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.

In[1]:=
```
Import["https://quest.qtechtheory.org/QuEST.m"]
```

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

In[2]:=
```
? CreateRemoteQuESTEnv
? CreateLocalQuESTEnv
? CreateDownloadedQuESTEnv
```

CreateRemoteQuESTEnv[id] connects to the remote Igor server (on port 50000+id and 50100+id) 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[] connects to a local Mathematica backend, running single–CPU QuEST. This requires a compatible 'quest_link' executable is in the same directory as the notebook. This should be called once. The QuEST function defintions can be cleared with DestroyQuESTEnv[link].

CreateDownloadedQuESTEnv[] downloads a MacOS–CPU–QuEST server from quest.qtechtheory.org, gives it permission to run then locally connects to it. This should be called once. The QuEST function defintions can be cleared with DestroyQuESTEnv[link].

Here, we'll automatically download the MacOS QuEST executable and locally connect.

In[5]:=
```
env = CreateDownloadedQuESTEnv[];
```

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

In[6]:=
```
? QuEST`*
```

▼ QuEST`

AddWeightedStates	CreateLocalQuESTEnv	Kraus
ApplyCircuit	CreateQureg	M
ApplyOneQubitDampingErr-or	CreateRemoteQuESTEnv	Operator
ApplyOneQubitDephaseErr-or	Damp	P
ApplyOneQubitDepolariseE-rror	Deph	PackageExport
ApplyTwoQubitDephaseErr-or	Depol	R
ApplyTwoQubitDepolariseE-rror	DestroyAllQuregs	Rx
CalcExpectedValue	DestroyQuESTEnv	Ry
CalcFidelity	DestroyQureg	Rz
CalcHilbertSchmidtDistance	DrawCircuit	S
CalcInnerProduct	GetAllQuregs	SetMatrix
CalcProbOfOutcome	GetMatrix	SetWeightedStates
CalcPurity	H	SWAP
Circuit	InitClassicalState	T
CloneQureg	InitPlusState	U
CollapseToOutcome	InitPureState	X
CreateDensityQureg	InitStateFromAmps	Y
CreateDownloadedQuESTE-nv	InitZeroState	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.

In[7]:=
```
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.

In[10]:=
```
ψ
```

Out[10]= 0

In[11]:=
```
ρ
```
Out[11]=
```
1
```

In[12]:=
```
GetAllQuregs[]
```
Out[12]=
```
{0, 1}
```

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

In[13]:=
```
InitPlusState @ ψ;
CalcProbOfOutcome[ψ, 5, 1]
```
Out[14]=
```
0.5
```

In[15]:=
```
? 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 **GetMatrix** (which is initially $\psi = |0\rangle$ and $\rho = |0\rangle\langle 0|$).

In[17]:=
```
Dimensions @ GetMatrix[ψ]
```
Out[17]=
```
{512}
```

In[18]:=
```
Dimensions @ GetMatrix[ρ]
```
Out[18]=
```
{512, 512}
```

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

In[19]:=
```
DestroyQureg[ψ]
DestroyQureg[ρ]
```

or all at once.

In[21]:=
```
DestroyAllQuregs[];
```

Specifying gates

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

In[22]:=
```
? H
```

H is the Hadamard gate.

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

In[23]:=
```
? Ry
```

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

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

In[24]:=
```
? 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. $\text{Kraus}_2\left[\left\{\begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1-p} \end{pmatrix}, \begin{pmatrix} 0 & \sqrt{p} \\ 0 & 0 \end{pmatrix}\right\}\right]$ ...

In[25]:=
```
? 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 $M_{0,1,2,3}$ ...

In[26]:=
```
? SWAP
? M
```

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

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

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

In[28]:=
```
? R
```

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

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

In[29]:= $C_{0,1,2}\left[U_{6,3}\left[\begin{pmatrix} e^{i\frac{\pi}{3}} & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}\right]\right];$

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

In[30]:=
```
? 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.

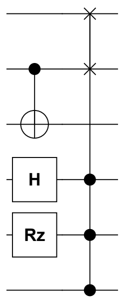
Kraus[ops] applies a one or two–qubit Kraus map (given as a list of Kraus operators) to a density matrix.

Applying circuits

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

In[34]:= $\{H_2,\ Rz_1[.3],\ C_4[X_3],\ C_{0,1,2}[SWAP_{4,5}]\};$
DrawCircuit[%]

Out[35]=



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

In[36]:= Circuit[$H_2\ Rz_1[.3]\ C_4[X_3]\ C_{0,1,2}[SWAP_{4,5}]$]

Out[36]= $\{H_2,\ Rz_1[0.3],\ C_4[X_3],\ C_{0,1,2}[SWAP_{4,5}]\}$

In[37]:= Operator[$H_2\ Rz_1[.3]\ C_4[X_3]\ C_{0,1,2}[SWAP_{4,5}]$]

Out[37]= $\{C_{0,1,2}[SWAP_{4,5}],\ C_4[X_3],\ Rz_1[0.3],\ H_2\}$

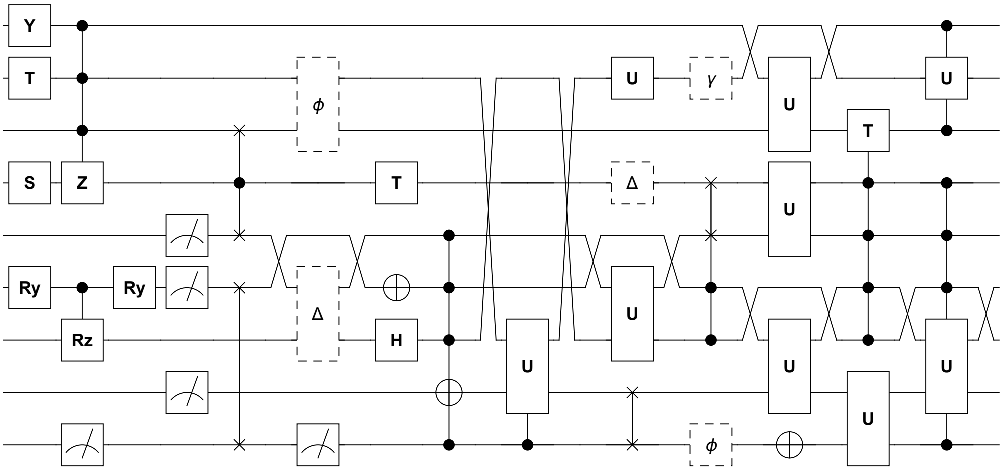
Circuits can be specified in terms of symbols/parameters, though which must be assigned numerical values before simulation.

In[38]:=
```
m1 = ( 0          i
       Exp[.3 i]  0 );
m2 = ( 1 0 0 0
       0 0 1 0
       0 1 0 0
       0 0 0 1 );

u[θ_] := Circuit[
    S5 T7 Y8 Ry3[θ] C3[Rz2[θ]] C8,7,6[Z5] M0 Ry3[θ] M1,3,4 SWAP0,3 C5[SWAP4,6]
     Depol2,4[θ/100] Deph7,6[θ/400] M0 H2 X3 T5 C0,2,3,4[X1] C0[U1,7[m2]] U2,4[m2]
     U7[m1] SWAP0,1 Depol5[θ/300] Deph0[θ/200]  Damp7[θ/500] C2,3[SWAP4,5]
     U3,1[m2] U4,5[m2] U6,8[m2] X0 U0,1[m2] C2,3,4,5[T6] C0,2,4,5[U1,3[m2]] C6,8[U7[m1]]
  ];

DrawCircuit @ u[θ]
```

Out[41]=



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

In[42]:=
```
ψ = CreateQureg[3];
ApplyCircuit[ Circuit[H0 X1 Ry2[π/3]], ψ];
GetMatrix[ψ]
```

Out[44]=
```
{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}
```

In[45]:=
```
? 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

In[46]:=
```
ApplyCircuit[ Circuit[ M0 M1,2], ψ]
```
Out[46]= {{1}, {1, 0}}

Remember these measurements are **destructive**

In[47]:=
```
ApplyCircuit[ Circuit[ M0,1,2], ψ]
```
Out[47]= {{1, 1, 0}}

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

In[48]:=
```
ApplyCircuit[ Rx0[ϕ], ψ]
```
» Error: Circuit contains non-numerical parameters!

Out[48]= $Failed

Circuits applied to density matrices are no different

In[49]:=
```
ApplyCircuit[u[0], InitPlusState @ CreateDensityQureg[9]]
```
Out[49]= {{1}, {0, 0, 1}, {0}}

Analysing quantum states

In[50]:=
```
DestroyAllQuregs[];
```

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

In[51]:=
```
ρ = InitPlusState @ CreateDensityQureg @ numQb;
ApplyCircuit[ Depol0,1[.1], ρ];
CalcPurity[ρ]
? CalcPurity
```
Out[53]= 0.848533

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

In[55]:=
```
ψ = InitPlusState @ CreateQureg @ numQb;
CalcFidelity[ρ, ψ]
? CalcFidelity
```
Out[56]= 0.92

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

In[58]:=
```
CalcProbOfOutcome[ρ, 0, 0]
ApplyCircuit[ Damp₀[.1], ρ];
CalcProbOfOutcome[ρ, 0, 0]
? CalcProbOfOutcome
```

Out[58]= 0.5

Out[60]= 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.

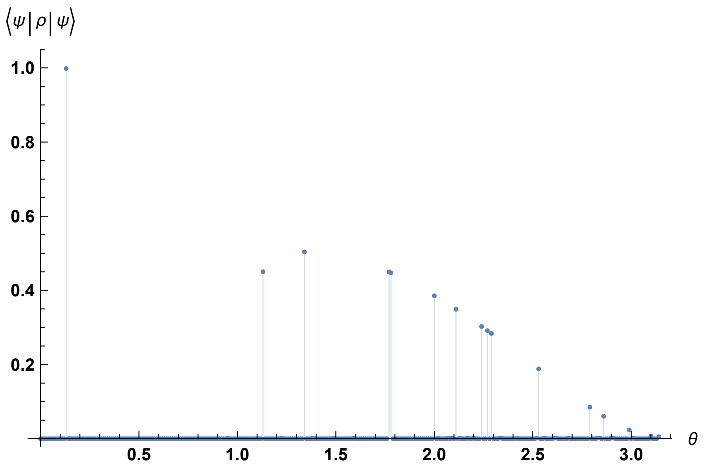
In[62]:=
```
ApplyCircuit[u[0], InitPlusState @ ψ];

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.

In[65]:=
```
ListPlot[
     Transpose[{params, fids}],
     AxesLabel → {"θ", "⟨ψ|ρ|ψ⟩"},
     Filling → Bottom
]
```

Out[65]=



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

In[66]:=
```
DestroyAllQuregs[];
DestroyQuESTEnv[env];
```