Simulation, optimization and applications of near-term quantum computers

Thomas Ayral, Atos Quantum Lab

December 3rd, 2019

Bridging the gap

Many theoretical ideas

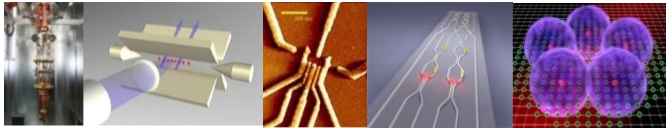
- **Quantum algorithmics**
  Very few known algorithms (+need many good quality gates & many qubits)

- **NISQ: Hybrid quantum classical programs**
  - Variational Quantum Eigensolver and Simulator (VQE, VQS)
  - Error mitigation…

?

No killer app yet!

Many hardware implementations



- Different connectivities
- Different gatesets
- Different gate qualities/durations
- Different readout errors
- Different coherence times
- Different scalability
- Analog vs digital
- Adiabatic vs diabatic

Outline

- **Simulation**

  efficiently simulating the (imperfect) behavior of each platform

- **Optimization**

  developing quantum programs in a hardware-agnostic way… but suitably optimized for target platform

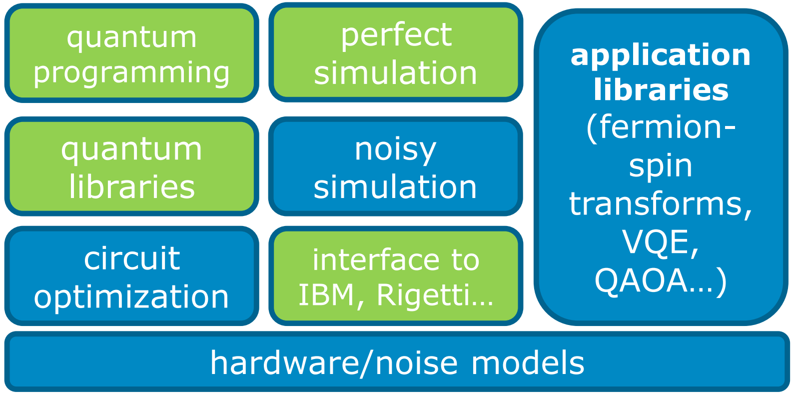
- **Applications**

  Focus: Schwinger model energy with variational algorithms

Atos Quantum Learning Machine

a dedicated quantum programming platform

myQLM: "QLM lite", free!



quantum programming

perfect simulation

quantum libraries

noisy simulation

circuit optimization

interface to IBM, Rigetti…

application libraries (fermion-spin transforms, VQE, QAOA…)

hardware/noise models

Atos quantum: worldwide…

On-going PoC with

BAYER & RWTH AACHEN UNIVERSITY

… and at home: our collaborations

Quantum programming, classical simulation

BPI project Quantex (with Paris Saclay, CNRS-LORIA, CEA-LETI). *Quantum programming, hardware-acceleration for classical simulation of quantum circuits*

ANR SoftQPro (with Paris Saclay, CNRS-LORIA, CEA-LIST). *Numerical simulation of high-level quantum programming languages*.

CIFRE PhD thesis with Supélec/Saclay: *numerical techniques for quantum circuit generation*

Quantum algorithmics

QUANTERA QuantAlgo (with CNRS-IRIF, CWI-QuSoft, Cambridge, Univ. of Latvia, Univ. Libre de Bruxelles). *Machine Learning, exploration of use cases.*

ANR QuData (with CNRS-IRIF, Paris-Sorbonne, CNRS-LABRI). *Assessment of industrial use cases*.

CIFRE PhD thesis with IRIF: *algorithms for Quantum Machine Learning.*

Quantum hardware

EU-Flagship project AQTION (with Univ. Innsbruck (Blatt group), Oxford, ETHZ, Mainz, Fraunhofer, Swansea, Toptica). *Programming frontend, compilation, industrial use cases*

Chaire industrielle NASNIQ (with CEA-DRF Quantronics lab). *Computational architecture, noise models*

Merlion project Siliquon qubits (with CNRS-Néel, Singapore Institute of Quantum Computing). *Noise simulation of Si Qubits*

Analog quantum simulation

EU-Flagship PASQUANS Flagship project (with Institut d'Optique, Univ. Innsbruck (Zoller group), ETHZ, Univ Munich, LKB, Univ Strathclyde, Univ Ulm, Univ Padova, Univ Heidelberg).

WP leader of applications

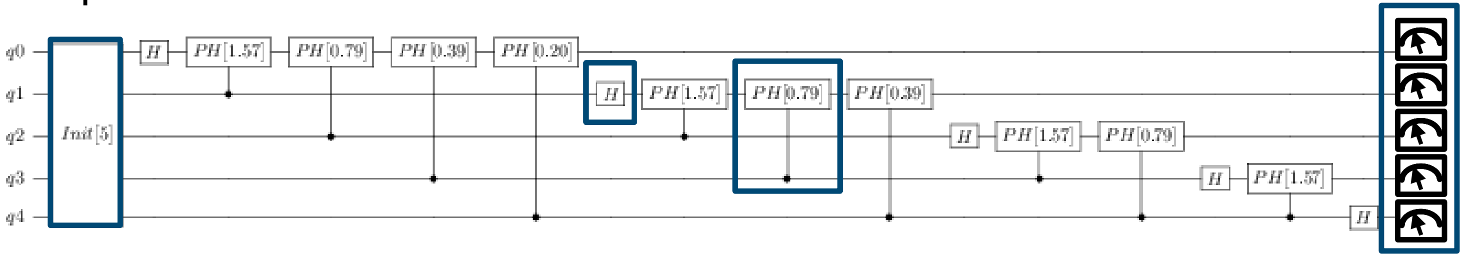
1

Simulation

efficiently simulating the (imperfect) behavior of each platform

The quantum Fourier transform: theory…

- Textbook quantum circuit:



state preparation ➡ computation: apply quantum gates ➡ final step: measurement

- In theory: **exponential speedup** (over fast Fourier transform)

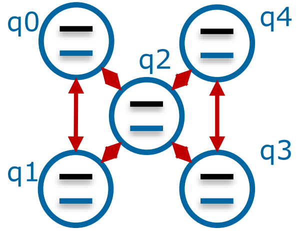
- Key ingredient of **Shor's factoring algorithm**

Promise of huge speedups… if "ideal" quantum computer

… and practice

Hardware constraints

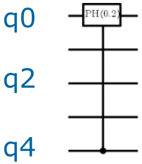
Limited connectivity



Limited expressivity

“only CNOT gates”

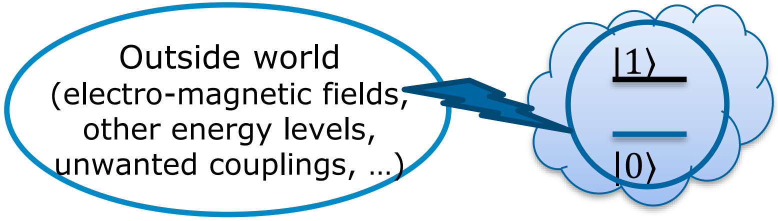
how to run such a gate:



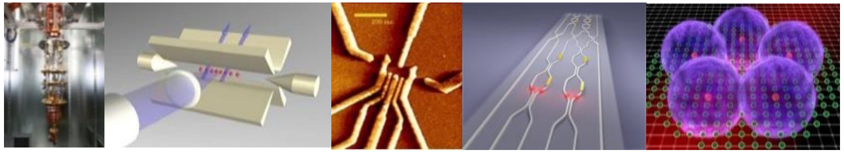
insert swaps, rewrite as combination of native gates…

Larger gate counts

Quantum decoherence



Outside world
(electro-magnetic fields,
other energy levels,
unwanted couplings, …)



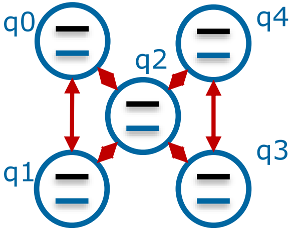
Shorter time window

This information is essential to assess quality of final result

… and practice

Hardware constraints

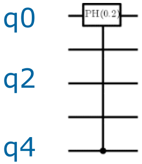
Limited connectivity



Limited expressivity

“only CNOT gates”

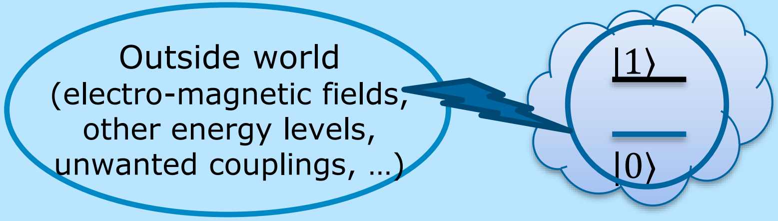
how to run such a gate:



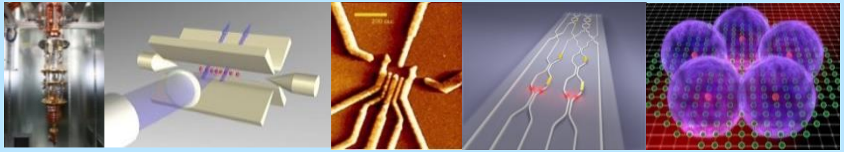
insert swaps, rewrite as combination of native gates…

Larger gate counts

Quantum decoherence



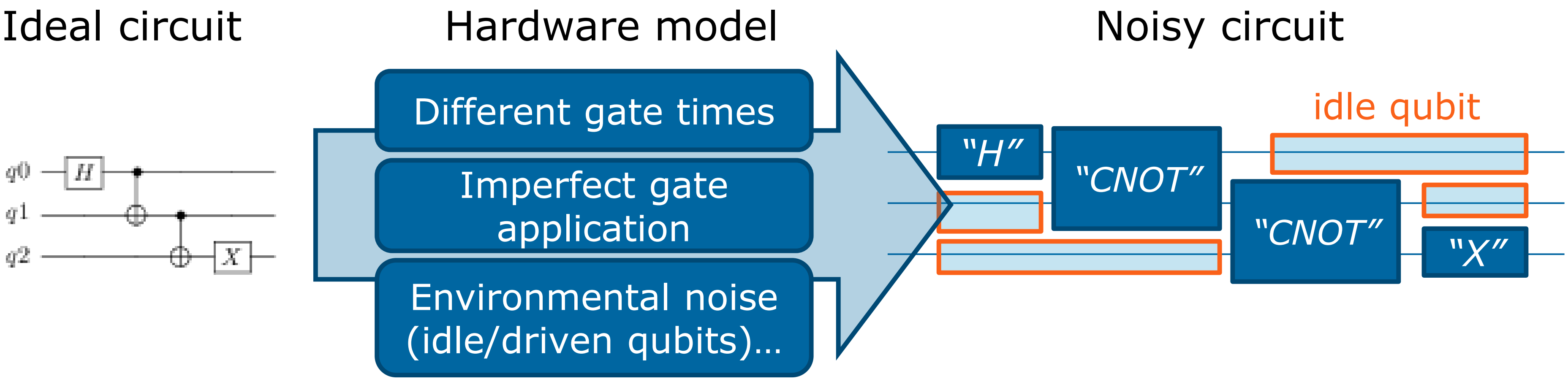
Outside world
(electro-magnetic fields,
other energy levels,
unwanted couplings, …)



Shorter time window

This information is essential to assess quality of final result

Discrete modeling of noise



Ideal circuit

Hardware model

Noisy circuit

Different gate times

Imperfect gate application

Environmental noise (idle/driven qubits)…

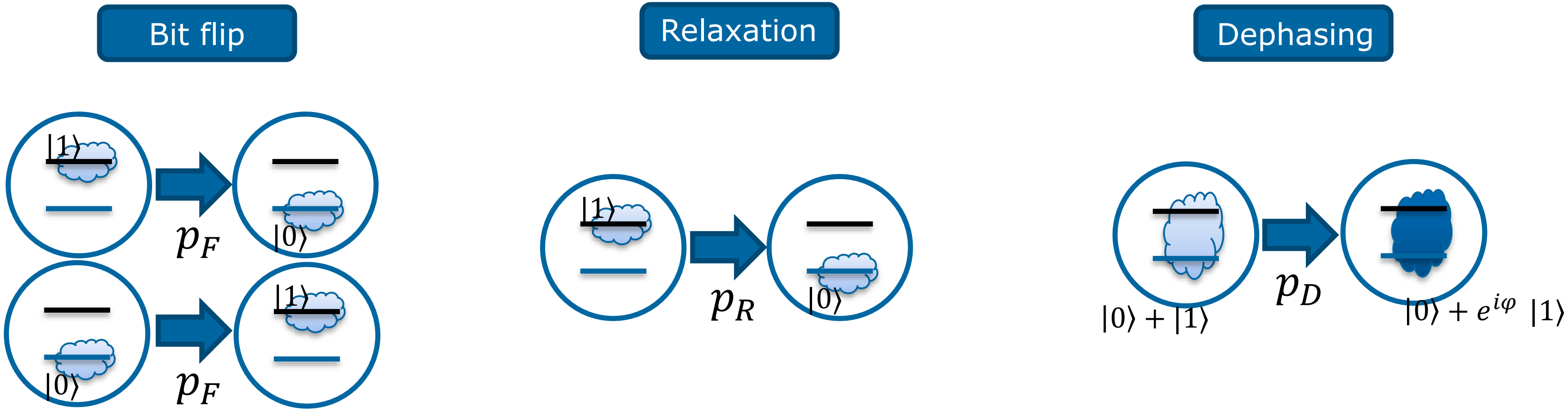
- Modular, hence **scalable** approach
- Can capture **large variety of noises** (incl. spatially correlated, crosstalk, leakage…)

- **Caveat**: does not capture all memory effects

What do the boxes look like?

Textbook noise models…

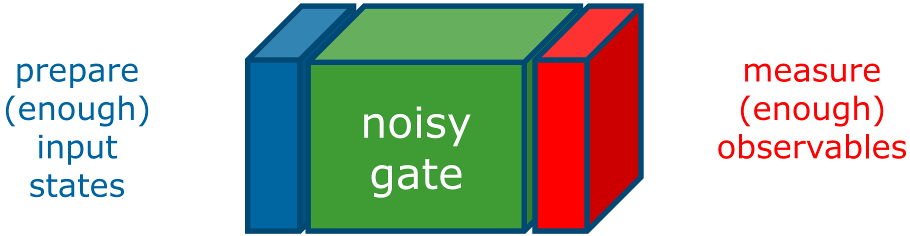
see e.g. Nielsen & Chuang



Challenge: what is p_F, p_R, p_D … for a given hardware?
Are there other important types of noise?

… and more 'ab initio' approaches: tomography

- Characterize noise processes:
  "process tomography"



- Two complementary strategies:

Experiment (phenomenology)

- **Quantum process tomography**: assume known state preparation & measure (aka SPAM) (!)
- **Gateset tomography**: consistent handling of SPAM error
  Merkel et al / Blume-Kohout et al 2013

Advantage: phenomenological ("black box") approach

Numerics (microscopic model)

- Write **Hamiltonian model** for given operation
- Solve **Schrödinger/master equation** for all inputs

Advantage: inputs are usually experimentally accessible (microscopic) parameters.

Ideal vs Noisy quantum computation

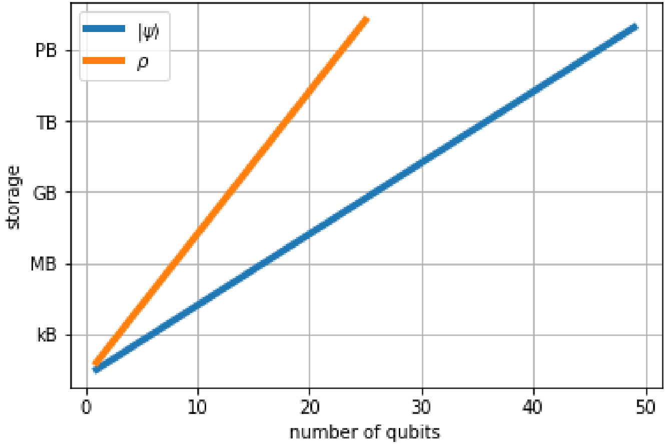
Ideal circuit simulation

- Quantum state:
  - pure state $|\psi\rangle$
  - generically: $\mathbf{2^n}$ **complex numbers**
- Gates:
  - **unitary operators U**

  acting on subsets of qubits
- Simulation:

$$|\psi_f\rangle = U_M \dots U_1 |\psi_i\rangle$$

Memory footprint



Noisy simulation

- Quantum state:
  - mixed state $\rho = \sum_i p_i |\psi_i\rangle\langle\psi_i|$
  - generically: $\mathbf{2^n \times 2^n}$ **complex numbers**
- Gates:
  - **linear, "CPTP" maps $\mathcal{E}(\rho)$** (completely positive, trace preserving)
- Simulation:

$$\rho_f = \mathcal{E}_M \circ \dots \circ \mathcal{E}_1(\rho_i)$$

Ideal circuit simulation

Unitary evolution $|\psi_f\rangle = U_M \ldots U_1|\psi_i\rangle$

"Brute-force" simulation

- Store 2^n amplitude vector:

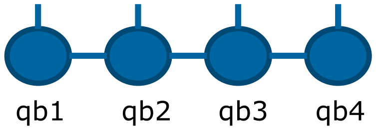
$$|\psi\rangle = \sum_{b_1..b_N} a_{b_1..b_N}|b_1..b_N\rangle$$

$a_{b_1b_2b_3b_4} =$ 

- Up to 40-41 qubits

Matrix product states (and tensor networks)

- MPS representation (4 qbits):



see e.g
Schollwöck '10,
Orus '14

- "Bond dimension" and entanglement relation
- Can simulate >40 qubits

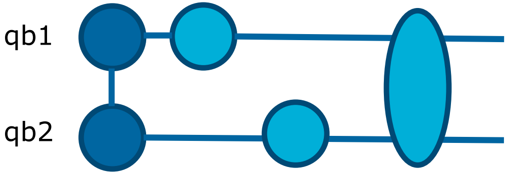
Stabilizer simulation

- Only "Clifford" gates: can represent state with $O(n^2)$ cost
- Can simulate >1000 qubits
- Extensions for "Clifford+T"…

Gottesman & Knill

Tensor-network contraction

- Space-time network (here: 2 qbits + 3 gates)



- Challenge: find fast contraction order

Noisy simulation

Density-matrix evolution:

- Use e.g 'Kraus' representation

$$\rho_f = \sum_{k_M} \left(E_{k_M}^{(M)} \dots \left(\sum_{k_1} E_{k_1}^{(1)} \boldsymbol{\rho} E_{k_1}^{(1)\dagger} \right) \dots E_{k_M}^{(M)\dagger} \right)$$

- ~OK for <20 qubits

Stochastic sampling:

- $\rho_f \approx \frac{1}{N} \sum_{i=1..N} |\psi_i\rangle\langle\psi_i|$
- Can use ideal circuit simulator to compute each "trajectory" $|\psi_i\rangle$
- Can simulate many more qubits

Near-Clifford simulation

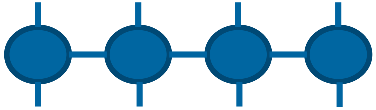
- Decompose $\mathcal{E}^{(j)} = \sum_i q_i^{(j)} S_i$

  with S_i stabilizer channel
- Sample resulting sum over stabilizer channels

Bennink et al

Matrix product operators (and tensor networks)

- MPO representation (4 qbits):



2

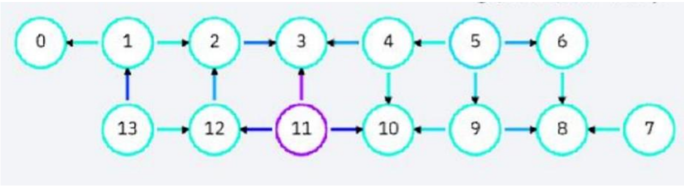
Optimization

developing quantum programs in a hardware-agnostic way…
but suitably optimized for a target platform

Example 1: Compilation of a Quantum Fourier Transform

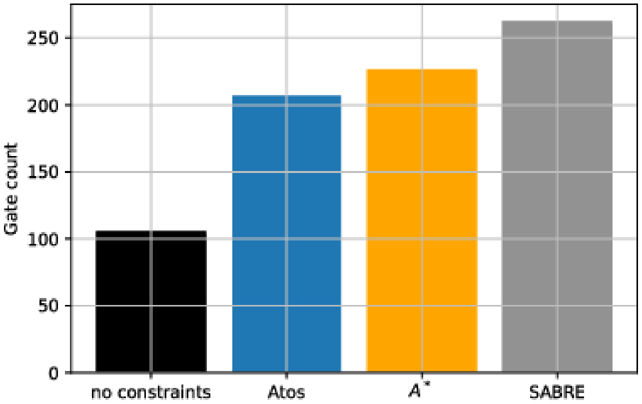
- Fourier transform on IBM's 14-qbit Melbourne chip
  Need to insert SWAPs + use only CNOTs

Connectivity map:

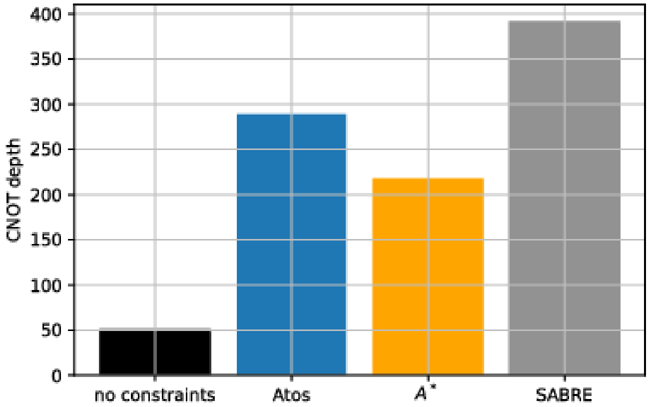


- Two different optimization metrics:

Gate count



CNOT depth

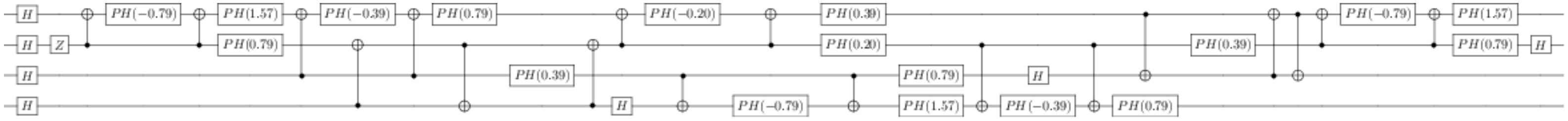


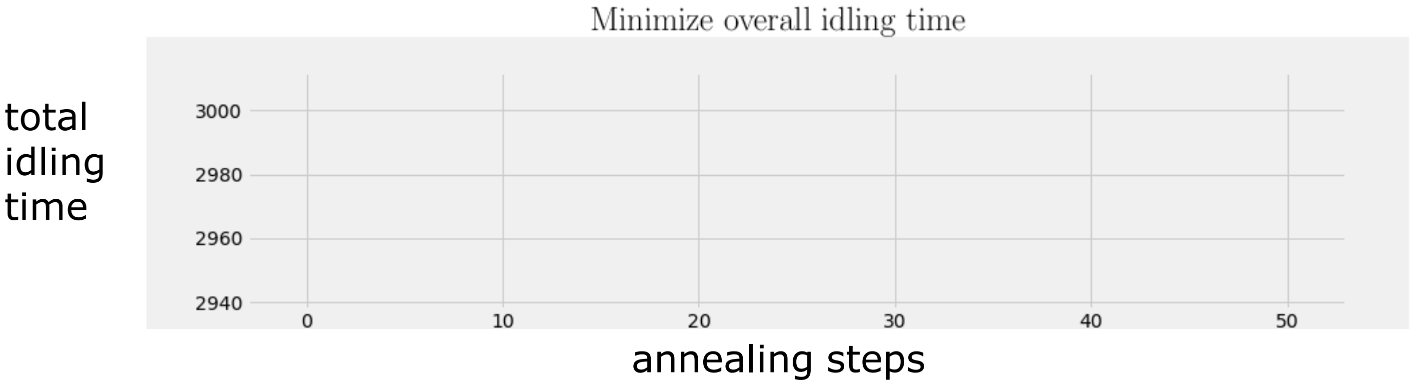
A^*: 1712.04722, SABRE: 1809.02573

1 SWAP=3 CNOTs, 1 C-PH=2 CNOTs

Example 2: Minimization of the total idling time

- Start compiled Quantum Fourier (here 4 qubits)
- Use **commutation patterns** to reduce **total idling time**
- Minimization via (classical) **simulated annealing**



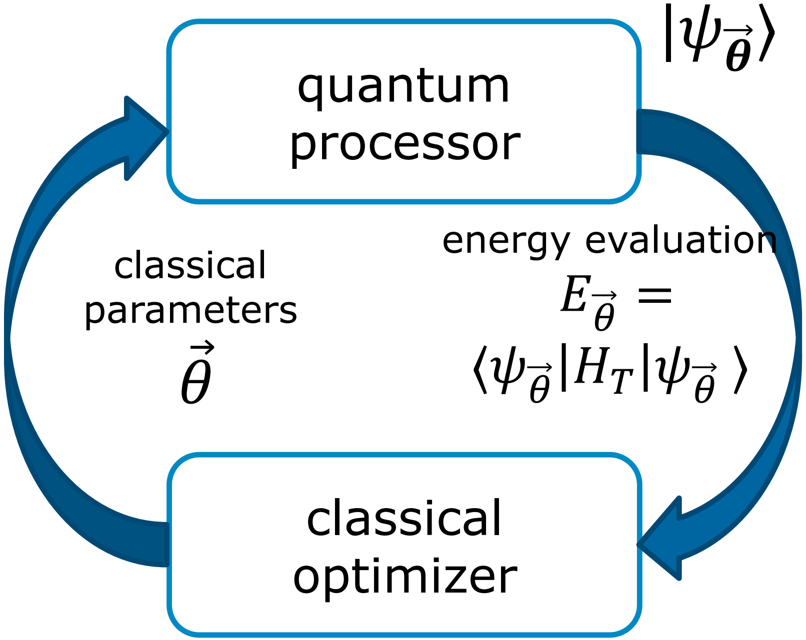


3

Applications with near-term processors

Variational quantum algorithms

- Overcome
  - short coherence times
- Split work between
  - quantum processor
  - classical optimizer
- **Goal:**
  - find lowest eigenenergy of Hamiltonian H_T



Applications of variational quantum algorithms

- Quantum chemistry (“VQE”)
  - Many small molecules (H2, LiH, …)
  - Well-known variational states: unitary coupled cluster (UCC), etc.

- Combinatorial optimization (“QAOA”)
  - Special ansatz inspired from quantum annealing

- **Focus:** Variational quantum algorithms for **quantum field theory**?
  - Lattice QCD: a gauge theory plagued by Monte-Carlo sign problem in interesting regimes (hot quark-gluon plasma, neutron stars…)
  - Here, take Schwinger model (1+1-dim QED) as proxy for lattice QCD physics

Challenge I:
translate problem to quantum computer language

- (Kogut-Susskind) **fermions**… in **spin**/qbit-based quantum computer?
  - Jordan-Wigner transformation

- infinite gauge degrees of freedom… in finite-dim quantum computer?
  - Use Gauss law to eliminate gauge d.o.f -> traded for exotic **long-range spin-spin interactions**

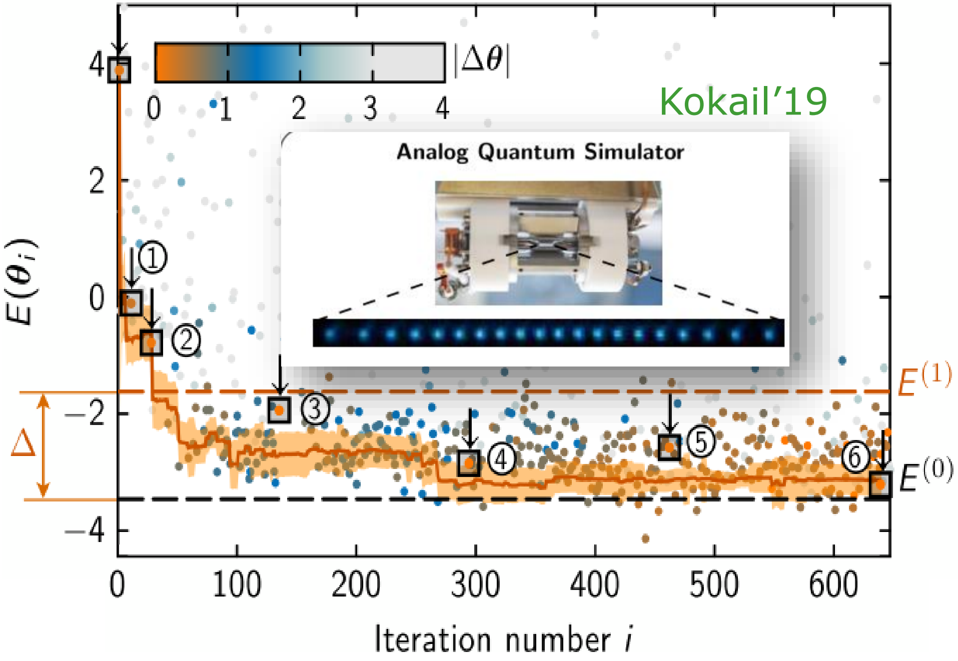
- **Final Hamiltonian:**

$$H_T = \sum_j \sigma_j^+ \sigma_{j+1}^- + h.c + \frac{m}{2} \sum_j (-)^j \sigma_j^z + \frac{g}{4} \sum_j \left(\sum_{l \leq j} \sigma_l^z + (-)^l \right)^2$$

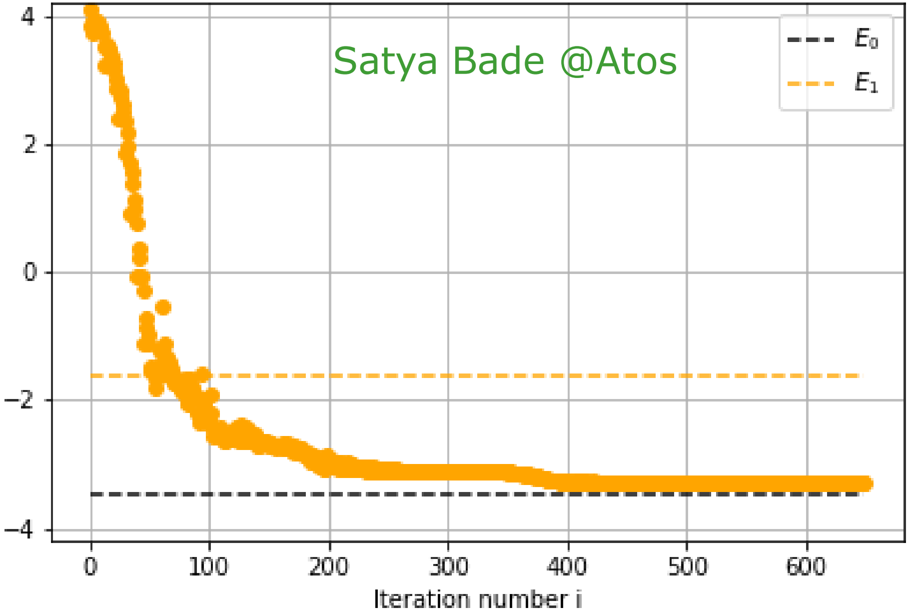
with mass m, coupling g

Challenge II: Experimental realization and simulation

- Experimental result (Nions=8):



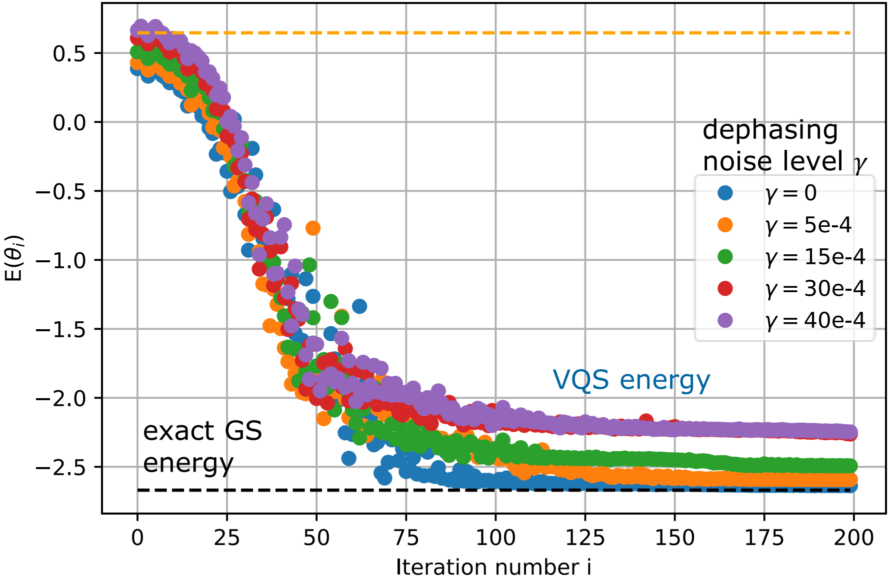
- Our simulation (Nions=8)



See also: digital computation with 4 ions (Martinez et al, Nature '16), 5 SC qubits (Klco et al PRA '18), …

Going beyond: impact of noise

- Noisy simulations with varying noise levels
  - Dephasing noise (Lindblad equation)

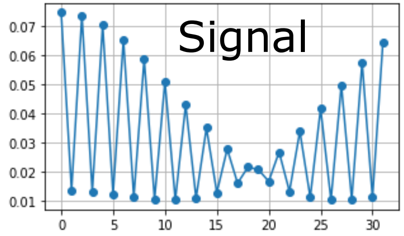


4

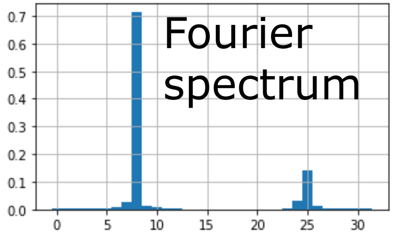
Demonstration

A Noisy Quantum Fourier Transform on the Atos Quantum Learning Machine

- Key ingredient of quantum factoring algorithms (“Shor algorithm”)
- From time to frequency:



$P = 32 = 2^5$ time points



- Best known classical algorithm: Fast Fourier Transform

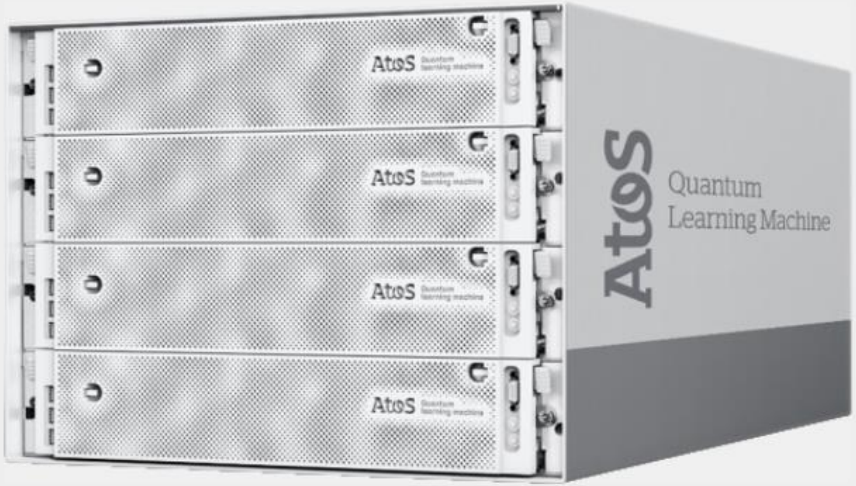
  Number of operations: $\sim \boldsymbol{P \log P}$ for a signal of size P

- Quantum Fourier Transform (QFT):

  $\boldsymbol{\log P \log P}$ operations: **exponential speedup!**

A Noisy Quantum Fourier Transform on the Atos Quantum Learning Machine

Demonstration: simulate a (noisy) processor on the QLM



Part I: Writing an universal quantum program

Part II: IBM QX4 - superconducting qubits

Part III: AQTion-related results - trapped ions

Part I: Writing an universal quantum program

Example: Quantum Fourier transform, $\mathrm{QFT}(|x\rangle) = \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} \left(e^{\frac{2i\pi}{2^n}}\right)^{xk} |k\rangle$

Key ingredient to e.g **Shor's factoring algorithm.**

I.1 Writing the quantum program

The QLM provides usual quantum gates + libraries of quantum routines

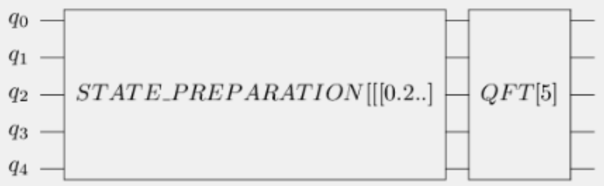
Entrée [1]:

```
from qat.lang.AQASM import Program
from qat.lang.AQASM.qftarith import QFT
from qat.linalg.oracles import StatePreparation
from demo_init import prepare_ft_signal, format_qft_output

nqbits = 5
prog = Program()
reg = prog.qalloc(nqbits)
state_prep = StatePreparation(prepare_ft_signal(nqbits))
prog.apply(state_prep, reg)
prog.apply(QFT(nqbits), reg)
qft_circ_boxed = prog.to_circ(inline=False)

%qatdisplay qft_circ_boxed
```

Out[1]:



I.2 Ideal simulation

```
Entrée [2]: import numpy as np
            import matplotlib.pyplot as plt
            %pylab inline
            from qat.qpus import LinAlg

            results = LinAlg().submit(qft_circ_boxed.to_job())

            resprobs = np.zeros(shape=(2**nqbits))
            for result in results.raw_data: resprobs[result.state.int]=result.probability
            plt.figure(figsize=(10,4))
            plt.bar(np.array(range(2**nqbits)), np.abs(format_qft_output(resprobs)), width=4/5, label="ideal processor")
            plt.legend(); plt.grid(); plt.xlabel(r"$\omega$");
```

```
Populating the interactive namespace from numpy and matplotlib
```

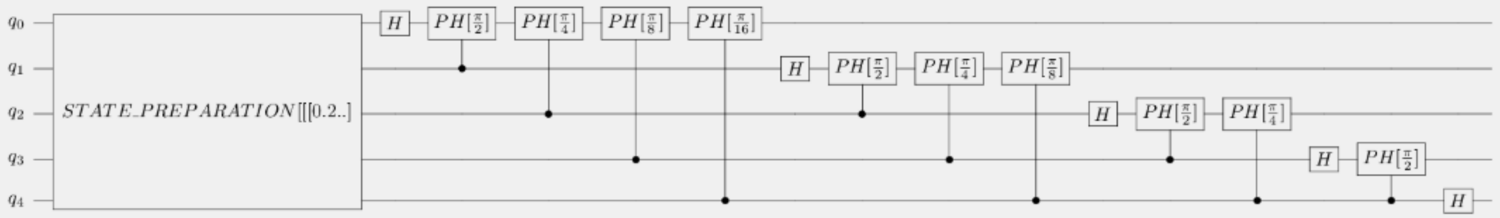
I.3 Universal circuit

Entrée [3]:

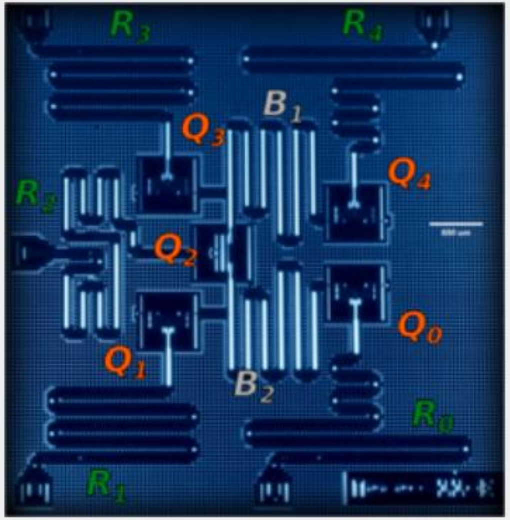
```
qft_circ = prog.to_circ(inline=True)
print("Number of gates:", len(qft_circ))
%qatdisplay qft_circ
```

```
Number of gates: 16
```

Out[3]:

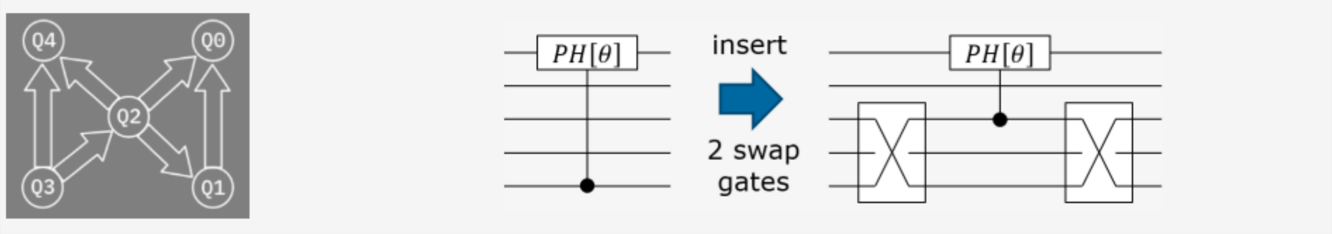


Part II: IBM QX4 - superconducting qubits



II.1 Connectivity

IBM QX4 has a limited qubit connectivity:



insert

2 swap gates

Entrée [4]:

```
%cat graph_ibmqx4.json
```

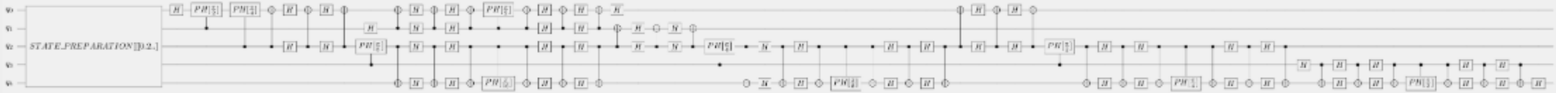
```
{"edges": {"1": [0], "2": [0, 1, 4], "3": [2, 4]}}
```

Entrée [5]:

```
import json
from qat.core.simutil import optimize_circuit
from qat.core import Topology, HardwareSpecs, TopologyType
graph_dict = json.load(open("graph_ibmqx4.json", 'r'))["edges"]
topology = Topology(type=TopologyType.CUSTOM, is_directed=True,
                    graph={int(vertex): edges for vertex, edges in graph_dict.items()})
hw_specs = HardwareSpecs(nbqbits=nqbits, topology=topology)
from qat.plugins import Nnizer
nnizer = Nnizer(ignore_large_gates=True)
qft_conn_circ = optimize_circuit(qft_circ, nnizer, specs=hw_specs)
print("Number of gates:", len(qft_conn_circ))
%qatdisplay qft_conn_circ
```

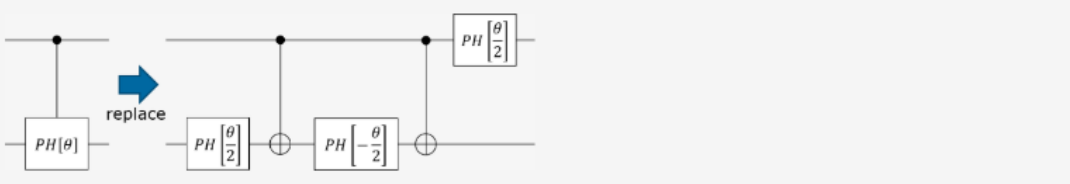
```
Number of gates: 107
```

Out[5]:



II.2 Compilation (continued)

IBM QX4 only accepts CNOT gates for two-qubit operations:

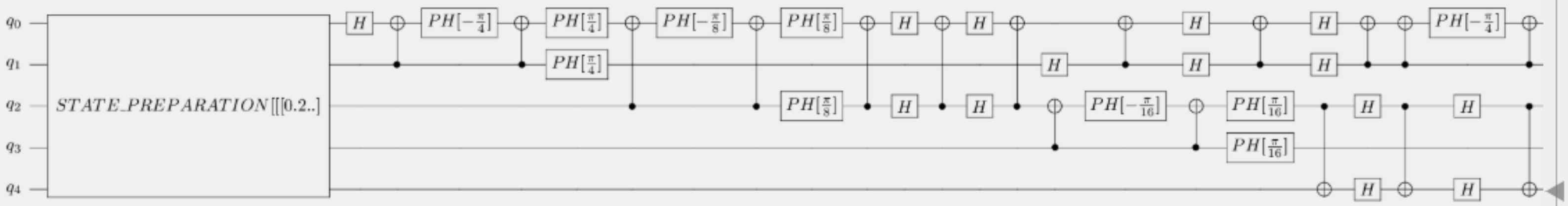


Entrée [6]:

```
from qat.pbo import VAR
from qat.plugins import GateRewriter
compiler = GateRewriter()
theta = VAR()
compiler.add_gate("C-PH",
                  variables=[theta],
                  pattern=[("CNOT", [0, 1]),  ("PH", [1], -theta / 2),  ("CNOT", [0, 1]),
                           ("PH", [1], theta / 2), ("PH", [0], theta / 2)])
compiler.add_abstract_gate("STATE_PREPARATION", lambda x: StatePreparation(x, False))
qft_conn_gates_circ = optimize_circuit(qft_conn_circ, compiler)
print("Number of gates:", len(qft_conn_gates_circ))
%qatdisplay qft_conn_gates_circ
```

```
Number of gates: 147
```

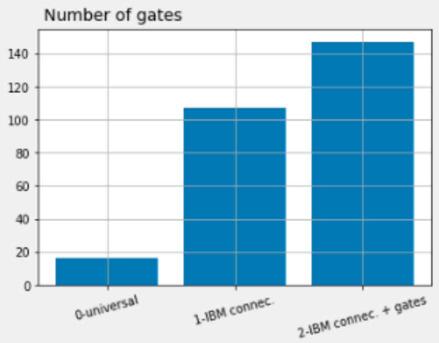
Out[6]:



II.3 Gate counts before and after compilation

Entrée [7]:

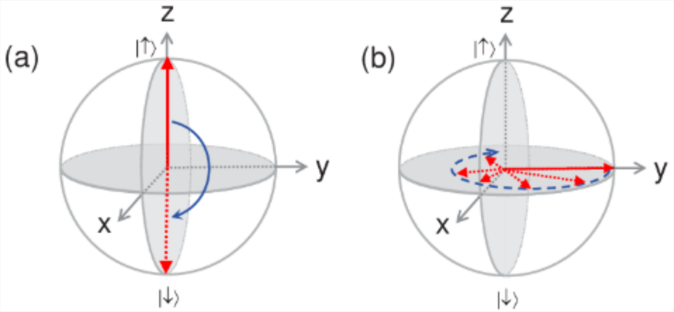
```
import matplotlib.pyplot as plt
%matplotlib inline
plt.bar(["0-universal", "1-IBM connec.", "2-IBM connec. + gates"],
        [len(c.ops) for c in [qft_circ, qft_conn_circ, qft_conn_gates_circ]])
plt.xticks(rotation=15); plt.text(-0.5,160,"Number of gates", size=14); plt.grid();
```



II.4 Noisy simulation

Live simulation of noisy simulation on Atos QLM, with **simplified hardware model**:

- "Idle" qubits suffer from decoherence: **amplitude damping** (A.D) and **pure dephasing** (P.D).
- Relaxation and dephasing times from constructor: T_1 and T_2



Entrée [8]:

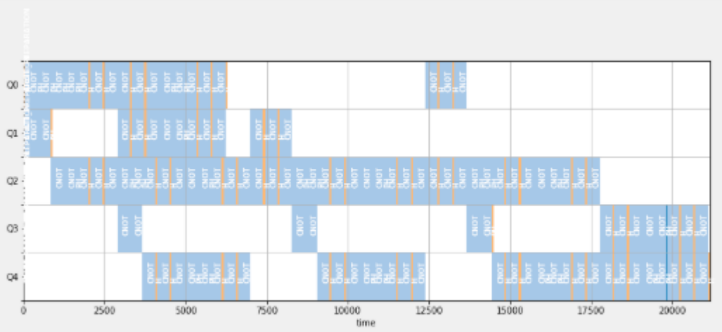
```
from qat.hardware import DefaultGatesSpecification, HardwareModel
from qat.core.circuit_builder.matrix_util import get_predef_generator
from qat.quops.quantum_channels import ParametricPureDephasing, ParametricAmplitudeDamping
gate_durations = {"H":60, "X":120, "Y":120, "S":1,"T":1, "D-T":1, "Z":1,
                  "RZ": lambda angle: 1, "PH": lambda angle: 1, "C-PH": lambda angle: 1,
                  "CNOT":386, "SWAP" :100, "STATE_PREPARATION": 10}
pred_generator = get_predef_generator()
pred_generator["STATE_PREPARATION"] = prepare_ft_signal(nqbits)
ibm_gates_spec = DefaultGatesSpecification(gate_durations, predef_generator=pred_generator)
T1, T2 = 44000, 38900 #nanosecs
amp_damping = ParametricAmplitudeDamping(T_1=T1)
pure_dephasing = ParametricPureDephasing(T_phi=1/(1/T2 - 1/(2*T1)))
ibm_hardware = HardwareModel(ibm_gates_spec, idle_noise=[amp_damping, pure_dephasing])
```

II.5 Temporal representation of the quantum algorithm

Entrée [9]:

```
print("Number of gates:", len(qft_conn_gates_circ))
%qatdisplay qft_conn_gates_circ ibm_hardware
```

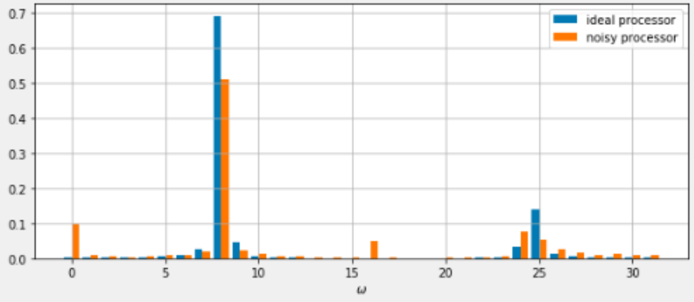
```
Number of gates: 147
```



II.6 Result: Noisy Fourier spectrum

Entrée [10]:

```
import numpy as np
from qat.qpus import LinAlg, NoisyQProc
from qat.plugins import QuameleonPlugin
plt.figure(figsize=(10,4))
ideal_qpu = LinAlg()
noisy_qpu = nnizer | compiler | QuameleonPlugin(specs=hw_specs) | NoisyQProc(hardware_model=ibm_hardware)
for nqpu, (qpu, label) in enumerate([(ideal_qpu, "ideal processor"), (noisy_qpu, "noisy processor")]):
    results = qpu.submit(qft_circ.to_job())
    allprobs = np.zeros(shape=(2**nqbits))
    for result in results.raw_data: allprobs[result.state.int]=result.probability
    plt.bar(np.array(range(2**nqbits))+nqpu*2/5-1/5, np.abs(format_qft_output(allprobs)), width=2/5, label=label)
plt.legend(); plt.grid(); plt.xlabel(r"$\omega$");
```



II.7 Fidelity optimization

Entrée [11]:

```
from qat.plugins import PatternManager
from qat.noisy.noisy_circuit import total_idling_time

metric = lambda circ: -total_idling_time(circ, ibm_hardware)
manager = PatternManager(global_metric=metric)
x, y = VAR(), VAR()
group1 = manager.new_group()
group1.add_pattern([('PH', [1], x), ('CNOT', [0, 1]), ('PH', [1], y), ('CNOT', [0, 1])])
group1.add_pattern([('CNOT', [0, 1]), ('PH', [1], y), ('CNOT', [0, 1]), ('PH', [1], x)])
group2 = manager.new_group()
group2.add_pattern([('PH', [1], x), ('CNOT', [0, 1]), ('H', [0]), ('H', [1]), ('CNOT', [0, 1]), ('H', [0]), ('H', [1]),  ('CNOT', [0, 1])])
group2.add_pattern([('CNOT', [0, 1]), ('H', [0]), ('H', [1]), ('CNOT', [0, 1]), ('H', [0]), ('H', [1]), ('CNOT', [0, 1]), ('PH', [0], x)])
group3 = manager.new_group()
group3.add_pattern([('PH', [0], x), ('CNOT', [0, 1]), ('H', [0]), ('H', [1]), ('CNOT', [0, 1]), ('H', [0]), ('H', [1]), ('CNOT', [0, 1])])
group3.add_pattern([('CNOT', [0, 1]), ('H', [0]), ('H', [1]), ('CNOT', [0, 1]), ('H', [0]), ('H', [1]), ('CNOT', [0, 1]),('PH', [1], x)])
group4 = manager.new_group()
group4.add_pattern([('PH', [1], x), ("H", [1]), ("CNOT", [0, 1]), ("H", [1])])
group4.add_pattern([("H", [1]), ("CNOT", [0, 1]), ("H", [1]), ('PH', [1], x)])
group5 = manager.new_group()
group5.pattern_to_remove([('PH', [0], x), ('PH', [0], y)])
group5.add_pattern([('PH', [0], x + y)])
group6 = manager.new_group()
group6.pattern_to_remove([('CNOT', [0, 1]), ('CNOT', [0, 1])])
group6.pattern_to_remove([('H', [0]), ('H', [0])])
group6.add_pattern([])
manager.add_abstract_gate("STATE_PREPARATION", lambda x: StatePreparation(x, False))
qft_fid_optim_circ = manager.replace_pattern(
    qft_conn_gates_circ, method="annealing", max_iterations=250)
print("Number of gates:", len(qft_fid_optim_circ))
%qatdisplay qft_fid_optim_circ
```

```
Number of gates: 79
```

Out[11]:

II.8 Results

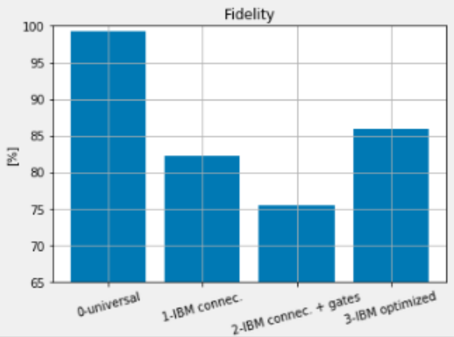
Entrée [12]:

```
from qat.noisy import compute_fidelity
print("Initial idle time:", total_idling_time(qft_conn_gates_circ, ibm_hardware))
print("Final idle time:", total_idling_time(qft_fid_optim_circ, ibm_hardware))

fid_1 = compute_fidelity(qft_circ, ibm_hardware, use_linalg=True)
fid_2 = compute_fidelity(qft_conn_circ, ibm_hardware, use_linalg=False)
fid_3 = compute_fidelity(qft_conn_gates_circ, ibm_hardware, use_linalg=False)
fid_4 = compute_fidelity(qft_fid_optim_circ, ibm_hardware, use_linalg=False)

plt.bar(["0-universal", "1-IBM connec.", "2-IBM connec. + gates", "3-IBM optimized"],
        [fid_1[0]*100, fid_2[0]*100, fid_3[0]*100, fid_4[0]*100])
plt.title("Fidelity"); plt.xticks(rotation=15); plt.ylim(65,100); plt.grid();
plt.ylabel("[%]");
```

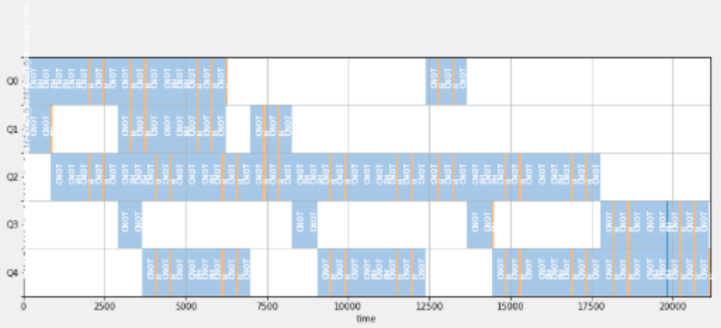
```
Initial idle time: 56722
Final idle time: 28106
```

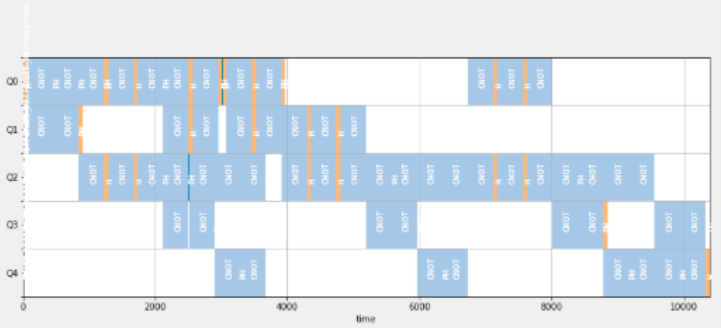


Entrée [13]:

```
print("Number of gates:", len(qft_conn_gates_circ))
%qatdisplay qft_conn_gates_circ ibm_hardware
print("Number of gates:", len(qft_fid_optim_circ))
%qatdisplay qft_fid_optim_circ ibm_hardware
```

```
Number of gates: 147
Number of gates: 79
```

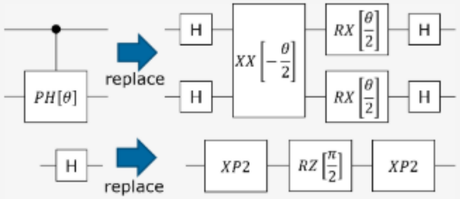




Part III: AQTion-related results - trapped ions

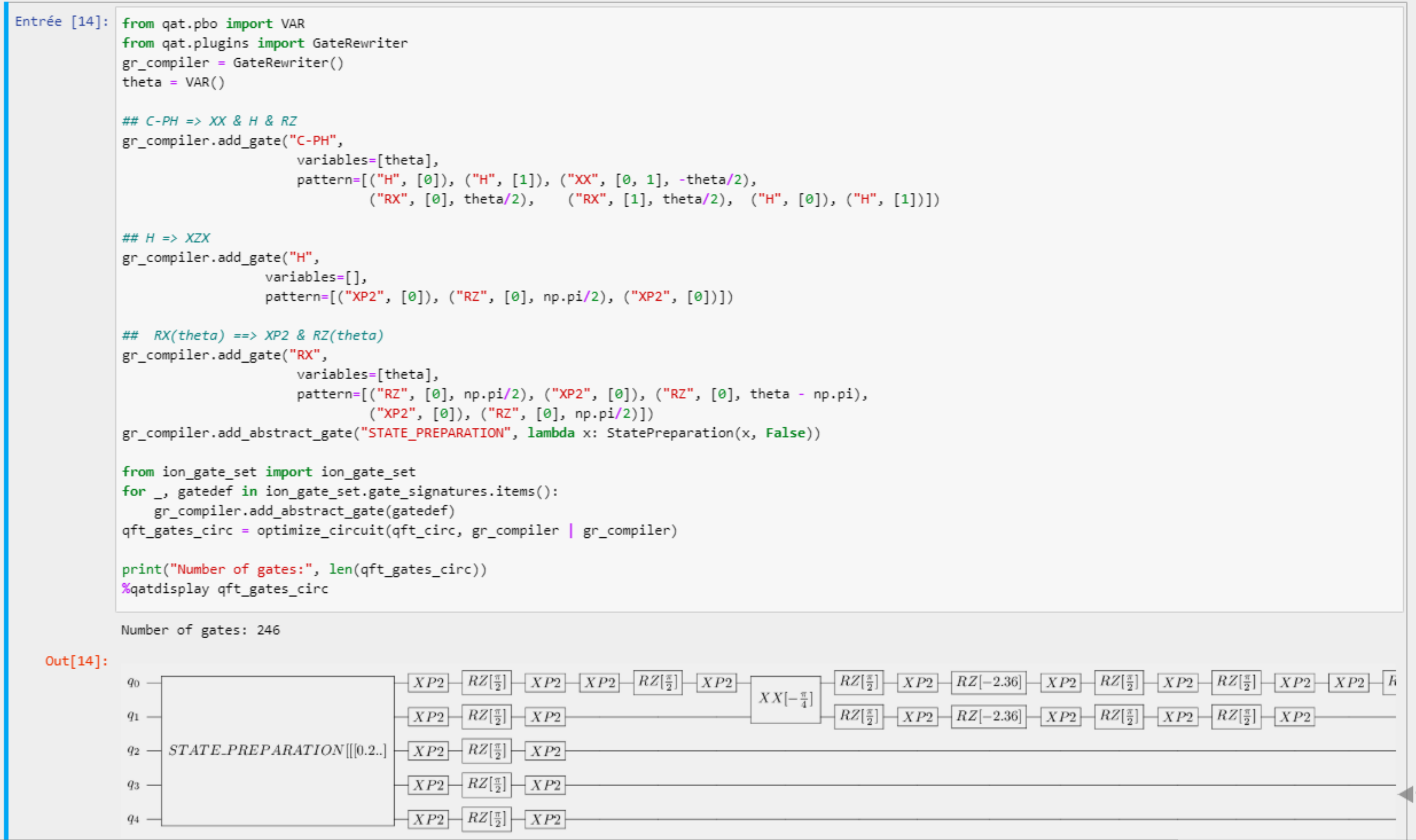
III.1 Compiling for ions

We use the same plugin as for IBM, but with different patterns:



Use abstract gates to create custom parametrized gates, like

$$XX[\phi] = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 0 & 0 & -ie^{i\phi} \\ 0 & 1 & -i & 0 \\ 0 & -i & 1 & 0 \\ -ie^{-i\phi} & 0 & 0 & 1 \end{bmatrix}$$



Entrée [14]:

```python
from qat.pbo import VAR
from qat.plugins import GateRewriter
gr_compiler = GateRewriter()
theta = VAR()

## C-PH => XX & H & RZ
gr_compiler.add_gate("C-PH",
                     variables=[theta],
                     pattern=[("H", [0]), ("H", [1]), ("XX", [0, 1], -theta/2),
                              ("RX", [0], theta/2),    ("RX", [1], theta/2),  ("H", [0]), ("H", [1])])

## H => XZX
gr_compiler.add_gate("H",
                 variables=[],
                 pattern=[("XP2", [0]), ("RZ", [0], np.pi/2), ("XP2", [0])])

##  RX(theta) ==> XP2 & RZ(theta)
gr_compiler.add_gate("RX",
                     variables=[theta],
                     pattern=[("RZ", [0], np.pi/2), ("XP2", [0]), ("RZ", [0], theta - np.pi),
                              ("XP2", [0]), ("RZ", [0], np.pi/2)])
gr_compiler.add_abstract_gate("STATE_PREPARATION", lambda x: StatePreparation(x, False))

from ion_gate_set import ion_gate_set
for _, gatedef in ion_gate_set.gate_signatures.items():
    gr_compiler.add_abstract_gate(gatedef)
qft_gates_circ = optimize_circuit(qft_circ, gr_compiler | gr_compiler)

print("Number of gates:", len(qft_gates_circ))
%qatdisplay qft_gates_circ
```

```
Number of gates: 246
```

Out[14]:

III.2 Noisy simulation with tomography data

Here we load a noise model computed by performing tomography of some gates of the trapped-ion processor of the University of Innsbruck (UIBK):

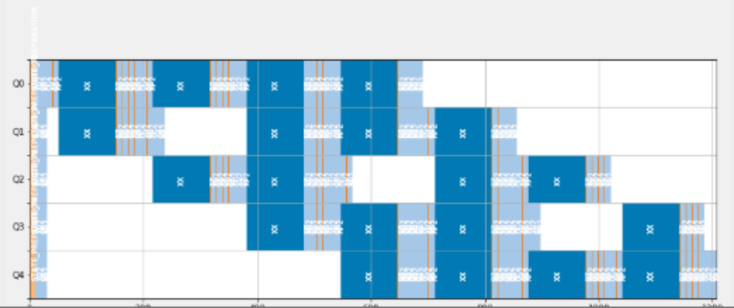
Entrée [15]:

```
from qat.errmitigation.UIBK import get_hdw_gate_spec
from qat.quops import QuantumChannelKraus

#loading noise model computed from linear gateset tomography
ion_gspec = get_hdw_gate_spec(channel_representation="Kraus")

# adding perfect XX (no tomography data) and RZ gate (always perfect because done 'in software')
ion_gspec.quantum_channels["XX"] = lambda angle: QuantumChannelKraus([gen_XX(angle, 2)])
ion_gspec.quantum_channels["RZ"] = lambda angle: QuantumChannelKraus([gen_Z(angle)])
ion_gspec.quantum_channels["STATE_PREPARATION"] \
    = QuantumChannelKraus([prepare_ft_signal(nqbits)], name="state_preparation")
ion_gspec.gate_times = {"RZ":lambda angle: 1,
                        "XX": lambda angle: 100,
                        "STATE_PREPARATION": 10, "XP2": 10}
ion_hardware2 = HardwareModel(gates_specification=ion_gspec)
print("Number of gates:", len(qft_gates_circ))
%qatdisplay qft_gates_circ ion_hardware2
```

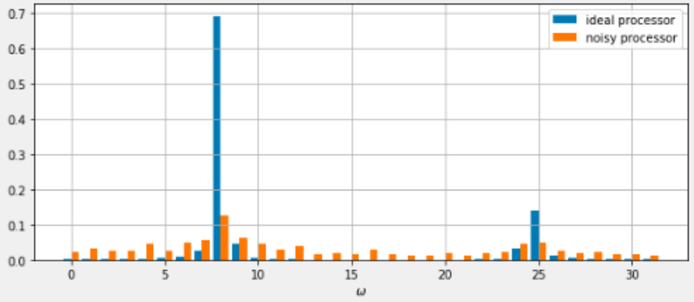
```
Number of gates: 246
```



Entrée [16]:

```
from ion_gate_set import gen_X, gen_Z, gen_XX
compiler = gr_compiler | gr_compiler #need 2 passes to replace all patterns
ideal_qpu = LinAlg()
noisy_qpu = compiler | NoisyQProc(hardware_model=ion_hardware2)

plt.figure(figsize=(10,4))
for nqpu, (qpu, label) in enumerate([(ideal_qpu, "ideal processor"), (noisy_qpu, "noisy processor")]):
    results = qpu.submit(qft_circ.to_job())
    allprobs = np.zeros(shape=(2**nqbits))
    for result in results.raw_data: allprobs[result.state.int]=result.probability
    plt.bar(np.array(range(2**nqbits))+nqpu*2/5-1/5, np.abs(format_qft_output(allprobs)), width=2/5, label=label)
plt.legend(); plt.grid(); plt.xlabel(r"$\omega$");
```



In summary,

We have

⇒ defined a simple universal circuit

with an Oracle and a QRoutine from the QLib

⇒ simulated it on a perfect QPU

with the ideal Linear Algebra simulator

⇒ optimized it for specific hardware (superconducting qubits and trapped ions)

with NNizer (for topology),

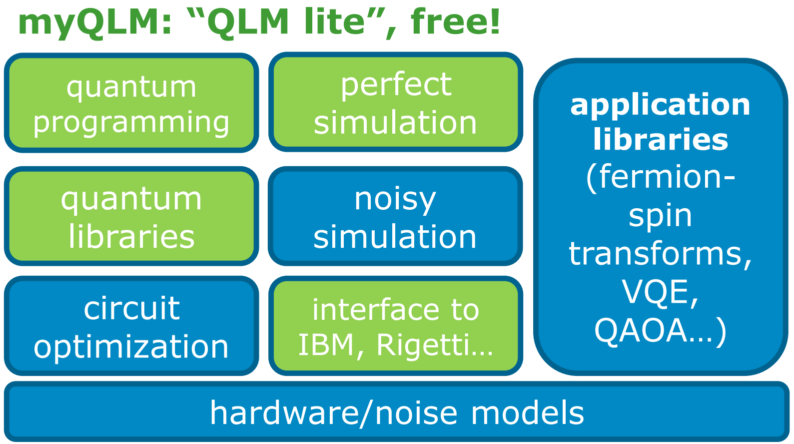
with Pattern-Based Optimizer (for gate replacement and idle time optimization)

with Abstract Gates

with Noise Simulation

Conclusion

- Hands-on tutorial this afternoon (3:30pm) with Jean-Christophe Jaskula
  - make sure you have downloaded the myQLM docker
- Menu:
  - write and run quantum programs
  - varational minimization using quantum circuits



myQLM: "QLM lite", free!

Thank you

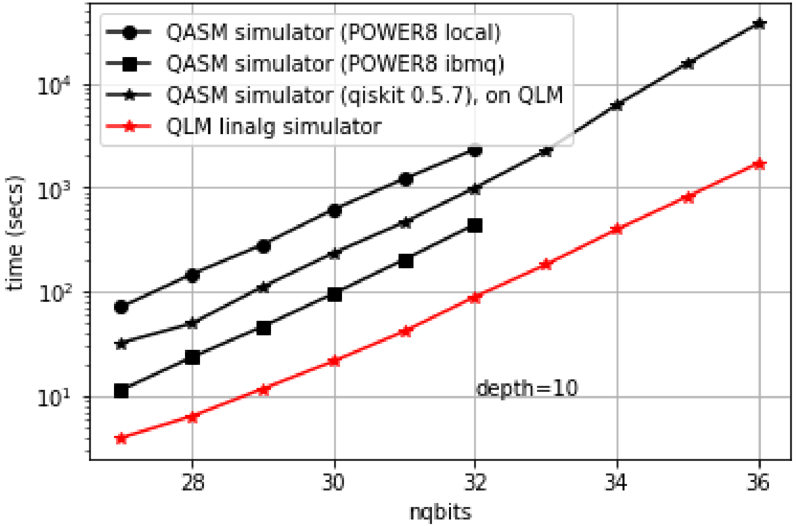
thomas.ayral@atos.net

Speed comparison

QLM vs IBM Qiskit simulator

Perfect simulation

‘Quantum volume’ benchmark circuit

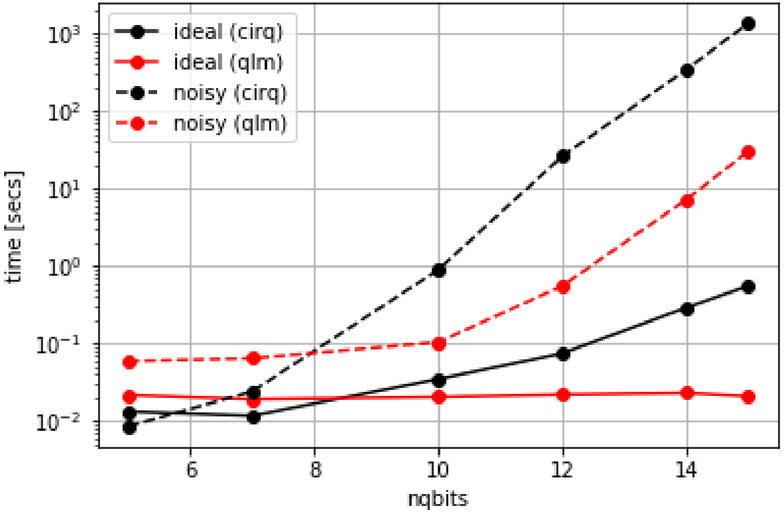


QLM: **qat.linalg** simulator
IBM POWER8 benchmark data from
www.ibm.com/blogs/research/2018/05/quantum-circuits

QLM vs Google cirq simulator

Perfect + noisy simulation

H+CNOT circuit



QLM: **qat.linalg and qat.noisy (deterministic-vectorized)** simulator