QUANTUM MACHINES

USE CASE

QUANTUM ORCHESTRATION FOR NV & Other Defect Centers

Find out how you can leverage the Quantum Orchestration Platform to perform groundbreaking experiments in your NV center lab with these real-life use cases.

DYNAMICAL DECOUPLING (XY8-N AS AN EXAMPLE)

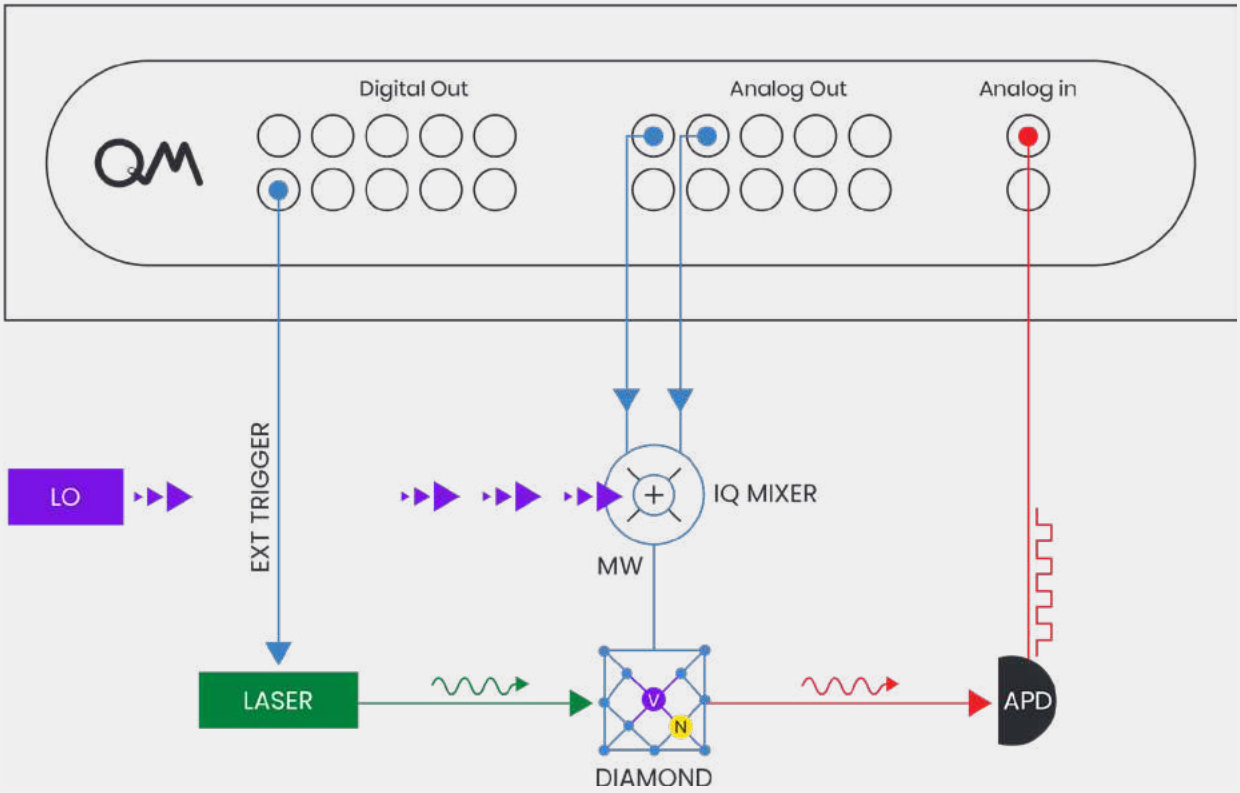


Fig. 1 Experimental setup for dynamical decoupling using the NV center in diamond. The MW control signal is generated using IQ modulation with two analog outputs of the OPX+.

NV centers are excellent systems for sensing. Not only are they highly sensitive to various types of forces (magnetic, electric, strain, etc.), but their small size allows for spatial resolutions of a few nm. In this example, we want to demonstrate how one of the most common sensing methods with NV centers, namely dynamical decoupling (DD), can be effortlessly implemented using the Quantum Orchestration Platform (QOP). It's a great showcase of the real-time paradigm the OPX+ operates on, which removes the need for creating long arbitrary waveforms before starting the experiment.

Dynamical decoupling is typically used in quantum control to prolong the coherence of the spin system. This is achieved by a periodic sequence of control pulses, which refocus the environmental effects and hence attenuate noise. Since phases accumulated from frequency components close to the pulse spacing are being enhanced, DD effectively acts as a frequency filter. Thus, the technique can be used for noise spectroscopy.

Fig. 1 shows the experimental setup. The OPX+ is controlling the laser via a digital marker output. Two analog outputs of the OPX+ are used for IQ modulation of the MW signal controlling the NV spin. The output pulses of the APD, which collects the fluorescence of the NV center, are sent to an analog input of the OPX+ for time tagging.

In this example, we want to focus on one of the most common DD sequences used for NV-based sensing, namely the XY8-N sequence. The XY8-N sequence consists of the following pulse

sequence: $(\pi_x-\pi_y-\pi_x-\pi_y-\pi_y-\pi_x-\pi_y-\pi_x)^N$, where N is the so-called XY8 order and the indices x and y correspond to the rotation axis in the rotating frame. The pulses are spaced equidistantly with a spacing of τ. The entire sequence is shown in Fig. 2a.

The XY8 sequence is applied after the NV electron spin is brought into a superposition state |-1>+|0> by an initial $(\pi/2)_x$-pulse. After the decoupling sequence, the spin state is mapped onto the spin population by a final $(\pi/2)_x$-pulse (see Fig. 2a). Finally, the NV center is read out optically by a laser pulse, which simultaneously repolarizes the electron spin state. Fig. 2b shows the filter function of the sequence. The central frequency $\nu=1/2\tau$ is defined by the periodicity of the pulses, while the width $\Delta\nu=1/N\tau$ depends on the total acquisition time $T=N\tau$

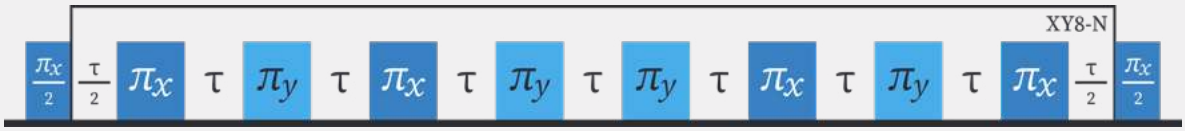


Fig. 2a XY8-N sequence with waiting time τ.

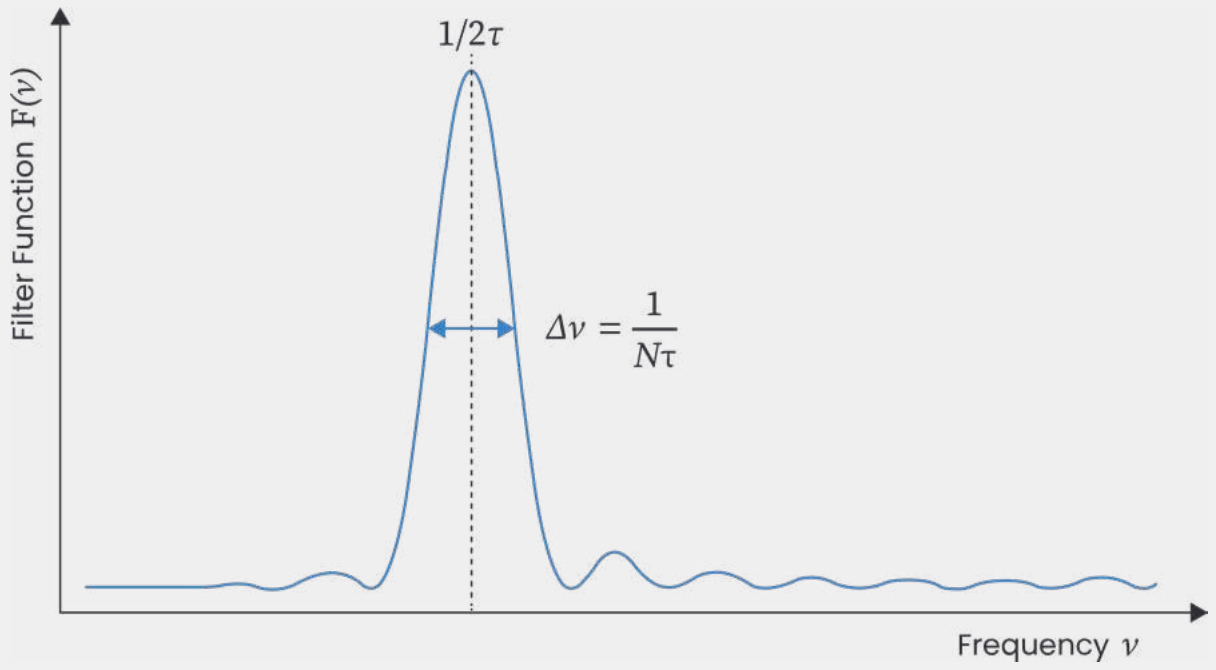


Fig. 2b Filter function of the XY8-N sequence. The central frequency is determined by the waiting time τ, while the width is defined by the total acquisition time $T=N\tau$.

The example QUA program runs a `for_each` loop, which iterates through a given list of τ values. Additionally, an outer `for` for loop averages over many sweeps. In QUA, we can define macros that make code shorter and clearer. In this example, we use the macro `xy8_n(n)` to create all the XY8 sequence pulses in a single line. The macro dynamically creates the XY8 sequence according to the order specified in parameter n. The macro creates all pulses and wait times by looping over another helper macro, `xy8_block()`, which creates the 8 pulses of a single XY8 block. The π_y pulses are generated by rotating the frame of the spin using the built-in `frame_rotation()` function. Then, the frame is reset back into its initial state by calling `reset_frame()`.

```
from qm.QuantumMachinesManager import QuantumMachinesManager
from qm.qua import *
from qm import SimulationConfig
import matplotlib.pyplot as plt
import numpy as np


NV_IF = 100e6
t_min = 4
t_max = 100
dt = 1
t_vec = np.arange(t_min, t_max, dt)

repsN = 3
simulate = True


with program() as xy8:
    # Realtime FPGA variables
    a = declare(int)  # For averages
    i = declare(int)  # For XY8-N
    t = declare(int)  # For tau
    times = declare(int, size=100)  # Time-Tagging
    counts = declare(int)  # Counts
    counts_ref = declare(int)
    diff = declare(int)  # Diff in counts between counts & counts_ref
    counts_st = declare_stream()  # Streams for server processing
    counts_ref_st = declare_stream()
    diff_st = declare_stream()

```

```
31
32          with for_(a, 0, a < 1e6, a + 1):
33              play("laser", "qubit")
34
35              with for_(t, t_min, t <= t_max, t+dt):  # Implicit Align
36                  # Play meas (pi/2 pulse at x)
37                  play("pi_half", "qubit")
38                  xy8_n(repsN)
39                  play("pi_half", "qubit")
40                  measure("readout", "qubit", None, time_tagging.raw(times, 300,
41                         counts))
42                  # Time tagging done here, in real time
43
44                  # Plays ref (pi/2 pulse at -x)
45                  play("pi_half", "qubit")
46                  xy8_n(repsN)
47                  frame_rotation(np.pi, "qubit")
48                  play("pi_half", "qubit")
49                  reset_frame('qubit')  # Such that next tau would start in x.
50                  measure("readout", "qubit", None, time_tagging.raw(times, 300,
51                         counts_ref))
52                  # Time tagging done here, in real time
53
54                  # save counts:
55                  assign(diff, counts - counts_ref)
56                  save(counts, counts_st)
57                  save(counts_ref, counts_ref_st)
58                  save(diff, diff_st)
59
60          with stream_processing():
61              counts_st.buffer(len(t_vec)).average().save("dd")
62              counts_ref_st.buffer(len(t_vec)).average().save("ddref")
63              diff_st.buffer(len(t_vec)).average().save("diff")
64
65
66      qmm = QuantumMachinesManager()
67      qm = qmm.open_qm(config)
68      job = qm.execute(xy8, duration_limit=0, time_limit=0)
69
70
71      def xy8_n(n):
72          # Assumes it starts frame at x, if not, reset_frame before
73          wait(t, "qubit")
```

```
74
75              xy8_block()
76
77              with for_(i, 0, i < n - 1, i + 1):
78                  wait(2 * t, "qubit")
79                  xy8_block()
80
81              wait(t, "qubit")
82
83
84          def xy8_block():
85              play("pi", "qubit")  # 1 X
86              wait(2 * t, "qubit")
87
88              frame_rotation(np.pi / 2, "qubit")
89              play("pi", "qubit")  # 2 Y
90              wait(2 * t, "qubit")
91
92              reset_frame("qubit")
93              play("pi", "qubit")  # 3 X
94              wait(2 * t, "qubit")
95
96              frame_rotation(np.pi / 2, "qubit")
97              play("pi", "qubit")  # 4 Y
98              wait(2 * t, "qubit")
99
100             play("pi", "qubit")  # 5 Y
101             wait(2 * t, "qubit")
102
103             reset_frame("qubit")
104             play("pi", "qubit")  # 6 X
105             wait(2 * t, "qubit")
106
107             frame_rotation(np.pi / 2, "qubit")
108             play("pi", "qubit")  # 7 Y
109             wait(2 * t, "qubit")
110
111             reset_frame("qubit")
112             play("pi", "qubit")  # 8 X
113
```

QUA code for the XY8-N dynamical decoupling sequence

For the NV centers optical readout, we utilize the built-in time tagging functionality of the QOP. A call of the `measure` statement starts the time tagging. A time tag for each detected pulse from the APD is saved into the real-time array `times`, and the total number of detected photons is saved into the integer variable `counts`. Concurrently, the `measure` statement generates a readout pulse, which here is the trigger pulse going to the laser system. The same sequence is repeated a second time with a final $(\pi/2)_{-x}$. The photons detected during this are saved in the variable `counts_ref` and act as a reference signal.

At the end of each loop, we save the photon counts into a so-called stream, using the `save()` function. These streams allow streaming data to the client PC while the program is still running. The stream processing feature also offers a rich library of data processing functions which can be applied to streams. The QOP server performs the processing before sending it to the client PC. It can significantly reduce the amount of transferred data by limiting it to the user's preferred result. In this example, we use the `average()` function to average the data while streaming.

ADDING A RANDOMIZED PHASE TO THE XY8 SEQUENCE

A known issue with pulsed DD, is the emergence of spurious harmonics due to the pulses' finite length. A theoretically "easy" solution is to introduce a random global phase to each iteration of the experiment [1]. This solution quickly becomes very taxing when trying to utilize it by a-priory waveform creation, because the number of repetitions we need to upload has to be very large $(N \gg 100)$ for the phase to be considered random. As the OPX+ generates this sequence on the fly, we can utilize its internal random number generator to easily create this randomized XY8-N sequence by adding a couple lines of code to the one above:

First, we'll define an additional variable ϕ.

```
22    ...
23    phi = declare(fixed,value=0) # Random phase
24    ...
```

Second, we'll add a line assigning a new random phase for each iteration using the command `Random().rand_fixed()` which returns a random number between 0 and 1.

```
34        ...
35        assign(phi,Random().rand_fixed()*2*np.pi)
36        play("pi_half", "qubit")
37        xy8_n(repsN,phi)
38        ...
```

The last step would be to add a `frame_rotation()` command before each π_x and `reset_frame()` at the end of the xy8_block macro.

```
111     ...
112     frame_rotation(phi, "qubit")
113     play("pi", "qubit")  # 8 X
114     reset_frame("qubit")
115     ...
```

ACTIVE INITIALIZATION AND SINGLE SHOT READOUT OF A NUCLEAR SPIN

One of the NV's major strengths is its ability to utilize its nuclear spin environment as a quantum register for complex protocols that involve more than a single qubit. This utility was already shown in many experiments, including quantum sensing [2,3], quantum computation [4], and quantum networks [5]. All these experiments share the need to initialize the nuclear spin to a known state, and efficiently readout that state via manipulation of the NV electron spin. In order to actively initialize the nuclear spin, we first need to be able to determine its state, so we will first look at the single-shot readout (SSRO) [6].

Here we will discuss a protocol for a SSRO on the intrinsic ^{14}N nuclear spin. In general the single-shot readout consists of a conditional rotation on the NV, a laser pulse to read its state, and then repeating these two steps N times to accumulate enough photons for state separation. As the ^{14}N is a spin 1, two consecutive SSRO steps are necessary. In the first SSRO, the conditional π pulse will be performed if the nuclear spin is at the $|0_N\rangle$ sublevel. This will tell us whether the nuclear spin is at the $|0_N\rangle$ sublevel or at the $|\pm 1_N\rangle$ sublevels. If we are at $|0_N\rangle$, the readout is done, if not, then a second SSRO needs to be initiated to determine whether the nuclear spin is at the $|+1_N\rangle$ or $|-1_N\rangle$ sublevel. The ability to skip the second SSRO step exemplifies the OPX+'s capabilities, as without real-time measurement based decision making, one could waste precious nuclear spin coherence time on an unnecessary step.

As the SSRO is usually part of a larger sequence, the example QUA program is written as a macro named `SSRO()`, making it easier to use for different protocols. The basic building block of the macro is composed from a conditional rotation, defined by the `CnNOTe()` macro and the `measure()` command.

Let's first start with the most basic building block: the C_nNOT_e gate on the electron spin. This gate inverts the electron spin only for a specific nuclear spin state. This is achieved by choosing the correct frequency in the hyperfine resolved ODMR spectra of the NV center. In the QUA script below, we define this gate operation in function `CnNOTe()`. The `CnNOTe()` macro accepts the nuclear spin state for which the electron spin should be flipped as an integer

(-1: $|-1_n\rangle$, +1: $|+1_n\rangle$, 0: $|0_n\rangle$). It calculates the corresponding new intermediate frequency according to the given nuclear spin state by adding or subtracting the hyperfine splitting value *(hf_splitting ≈ 2.16 MHz)* from the central frequency f_0 of the NV centers hyperfine spectra. We use the built-in QUA function `update_frequency()` to update the *quantum element 'sensor' frequency*, which corresponds to the sensor spin. As a result, all of the following pulses played to this *quantum element* will be at this new frequency. Finally, we play a π-pulse to the sensor spin using the `play` command, which enables us to dynamically update the pulse's amplitude and duration, to reduce the pulse's frequency bandwidth.

```
def CnNOTe(condition_state):
    """
    CNOT-gate on the electron spin.
    condition_state is in [-1, 0, 1] for a spin 1 nuclear or [-1, 1] for a
    spin half nuclear and gives the nuclear spin
    state for which the electron spin is flipped.
    """
    align(*all_elements)
    update_frequency("sensor", NV_IF + condition_state * hf_splitting)
    play("pi_x"*amp(0.1), "sensor", duration= (pi_length/4)*10)
    update_frequency("sensor", NV_IF)
    align(*all_elements)
```

QUA Code for the C_nNOT_e macro

The `measure()` statement has a time tagging module, which counts the arriving photons while simultaneously executing the laser pulse. This module allows time tagging of pulses via the analog inputs of the OPX+. The arrival times of all photons detected during the detection window are saved into the real-time array time tags. Additionally, the total number of detected photons is saved into the variable *counts*. These two operations are written within a for loop that runs for N times to accumulate enough photons for state differentiation.

Once the *for* loop is over, we use the *if* statement to compare the total number of photons against a predefined threshold, SSRO_*threshold*. If the number of photons is lower than the threshold, the nuclear spin is at the $|0_N\rangle$, and we can continue with the experiment. If the number of photons is higher than the threshold, we need to determine whether the nuclear spin is in the $|+1_N\rangle$ or $|-1_N\rangle$ states. This is done by repeating the SSRO, this time with the `CnNOTe()` receiving the integer 1. After the loop, we know the nuclear spin is at the $|+1_N\rangle$ if the photon count is lower than the threshold, and $|-1_N\rangle$ if it is higher.

```
def SSRO(N, result):
    """Determine the state of the nuclear spin"""
    i = declare(int)
    res_vec = declare(int, size=10)
    counts = declare(int)
    ssro_count = declare(int,value=0)
    # run N repetitions
    with for_(i, 0, i < N, i + 1):
        wait(100, "sensor")
        CnNOTe(0)
        measure(
            "readout",
            "sensor",
            None,
            time_tagging.analog(res_vec, 300, counts),
        )
        assign(ssro_count, ssro_count + counts) # sums up the total photons
             detected during the SSRO
    # compare photon count to threshold and save result in variable "state" or
             continue to the next step
    with if_(ssro_count < SSRO_threshold):
        assign(result, 0)
    with else_():
        assign(ssro_count, 0)
        with for_(i, 0, i < N, i + 1):
            wait(100, "sensor")
            CnNOTe(1)
            measure(
                "readout",
                "sensor",
                None,
                time_tagging.analog(res_vec, 300, counts),
            )
            assign(ssro_count, ssro_count + counts) # sums up the total photons
                    detected during the SSRO
        # compare photon count to threshold and save result in variable "state"
        with if_(ssro_count < SSRO_threshold):
            assign(result, 1)
        with else_():
            assign(result, -1)

```

QUA Code for the single shot readout macro in the case of ^{14}N nuclear spin.

The OPX+ real-time capabilities become even more prominent when one wants to initialize the nuclear spin to a specific state ($|0_N\rangle$ for example). Instead of postselection, which wastes time performing unwanted experiments or employing an elaborate Swap gate (as it is a 3 level system), which takes a long time and usually has limited fidelity, the OPX+ enables the user to precisely perform the necessary operation based on the nuclear spin SSRO result.

In the case of nuclear spin initialization, we would start, like above, with a SSRO on the $|0_N\rangle$ sublevel. If the nuclear spin is at that level, we are done. If not, a second SSRO is performed to determine whether the nuclear spin is at the $|+1_N\rangle$ or $|-1_N\rangle$. Based on the result of the second SSRO, an RF π pulse on resonance with the desired transition can then be applied on the nuclear spin to bring it back to the $|0_N\rangle$. If a π pulse was performed, we can repeat the process to make sure the nuclear spin is in the desired state.

The code example is written in the `init_nuclear_spin()`, which starts with the `SSRO()` macro to initially determine the nuclear spin state. Then, if it is not in the $|0_N\rangle$, the code enters an indeterministic while loop, until the `SSRO()` determines the nuclear spin is in the $|0_N\rangle$ state. After each SSRO, a selective π pulse will be applied to the nuclear spin with a frequency determined by the result of the SSRO.

```
def init_nuclear_spin():
  state = declare(int)
  SSRO(N_SSRO, state)
  with while_(state == ~0):
     with if_(state == -1):
        # assigning a frequency on resonanace with the 0 -> -1
           nuclear transition
        update_frequency("memory", memory_IF_m1)
        play("pi_x", "memory")
        update_frequency("memory", memory_IF_p1)
     with else_():
        play("pi_x", "memory")
      SSRO(N_SSRO, state)
```

QUA Code for active initialization of the ^{14}N nuclear spin.

NANOSCALE NMR WITH A NUCLEAR SPIN MEMORY

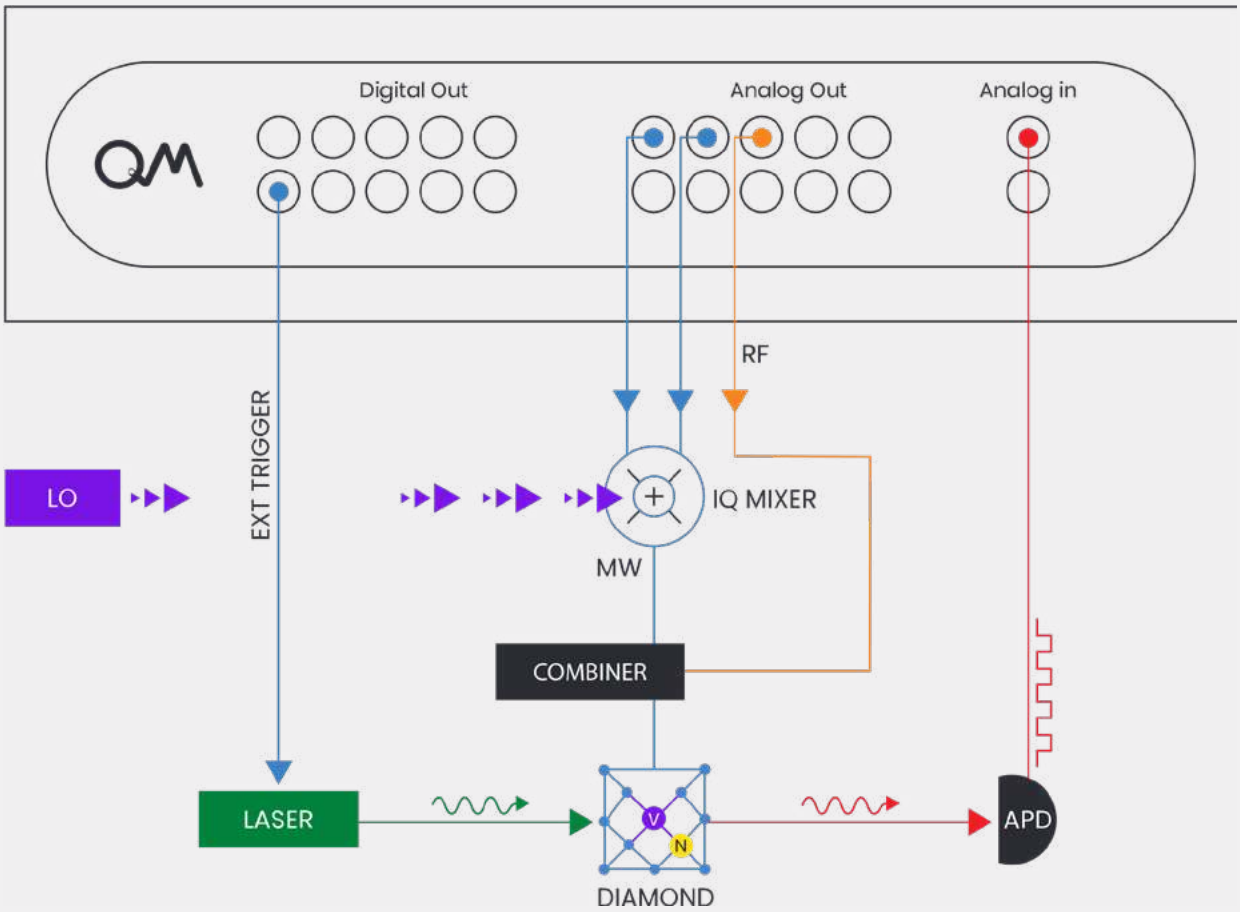


Fig. 3 *Setup for nanoscale NMR using a nuclear spin memory. The MW control sequence for the electron spin is created using IQ modulation with two analog outputs of the OPX+ (blue). The RF signal for nuclear spin manipulation is directly synthesized with the OPX+ (orange). The pulses of the APD are time-tagged by the OPX+ via one of the analog inputs (red).*

With the QOP and QUA, we can write even the most complex experiments as short and clear single programs. To demonstrate this, let's look at an NV-based NMR experiment that utilizes a nuclear spin as an additional memory [2,3]. It is possible to drastically enhance the spectral resolution by using the long lifetime of the nuclear spin as a resource. This technique allows nanoscale NMR with chemical contrast, e.g. [7].

NMR using NV centers is typically based on imprinting the Larmor precession of sample spins into the phase of a superposition state of the NV electron spin state [8]. We can achieve this, for example, by using Ramsey spectroscopy, Hahn echo sequences or dynamical decoupling. The spectral resolution of these methods is limited by the duration of the phase accumulation period, and consequently, is limited by the coherence time T_2^{sens} of the sensor spin. It's possible to overcome this limitation by performing correlation spectroscopy [9] . Here, the signal is generated by correlating the results of two subsequent phase accumulation sequences separated by the correlation time T_C. During

the correlation time, the phase information is stored (partially) in the polarization of the sensor spin. Hence, the possible correlation time, and therefore the spectral resolution, is limited by the spin-relaxation time T_1^{sens} (>T_2^{sens}) of the sensor.

It's possible to improve this even further by utilizing a memory spin, which has a much longer longitudinal lifetime. In the correlation spectroscopy experiment we discuss here, the information is stored on the nuclear spin (memory) instead of on the NV spin (sensor). As a result, the achievable correlation time is significantly increased. The intrinsic nitrogen nuclear spin of the NV center is a perfect candidate to act as this memory spin. It is strongly coupled to the NV center electron spin, which acts as the sensor, while its coupling to other electron or nuclear spin is negligible. When applying a strong bias magnetic field (3T) aligned along the NV-axis, we can achieve memory lifetimes T_2^{mem} on the order of several seconds. In this example, we assume that the used NV center incorporates a 14N nuclei with a 1-spin and the eigenstates $|+1_n\rangle$, $|-1_n\rangle$ and $|0_n\rangle$.

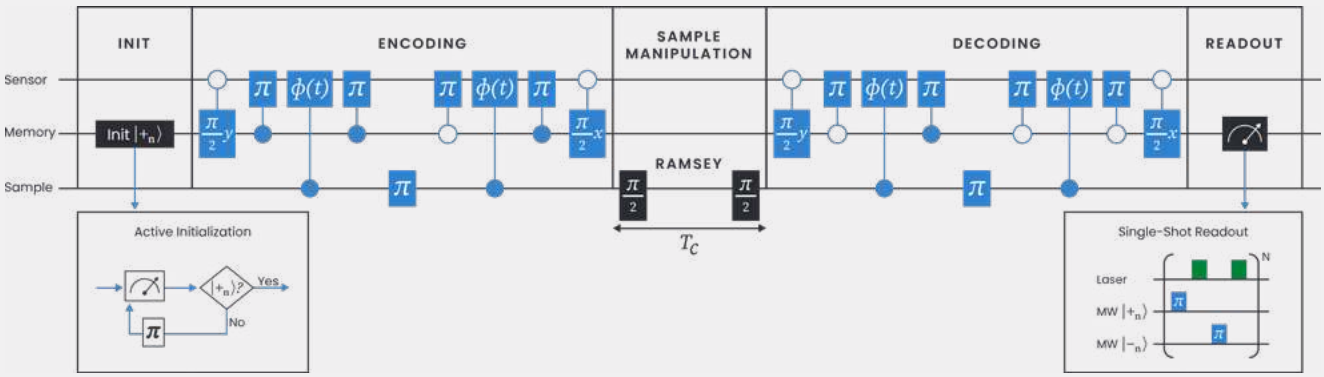


Fig. 4 A sequence of NMR with memory spin. It consists of active initialization of the memory spin, encoding, sample manipulation, decoding, and a final readout of the memory spin via single-shot readout.

Fig 4 shows the complete sequence. It consists of five steps: initialization, encoding, sample manipulation, decoding, and readout. The encoding aims to encode the sample sensor interaction into the spin population of the memory spin. First, the memory spin is brought from its initial state $|0_n\rangle$ into a superposition state by a $\pi/2$-pulse. Next, entanglement between sensor and memory is established for two phase-accumulation windows. The entanglement is created and destroyed by nuclear spin state selective π-pulses performed on the sensor spin. While the sensor and memory spins are entangled, the interaction with the sample spins leads to a phase accumulation on the memory spin superposition state. In between the two phase-accumulation periods, the sample is actively flipped by a resonant π-pulse. The final phase $\Delta\phi=\phi 2-\phi 1$, where ϕ_1 and ϕ_2 are the accumulated phases during the first and second accumulation window respectively, is then mapped into the memory spin population by a final $\pi/2$-pulse on the memory spin. The decoding sequence is identical, except for the conditions of the C_nNOT_e gates on the sensor spin.

For initialization and readout, we use the macros `SSRO()` and `init_nuclear_spin()`, that are described above.

```python
from qm.qua import *
from qm.QuantumMachinesManager import QuantumMachinesManager
from configuration import *
import numpy as np

all_elements = ["sensor", "sample", "memory"]
N_avg = 1e6
N_SSRO = 5000
hf_splitting = 2.16e6  # N14 hyperfine splitting (NV_IF is moved by -1.08e6)
t_e = 2000
tau_vec = [int(i) for i in np.arange(1e3, 5e4, 5e3)]
SSRO_threshold = 200


with program() as prog:
    """
    Main script
    """
    n = declare(int)
    tau = declare(int)
    result_vec = declare(int, size=len(tau_vec))
    c = declare(int)

    with for_(n, 0, n < N_avg, n + 1):
        assign(c, 0)
        with for_each_(tau, tau_vec):
            init_nuclear_spin()
            encode(t_e)
            align(*all_elements)
            play("pi_2", "sample")
            wait(tau, "sample")
            play("pi_2", "sample")
            align(*all_elements)
            play("laser", "sensor")
            decode(t_e)
            SSRO(N_SSRO, result_vec[c])
            assign(c, c + 1)

    with for_(n, 0, n < result_vec.length(), n + 1):
        save(result_vec[n], "result")


def init_nuclear_spin():
```

```
        state = declare(int)
        SSRO(N_SSRO, state)
        with while_(state == ~0):
          with if_(state == -1):
                  # assignig a frequency on resonanace with the 0 -> -1
                         nuclear transition
                  update_frequency("memory", memory_IF_m1)
                  play("pi_x", "memory")
                  update_frequency("memory", memory_IF_p1)
              with else_():
                  play("pi_x", "memory")
              SSRO(N_SSRO, state)


    def SSRO(N, result):
        """Determine the state of the nuclear spin"""
        i = declare(int)
        res_vec = declare(int, size=10)
        counts = declare(int)
        ssro_count = declare(int,value=0)

        # run N repetitions
        with for_(i, 0, i < N, i + 1):
            wait(100, "sensor")
            CnNOTe(0)
            measure(
                "readout",
                "sensor",
                None,
                time_tagging.analog(res_vec, 300, counts),
            )
            assign(ssro_count, ssro_count + counts) # sums up the total photons
                detected during the SSRO
        # compare photon count to threshold and save result in variable "state"
                or continue to the next step
        with if_(ssro_count < SSRO_threshold):
            assign(result, 0)
        with else_():
            assign(ssro_count, 0)
            with for_(i, 0, i < N, i + 1):
                wait(100, "sensor")
                CnNOTe(1)
                measure(
                    "readout",
```

```
                    "sensor",
                    None,
                    time_tagging.analog(res_vec, 300, counts),
                )
                assign(ssro_count, ssro_count + counts) # sums up the total
                       photons detected during the SSRO
            # compare photon count to threshold and save result in variable
                       "state"
            with if_(ssro_count < SSRO_threshold):
                assign(result, 1)
            with else_():
                assign(result, -1)


    def CnNOTe(condition_state):
        """
        CNOT-gate on the electron spin.
        condition_state is in [-1, 0, 1] for a spin 1 nuclear or [-1, 1] for a
        spin half nuclear and gives the nuclear spin
        state for which the electron spin is flipped.
        """
        align(*all_elements)
        update_frequency("sensor", NV_IF + condition_state * hf_splitting)
        play("pi_x"*amp(0.1), "sensor", duration= (pi_length/4)*10)
        update_frequency("sensor", NV_IF)
        align(*all_elements)


    def encode(t):
        """
        Play the encoding sequence with wait time t.
        """
        align(*all_elements)
        reset_frame("memory")
        play("pi_2_x", "memory")
        CnNOTe(0)
        wait(t // 4, "sensor")
        CnNOTe(0)
        play("pi", "sample")
        CnNOTe(1)
        wait(t // 4, "sensor")
        CnNOTe(0)
        play("pi_2_y", "memory")
```

```
130             align(*all_elements)
131
132
133         def decode(t):
134             """
135             Play the decoding sequence with wait time t.
136             """
137             align(*all_elements)
138             play("pi_2_x", "memory")
139             CnNOTe(1)
140             wait(t // 4, "sensor")
141             CnNOTe(0)
142             play("pi", "sample")
143             CnNOTe(1)
144             wait(t // 4, "sensor")
145             CnNOTe(1)
147             play("pi_2_y", "memory")
148             align(*all_elements)
149
150
151         qmm = QuantumMachinesManager()
152         qm = qmm.open_qm(config)
153         job = qm.execute(prog)
```

QUA code for NMR with a nuclear spin memory as detailed in figure 4.

The **encoding (decoding)** sequence is defined in the function `encode()` (`decode()`). The different pulses are executed using `play` statements and the `CnNOTe()` macro. The QUA-function `align()` is used to define the timing of the individual pulses. One of the basic principles of QUA is that every command is executed as early as possible. Hence, when not specified otherwise, pulses played on different quantum elements are played in parallel. To ensure that pulses play in a specific order, we use the built-in `align()` function. This function causes all specified quantum elements to wait until all previous commands are complete, and so it aligns them in time.

Finally, the main script runs the whole sequence for different values of the waiting time of the Ramsey sequence played on the sample spin in a `foreach` loop. Additionally, the experiment is averaged over `N_avg` repetitions by an outer `for` loop. The result of each measurement is saved into the corresponding item of the result vector `result_vec`. Then, the result vector is saved element-wise using the `save()` function and streamed to the user PC.

References

[1] *Z. Wang et al., "Randomisation of Pulse Phases for Unambiguous and Robust Quantum Sensing", Phys. Rev. Lett. 122, 200403 (2019)*

[2] *S. Zaiser et al., "Enhancing quantum sensing sensitivity by a quantum memory", Nature Comm. 7, 12279 (2016)*

[3] *M. Pfender et al., "Nonvolatile nuclear spin memory enables sensor-unlimited nanoscale spectroscopy of small spin clusters", Nature Comm. 8, 834 (2017)*

[4] *T. H. Taminiau et al., "Universal control and error correction in multi-qubit spin registers in diamond", Nature nano. 9, 171–176 (2014)*

[5] *M. Pompili et al., "Realization of a multi-node quantum network of remote solid-state qubits", Science 372, 6539 (2021)*

[6] *P. Neumann et al., "Single-Shot Readout of a Single Nuclear Spin", Science 329, 542-544 (2010)*

[7] *N. Aslam et al., "Nanoscale nuclear magnetic resonance with chemical resolution", Science 357, 67-71 (2017)*

[8] *T. Staudacher et al., "Nuclear Magnetic Resonance Spectroscopy on a (5-Nanometer)^3 Sample Volume", Science 339, 561-563 (2013)*

[9] *A. Laraoui et al., "High-resolution correlation spectroscopy of 13C spins near a nitrogen-vacancy centre in diamond", Nature Comm. 4, 1651 (2013)*

"Dedicated hardware for controlling and operating quantum bits is something we have all been dreaming of. Quantum Machines has answered this call by allowing us and others in the field to scale up with ease and with far greater functionality than was ever possible"

Prof. Amir Yacoby, Harvard University

HARVARD UNIVERSITY

"Replacing 3 devices with one synchronized, orchestrated machine tremendously simplifies lab workflow. Now, our pulse sequences run in a fraction of the time of any other device combo. Plus, we can talk to the FPGA in human-speak to run real-time calculations that were too complicated before! Along with the yoga-level flexibility of QM's engineers, the OPX truly is a trailblazer."

Dr. Amit Finkler, Weizmann Institute of Science

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

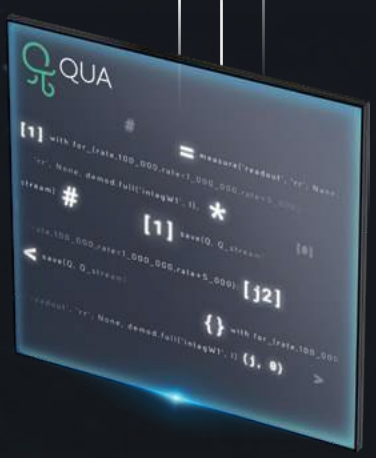
ACHIEVE THE FASTEST TIME TO RESULTS

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

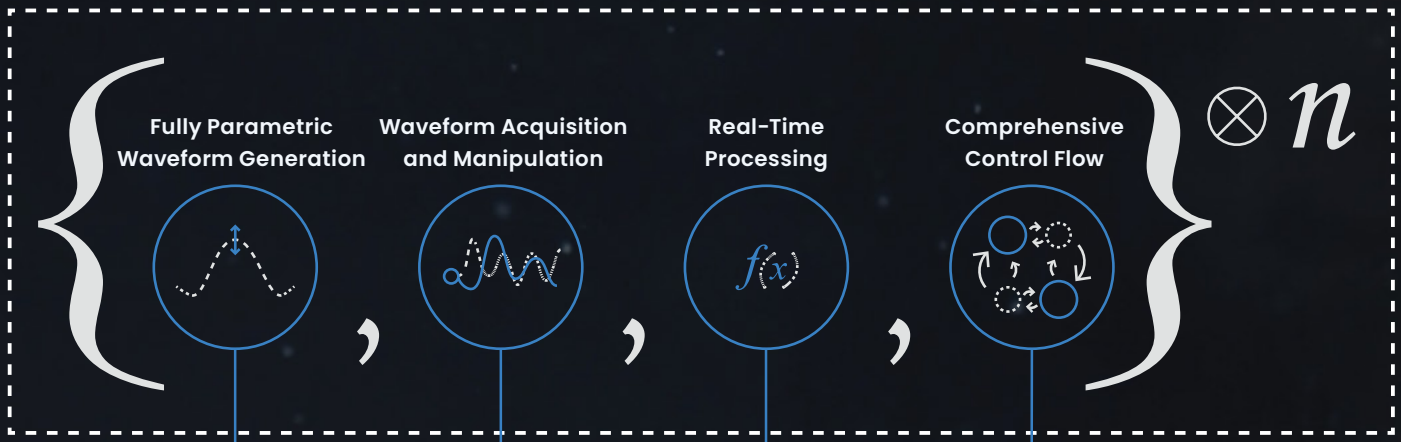
CODE QUANTUM PROGRAMS SEAMLESSLY

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*

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