Release Summary

V0.12

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

This *major* release significantly extends QuESTlink's capabilities for efficient and convenient simulation of quantum variational algorithms, as well as generally improving quality-of-life. It includes asymptotically improved functions to simulate quantum gradient descent and quantum natural gradient (even a noise aware version!), the relaxing of previous ansatz constraints, functions to generate Pauli strings and ansatz circuits, and improved error messages. This release involved a complete refactor of QuESTlink's C++ backend which is now more modular, defensively-designed, and enables backend circuit-level optimisations and functions. It also adds native MacoS ARM (M1) support.

New features

GetKnownCircuit

Three new ansatz circuits have been added to **GetKnownCircuit**.

```
? GetKnownCircuit
```

Symbol

GetKnownCircuit["QFT", qubits]

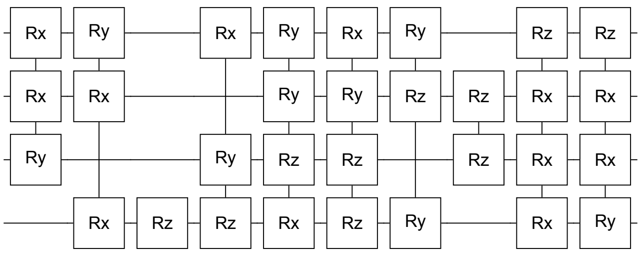
GetKnownCircuit["Trotter", hamil, order, reps, time]
(https://arxiv.org/pdf/math–ph/0506007.pdf)

GetKnownCircuit["HardwareEfficientAnsatz", reps, paramSymbol, qubits]
(https://arxiv.org/pdf/1704.05018.pdf)

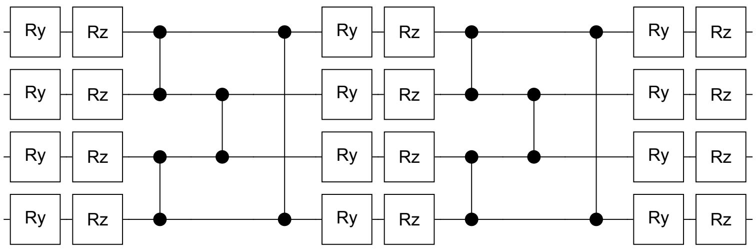
GetKnownCircuit["TrotterAnsatz", hamil, order, reps, paramSymbol]
(https://arxiv.org/pdf/1507.08969.pdf)

GetKnownCircuit["LowDepthAnsatz", reps, paramSymbol, qubits]
(https://arxiv.org/pdf/1801.01053.pdf)

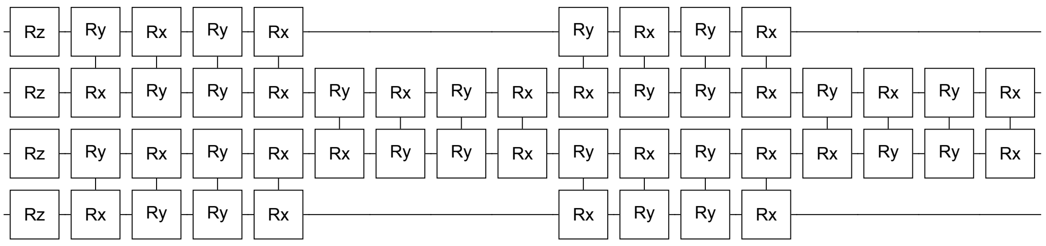
```
GetKnownCircuit["TrotterAnsatz",
  GetRandomPauliString[4, 10], 1, 1, x] // DrawCircuit
```



```
GetKnownCircuit["HardwareEfficientAnsatz", 2, x, 4] // DrawCircuit
```



```
GetKnownCircuit["LowDepthAnsatz", 2, x, 4] // DrawCircuit
```


Rz Ry Rx Ry Rx Ry Rx Ry Rx
Rz Rx Ry Ry Rx Ry Rx Ry Rx Rx Ry Ry Rx Ry Rx Ry Rx
Rz Ry Rx Ry Rx Rx Ry Ry Rx Ry Rx Ry Rx Rx Ry Ry Rx
Rz Rx Ry Ry Rx Rx Ry Ry Rx

ApplyCircuitDerivs

ApplyCircuitDerivs (replacing the previously named **CalcQuregDerivs**) can now accept *any* continuously parametrized circuit *or* channel!

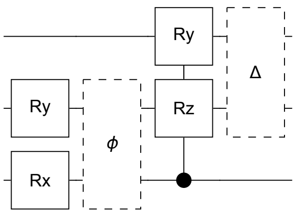
```
? ApplyCircuitDerivs
```

Symbol

ApplyCircuitDerivs[inQureg, circuit, varVals, outQuregs] modifies
outQuregs to be the result of applying the derivatives (with respect to variables
in varVals) of the given symbolic circuit to inQureg (which remains unmodified).
- varVals is a list {symbol -> value, ...} of all variables present in the circuit parameters.
- outQuregs is a list of quregs to set to the respective
derivative of circuit upon inQureg, according to the order of vars.
- Variable repetition, multi–parameter gates, variable–dependent element–wise
matrices, variable–dependent channels, and operators whose parameters are
(numerically evaluable) functions of variables are all permitted within the circuit.
In effect, every continuously–parameterised circuit or channel is permitted.

ApplyCircuitDerivs[inQureg, circuit, varVals, outQuregs, workQuregs] use the given persistent
workspace quregs to avoid tediously creating and destroying any internal quregs, for a speedup.
For convenience, any number of workspaces can be passed, but only the first is needed and used.

```
inρ = CreateDensityQureg[3] // InitPlusState;
dρ = CreateDensityQuregs[3, 3];
u = Circuit[ Rx_0[a] Ry_1[b^2 - a] Deph_0,1[c a] C_0[R[c / a, Z_1 Y_2]] Depol_1,2[a - b + c]];

DrawCircuit[u]
```



```
ApplyCircuitDerivs[inρ, u, {a → .3, b → .4, c → .5}, dρ];
GetQuregMatrix @ dρ⟦1⟧ // Chop // MatrixForm
```

```
 0.0523597    0.39998    -0.160934   -0.216647  -0.0809736  -0.156116   -0.160934
  0.39998    0.0895311    0.233672    0.329359    0.26444     0.459375    0.233672
 -0.160934    0.233672  -0.0523597  -0.123092   -0.160934   -0.216647   -0.185693
 -0.216647    0.329359   -0.123092   -0.204406   -0.216647   -0.290937   -0.258633
-0.0809736    0.26444    -0.160934   -0.216647   0.0523597  -0.0205754  -0.160934
 -0.156116    0.459375   -0.216647   -0.290937  -0.0205754  0.0151884   -0.216647
 -0.160934    0.233672   -0.185693   -0.258633   -0.160934   -0.216647  -0.0523597
  0.233672   -0.0309304   0.206428    0.358791    0.233672    0.329359    0.341968
```

CalcExpecPauliStringDerivs

CalcExpecPauliStringDerivs rapidly computes the gradient of a variational observable *without* having to create dedicated statevectors like above, using a novel algorithm (https://arxiv.org/abs/2009.02823). This returns the quantity:

$$\nabla_{\theta_i} \left\langle \text{in}\psi \right| \hat{u}[\vec{\theta}]^\dagger \; \hat{h} \; \hat{u}[\vec{\theta}] \left| \text{in}\psi \right\rangle$$

It can furthermore compute the gradient of *noisy channels* upon *density matrices* (using another novel algorithm, see Ch5.7 of https://pdfhost.io/v/khdW2GJgP_Thesis_Tyson_Jones). In this instance, it returns:

$$\nabla_{\theta_i} \text{Trace}\left[\hat{h} \;\; \text{channel}_{\vec{\theta}} \{\text{in}\rho\}\right]$$

These quantities appear in many quantum variational algorithms like quantum gradient descent, quantum natural gradient and variational imaginary time evolution. Parameters can be repeated between gates, gates can be multi-parameterised, and feature arbitrary (albeit continuous) functions of the variational parameters.

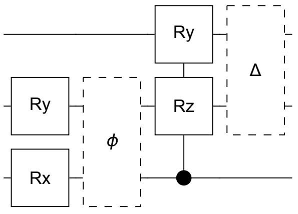
```
? CalcExpecPauliStringDerivs
```

Symbol

CalcExpecPauliStringDerivs[inQureg, circuit, varVals, pauliString] returns the gradient vector of the pauliString expected values, as produced by the derivatives of the circuit (with respect to varVals, {var -> value}) acting upon the given initial state (inQureg).
CalcExpecPauliStringDerivs[inQureg, circuit, varVals, pauliQureg] accepts a Qureg pre-initialised as a pauli string via SetQuregToPauliString[] to speedup density-matrix simulation.
CalcExpecPauliStringDerivs[inQureg, circuit, varVals, pauliStringOrQureg, workQuregs] uses the given persistent workspaces (workQuregs) in lieu of creating them internally, and should be used for optimum performance. At most four workQuregs are needed.

- Variable repetition, multi-parameter gates, variable-dependent element-wise matrices, variable-dependent channels, and operators whose parameters are (numerically evaluable) functions of variables are all permitted.
- All operators must be invertible, trace-preserving and deterministic, else an error is thrown.
- This function runs asymptotically faster than ApplyCircuitDerivs[] and requires only a fixed memory overhead.

```
DrawCircuit[u]
```



```
h = .5 X0 Y1 + Z0 Z1 + Z0 + 2 Z2 - 4 Z1 - .3 X1 Z2 + Y0 Y1 Y2 + .2 X0 X2 X1;
```

$h = .5 X_0 Y_1 + Z_0 Z_1 + Z_0 + 2 Z_2 - 4 Z_1 - .3 X_1 Z_2 + Y_0 Y_1 Y_2 + .2 X_0 X_2 X_1;$

```
CalcExpecPauliStringDerivs[inρ, u, {a → .4, b → .2, c → .5}, h]
{0.650992, -1.26729, 1.6246}
```

Let’s compare the first element to its analytic form:

```
∂a Tr[CalcPauliExpressionMatrix[h] .
    Transpose @ ArrayReshape[CalcCircuitMatrix[u] . Flatten @
        Transpose @ GetQuregMatrix[inρ], {2^3, 2^3}]] /. {a → .4, b → .2, c → .5}
0.650992 + 0. i
```

CalcMetricTensor

CalcMetricTensor rapidly computes the metric tensor of the given parameterised circuit or channel. If a pure circuit is given, this function returns the quantum geometric tensor like that appearing in quantum natural gradient and variational imaginary time evolution:

$$G_{ij} = \left\langle \frac{\partial\psi[\vec{\theta}]}{\partial\theta_i} \middle| \middle| \frac{\partial\psi[\vec{\theta}]}{\partial\theta_j} \right\rangle - \left\langle \frac{\partial\psi[\vec{\theta}]}{\partial\theta_i} \middle| \middle| \psi[\vec{\theta}] \right\rangle \left\langle \psi[\vec{\theta}] \middle| \middle| \frac{\partial\psi[\vec{\theta}]}{\partial\theta_j} \right\rangle \qquad \text{where } \left| \psi[\vec{\theta}] \right\rangle = \hat{u}[\vec{\theta}] \left| \text{in}\psi \right\rangle .$$

When a noisy channel is given, this function returns the Hilbert-Schmidt derivative metric

$$M_{ij} = \frac{1}{2} \text{Tr}\left[\frac{\partial\rho[\vec{\theta}]}{\partial\theta_i} \frac{\partial\rho[\vec{\theta}]}{\partial\theta_j} \right] \qquad \text{where} \quad \rho[\vec{\theta}] = \text{channel}_{\vec{\theta}}\{\text{in}\rho\} ,$$

which in many settings well approximates the quantum Fisher information matrix and can replace it in quantum natural gradient for a superior noise-aware minimisation.

```
? CalcMetricTensor
```

Symbol

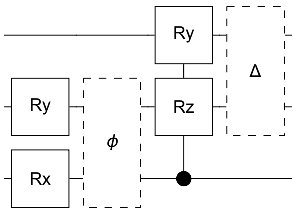
CalcMetricTensor [inQureg, circuit, varVals] returns the natural gradient metric tensor, capturing the circuit derivatives (produced from initial state inQureg) with respect to varVals, specified with values {var -> value, ...}.

CalcMetricTensor [inQureg, circuit, varVals, workQuregs] uses the given persistent workspace quregs (workQuregs) in lieu of creating them internally, and should be used for optimum performance. At most four workQuregs are needed.

- For state–vectors and pure circuits, this returns the quantum geometric tensor, which relates to the Fubini–Study metric, the classical Fisher information matrix, and the variational imaginary–time Li tensor with Berry connections.
- For density–matrices and noisy channels, this function returns the Hilbert–Schmidt derivative metric, which well approximates the quantum Fisher information matrix, though is a more experimentally relevant minimisation metric (https://arxiv.org/abs/1912.08660).
- Variable repetition, multi–parameter gates, variable–dependent element–wise matrices, variable–dependent channels, and operators whose parameters are (numerically evaluable) functions of variables are all permitted.
- All operators must be invertible, trace–preserving and deterministic, else an error is thrown.
- This function runs asymptotically faster than ApplyCircuitDerivs[] and requires only a fixed memory overhead.

```
u
DrawCircuit[u]
```

$\left\{Rx_0[a],\ Ry_1\left[-a+b^2\right],\ Deph_{0,1}[a\,c],\ C_0\left[R\left[\frac{c}{a},\ Y_2\,Z_1\right]\right],\ Depol_{1,2}[a-b+c]\right\}$

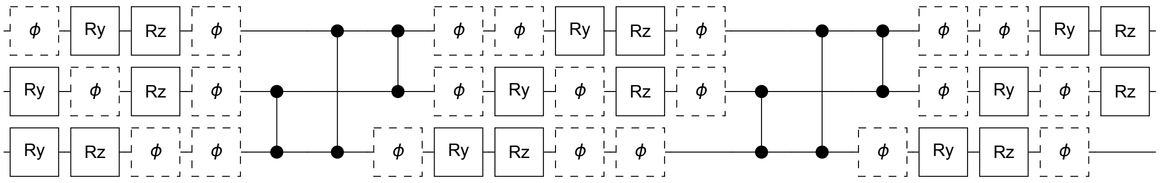


```
CalcMetricTensor[inρ, u, {a → .1, b → .2, c → .3}] // Chop // MatrixForm
```

$$\begin{pmatrix} 142.846 & -1.39051 & -45.6214 \\ -1.39051 & 0.974851 & -0.974446 \\ -45.6214 & -0.974446 & 16.6879 \end{pmatrix}$$

These functions make variational simulation trivial. We here demonstrate noise-aware quantum natural gradient, whereby dephasing noise strength is correlated with the prior gate parameter.

```
u = GetKnownCircuit["HardwareEfficientAnsatz", 2, θ, 3];
i = 1;
u = Riffle[Partition[u, 3], ConstantArray[
      Table[Deph_q[.2 + 10^-4 θ[i++]], {q, 0, 2}], Length[u]]] // Flatten;
DrawCircuit[u]
```



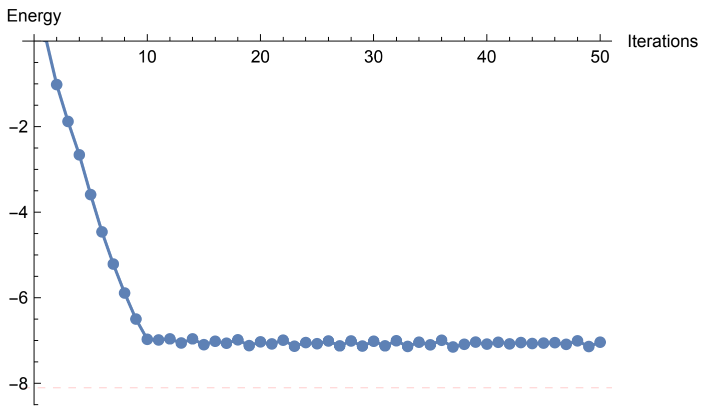
```
{ρ, inρ, wρ} = CreateDensityQuregs[3, 3];
InitPlusState[inρ];
```

If one foregoes measuring the energy, then each iteration of minimisation requires only *three* function calls!

```
Δt = .01;
nt = 50;
θVals = Table[θ[i] → RandomReal[], {i, u⟦-1, 1, 1⟧}];

eVals = Table[
       v = CalcExpecPauliStringDerivs[inρ, u, θVals, h];
       m = CalcMetricTensor[inρ, u, θVals] // Chop;

       Δθ = Fit[{m, v}, FitRegularization → {"Tikhonov", 10^-4}];
       θVals⟦All, 2⟧ += - Δθ Δt ;

       ApplyCircuit[CloneQureg[ρ, inρ], u /. θVals];
       CalcExpecPauliString[ρ, h, wρ],

       nt
  ];
```

Here is how the energy evolved

```
eMin = CalcPauliStringMinEigVal[h];

ListLinePlot[eVals,
     AxesLabel → {"Iterations", "Energy"},
     PlotRange → {0, 1.05 eMin},
     PlotMarkers → Automatic,
     GridLines → {{}, {{eMin, Directive[Red, Dashed]}}}
]
```



Error reporting

Functions which process circuits can now precisely report the erroneous operator.

```
CalcCircuitMatrix[J₀]
```

••• CalcCircuitMatrix: Circuit contained an unrecognised or unsupported gate: J_0

```
$Failed
```

```
ApplyCircuit[CreateQureg[2], Circuit[X₃]]
```

••• ApplyCircuit: Cannot simulate X_3. Invalid target qubit. Must be >=0 and <numQubits. The qureg (id 7) has been restored to its prior state.

```
$Failed
```

ApplyCircuit$\left[\text{CreateQureg[2]}, U_0\left[\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}\right]\right]$

••• ApplyCircuit: Cannot simulate $U_0\left[\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}\right]$. Matrix is not unitary. The qureg (id 8) has been restored to its prior state.

```
$Failed
```

GetRandomPauliString

```
? GetRandomPauliString
```

Symbol

GetRandomPauliString[numQubits, numTerms , {minCoeff, maxCoeff}] generates a random Pauli string with unique Pauli tensors.
GetRandomPauliString[numQubits, All, {minCoeff, maxCoeff}] will generate all 4^numQubits unique Pauli tensors.
GetRandomPauliString[numQubits, {minCoeff, maxCoeff}] will generate 4 numQubits^4 unique terms / Pauli tensors, unless this exceeds the maximum of 4^numQubits.
GetRandomPauliString[numQubits] will generate random coefficients in [–1, 1].
All combinations of optional arguments are possible.

```
GetRandomPauliString[3, 10]
```

$0.940567\, Y_0 + 0.0133663\, X_0\, Y_2 + 0.407032\, X_1\, Y_2 - 0.240966\, X_1\, Y_0\, Y_2 - 0.458556\, X_0\, Y_2\, Z_1 + 0.600443\, Z_0\, Z_1 + 0.502558\, Z_2 - 0.756216\, X_0\, Z_2 - 0.077569\, X_1\, Y_0\, Z_2 - 0.341494\, Z_0\, Z_1\, Z_2$

```
GetRandomPauliString[2, 10^5, {0, 1}]
```

GetRandomPauliString : More terms were requested than there are unique Pauli tensors. Hide this warning with Quiet[].

$0.508625\, Id_0\, Id_1 + 0.33438\, X_0 + 0.488285\, X_1 + 0.554076\, X_0\, X_1 + 0.663799\, Y_0 + 0.846016\, X_1\, Y_0 + 0.669192\, Y_1 + 0.868138\, X_0\, Y_1 + 0.0866329\, Y_0\, Y_1 + 0.731169\, Z_0 + 0.199964\, X_1\, Z_0 + 0.0695654\, Y_1\, Z_0 + 0.365943\, Z_1 + 0.762892\, X_0\, Z_1 + 0.686691\, Y_0\, Z_1 + 0.145336\, Z_0\, Z_1$

CalcPauliStringMinEigVal

```
? CalcPauliStringMinEigVal
```

Symbol

CalcPauliStringMinEigVal[pauliString] returns the ground–state energy of the given real–weighted sum of Pauli tensors.
CalcPauliStringMinEigVal[pauliString, MaxIterations –> n] specifies to use at most n iterations in the invoked Arnaldi/Lanczos's method

```
CalcPauliStringMinEigVal @ GetRandomPauliString[15, 10]
```

```
-2.9193
```

By using sparse matrices, this function will require exponentially less memory and be be significantly faster than alternative methods (like **Min @ Eigenvalues @ CalcPauliStringMatrix[h]**) when run on large systems (e.g. 15 qubits) with few terms.

SetQuregToPauliString

```
? SetQuregToPauliString
```

Symbol

SetQuregToPauliString[qureg, pauliString] overwrites the given density matrix to become a dense matrix representation of the given pauli string.
The state is likely no longer a valid density matrix but is useful as a persistent Z-basis representation of the pauli string, to be used in functions like CalcDensityInnerProduct[] and CalcExpecPauliStringDerivs[].

```
h = GetRandomPauliString[3, 3]
CalcPauliExpressionMatrix[h] // Chop // MatrixForm
```

$-0.332144\, X_1\, X_2 + 0.732511\, X_0\, Y_1\, Z_2 + 0.136264\, X_1\, Z_0\, Z_2$

```
        0                  0              0.136264     0. - 0.732511 i        0              0
        0                  0          0. - 0.732511 i    -0.136264            0              0
    0.136264       0. + 0.732511 i         0                0            -0.332144           0
0. + 0.732511 i       -0.136264            0                0                0           -0.33
        0                  0           -0.332144            0                0              0
        0                  0               0            -0.332144            0              0
   -0.332144               0               0                0            -0.136264     0. - 0.7
        0              -0.332144           0                0         0. - 0.732511 i    0.13
```

```
ρ = CreateDensityQureg[3];
SetQuregToPauliString[ρ, h];
GetQuregMatrix[ρ] // Chop // MatrixForm
```

```
        0                  0              0.136264     0. - 0.732511 i        0              0
        0                  0          0. - 0.732511 i    -0.136264            0              0
    0.136264       0. + 0.732511 i         0                0            -0.332144           0
0. + 0.732511 i       -0.136264            0                0                0           -0.33
        0                  0           -0.332144            0                0              0
        0                  0               0            -0.332144            0              0
   -0.332144               0               0                0            -0.136264     0. - 0.7
        0              -0.332144           0                0         0. - 0.732511 i    0.13
```

SampleClassicalShadow

SampleClassicalShadow encodes a density matrix into a number of sampled Pauli product outcomes, as per this paper.

```
? SampleClassicalShadow
```

Symbol

SampleClassicalShadow[qureg, numSamples] returns a
sequence of pseudorandom measurement bases (X, Y and Z) and their
outcomes (as bits) when performed on all qubits of the given input state.

• The output has structure { {bases, outcomes}, ...} where bases
is a list of Pauli bases (encoded as 1=X, 2=Y, 3=Z) specified per-qubit,
and outcomes are the corresponding classical qubit outcomes (0 or 1).

```
ρ = InitPlusState @ CreateDensityQureg[3];
ApplyCircuit[ρ, u /. {a → .4, b → .2, c → .5}];

SampleClassicalShadow[ρ, 5] // Column
```

ApplyCircuit: Circuit contains non-numerical or non-real parameters!

```
{{2, 1, 2}, {0, 0, 1}}
{{2, 1, 2}, {1, 0, 0}}
{{2, 1, 1}, {1, 0, 0}}
{{3, 2, 2}, {1, 0, 0}}
{{3, 3, 2}, {1, 0, 0}}
```

New gates

```
? Fac
```

Symbol

Fac[scalar] is a non-physical operator which multiplies the given complex
scalar onto every amplitude of the quantum state. This is directly multiplied
onto state-vectors and density-matrices, and may break state normalisation.

```
? UNonNorm
```

Symbol

UNonNorm[matr] is treated like a general unitary gate U, but with relaxed normalisation conditions
on the matrix. This is distinct to gate Matr, which will be internally assumed non-unitary.

```
? Matr
```

Symbol

Matr[matrix] is an arbitrary operator with any number of target qubits, specified as a completely general (even non–unitary) square complex matrix. Unlike UNonNorm, the given matrix is not internally assumed unitary. It is hence only left–multiplied onto density matrices.

Changes

CalcDensityInnerProduct(s)

CalcDensityInnerProduct(**s**) now return *complex* scalars, capturing the full Hilbert-Schmidt scalar product when the input density matrices are not normalised.

```
? CalcDensityInnerProduct
```

Symbol

CalcDensityInnerProduct[qureg1, qureg2] returns the the Hilbert schmidt scalar product between two given density matrices. If both quregs are valid/normalised, the result will be a real scalar, though may have a tiny non–zero imaginary component due to numerical imprecision. If either qureg is not a valid density matrix, the result may be a complex scalar.

```
{a, b} = CreateDensityQuregs[3, 2];
SetQuregMatrix[a, RandomVariate @ CircularSymplecticMatrixDistribution @ 4];

CalcDensityInnerProduct[a, b]
-0.190231 + 0.393956 i
```

Mix* deprecated

All explicit mixing functions like **MixDamping** have been deprecated, since they are more concisely invoked with **ApplyCircuit**.

```
? MixTwoQubitDephasing
```

Symbol

This function is deprecated. Please instead use ApplyCircuit with gate Deph.

These deprecated functions are still callable for backwards compatibility

```
MixTwoQubitDepolarising[b, 0, 1, .5];
CalcPurity[b]
```

```
ApplyCircuit: The function MixTwoQubitDepolarising[] is deprecated, though has still been performed. In future,
    please use ApplyCircuit[] with the Depol[] gate instead, or temporarily hide this message using
    Quiet[].
```

```
0.413333
```

PauliSum -> PauliString

Any reference to **PauliSum** in the QuESTlink API has been renamed to **PauliString**.

```
? CalcExpecPauliSum
```

```
Symbol
This function is deprecated. Please instead use CalcExpecPauliString.
```

These deprecated functions are still callable for backwards compatibility

```
CalcExpecPauliSum[a, GetRandomPauliString[3], b]
```

```
CalcExpecPauliString: The function CalcExpecPauliSum[] is deprecated. Use CalcExpecPauliString[] or temporarily
    hide this message using Quiet[].
```

```
-2.61389
```

CalcPauliExpressionMatrix

CalcPauliExpressionMatrix now returns a *sparse* matrix for more efficient handling of large operators, especially symbolic ones with few terms.

```
CalcPauliExpressionMatrix @ GetRandomPauliString[10, 10]
```

```
SparseArray[ Specified elements: 10 240
             Dimensions: {1024, 1024} ]
```

CalcPauliStringMatrix remains useful for (significantly) faster parallel handling of numerical real-coefficient operators.

AssertValidChannels

By default, functions like **CalcCircuitMatrix** will now assert that the given decoherence channels are completely-positive and trace-preserving, and infer assumptions about their parameters which are then used to simplify subsequent expressions. This behaviour can be disabled with **AssertValidChannels -> False**.

```
? AssertValidChannels
```

Symbol

Optional argument to CalcCircuitMatrix and GetCircuitSuperoperator
(default True), specifying whether to simplify their outputs by asserting that all
channels therein are completely–positive and trace–preserving. For example,
this asserts that the argument to a damping channel lies between 0 and 1.

```
CalcCircuitMatrix[Depol_0[x]] // MatrixForm
```

$$\begin{pmatrix} 1-\frac{2x}{3} & 0 & 0 & \frac{2x}{3} \\ 0 & 1-\frac{4x}{3} & 0 & 0 \\ 0 & 0 & 1-\frac{4x}{3} & 0 \\ \frac{2x}{3} & 0 & 0 & 1-\frac{2x}{3} \end{pmatrix}$$

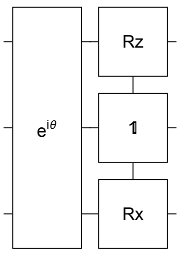
```
CalcCircuitMatrix[Depol_0[x], AssertValidChannels → False] // MatrixForm
```

$$\left(\begin{array}{cc} \sqrt{1-x}\ \text{Conjugate}\left[\sqrt{1-x}\right]+\frac{1}{3}\sqrt{x}\ \text{Conjugate}\left[\sqrt{x}\right] & 0 \\ 0 & \sqrt{1-x}\ \text{Conjugate}\left[\sqrt{1-x}\right]-\frac{1}{3}\sqrt{x}\ \text{Conj} \\ 0 & 0 \\ \frac{2}{3}\sqrt{x}\ \text{Conjugate}\left[\sqrt{x}\right] & 0 \end{array}\right.$$

DrawCircuit

DrawCircuit will now render global phase gates (**G**) and the identity suboperators of Pauli gadgets **R**

```
DrawCircuit @ Circuit[G[θ] × R[π, X_0 Id_1 Z_2] ]
```



UNonNorm and Matr

Previously, **Matr** was a normalisation-relaxed form of the restrictedly-unitary operator **U**. Now, **UNonNorm** fills that role while **Matr** is never assumed nor treated as a unitary. This means that when applied to density-matrices, **Matr** will only ever left-multiply (never being conjugated and right-multiplied like **UNonNorm**).

```
? Matr
? UNonNorm
```

Symbol

Matr[matrix] is an arbitrary operator with any number of target qubits, specified as a completely general (even non–unitary) square complex matrix. Unlike UNonNorm, the given matrix is not internally assumed unitary. It is hence only left–multiplied onto density matrices.

Symbol

UNonNorm[matr] is treated like a general unitary gate U, but with relaxed normalisation conditions on the matrix. This is distinct to gate Matr, which will be internally assumed non–unitary.

CalcQuregDerivs -> ApplyCircuitDerivs

CalcQuregDerivs has been renamed to **ApplyCircuitDerivs**, along with its significant extensions as per above. Note too that the order of the arguments changed.

```
? CalcQuregDerivs
```

Symbol

This function is deprecated. Please instead use ApplyCircuitDerivs.