# All You Wanted to Know About Quantum Programming Without Daring to Ask

Benoît Valiron, CentraleSupélec/LMF

October 17th, 2023

Telecom Paris, Palaiseau

# Plan

Quantum Computation

Design of Quantum Programming Languages

Conclusion

# Plan

Quantum Computation

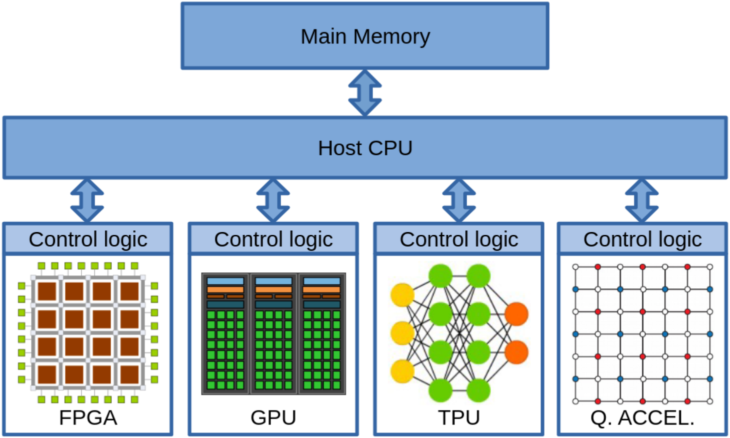
- Model of Computation
- Overview of Quantum Algorithms
- Case Study

Design of Quantum Programming Languages

Conclusion

# Plan

# Model of Computation: Co-Processor




Riesebos, L., et al. "Quantum Accelerated Computer Architectures."
Proc. IEEE International Symposium on Circuits and Systems (ISCAS), 2019.

# Model of computation: Co-Processor

| | Leading technologies in NISQ era[1] | | Candidate technologies beyond NISQ | | |
|---|---|---|---|---|---|
| Qubit type or technology | **Superconducting[2]** | **Trapped ion** | **Photonic** | **Silicon-based[3]** | **Topological[8]** |
| Description of qubit encoding | Two-level system of a superconducting circuit | Electron spin direction of ionized atoms in vacuum | Occupation of a waveguide pair of single photons | Nuclear or electron spin or charge of doped P atoms in Si | Majorana particles in a nanowire |
| Physical qubits[4,5] | IBM: 20, Rigetti: 19, Alibaba: 11, Google: 9 | Lab environment: AQT[6]: 20, IonQ: 14 | **6×3[9]** | **2** | target: 1 in 2018 |
| Qubit lifetime | **~50–100 µs** | **~50 s** | **~150 µs** | **~1–10 s** | target ~100 s |
| Gate fidelity[7] | **~99.4%** | **~99.9%** | **~98%** | **~90%** | target ~99.9999% |
| Gate operation time | **~10–50 ns** | **~3-50 µs** | **~1 ns** | **~1–10 ns** | – |
| Connectivity | Nearest neighbors | All-to-all | To be demonstrated | Nearest neighbor | – |
| Scalability | No major road-blocks near-term | Scaling beyond one trap (>50 qb) | Single photon sources and detection | Novel technology potentially high scalability | ? |
| Maturity or technology readiness level | TRL[10] 5 | TRL 4 | TRL 3 | TRL 3 | TRL 1 |
| Key properties | Cryogenic operation<br>Fast gating<br>Silicon technology | Improves with cryogenic temperatures<br>Long qubit lifetime<br>Vacuum operation | Room temperature<br>Fast gating<br>Modular design | Cryogenic operation<br>Fast gating<br>Atomic-scale size | Estimated:<br>Long lifetime<br>High fidelities |

From 2018 : https://www.bcg.com/publications/2018/next-decade-quantum-computing-how-play

# Model of Computation: Co-Processor

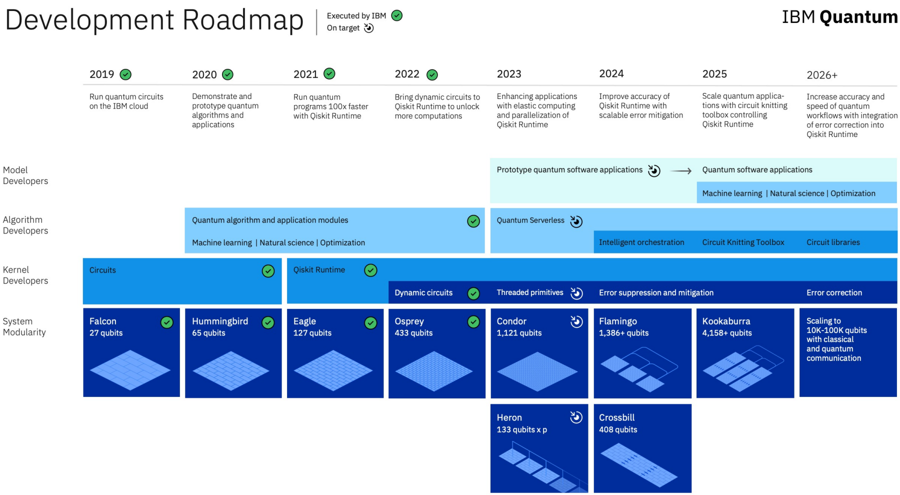
NISQ era

- *Noisy Intermediate Scale Quantum*
- Small-to-medium memory sizes, noisy
- Tradeoffs: Number of Qubit, Noise/Fidelity, Connectivity.
- Challenge: Emulation! (with Tensor Network, *etc*)

LSQ era

- *Large-Scale Quantum*
- Stabilized, logical qubits
- Tradeoffs: Error Correction, Layout, Compilation
- Challenges: Hardware!

# Model of Computation: Co-Processor




Roadmap from 2022

# Model of Computation: Co-Processor

What **COULD** quantum algorithms be good for?

- factoring
  - for breaking modern cryptography
- simulating quantum systems
  - for more efficient molecule distillation procedure
- solving linear systems
  - for high-performance computing
- solving optimization problems
  - for big learning
- ... more than 300 algorithms: `http://math.nist.gov/quantum/zoo/`

# Model of Computation: Co-Processor

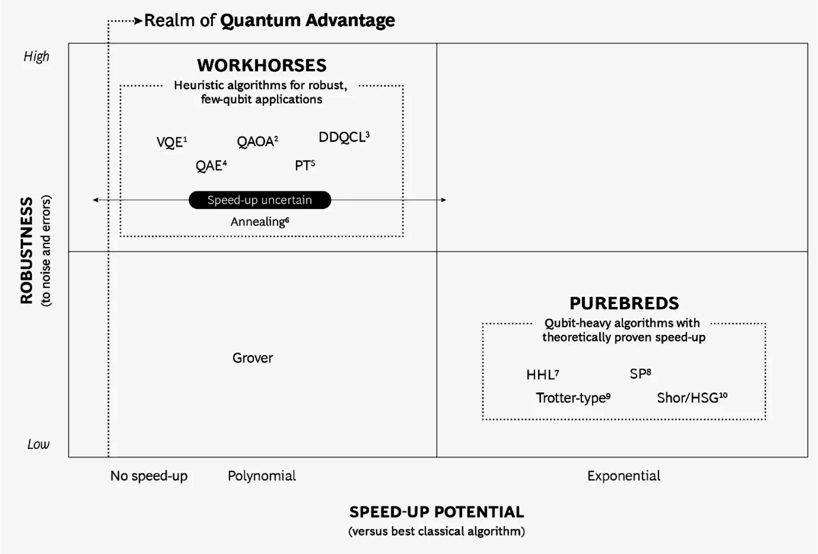
Dichotomy between

- Quantum algorithms as theoretical tools for complexity analysis
- Quantum algorithms as practical tools for concrete problems

Challenges, assuming that a physical machine is available

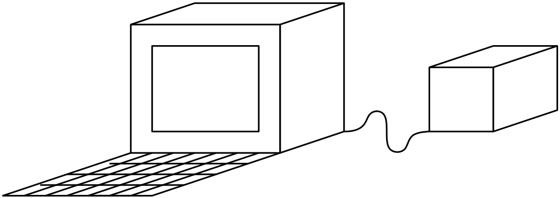
- Designing the right computational model
- Moving from mathematical representation to code
- Resource estimation, optimization
- Compilation and low-level representation
- Debugging/unit testing hard : code analysis and verification

# Model of Computation: Co-Processor

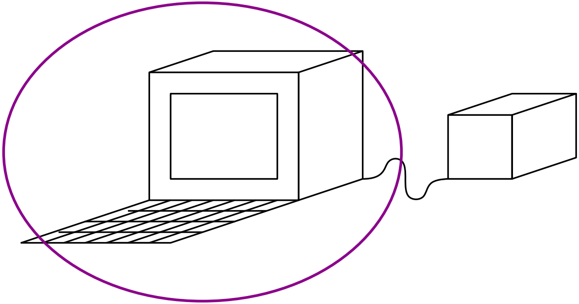



From 2018 : https://www.bcg.com/publications/2018/next-decade-quantum-computing-how-play

# Model of Computation: Quantum Circuit

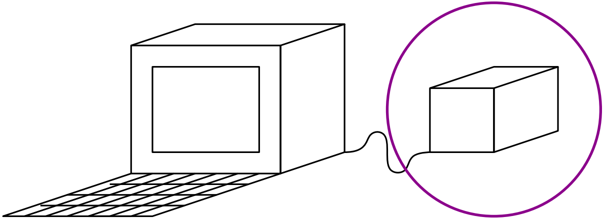


# Model of Computation: Quantum Circuit



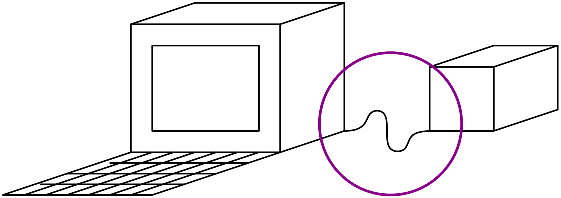
The program lives here

# Model of Computation: Quantum Circuit



This only holds the quantum memory

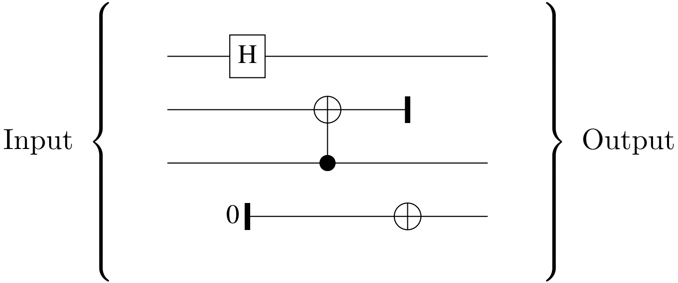
# Model of Computation: Quantum Circuit



Series of instructions/feedbacks

# Model of Computation: Quantum Circuit

- Sequential stream of local instructions
- Updating the memory: Reversible, unitary operations
- Reading the memory: probabilistic, destructive measures




No "quantum loop" or "conditional escape".

# Model of Computation: Quantum Memory

A quantum register with $n$ quantum bits is a complex combination of strings of $n$ bits in a Hilbert space. E.g. for $n = 3$:

$$\begin{array}{ccc} & -\frac{1}{2} & \cdot \left|0\,0\,0\right\rangle \\ + & \frac{1}{2} & \cdot \left|0\,0\,1\right\rangle \\ + & \frac{i}{2} & \cdot \left|1\,1\,0\right\rangle \\ - & \frac{i}{2} & \cdot \left|1\,1\,1\right\rangle \end{array}$$

with a norm condition.

The state of an $n$-qubit register lives in $\mathcal{H}_n \triangleq \mathbb{C}^{2^n}$.
$\rightarrow$ vectors of dimension $2^n$.
$\rightarrow$ basis elements: bitstrings of size $n$.

# Model of Computation: Quantum Memory

The joint state of two quantum registers of sizes $m$ and $n$ lives in the tensor product space $\mathcal{H}_m \otimes \mathcal{H}_n$.

$$\left( \begin{array}{ccc} & \frac{1}{\sqrt{2}} & |0\,0\rangle \\ + & \frac{i}{\sqrt{2}} & |1\,1\rangle \end{array} \right) \otimes \left( \begin{array}{ccc} & \frac{1}{\sqrt{2}} & |0\,0\,1\rangle \\ + & \frac{i}{\sqrt{2}} & |1\,0\,0\rangle \\ + & \frac{i}{\sqrt{2}} & |0\,1\,0\rangle \end{array} \right)$$

# Model of Computation: Quantum Memory

The joint state of two quantum registers of sizes $m$ and $n$ lives in the tensor product space $\mathcal{H}_m \otimes \mathcal{H}_n$.

$$
\begin{aligned}
& \frac{1}{\sqrt{2}} \; |0\,0\rangle \otimes \left( \begin{array}{ccc} & \frac{1}{\sqrt{2}} & |0\,0\,1\rangle \\ + & \frac{i}{\sqrt{2}} & |1\,0\,0\rangle \\ + & \frac{i}{\sqrt{2}} & |0\,1\,0\rangle \end{array} \right) \\
+ \; & \frac{i}{\sqrt{2}} \; |1\,1\rangle \otimes \left( \begin{array}{ccc} & \frac{1}{\sqrt{2}} & |0\,0\,1\rangle \\ + & \frac{i}{\sqrt{2}} & |1\,0\,0\rangle \\ + & \frac{i}{\sqrt{2}} & |0\,1\,0\rangle \end{array} \right)
\end{aligned}
$$

# Model of Computation: Quantum Memory

The joint state of two quantum registers of sizes $m$ and $n$ lives in the tensor product space $\mathcal{H}_m \otimes \mathcal{H}_n$.

$$
\begin{aligned}
& \tfrac{1}{2} && |0\,0\rangle \otimes |0\,0\,1\rangle \\
& \tfrac{i}{2} && |0\,0\rangle \otimes |1\,0\,0\rangle \\
& \tfrac{i}{2} && |0\,0\rangle \otimes |0\,1\,0\rangle \\
+ \quad & \tfrac{i}{2} && |1\,1\rangle \otimes |0\,0\,1\rangle \\
& \tfrac{-1}{2} && |1\,1\rangle \otimes |1\,0\,0\rangle \\
& \tfrac{-1}{2} && |1\,1\rangle \otimes |0\,1\,0\rangle
\end{aligned}
$$

# Model of Computation: Quantum Memory

The joint state of two quantum registers of sizes $m$ and $n$ lives in the tensor product space $\mathcal{H}_m \otimes \mathcal{H}_n$.

$$
\begin{array}{rll}
 & \frac{1}{2} & |0\,0\,0\,0\,1\rangle \\
 & \frac{i}{2} & |0\,0\,1\,0\,0\rangle \\
 & \frac{i}{2} & |0\,0\,0\,1\,0\rangle \\
+ & \frac{i}{2} & |1\,1\,0\,0\,1\rangle \\
 & \frac{-1}{2} & |1\,1\,1\,0\,0\rangle \\
 & \frac{-1}{2} & |1\,1\,0\,1\,0\rangle
\end{array}
$$

# Model of Computation: Quantum Memory

The joint state of two quantum registers of sizes $m$ and $n$ lives in the tensor product space $\mathcal{H}_m \otimes \mathcal{H}_n$.

$$
\begin{array}{lll}
 & \frac{1}{2} & |0\,0\,0\,0\,1\rangle \\
 & \frac{i}{2} & |0\,0\,1\,0\,0\rangle \\
 & \frac{i}{2} & |0\,0\,0\,1\,0\rangle \\
+ & \frac{i}{2} & |1\,1\,0\,0\,1\rangle \\
 & \frac{-1}{2} & |1\,1\,1\,0\,0\rangle \\
 & \frac{-1}{2} & |1\,1\,0\,1\,0\rangle
\end{array}
$$

$\mathcal{H}_m \otimes \mathcal{H}_n$ is of dimension $2^m \times 2^n$.

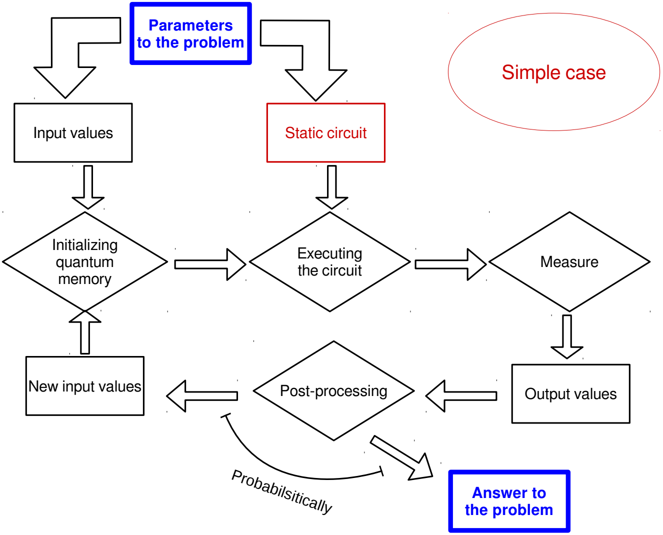
# Model of computation: Quantum Memory

Takeaway

- Classical data in superposition.
- Internal updates are reversible and local.
- No-cloning theorem: quantum information cannot be copied
- Probablistic, destructive reading
- No control flow in the quantum co-processor
- Some parallelism thanks to data superposition

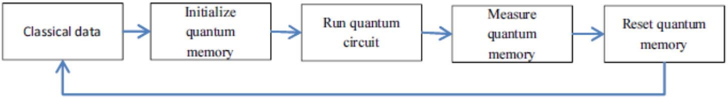
# Plan

# Structure of Quantum Algorithms

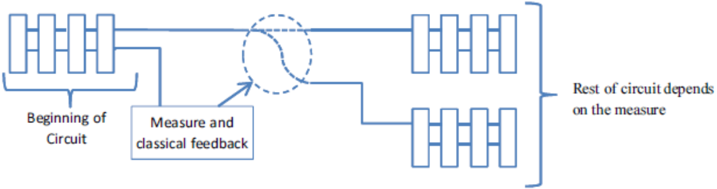


# Structure of Quantum Algorithms

## Simple scheme




## General scheme




- Quantum circuits $\neq$ hardware design
- Hybrid classical/quantum computation

# Internal of current quantum algorithms

The techniques used to described quantum algorithms are diverse.

1. Quantum primitives.
   - Quantum Fourier Transform.
     Assuming $\omega = 0.\mathtt{xy}$, we want

$$\begin{array}{rl} & (e^{2\pi i\omega})^0 \cdot 00 \\ + & (e^{2\pi i\omega})^1 \cdot 01 \\ + & (e^{2\pi i\omega})^2 \cdot 10 \\ + & (e^{2\pi i\omega})^3 \cdot 11 \end{array} \qquad \longmapsto \qquad 1 \cdot \mathtt{xy}$$

# Internal of current quantum algorithms

The techniques used to described quantum algorithms are diverse.

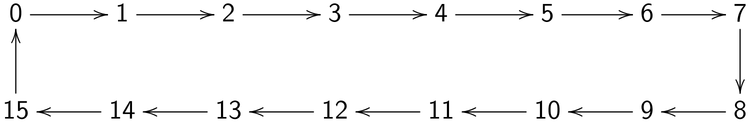
1. Quantum primitives.
   - Quantum Fourier Transform.
   - Amplitude amplification.
     Qubit 3 in state 1 means good.

$$
\begin{array}{rl}
 & \alpha_0 \cdot 000 \\
+ & \alpha_1 \cdot 011 \\
+ & \alpha_2 \cdot 100 \\
+ & \alpha_3 \cdot 110
\end{array}
\longmapsto
\begin{array}{rl}
 & {\scriptstyle \alpha_0} \cdot 000 \\
+ & \boldsymbol{\alpha_1} \cdot 011 \\
+ & {\scriptstyle \alpha_2} \cdot 100 \\
+ & {\scriptstyle \alpha_3} \cdot 110
\end{array}
$$

# Internal of current quantum algorithms

The techniques used to described quantum algorithms are diverse.

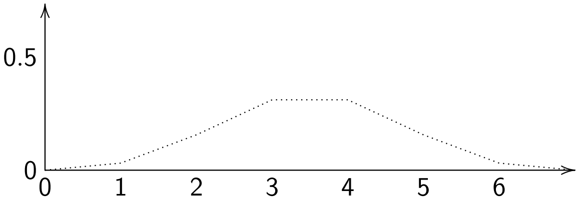
1. Quantum primitives.
   - Quantum Fourier Transform.
   - Amplitude amplification.
   - Quantum walk.



# Internal of current quantum algorithms

The techniques used to described quantum algorithms are diverse.

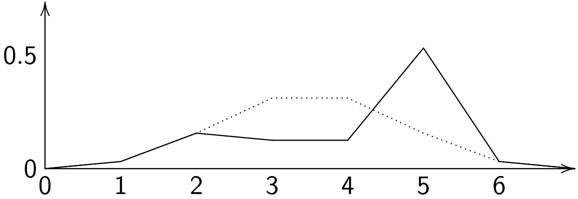
1. Quantum primitives.
   - Quantum Fourier Transform.
   - Amplitude amplification.
   - Quantum walk.
     After 5 steps of a probabilistic walk:



# Internal of current quantum algorithms

The techniques used to described quantum algorithms are diverse.

1. Quantum primitives.
   - Quantum Fourier Transform.
   - Amplitude amplification.
   - Quantum walk.
     After 5 steps of a quantum walk:



# Internal of current quantum algorithms

The techniques used to described quantum algorithms are diverse.

2. Oracles.
   - Take a classical function $f : \texttt{Bool}^n \to \texttt{Bool}^m$.
   - Construct

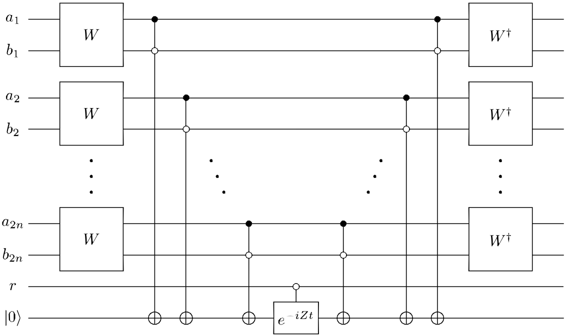
$$\begin{array}{rccc} \overline{f}: & \texttt{Bool}^{n+m} & \longrightarrow & \texttt{Bool}^{n+m} \\ & (x, y) & \longmapsto & (x, y \oplus f(x)) \end{array}$$

   - Build the unitary $U_f$ acting on $n + m$ qubits computing $\overline{f}$.

# Internal of current quantum algorithms

The techniques used to described quantum algorithms are diverse.

3. Blocks of loosely-defined low-level circuits.

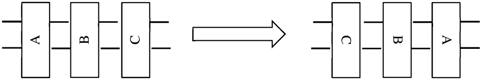



This is not a formal specification!

# Internal of current quantum algorithms

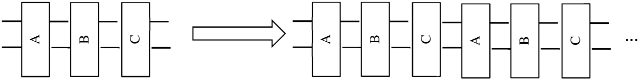
The techniques used to described quantum algorithms are diverse.

4. High-level operations on circuit:
   - Circuit inversion.




   (the circuit needs to be reversible. . . )

   - Repetition of the same circuit.




   (needs to have the same input and output arity. . . )

   - Controlling of circuits

# Internal of current quantum algorithms

The techniques used to described quantum algorithms are diverse.

5. Classical processing.
    - Generating the circuit...
    - Computing the input to the circuit.
    - Processing classical feedback in the middle of the computation.
    - Analyzing the final answer (and possibly starting over).

# Plan

# Case study: QLS algorithm

Considering a vector $\vec{b}$ and the system

$$A \cdot \vec{x} = \vec{b},$$

compute the value of $\quad \langle \vec{x} \,|\, \vec{r} \rangle \quad$ for some vector $\vec{r}$.

Practical situation: the matrix $A$ corresponds to the finite-element approximation of the scattering problem:

`arXiv:1505.06552`

# Case study: QLS algorithm

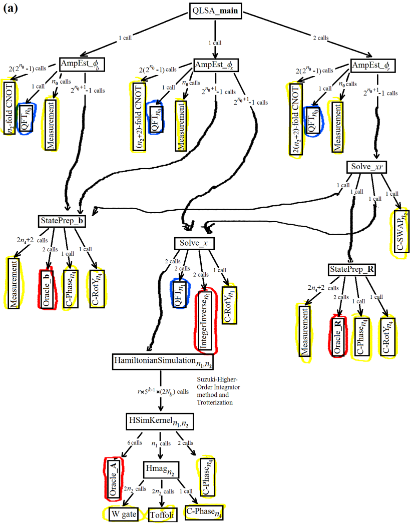
Three oracles:

- for $\vec{r}$ and for $\vec{b}$: input an index, output (the representation of) a complex number
- for $A$: input two indexes, output also a complex number

Many quantum primitives

- Amplitude estimation
- Phase estimation
- Amplitude amplification
- Hamiltonian simulation

# Case study: QLS algorithm




- Yellow: Elementary gates.
- Red: Oracles.
- Blue: QFT's.
- Black: Subroutines.
- *Parameters:*
  Dimensions of the space;
  Precision for each of the vectors;
  Allowed error;
  Various parameters for $A$...
  In total, 19 parameters.

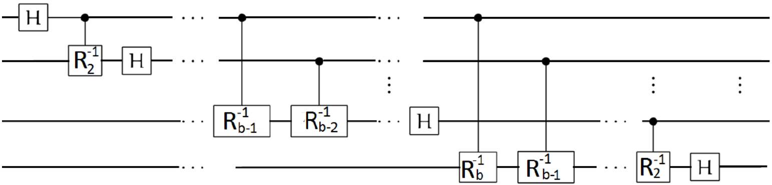
# Case study: QLS algorithm

Oracle R is given by the function

```
calcRweights y nx ny lx ly k theta phi =
    let (xc',yc') = edgetoxy y nx ny in
    let xc = (xc'-1.0)*lx - ((fromIntegral nx)-1.0)*lx/2.0 in
    let yc = (yc'-1.0)*ly - ((fromIntegral ny)-1.0)*ly/2.0 in
    let (xg,yg) = itoxy y nx ny in
    if (xg == nx) then
        let i = (mkPolar ly (k*xc*(cos phi)))*(mkPolar 1.0 (k*yc*(sin phi)))*
                ((sinc (k*ly*(sin phi)/2.0)) :+ 0.0) in
        let r = ( cos(phi) :+ k*lx )*((cos (theta - phi))/lx :+ 0.0) in i * r
    else if (xg==2*nx-1) then
        let i = (mkPolar ly (k*xc*cos(phi)))*(mkPolar 1.0 (k*yc*sin(phi)))*
                ((sinc (k*ly*sin(phi)/2.0)) :+ 0.0) in
        let r = ( cos(phi) :+ (- k*lx))*((cos (theta - phi))/lx :+ 0.0) in i * r
    else if ( (yg==1) && (xg<nx) ) then
        let i = (mkPolar lx (k*yc*sin(phi)))*(mkPolar 1.0 (k*xc*cos(phi)))*
                ((sinc (k*lx*(cos phi)/2.0)) :+ 0.0) in
        let r = ( (- sin phi) :+ k*ly )*((cos(theta - phi))/ly :+ 0.0) in i * r
    else if ( (yg==ny) && (xg<nx) ) then
        let i = (mkPolar lx (k*yc*sin(phi)))*(mkPolar 1.0 (k*xc*cos(phi)))*
                ((sinc (k*lx*(cos phi)/2.0)) :+ 0.0) in
        let r = ( (- sin phi) :+ (- k*ly) )*((cos(theta - phi)/ly) :+ 0.0) in i * r
    else 0.0 :+ 0.0
```

# Case study: circuit snippets

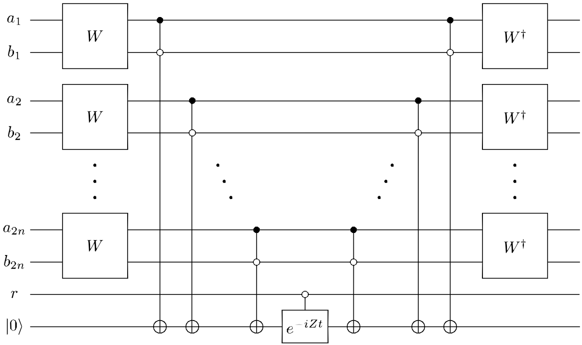
The algorithms create circuits whose sizes and shapes depend on the parameters. E.g. the size of the input register:




(QFT)

# Case study: circuit snippets

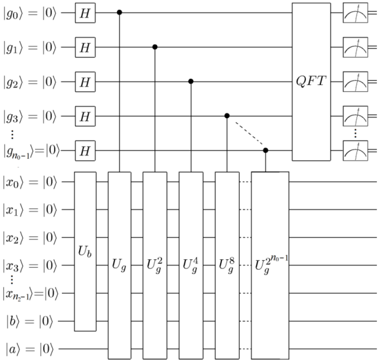
The algorithms create circuits whose sizes and shapes depend on the parameters. E.g. the size of the input register:




(diffusion step in BWT)

# Case study: circuit snippets

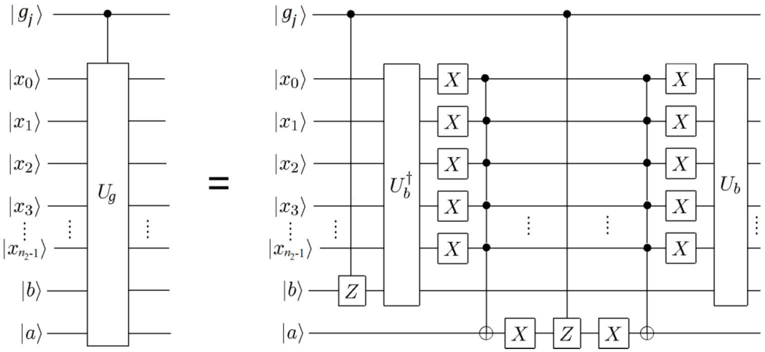
The algorithms create circuits whose sizes and shapes depend on the parameters. E.g. the size of the input register:




(piece of one subroutine of QLS)

# Case study: circuit snippets

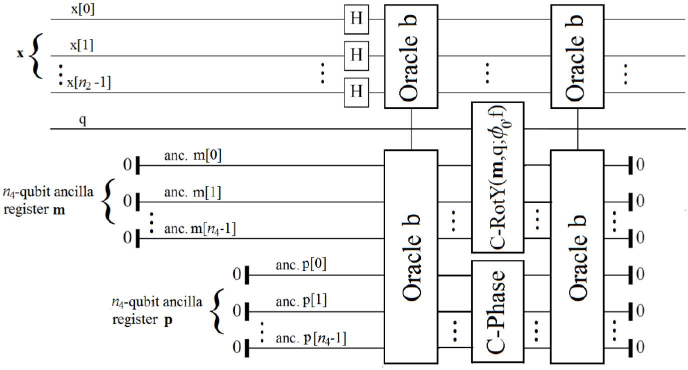
The algorithms create circuits whose sizes and shapes depend on the parameters. E.g. the size of the input register:




(the subroutine $U_g$)

# Case study: circuit snippets

The algorithms create circuits whose sizes and shapes depend on the parameters. E.g. the size of the input register:




(the subroutine $U_b$)

# Plan

# Lessons learned

- Circuit construction
  - Procedural: Instruction-based, one line at a time
  - Declarative: Circuit combinators
    - Inversion
    - Repetition
    - Control
    - Computation/uncomputation
- Circuits as inputs to other circuits
- Regularity with respect to the size of the input
- Distinction parameter / input
- Need for automation for oracle generation

# Plan

# Programming framework

Two approaches

- Circuit as a record
  - One type `circuit`
  - Qubits $\equiv$ wire numbers
  - Native: vertical/horizontal concatenation, gate addition
- Circuit as a function
  - Qubits $\equiv$ first-order objects
  - Input wires $\equiv$ function input
  - Output wires $\equiv$ function output

# Circuits as Records

Simplest model: an object holding all of the circuit structure

- Classical wires
- Quantum wires
- List of gates (or directed acyclic graph)
- This is for instance QisKit/QASM model

In this system

- Static circuit
- No high-level hybrid interaction: sequence
  1. circuit generation
  2. circuit evaluation
  3. measure
  4. classical post-processing
  5. back to (1)

# Circuits as Records

Procedural construction (QisKit)

```
q = QuantumRegister(5)
c = ClassicalRegister(1)
circ = QuantumCircuit(q,c)

circ.h(q[0])
for i in range(1,5):
  circ.cx(q[0], q[i])
circ.meas(q[4],c[0])
```

- Static ID For registers
- Wires are numbers
- Gate $\equiv$ instruction
- Classical control: Circuit building
- Explicit "run" of circuit

Combinators: return a record `circuit`

- `circ.control(4)`
- `circ.inverse()`
- `circ.append(other-circuit)`

# Circuits as Functions

A function

```
a -> Circ b
```

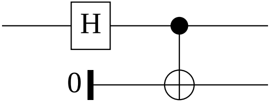
- Inputs something of type `a`
- Outputs something of type `b`
- As a side-effect, generates a circuit snippet.

Or

- Inputs a value of type `a`
- Outputs a computation of type `b`

# Circuits as Functions

The circuit




can be typed with

```
Qubit -> Circ (Qubit,Qubit)
```

- Inputs one qubit
- Outputs a pair of qubits
- Spits out some gates when evaluated

The gates are however encapsulated in the function

# Circuits as Functions

## Representing circuits (Quipper)

Name of circuit — Input: one wire — Indeed a circuit — Two output wires

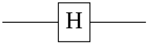
```
myCircuit :: Qubit -> Circ (Qubit, Qubit)

myCircuit q = do
      ...
      ...
      return (x,y)
```

- `myCircuit` — Name of circuit
- `Qubit` — Input: one wire
- `Circ` — Indeed a circuit
- `Qubit)` — Two output wires
- `do` — Start a procedural sequence
- `q` — The name of the input wire
- `(x,y)` — The two output wires

# Circuits as Functions

Procedural presentation of circuits:

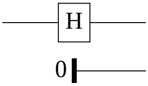
```
prog :: Qubit -> Circ (Qubit,Qubit)
prog q = do
  hadamard_at q
  r <- qinit False
  qnot_at r ‘controlled‘ q
  return (q,r)
```



# Circuits as Functions

Procedural presentation of circuits:

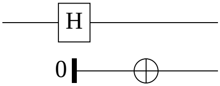
```
prog :: Qubit -> Circ (Qubit,Qubit)
prog q = do
  hadamard_at q
  r <- qinit False
  qnot_at r ‘controlled‘ q
  return (q,r)
```



# Circuits as Functions

Procedural presentation of circuits:

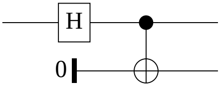
```
prog :: Qubit -> Circ (Qubit,Qubit)
prog q = do
  hadamard_at q
  r <- qinit False
  qnot_at r `controlled` q
  return (q,r)
```



# Circuits as Functions

Procedural presentation of circuits:

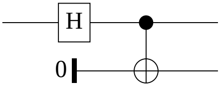
```
prog :: Qubit -> Circ (Qubit,Qubit)
prog q = do
  hadamard_at q
  r <- qinit False
  qnot_at r ‘controlled‘ q
  return (q,r)
```



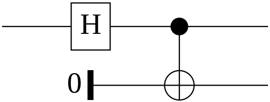
# Circuits as Functions

Procedural presentation of circuits:

```
prog :: Qubit -> Circ (Qubit,Qubit)
prog q = do
  hadamard_at q
  r <- qinit False
  qnot_at r ‘controlled‘ q
  return (q,r)
```



# Circuits as Functions



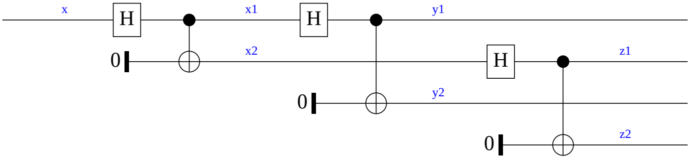

```
import Quipper

circ ::
   Qubit -> Circ (Qubit,Qubit)
circ x = do
   y <- qinit False
   hadamard_at x
   qnot_at y `controlled` x
   return (x,y)
```

- Qubits $\equiv$ first-class variable
- Circuit $\equiv$ function
- Wires $\equiv$ inputs and outputs
- Mix classical/quantum

# Circuits as Functions

Wires do not have "fixed" location

```
circ2 :: Qubit -> Circ ()
circ2 x = do
   (x1,x2) <- circ x
   (y1,y2) <- circ x1
   (z1,z2) <- circ x2
   return ()
```




- Qubit $\not\equiv$ Wire number
- Circuits as functions: can be applied
- More expressive types

# Circuit Combinators

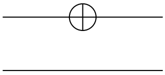
## Controls

```
controlled ::  ControlSource b => Circ a -> b -> Circ a
```

- Input: Computation generating a circuit C
- Input: Something that can be controlled (e.g. `Qubit`)
- Output: Computation generating the controlled circuit C

## Example

```
prog :: (Qubit,Qubit) -> Circ (Qubit,Qubit)
prog (p,q) = do
  qnot_at p
  return (p,q)
```



# Circuit Combinators

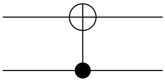
## Controls

```
controlled :: ControlSource b => Circ a -> b -> Circ a
```

- Input: Computation generating a circuit C
- Input: Something that can be controlled (e.g. `Qubit`)
- Output: Computation generating the controlled circuit C

## Example

```
prog :: (Qubit,Qubit) -> Circ (Qubit,Qubit)
prog (p,q) = do
  controlled (qnot_at p) q
  return (p,q)
```



# Circuit Combinators

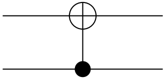
## Controls

```
controlled :: ControlSource b => Circ a -> b -> Circ a
```

- Input: Computation generating a circuit C
- Input: Something that can be controlled (e.g. `Qubit`)
- Output: Computation generating the controlled circuit C

## Example

```
prog :: (Qubit,Qubit) -> Circ (Qubit,Qubit)
prog (p,q) = do
  qnot_at p `controlled` q
  return (p,q)
```



with infix notation

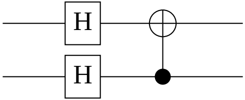
# Circuit Combinators

## Controls

```
controlled :: ControlSource b => Circ a -> b -> Circ a
```

## It works on any (reversible) circuit

```
prog :: (Qubit,Qubit) -> Circ (Qubit,Qubit)
prog (p,q) = do
  hadamard_at p
  hadamard_at q
  qnot_at p `controlled` q
  return (p,q)
```



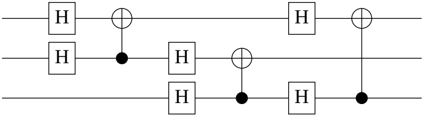
# Circuit Combinators

## Controls

```
controlled :: ControlSource b => Circ a -> b -> Circ a
```

## It works on any (reversible) circuit

```
prog :: (Qubit,Qubit) -> Circ (Qubit,Qubit)
...

prog2 :: (Qubit,Qubit,Qubit) -> Circ ()
prog2 (p,q,r) = do
  prog (p,q)
  prog (q,r)
  prog (p,r)
  return ()
```



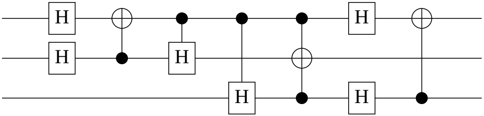
# Circuit Combinators

## Controls

```
controlled :: ControlSource b => Circ a -> b -> Circ a
```

## It works on any (reversible) circuit

```
prog :: (Qubit,Qubit) -> Circ (Qubit,Qubit)
...

prog2 :: (Qubit,Qubit,Qubit) -> Circ ()
prog2 (p,q,r) = do
  prog (p,q)
  prog (q,r) `controlled` p
  prog (p,r)
  return ()
```



# Plan

# Families of Circuits

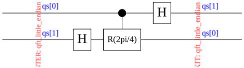
A program

- Inputs classical parameters
- Construct a circuit from these parameters
- Run the circuit

Circuits: parametrized families

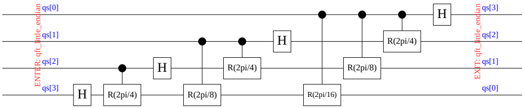
# Families of Circuits

Example: QFT



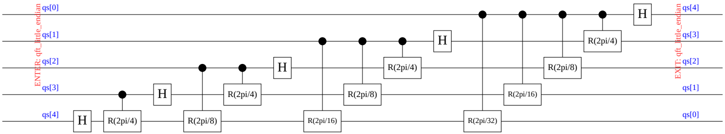
# Families of Circuits

## Example: QFT



# Families of Circuits

## Example: QFT



# Families of Circuits

With the help of lists:

```
myCircuit :: [Qubit] -> Circ [Qubit]

myCircuit qs = do
      ...
      ...
      return ...
```

- Name of circuit
- Input a **list of wires**
- Indeed a circuit
- Output a **list of wires**
- Start a procedural sequence
- The name of the input list
- The output list

# Families of Circuits

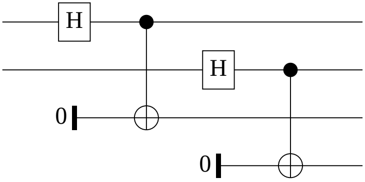
List combinators, e.g.

```
mapM ::  (a -> Circ b) -> [a] -> Circ [b]
```

Mixed presentation of circuits:

```
prog :: Qubit -> Circ (Qubit,Qubit)
prog q = do
  hadamard_at q
  r <- qinit False
  qnot_at r ‘controlled‘ q
  return (q,r)

prog2 :: [Qubit] -> Circ [(Qubit,Qubit)]
prog2 l = mapM prog l
```

List of size 2:



# Families of Circuits

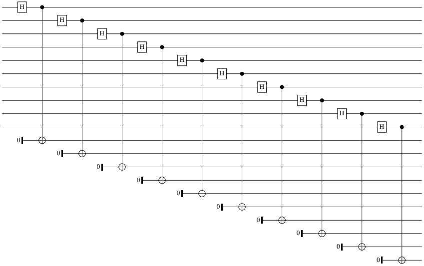
List combinators, e.g.

```
mapM ::  (a -> Circ b) -> [a] -> Circ [b]
```

Mixed presentation of circuits:

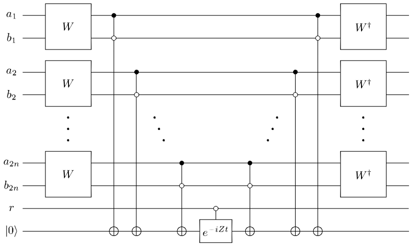
List of size 10:

```
prog :: Qubit -> Circ (Qubit,Qubit)
prog q = do
  hadamard_at q
  r <- qinit False
  qnot_at r ‘controlled‘ q
  return (q,r)

prog2 :: [Qubit] -> Circ [(Qubit,Qubit)]
prog2 l = mapM prog l
```



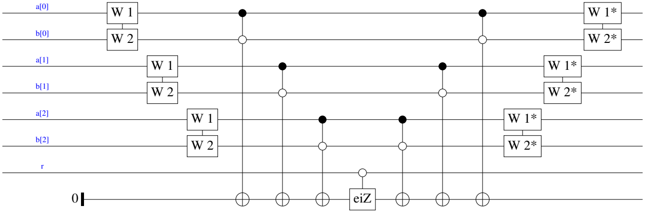
# Example: BWT

```
import Quipper

w :: (Qubit,Qubit) -> Circ (Qubit,Qubit)
w = named_gate "W"

toffoli :: Qubit -> (Qubit,Qubit) -> Circ Qubit
toffoli d (x,y) =
  qnot d `controlled` x .==. 1 .&&. y .==. 0

eiz_at :: Qubit -> Qubit -> Circ ()
eiz_at d r =
  named_gate_at "eiZ" d `controlled` r .==. 0

circ :: [(Qubit,Qubit)] -> Qubit -> Circ ()
circ ws r = do
  label (unzip ws,r) (("a","b"),"r")
  d <- qinit 0
  mapM_ w ws
  mapM_ (toffoli d) ws
  eiz_at d r
  mapM_ (toffoli d) (reverse ws)
  mapM_ (reverse_generic w) (reverse ws)
  return ()

main = print_generic EPS circ (replicate 3 (qubit,qubit)) qubit
```



# Example: BWT

Result (3 wires):



# Example: BWT

Result (30 wires):

# Plan

Quantum Computation

Design of Quantum Programming Languages

Conclusion

# Conclusion

## Quantum Computation and Programming

- A lot of classical programming!
- Many challenges, both at high and low-level
- Research active to match theory with practice.

## The QuaCS team at LMF:

(On the other side of the N118)

- Design of quantum programming languages
- Model of quantum computation
- Compilation toolchain
- Circuit synthesis and optimization
- Intermediate representation
- Code certification

QuaCS