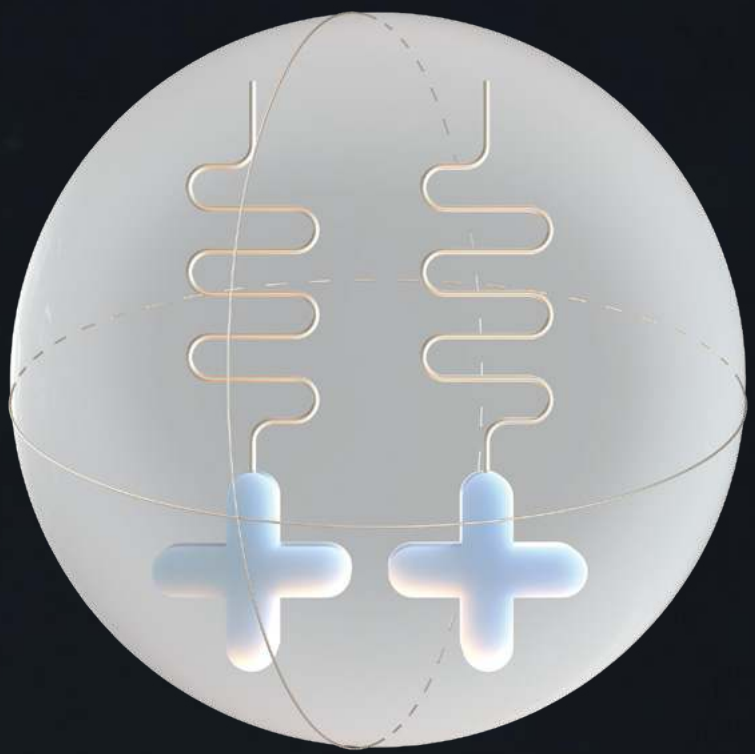
QUANTUM MACHINES

USE CASE

Quantum Orchestration for Superconducting Qubits

With the Quantum Orchestration Platform, there's no limit to the kind of experiments you can run. Find out how to supercharge your superconducting qubits research with these real-world use cases.



RAMSEY MEASUREMENT

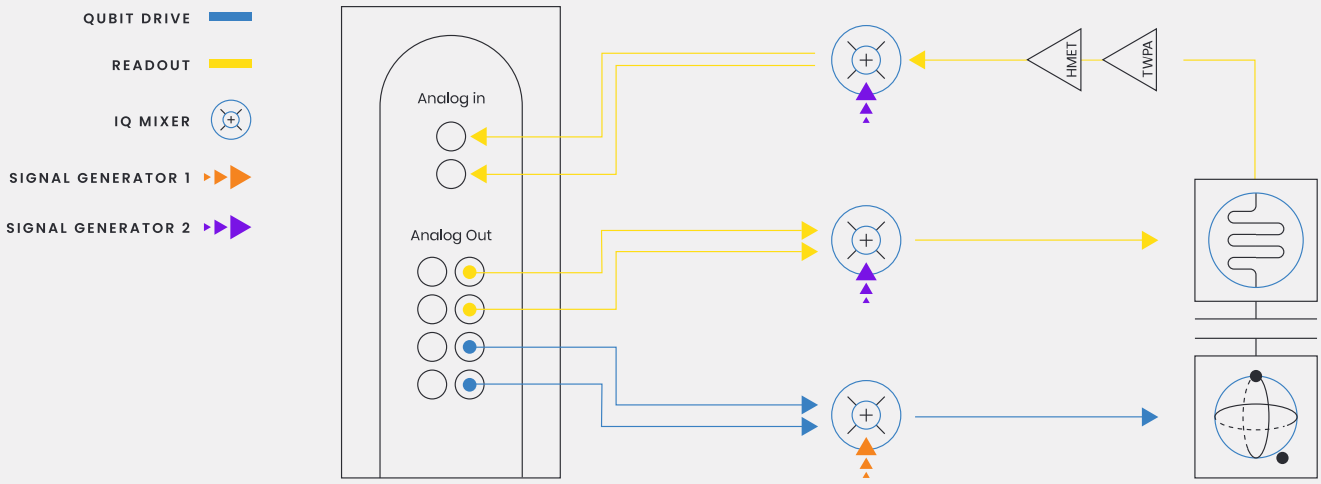


Fig. 1 *A schematic description of the connectivity between the OPX+ and the experimental system for the Ramsey experiment. A single qubit is characterized, and its state is measured with a readout resonator. The IF signals from the OPX+ are upconverted and downcoverted using IQ mixers.*

The Ramsey sequence is the most basic direct probe of quantum coherence in a two-level system. It consists of four steps: An initial $Y/2$ pulse, followed by a free evolution, an additional $Y/2$ pulse in the rotating frame of the qubit, and lastly, a projective measurement to determine the state of the qubit in the $|0\rangle$, $|1\rangle$ basis. This sequence is repeated multiple times, which allows us to build up measurement statistics. We repeat it for different free-evolution times.

The Ramsey sequence directly measures quantum coherence. During the free evolution time, the qubit will precess in the frame rotating with the drive frequency at a rate of $\Delta f = f_{\mathrm{drive}} - f_{01}$. Due to this, it will pick up a phase which depends on both the free evolution time and on Δf. If we repeat this many times, only a coherent and repeatable phase evolution will lead to oscillations in the measured state of the qubit. Consequently, the amplitude of these oscillations directly reflects the coherence of the phase acquired during free evolution.

We typically use this sequence to determine two vital parameters of the qubit: The T_2^* time, which is the low-frequency dephasing noise from the environment [1], as well as the qubit transition frequency, which can be accurately measured since it is the only frequency for which no oscillations are visible.

The following QUA program implements this Ramsey measurement sequence. It is crucial to emphasize that although this code is Python-embedded, it is an independent programming language. A proprietary compiler will compile the code written here into a set of machine instructions that the pulse processing unit (PPU) inside the OPX+ will then execute in real-time and with nanosecond-scale latency.

RAMSEY WITH FREQUENCY TUNING

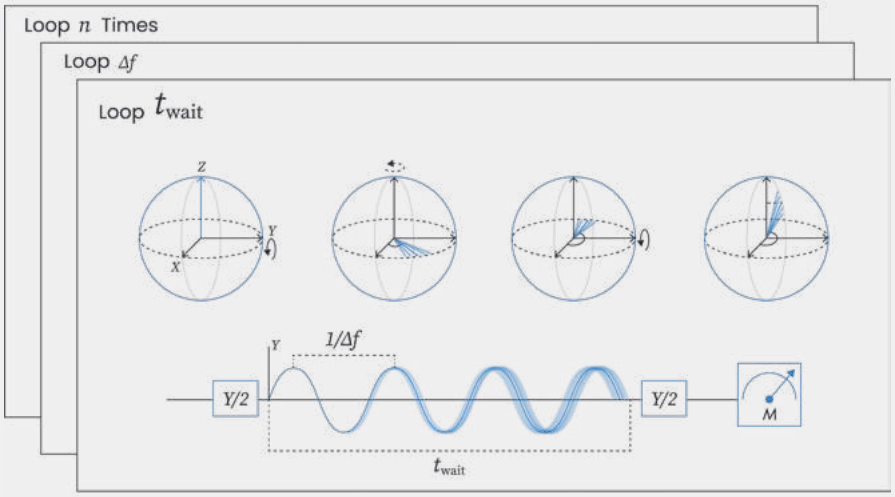


Fig. 2 *The Ramsey interference sequence for measuring T_2^* and f_{01} of the qubit. Three nested loops are used, and t_{wait} and Δf are scanned over. The top part of the figure corresponds to the state of the qubit on the Bloch sphere during the sequence, before and after each gate operation. During the free evolution stage, the phase acquired drifts between different iterations, reflected as smearing in Y's sinusoidal variation and in the bloch vector component.*

The program consists of 3 nested for loops; a schematic can be seen in Fig 2, and the code is implemented in QUA on the following page. The first external loop is an averaging loop. The second loop loops over the frequency, which is updated to a new value when we call `update_frequency('qubit1', f_sweep)` in line 26, and the third one loops over the wait time between the two pulses.

We play two $Y/2$ pulses in each iteration, here generated in QUA with the statement `play ('pi_pulse' * amp(0.5), 'qubit1' )` in lines 27 and 29. This statement plays a π-pulse as defined in the program configuration, and scales its amplitude by half. The program configuration, which is not shown here for the sake of brevity, is where all the static information, including waveform samples, output frequencies and state discriminators. is stored. Between the pulses, we wait for a time corresponding to the QUA variable `free_time`. Then, in the `measure` statement on line 31, we perform the following sequence.

Send a drive pulse to the readout resonator, wait for a fixed amount of time, acquire the output from the readout resonator, demodulate it, and store the demodulated result in the QUA variables `I` and `Q`. We can then either send the results back to the PC using the `save` statements in lines 34 or 35, or use the IQ values inside the QUA program for whatever we choose.

The process is repeated for different qubit offset frequencies and wait times. As explained above, for a given frequency offset, we observe oscillations with a period which corresponds to $1/\Delta f$, which gives the characteristic lineshapes shown in Fig. 3. The visibility of these lineshapes decays uniformly as time progresses. The run time of this program, including data acquisition, was approximately 10 minutes.

A crucial ingredient in the program is the `reset_qubit1` method. This method is a simple yet illustrative example of an active reset sequence that demonstrates how easy it is to perform feedback operations on the OPX+. All that is needed is to store the result of the measurement into a QUA variable, and then use this variable in a `while` loop to perform a conditioned π-pulse operation!

```python
from qm.qua import *


def reset_qubit1():
    qv = declare(fixed)
    align('res', 'qubit1')
    measure('meas_op_res', 'res', None, demod.full('integ_w_s', qv))

    # qv is the discriminating IQ variable.
    with while_(qv > -0.005):
        play('pi_op_qubit1', 'qubit1')
        measure('meas_op_res', 'res', None, demod.full('integ_w_s', qv))


with program() as prog:
    f_sweep = declare(int)
    I = declare(fixed)
    Q = declare(fixed)
    n = declare(int)
    free_time = declare(int)

    with for_(n, 0, n < 1000, n + 1):
        with for_(f_sweep, 99e6, f_sweep < 101e6, f_sweep + 0.01e6):
            with for_(free_time, 20, free_time < 8000, free_time + 10):
                reset_qubit1()
                update_frequency('qubit1', f_sweep)
                play('pi_op_qubit1' * amp(0.5), 'qubit1')
                wait(free_time, 'qubit1')
                play('pi_op_qubit1' * amp(0.5), 'qubit1')
                align('qubit1', 'res')
                measure('meas_op_res', 'res', None,
                        demod.full('integ_w_c', I),
                        demod.full('integ_w_s', Q))
                save(I, 'I')
                save(Q, 'Q')
```

QUA code for Ramsey 2D scan *This QUA code was used to generate the data seen in Fig. 3.*

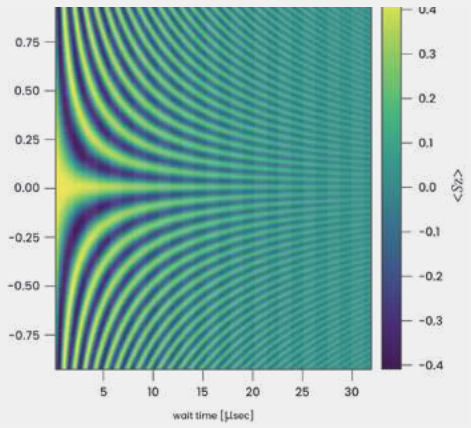


Fig. 3 *The expectation value of $\langle S_z \rangle$ for different wait times and frequency detuning values. The oscillation period is $1/\Delta f$. The visibility of the interference oscillations decays with wait time, from which the T_2^* coherence time of the qubit can be inferred. This measurement took about 10 minutes with the OPX+.*

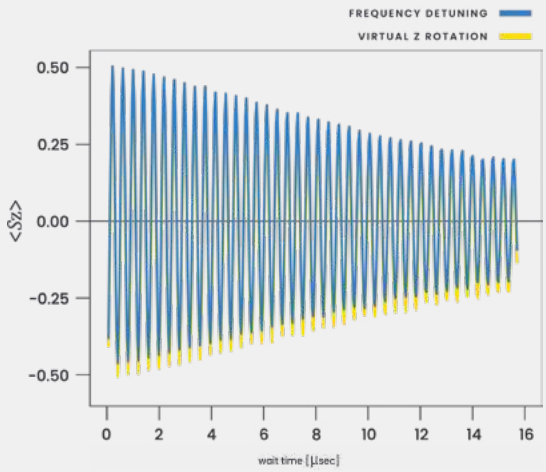


Fig. 4 *Decaying oscillations for real frequency detuning of 0.251MHz (yellow line), showing asymmetry, and for a virtual Z rotation which allows us to accumulate a phase equal to $2\pi\Delta f t_{\text{wait}}$. during the free evolution stage.*

RAMSEY WITH OPTIMAL $\pi/2$ PULSES

In the previous example, we glossed over an important detail: when we played the pulses, the frequency detuning was also on, and thus our pulses were slightly detuned and contained a (small) Z component. This can lead to asymmetric oscillations which are not centered around $\langle S_z \rangle = 0$, as the qubit will not process on the equatorial plane exactly (see Fig. 4). If we wanted to get rid of this Z component, we would have to detune the frequency only during the free evolution time. We can do this by artificially adding a phase during the free evolution stage, proportional to the wait time.

In QUA, we can easily do this using the `frame_rotation_2pi`. statement, as seen in the example program. This statement is accumulative, which means that calling it with a value α will add $2\pi\alpha$ to the phase of subsequent pulses. This allows it to function as an actual frame rotation or as a virtual Z-rotation operation [2].

In fig.4 we compare two scenarios: in the first, we detune the drive frequency by 0.251MHz throughout the experiment. In the second, we update the frame by $0.251\text{MHz} \times t_{wait}$ using `frame_rotation_2pi`. We can see that both lead to the same oscillation frequency. However, in the second case the oscillations are more symmetric and centered around $\langle S_z \rangle = 0$, thus enabling us to perform a better fitting procedure for T_2^* and the frequency detuning.

```python
from qm.qua import *

free_time_step = 20
offset_freq = 0.251e6  # the artificial offset frequency added during free evolution time only

# factor of 4 is due to free time being in units of 4nsec
rot_angle = free_time_step * 4 * offset_freq * 1e-9

with program() as ramsey_2d:
    I = declare(fixed)
    Q = declare(fixed)
    n = declare(int)
    free_time = declare(int)
    total_angle = declare(fixed, value=0)

    with for_(n, 0, n < 1000, n + 1):
        assign(total_angle, 0)
        with for_(free_time, free_time_step, free_time < 4000, free_time + free_time_step):
            reset_qubit1()
            assign(total_angle, total_angle + rot_angle)
            play('pi_op_qubit1' * amp(0.5), 'qubit1')
            wait(t_wait, 'qubit1')
            frame_rotation_2pi(total_angle, 'qubit')
            play('pi_op_qubit1' * amp(0.5), 'qubit1')
            align('qubit1', 'res')
            measure('meas_op_res', 'res', None, demod.full('integ_w_c', I), demod.full('integ_w_s', Q))

            save(I, 'I')
            save(Q, 'Q')
```

QUA Code for 1D Ramsey scan *This QUA code was used to generate the dataset in Fig. 4. The method reset_qubit1() is defined in the QUA code for generating the 2D Ramsey scan above.*

QUANTUM ERROR-CORRECTION AND REAL-TIME FEEDBACK

Quantum Error-Correction is a great example demonstrating the integration of many of the benefits of the Quantum Orchestration Platform (QOP) and the OPX+. Here we show an example of how you can easily implement a 3-qubit bit-flip code on a superconducting qubit device with the QOP.

Figure 1 shows the experimental setup. Five transmon qubits are controlled using microwave signals which are IQ modulated by ten analog output channels of the two OPXs+, for XY control.

Flux bias signals are generated directly by five additional analog output channels of the OPXs+, for Z control.

Each qubit is coupled to a readout resonator and all five resonators are coupled to the same transmission line. The transmission line is probed using another microwave signal that is IQ modulated by two analog output channels of an OPX+ and measured after downconversion by the OPX+ analog input channel.

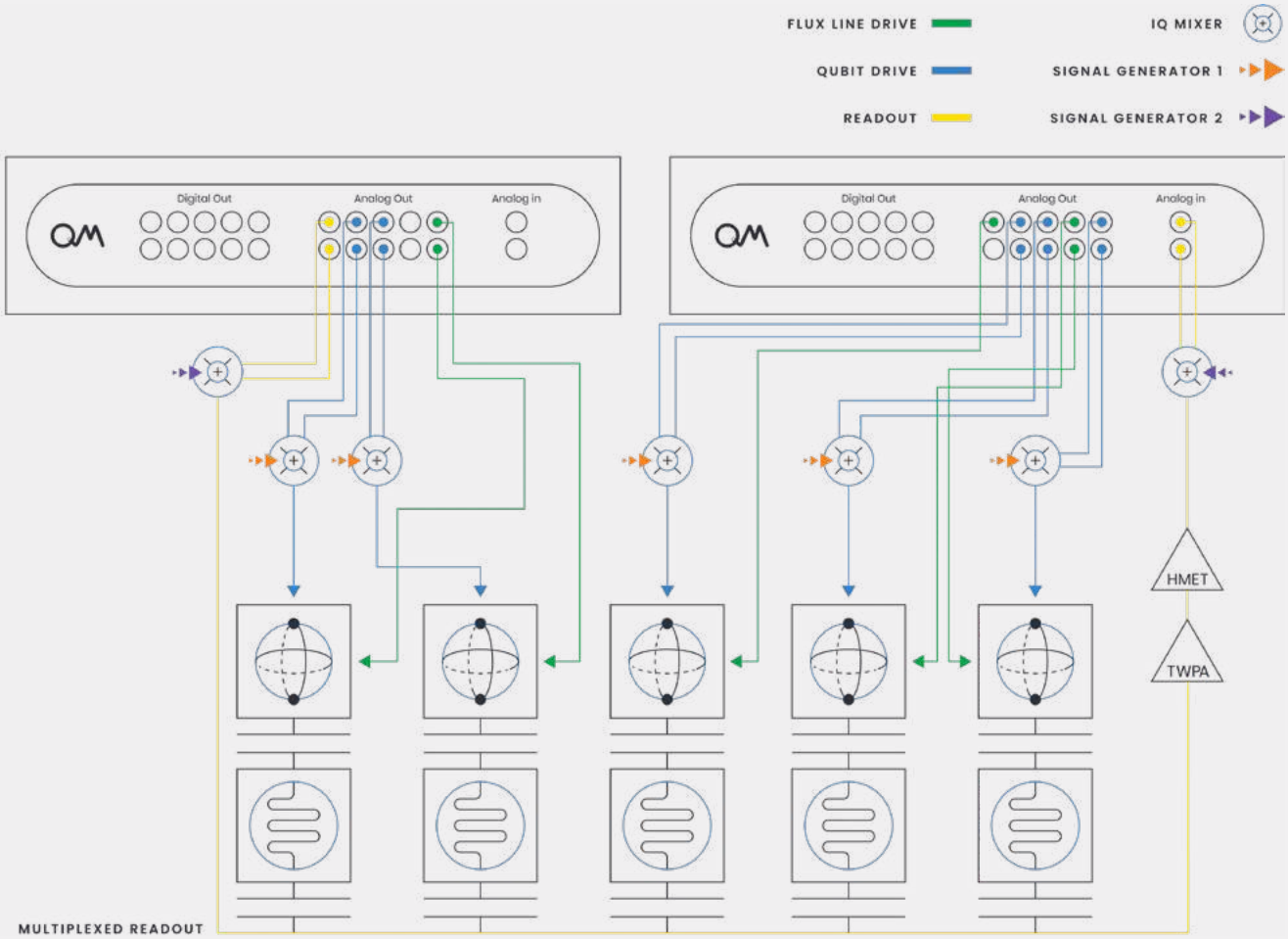


Fig. 1 *A setup for interfacing five transmons. Two OPXs+ are set up to control the MW and flux tuning lines of five transmons.*

Figure 2 shows the 3-qubit bit-flip code circuit. A logical qubit state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ is encoded using three physical qubits in the state $\alpha|000\rangle + \beta|111\rangle$. If the first qubit is prepared in the original state $|\psi\rangle$, then this can be done by performing two CNOT gates as shown. In the Encoding stage of the circuit in the figure

The idea of the 3-qubit bit-flip code is that a single bit flip in the encoded state can be detected by measuring and tracking the parity of two pairs of qubits (Repeat Error Tracking stage of the circuit in Fig.2). This can be N times track the bit-flips during some time, after which a correction is applied to the three qubits according to the error tracking results (Correction stage in Figure 2). Finally, the state is decoded back to the state of a single physical qubit $|\psi\rangle_L$.

The parity measurements can be done by employing two more ancilla qubits initialized in the $|0\rangle$ state before every measurement sequence, as shown in Fig.2 To measure the parity of a pair of qubits, say qubit 1 and 2, one CNOT gate is applied to qubit 1 and the ancilla qubit, and one CNOT gate is applied to qubit 2 and the same ancilla. This entangles the parity of the two qubits with the state of the ancilla, which can then be measured to determine the parity.

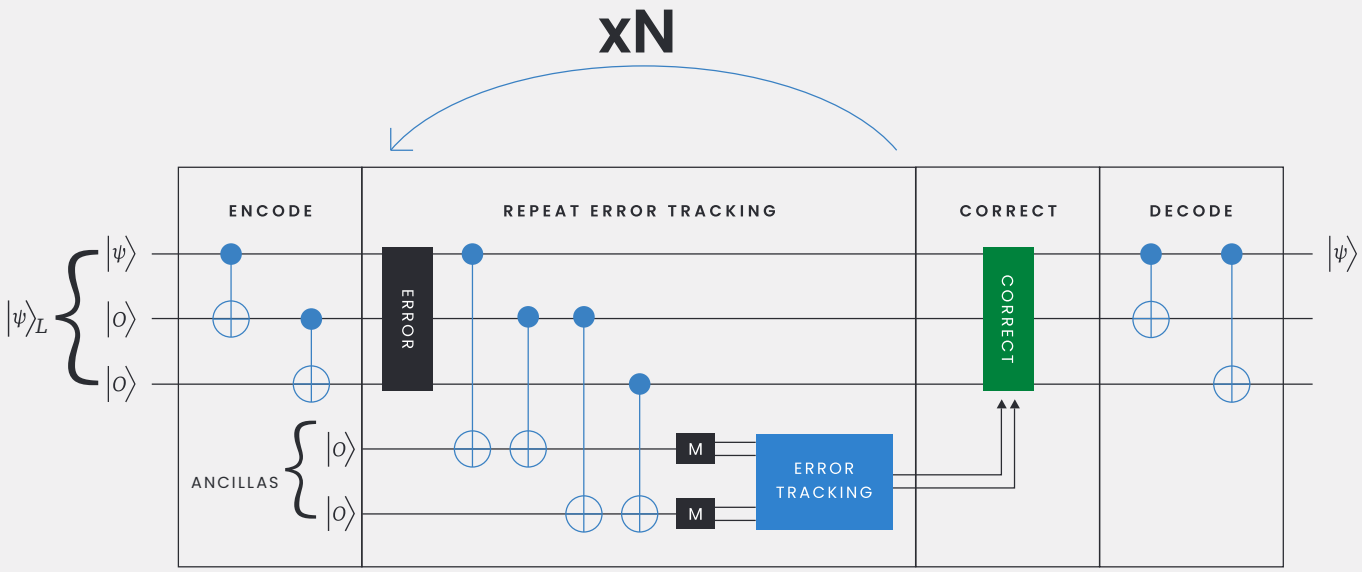


Fig. 2 N = amount of repeated measurements. Circuit for a 3-qubit error correction code. Two ancilla qubits are used to enable both error detection and correction. $|\psi\rangle_L$ is the logical qubit, expressed in the extended Hilbert space. The error tracking process is repeated N times.

From the control perspective, running such a protocol is very demanding since it requires:

- **Fast real-time processing:** to manage the error tracking in every iteration of the Repeated Error Tracking stage. Faster than the desired duration of the iteration, which is typically 100s of ns in superconducting circuits [3].
- **Control flow:** to enable Error Tracking repetition for as many times as one requires
- **Low-latency feedback:** to correct the error based on the error tracking results

```python
from qm.qua import *

drive_elements = ['q0_xy', 'q1_xy', 'q2_xy', 'a0_xy', 'a1_xy']
readout_elements = ['q0_resonator', 'q1_resonator', 'q2_resonator', 'a0_resonator', 'a1_resonator']
all_elements = drive_elements + readout_elements

repetitions = 10
with program() as bit_flip_code:

    qb_states = declare(bool, size=3)
    an_states = declare(bool, size=2)
    flips = declare(bool, size=3, value=[False, False, False])
    i = declare(int)

    # Preparation
    reset_qubits(drive_elements, readout_elements)
    play('pi/2', 'q0_xy')

    # Encoding
    CNOT('q0_xy', 'q1_xy')
    CNOT('q0_xy', 'q2_xy')

    # Error tracking
    with for_(i, 0, i<repetitions, i+1):
        CNOT('q0_xy', 'a0_xy')
        CNOT('q1_xy', 'a0_xy')
        CNOT('q1_xy', 'a1_xy')
        CNOT('q2_xy', 'a1_xy')
        measure_states(['a0_resonator', 'a1_resonator'], an_states, [0,0])
        save_vector(an_states, 'states')
        with if_(an_states[0]==0 & an_states[1]==1):
            play('pi', 'a1_xy')
            assign(flips[2], ~flips[2])
        with if_(an_states[0]==1 & an_states[1]==0):
            play('pi', 'a0_xy')
            assign(flips[0], ~flips[0])
        with if_(an_states[0]==1 & an_states[1]==1):
            play('pi', 'a0_xy')
            play('pi', 'a1_xy')
            assign(flips[1], ~flips[1])
        save_vector(flips, 'all_ground')

    # Error correction
    with if_(flips[0]):
        play('pi', 'q0_xy')
    with if_(flips[1]):
        play('pi', 'q1_xy')
    with if_(flips[2]):
        play('pi', 'q2_xy')

    # Decoding
    CNOT('q0_xy', 'q1_xy')
    CNOT('q0_xy', 'q2_xy')
```

QUA Code *for implementing the 3-qubit bit-flip code.*

"Having tried several instruments in the past, I'm very impressed by Quantum Machines' OPX. It finally removes the need for us to develop any skills in FPGA programming while still benefiting from advanced FPGA capabilities in our experiments."

Prof. Benjamin Huard, ENS de Lyon

ENS DE LYON

To implement the 3-qubit bit-flip code in QUA, first, classical variables are initialized. We store measurements of the three data qubits in a boolean vector `qb_states`, measurements of the two ancilla states in the boolean vector `an_states`, and the parity of the number of `flips` on each data qubit in the boolean vector flips.

Then, the preparation stage of the circuit is defined. Here we first reset the five qubits using the function `reset_qubits(elements)`. For specific implementation of an active reset function of a single qubit, which uses real-time feedback and resets the qubits quickly and efficiently with QUA, please refer to the Ramsey example above. Here we focus on the error correction code.

After we initialize the qubits we apply a $\pi/2$ pulse to the first data qubit in order to prepare the qubit in an initial state. Then, we entangle the data qubit with the two ancillary qubits.

The error tracking phase inside a for-loop repeats blocks of CNOT gates followed by a measurement of the output resonators, from which an error syndrome can be extracted. If an error is detected, one of the three conditional statements can apply to the appropriate recovery gate. The actual waveforms required to perform the CNOT gate vary between quantum computer implementations, and the specific CNOT method can be implemented straightforwardly using the same constructs we've seen so far.

References

[1] P. Krantz, M. Kjaergaard, F. Yan, T. P. Orlando, S. Gustavsson, and W. D. Oliver, "A quantum engineer's guide to superconducting qubits," Appl. Phys. Rev., vol. 6, no. 2, p. 021318, Jun. 2019.

[2] D. C. McKay, C. J. Wood, S. Sheldon, J. M. Chow, and J. M. Gambetta, "Efficient Z gates for quantum computing," Phys. Rev. A, vol. 96, no. 2, p. 022330, Aug. 2017.

[3] M. D. Reed et al., "Realization of three-qubit quantum error correction with superconducting circuits," Nature, vol. 482, no. 7385, pp. 382–385, Feb. 2012.Nat. Commun., vol. 5, no. 1, pp. 1–6, Oct. 2014.

[4] S. J. Devitt, W. J. Munro, and K. Nemoto, "Quantum error correction for beginners," Reports Prog. Phys., vol. 76, no. 7, pp. 76001–76036, Jul. 2013.

"I must say I'm very happy with QM's Quantum Orchestration Platform. It's the single most reliable piece of equipment I've got in the lab. I operate it remotely and never had any problems. I strongly recommend the OPX and the QOP to my colleagues. It is by far the simplest way to go qubit physics."

Dr. Emmanuel Flurin, CEA Saclay, Quantronics Group

The Quantum Orchestration Platform

AN END TO END QUANTUM CONTROL SOLUTION TO DRIVE THE FASTEST TIME TO RESULTS, AT ANY SCALE

OPX+

RUN STATE OF THE ART EXPERIMENTS WITH EASE

An architecture designed from the ground up for quantum control, the OPX+ lets you run the quantum experiments of your dreams right from the installation. With a quantum feature-rich environment, the OPX+ is built for scale and performance. Now, you can **run the most complex quantum algorithms and experiments in a fraction of the development time**.

PULSE PROCESSING UNIT

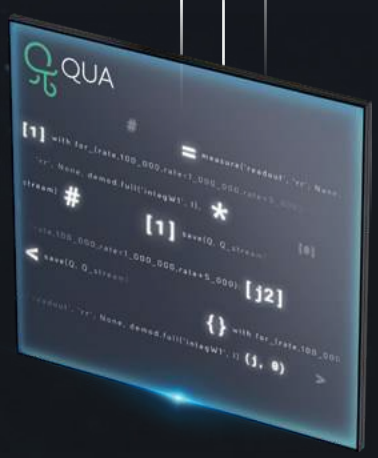
ACHIEVE THE FASTEST TIME TO RESULTS

Within the OPX+ is the Pulse Processing Unit, QM's leading-edge quantum control technology. Progress with incomparable speed and extreme flexibility. Run even the most demanding experiments efficiently, with the fastest runtimes and the lowest latencies in the industry, including quantum protocols that require **real-time waveform generation, real-time waveform acquisition, real-time comprehensive processing, and control flow.**

QUA

CODE QUANTUM PROGRAMS SEAMLESSLY

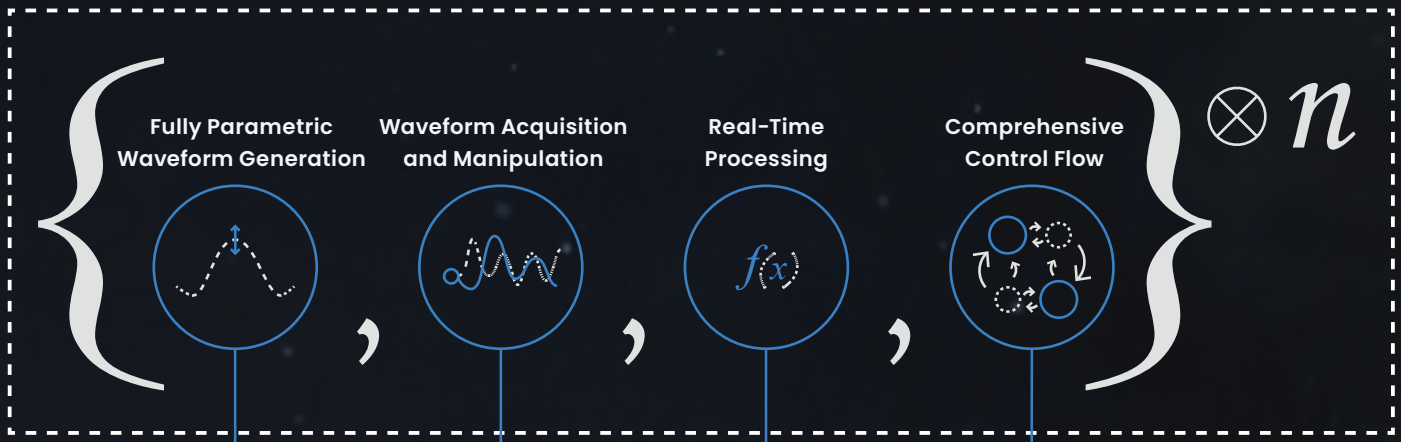
Implement the protocols of your wildest dreams as easily as writing pseudocode. Designed for quantum control, QUA is the first universal quantum pulse-level programming language. Code even the most advanced programs and run them with the best possible performance. Natively describe your most challenging experiments, from complex AI-based multi-qubit calibrations to multi-qubit quantum error correction.



**All of the information above is also valid for the OPX*

QUANTUM MACHINES

YOUR PROTOCOLS LIVE IN THIS PHASE SPACE



Real-Time Waveform Generation

- Fully parametrized: length, frequency, phase (relative & absolute), amplitude, bandwidth, chirp
- Compensations: Crosstalk matrix, FIR and IIR filters
- And many more

Real-Time Measurements

- High fidelity analog to digital conversion
- General integration & demodulation
- Weighted Integrations accumulated integrations, sliced integrations, etc.
- Time Tagging, TTL counting

Real-Time Processing

Turing complete:
- Basic arithmetics
- Evaluation of trigonometric functions
- Vector operations
- Casting of variable types
- And more (Turing complete)

Real-Time Control Flow

- If/else
- For loops
- While loops
- Switch case

Real-Time 'Quantum' Estimations

- State estimations
- Error estimations
- Bayesian estimations
- Correlation functions
- Neural nets based estimations

Real-Time Multi-Qubit Feedback

- Qubit stabilization & tracking
- Quantum error correction
- And many more

THE QUANTUM ORCHESTRATION PLATFORM COVERS THIS SPACE!

- Easily express quantum algorithms and experimental protocols that comprise all of the above.
- Seamlessly sync measurements, real-time calculations, and pulses of different quantum elements.
- Loop over a wide range of parameters in real-time, including intermediate frequencies, amplitudes, phases, delays, integration parameters, measurement axes, etc.
- Use if/else and switch-case statements to condition operations in real time (real time feedback).
- Define procedures (macros) to be reused in the code and access an extensive family of libraries.

QUANTUM MACHINES

If you wish to learn more:

info@quantum-machines.co

About Quantum Machines

Quantum Machines (QM) drives quantum breakthroughs that accelerate the path towards the new age of quantum computing. The company's Quantum Orchestration Platform (QOP) fundamentally redefines the control and operations architecture of quantum processors.

The full-stack hardware and software platform is capable of running even the most complex algorithms right out of the box, including quantum error correction, multi-qubit calibration, and more. Helping achieve the full potential of any quantum processor, the QOP allows for unprecedented advancement and speed-up of quantum technologies as well as the ability to scale into the thousands of qubits. Visit us at: www.quantum-machines.co

---