.single_qubit_rabi_angle_calibrations[qubit]),
                          amp0=50.0,
                          amp1=50.0,
                          freq0=self.aom_center_frequency-self.adjusted_carrier_splitting/2,
                          freq1=self.aom_center_frequency+self.adjusted_carrier_splitting/2,
                          phase0=0,
                          phase1=phase,
                          fb_enable_mask=0b01,
                          sync_mask=0b11)]

    def gate_R(self, qubit, theta, phi):
        return self.gate_R_copropagating_square(qubit, theta, phi)

    def gate_Rx(self, qubit, phi):
        return self.gate_R(qubit, 0, phi)
```

Part 2: Jaqal syntax

References:
qscout.sandia.gov/jaqal
arXiv:2003.09382

Whitespace, Identifiers, and Comments

Whitespace & Identifiers

- Whitespace ignored, but newlines indicate sequential instructions.
- *Identifiers* are gate names, register names, register aliases, constant names, and macro arguments. (None can be overloaded.)
- Identifiers must use characters from `[a-z]`, `[A-Z]`, `[0-9]`, `_`.
- Identifiers cannot start with numerals.
- Identifiers cannot be Jaqal language `keywords`.
- Identifiers are case sensitive.

Comments

- C++ style: `//` for single-line and `/* … */` for multi-line comments.

Header statements

Register

```
register q[7] // Hardware spec IDs the qubits.
```

Notes: Program is rejected immediately if machine cannot supply enough qubits. Array indices start with `0`. Only one register allowed.

Map

```
map ancilla q[0]
map qubits q
map sbits q[1:7:2]
```

```
sbits[0] = q[1]
sbits[1] = q[3]
sbits[2] = q[5]
```

Note: Follows python slicing syntax. Final argument must be the register (or a register slice). (Prevents self-referentiality.)

Header statements

Let

```
let num_iters   4
let pi          3.1415926536   // delicious!
let pi_3_32     0.2945243113   // 3*pi/32
let pi_m_8     -0.3926990817   // -pi/8
```

Note: Remember, no classical computing, so the following is invalid:

```
let pi_2 3.14/2
```

Body statements

Sequential gate block

- `{}` curly brackets. Newlines or semicolons `;` are separators

Parallel gate block

- `<>` angle brackets. Newlines or vertical-pipes `|` are separators.

Gate block nesting

- OK, except blocks cannot be nested directly within other blocks of the same type.

Gate block examples

```
{ Sx q[0] ; Sx q[1] }
< Sx q[0] | Sx q[1] >
{
    Sx q[0]
    Sx q[1]
}
<
    Sx q[0]
    Sx q[1]
>
```

```
{ Sxx q[0] q[1]
  < Sx q[0] | Sx q[1] > }
< Sx q[0]
  { Sx q[1] ; Sx q[1] } >
<
    Rx q[0] 0.0000001
    Ry q[1] 3.1415926
>
```

Note: QSCOUT v1 native gates pad idles at the *end*.

Body statements

Forbidden in a gate block

- Any empty statement in a sequential block
- Only empty statements in a parallel block
- Header statements
- Macro definitions (will define macro syntax later)
- Loop blocks in a parallel block (will define loop syntax later)

Allowed in a gate block, but may cause problems

- Parallel gates forbidden by hardware design rules.
- Sequential gates forbidden by hardware design rules.

Example:

- QSCOUT v1 cannot execute gates in parallel with MS gates.

Macros

```
// Implementation from [Maslov, 2017]
macro cnot control target {
    Sy control ; Sxx control target
    < Sxd control | Sxd target > ; Syd control
}
```

Notes:

- Even one-gate macros require a sequential code block.
- Line break not allowed before the initial `{`. (Aids parsing.)
- Macro argument name takes precedence over any other named identifiers.
- Macros can use other macros only if they defined earlier in the file.
  - This prevents recursion in macro definitions; there is no stack!

Body statements

Loops

```
loop 1024 { // Prepare and measure a Bell state 1024 times
    prepare_all ; Sxx q[0] q[1] ; measure_all
}
let run_code_block 0 // A way to hack true/false with 0/1
loop run_code_block {
    prepare_all ; Sxx q[0] q[1] ; measure_all
}
```

Notes:

- Even one-gate loops require a sequential code block.
- Line break not allowed before the initial `{`.
- Only constant integers allowed for loop argument.
- Reminder: No loops allowed in parallel blocks.

Output format

Little endian and newline terminated

```
register q[2]
loop 2 { prepare_all ; Px q[0] ; measure_all }
loop 2 { prepare_all ; Px q[1] ; measure_all }
```

```
10
10
01
01
```

A final note on metaprogramming

Wrong: No arithmetic allowed

```
macro CRz control target angle {
    Rz target angle/2
    CNOT control target
    Rz target -angle/2
    CNOT control target
}
```

Right: Precompute angle arguments with metaprogramming

```
macro CRz control target angle_2 angle_m_2 {
    Rz target angle_2
    CNOT control target
    Rz target angle_m_2
    CNOT control target
}
```

Questions?

Installing JaqalPaq

PRESENTED BY

Antonio Russo

September 13, 2021

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Outline

- Where to find JaqalPaq
- Platform specific considerations
- Preparing the virtual environment
- Installing JaqalPaq
- How to run the tests
- Extra goodies

If you're in a hurry:

- Please use a venv
  - pip install --upgrade Cython numpy pip wheel
    pip install ipykernel JaqalPaq'[pygsti-integration]' QSCOUT-gatemodels
- (Inside the venv) If you're using Jupyter, to install a kernel:
  - ipython kernel install --name=jaqal
- (Outside the venv) To remove an old kernel:
  - jupyter kernelspec remove jaqal
- https://gitlab.com/jaqal/jaqalpaq has these instructions

Where to find JaqalPaq

- We’re on Gitlab: https://gitlab.com/jaqal
  - Required: git clone https://gitlab.com/jaqal/jaqalpaq
  - For QSCOUT-specific functions: git clone https://gitlab.com/jaqal/qscout-gatemodels
  - Transpilers: git clone https://gitlab.com/jaqal/jaqalpaq-extras
- Also on pyPI.org: https://pypi.org/project/JaqalPaq/
  - pip install JaqalPaq QSCOUT-gatemodels JaqalPaq-extras
- Documentation is at https://jaqalpaq.readthedocs.io

Platform specific considerations

- On Mac:
  - If you have macOS 10.14 (Mojave), may need to run the following to get Cython support:
  - export MACOSX_DEPLOYMENT_TARGET=10.14
    open /Library/Developer/CommandLineTools/Packages/macOS_SDK_headers_for_macOS_10.14.pkg
- On Windows:
  - (recommended) Use native Linux virtualization layer: WSL (Ubuntu or Debian) (see Linux instructions);
  - Use a conda virtual environment; or,
  - Install MS Visual C++ and Python3 directly
  - “pip version of pywin32 did not link dlls properly. Reinstall via "conda install pywin32" and then re-install everything and install "pip install resources" and "pip install psutil" ” **Thanks Kristin Beck!**
- Linux
  - Make sure you have gcc, python3 and python3 development files:
    - apt install python3-dev build-essential
    - yum install python3-devel gcc gcc-c++
- Test for headers with python3-config --includes

Preparing the virtual environment

- I use conda-forge:

```
conda create -n your-preferred-name python=3.9
conda activate your-preferred-name
conda install Cython numpy
```

- A venv will also work well:
- python3 -m venv /path/to/venv
  source /path/to/venv/bin/activate
  pip install --upgrade pip wheel Cython numpy
- If you want to use Jupyter, install ipykernel (conda install ipykernel or pip install ipykernel)
- ipython kernel install --name=jaqal (remove with jupyter kernelspec remove jaqal)

Installing JaqalPaq

- pip install JaqalPaq'[pygsti-integration,notebooks]' QSCOUT-gatemodels \
  JaqalPaq-extras'[qiskit,pyquil,cirq,projectq,pytket,tutorial]'
  - **Everything should be installed at the same time to ensure compatible dependencies**
- But, you don’t have to use everything:
  - pygsti-integration gives the emulator
  - notebooks installs all dependencies needed for the example chemistry notebooks
  - tutorial installs additional dependencies for the transpiler tutorial
  - The five JaqalPaq-extras targets install compatible versions of the associated 3rd-party packages
  - QSCOUT-gatemodels provides the native QSCOUT instruction set
- If pygsti complains about Cython extensions (and you care):
  - Make sure you installed the compiler and development headers (and numpy)
  - pip uninstall pygsti && pip cache remove pyGSTi **Case sensitive for cache remove!**
  - pip install --no-binary pygsti

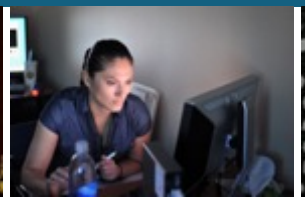
How to run the tests

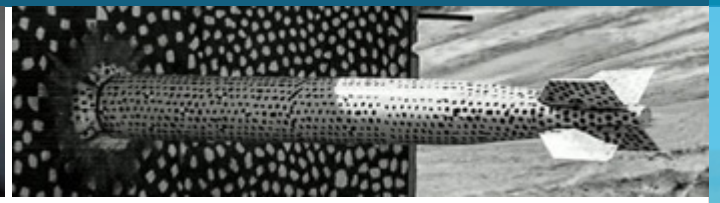
- Tests are distributed with the pyPI packages under <PREFIX>/share/jaqalpaq
- In a venv:
  - cd "$VIRTUAL_ENV/share/jaqalpaq"
- In a Conda environment
  - cd "$CONDA_PREFIX/share/jaqalpaq"
- Make sure to install pytest
  - conda install pytest
  - pip install pytest
- Run pytest on the tests directory:
  - pytest tests

Extra goodies

- Vim Jaqal syntax highlighting installed under <PREFIX>/share/vim/{addons,ftdetect}
- All the example notebooks are under <PREFIX>/share/jaqalpaq/examples
- Sample jaqal files (syntactically correct Jaqal) in <PREFIX>/share/jaqalpaq/examples/jaqal

Writing Jaqal with Jaqalpaq





Presented by

Jay Van Der Wall, SNL

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Why should I use Jaqalpaq?

You don't have to! (But you do have to use Jaqal)

Jaqal is a simple, low level language. There is no compiler to Jaqal.

Primary use cases for Jaqalpaq:

- Running many similar gates in the same file
- Running many similar files with feedback

Alternatives to Jaqalpaq

- Running a single Jaqal file
- Programmatically editing Jaqal as a string

Jaqal Example

```
register q[1]
let theta 1.234
prepare_all
Rx q[0] theta
measure_all
```

Jaqalpaq Example

```
from jaqalpaq.core import CircuitBuilder
from jaqalpaq.run import run_jaqal_circuit
from qscout.v1 import native_gates
b = CircuitBuilder(native_gates=native_gates.NATIVE_GATES)
q = b.register("q", 1)
θ = b.let("theta", 1.234)
b.gate("prepare_all")
b.gate("Rx", q[0], θ)
b.gate("measure_all")
c = b.build()
r = run_jaqal_circuit(c)
print(r.subcircuits[0].probability_by_int)
```

```
register q[1]
let theta 1.234
prepare_all
Rx q[0] theta
measure_all
```

Contact your POC for the latest recommended way to do simple scans like changing a single angle. Jaqalpaq may be overkill.

Parallel Gates

```
b = CircuitBuilder(native_gates=native_gates.NATIVE_GATES)
q = b.register("q", 2)
θ = b.let("theta", 1.234)
φ = b.let("phi", -0.123)
b.gate("prepare_all")
bb = b.block(parallel=True)
bb.gate("Rx", q[0], θ)
bb.gate("Rx", q[1], φ)
b.gate("measure_all")
```

```
let theta 1.234
let phi -0.123
register q[2]
prepare_all
<Rx q[0] theta | Rx q[1] phi>
measure_all
```

Check with your POC for the latest restrictions on parallel gates!

Loops

```
from jaqalpaq.core import CircuitBuilder, SequentialBlockBuilder
from qscout.v1 import native_gates
b = CircuitBuilder(native_gates=native_gates.NATIVE_GATES)
q = b.register("q", 1)
θ = b.let("theta", 1.234)
i = b.let("iters", 5)
b.gate("prepare_all")
bb = SequentialBlockBuilder()
bb.gate("Rx", q[0], θ)
lb = b.loop(i, bb)
b.gate("measure_all")
c = b.build()
```

```
register q[1]
let theta 1.234
let iters 5
prepare_all
loop iters {
  Rx q[0] theta
}
measure_all
```

Macros

```
b = CircuitBuilder(native_gates=native_gates.NATIVE_GATES)
q = b.register("q", 1)
θ = b.let("theta", 1.234)
bb = SequentialBlockBuilder()
bb.gate("Rx", "q", θ)
m = b.macro("rot", ["q"], bb)
b.gate("prepare_all")
b.gate("rot", q[0])
b.gate("measure_all")
```

```
register q[1]
let theta 1.234
macro rot q {
  Rx q theta
}
prepare_all
rot q[0]
measure_all
```

Debugging with the generator

```
from jaqalpaq.core import CircuitBuilder
from jaqalpaq.run import run_jaqal_circuit
from qscout.v1 import native_gates
from jaqalpaq.generator import generate_jaqal_program
b = CircuitBuilder(native_gates=native_gates.NATIVE_GATES)
q = b.register("q", 1)
θ = b.let("theta", 1.234)
b.gate("prepare_all")
b.gate("Rx", q[0], θ)
b.gate("measure_all")
c = b.build()
print(generate_jaqal_program(c))
```

```
let theta 1.234

register q[1]

prepare_all
Rx q[0] theta
measure_all
```

Notice the order is not the same, but program is equivalent.

For More Information

Code used to build a circuit:

- https://gitlab.com/jaqal/jaqalpaq/-/blob/master/src/jaqalpaq/core/circuitbuilder.py

API of resulting objects:

- https://gitlab.com/jaqal/jaqalpaq/-/blob/master/src/jaqalpaq/core/circuit.py

jaqalpaq.core documentation:

- https://jaqalpaq.readthedocs.io/en/release-v1.0/jaqalpaq.core.html

Contact your POC if your use case isn't covered.

Using the Jaqal Emulator

PRESENTED BY

Benjamin Morrison, Kenneth Rudinger, Antonio Russo

September 13, 2021

QSCOUT Software Stack webinar

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Outline

QSCOUT

- Introduction (Kenny)
  - Anatomy of Jaqal circuit
  - Command line interface demo
- Python Interface
  - Unitary emulator (Ben)
  - Noisy emulator (Antonio)
- Conclusions (Kenny)

Anatomy of a Jaqal circuit

QSCOUT

Circuit

```
from qscout.v1.std usepulses *

register q[2]

let pi2 1.5707963267948966
let pi4 0.7853981633974483

macro hadamard target { // A Hadamard gate can be implemented as
    Sy target           // a pi/2 rotation around Y
    Px target           // followed by a pi rotation around X.
}
macro cnot control target {  // CNOT implementation from Maslov (2017)
    Sy control
    MS control target 0 pi2
    <Sxd control | Sxd target>  // we can perform these in parallel
    Syd control
}

prepare_all
 hadamard q[0]
 cnot q[0] q[1]
measure_all

loop 10 {
 prepare_all
 hadamard q[0]
 cnot q[0] q[1]
 Px q[0]
 measure_all
}
```

A *circuit* is the whole Jaqal file.

Anatomy of a Jaqal circuit

QSCOUT

Circuit

```
from qscout.v1.std usepulses *

register q[2]

let pi2 1.5707963267948966
let pi4 0.7853981633974483

macro hadamard target { // A Hadamard gate can be implemented as
    Sy target           // a pi/2 rotation around Y
    Px target           // followed by a pi rotation around X.
}
macro cnot control target {  // CNOT implementation from Maslov (2017)
    Sy control
    MS control target 0 pi2
    <Sxd control | Sxd target>  // we can perform these in parallel
    Syd control
}

prepare_all
 hadamard q[0]
 cnot q[0] q[1]
measure_all

loop 10 {
 prepare_all
 hadamard q[0]
 cnot q[0] q[1]
 Px q[0]
 measure_all
}
```

Subcircuit

A *circuit* is the whole Jaqal file. A *subcircuit* is a portion of code delineated by prepare_all and measure_all.

Anatomy of a Jaqal circuit

QSCOUT

Circuit

```
from qscout.v1.std usepulses *

register q[2]

let pi2 1.5707963267948966
let pi4 0.7853981633974483

macro hadamard target { // A Hadamard gate can be implemented as
    Sy target           // a pi/2 rotation around Y
    Px target           // followed by a pi rotation around X.
}
macro cnot control target {  // CNOT implementation from Maslov (2017)
    Sy control
    MS control target 0 pi2
    <Sxd control | Sxd target>  // we can perform these in parallel
    Syd control
}

prepare_all
 hadamard q[0]
 cnot q[0] q[1]
measure_all

loop 10 {
 prepare_all
 hadamard q[0]
 cnot q[0] q[1]
 Px q[0]
 measure_all
}
```

Subcircuit

Subcircuit

A *circuit* is the whole Jaqal file. A *subcircuit* is a portion of code delineated by prepare_all and measure_all.

Command-line interface

Can emulate any valid .jaqal file with (unitary) emulation by invoking the command line interface (CLI) command jaqal-emulate. This will yield both bit-string readouts (one per compiler pass of each measurement) and outcome probabilities (one per subcircuit- i.e., one per *written instance* of measure_all).

Command-line interface

QSCOUT

Can emulate any valid .jaqal file with (unitary) emulation by invoking the command line interface (CLI) command jaqal-emulate. This will yield both bit-string readouts (one per compiler pass of each measurement) and outcome probabilities (one per subcircuit- i.e., one per *written instance* of measure_all.

Circuit

```
from qscout.v1.std usepulses *

register q[2]

let pi2 1.5707963267948966
let pi4 0.7853981633974483

macro hadamard target { // A Hadamard gate can be implemented as
    Sy target           // a pi/2 rotation around Y
    Px target           // followed by a pi rotation around X.
}
macro cnot control target {  // CNOT implementation from Maslov (2017)
    Sy control
    MS control target 0 pi2
    <Sxd control | Sxd target>  // we can perform these in parallel
    Syd control
}

prepare_all
 hadamard q[0]
 cnot q[0] q[1]
measure_all

loop 10 {
 prepare_all
 hadamard q[0]
 cnot q[0] q[1]
 Px q[0]
 measure_all
}
```

Subcircuit

Subcircuit

Command-line interface

QSCOUT

Can emulate any valid .jaqal file with (unitary) emulation by invoking the command line interface (CLI) command jaqal-emulate. This will yield both bit-string readouts (one per compiler pass of each measurement) and outcome probabilities (one per subcircuit- i.e., one per *written instance* of measure_all.

Circuit

```
from qscout.v1.std usepulses *

register q[2]

let pi2 1.5707963267948966
let pi4 0.7853981633974483

macro hadamard target { // A Hadamard gate can be implemented as
    Sy target           // a pi/2 rotation around Y
    Px target           // followed by a pi rotation around X.
}
macro cnot control target {  // CNOT implementation from Maslov (2017)
    Sy control
    MS control target 0 pi2
    <Sxd control | Sxd target>  // we can perform these in parallel
    Syd control
}

prepare_all
 hadamard q[0]
 cnot q[0] q[1]
measure_all

loop 10 {
 prepare_all
 hadamard q[0]
 cnot q[0] q[1]
 Px q[0]
 measure_all
}
```

Subcircuit

Subcircuit

QSCOUT_User_Kickoff_1 — -bash — 86×20

~/qscout-kmr/QSCOUT_User_Kickoff_1 — -bash

```
(venv-jaqal-3) s1013800:QSCOUT_User_Kickoff_1 kmrudin$ jaqal-emulate DemoJaqal1.jaqal
11
10
10
10
01
10
10
10
10
10
10
Subcircuit 0:
00: 0.5000000000000002
11: 0.5000000000000002
Subcircuit 1:
10: 0.5000000000000002
01: 0.5000000000000002
(venv-jaqal-3) s1013800:QSCOUT_User_Kickoff_1 kmrudin$
```

Command-line interface

QSCOUT

Can emulate any valid .jaqal file with (unitary) emulation by invoking the command line interface (CLI) command jaqal-emulate. This will yield both bit-string readouts (one per compiler pass of each measurement) and outcome probabilities (one per subcircuit- i.e., one per *written instance* of measure_all.

Circuit

```
from qscout.v1.std usepulses *

register q[2]

let pi2 1.5707963267948966
let pi4 0.7853981633974483

macro hadamard target { // A Hadamard gate can be implemented as
    Sy target           // a pi/2 rotation around Y
    Px target           // followed by a pi rotation around X.
}
macro cnot control target {  // CNOT implementation from Maslov (2017)
    Sy control
    MS control target 0 pi2
    <Sxd control | Sxd target>  // we can perform these in parallel
    Syd control
}

prepare_all
 hadamard q[0]
 cnot q[0] q[1]
measure_all

loop 10 {
 prepare_all
 hadamard q[0]
 cnot q[0] q[1]
 Px q[0]
 measure_all
}
```

Subcircuit

Subcircuit

Add “-s” to suppress individual bitstring readout

QSCOUT_User_Kickoff_1 — -bash — 107×14

~/qscout-kmr/QSCOUT_User_Kickoff_1 — -bash

```
(venv-jaqal-3) s1013800:QSCOUT_User_Kickoff_1 kmrudin$ jaqal-emulate DemoJaqal1.jaqal -s
Subcircuit 0:
00: 0.4999999999999999
11: 0.5000000000000001
Subcircuit 1:
10: 0.4999999999999999
01: 0.5000000000000001
(venv-jaqal-3) s1013800:QSCOUT_User_Kickoff_1 kmrudin$
```

Command-line interface

QSCOUT

Can emulate any valid .jaqal file with (unitary) emulation by invoking the command line interface (CLI) command jaqal-emulate. This will yield both bit-string readouts (one per compiler pass of each measurement) and outcome probabilities (one per subcircuit- i.e., one per *written instance* of measure_all.

Add "-s" to suppress individual bitstring readout

Circuit

```
from qscout.v1.std usepulses *

register q[2]

let pi2 1.5707963267948966
let pi4 0.7853981633974483

macro hadamard target { // A Hadamard gate can be implemented as
    Sy target           // a pi/2 rotation around Y
    Px target           // followed by a pi rotation around X.
}
macro cnot control target {  // CNOT implementation from Maslov (2017)
    Sy control
    MS control target 0 pi2
    <Sxd control | Sxd target>  // we can perform these in parallel
    Syd control
}

prepare_all
 hadamard q[0]
 cnot q[0] q[1]
measure_all

loop 10 {
 prepare_all
 hadamard q[0]
 cnot q[0] q[1]
 Px q[0]
 measure_all
}
```

Subcircuit

Subcircuit

QSCOUT_User_Kickoff_1 — -bash — 107×14

~/qscout-kmr/QSCOUT_User_Kickoff_1 — -bash

```
(venv-jaqal-3) s1013800:QSCOUT_User_Kickoff_1 kmrudin$ jaqal-emulate DemoJaqal1.jaqal -s
Subcircuit 0:
00: 0.4999999999999999
11: 0.5000000000000001
Subcircuit 1:
10: 0.4999999999999999
01: 0.5000000000000001
(venv-jaqal-3) s1013800:QSCOUT_User_Kickoff_1 kmrudin$
```

For help: jaqal-emulate --help

Additional use-cases

QSCOUT

Custom gates:

- E.g., "Cross-resonance" gate
- If it's not supported by QSCOUT native hardware, best bet is to write a macro defining your gate in terms of native QSCOUT operations.
- If you want the hardware to do something *different* physically, you'll want to use JaqalPaw. (Talk to us!)

More realistic noise models coming soon!

- Will use custom-built (QSCOUT-provided) objects for continuously parameterized gates that leverage actual experimental data.
- Functionality will be provided by new Python "package" interpygate (interpolation+python+gate).
- API TBD but will be available "soon."

Questions? Email us!

- kmrudin@sandia.gov
- bcamorr@sandia.gov
- arusso@sandia.gov

~~Back up~~ Backup Slides

QSCOUT

In [1]:
```
from jaqalpaq.parser import parse_jaqal_string
from jaqalpaq.emulator import run_jaqal_circuit
from jaqalpaq.core.result import parse_jaqal_output_list
```

A Familiar Circuit

In [2]:
```
jaqal_circuit = parse_jaqal_string("""
from qscout.v1.std usepulses *

register q[2]

let pi2 1.5707963267948966
let pi4 0.7853981633974483

macro hadamard target { // A Hadamard gate can be implemented as
    Sy target           // a pi/2 rotation around Y
    Px target           // followed by a pi rotation around X.
}
macro cnot control target {  // CNOT implementation from Maslov (2017)
    Sy control
    MS control target 0 pi2
    <Sxd control | Sxd target>  // we can perform these in parallel
    Syd control
}

prepare_all
 hadamard q[0]
 cnot q[0] q[1]
measure_all

loop 10 {
 prepare_all
 hadamard q[0]
 cnot q[0] q[1]
 Px q[0]
 measure_all
}

""")
```

Unitary Emulation

In [3]:
```
jaqal_result = run_jaqal_circuit(jaqal_circuit)
```

In [4]:
```
jaqal_result
```

Out[4]:
```
<jaqalpaq.core.result.ExecutionResult at 0x7f4adb61b610>
```

In [5]:
```
jaqal_result.readouts
```

Out[5]:
```
[<Readout 00 index 0 from 0>,
 <Readout 10 index 1 from 1>,
 <Readout 01 index 2 from 1>,
 <Readout 10 index 3 from 1>,
 <Readout 10 index 4 from 1>,
 <Readout 10 index 5 from 1>,
 <Readout 01 index 6 from 1>,
 <Readout 01 index 7 from 1>,
 <Readout 10 index 8 from 1>,
 <Readout 10 index 9 from 1>,
 <Readout 10 index 10 from 1>]
```

In [6]:
```
jaqal_result.readouts[1].as_int, jaqal_result.readouts[1].as_str
```

Out[6]:
```
(1, '10')
```

In [7]:
```
jaqal_result.subcircuits
```

Out[7]:
```
[<ProbabilisticSubcircuit 0@[7]>, <ProbabilisticSubcircuit 1@[8, 8]>]
```

In [8]:
```
jaqal_result.subcircuits[1].readouts
```

Out[8]:
```
[<Readout 10 index 1 from 1>,
 <Readout 01 index 2 from 1>,
 <Readout 10 index 3 from 1>,
 <Readout 10 index 4 from 1>,
 <Readout 10 index 5 from 1>,
 <Readout 01 index 6 from 1>,
 <Readout 01 index 7 from 1>,
 <Readout 10 index 8 from 1>,
 <Readout 10 index 9 from 1>,
 <Readout 10 index 10 from 1>]
```

In [9]:
```
jaqal_result.subcircuits[1].probability_by_int
```

Out[9]:
```
array([9.85601627e-33, 5.00000000e-01, 5.00000000e-01, 1.39383796e-32])
```

In [10]:
```
print('\n'.join([r.as_str for r in jaqal_result.readouts]))
```

```
00
10
01
10
10
10
01
01
10
10
10
```

In [11]:
```
data_from_qscout = """00
01
01
01
01
01
01
01
01
01
01"""
qscout_result = parse_jaqal_output_list(jaqal_circuit, data_from_qscout.split("\n"))
qscout_result.readouts
```

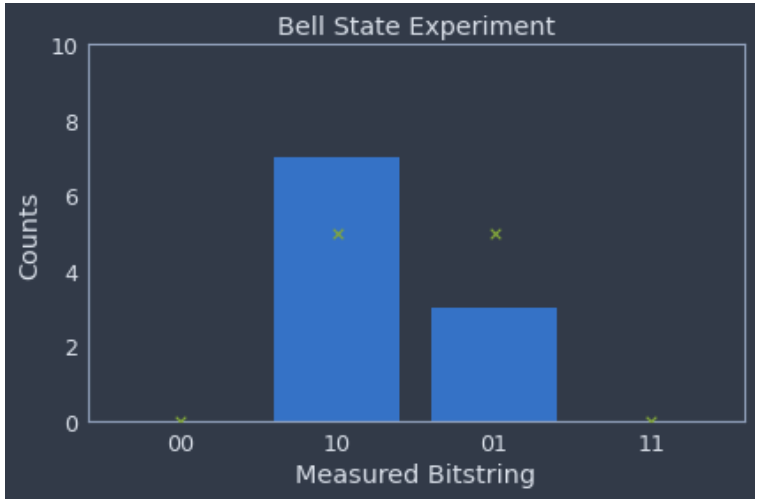
Out[11]:
```
[<Readout 00 index 0 from 0>,
 <Readout 01 index 1 from 1>,
 <Readout 01 index 2 from 1>,
 <Readout 01 index 3 from 1>,
 <Readout 01 index 4 from 1>,
 <Readout 01 index 5 from 1>,
 <Readout 01 index 6 from 1>,
 <Readout 01 index 7 from 1>,
 <Readout 01 index 8 from 1>,
 <Readout 01 index 9 from 1>,
 <Readout 01 index 10 from 1>]
```

Processing the Results

In [12]:
```
import matplotlib
from matplotlib import pyplot
from numpy import bincount
```

In [13]:

```
fig, ax = pyplot.subplots()
ax.set(ylim=(0,10))
data = [r.as_int for r in jaqal_result.subcircuits[1].readouts]
hist = bincount(data, None, 4)
probs = jaqal_result.subcircuits[1].probability_by_int
labels = [f"{n:02b}"[::-1] for n in range(4)]
ax.bar(range(4), height=hist, tick_label=labels)
ax.scatter(range(4), probs * 10, marker='x', zorder=3)
ax.set_xlabel('Measured Bitstring')
ax.set_ylabel('Counts')
ax.set_title('Bell State Experiment')
fig.tight_layout()
pyplot.show()
```



Noisy Density Matrix Simulation

In [14]:
```
from qscout.v1.noisy import SNLToy1
```

In [15]:
```
jaqal_result = run_jaqal_circuit(jaqal_circuit, backend=SNLToy1(2, depolarization=2e-3))
```

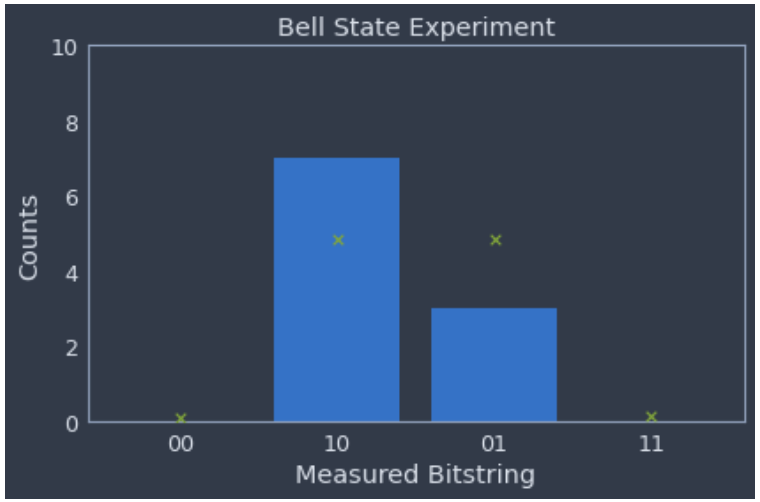
In [16]:
```
jaqal_result
```

Out[16]:
```
<jaqalpaq.core.result.ExecutionResult at 0x7f4adb3a1610>
```

In [17]:
```
jaqal_result.subcircuits[1].probability_by_int
```

Out[17]:
```
array([0.01231296, 0.48720558, 0.48672233, 0.01375913])
```

In [18]:
```
fig, ax = pyplot.subplots()
ax.set(ylim=(0,10))
data = [r.as_int for r in jaqal_result.subcircuits[1].readouts]
hist = bincount(data, None, 4)
probs = jaqal_result.subcircuits[1].probability_by_int
labels = [f"{n:02b}"[::-1] for n in range(4)]
ax.bar(range(4), height=hist, tick_label=labels)
ax.scatter(range(4), probs * 10, marker='x', zorder=3)
ax.set_xlabel('Measured Bitstring')
ax.set_ylabel('Counts')
ax.set_title('Bell State Experiment')
fig.tight_layout()
pyplot.show()
```



What is this object though?

In [19]:
```
con_model = SNLToy1(2, depolarization=2e-3)
```

In [20]:
```
con_model.__dict__.keys()
```

Out[20]:
```
dict_keys(['depolarization', 'rotation_error', 'phase_error', 'n_qubits', 'model', 'gate_durations'])
```

Parameters:

In [21]:
```
con_model.depolarization, con_model.rotation_error, con_model.phase_error
```

Out[21]:
```
(0.002, 0.01, 0.01)
```

A process matrix for every gate:

In [22]:
```
con_model.model
```

Out[22]:
```
<pygsti.objects.localnoisemodel.LocalNoiseModel at 0x7f4adb3858b0>
```

A duration for every gate:

In [23]:
```
con_model.gate_durations
```

Out[23]:
```
{'R': <bound method SNLToy1.gateduration_R of <qscout.v1.noisy.SNLToy1 object at 0x7f4adb3c0c10>>,
 'MS': <bound method SNLToy1.gateduration_MS of <qscout.v1.noisy.SNLToy1 object at 0x7f4adb3c0c10>>,
 'Rz': <bound method SNLToy1.gateduration_Rz of <qscout.v1.noisy.SNLToy1 object at 0x7f4adb3c0c10>>,
 'Rx': <bound method AbstractNoisyNativeEmulator._curry.<locals>._inner.<locals>.newop of <qscout.v1.n
oisy.SNLToy1 object at 0x7f4adb3c0c10>>,
 'Ry': <bound method AbstractNoisyNativeEmulator._curry.<locals>._inner.<locals>.newop of <qscout.v1.n
oisy.SNLToy1 object at 0x7f4adb3c0c10>>,
 'Px': <bound method AbstractNoisyNativeEmulator._curry.<locals>._inner.<locals>.newop of <qscout.v1.n
oisy.SNLToy1 object at 0x7f4adb3c0c10>>,
 'Py': <bound method AbstractNoisyNativeEmulator._curry.<locals>._inner.<locals>.newop of <qscout.v1.n
oisy.SNLToy1 object at 0x7f4adb3c0c10>>,
 'Pz': <bound method AbstractNoisyNativeEmulator._curry.<locals>._inner.<locals>.newop of <qscout.v1.n
oisy.SNLToy1 object at 0x7f4adb3c0c10>>,
 'Sx': <bound method AbstractNoisyNativeEmulator._curry.<locals>._inner.<locals>.newop of <qscout.v1.n
oisy.SNLToy1 object at 0x7f4adb3c0c10>>,
 'Sy': <bound method AbstractNoisyNativeEmulator._curry.<locals>._inner.<locals>.newop of <qscout.v1.n
oisy.SNLToy1 object at 0x7f4adb3c0c10>>,
 'Sz': <bound method AbstractNoisyNativeEmulator._curry.<locals>._inner.<locals>.newop of <qscout.v1.n
oisy.SNLToy1 object at 0x7f4adb3c0c10>>,
 'Sxd': <bound method AbstractNoisyNativeEmulator._curry.<locals>._inner.<locals>.newop of <qscout.v1.
noisy.SNLToy1 object at 0x7f4adb3c0c10>>,
 'Syd': <bound method AbstractNoisyNativeEmulator._curry.<locals>._inner.<locals>.newop of <qscout.v1.
noisy.SNLToy1 object at 0x7f4adb3c0c10>>,
 'Szd': <bound method AbstractNoisyNativeEmulator._curry.<locals>._inner.<locals>.newop of <qscout.v1.
noisy.SNLToy1 object at 0x7f4adb3c0c10>>,
 'Sxx': <bound method AbstractNoisyNativeEmulator._curry.<locals>._inner.<locals>.newop of <qscout.v1.
noisy.SNLToy1 object at 0x7f4adb3c0c10>>}
```

The model and gate durations are all we actually need.

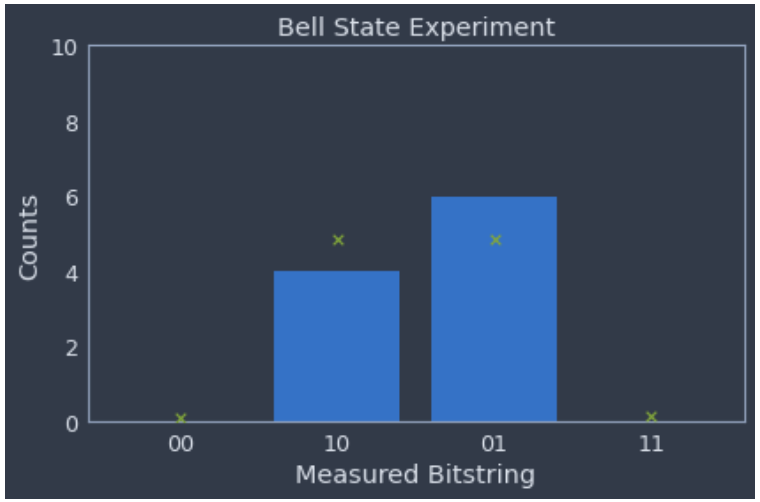
You can manually assemble them into with `CircuitEmulator`:

In [24]:
```
from jaqalpaq.emulator.pygsti import CircuitEmulator

backend = CircuitEmulator(model=con_model.model, gate_durations=con_model.gate_durations)
```

In [25]:
```
jaqal_result = run_jaqal_circuit(jaqal_circuit, backend=backend)
```

In [26]:

```
fig, ax = pyplot.subplots()
ax.set(ylim=(0,10))
data = [r.as_int for r in jaqal_result.subcircuits[1].readouts]
hist = bincount(data, None, 4)
probs = jaqal_result.subcircuits[1].probability_by_int
labels = [f"{n:02b}"[::-1] for n in range(4)]
ax.bar(range(4), height=hist, tick_label=labels)
ax.scatter(range(4), probs * 10, marker='x', zorder=3)
ax.set_xlabel('Measured Bitstring')
ax.set_ylabel('Counts')
ax.set_title('Bell State Experiment')
fig.tight_layout()
pyplot.show()
```



I want to make my own error model!

(Let's ignore crosstalk for now.)

You need:

- A process matrix for every gate
- A process matrix describing a qubit that is not being driven

- The duration of the gates

  You could do this by hand, but we have a helper class to make this a little easier:

In [27]:

```python
# Useful tools:
from numpy import abs, diag, pi, kron
import pygsti

# Description of the ideal gates
from qscout.v1.native_gates import U_R, U_Rz, U_MS

# This superclass will handle some details
from jaqalpaq.emulator.pygsti import AbstractNoisyNativeEmulator

# The superclass needs a description of the gates to handle
from qscout.v1.native_gates import NATIVE_GATES


class MyToyEmulator(AbstractNoisyNativeEmulator):
    # This tells AbstractNoisyNativeEmulator what gate set we're modeling:
    jaqal_gates = NATIVE_GATES

    def __init__(self, *args, **kwargs):
        """Builds a MyCustomEmulator instance for particular parameters

        :param depolarization float: (default 1e-3) The depolarization during
          one pi/2 gate.
        :param rotation_error float: (default 1e-2) The over-rotation angle during one
          pi/2 gate.
        :param phase_error: (default 1e-2) The error in the x-y angle for (non-Z)
          rotation gates.
        """

        # Equivalent to
        # self.depolarization = kwargs.pop('depolarization', 1e-3 )
        # ...
        self.set_defaults(
            kwargs, depolarization=1e-3, rotation_error=1e-2, phase_error=1e-2
        )

        # Pass through the balance of the parameters to AbstractNoisyNativeEmulator
        # In particular: passes the number of qubits to emulated (in args)
        super().__init__(*args, **kwargs)

    # For every gate, we need to specify a superoperator and a duration:

```

```
    # pyGSTi has gate names that start with capital G
    # To avoid name clashes, we prepend our gates with J:
    # GJR

    # We DO NOT currently support qubit-dependent error models (q will always be None)
    # Support for this is planned.  In the future, you will be able to write case
    # statements returning different process matrices for different qubits.
    def gateduration_R(self, q, axis_angle, rotation_angle):
        return abs(rotation_angle) / (pi / 2)

    def gate_R(self, q, axis_angle, rotation_angle):
        # We model the decoherence and over-rotation as a function of the gate duration:
        duration = self.gateduration_R(q, axis_angle, rotation_angle)

        # I.e., we scale the rotation and depolarization error by the time
        scaled_rotation_error = self.rotation_error * duration
        depolarization_term = (1 - self.depolarization) ** duration

        # Combine these all, returning a superoperator in the Pauli basis
        return pygsti.unitary_to_pauligate(
            U_R(axis_angle + self.phase_error, rotation_angle + scaled_rotation_error)
        ) @ diag([1, depolarization_term, depolarization_term, depolarization_term])

    # GJMS
    def gateduration_MS(self, q0, q1, axis_angle, rotation_angle):
        # Assume MS pi/2 gate 10 times longer than Sx, Sy, Sz
        return 10 * abs(rotation_angle) / (pi / 2)

    def gate_MS(self, q0, q1, axis_angle, rotation_angle):
        duration = self.gateduration_MS(q0, q1, axis_angle, rotation_angle)

        scaled_rotation_error = self.rotation_error * duration
        depolarization_term = (1 - self.depolarization) ** duration

        return pygsti.unitary_to_pauligate(
            U_MS(axis_angle + self.phase_error, rotation_angle + scaled_rotation_error)
        ) @ kron(diag([1] + 3*[depolarization_term]), diag([1] + 3*[depolarization_term]))


    # Rz is performed entirely in software.
    # GJRz
    def gateduration_Rz(self, q, angle):
        return 0
```

```python

    def gate_Rz(self, q, angle):
        return pygsti.unitary_to_pauligate(U_Rz(angle))

    # A process matrix for the idle behavior of a qubit.
    # Gidle
    def idle(self, q, duration):
        depolarization_term = (1 - self.depolarization) ** duration

        return diag([1, depolarization_term, depolarization_term, depolarization_term])


    # We'll model the Rx gate as just an R gate with a particular axis_angle:

    # GJRx
    def gateduration_Rx(self, q, rotation_angle):
        # return self.gateduration_R(q, 0, rotation_angle)
        return abs(rotation_angle) / (pi / 2)

    def gate_Rx(self, q, rotation_angle):
        # return self.gate_R(q, 0, rotation_angle)
        duration = self.gateduration_Rx(q, rotation_angle)

        scaled_rotation_error = self.rotation_error * duration
        depolarization_term = (1 - self.depolarization) ** duration

        return pygsti.unitary_to_pauligate(
            U_R(0 + self.phase_error, rotation_angle + scaled_rotation_error)
        ) @ diag([1, depolarization_term, depolarization_term, depolarization_term])


    # We could do this for all the other gates, but that would be tedious.
    # Instead, we'll fill in the parameters with some syntactic sugar:

    # Instead of copy-pasting the above definitions, use _curry to create new methods
    # with some arguments.  None is a special argument that means: require an argument
    # in the created function and pass it through.
    C = AbstractNoisyNativeEmulator._curry

    # For reference:
    # def gate_R(self, q, axis_angle, rotation_angle):

    # GJRy ... etc
```

```
    gateduration_Ry, gate_Ry = C((None, pi / 2, None), gateduration_R, gate_R)
    gateduration_Px, gate_Px = C((None, 0.0, pi), gateduration_R, gate_R)
    gateduration_Py, gate_Py = C((None, pi / 2, pi), gateduration_R, gate_R)
    gateduration_Pz, gate_Pz = C((None, pi), gateduration_Rz, gate_Rz)
    gateduration_Sx, gate_Sx = C((None, 0.0, pi / 2), gateduration_R, gate_R)
    gateduration_Sy, gate_Sy = C((None, pi / 2, pi / 2), gateduration_R, gate_R)
    gateduration_Sz, gate_Sz = C((None, pi / 2), gateduration_Rz, gate_Rz)
    gateduration_Sxd, gate_Sxd = C((None, 0.0, -pi / 2), gateduration_R, gate_R)
    gateduration_Syd, gate_Syd = C((None, pi / 2, -pi / 2), gateduration_R, gate_R)
    gateduration_Szd, gate_Szd = C((None, -pi / 2), gateduration_Rz, gate_Rz)
    gateduration_Sxx, gate_Sxx = C((None, None, 0.0, pi / 2), gateduration_MS, gate_MS)

    del C
```

In [28]:

```
backend = MyToyEmulator(2)
```

In [29]:

```
jaqal_result = run_jaqal_circuit(jaqal_circuit, backend=backend)
```

In [ ]:

```python
from qscout.v1.native_gates import NATIVE_GATES
from jaqalpaq.emulator.pygsti import AbstractNoisyNativeEmulator

class YourCustomEmulator(AbstractNoisyNativeEmulator):
    jaqal_gates = NATIVE_GATES

    def __init__(self, *args, **kwargs):
        self.set_defaults(
            kwargs, #your custom parameters
        )
        super().__init__(*args, **kwargs)

    # GJR
    def gateduration_R(self, q, axis_angle, rotation_angle):
        return # your model's duration

    def gate_R(self, q, axis_angle, rotation_angle):
        return # your error model's superoperator in the Pauli basis

    # Gidle
    def idle(self, q, duration):
        return # your model's process matrix acting for the idle qubit

    # the rest of the gates
```

QSCOUT Webinar Variational Quantum Algorithms

PRESENTED BY

Oliver Maupin

September 13, 2021

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Overview

- **Why Quantum Chemistry?**
- **The Variational Quantum Eigensolver (VQE) Algorithm**
- **Ground state energy of HeH^+**
  - Hamiltonian reduction
  - Ansatz
  - Energy functions
- **Hardware Results**
- **Q&A**

Why Quantum Chemistry?

Quantum Chemistry is a promising application for NISQ devices

- Quantum circuits are good at simulating quantum systems
- Calculations can be performed using low-depth circuits
- No need for fault-tolerant error correcting codes
- Classical literature gives us a good starting point

Computing the ground state energy

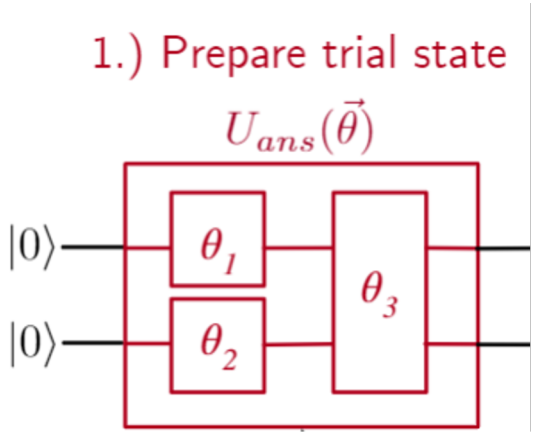
- Encode the electronic Hamiltonian to act on qubits
- Minimize the Hamiltonian as a cost function

How do we find the minimum energy state?

The Variational Quantum Eigensolver Algorithm (VQE)

1. State is prepared by a parameterized quantum circuit

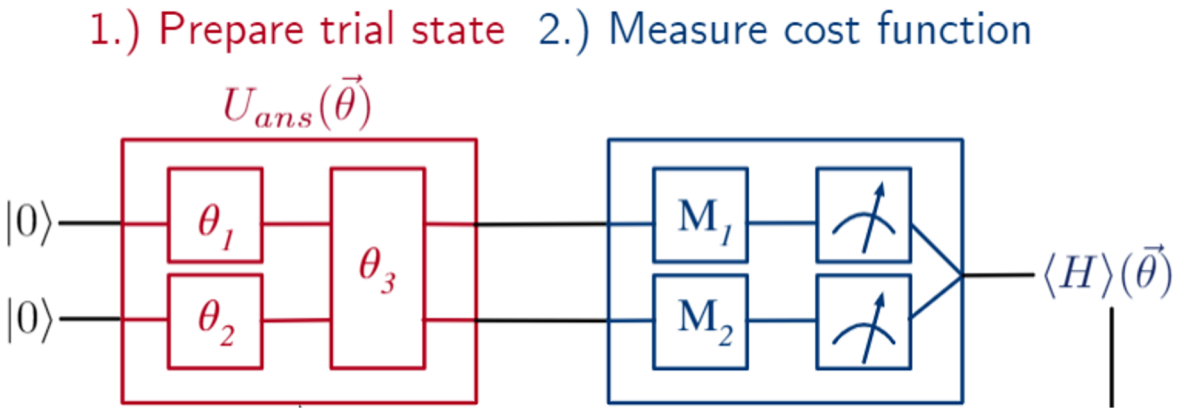
- Begin in the Hartree Fock state
- Act on the state with a circuit parameterized by angles (θ_1, θ_2, … θ_n) that transforms it into a state in the chosen *Ansatz*
- Unitary Coupled Cluster Ansatz for Single and Double Excitations (UCCSD)



The Variational Quantum Eigensolver Algorithm (VQE)

2. State is measured by a quantum circuit

- Create circuit for each term in the Hamiltonian
- Rotates qubits to Z basis depending on Pauli term
- Sample circuits repeatedly to determine expectation value for the energy

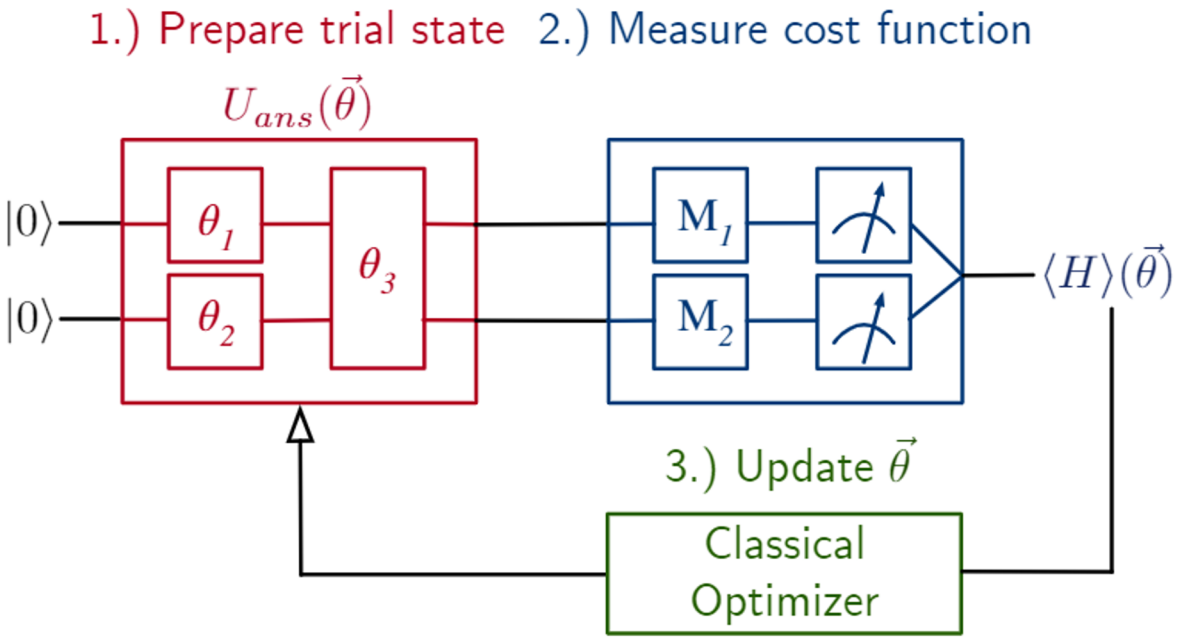


The Variational Quantum Eigensolver Algorithm (VQE)

3. Feed energy and angles into classical optimization loop

- Takes in energy and angles (θ_1, θ_2, … θ_n)
- Performs gradient descent search to find minimum
- Determines new angles (θ_1', θ_2', … θ_n')

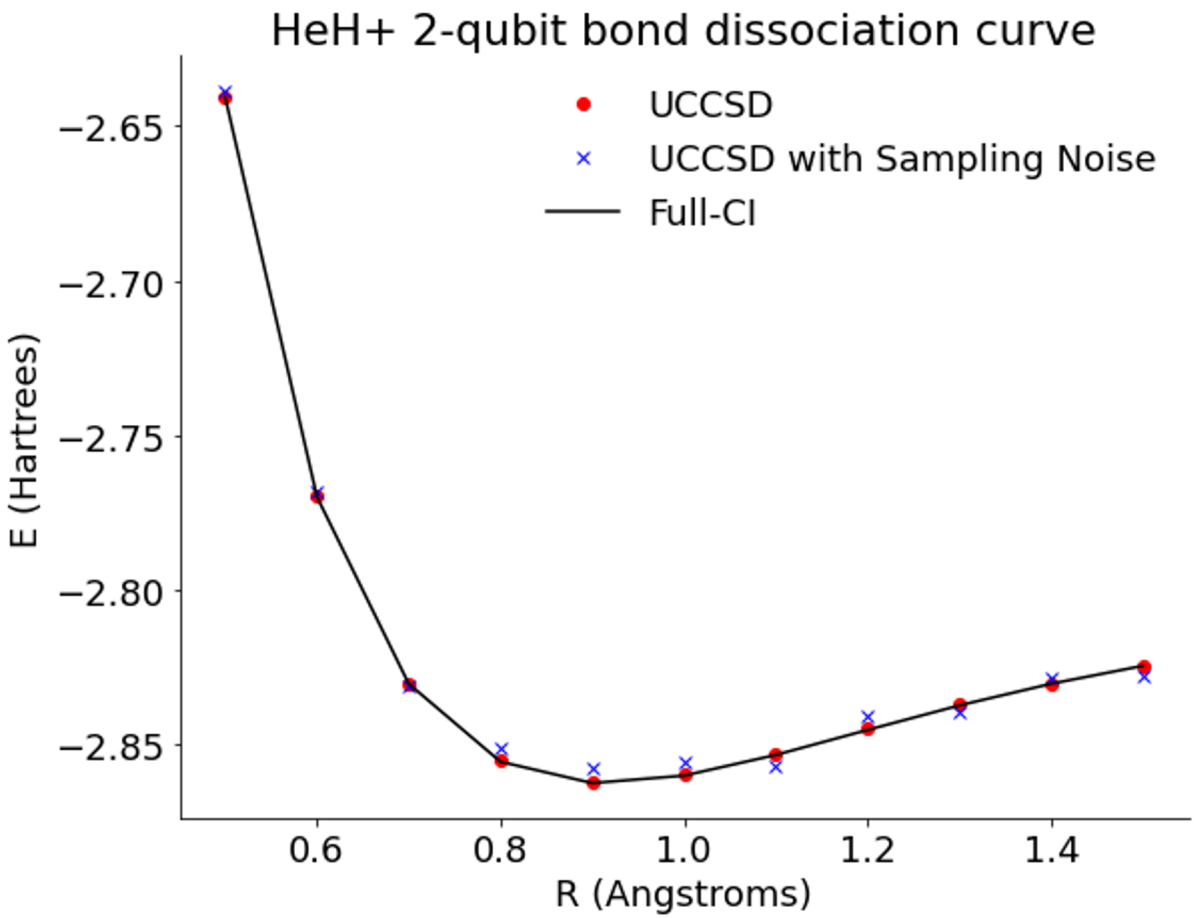
Repeat steps 1-3 until minimum is found



Ground state energy of HeH^+

Broad strokes of code

- Generate molecular data using PySCF
- Calculate the fermionic Hamiltonian using OpenFermion
- Encode and reduce the Hamiltonian to act on qubits
- Prepare the ansatz circuit
- Measure expectation values of Pauli terms
- Calculate energy
- Classical optimization loop
- Plot results!



Dissociation curve of HeH^+ for a tapered 2-qubit Hamiltonian.

Code Review

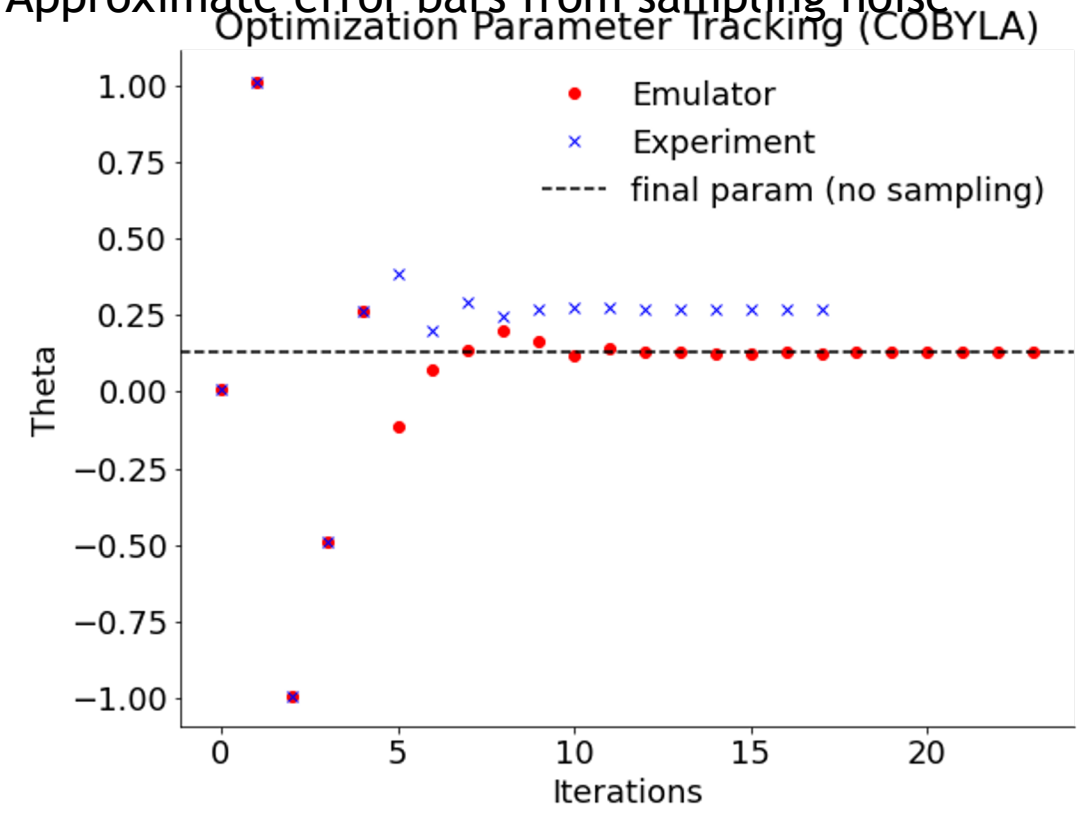
Hardware Results

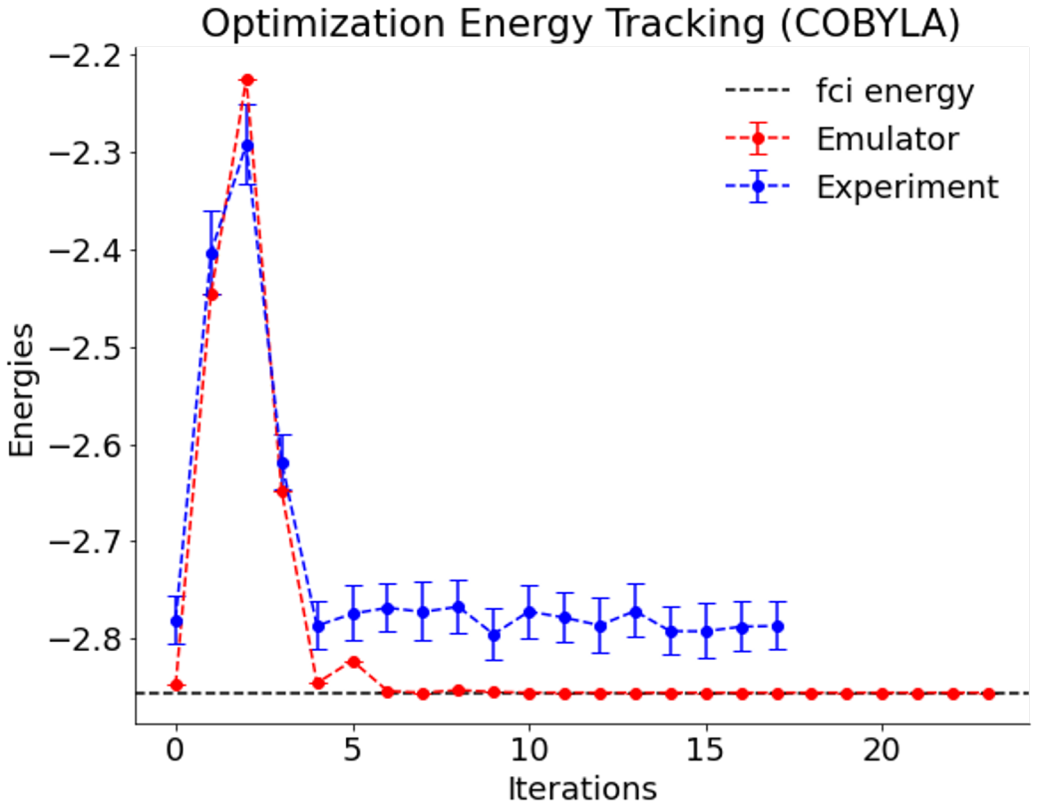
Result for HeH+ with r = 0.8Å separation

Ran on QSCOUT with 1000 shots per circuit

Tracked intermediate results of optimization, shown below

Approximate error bars from sampling noise





Questions?

Code Review

QSCOUT

Helium Hydride (Tapered HeH+) Exemplar

Step 0: Import various libraries

In [1]:

```
# Imports for QSCOUT
import jaqalpaq
from jaqalpaq.core import circuitbuilder
from jaqalpaq.core.circuit import normalize_native_gates
from jaqalpaq import emulator
from qscout.v1 import native_gates
from qscout.v1 import noisy

# Imports for basic mathematical functionality
from math import pi
import numpy as np

# Imports for OpenFermion(-PySCF)
import openfermion as of
from openfermion.chem import MolecularData
from openfermionpyscf import run_pyscf

# Import for VQE optimizer
from scipy import optimize
```

QSCOUT

Step 1: SCF calculation to assemble the second-quantized Hamiltonian

In [2]:

```
# Set the basis set, spin, and charge of the HeH+ molecule
basis = 'sto-3g'
multiplicity = 1
charge = 1 #Charge is 1 for HeH+

# Set calculation parameters
run_scf = 1
run_fci = 1
delete_input = True
delete_output = False

# Generate molecule at some bond length (0.8 Angstroms here)
geometry = [('He', (0., 0., 0.)), ('H', (0., 0., 0.8))]
molecule = MolecularData(geometry, basis, multiplicity, charge, filename='./HeH+_2_sto-3g_single_0.8') #Set file location of data

# Run pyscf to generate new molecular data for sto-3g HeH+
molecule = run_pyscf(molecule, run_scf=run_scf, run_fci=run_fci, verbose=False)

print("Bond Length in Angstroms: {}".format(0.8))
print("FCI (Exact) energy in Hartrees: {}".format(molecule.fci_energy))
```

Out [2]:

```
Bond Length in Angstroms: 0.8
FCI (Exact) energy in Hartrees: -2.8557086272280108
```

QSCOUT

Step 2: Convert the fermionic Hamiltonian to a qubit Hamiltonian

In [3]:

```
# Get the fermionic Hamiltonian for HeH+ and map it using the BK encoding
hamiltonian = molecule.get_molecular_hamiltonian()
hamiltonian_ferm = of.get_fermion_operator(hamiltonian)
hamiltonian_bk = of.symmetry_conserving_bravyi_kitaev(hamiltonian_ferm,
                                    active_orbitals=4, active_fermions=2)

# Define terms and coefficients of our Hamiltonian
terms = []
cs = [] #Coefficients
for term in hamiltonian_bk.terms:
    paulis = [None, None]
    for pauli in term:
        paulis[pauli[0]] = pauli[1]
    terms += [paulis]
    cs += [hamiltonian_bk.terms[term]]
```

QSCOUT

Step 3: Define UCC Ansatz circuit in JaqalPaq

In [4]:

```
def ansatz(theta, term):
    builder = circuitbuilder.CircuitBuilder(native_gates=normalize_native_gates(native_gates.NATIVE_GATES))

    # Create a qubit register
    q = builder.register('q', 2)

    # Prepare the Hartree Fock state
    builder.gate('prepare_all')
    builder.gate('Px', q[0])
    builder.gate('Px', q[1])

    # Apply the UCC Ansatz exp[-i*theta(X1 Y0)]
    builder.gate('MS', q[1], q[0], 0, np.pi/2)
    builder.gate('Rz', q[1], theta)
    builder.gate('MS', q[1], q[0], 0, -np.pi/2)

    # Change basis for measurement depending on term
    for j, qubit in enumerate(term):
        if qubit == 'X':
            builder.gate('Sy', ('array_item', q, j))
            builder.gate('Px', ('array_item', q, j))
        if qubit == 'Y':
            builder.gate('Sxd', ('array_item', q, j))

    builder.gate('measure_all')

    circuit = builder.build()

    return circuit
```

QSCOUT

Step 4: Define functions to calculate energy expectation value of Ansatz state

In [5]:

```
def ansatz_sampling(theta, sample_noise):
    term_probs = []
    for i in range(len(terms)):
        if sample_noise:
            probs = np.zeros(4)
            circuit = ansatz(theta, terms[i])
            sim_result = emulator.run_jaqal_circuit(circuit) #Run circuit on emulator
            sim_probs = sim_result.subcircuits[0].probability_by_int
            sample = np.random.choice(4, size=n_samples, p=sim_probs) #Sample from distribution to get probabilities

            for count in sample:
                probs[count] += 1 #Increment state counter
            probs = probs/n_samples #Determine probabilities from sampling
            term_probs += [probs] #Combine lists of probs of each term in Hamiltonian

        else: #Exact solution without sampling
            circuit = ansatz(theta, terms[i])
            sim_result = emulator.run_jaqal_circuit(circuit) #Run circuit on emulator
            sim_probs = sim_result.subcircuits[0].probability_by_int
            term_probs += [sim_probs]

    return term_probs
```

QSCOUT

In [6]:

```
# Calculate energy of one term of the Hamiltonian for one possible state
def term_energy(term, state, coefficient, prob):
    parity = 1
    for i in range(len(term)):
        #Change parity if state is occupied and is acted on by a pauli operator
        if term[i] != None and state[i] == '1':
            parity = -1*parity
    return coefficient*prob*parity

# Calculate energy of the molecule for a given value of theta
def calculate_energy(theta, sample_noise):
    energy = 0
    probs = ansatz_sampling(theta[0], sample_noise) #Convert tuple (from optimization) to float for circuit
    for i in range(len(terms)): #For each term in the hamiltonian
        for j in range(len(probs[0])): #For each possible state
            term = terms[i]
            state = '{0:02b}'.format(j)[::-1] #convert state to binary (# of qubits)
            coefficient = cs[i].real
            prob = probs[i][j]
            #print(term, state, coefficient, prob)
            energy += term_energy(term, state, coefficient, prob)
    return energy
```

QSCOUT

Step 5: Optimize Ground State Energy for given Bond Length

In [7]:

```
# Minimize the energy using classical optimization
n_samples = 10000
optimize.minimize(fun=calculate_energy, x0=[0.01], args=(False), method="COBYLA")
```

Out [7]:

```
     fun: -2.8553440209439667
   maxcv: 0.0
 message: 'Optimization terminated successfully.'
    nfev: 24
  status: 1
 success: True
       x: array([0.12747578])
```

QSCOUT

Step 6: Loop over previous steps to calculate ground state energy at different bond lengths

In [8]:

```
# Set the basis set, spin, and charge of the HeH+ molecule
basis = 'sto-3g'
multiplicity = 1
charge = 1 #Charge is 1 for HeH+

# Set calculation parameters
run_scf = 1
run_fci = 1
delete_input = True
delete_output = False

optimized_energies = [[], []]
error_bars = []
exact_energies = []

# Loop over bond lengths from 0.5 to 1.5 angstroms
n_samples = 10000 # Sample circuit
n_pts = 11 # Number of points
bond_lengths = np.linspace(0.5,1.5,n_pts)
for diatomic_bond_length in bond_lengths:
    # Generate molecule at some bond length
    geometry = [('He', (0., 0., 0.)), ('H', (0., 0., diatomic_bond_length))]
    description=str(round(diatomic_bond_length, 2))
    molecule = MolecularData(
        geometry, basis, multiplicity, charge,
        description=description,
        filename='./HeH+_2_sto-3g_single_dissociation_'+description)
```

QSCOUT

```python
# Run pyscf to generate new molecular data for sto-3g H2
molecule = run_pyscf(molecule, run_scf=run_scf, run_fci=run_fci, verbose=False)

# Get the fermionic Hamiltonian for HeH+ and map it using the BK encoding
hamiltonian = molecule.get_molecular_hamiltonian()
hamiltonian_ferm = of.get_fermion_operator(hamiltonian)
hamiltonian_bk = of.symmetry_conserving_bravyi_kitaev(hamiltonian_ferm, active_orbitals=4, active_fermions=2)

# Define terms and coefficients of our Hamiltonian
terms = []
cs = [] #Coefficients
for term in hamiltonian_bk.terms:
    paulis = [None, None]
    for pauli in term:
        paulis[pauli[0]] = pauli[1]
    terms += [paulis]
    cs += [hamiltonian_bk.terms[term]]

# Minimize the expectation value of the energy using a classical optimizer (SPSA)
exact_energies.append(molecule.fci_energy)
print("R={}\t Exact Energy: {}".format(str(round(diatomic_bond_length, 2)), molecule.fci_energy))

for i in range(2):
    result = optimize.minimize(fun=calculate_energy, x0=[0.01], args=(i), method="COBYLA")
    optimized_energies[i].append(result.fun)
    print("R={}\t Optimized Energy: {}\t Sampling Noise: {}".format(str(round(diatomic_bond_length, 2)), result.fun, bool(i)))
print('\n')
```

QSCOUT

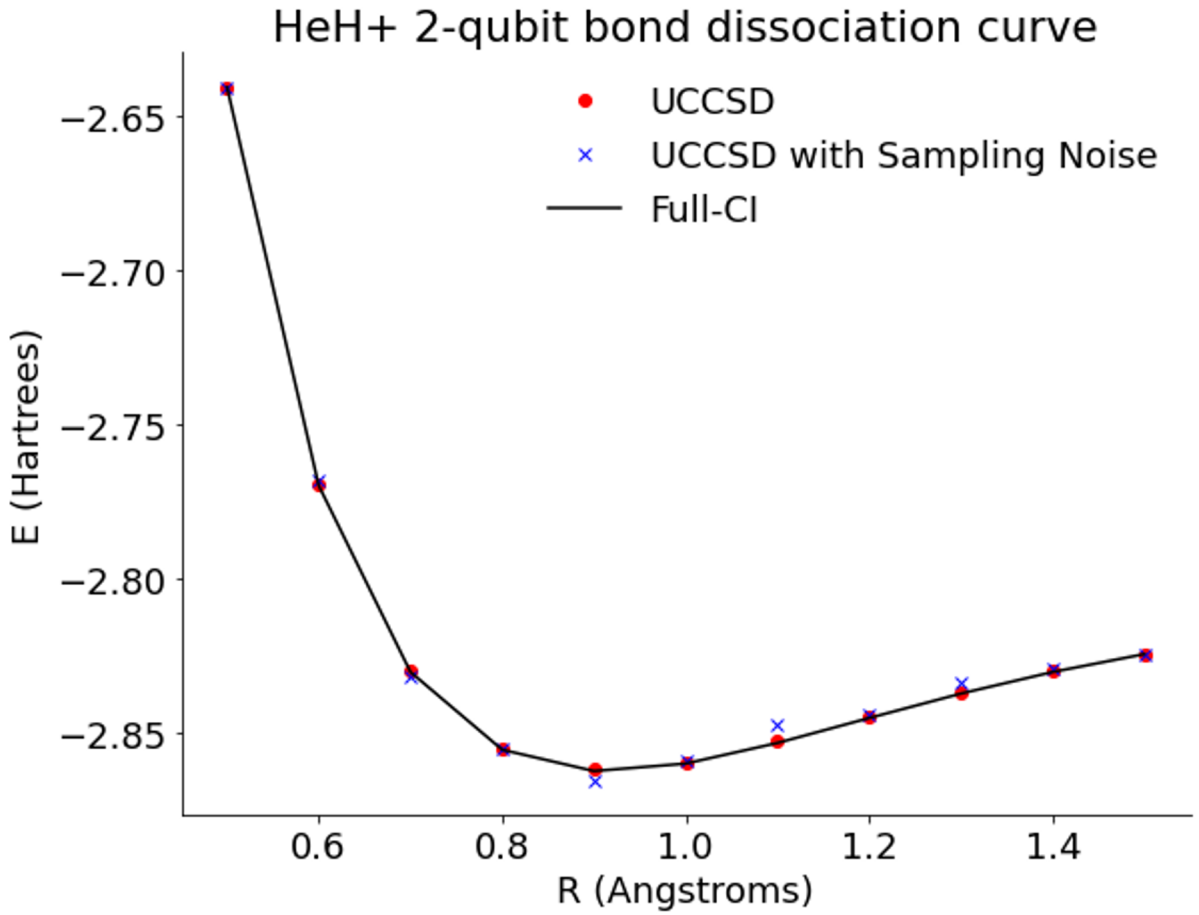
Step 7: Plot the dissociation curve

In [9]:

```
import matplotlib
import matplotlib.pyplot as pyplot

# Plot the various energies for different bond lengths
fig = pyplot.figure(figsize=(10,7))
pyplot.rcParams['font.size']=18
bkcolor = '#ffffff'
ax = fig.add_subplot(1, 1, 1)
pyplot.subplots_adjust(left=.2)
ax.set_xlabel('R (Angstroms)')
ax.set_ylabel(r'E (Hartrees)')
ax.set_title(r'HeH+ 2-qubit bond dissociation curve')
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
bond_lengths = [float(x) for x in bond_lengths]
ax.plot(bond_lengths, optimized_energies[0], 'o', label='UCCSD', color='red')
ax.plot(bond_lengths, optimized_energies[1], 'x', label='UCCSD with Sampling Noise', color='blue')
ax.plot(bond_lengths, exact_energies, '-', label='Full-CI', color='black')

ax.legend(frameon=False)
pyplot.show()
```

QSCOUT

out [9]:


HeH+ 2-qubit bond dissociation curve
UCCSD
UCCSD with Sampling Noise
Full-CI
E (Hartrees)
−2.65
−2.70
−2.75
−2.80
−2.85
0.6
0.8
1.0
1.2
1.4
R (Angstroms)