Tutorial

A step-by-step introduction to the main facilities of QuEST-MMA.

Table of contents:

Connecting to QuEST

Import the QuEST-MMA package . Further functions will be loaded once connected to an QuEST environment.

```
Import["https://qtechtheory.org/questlink.m"]
```

One then connects to a QuEST runtime environment, which can be local or remote.

```
? CreateRemoteQuESTEnv
? CreateLocalQuESTEnv
? CreateDownloadedQuESTEnv
```

CreateRemoteQuESTEnv[ip, port1, port2] connects to a remote QuESTlink server at ip, at the given ports, and defines several QuEST functions, returning a link object. This should be called once. The QuEST function defintions can be cleared with DestroyQuESTEnv[link].

CreateLocalQuESTEnv[fn] connects to a local 'quest_link' executable, located at fn, running single–CPU QuEST. This should be called once. The QuEST function defintions can be cleared with DestroyQuESTEnv[link].
CreateLocalQuESTEnv[] connects to a 'quest_link' executable in the working directory.

CreateDownloadedQuESTEnv[] downloads a precompiled single–CPU QuESTlink binary (specific to your operating system) directly from Github, then locally connects to it. This should be called once, before using the QuESTlink API.
CreateDownloadedQuESTEnv[os] forces downloaded of the binary for operating system 'os', which must one of {Windows, Linux, Unix, MacOS, MacOSX}.

We'll automatically download a QuEST executable and locally connect.

```
env = CreateDownloadedQuESTEnv[];
```

This loads further package functions and circuit symbols, listed below.

```
? QuEST`*
```

▼ QuEST`

ApplyCircuit	CloneQureg	GetAllQuregs	MixTwoQubitDepolarising
ApplyPauliSum	CollapseToOutcome	GetAmp	Operator
CalcCircuitMatrix	CreateDensityQureg	GetPauliSumFromCoeffs	P
CalcDensityInnerProduct	CreateDensityQuregs	GetQuregMatrix	PackageExport
CalcDensityInnerProducts	CreateDownloadedQuESTEnv	H	R
CalcExpecPauliProd	CreateLocalQuESTEnv	InitClassicalState	Rx
CalcExpecPauliSum	CreateQureg	InitPlusState	Ry
CalcFidelity	CreateQuregs	InitPureState	Rz
CalcHilbertSchmidtDistance	CreateRemoteQuESTEnv	InitStateFromAmps	S
CalcInnerProduct	Damp	InitZeroState	SetQuregMatrix
CalcInnerProducts	Deph	IsDensityMatrix	SetWeightedQureg
CalcPauliSumMatrix	Depol	Kraus	SWAP
CalcProbOfOutcome	DestroyAllQuregs	M	T
CalcPurity	DestroyQuESTEnv	MixDamping	U
CalcQuregDerivs	DestroyQureg	MixDephasing	X
CalcTotalProb	DrawCircuit	MixDepolarising	Y
Circuit	G	MixTwoQubitDephasing	Z

Creating quantum registers

Now that we're connected to a QuEST runtime environment, we can allocate quantum registers as state vectors or density matrices.

```
numQb = 9;
ψ = CreateQureg[numQb];
ρ = CreateDensityQureg[numQb];
```

These registers are stored in the environment which may be remote. The Mathematica kernel only knows the IDs by which to identify these structures to the QuEST environment.

```
ψ
0
```

```
ρ
1
```

```
GetAllQuregs[]
{0, 1}
```

This means we can create, operate on and study states that are too large to fit in Mathematica, or even this machine!

```
InitPlusState @ ψ;
CalcProbOfOutcome[ψ, 5, 1]
0.5
```

```
? InitPlusState
? CalcProbOfOutcome
```

InitPlusState[qureg] sets the qureg to state |+> (and returns the qureg id).

CalcProbOfOutcome[qureg, qubit, outcome]
returns the probability of measuring qubit in the given outcome.

With some overhead, we can view the state with **GetQuregMatrix** (which is initially $\psi = |0\rangle$ and $\rho = |0\rangle\langle 0|$).

```
Dimensions @ GetQuregMatrix[ψ]
{512}
```

```
Dimensions @ GetQuregMatrix[ρ]
{512, 512}
```

The state vectors will live in the QuEST environment until individually destroyed...

```
DestroyQureg[ψ]
DestroyQureg[ρ]
```

or all at once.

```
DestroyAllQuregs[];
```

Specifying gates

Individual gates have syntax $\mathbf{GateName}_{\mathbf{targetQubit}}$ where the **targetQubit** index is subscript (ctrl-minus) and indexes from 0. E.g. $\boldsymbol{H_3}$ represents a Hadamard on the 4th qubit

```
? H
```

H is the Hadamard gate.

Some gates additionally accept parameters in square brackets, e.g. $\mathbf{Ry_2[\phi]}$

```
? Ry
```

Ry[theta] is a rotation of theta around the y–axis of the Bloch sphere.

This can include matrices, e.g. $U_3\left[\begin{pmatrix} 0 & i \\ \mathrm{Exp}[.3\, i] & 0 \end{pmatrix}\right]$ ...

```
? U
```

U[matrix] is a general 1 or 2 qubit unitary gate, enacting the given 2x2 or 4x4 matrix.

and lists of matrices, e.g. $\mathbf{Kraus_2}\left[\left\{\begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1-p} \end{pmatrix}, \begin{pmatrix} 0 & \sqrt{p} \\ 0 & 0 \end{pmatrix}\right\}\right]$ ...

```
? Kraus
```

Kraus[ops] applies a one or two–qubit Kraus map (given as a list of Kraus operators) to a density matrix.

Multiple target qubits are comma separated, or supplied as a list, e.g. $\mathbf{SWAP_{0,3}}$ and $M_{0,1,2,3}$ ...

```
? SWAP
? M
```

SWAP is a 2 qubit gate which swaps the state of two qubits.

M is a destructive measurement gate which measures the indicated qubits in the Z basis.

unless specified as Pauli sequences, e.g. $\mathbf{R}[\phi, X_2\, Y_3\, Z_0]$

```
? R
```

R[theta, paulis] is the unitary Exp[–i θ/2 paulis].

Controlled gates are merely wrapped in $C_{\text{control qubits}}[\,]$, e.g. $C_{1,2}[X_3]$ is a doubly-controlled NOT

$$C_{0,1,2}\left[U_{6,3}\left[\begin{pmatrix} e^{i\frac{\pi}{3}} & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}\right]\right];$$

Some operations like decoherence are only relevant for density matrices (states created with **createDensityQureg**)

```
? Deph
? Depol
? Damp
? Kraus
```

Deph[prob] is a 1 or 2 qubit dephasing with probability prob of error.

Depol[prob] is a 1 or 2 qubit depolarising with probability prob of error.

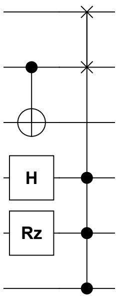
Damp[prob] is 1 qubit amplitude damping with the givern decay probability.

Kraus[ops] applies a one or two–qubit Kraus map (given as a list of Kraus operators) to a density matrix.

Applying circuits

A circuit can be written verbosely as a **list** (to be applied left-to-right) of gates...

```
{H_2, Rz_1[.3], C_4[X_3], C_{0,1,2}[SWAP_{4,5}]};
DrawCircuit[%]
```



or concisely as a direct **product wrapped in Circuit[]** to prevent automatic commutation (or to be reversed, **Operator[]**)

```
Circuit[ H_2 Rz_1[.3] C_4[X_3] C_{0,1,2}[SWAP_{4,5}] ]
```

```
{H_2, Rz_1[0.3], C_4[X_3], C_{0,1,2}[SWAP_{4,5}]}
```

```
Operator[ H_2 Rz_1[.3] C_4[X_3] C_{0,1,2}[SWAP_{4,5}] ]
```

```
{C_{0,1,2}[SWAP_{4,5}], C_4[X_3], Rz_1[0.3], H_2}
```

Circuits can be specified in terms of symbols/parameters, though which must be assigned numerical values before simulation.

$$m1 = \begin{pmatrix} 0 & \mathbb{i} \\ \text{Exp}[.3\,\mathbb{i}] & 0 \end{pmatrix};$$

$$m2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix};$$

u[θ_] := Circuit[

$S_5\, T_7\, Y_8\, Ry_3[\theta]\, C_3[Rz_2[\theta]]\, C_{8,7,6}[Z_5]\, M_0\, Ry_3[\theta]\, M_{1,3,4}\, SWAP_{0,3}\, C_5[SWAP_{4,6}]$

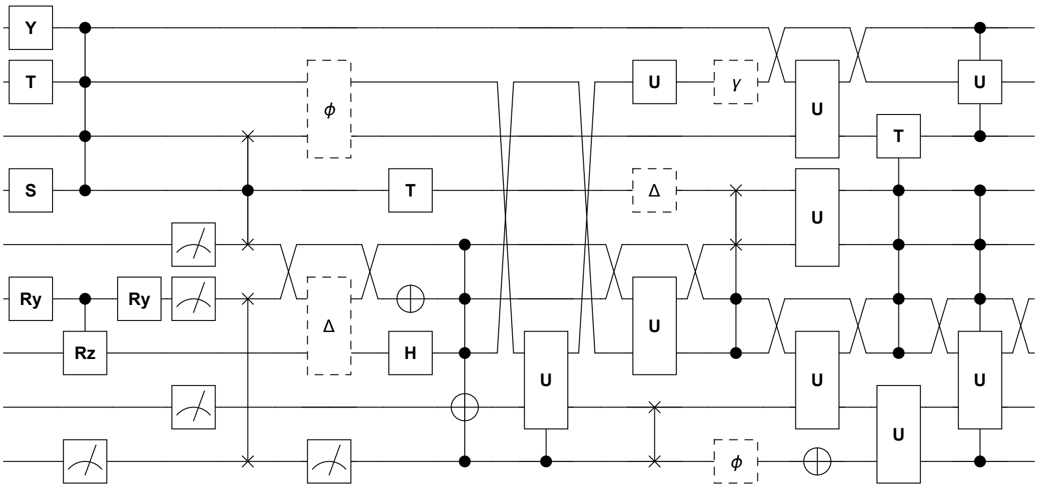
$Depol_{2,4}[\theta/100]\, Deph_{7,6}[\theta/400]\, M_0\, H_2\, X_3\, T_5\, C_{0,2,3,4}[X_1]\, C_0[U_{1,7}[m2]]\, U_{2,4}[m2]$

$U_7[m1]\, SWAP_{0,1}\, Depol_5[\theta/300]\, Deph_0[\theta/200]\, Damp_7[\theta/500]\, C_{2,3}[SWAP_{4,5}]$

$U_{3,1}[m2]\, U_{4,5}[m2]\, U_{6,8}[m2]\, X_0\, U_{0,1}[m2]\, C_{2,3,4,5}[T_6]\, C_{0,2,4,5}[U_{1,3}[m2]]\, C_{6,8}[U_7[m1]]$

];

DrawCircuit @ u[θ]



Circuits can be applied to instantiated quantum registers through **ApplyCircuit**

ψ = CreateQureg[3];

ApplyCircuit[Circuit[$H_0\, X_1\, Ry_2[\pi/3]$], ψ];

GetQuregMatrix[ψ]

```
{0. + 0. i, 0. + 0. i, 0.612372 + 0. i, 0.612372 + 0. i,
 0. + 0. i, 0. + 0. i, 0.353553 + 0. i, 0.353553 + 0. i}
```

? ApplyCircuit

ApplyCircuit[circuit, qureg] modifies qureg by applying the circuit. Returns any measurement outcomes, grouped by M operators and ordered by their order in M.

ApplyCircuit[circuit, inQureg, outQureg] leaves inQureg unchanged, but modifies outQureg to be the result of applying the circuit to inQureg.

ApplyCircuit returns a list of the random measurement outcomes (if any), ordered and grouped by the ordering of **M** in the circuit

```
ApplyCircuit[Circuit[M₀ M₁,₂], ψ]
```

```
{{0}, {1, 0}}
```

Remember these measurements are **destructive**

```
ApplyCircuit[Circuit[M₀,₁,₂], ψ]
```

```
{{0, 1, 0}}
```

Remember that symbols/parameters in the circuit *must* be given numerical values before evaluation

```
ApplyCircuit[Rx₀[ϕ], ψ]
```

ApplyCircuit: Circuit contains non−numerical parameters!

```
$Failed
```

Circuits applied to density matrices are no different

```
ApplyCircuit[u[0], InitPlusState @ CreateDensityQureg[9]]
```

```
{{0}, {1, 0, 1}, {0}}
```

Analysing quantum states

```
DestroyAllQuregs[];
```

Quantum registers can be studied without expensively copying their state vector or density matrix to Mathematica from the QuEST environment.

```
ρ = InitPlusState @ CreateDensityQureg @ numQb;
ApplyCircuit[Depol₀,₁[.1], ρ];
CalcPurity[ρ]
? CalcPurity
```

```
0.848533
```

CalcPurity[qureg] returns the purity of the given density matrix.

```
ψ = InitPlusState @ CreateQureg @ numQb;
CalcFidelity[ρ, ψ]
? CalcFidelity
```

```
0.92
```

CalcFidelity[qureg1, qureg2] returns the fidelity between the given states.

```
CalcProbOfOutcome[ρ, 0, 0]
ApplyCircuit[ Damp0[.1], ρ];
CalcProbOfOutcome[ρ, 0, 0]
? CalcProbOfOutcome
```

```
0.5
```

```
0.55
```

CalcProbOfOutcome[qureg, qubit, outcome]
returns the probability of measuring qubit in the given outcome.

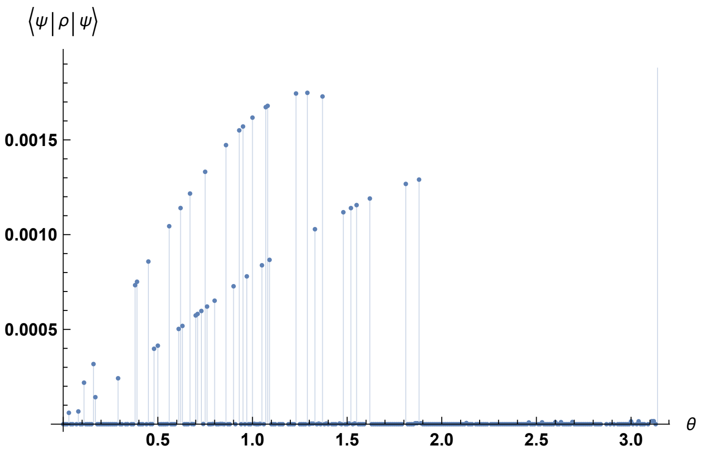
This allows us to express complicated calculations succinctly, and evaluate them quickly.

```
ApplyCircuit[u[0], InitPlusState @ ψ];

params = Range[0, π, .01];
fids = Table[
        ApplyCircuit[u[θ], InitPlusState @ ρ];
        CalcFidelity[ρ, ψ],
        {θ, params}
  ];
```

Here we've calculated how smoothly varying the noise level ***θ*** in our complicated **u[*θ*]** circuit (drawn here) affects the fidelity with its initial |+⟩⟨+| state. Note the results here are *random* since our circuit contains projective measurement gates.

```
ListPlot[
      Transpose[{params, fids}],
      AxesLabel → {"θ", "⟨ψ|ρ|ψ⟩"},
      Filling → Bottom
]
```



Finally, we free the state-vectors from the QuEST environment and disconnect from **quest_link** (killing the process).

```
DestroyAllQuregs[];
DestroyQuESTEnv[env];
```