Introducing Qibo

An open-source full stack API for quantum simulation and hardware control

Andrea Pasquale and Stavros Efthymiou on the behalf of the Qibo Team

2nd December 2022

UNIVERSITÀ
DEGLI STUDI
DI MILANO

Quantum Computing and HPC

In order to simulate a quantum circuit with n qubits we need to be able to manipulate a 2^n components vector.

In Schrödinger's approach each gate is applied to the state via the following matrix multiplication

$$\psi'(\sigma_1, \ldots, \sigma_n) = \sum_{\boldsymbol{\tau'}} G(\boldsymbol{\tau}, \boldsymbol{\tau'}) \psi(\sigma_1, \ldots, \boldsymbol{\tau'}, \ldots, \sigma_n) \tag{1}$$

where the gate targeting n_{tar} qubits is represented by the $2^{n_{\text{tar}}} \times 2^{n_{\text{tar}}}$ complex matrix $G(\boldsymbol{\tau}, \boldsymbol{\tau'}) = G(\tau_1, \ldots, \tau_{n_{\text{tar}}}, \tau'_1, \ldots, \tau'_{n_{\text{tar}}})$ and $\sigma_i, \tau_i \in \{0, 1\}$.

Example

What can go wrong?

The main issue when building a quantum simulator is the exponential scaling.

From a coding point of view we need to take care of the following:

What can go wrong?

The main issue when building a quantum simulator is the exponential scaling.

From a coding point of view we need to take care of the following:

- Storing in the RAM the state vector

What can go wrong?

The main issue when building a quantum simulator is the exponential scaling.

From a coding point of view we need to take care of the following:

- Storing in the RAM the state vector
- Implement efficiently linear algebra operations

What can go wrong?

The main issue when building a quantum simulator is the exponential scaling.

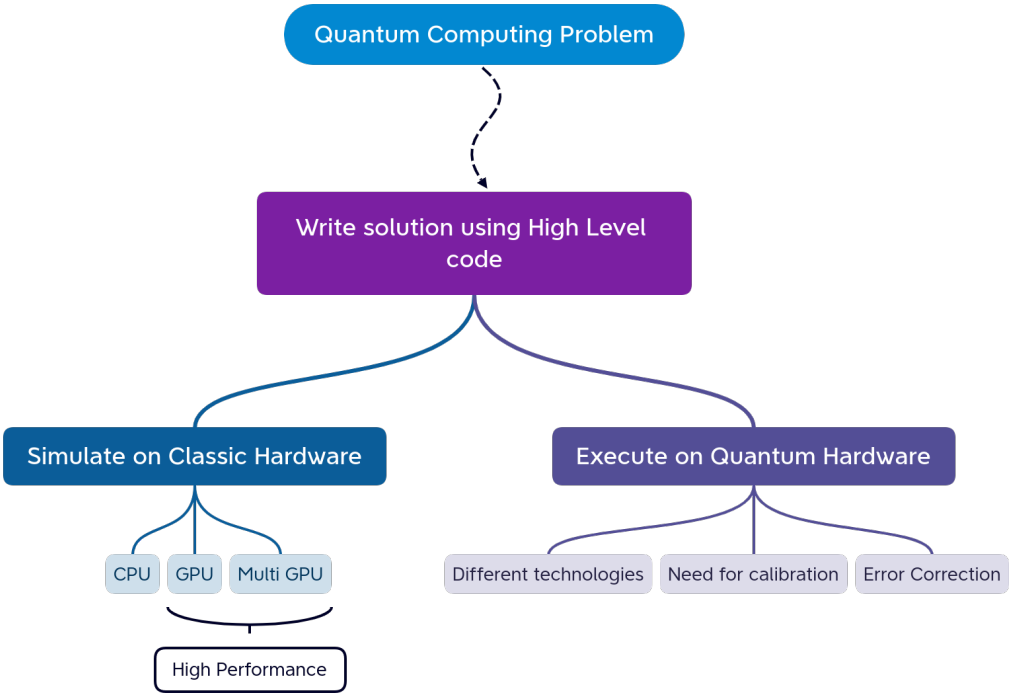
From a coding point of view we need to take care of the following:

- Storing in the RAM the state vector
- Implement efficiently linear algebra operations

Coding a **good quantum simulator** corresponds to coding a **good engine** to perfom linear algebra operations.

$\Rightarrow$ HPC comes into play.

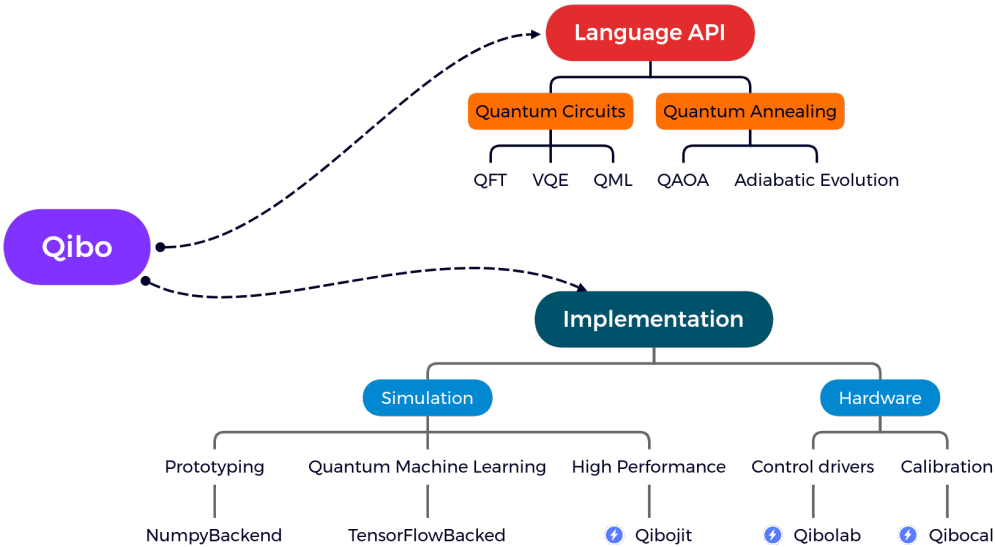
Challenge



Introducing Qibo

Qibo

Qibo is an **open-source** full stack API for **quantum simulation** and quantum hardware control and calibration.



Simulation backends in Qibo

As already done in many quantum simulation library, Qibo provides multiple backends for simulating quantum circuit, i.e. to perform the matrix multiplication of Eq. REF.

The Qibo package is shipped with two basic simulators: `numpy` and `tensorflow`.

`numpy`

`tensorflow`

Numpy Backend

Simulator based on `numpy`:

- `np.ndarray`
- `numpy` primitives

FEATURES

- Cross-architecture (x86, arm64, etc)
- Cross-platform
- Fast for small circuits
- Fast for single-threaded operations

TensorFlow Backend

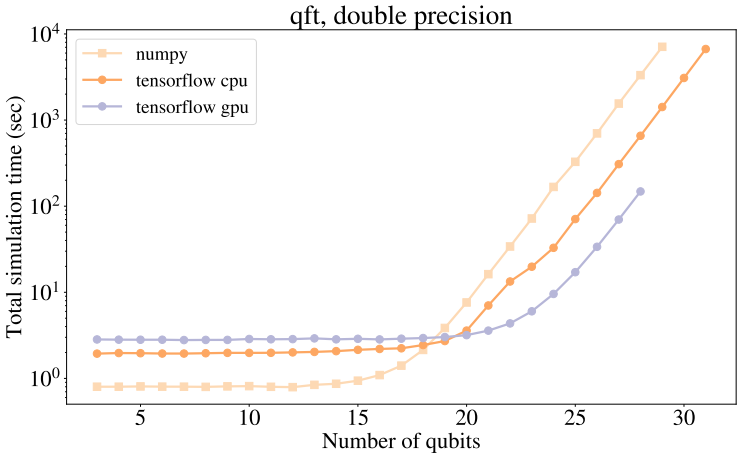
Simulator based on `tensorflow` primitives:

- `tf.Tensor`
- `tf.matmul` and `tf.matmul`

FEATURES

- Multithreading CPU
- Single GPU
- Gradient descent on quantum circuits
- QML using Qibo

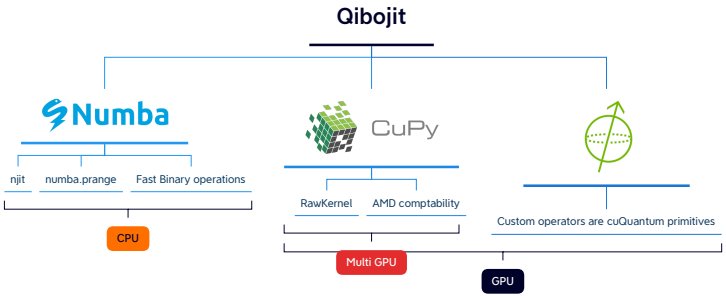
Performance



- Increasing the computational times
- Tensorflow better than numpy
- GPU helps!

Can we do better than this?

To efficiently simulate circuits with large number of qubits we designed a new backend **qibojit**.



FEATURES

- *in-place* updates
- exploit *sparsity of matrix*
- Just-in-Time compilation
- CuQuantum compatibility

What is Just-in-Time (JIT) compilation?

JIT: a method for improving the performance of interpreted programs.

Static compiler: reads a programme, looks at the code and tries to convert it into machine code

Interpreter: looks at the programme, does not convert to machine code and it execute it almost as it is

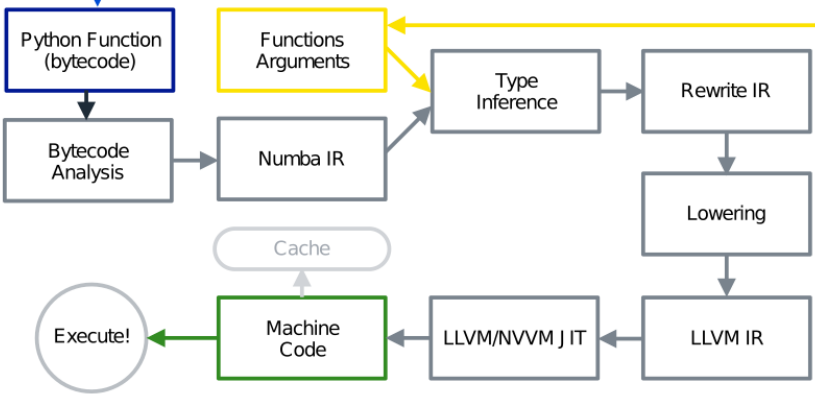
Just-in-Time compiler: starts a programme running an interpreter and dynamically produce machine code based on the observation of the programme

JIT compilers can be **faster** than a static compiler because they can get more information by running the programme instead of just looking at the programme at compile time!

INTRODUCE DRY RUN vs SIMULATION TIME

Numba example

```
@cuda.jit
def axpy(r, a, x, y)
    ...
>>>  axpy (r, a, x,y)
```



Qibojit - Example

```python
from numba import njit, prange

@njit(parallel=True, cache=True)
def apply_gate_kernel(state, gate, target):
    """Operator that applies an arbitrary one-qubit gate.

    Args:
        state (np.ndarray): State vector of size (2 **
↪   nqubits,).
        gate (np.ndarray): Gate matrix of size (2, 2).
        target (int): Index of the target qubit.
    """
    k = 1 << target
    # for one target qubit: loop over half states
    nstates = len(states) // 2
    for g in prange(nstates):
        # generate index with fast binary operations
        i1 = ((g >> m) << (m + 1)) + (g & (k - 1))
        i2 = i1 + k
        state[i1], state[i2] = (gate[0, 0] * state[i1] + \
                                gate[0, 1] * state[i2],
                                gate[1, 0] * state[i1] + \
                                gate[1, 1] * state[i2])
    return state
```

Observations

- `@njit`
- `prange`
- fast binary operations

Qibojit GPU backends : Cupy

Same approach followed using Numba: JIT compilation.

Custom operators implemented using `cupy.RawKernel`:

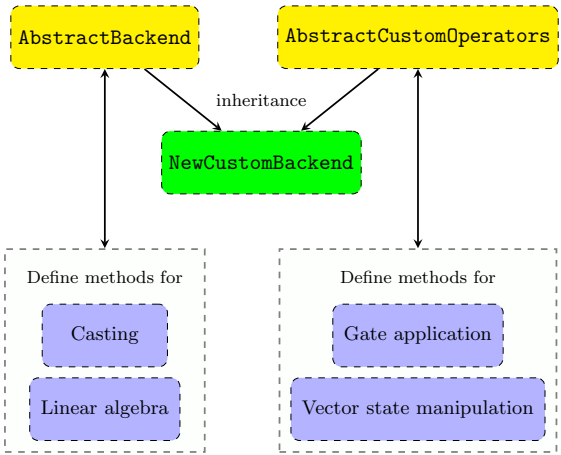
1. Write custom CUDA kernels written in C++

```
apply_gate_kernel = (
    f"""
#include <cupy/complex.cuh>
{_apply_gate}"""
    + """
// C++ implementation of gates.py:apply_gate_kernel()
extern "C"
__global__ void apply_gate_kernel(T* state, long tk, int m, const T* gate) {
  const long g = blockIdx.x * blockDim.x + threadIdx.x;
  const long i = ((long)((long)g >> m) << (m + 1)) + (g & (tk - 1));
  _apply_gate(state[i], state[i + tk], gate);
}
"""
)
```

2. At the first invocation the kernel will be compiled using nvcc
3. After the first invocation it is cached for each device

CuQuantum Backend

Writing custom operators takes time and efforts and can be highly non-trivial. Instead of writing the custom operators yourself the modular layout of Qibo enable the user to write its custom backend.



First successful application: CuQuantum Backend.

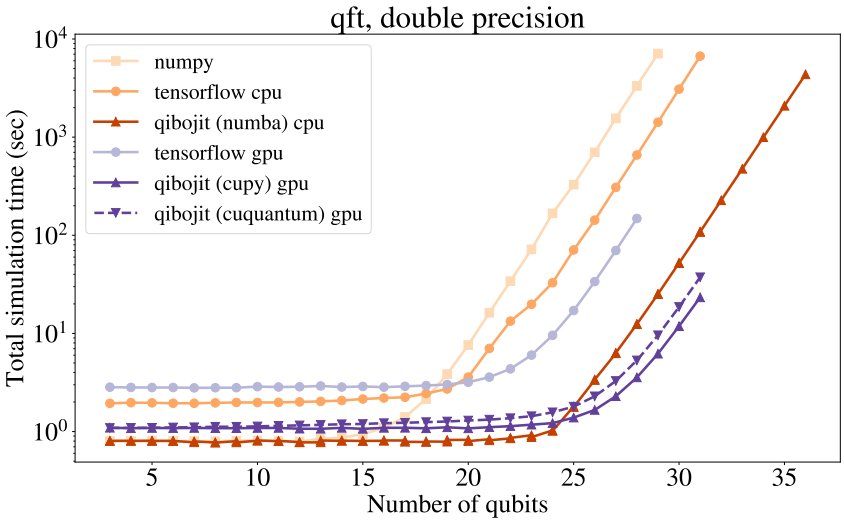
Nvidia CuQuantum

cuQuantum has a Python API which delivers all the functionalities `cuStateVec` and `cuTensorNet` with Python API.

Starting from the `Cupy` we replaced the custom operators with CuQuantum primitives.

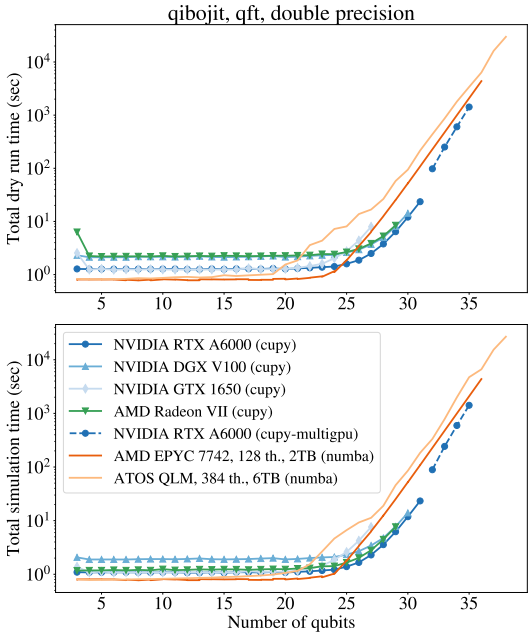
The compatibility with Cupy allows to fallback of the custom operators of the `CupyBackend` to mantain good performance without complicating the code.

Benchmarks



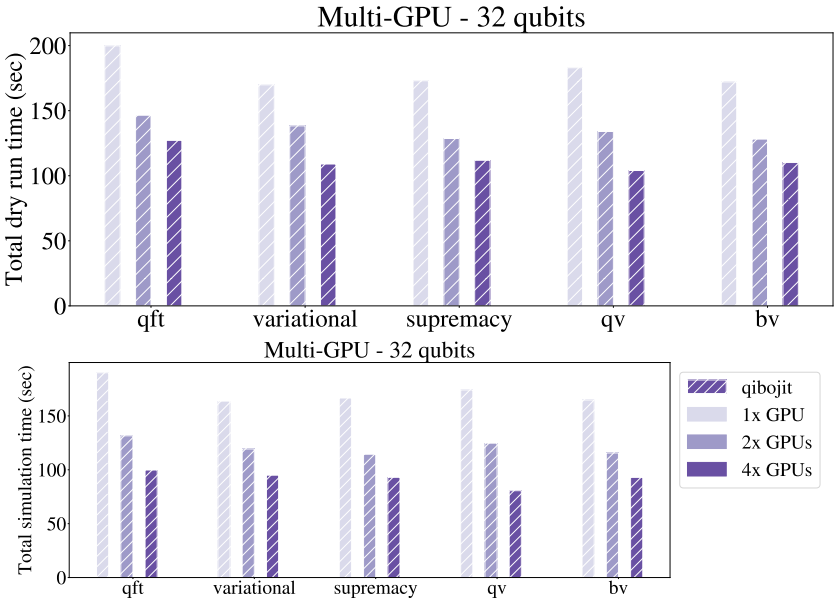
Benchmark library: `https://github.com/qiboteam/qibojit-benchmarks`

Benchmark on different devices

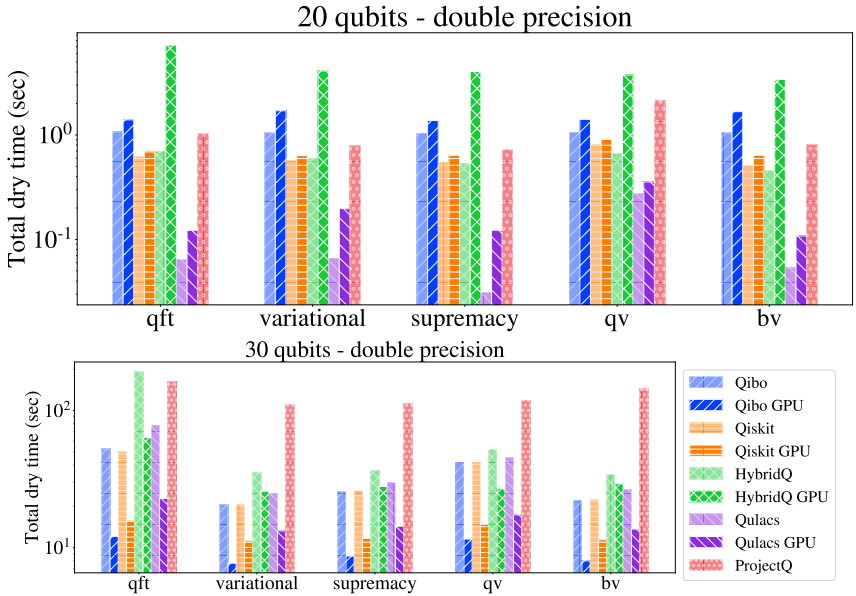


Multi-GPU support

`CupyBackend` supports also multi-GPU architectures



How does Qibo perform against the other libraries?

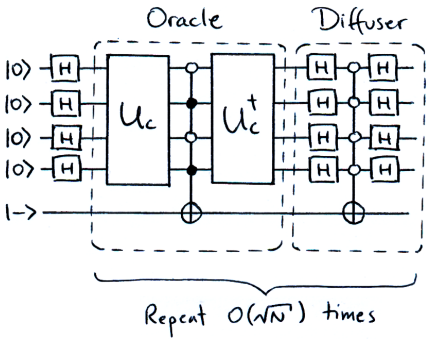


Benchmark library: `https://github.com/qiboteam/qibojit-benchmarks`

Grover's algorithm

What is Grover's Algorithm?

Grover's algorithm is a quantum search algorithm that can search for a value or element in an unsorted set in $\mathcal{O}(\sqrt{N})$ as opposed to classical search algorithms that at worse will find an element in $\mathcal{O}(N)$ time.



Quantum advantage originates from:

- **Superposition**: Perform an operation to all possible solutions at the same time.
- **Interference**: Change sign of the amplitude of the correct solution
- **Entanglement**: Non-trivial sharing of information between states

Important operations

Welsh-Hadamard transform: Apply Hadamard gate to every qubit:

$$\mathbf{H} \to \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

All possible binary strings with equal amplitude amplitude

$$(-1)^{\bar{x} \cdot \bar{y}} 2^{-\frac{n}{2}}$$

Plus or minus sign depending on the number of ones in the initial and final state

Selective face rotation: Apply a phase to just some specific states.

$$\begin{pmatrix} e^{i\phi_{00}} & 0 & 0 & 0 \\ 0 & e^{i\phi_{01}} & 0 & 0 \\ 0 & 0 & e^{i\phi_{10}} & 0 \\ 0 & 0 & 0 & e^{i\phi_{11}} \end{pmatrix}$$

Grover's algorithm uses this matrix with $\phi_i = \pi$ if the state i fulfills a condition, and $\phi_j = 0$ otherwise.

Oracle

Operator that changes the sign of the amplitudes of the quantum states that encode solutions of the problem.

Common way to change the sign once the solution is detected: **use an ancillary qubit**.

Ancilla initialized with an X gate followed by a Hadamard gate:

$$|\psi_a\rangle = HX\,|0\rangle = H\,|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

When the X gate is applied:

$$X\,|\psi_a\rangle = \frac{1}{\sqrt{2}}(|1\rangle - |0\rangle) = -\,|\psi_a\rangle$$

Hint: use CNOT gates to change the sign of the solution.

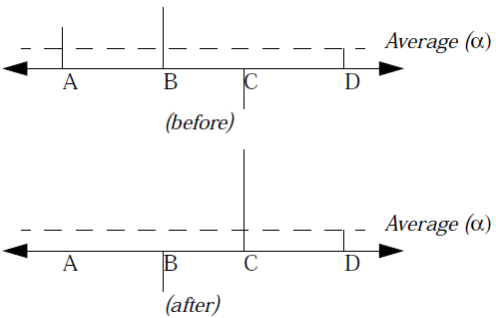
Diffusion transform

The diffusion transform matrix is a matrix defined as follows:

$$D_{ij} = \begin{cases} \frac{2}{N}, & \text{if } i \neq j \\ -1 + \frac{2}{N}, & \text{if } i = j \end{cases} \tag{2}$$

This can be achieved by applying a Welsh-Hadamard transform on all qubits. Then changing the sign of the $|000\ldots000\rangle$ state, and applying once again a Hadamard gate on every qubit.

The diffuser implements an inversion about the average.



Reason for scaling

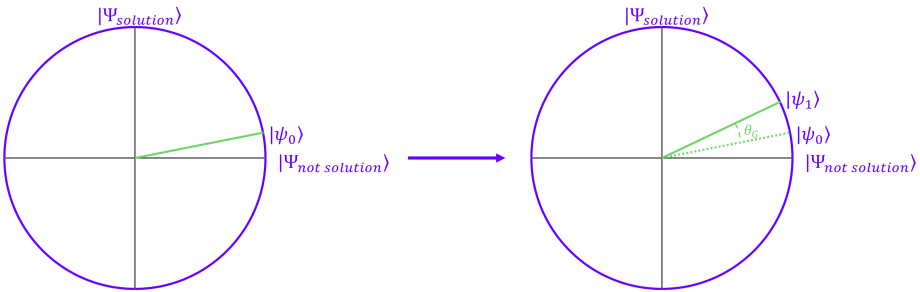
The quantum state can be understood as a superposition of:

$$|\psi_i\rangle = k_i \, |\Psi_{\text{solution}}\rangle + l_i \, |\Psi_{\text{not solution}}\rangle$$

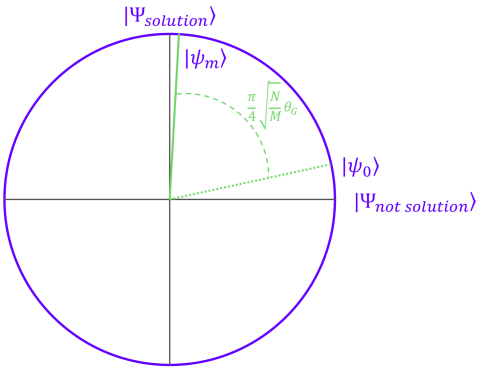
At the start of the algorithm, we can consider $k_0 = \sin\theta$ and $l_0 = \cos\theta$ with $\sin^2\theta = 1/N$.

After the i-th Grover step:

$$k_j = \sin(2j+1)\theta \quad \text{and} \quad l_j = \frac{1}{\sqrt{N-1}} \cos(2j+1)\theta$$



Reason for scaling



In order to achieve $k_m = 1$ it follows that $(2m+1)\theta = \pi/2$

For large number of N:

$$\theta \approx \sin\theta = 1/\sqrt{N}$$

The number of iterations needed is the closest integer to

$$\frac{\pi}{4}\sqrt{N}$$

in case of a single solution.

This can be extended to $\frac{\pi}{4}\sqrt{\frac{N}{M}}$ when considering multiple solutions.

Outlook

Outlook

Qibo is growing to accomodate different tasks:

- ✔ High-performance quantum simulation: **qibojit**
- ✔ Hardware control: **qibolab**
- ✔ Hardware calibration: **qcvv**

What makes Qibo different from other libraries:

- ✚ Public available as an open source project.
- ✚ Modular layout design with possibility of adding
  - a new backend for simulation
  - a new platform for hardware control
- ✚ Community driven effort

`https://github.com/qiboteam/qibo`
`https://qibo.readthedocs.io/en/stable/`

Thanks for listening!