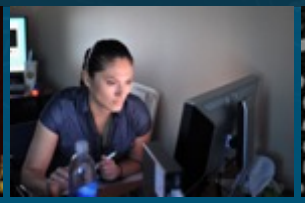
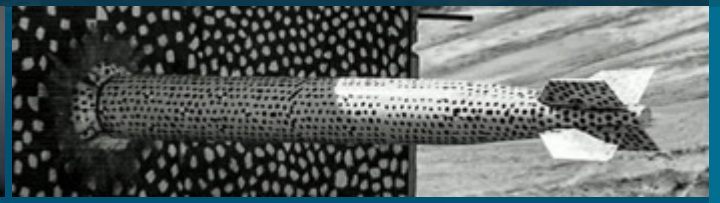
QSCOUT Webinar: Jaqal Overview





Presented by

Andrew J. Landahl

February 28, 2023

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 to the QSCOUT Software Stack Webinar!

Agenda (Times are in Mountain Daylight Time)

Day 1: (Tue 2/28/23)
10:00 am to 10:05 am: [05 mins] **Welcome & Introduction**, Susan Clark
10:05 am to 10:45 am: [40 mins] **Logistics & Jaqal Overview**, Andrew Landahl
10:45 am to 11:00 am: [15 mins] **Writing Jaqal code using JaqalPaq**, Jay Van Der Wall
11:00 am to 11:10 am: [10 mins] Break / Q&A
11:10 am to 12:00 pm: [50 mins] **SuperStaq software tools**, Ben Hall
12:00 pm to 12:10 pm: [10 mins] Break / Q&A
12:10 pm to 12:30 pm: [30 mins] **QSCOUT Emulator in JaqalPaq**, Antonio Russo, Kenny Rudinger, Benjamin Morrison
12:30 pm to 01:00 pm: [30 mins] Final interactive questions and answers for Day 1

Day 2: (Wed 3/1/23)
10:00 am to 10:45 am: [45 mins] **Batching & Subcircuits Overview**, Chris Yale
10:45 am to 10:55 am: [10 mins] Break / Q&A
10:55 am to 11:55 am: [60 mins] **JaqalPaw Overview**, Dan Lobser
11:55 am to 12:05 pm: [10 mins] Break / Q&A
12:05 pm to 12:35 pm: [30 mins] **JaqalPaw Exemplar Programs**, Matt Chow
12:35 pm to 01:00 pm: [25 mins] Final interactive questions and answers for Day 2

Our Round-Three Users

Authors	Institution(s)	Title
Kun Zhang and Vladimir Korepin	Stony Brook University	**Quantum search algorithms on NISQ processors**
Dmitri Kharzeev, Kazuki Ikeda, Adrien Florio, Kwangmin Yu	Stony Brook University, BNL	**Quantum Computing for Schwinger Model on NISQ computers**
Kwangmin Yu	BNL	**Optimal Quantum Pulse Control for Quantum Monte Carlo Integration on NISQ devices**
Jeffrey Young, Austin Adams, Akihiro Hayashi, Tom Conte, Eugene Dumitrescu, Chris Seck	Georgia Tech and ORNL	**Co-Design of Pulse-Level Control and Quantum Circuit Compilation For Trapped Ion Systems**
Marina Radulaski, Marina Marinkovic	UC Davis, ETH Zurich	**Open quantum system Tavis-Cummings modeling on trapped ion hardware**
Ho Lun Tang, Evangelos Piliouras, Sophia Economou, Edwin Barnes	Virginia Tech	**Globally optimal noise-robust gates for trapped ion computers**
Kathleen E. Hamilton, Tzula B. Propp,	ORNL and UNM	**Graph State Preparation in Decoherence-Free Subspaces on Trapped Ions**

Fun mix of algorithms, simulations, and quantum control

JaqalPaq 1.1 installation

Already installed JaqalPaq 1.0 and need to upgrade?

```
pip install -U jaqalpaq qscout-gatemodels
pip install -U jaqalpaq-extras
```

Need to install JaqalPaq 1.1 from scratch?

```
pip install --upgrade Cython numpy pip wheel
pip install JaqalPaq[pygsti-integration] QSCOUT-gatemodels
```

If you are within a virtual environment (strongly recommended) and use Jupyter, you can install the kernel with the commands:

```
pip install ipykernel
ipython kernel install --user --name=jaqal
```

More detailed instructions at https://pypi.org/project/JaqalPaq

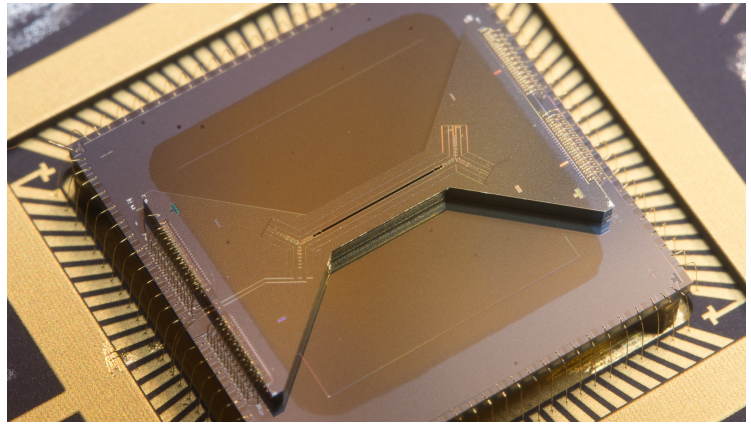
Part 1: Motivation

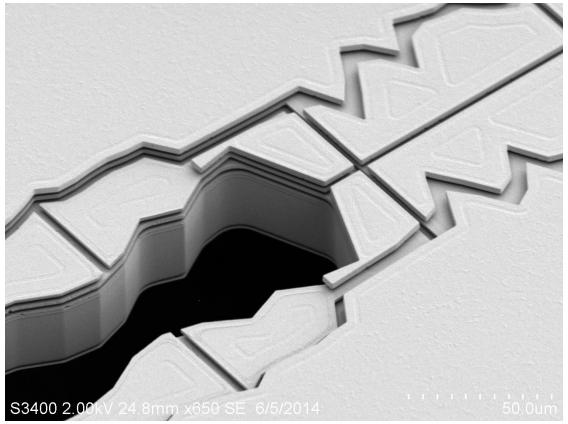
"Computer science is no more about computers than astronomy is about telescopes."

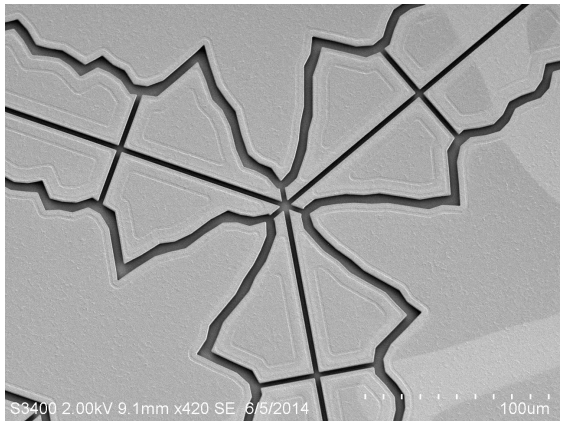
—*(Mis?)attributed to E. Dijkstra*

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.

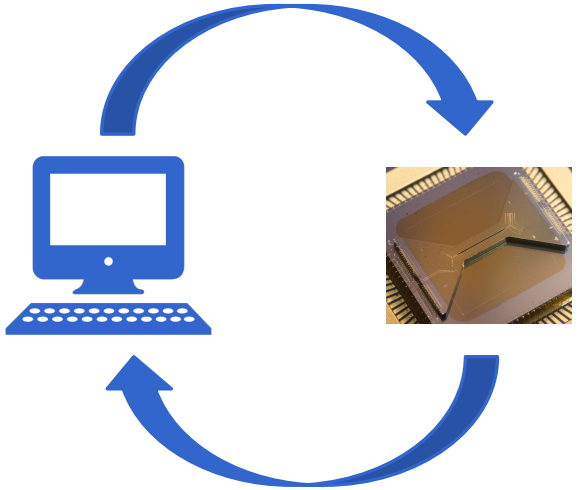
A purely quantum language?

Verboten

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

Why this madness?

- Ions are slow: Metaprogramming works!
- Why reinvent classical programming?
- Mid-circuit measurement language support will be available for next-gen QSCOUT.



Part 2: Ops specific to QSCOUT

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.

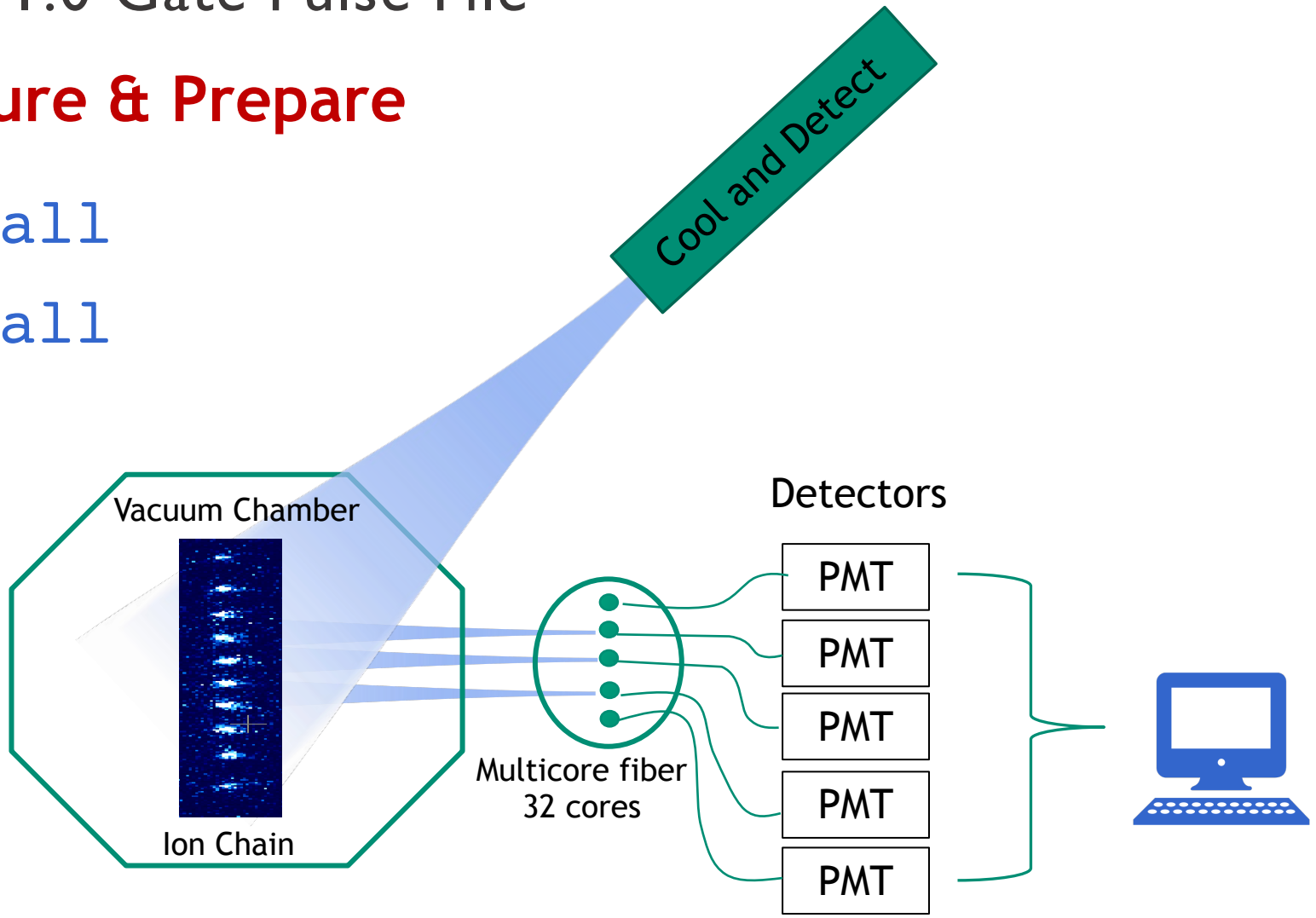
Examples

```
from qscout.v1.std usepulses *

from qscout.v1.stretched usepulses *

from my_favorite_tech.v1.std usepulses *
```

QSCOUT 1.0 Gate Pulse File

Gates: Measure & Prepare

```
prepare_all
measure_all
```



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

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.

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>
```

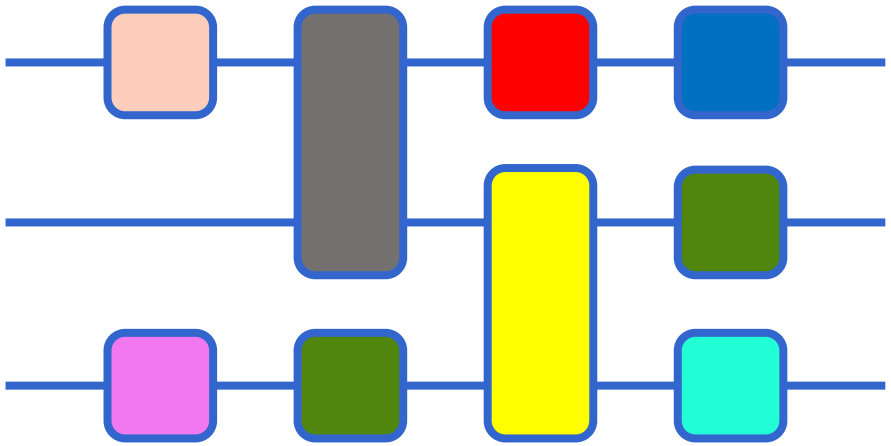
$$\mathrm{S}_{xx}(q) = \exp\left[-i\left(\frac{\pi}{4}\right)(X \otimes X)\right]$$

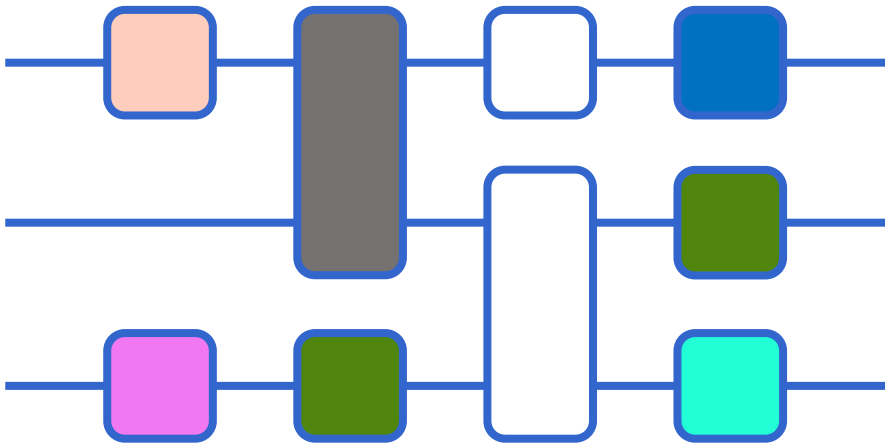
QSCOUT 1.0 Gate Pulse File

Gates: Idles with known timing (14 new gates)

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

Why?





Sneak Peek: JaqalPaw

```
    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 3: Jaqal syntax (cross-platform, in principle)

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.

Body statements

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.

Why use metaprogramming (like JaqalPaq, discussed next!)

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
}
```

Jaqal extras to be discussed in future webinar tutorials

Transpilation from other quantum assembly languages

- Discussed in SuperStaq tutorial.

Output format

- Discussed in emulator tutorial.

Subcircuits

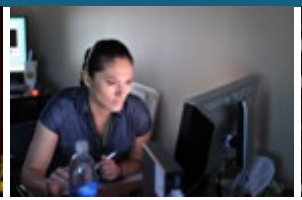
- Discussed in batch processing tutorial.

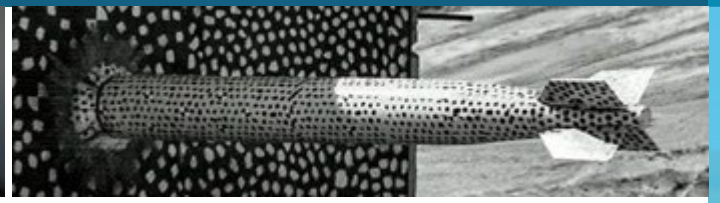
Creating new Jaqal instructions

- Discussed in JaqalPaw tutorial.

Questions?

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
- Emulation

Alternatives to Jaqalpaq

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

Scope of This Talk

Covered in this talk

- Simple circuits with Python string formatting
- Complex circuits with Qsyntax
- Displaying circuits for debugging or admiring

Covered in later talks

- Executing circuits (on emulator or hardware)
- Return value from executed circuits
- Batching and overriding parameters

Examples in this talk will assume execution in a Jupyter notebook

String Manipulation vs. Qsyntax

String manipulation (Python f-strings, `.format()`, `%` operator, etc) is an easy and intuitive way to dynamically produce Jaqal code.

This is very clear for simple examples

More complicated examples become harder to read

Rule of thumb: Once you need to concatenate strings it's time to use Qsyntax.

```
from qscout.v1.std usepulses *
register q[3]
let angle 1.234
subcircuit {
  Rx q[0] angle
  Rz q[1]
  MS q[0] q[1] 1.23 4.56
}
```

Example

```
jaqal = """
from qscout.v1.std usepulses *
register q[3]
let angle {}
subcircuit {{
  Rx q[0] angle
  Rz q[1]
  MS q[0] q[1] 1.23 4.56
}}""".format(1.234)
res = run_jaqal_string(jaqal)
```

Jaqalpaq Qsyntax Example

```
from math import pi
from jaqalpaq.qsyntax import circuit
@circuit
def rotate_theta(Q):
    Q.usepulses("qscout.v1.std")
    q = Q.register(1, name="q")
    theta = Q.let(pi / 2 , name="theta")
    with Q.subcircuit():
        Q.Rx(q[0], theta)
res = run_jaqal_circuit(rotate_theta())
```

```
from qscout.v1.std usepulses *
register q[1]
let theta 1.5707963267948966
subcircuit {
  Rx q[0] theta
}
```

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

Parallel Gates and Loops

```
from jaqalpaq.qsyntax import circuit
@circuit
def parallel_rotate(Q):
    Q.usepulses("qscout.v1.std")
    q = Q.register(2, name="q")
    with Q.subcircuit():
        with Q.parallel():
            Q.Rx(q[0], 1.234)
            Q.Rx(q[1], -0.123)
        with Q.loop(10):
            Q.Rz(q[0], 0.543)
```

```
from qscout.v1.std usepulses *
register q[2]
subcircuit {
  <
    Rx q[0] 1.234
    Rx q[1] -0.123
  >
  loop 10 {
    Rz q[0], 0.543
  }
}
```

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

Using Python Functions

```
from jaqalpaq.qsyntax import circuit
def rotate_half(Q, qubit, angle):
    Q.Rx(qubit, angle / 2)
@circuit
def functions(Q):
    Q.usepulses("qscout.v1.std")
    q = Q.register(1, name="q")
    theta = 1.234
    with Q.subcircuit():
        rotate_half(Q, q[0], theta)
```

```
from qscout.v1.std usepulses *
register q[1]
subcircuit {
  Rx q[0] 0.617
}
```

Jaqal macros and map statements are redundant and not supported in Qsyntax

Parameterizing The Entire Circuit

```
from jaqalpaq.qsyntax import circuit
@circuit
def param_circuit(Q, qsize, theta):
    Q.usepulses("qscout.v1.std")
    q = Q.register(qsize, name="q")
    with Q.subcircuit():
        for i in range(0, qsize, 2):
            Q.Rx(q[i], theta)
        for i in range(qsize - 1):
            Q.MS(q[i], q[i+1], 0.12, 0.34)
run_jaqal_circuit(param_circuit(4, 1.23))
```

```
// qsize=4, theta=1.23
from qscout.v1.std usepulses *
register q[4]
subcircuit {
  Rx q[0] 1.23
  Rx q[2] 1.23
  MS q[0] q[1] 0.12 0.34
  MS q[1] q[2] 0.12 0.34
  MS q[2] q[3] 0.12 0.34
}
```

Debugging with the Generator

```
from jaqalpaq.qsyntax import circuit
from jaqalpaq.generator import generate_jaqal_program
@circuit
def debug_me(Q):
    Q.usepulses("qscout.v1.std")
    q = Q.register(1, name="q")
    theta = Q.let(1.234, name="theta")
    with Q.subcircuit():
        Q.Rx(q[0], theta)
print(generate_jaqal_program(debug_me()))
```

```
from qscout.v1.std usepulses *

let theta 1.234

register q[1]

subcircuit {
  Rx q[0] theta
}
```

Notice the order of header statements may not be the same, but the program will be equivalent.

For More Information

Code used to build a circuit:

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

Example circuits and notebooks:

- https://gitlab.com/jaqal/jaqalpaq/-/tree/master/examples

Online documentation (auto-generated):

- https://jaqalpaq.readthedocs.io/

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

SupersteaQ for QSCOUT Users

February 28, 2023

Ben Hall, Pranav Gokhale, Victory Omole, Rich Rines, Andrew Litteken, Stephanie Lee

Jupyter Notebook Demonstration

Follow along at: **https://hub.super.tech**
Documentation available at: **https://docs-superstaq.readthedocs.io/**

1. Sign up for an account and log in.
2. Go to this URL: **https://tinyurl.com/2p9dtyxj**
3. Update the token variable.

Remember to manually save your changes!

Outline

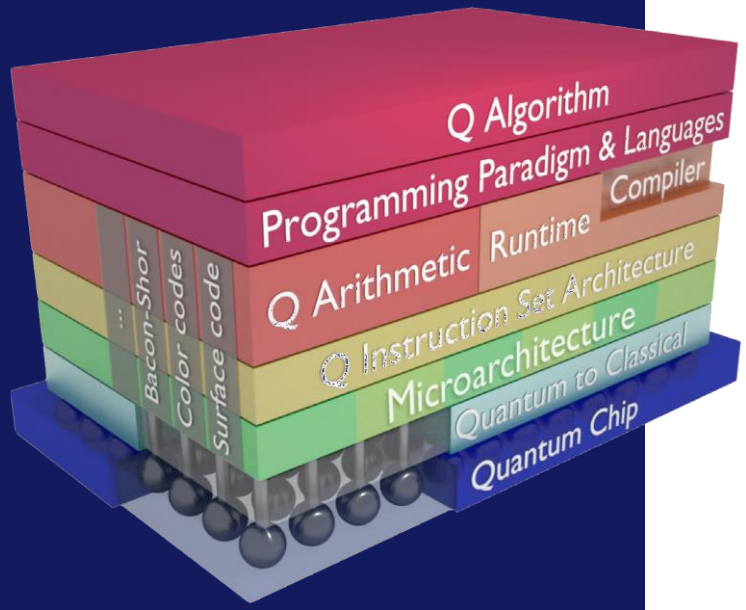
- Introduction to SuperstaQ

- QSCOUT Compilation via SuperstaQ

- Example 1: Entangling Basis Compilation

- Example 2: SWAP Mirroring

- Questions



Cirq

OpenAPI / OpenQASM

Qiskit

ANY
LANGUAGE

SuperstaQ

ANY
HARDWARE

AQT

IonQ

IBM

QSCOUT

Rigetti

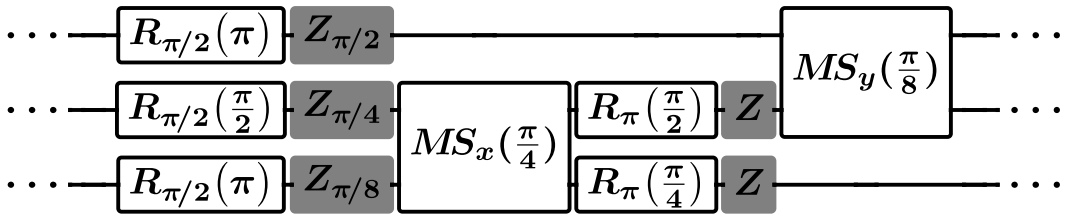
QSCOUT Compilation via SuperstaQ

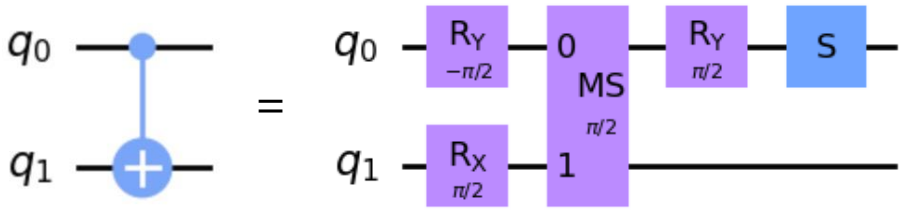
Endpoint: qscout_compile()

- Input: arbitrary quantum circuits (submitted via Cirq/Qiskit/OpenQASM)
- Output: Device-compiled circuits (Cirq/Qiskit/OpenQASM), and Jaqal programs

Gate set:

- Arbitrary X/Y rotations
- Virtual Z rotations
- Configurable entangler:
  - XX/YY (i.e. MS_{PEC}) or ZZ as basis rotation
  - Parameterized (or discrete) rotations
- All-to-all connectivity





Notebook Example: QSCOUT Compilation via SuperstaQ

```python
# compile with qscout compile
compiler_output = service.qscout_compile(circuit1)
print(compiler_output.circuit)
```

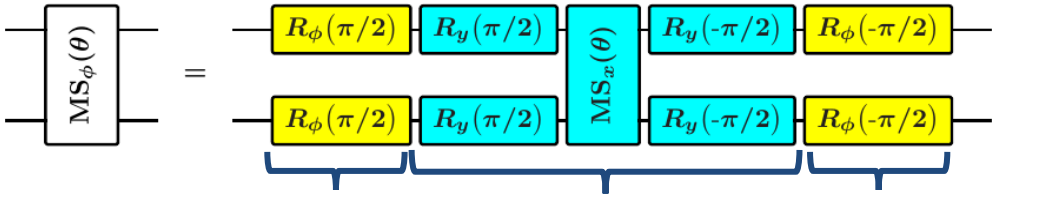
```
0: ──PhX(-0.5)^0.5──MS(-0.113π)──PhX(0.5)^0.5──Z──M──
                    │                          │
1: ──PhX(0.5)^0.5──MS(-0.113π)──PhX(-0.5)^0.5──Z──M──
```

```python
# get jaqal program
print(compiler_output.jaqal_program)
```

```
from qscout.v1.std usepulses *

register allqubits[2]

prepare_all
<
        R allqubits[0] 1.5707963267948966 1.5707963267948966
        R allqubits[1] -1.5707963267948966 1.5707963267948966
>
MS allqubits[0] allqubits[1] 0 -0.6522200685959443
<
        R allqubits[0] -1.5707963267948966 1.5707963267948966
        R allqubits[1] 1.5707963267948966 1.5707963267948966
>
<
        Rz allqubits[0] -3.141592653589793
        Rz allqubits[1] -3.141592653589793
>
measure_all
```

Entangling Basis Compilation

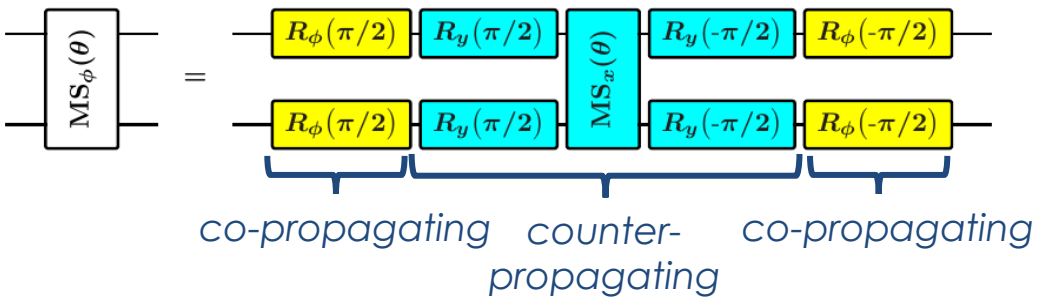
QSCOUT MS gate implementation:



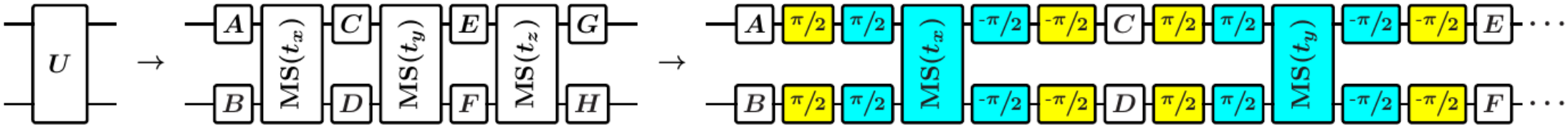
co-propagating *counter-propagating* *co-propagating*

Entangling Basis Compilation

QSCOUT MS gate implementation:

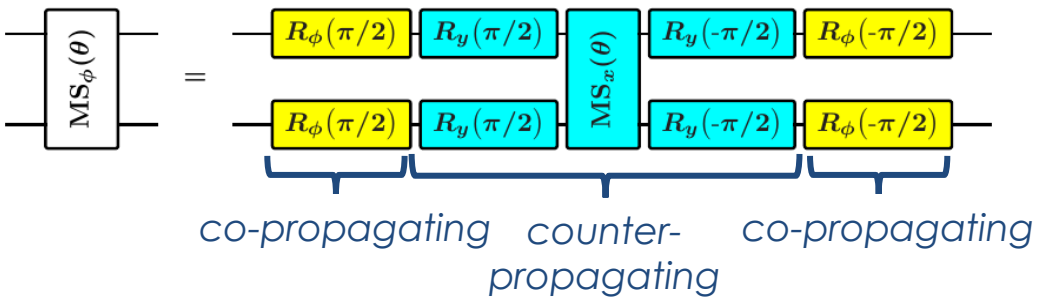


Standard TIQC compilation:

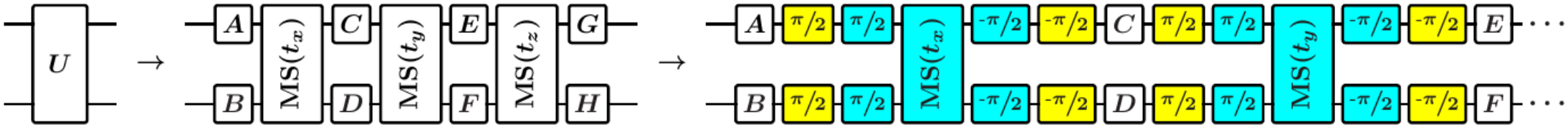


Entangling Basis Compilation

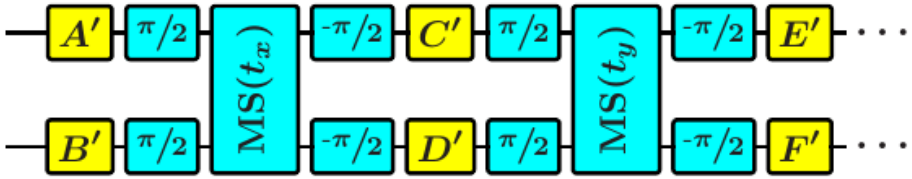
QSCOUT MS gate implementation:



Standard TIQC compilation:

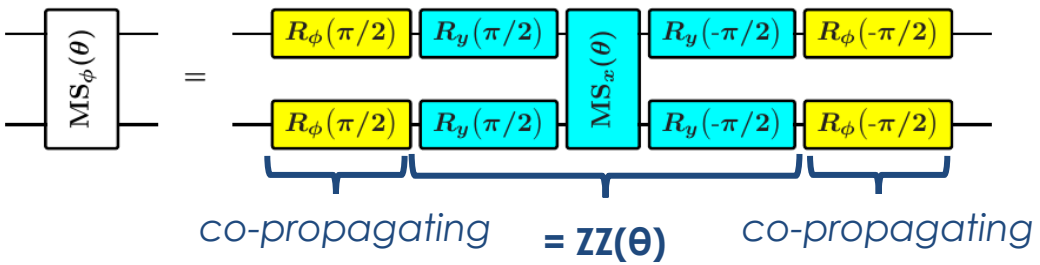


⟶ *Better: merge co-propagating gates:*

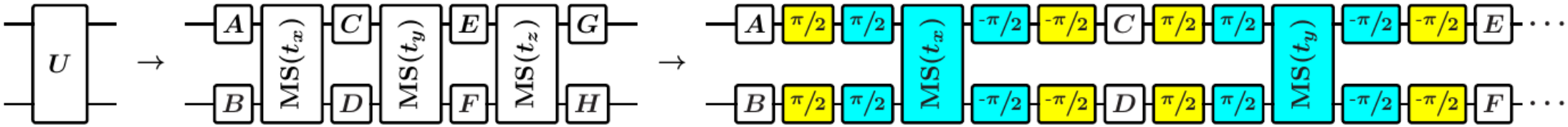


Entangling Basis Compilation

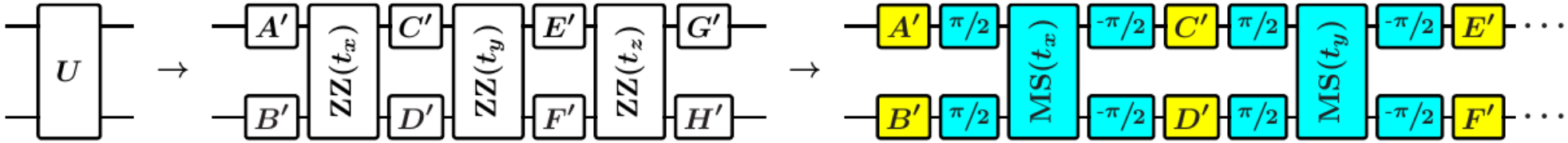
QSCOUT MS gate implementation:



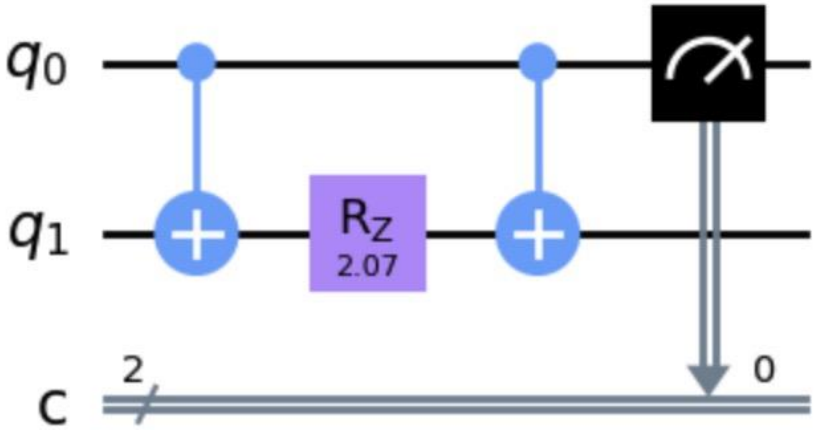
Standard TIQC compilation:

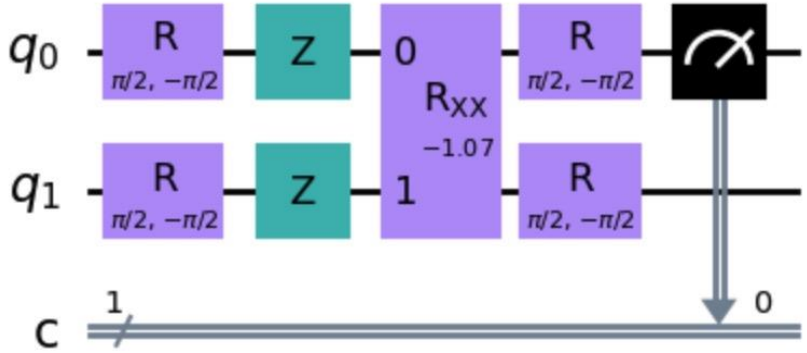


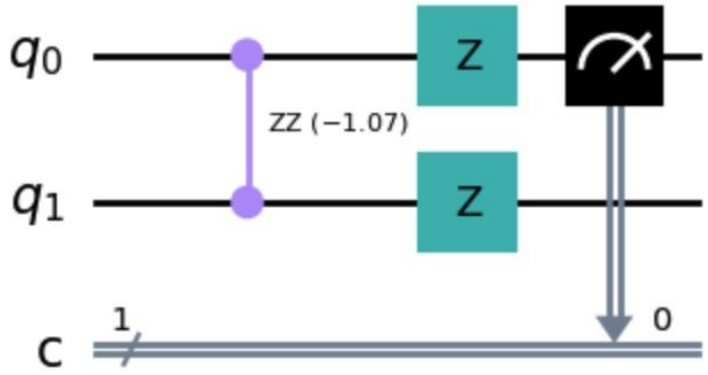
ZZ-basis compilation (via SuperstaQ):



Notebook Example: Entangling Basis Compilation







SWAP Mirroring

Sometimes (*SWAP* · *gate*) is easier to implement than *gate*

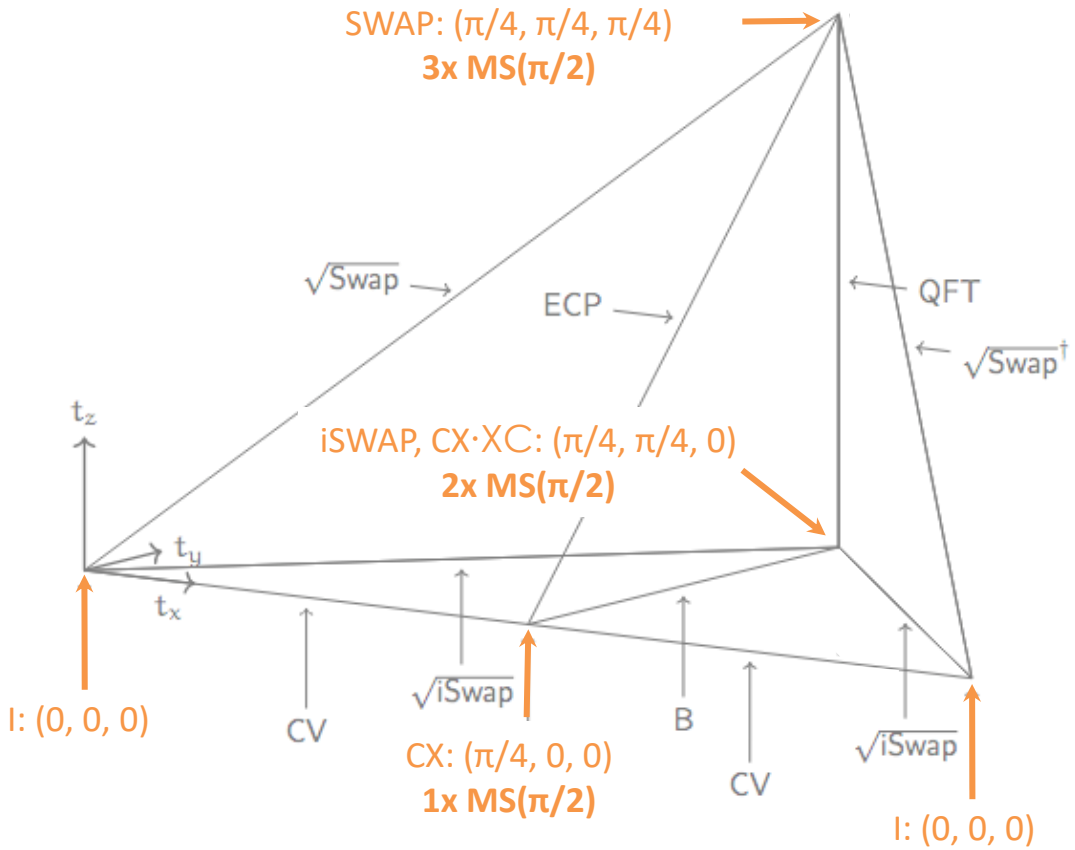
Strategy: insert (*SWAP*·*SWAP*) = 1 into circuit

⟶ Absorb the 1st SWAP into the preceding gate

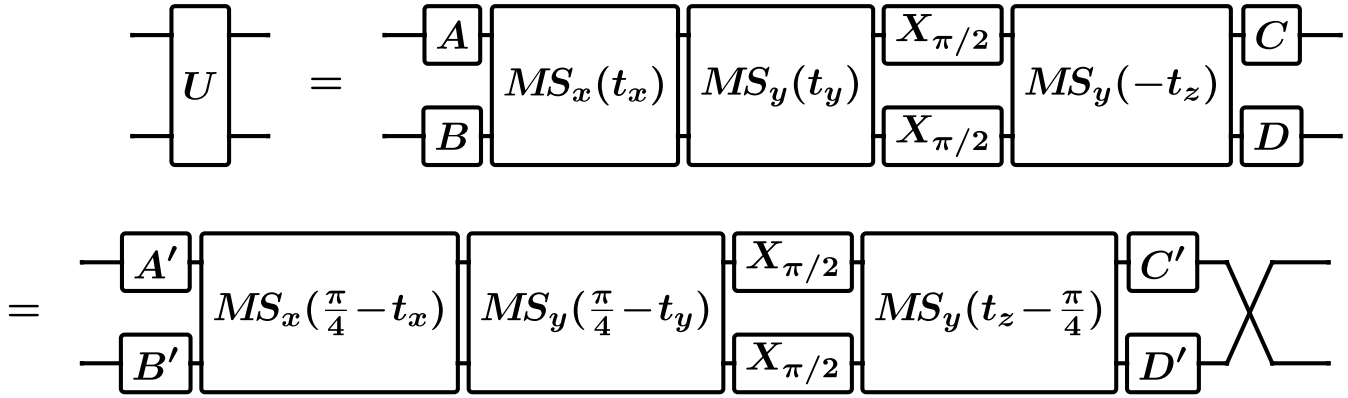
⟶ Implement the 2nd "virtually" (by relabeling qubits)

SWAP Mirroring

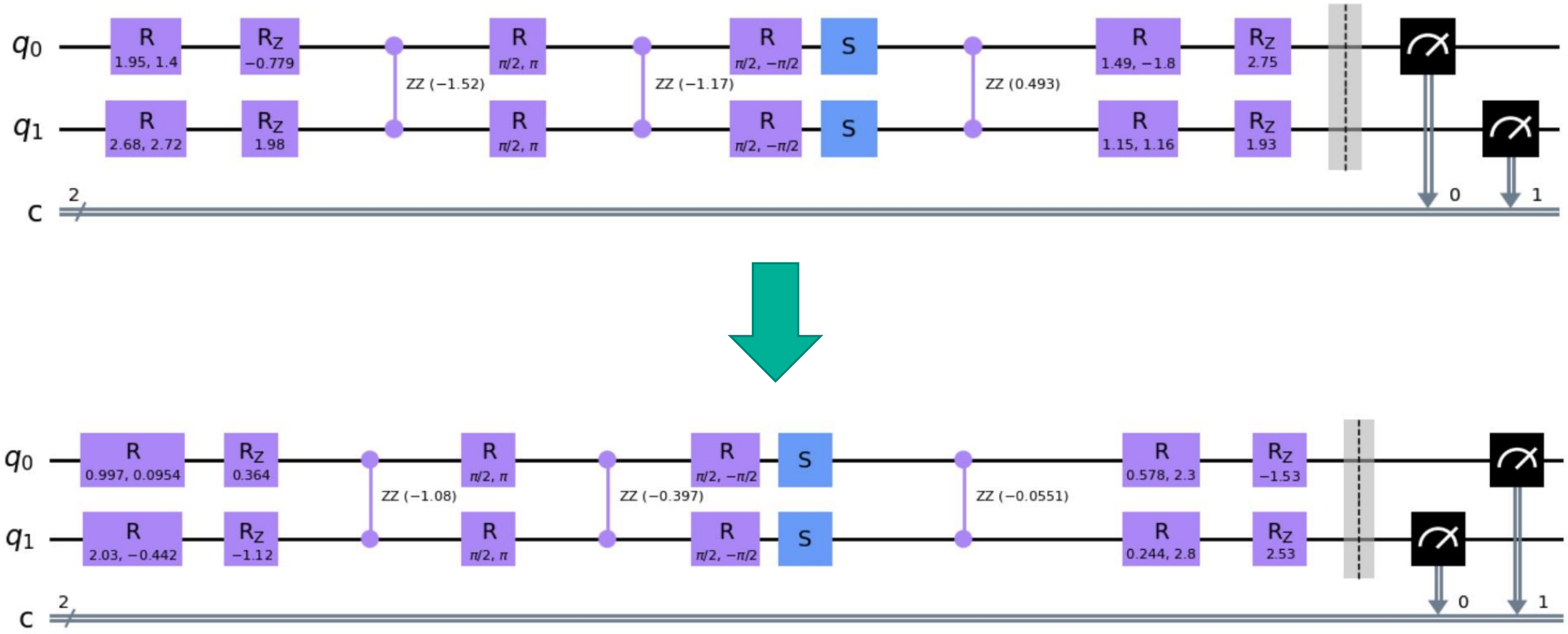
Cartan decomposition in SU(4): $$U = \left(u_3 \otimes u_4\right) \underbrace{\exp\left\{it_x XX + it_y YY + it_z ZZ\right\}}_{\text{entangling part. } |t_z| \le t_y \le t_x \le \pi/4} \left(u_1 \otimes u_2\right)$$



Local invariants (t_x, t_y, t_z) determine required MS pulses:



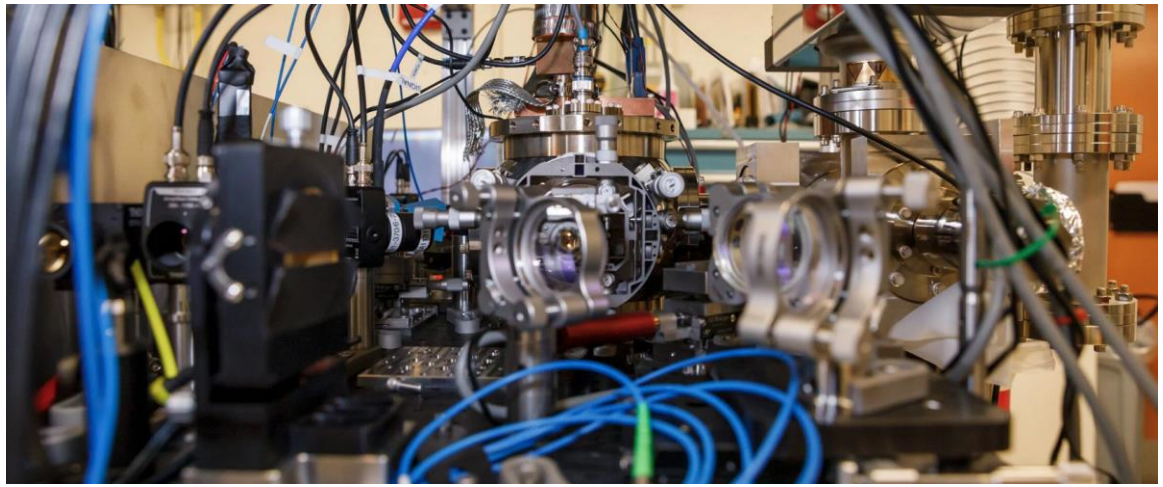
Notebook Example: SWAP Mirroring



Simulator

```
superstaq = qss.superstaq_provider.SuperstaQProvider()
backend = superstaq.get_backend("sandia_qscout_qpu")
qc = qiskit.QuantumCircuit(2, 2)
qc.h(0)
qc.cx(0, 1)
qc.measure(0, 0)
qc.measure(1, 1)
job = backend.run(qc, shots=100, method="dry-run")
print(job.result().get_counts())
```

```
{'00': 49, '11': 51}
```



Questions?

For any additional questions, feedback, or feature requests, please email **info@super.tech**

Infleqtion

SUPER.TECH

ColdQuanta

Using the Jaqal Emulator

PRESENTED BY

Benjamin Morrison, Kenneth Rudinger, Antonio Russo

February 28, 2023

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

- 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

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
}

subcircuit {
 hadamard q[0]
 cnot q[0] q[1]
}


subcircuit    {
 hadamard q[0]
 cnot q[0] q[1]
 Px q[0]
}
```

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
}

subcircuit {
 hadamard q[0]
 cnot q[0] q[1]
}


subcircuit    {
 hadamard q[0]
 cnot q[0] q[1]
 Px q[0]
}
```

Subcircuit

A *circuit* is the whole Jaqal file. A *subcircuit* is a portion of code contained in the `subcircuit` block. (Long-time Jaqal users: This is equivalent to a block delimited 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
}

subcircuit {
 hadamard q[0]
 cnot q[0] q[1]
}


subcircuit    {
 hadamard q[0]
 cnot q[0] q[1]
 Px q[0]
}
```

Subcircuit

Subcircuit

A *circuit* is the whole Jaqal file. A *subcircuit* is a portion of code contained in the `subcircuit` block. (Long-time Jaqal users: This is equivalent to a block delimited 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 outcome probabilities for each subcircuit.

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 outcome probabilities for each subcircuit.

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
}

subcircuit {
 hadamard q[0]
 cnot q[0] q[1]
}


subcircuit    {
 hadamard q[0]
 cnot q[0] q[1]
 Px q[0]
}
```

Subcircuit

Subcircuit

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 outcome probabilities for each subcircuit.

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
}

subcircuit {
 hadamard q[0]
 cnot q[0] q[1]
}


subcircuit    {
 hadamard q[0]
 cnot q[0] q[1]
 Px q[0]
}
```

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
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

Can emulate any valid .jaqal file with (unitary) emulation by invoking the command line interface (CLI) command `jaqal-emulate`. This will yield outcome probabilities for each subcircuit.

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
}

subcircuit {
 hadamard q[0]
 cnot q[0] q[1]
}


subcircuit    {
 hadamard q[0]
 cnot q[0] q[1]
 Px q[0]
}
```

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
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!)

Questions? Email us!

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

~~Back up~~ Backup Slides

[1]:
```python
from jaqalpaq.parser import parse_jaqal_string
from jaqalpaq.run import run_jaqal_circuit
from jaqalpaq.core.result import parse_jaqal_output_list
```

1 A Familiar Circuit

[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
}

subcircuit {
 hadamard q[0]
```

```
 cnot q[0] q[1]
}

subcircuit {
 hadamard q[0]
 cnot q[0] q[1]
 Px q[0]
}

""")
```

2 Unitary Emulation

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

[4]:
```
jaqal_result
```

[4]:
```
<jaqalpaq.core.result.ExecutionResult at 0x85aa69c9ca30>
```

[5]:
```
jaqal_result.subcircuits
```

[5]:
```
[<EmulatorSubcircuit 0@[7]>, <EmulatorSubcircuit 1@[16]>]
```

[6]:
```
jaqal_result.subcircuits[0].probability_by_int
```

[6]:
```
array([5.00000000e-01, 1.54074396e-32, 6.16297582e-33, 5.00000000e-01])
```

[7]:
```
jaqal_result.subcircuits[1].probability_by_int
```

[7]:
```
array([7.08490623e-33, 5.00000000e-01, 5.00000000e-01, 8.03767555e-33])
```

[8]:
```
jaqal_result.subcircuits[0].simulated_probability_by_int
```

[8]:
```
array([5.00000000e-01, 1.54074396e-32, 6.16297582e-33, 5.00000000e-01])
```

[9]:
```
jaqal_result.subcircuits[0].relative_frequency_by_int
```

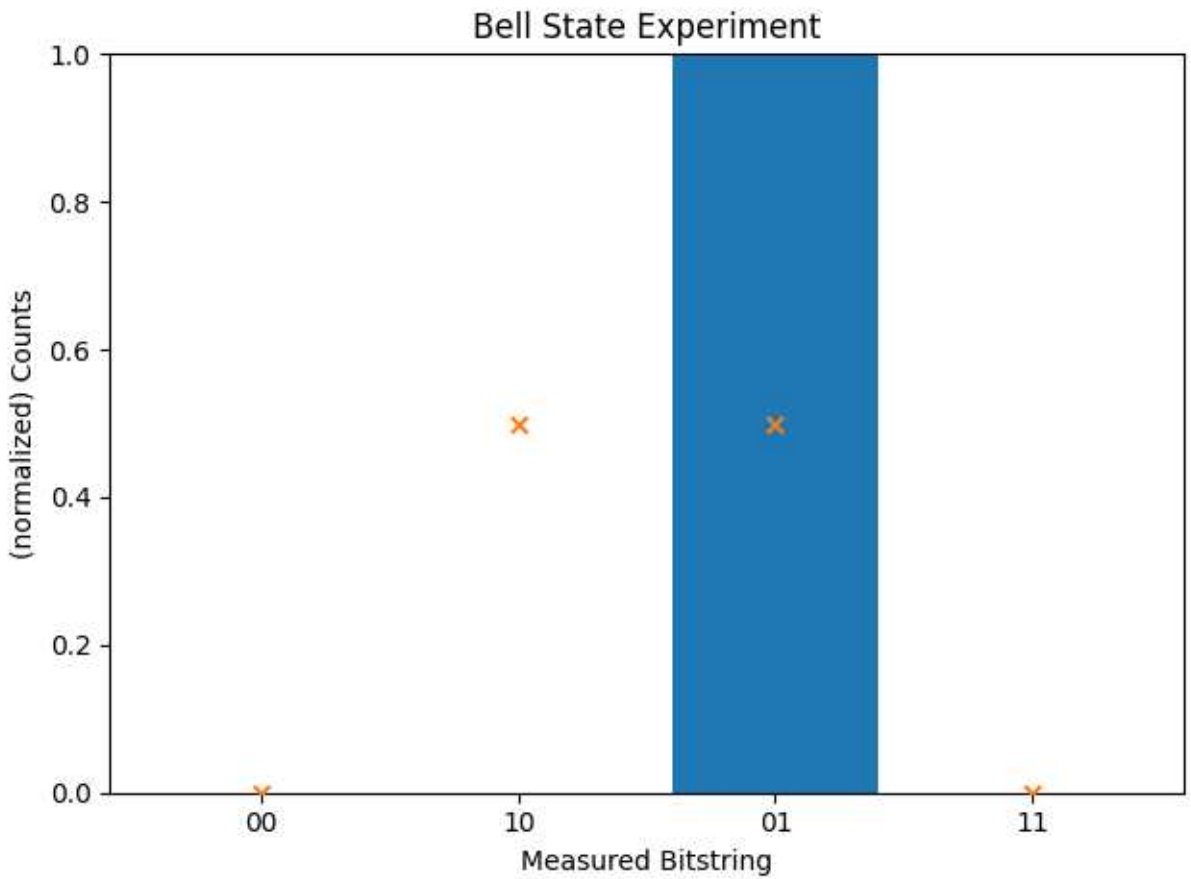
[9]:
```
array([0., 0., 0., 1.])
```

3 Processing the Results

[10]:
```
import matplotlib
from matplotlib import pyplot
```

[11]:
```
fig, ax = pyplot.subplots()
ax.set(ylim=(0,1))
hist = jaqal_result.subcircuits[1].relative_frequency_by_int
probs = jaqal_result.subcircuits[1].simulated_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, marker='x', zorder=3)
ax.set_xlabel('Measured Bitstring')
ax.set_ylabel('(normalized) Counts')
ax.set_title('Bell State Experiment')
```

```
fig.tight_layout()
pyplot.show()
```



4 Noisy Density Matrix Simulation

[12]:
```
from qscout.v1.std.noisy import SNLToy1
```

[13]:
```
jaqal_result = run_jaqal_circuit(jaqal_circuit, backend=SNLToy1(2, depolarization=2e-3))
```

[14]:
```
jaqal_result
```

[14]:
```
<jaqalpaq.core.result.ExecutionResult at 0x85aa69c9ca30>
```

[15]:
```
jaqal_result.subcircuits[1].probability_by_int
```

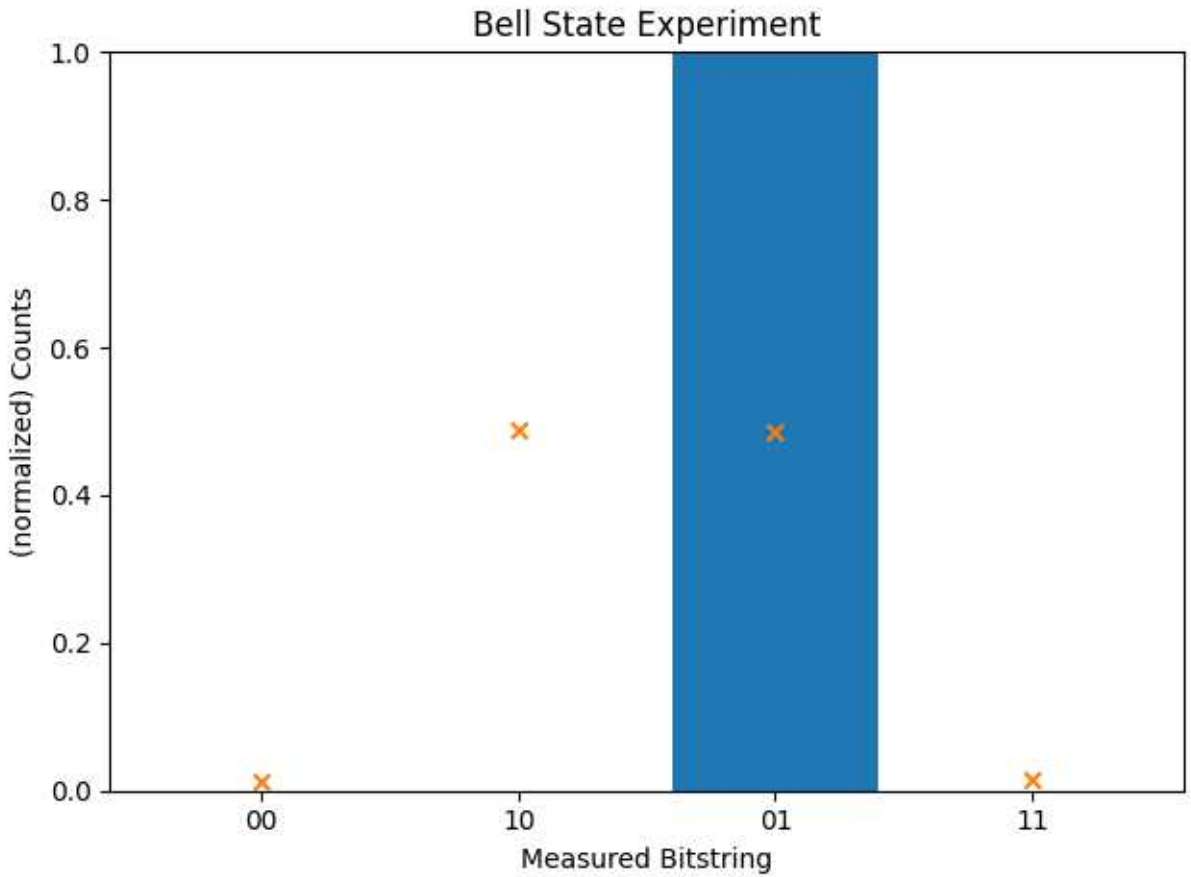
[15]:
```
array([0.01231296, 0.48720558, 0.48672233, 0.01375913])
```

[16]:
```
jaqal_result.subcircuits[1].relative_frequency_by_int
```

[16]:
```
array([0., 0., 1., 0.])
```

[17]:
```
fig, ax = pyplot.subplots()
ax.set(ylim=(0,1))
hist = jaqal_result.subcircuits[1].relative_frequency_by_int
probs = jaqal_result.subcircuits[1].simulated_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, marker='x', zorder=3)
```

```
ax.set_xlabel('Measured Bitstring')
ax.set_ylabel('(normalized) Counts')
ax.set_title('Bell State Experiment')
fig.tight_layout()
pyplot.show()
```


Bell State Experiment
1.0
0.8
0.6
0.4
0.2
0.0
(normalized) Counts
00
10
01
11
Measured Bitstring

5 What is this object though?

[18]:
```
con_model = SNLToy1(2, depolarization=2e-3)
```

[19]:
```
con_model.__dict__.keys()
```

[19]:
```
dict_keys(['depolarization', 'rotation_error', 'phase_error', 'n_qubits',
'stretched_gates', 'model', 'gate_durations'])
```

Parameters:

[20]:
```
con_model.depolarization, con_model.rotation_error, con_model.phase_error
```

[20]:
```
(0.002, 0.01, 0.01)
```

A process matrix for every gate:

[21]:
```
con_model.model
```

[21]:
```
<pygsti.models.localnoisemodel.LocalNoiseModel at 0x85aa69c9ca30>
```

A duration for every gate:

[22]:
```
con_model.gate_durations
```

[22]:
```
{'R': <bound method SNLToy1.gateduration_R of <qscout.v1.std.noisy.SNLToy1
object at 0x85aa69c9ca30>>,
 'MS': <bound method SNLToy1.gateduration_MS of <qscout.v1.std.noisy.SNLToy1
object at 0x85aa69c9ca30>>,
 'Rz': <bound method SNLToy1.gateduration_Rz of <qscout.v1.std.noisy.SNLToy1
```

```
object at 0x85aa69c9ca30>>,
 'Rx': <bound method ExtensibleBackend._curry.<locals>._inner.<locals>.newop of
<qscout.v1.std.noisy.SNLToy1 object at 0x85aa69c9ca30>>,
 'Ry': <bound method ExtensibleBackend._curry.<locals>._inner.<locals>.newop of
<qscout.v1.std.noisy.SNLToy1 object at 0x85aa69c9ca30>>,
 'Px': <bound method ExtensibleBackend._curry.<locals>._inner.<locals>.newop of
<qscout.v1.std.noisy.SNLToy1 object at 0x85aa69c9ca30>>,
 'Py': <bound method ExtensibleBackend._curry.<locals>._inner.<locals>.newop of
<qscout.v1.std.noisy.SNLToy1 object at 0x85aa69c9ca30>>,
 'Pz': <bound method ExtensibleBackend._curry.<locals>._inner.<locals>.newop of
<qscout.v1.std.noisy.SNLToy1 object at 0x85aa69c9ca30>>,
 'Sx': <bound method ExtensibleBackend._curry.<locals>._inner.<locals>.newop of
<qscout.v1.std.noisy.SNLToy1 object at 0x85aa69c9ca30>>,
 'Sy': <bound method ExtensibleBackend._curry.<locals>._inner.<locals>.newop of
<qscout.v1.std.noisy.SNLToy1 object at 0x85aa69c9ca30>>,
 'Sz': <bound method ExtensibleBackend._curry.<locals>._inner.<locals>.newop of
<qscout.v1.std.noisy.SNLToy1 object at 0x85aa69c9ca30>>,
 'Sxd': <bound method ExtensibleBackend._curry.<locals>._inner.<locals>.newop of
<qscout.v1.std.noisy.SNLToy1 object at 0x85aa69c9ca30>>,
 'Syd': <bound method ExtensibleBackend._curry.<locals>._inner.<locals>.newop of
<qscout.v1.std.noisy.SNLToy1 object at 0x85aa69c9ca30>>,
 'Szd': <bound method ExtensibleBackend._curry.<locals>._inner.<locals>.newop of
<qscout.v1.std.noisy.SNLToy1 object at 0x85aa69c9ca30>>,
 'Sxx': <bound method ExtensibleBackend._curry.<locals>._inner.<locals>.newop of
<qscout.v1.std.noisy.SNLToy1 object at 0x85aa69c9ca30>>}
```

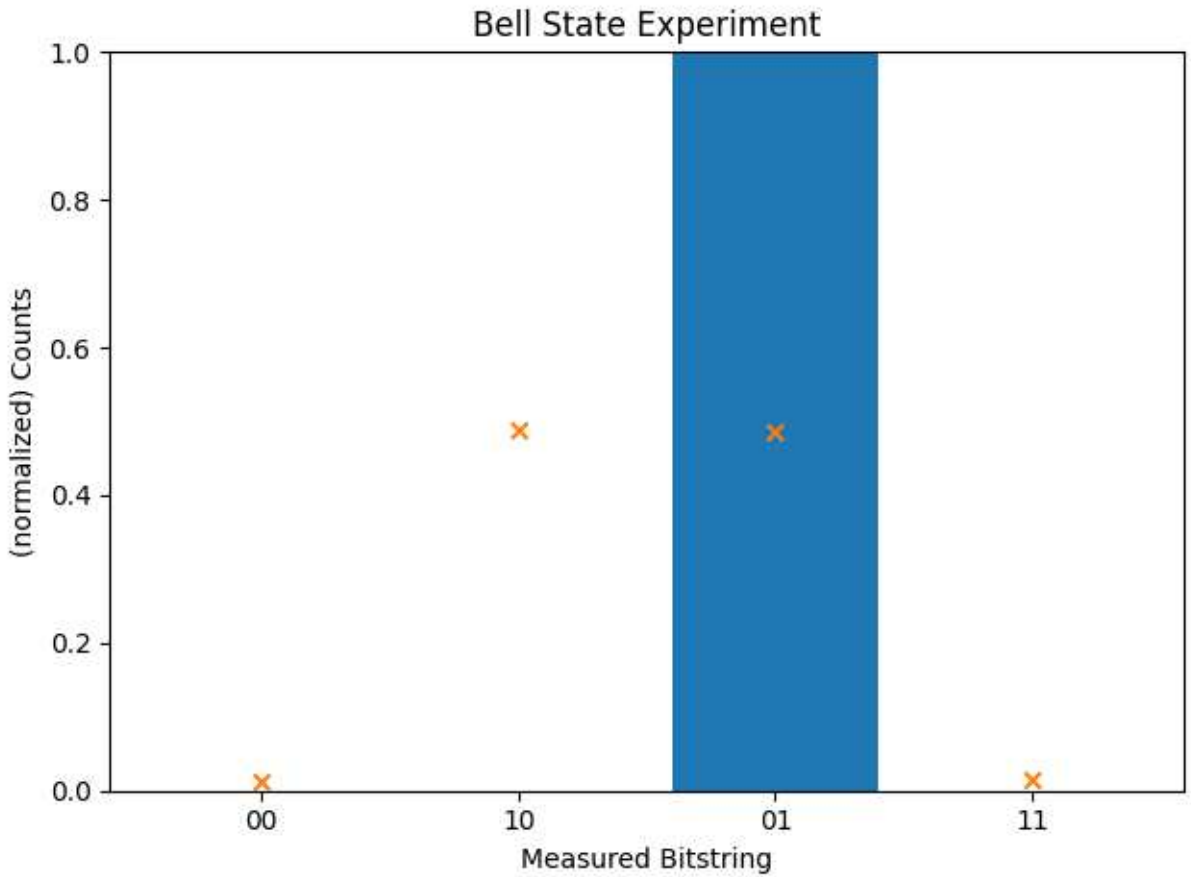
The model and gate durations are all we actually need.

You can manually assemble them with `CircuitEmulator`:

[23]:
```
from jaqalpaq.emulator.pygsti import CircuitEmulator

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

[24]:
```
jaqal_result = run_jaqal_circuit(jaqal_circuit, backend=backend)
```

[25]:
```
fig, ax = pyplot.subplots()
ax.set(ylim=(0,1))
hist = jaqal_result.subcircuits[1].relative_frequency_by_int
probs = jaqal_result.subcircuits[1].simulated_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, marker='x', zorder=3)
ax.set_xlabel('Measured Bitstring')
ax.set_ylabel('(normalized) Counts')
ax.set_title('Bell State Experiment')
fig.tight_layout()
pyplot.show()
```


Bell State Experiment
1.0
0.8
0.6
0.4
0.2
0.0
(normalized) Counts
00
10
01
11
Measured Bitstring

6 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:

[26]:
```
# Useful tools:
from numpy import abs, diag, pi, kron
import pygsti

# Description of the ideal gates
from qscout.v1.std.jaqal_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.std.jaqal_gates import ALL_GATES


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

    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)
```

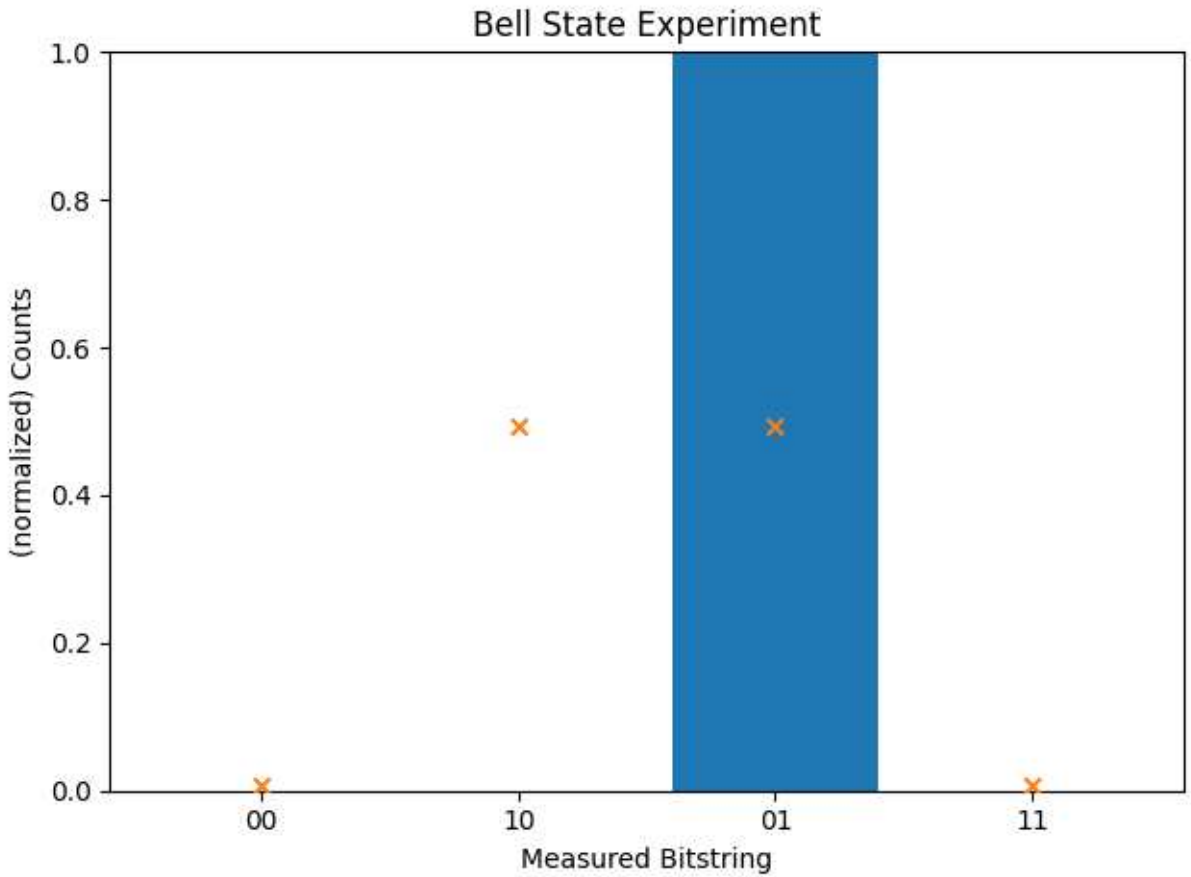
```
        duration = self.gateduration_Rx(q, rotation_angle)

        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
```

[27]:
```
backend = MyToyEmulator(2)
```

[28]:
```
jaqal_result = run_jaqal_circuit(jaqal_circuit, backend=backend)
```

[29]:
```
fig, ax = pyplot.subplots()
ax.set(ylim=(0,1))
hist = jaqal_result.subcircuits[1].relative_frequency_by_int
probs = jaqal_result.subcircuits[1].simulated_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, marker='x', zorder=3)
ax.set_xlabel('Measured Bitstring')
ax.set_ylabel('(normalized) Counts')
ax.set_title('Bell State Experiment')
fig.tight_layout()
pyplot.show()
```



The minimal skeleton to use `AbstractNoisyNativeEmulator`

[30]:
```python
from qscout.v1.std.jaqal_gates import ALL_GATES
from jaqalpaq.emulator.pygsti import AbstractNoisyNativeEmulator

class YourCustomEmulator(AbstractNoisyNativeEmulator):
    jaqal_gates = ALL_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
```