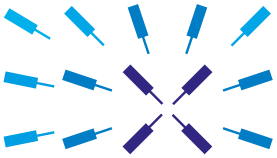
Application Note



Zurich Instruments

Repeat-Until-Success Algorithm on Superconducting Qubits with Real-Time Feedback

Applications: Quantum Computing, Quantum Simulation
Products: QCCS

Release date: June 2023

Abstract

Repeat-Until-Success (RUS) is a technique for performing quantum gate decomposition, which is crucial for enabling the execution of relevant algorithms on quantum computers. In this Application Note we look at Repeat-Until-Success from two points of view. First, we present it from a theoretical perspective and we address one of its most promising future applications, quantum neural networks. Second, we present the practical implementation of a single-qubit example of Repeat-Until-Success in a tabletop demonstration with the Zurich Instruments Quantum Computing Control System (QCCS). This example, designed for superconducting qubits, illustrates real-time feedback concepts that can be readily scaled to multi-qubit algorithms.

Introduction

With the steady progress in quantum computing technology, an important need is having algorithms that efficiently convert a high-level quantum algorithm into low-level fault-tolerant quantum gates. This task is usually broken down in two steps: first, a universal set of gates has to be chosen, and second, there must be a decomposition algorithm that can translate an arbitrary quantum circuit into a sequence of gates from the chosen universal set. While the choice of the gate set is typically guided by properties of the underlying technological platform, there is a much broader range of possibilities when it comes to the decomposition procedure. A good decomposition algorithm should minimize the number of gates needed to implement the quantum algorithm, and at the same time ensure that the quantum algorithm is implemented to a given accuracy specified by an approximation error ϵ.

Among the known decomposition algorithms, we'll focus on a technique called Repeat-Until-Success (RUS) and first proposed in 2014 by [1]. RUS is a non-deterministic family of decomposition algorithms aiming at decomposing a given gate U using only gates from a certain set. The working principle of RUS relies on some steps that are common to many quantum algorithms: (a) we take the target qubits initialized in an arbitrary state $|\psi\rangle$ and we entangle them with additional ancilla qubits initialized in a known state; (b) we apply gates from the chosen set on the target and ancilla qubits; (c) we measure the state of the ancilla qubits and we take action conditioned on the measurement outcome. With an appropriate choice of the applied gates, we can achieve that some of the possible measurement outcomes correspond to the target qubits being collapsed in the target state $U|\psi\rangle$. We call these success outcomes and if they are obtained then the algorithm is finished. All the other outcomes correspond to the target qubits begin collapsed in an undesired state, and we call them failure outcomes. In those cases a recovery operation is applied to reset the qubits to the initial state, and the procedure is repeated until a success outcome is obtained, hence the name "Repeat-Until-Success".

RUS has been proven to be a powerful tool for quantum computation, as it finds applications for practical problems. For instance, RUS can be applied in the context of fault-tolerant quantum computation for implementing fault-tolerant gates that require magic state distillation, such as the T gate. Magic state distillation

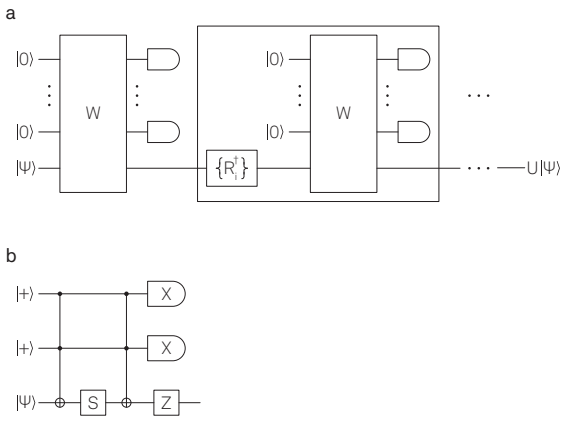


Figure 1. (a) Circuit representation of a typical RUS algorithm. (b) Example of a circuit that can be used in a RUS algorithm.

consists in the preparation of a specific state, starting from many qubits in mixed quantum states, and it has been shown that RUS provides an algorithm to implement it [2]. Furthermore, is has been shown [1] that RUS can ben used to decompose any single-qubit unitary using only gates from the universal set $\{H, T\}$ and drastically reducing the number of necessary T gates as compared to other decomposition algorithms, hence providing a better performance. Finally, RUS finds application in the pioneering field of quantum machine learning as well, as it can be used to implement quantum neural networks [3].

We will start this blog post with a deeper dive into the concept of Repeat-Until-Success algorithms. We will follow by presenting how RUS can be used to create a quantum neural network, one of the most exciting application of this technique. Finally, we will provide a tabletop demonstration of how its is possible to implement a RUS algorithm on superconducting qubits with real-time feedback using Zurich Instruments devices.

Concept

Repeat-Until-Success algorithms aim at applying a desired gate U to an arbitrary input state, using only gates from a (usually universal) set $\mathcal{U}$. RUS algorithms have the following general structure exemplarized in Figure 1(a):

1. m ancilla qubits are prepared in an initial known state (e.g. $|0\rangle^{\otimes m}$);
2. Given an input state $|\psi\rangle$ on n qubits (called target qubits), a unitary W is applied to all the $n+m$ qubits using gates from $\mathcal{U}$;
3. The ancilla qubits are measured in the computational basis. The output of the circuit corresponds to the state of the n target qubits after the measurement, which we denote by $\Phi_i|\psi\rangle$. Φ_i is a quantum channel that depends on the ancilla measurement outcome $i \in \{0,1\}^m$;
4. The measurement outcomes are divided into two sets: "success" and "failure". If the measurement outcome indicates "failure", a recovery operation is applied to the ancilla and target qubits in order to bring them again to the initial state, and the procedure is repeated. If the measurement outcome indicates "success", the output state is equal to the desired output $U|\psi\rangle$, and the algorithm finishes.

The example 3-qubit circuit shown in Figure 1(b) illustrates this concept. The circuit makes use of two ancilla qubits which are initialized in $|+\rangle$, and read out at the end of the ciruit. If the measurement outcome is 00, then the circuit has effectively applied the gate $U = \frac{I+2iZ}{\sqrt{5}}$ to the state $|\psi\rangle$ of the target qubit. For any other measurement outcome $|\psi\rangle$ is left unchanged, i.e. the circuit has implemented the identity on the target qubit. Therefore, one can implement U with RUS by repeatedly applying this circuit until the measurement outcome is 00.

The 3-qubit example is of didactic value, but is of a scale that is not used in practical applications. To get a sense of the complexity of the RUS concept when applied in the latter cases, we will look at an example designed for quantum machine learning.

RUS For Quantum Neural Networks

The Repeat-Until-Success scheme can be used to implement quantum neurons as the basic constituents of quantum neural networks. Quantum neural networks promise an advantage over the more well-known classical neural network in terms of their ability to train faster and to generalize on new data[4]. To understand the working principle of quantum neural networks, it's helpful to recall the concept of classical neural networks from which it is inspired. The constituents of classical neural networks are called neurons and are functions that take as input n real values $x_1, x_2, ..., x_n$ and produce one real-valued output. Both the inputs and the output can assume values only in a certain range, for instance $[0, 1]$ or $[-1, 1]$. In the following, we will use $[-1, 1]$ because of its convenience when it comes to quantum neural networks.

A neural network consists of layers, and each layer contains many neurons. The output of the neurons in one layer will become the input for the neurons of the following layer. The first layer of neurons receives the input of the problem, and the last layer produces an answer to the problem. This structure is illustrated in figure 2.

How can a neural network propagate information from one layer to another and eventually elaborate an answer? The crucial ingredient is the internal behavior of the neurons, i.e. how inputs are mapped to the output. Neurons calculate their output in the following way:

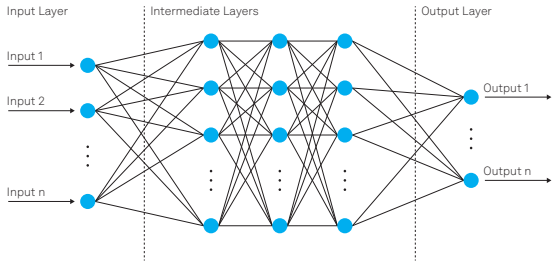


Figure 2. Scheme representing the structure of a neural network.

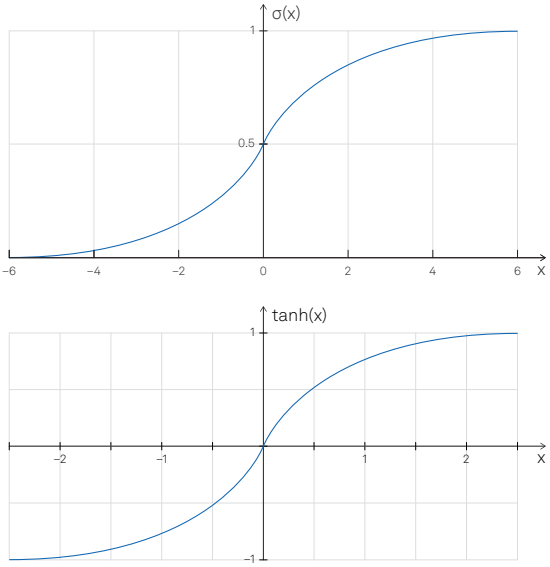


Figure 3. The sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$ (top) or the hyperbolic tangent $\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$ (bottom) are usually chosen as activation functions for neurons.

- the inputs x_i are summed with weights w_i and biased by b, obtaining

$$\theta = x_1 w_1 + x_2 w_2 + \cdots + x_n w_n + b.$$

  The weights determine which inputs will have more influence on the output of the neuron, while b is a constant offset for the weighted sum and provides the order of magnitude of the output;
- the weighted sum θ, which can assume arbitrary values, must be mapped to an output in the range $[-1, 1]$. This is done by passing θ through a (usually nonlinear) function that outputs a value in such a range. This function is called activation function, and functions typically used are the sigmoid or the hyperbolic tangent, as depicted in figure 3. The output, denoted by a, is called state of the neuron.

By optimizing the parameters of the network, i.e. the weights, the bias and the activation function, it is possible to train the neural network to find the desired answer to a problem.

Like a classical neural network, a quantum neural network consists of layers of neurons. However, in the quantum case the state of a neuron is associated to a qubit whose generic quantum state can be written as

$$R_y\left(a\frac{\pi}{2}+\frac{\pi}{2}\right)|0\rangle = \cos\left(a\frac{\pi}{4}+\frac{\pi}{4}\right)|0\rangle + \sin\left(a\frac{\pi}{4}+\frac{\pi}{4}\right)|1\rangle,$$

where $a \in [-1, 1]$. In the limit cases $a = -1$ and $a = 1$, the neuron is respectively in the quantum states $|0\rangle$ and $|1\rangle$, but for intermediate values of a the neuron is in a quantum superposition of $|0\rangle$ and $|1\rangle$.

The inputs of the neuron are given by n qubits in the state $|x\rangle = |x_1\rangle|x_2\rangle \cdots |x_n\rangle$. To create the equivalent of the classical weighted sum $\theta = x_1 w_1 + \cdots + x_n w_n + b$, we take an ancilla qubit initialized in $|0\rangle$ and for each input qubit, we apply to the ancilla a controlled rotation $R_y(2w_i)$ with the i-th input qubit used as control; this corresponds to applying $R_y(2x_i w_i)$. Lastly, we apply $R_y(2b)$ to the ancilla. After this procedure, the ancilla qubit will be in the state $R_y(2\theta)|0\rangle$.

The final step of the preparation of the quantum neuron is to perform a rotation $R_y(2q(\theta))$ on the output qubit, where q is a nonlinear, possibly sigmoid-like, activation function. We can implement this step using a RUS algorithm.

The corresponding circuit is shown in Figure 4. It acts on the output qubit conditioned by the outcome of the ancilla measurement. If the measurement returns 0, the circuit has applied $R_y(2q(\theta))$ on the output qubit, where $q(\theta) = \arctan(\tan^2(\theta))$ is a sigmoid-like function. If the measurement returns 1, the circuit has implemented $R_y(\pi/2)$ onto the output qubit. Therefore, one can apply RUS identifying the measurement outcome 0 with "success" and 1 with "failure". If a failure happens, we apply the recovery operation $R_y(-\pi/2)$ and we apply again the circuit of Figure 4. As soon a success happens, RUS has completed.

We can further improve the characteristics of the activation function and give a stronger threshold behavior to it. By threshold behavior we mean that for $\theta < \pi/4$ the output qubit is as close as possible to $R_y(\pi)|0\rangle = |1\rangle$ $(a = 1)$, and for $\theta < \pi/4$ the output qubit is as close as possible to $R_y(0)|0\rangle = |0\rangle$ $(a = -1)$. This is possible by concatenating multiple runs of the RUS proce-

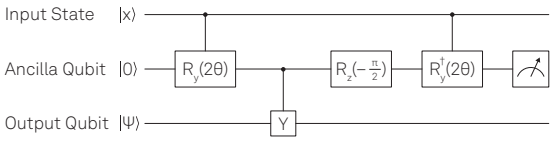


Figure 4. Building block of RUS implementing a quantum neuron. If the ancilla measurement returns 0, the circuit has applied $R_y(2q(\theta))$ on the output qubit, with $q(\theta) = \arctan(\tan^2(\theta))$. If the measurement returns 1, the circuit has applied $R_y(\pi/2)$ onto the output qubit.

dure. This means that we run RUS a first time, repeatedly applying the circuit in figure 4 and the recovery operation until a success happens. After that, we run RUS another time using the output of the previous run as the starting state for the output qubit. If we repeat this procedure for a total of k times, we will effectively apply the rotation $R_y(2q^{\circ k}(\theta))$ to the output qubit. Figure 5 shows the function $q^{\circ k}(\theta)$ for different values of k. It can be seen that for higher values of k, $q^{\circ k}(\theta)$ acquires a stronger threshold behavior.

Tabletop Demonstration

We are now going to show how to perform a repeat-until-success experiment designed for superconducting qubits with real-time feedback, using the Zurich Instruments QCCS. The experiment can be reproduced using a Jupyter Notebook example available on Zurich Instruments' github repository [5].

Experimental Setup

For this tabletop demonstration we use the setup shown in Figure 6, which consists of:

- a PQSC Programmable Quantum System Controller for feedback data processing
- an SHFQA Quantum Analyzer for qubit readout
- an SHFSG Signal Generator for qubit control

We also use an oscilloscope to view the readout and the control pulses generated by the SHFQA and SHFSG, respectively.

For the readout of qubits we use the SHFQA Quantum Analyzer. Each channel of this instrument is designed for parallel readout of up to 16 superconducting qubits on a single microwave line. In this experiment we will only read out one qubit. The SHFQA has integrated up- and down conversion capabilities and is hence able to generate the signals in the GHz frequency range that are required for typical readout resonators of superconducting qubits.

In this demonstration we simulate some qubit readouts with outcome 0 and some with outcome 1. To perform a simulated qubit readout we connect the signal

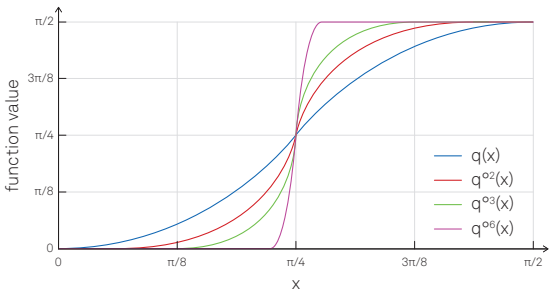


Figure 5. $q^{\circ k}(\theta)$ for different values of k. For higher k, $q^{\circ k}(\theta)$ exhibits a stronger threshold behavior.

output of the SHFQA to the signal input of the same channel, in a closed-loop configuration. We can then mimic the qubit state-dependent resonator response by using two pulses with a phase difference of 180 degrees: one pulse will simulate a qubit in $|0\rangle$ and the other pulse a qubit in $|1\rangle$. For more information about simulated qubit readout, please refer to the blog post Multiplexed Readout of Superconducting Qubits with the UHFQA [6].

For the control of superconducting qubits we use the SHFSG Signal Generator. This instrument is designed to perform the generation of arbitrary waveforms with frequencies from DC up to the GHz range, integrating frequency up conversion. Each channel of the SHFSG can be used to control one qubit.

The PQSC coordinates all the steps of the experiments by sending trigger signals and performing data processing to enable feedback operations with minimal latency, as required in a RUS experiment. The PQSC connects with both the SHFQA and the SHFSG with ZSync links, which have four roles:

- provide a common reference clock to all the instruments;
- distribute trigger signals from the PQSC to the devices to synchronize them to sub-nanoseconds levels;
- provide a data interface to transfer the results of qubit readout from the SHFQA to the PQSC;
- provide a data interface to transfer the processed feedback data from the PQSC to the SHFSG.

The PQSC features a memory called Readout Register Bank, which can store readout results measured by a Quantum Analyzer. The PQSC can then process the data stored in the Readout Register Bank with a feedback pipeline, whose output is eventually sent to other devices. This feedback pipeline has two operating modes:

- Register forwarding: a portion of the Readout Register Bank is forwarded to a ZSync output port without processing.
- Decoder Unit: the data from the Readout Register Bank is processed in the so-called Decoder Unit and the output is sent to a ZSync output port. The Decoder Unit is programmed by configuring a look-up table.

Now that we have described the setup used in this demonstration, we can start diving into the steps of the experiment.

The experiment: the essential skeleton of RUS

In this demonstration we want to illustrate how to perform the essential steps that are common to every repeat-until-success experiment, abstracting from the details of specific applications. For this purpose, we consider a one-qubit system initialized in $|0\rangle$ and we assume that the goal of RUS is to bring it in $|1\rangle$.

This trivial example shows how to perform the steps of a RUS experiment in the simplest and most generic way. We will implement the repeat-until-success procedure by dividing it in the following steps:

1. Gate application phase: the SHFSG sends a pulse to the qubit, which we call try pulse. The pulse represents the application of a gate, or a series of gates, to the qubit. The shape of this pulse is unimportant, and for simplicity, we use a square pulse.
2. Readout phase: the SHFQA reads out the state of the qubit and sends the result to the PQSC Readout Register Bank.
3. Feedback phase: depending on the readout result, the PQSC decides on a feedback response using the Decoder Unit and sends it to the SHFSG and SHFQA. The feedback response will be 1 if a failure has happened (and hence the SHFSG and SHFQA should repeat again the gate application and the readout). It will be 0 if a success has happened and therefore the experiment is finished. One can think of the feedback response as of a Boolean value answering the question "Should the experiment be repeated?". In our single-qubit example, where the goal is to bring the qubit into the state $|1\rangle$, this means that if the readout result is 0, the feedback response will be 1, while if the readout result is 1, the feedback response will be 0.
4. Feedback response evaluation: the SHFQA and the SHFSG receive the feedback response and react to it. They start again from the first step if the feedback response is a 1, or they stop if the feedback response is 0. We program the SHFSG to play a final success pulse to mark the end of the experiment.

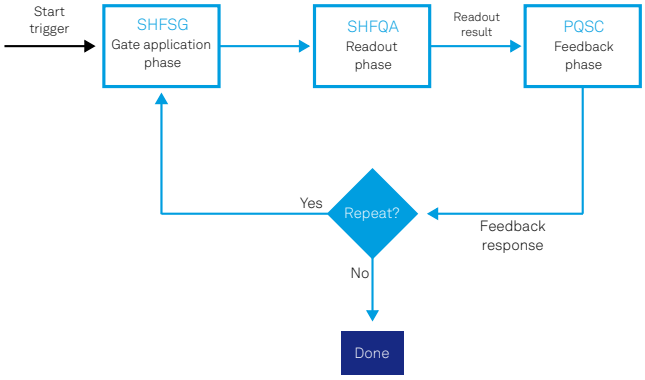


Figure 7. Implementation of the repeat-until-success logic on our setup.

The experiment will begin with an initial start trigger signal sent from the PQSC to the other instruments, after which the steps described above will start. The flow diagram in Figure 7 shows the logic of the experiment.

The timing of the experiment

In the example Jupyter Notebook provided with this application note, we show how to configure the instruments to work together to execute the steps described above. The most relevant aspect when performing a repeat-until-success experiment is the timing of the

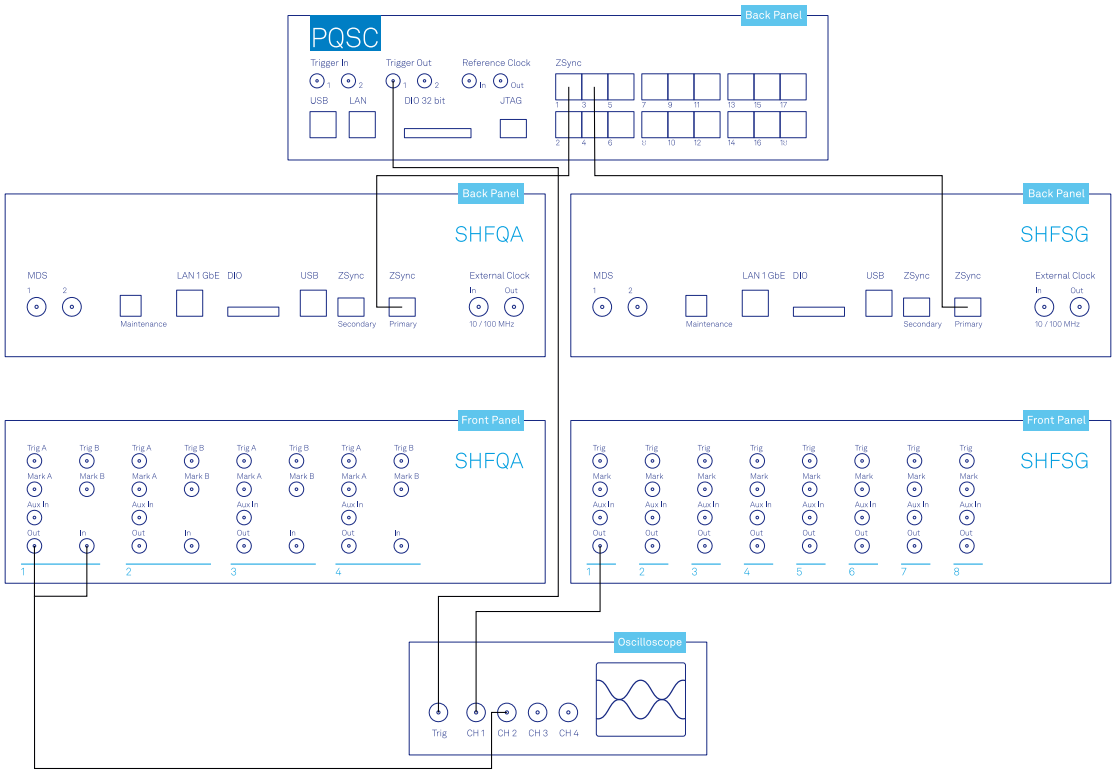


Figure 6. Diagram of the instruments used in the demonstration, together with their connections.

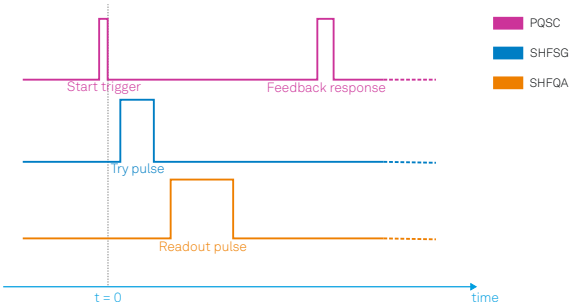


Figure 8. Idealized sequence of signals in one cycle of RUS. The PQSC line shows the digital signals sent by the PQSC to the SHFSG and SHFQA. The experiment starts at $t = 0$ with start trigger from the PQSC, then the SHFSG plays the try pulse. The SHFQA follows by performing the readout, and finally the PQSC sends the feedback response calculated from the readout result.

operations. In particular, we want to focus on the following two requirements:

- Before starting the readout, the SHFQA must wait until the SHFSG has finished playing the try pulse;
- There is a finite feedback latency owing to the propagation delay of the readout result to the PQSC and the time needed to calculate the feedback response by the PQSC. The SHFSG and SHFQA must read the feedback response only after this time has passed.

In Figure 8 we show a sketch the expected sequence of pulses and signals in one cycle of RUS.

The first requirement can be easily satisfied by writing a custom sequence program for the SHFQA. The SHFQA sequence program controls the timing of the Quantum Analyzer functional unit with the **`startQA`** instruction. In addition, the SHFQA architecture contains a dedicated AWG core called Player Zero for generating zero-valued spacer pulses with the **`playZero`** instruction. These can be used to match control pulse playback on the AWG cores of the SHFSG or HDAWG.

If a **`startQA`** is preceded by a **`playZero`** instruction, these two are executed in parallel. However, two **`playZero`** instructions are executed sequentially, and this allows us to delay the beginning of the qubit readout in our experiment. To that end, we program the SHFQA with the instructions shown in Figure 10.

First, we use the instruction **`playZero`** for a time interval equal to the duration of the try pulse being played by the SHFSG. We then use another **`playZero`** instruction followed by a **`startQA`**. The second **`playZero`** starts back-to-back with the first **`playZero`**, and simultaneously with the **`startQA`** instruction. In this way, we delay the execution of the readout until the SHFSG has finished playing the try pulse. The code snippets in Figure 9 show the sequence programs for the SHFSG and SHFQA.

With respect to the second timing requirement, we

```
// SHFSG sequence program
do {
  playWave(try_pulse);
  playZero(qa_len);
  failure = <get PQSC feedback>
} while (failure);

playWave(success_pulse);
```

```
// SHFQA sequence program
do {
    playZero(try_pulse_len);
    playZero(qa_length);
    startQA(...);
    failure = <get PQSC feedback>
} while (failure);
```

Figure 9. Core part of the sequence programs to control the simultaneous playback and readout on the SHFSG and SHFQA.

use the sequence instruction **`getFeedback`** to tell the SHFSG and SHFQA to read the PQSC feedback response from their ZSync connections. This instruction takes as a parameter the delay (measured from the last start trigger) that the instrument should wait before reading the feedback response. Our task is hence to estimate the quantity called feedback latency, i.e. is the amount of time between the initial start trigger and the moment when the PQSC makes the feedback response available on the ZSync links. This latency depends on the duration of the try pulse, on the time needed by the Quantum Analyzer to complete the readout, and also on the time needed by the PQSC to process the readout result into the feedback response. The package **`zhinst-utils`** provides a routine that allows us to calculate the feedback latency based on a model of the PQSC feedback pipeline. We only have to provide to this routine:

- a description of the feedback system (such as: the Quantum Analyzer is an SHFQA and the Signal Generator is an SHFSG);
- the duration of user-defined operations in the Quantum Analyzer (such as the integration length and our delay introduced by **`playZero`**). We call this quantity QA delay.

It looks like we solved the second requirement as well, but we should be careful. What about the iterations of RUS other than the first one? The delay to be waited that we pass to the sequence instruction **`getFeedback`** must always be measured with respect to the initial start trigger. This means that we would have to recalculate the feedback latency at each try cycle in order to include the latency of all the previous cycles. However, there's a fact regarding the feedback latency that allows us to speed up the calculations:

> If and only if the duration of each try cycle is an integer multiple of 200 ns, then the new feedback latency at the n-th cycle is equal to the feedback latency of a single cycle plus the duration of all the previous cycles!

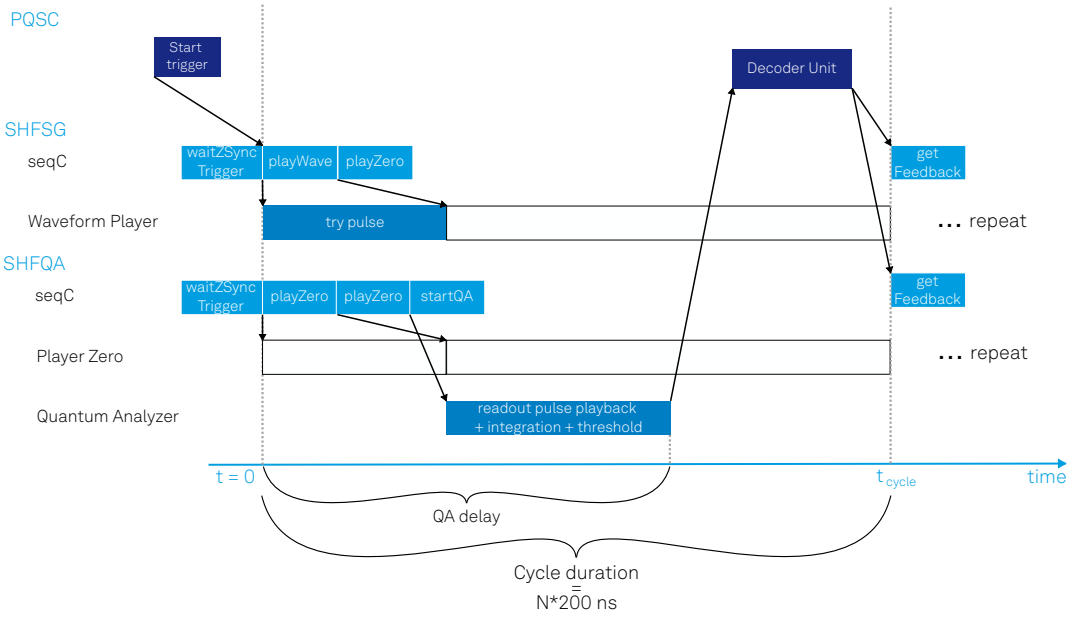


Figure 10. Timing of the sequence instructions and the corresponding waves played in one cycle of RUS. Notice the usage of **playZero** in the SHFQA sequencer program in order to delay the execution of the readout after the SHFSG waveform playback, as explained before.

If we engineer the cycle duration to be an integer multiple of 200 ns, we can avoid recalculating the feedback latency with the model at each iteration, and we can simply increment the previous feedback latency by one cycle duration. We can achieve that by choosing appropriate duration of the **playZero** instructions. In the Jupyter Notebook provided with this application note, we show how to implement this with our API.

Figure 10 also shows the QA delay and the cycle duration. The code snippets in Figure 11 show how to implement it in the sequence programs.

Results

Now we have all the tools to run the experiment. In our Jupyter notebook, we simulate two failures (qubit in $|0\rangle$) followed by one success (qubit in $|1\rangle$). After the final success event, the SHFSG will play a "success pulse" to mark the end of the experiment. In order to distinguish the success pulse from the try pulses, the success pulse will have double the duration and half the amplitude of the try pulses. We can sketch the sequence of pulses played by the SHFSG and SHFQA that we expect to see in the oscilloscope, as depicted in Figure 12(a). The figure shows the pulse envelopes, while the actual signals will also include sinusoidal modulation.

We can compare this sketch with the signals actually displayed on the oscilloscope, as shown in Figure 12(b). We can see that the actual signals match with the expected ones.

```
// SHFSG sequence program
do {
  playWave(try_pulse);
  playZero(qa_len);
  failure = getFeedback(
    ZSYNC_DATA_PQSC_DECODER,
    feedback_latency
  );
  // cycle_len = try_pulse_len + qa_len = N*200 ns
  feedback_latency += cycle_len;
} while (failure);
playWave(success_pulse);
```

```
// SHFQA seqC program
do {
  playZero(try_pulse_len);
  playZero(qa_len);
  startQA(...);
  failure = getFeedback(
    ZSYNC_DATA_PQSC_DECODER,
    feedback_latency
  );
  // cycle_len = try_pulse_len + qa_len = N*200 ns
  feedback_latency += cycle_len;
} while (failure);
```

Figure 11. Core part of the sequence programs of the SHFSG and SHFQA to control the conditional feedback action The parameter **feedback_latency** is initialized with the Python routine **get_latency**.

Conclusions

In this application note we have presented Repeat-Until-Success, a family of algorithms to perform gate

decomposition that finds applications to many problems of quantum computation. We have in particular explained how RUS can be used in the context of quantum machine learning to create quantum neural networks. Finally, we have demonstrated how to perform RUS on superconducting qubits using the Zurich Instruments QCCS, explaining how to take the most out of the real-time feedback capabilities offered by our instruments.

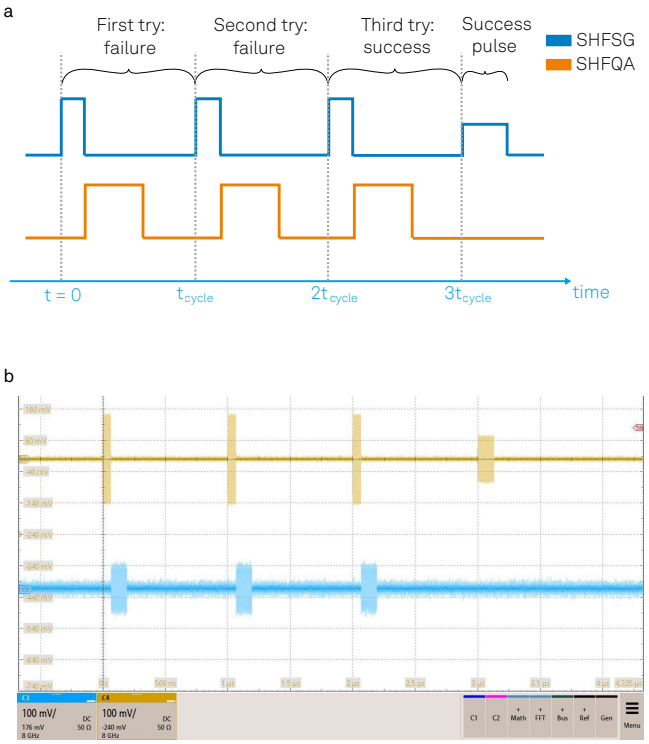


Figure 12. (a) Sketch of the envelopes of the expected signals played by the SHFSG and SHFQA in the execution of RUS with two simulated failures and one simulated success. (b) Oscilloscope measurement of the signals produced by the SHFSG (yellow) and SHFQA (blue) during a RUS experiment where we simulate 3 failures and one success.

References

[1] Adam Paetznick and Krysta M. Svore. Repeat-until-success: Non-deterministic decomposition of single-qubit unitaries. arXiv:1311.1074, 2013.

[2] Sergey Bravyi and Alexei Kitaev. Universal quantum computation with ideal clifford gates and noisy ancillas. Physical Review A, 71(2), 2005.

[3] Yudong Cao, Gian Giacomo Guerreschi, and Alán Aspuru-Guzik. Quantum neuron: an elementary building block for machine learning on quantum computers. arXiv:1711.11240, 2017.

[4] Amira Abbas, David Sutter, Christa Zoufal, Aurelien Lucchi, Alessio Figalli, and Stefan Woerner. The power of quantum neural networks. Nature Computational Science, 1:403, 2021.

[5] Elena Acinapura. Jupyter notebook: Repeat-Until-Success algorithm on superconducting qubits.

[6] Max Ruckriegel. Multiplexed Readout of Superconducting Qubits with the UHFQA, 2020.

Disclaimer: The content of this document is provided by Zurich Instruments 'as is'. Zurich Instruments makes no warranties with respect to the accuracy or completeness of the content of this document and reserves the right to make changes to the specification at any time without notice. All trademarks are the property of their respective owners.