GaussSynth.m -- The Two-Qubit Clifford + CS Circuit Synthesis Package

Written and Maintained by Andrew Glaudell

The GaussSynth.m package is a package for quantum compiling on two qubits using the Clifford group and the Controlled-Phase gate CS. The circuits which are exactly expressible over this gate set constitute every 4x4 unitary matrix which can be written as a matrix of Gaussian integers divided by some non-negative integer power of 2^(1/2) -- hence the package name. In this package, we supply a number of functions for performing quantum circuit synthesis on this gate set, both in the exact and approximate case. The algorithms in this package are based off of the work of Andrew Glaudell, Julien Ross, Matthew Amy, and Jake Taylor, and for details related to how these algorithms were developed, I suggest reading the articles [1-3] in the sources section below.

Package Details

Package Version: 1.0

Written for Mathematica Version: 12.0

History:
1.0 - Initial version, completed 11/4/2019

Keywords: Quantum Compiling, Quantum Circuit Synthesis, Clifford Group, Controlled Phase Gate, Normal Forms, Exact Synthesis, Approximate Synthesis

Sources:
[1] Amy, Matthew, Andrew N. Glaudell, and Neil J. Ross. "Number-theoretic characterizations of some restricted clifford+ t circuits." Quantum 4 (2020): 252.
[2] Glaudell, Andrew N., Neil J. Ross, and Jacob M. Taylor. "Optimal Two-Qubit Circuits for Universal Fault-Tolerant Quantum Computation." arXiv preprint arXiv:2001.05997 (2020). (under review)
[3] Glaudell, Andrew Noble. Quantum Compiling Methods for Fault-Tolerant Gate Sets of Dimension Greater than Two. Diss. 2019.

Warnings:
I have used a fair amount of input checking so that functions only accept inputs of the appropriate form. This comes at the cost of some speed -- that being said, these checks cause constant overhead, and so their performance impact is worth it to prevent some erroneous calculation from being carried out. If you don't care about this input checking, one could relatively easily define their own functions from my own internal ones to slightly speed up their performance.

Limitations:
This package is only intended for usage on two-qubit circuits. To perform circuit synthesis on larger circuits, I suggest loading this package and using these functions as subroutines.

Discussion:

Rather than describe these algorithms in detail here, I differ to the sources [1-3] listed above or the function descriptions.

Requirements:
None

```
BeginPackage["GaussSynth`"];
```

Function Usage

```
GaussSynth::usage = "GaussSynth is a package for quantum compiling for two-qubit 

Id::usage = "Id is the 4x4 Identity Matrix.";
X1::usage = "X1 is the unitary representation of the X⊗I gate.";
X2::usage = "X2 is the unitary representation of the I⊗X gate.";
Z1::usage = "Z1 is the unitary representation of the Z⊗I gate.";
Z2::usage = "Z2 is the unitary representation of the I⊗Z gate.";
W::usage = "W is the unitary representation of the primite 8th root of unity ω.";
H1::usage = "H1 is the unitary representation of the H⊗I gate.";
H2::usage = "H2 is the unitary representation of the I⊗H gate.";
S1::usage = "S1 is the unitary representation of the S⊗I gate.";
S2::usage = "S2 is the unitary representation of the I⊗S gate.";
CZ::usage = "CZ is the unitary representation of the CZ gate.";
CNOT12::usage = "CNOT12 is the unitary representation of the CNOT gate with contr
CNOT21::usage = "CNOT21 is the unitary representation of the CNOT gate with contr
EX::usage = "EX is the unitary representation of the SWAP (Exchange) gate.";
CS::usage = "CS is the unitary representation of the CS gate.";

R::usage = "R[P,Q] takes as input two strings P and Q of the form \"AB\" or \"-AB
    These correspond to two-qubit Hermitian Pauli operators, and if P and Q are d
    computes the corresponding unitary gate as defined in our work. These composi
    to study the Clifford + CS gate set.";

U4ToSO6::usage = "U4ToSO6[U] maps the 4x4 unitary U to the equivalent SO(6) repre
    for some phase φ and U' an element of SU(4).";

IdSO6::usage = "Id is the 6x6 Identity Matrix.";
X1SO6::usage = "X1SO6 is the SO(6) representation of the X⊗I gate.";
X2SO6::usage = "X2SO6 is the SO(6) representation of the I⊗X gate.";
Z1SO6::usage = "Z1SO6 is the SO(6) representation of the Z⊗I gate.";
Z2SO6::usage = "Z2SO6 is the SO(6) representation of the I⊗Z gate.";
ISO6::usage = "ISO6 is the SO(6) representation of the complex phase I. As SU(4) 
    we have \!\(\*SuperscriptBox[\(ISO6\), \(2\)]\) = IdSO6.";
```

```
H1SO6::usage = "H1SO6 is the SO(6) representation of the H⊗I gate.";
H2SO6::usage = "H2SO6 is the SO(6) representation of the I⊗H gate.";
S1SO6::usage = "S1SO6 is the SO(6) representation of the S⊗I gate.";
S2SO6::usage = "S2SO6 is the SO(6) representation of the I⊗S gate.";
CZSO6::usage = "CZSO6 is the SO(6) representation of the CZ gate. Note that this g
    by a primitive 8th root of unity \!\(\*SuperscriptBox[\(ω\), \(†\)]\) before p
CNOT12SO6::usage = "CNOT12SO6 is the SO(6) representation of the CNOT gate with co
    Note that this gate is not special unitary, and so we multiply by a primitive
    before performing the transformation.";
CNOT21SO6::usage = "CNOT21SO6 is the SO(6) representation of the CNOT gate with co
    Note that this gate is not special unitary, and so we multiply by aa primitiv
    before performing the transformation.";
EXSO6::usage = "EXSO6 is the SO(6) representation of the SWAP (Exchange) gate. No
    and so we multiply by a primitive 8th root of unity \!\(\*SuperscriptBox[\(ω\
CSSO6::usage = "CSSO6 is the SO(6) representation of the CS gate. Note that this g
    a primitive 16th root of unity \!\(\*SuperscriptBox[\(ω\), \(\(-1\)/2\)]\) be

RSO6::usage = "R[P,Q] takes as input two strings P and Q of the form \"AB\" or \"-
    These correspond to two-qubit Hermitian Pauli operators, and if P and Q are d
    computes the corresponding SO(6) gate as defined in our work. These composite
    to study the Clifford + CS gate set. Note that this gate is not special unita
    before performing the transformation.";

FromSequence::usage = "FromSequnce[str] reads in the string str and interprets tha

FromHexDec::usage = "FromHexDec[str] attempts to read in a string of a signed hexa
    The sign indicates whether the operator corresponds to using symmetric or asy
    the fifteen unique syllables. After the syllables comes the flag 00000 which 

CliffordQ::usage = "CliffordQ[U] returns True if U is a Clifford and False otherw
CliffordSynth::usage = "CliffordSynth[U] gives the index number (in Hexidecimal) a
    which constitute the Clifford which can be input as a string, an element of U
RightCliffordSimilar::usage = "CliffordSimilarRight[U,V] Returns True if there is 
LeftCliffordSimilar::usage = "CliffordSimilarLeft[U,V] Returns True if there is a

GaussianQ::usage = "GaussianQ[U] is a Boolean function which checks if U correspon

LDE::usage = "LDE[U] takes as input either a U(4) or SO(6) Clifford + CS operator 
OptimalCSCount::usage = "OptimalCSCount[U] takes as input any representation of a

SyllableList::usage = "A list of 15 syllables of CS-count one which are not right-
    Clifford C which conjugates CS as C.CS.\!\(\*SuperscriptBox[\(C\), \(†\)]\).
    4x4 Unitary representation, the operator's 6x6 SO(6) representation, and its 
SyllableListAsymmetric::usage = "An alternative list of 15 syllables of CS-count o
```

```
    Each syllable is equivalent to C.CS for C a Clifford. For each syllable, we s
    the operator's 6x6 SO(6) representation, and its name according to the genera

NormIt::usage = "NormIt[U,options] takes as input a Gaussian Clifford + T operato
    This normal form is output as a string of generators using the standard sylla
    The options for the \"OutputType\" are \"String\" or \"HexDec\", the options 
    and the options for \"UpToPhase\" are the booleans True and False. When eithe
    following characters: \"W\", \"S1\", \"S2\", \"H1\", \"H2\", \"CZ\", \"EX\", 
    \"S1\" is the gate S1, and so on.";

FrobeniusDistance::usage = "FrobeniusDistance[U,V] computes the distance between 

PauliRotation::usage = "PauliRotation[φ,ϵ,Pauli] finds a unitary Gaussian Cliffor
    of the Pauli rotation \!\(\*SuperscriptBox[\(e\), \(\(-iφ\)/2\\ P\)]\) for t
    The Pauli can be one of the fifteen strings \"XI\", \"YI\", \"ZI\", \"IX\", \
    \"XZ\", \"YZ\", or \"ZZ\".";
PauliRotationSequence::usage = "PauliRotationSequence[φ,ϵ,Pauli,options] finds a 
    of the Pauli rotation \!\(\*SuperscriptBox[\(e\), \(\(-iφ\)/2\\ P\)]\) for t
    The Pauli can be one of the fifteen strings \"XI\", \"YI\", \"ZI\", \"IX\", \
    \"XZ\", \"YZ\", or \"ZZ\". It then outputs a normalized sequence of Clifford 
    \"String\" or \"HexDec\" and the options for the \"SyllableType\" are \"Norma
PauliDecomposition::usage = "PauliDecomp[U] finds a list of 15 angle parameters \
    decomposition of the form \!\(\*SubscriptBox[\(∏\), \(1 ≤ j ≤ 15\)]\)\!\(\*Su
    \(\(-\*SubscriptBox[\(iφ\), \(j\)]\)/2\\ \*SubscriptBox[\(P\), \(j\)]\)]\) f
    the Paulis \!\(\*SubscriptBox[\(P\), \(J\)]\) is (ZI,XI,ZI,IZ,IX,IZ,XX,YY,ZZ,
ApproximateOp::usage = "Approximate[U,ϵ] finds an approximation within Frobenius 
    If U is an element of U(4), the result is a U(4) representation of a Clifford
    SO(6)  representation of a Clifford + CS circuit. Note that the Frobenius dis
    representation.";
ApproximateSequence::usage = "ApproximateSequence[U,ϵ,options] finds a normalized
    (in the Unitary representation and up to an irrelevant phase) for the input U
    string unless otherwise specified in the options. The options for the \"Outpu
    \"SyllableType\" are \"Normal\" or \"Asymmetric\", and the options for \"IfGa

RandomCliffCS::usage = "RandomCliffCS[n] pseudorandomly samples from the uniform 
```

Possible Errors

```
General::invldopt = "Option `2` for function `1` received invalid value `3`";
U4ToSO6::notunitary = "The argument must be a 4x4 unitary matrix.";
R::invalidpaulis = "The arguements `1` and `2` must be strings of the form \"AB\"
    Furthermore, the strings must correspond to commuting non-identity two-qubit
FromSequence::notstring = "You have not entered a string.";
FromList::notagate = "The string `1` is not one of \"W\", \"S1\", \"S2\", \"H1\",
    You may have forgotten a space between gate names or used a name for a gate w
FromHexDec::invalidnumber = "The string `1` is not a valid hexadecimal representa
    seperating the syllables from the Clifford. Otherwise, ensure the Clifford ha
    before being written in hexadecimal representation, and that your syllables o
CliffordQ::notacircuit = "You have not entered a string of operators, a valid hex
CliffordSynth::notaclifford = "Your input is not a Clifford operator in string fo
GaussianQ::notacircuit = "You have not entered a string of operators, a valid hex
LDE::notamatrix = "You have not entered an element of SO(6) or  U(4) which is a C
OptimalCSCount::notacircuit = "You have not entered a string of operators, a vali
NormIt::badopt = "The option `1` is not valid for `2`.";
NormIt::notacircuit = "You have not entered a string of operators, a valid hexade
FrobeniusDistance::notequidimensionalmatrices = "Your inputs are not two matrices
CandidateFinder::notreals = "Your inputs are not two real numbers";
PauliRotation::notreals = "Your input does not include two real numbers";
PauliRotation::invldstring = "Your input does not include one string from the set
    \"YX\", \"ZX\", \"XY\", \"YY\", \"ZY\", \"XZ\", \"YZ\", or \"ZZ\".";
PauliDecomposition::notanoperator = "Your input is neither an element of U(4) or
ApproximateOp::notreal = "Your error tolerance is not a real number.";
ApproximateOp::notanoperator = "Your input is neither an element of U(4) or SO(6)
RandomCliffCS::notanatural = "Your input is not a natural number.";
```

```
Begin["`Private`"];
```

Function Definitions

Functions for option checking

These functions will be used to check options for functions which accept them. For each such function, we must define a test[f,op] function for a particular option type op of function f. Credit for this code snippet goes to Mr. Wizard in the Stack Overflow post https://mathematica.stackexchange.com/questions/116623.

```
optsMsg[f_][op_, val_] :=
  test[f, op][val] || Message[General::invldopt, f, op, val];

Attributes[optsCheck] = {HoldFirst};

optsCheck @ head_[___, opts : OptionsPattern[]] :=
  And @@ optsMsg[head] @@@ FilterRules[{opts}, Options @ head];
```

Constants and Single-Qubit operators

For internal use only.

```
ω = (1+I)/Sqrt[2];
ζ = Exp[I*Pi/8];
s = DiagonalMatrix[{1,I}];
h = 1/Sqrt[2]*{{1,1},{1,-1}};
id = PauliMatrix[0];
x = PauliMatrix[1];
y = PauliMatrix[2];
z = PauliMatrix[3];
```

Unitary Representations of Two-qubit Clifford + CS operators

These operators are exported to the user as 4x4 matrices in the standard Mathematica format.

Fixed Two-Qubit Gates

These are the traditional two-qubit Clifford + CS gates.

```
Id = IdentityMatrix[4];
W = ω*Id;
S1 = KroneckerProduct[s,id];
S2 = KroneckerProduct[id,s];
H1 = KroneckerProduct[h,id];
H2 = KroneckerProduct[id,h];
CZ = DiagonalMatrix[{1,1,1,-1}];
CNOT12 = {
    {1,0,0,0},
    {0,1,0,0},
    {0,0,0,1},
    {0,0,1,0}
};
CNOT21 = {
    {1,0,0,0},
    {0,0,0,1},
    {0,0,1,0},
    {0,1,0,0}
};
EX = {
    {1,0,0,0},
    {0,0,1,0},
    {0,1,0,0},
    {0,0,0,1}
};
CS = DiagonalMatrix[{1,1,1,I}];
X1 = KroneckerProduct[x,id];
X2 = KroneckerProduct[id,x];
Z1 = KroneckerProduct[z,id];
Z2 = KroneckerProduct[id,z];
```

R[P,Q] Operators

This code provides the R[P,Q] gate constructor, as in [2].

```
NoPhaseHermitianPaulis = {"IX","IY","IZ","XI","XX","XY","XZ","YI","YX","YY","YZ",'
HermitianPaulis = Join[NoPhaseHermitianPaulis,Map["-"<>#&,NoPhaseHermitianPaulis]]

SinglePauliTranslater["I"] = id;
SinglePauliTranslater["X"] = x;
SinglePauliTranslater["Y"] = y;
SinglePauliTranslater["Z"] = z;

PauliTranslater[str_] := PauliTranslater[str] = Module[{ablist,a,b,phase},
    ablist = Characters @ StringTake[str,-2];
    {a,b} = Map[SinglePauliTranslater,ablist];
    phase = If[StringStartsQ[str,"-"],-1,1];
    phase * KroneckerProduct[a,b]
];

HermitianPauliQ[P_] := MemberQ[HermitianPaulis,P];
DistCommuteHermitePauliQ[P_?HermitianPauliQ,Q_?HermitianPauliQ] := Module[{p,q,cor
    p = PauliTranslater[P];
    q = PauliTranslater[Q];
    commutebool = p.q == q.p;
    distinctbool = Not[MemberQ[{Id,-Id},p.q]];
    commutebool && distinctbool
];
DistCommuteHermitePauliQ[P_,Q_] = False;

R[P_,Q_]/;DistCommuteHermitePauliQ[P,Q] := R[P,Q] = Module[{p,q},
    p = PauliTranslater[P];
    q = PauliTranslater[Q];
    Id +(I-1)/4*(Id - p).(Id - q)
];
R[_,_] := (
    Message[R::invalidpaulis];
    $Failed
);
```

Checks For U(4) and SO(6)

These Boolean functions determine whether an operator is an element of U(4) or SO(6), respectively.

```
U4Q[U_]:= UnitaryMatrixQ[U] && (Dimensions[U] == {4,4});
SO6Q[O_] := OrthogonalMatrixQ[O] && (Dimensions[O] == {6,6}) && (Det[O] == 1);
```

The SU(4)≅SO(6) Isomorphism

These definitions and functions allow one to compute the SO(6) representation of an element of U(4) (up to a phase).

Rules for Inner Products of Wedge Products

Defined using the unassigned Mathematica symbols of ∧, 〈, and 〉.

```
〈a_,0_,b_*c_〉:=b*〈a,0,c〉;
〈a_*b_,0_,c_〉 := Conjugate[a]*〈b,0,c〉;
〈a_+b_,0_,c_〉:=〈a,0,c〉+〈b,0,c〉;
〈a_,0_,b_+c_〉:=〈a,0,b〉+〈a,0,c〉;
〈x_∧y_,0_,u_∧v_〉 := (Conjugate[x].0.u)*(Conjugate[y].0.v) - (Conjugate[x].0.v)*(Co
```

Orthonormal Basis for Subscript[ℂ, 6]

This basis is such that computing the above inner products for an element of U(4,ℂ) will produce an element of SO(6,ℝ). Moreover, the representations for Clifford + CS operators are easy to work with in this basis.

```
Basis6 ={
    (I/Sqrt[2])*(UnitVector[4,1]∧UnitVector[4,2] - UnitVector[4,3]∧UnitVector[4,4
    (1/Sqrt[2])*(UnitVector[4,1]∧UnitVector[4,2] + UnitVector[4,3]∧UnitVector[4,4
    (I/Sqrt[2])*(UnitVector[4,2]∧UnitVector[4,3] - UnitVector[4,1]∧UnitVector[4,4
    (1/Sqrt[2])*(UnitVector[4,2]∧UnitVector[4,4] + UnitVector[4,3]∧UnitVector[4,1
    (I/Sqrt[2])*(UnitVector[4,2]∧UnitVector[4,4] - UnitVector[4,3]∧UnitVector[4,1
    (1/Sqrt[2])*(UnitVector[4,2]∧UnitVector[4,3] + UnitVector[4,1]∧UnitVector[4,4
};
```

Calculating the SO(6) Representation for an element of SU(4) (and from U(4) up to a phase)

Functions for mapping elements of SU(4) to SO(6) and U(4) to SO(6) (by converting that element of U(4) to an element of SU(4)).

```
SU4ToSO6[U_] := Table[Simplify[〈Basis6[[i]],U,Basis6[[j]]〉],{i,1,6},{j,1,6}];

U4ToSO6[U_/;U4Q[U]] := SU4ToSO6[1/(Det[U])^(1/4)*U];
U4ToSO6[U_] := (
    Message[U4ToSO6::notunitary];
    $Failed
);
```

SO(6) Representations of Two-qubit Clifford + CS operators

Calculated using our transformations. These operators are exported to the user as 6x6 matrices in the standard Mathematica format. Note that we have to multiply by overall phases to ensure that the transformation uses an element of SU(4).

```
IdSO6 = SU4ToSO6[Id];
ISO6 = SU4ToSO6[W.W];
H1SO6 = SU4ToSO6[H1];
H2SO6 = SU4ToSO6[H2];
S1SO6 = SU4ToSO6[ω^(-1)*S1];
S2SO6 = SU4ToSO6[ω^(-1)*S2];
CZSO6 = SU4ToSO6[ω^(-1)*CZ];
CNOT12SO6 = SU4ToSO6[ω^(-1)*CNOT12];
CNOT21SO6 = SU4ToSO6[ω^(-1)*CNOT21];
EXSO6 = SU4ToSO6[ω^(-3)*EX];
CSSO6 = SU4ToSO6[ζ^(-1)*CS];
X1SO6 = SU4ToSO6[X1];
X2SO6 = SU4ToSO6[X2];
Z1SO6 = SU4ToSO6[Z1];
Z2SO6 = SU4ToSO6[Z2];
RSO6[P_,Q_] := RSO6[P,Q] = SU4ToSO6[ζ^(-1)*R[P,Q]];
```

Custom Representations of SO(6) Clifford + CS operators

Our synthesis algorithms will use a custom data type for the SO(6) representation of a Clifford + CS operator. The basic data structure of this special representation is as follows:

{k,M} := 2^(-k/2) · M

This allows easy tracking of the lde. They are packed in SparseArrays to help make things even a little faster, as every Clifford is just a permutation matrix. These representations are for internal use only.

Switching between the standard representation for a 6x6 matrix and a representation specifically for Clifford + CS operators.

Functions for converting to and from the special representation.

```
SO6ToSpecialRep[M_] := Module[{LDE,IntegerMat},
    LDE = FullSimplify[Max[Map[Simplify[Log[Sqrt[2],Denominator[#]]]&,Flatten[Sim
    IntegerMat = FullSimplify[(Sqrt[2]^LDE)*M];
    {LDE,SparseArray[IntegerMat]}
];
SpecialRepToSO6[{k_,M_}] := Simplify[1/Sqrt[2]^k*Normal[M]];
```

Special Representations for SO(6) Clifford + CS operators

```
IdSp = SO6ToSpecialRep[IdSO6];
ISp = SO6ToSpecialRep[ISO6];
H1Sp = SO6ToSpecialRep[H1SO6];
H2Sp = SO6ToSpecialRep[H2SO6];
S1Sp = SO6ToSpecialRep[S1SO6];
S2Sp = SO6ToSpecialRep[S2SO6];
CZSp = SO6ToSpecialRep[CZSO6];
CNOT12Sp = SO6ToSpecialRep[CNOT12SO6];
CNOT21Sp = SO6ToSpecialRep[CNOT21SO6];
EXSp = SO6ToSpecialRep[EXSO6];
CSSp = SO6ToSpecialRep[CSSO6];
X1Sp = SO6ToSpecialRep[X1SO6];
X2Sp = SO6ToSpecialRep[X2SO6];
Z1Sp = SO6ToSpecialRep[Z1SO6];
Z2Sp = SO6ToSpecialRep[Z2SO6];
RSp[P_,Q_] := RSp[P,Q] = SO6ToSpecialRep[RSO6[P,Q]];
```

Basic Matrix Operations for the Special Representation

These internal functions are used to reduce the denominator exponent to the lde, multiply operators in the special representation, and invert the special representation.

```
KReduceOnce[{k_,a_}]:=If[AllTrue[a,EvenQ,2] && k>1,{k - 2,a/2},{k,a}];
KReduce[o_]:=FixedPoint[KReduceOnce,o];

Dot2Sp[{k1_,a1_},{k2_,a2_}]:= KReduce[{k1 + k2,a1.a2}];
DotSp[x__]:=Fold[Dot2Sp,IdSp,{x}];

InvSp[{k_,a_}]:={k,Transpose[a]};
```

Functions for Residues Modulo 2 and Finding Paired Matrix Rows

These functions are used for finding paired rows in operators in the special representation.

```
PatternMats[{k_,a_}] := Mod[a,2];

MatrixRowPairs[list_] := GatherBy[
    Range@Length[Normal[list]],
    Normal[list][[#]]&
];

RowPairs[x_] := Sort@MatrixRowPairs@PatternMats@x
```

String Reading

Functions for reading in a string of operators. The string is always read in as an element of U(4).

```
FromSequence[str_String] := Module[{strlist},
    strlist = StringSplit[str];
    FromList[strlist]
];
FromSequence[x_] := (
    Message[FromSequence::notstring];
    $Failed
);

FromList[{}] = Id;
FromList[{"W"}] = W;
FromList[{"S1"}] = S1;
FromList[{"S2"}] = S2;
FromList[{"H1"}] = H1;
FromList[{"H2"}] = H2;
FromList[{"CZ"}] = CZ;
FromList[{"CS"}] = CS;
FromList[{"EX"}] = EX;
FromList[{"X1"}] = X1;
FromList[{"X2"}] = X2;
FromList[{"Z1"}] = Z1;
FromList[{"Z2"}] = Z2;
FromList[{str_/;(StringMatchQ[str,"R[*,*]"] && StringCount[str,","] == 1)}] := Mod
    ops = StringSplit[str,{"R[",",","]"}];
    R @@ ops
];
FromList[{str_}]:= (
    Message[FromList::notagate,str];
    $Failed
);
FromList[{h_,t__}] := FromList[{h}] . FromList[{t}];
```

Hexadecimal Representations

We shall use strings of signed hexadecimal integers to represent Clifford + CS operators. This form is much more compact than, for example, the string form of an operator. The string must be of the following form:

\"(| -)(1-9 | a-f)* 00000 C\"

where C is the hexadecimal representation of an integer from 1 to 92160 and the first option is simply an optional "-" character.

```
ValidHexDecQ[num_String] := Module[{seperator,typenum,syllabletype,syllablescliff
    seperator = StringCount[num,"00000"] == 1;
```

```
    typenum = StringCount[num,"-"];
    syllabletype = (typenum == 0) || ((typenum == 1) && StringStartsQ[num,"-"]);
    syllablescliffordok = If[
        seperator && syllabletype,
            split = StringSplit[num,{"00000","-"}];
            {syllables,clifford} = If[Length[split] > 1,split,{"",split[[1]]}];
            syllablescharacterlist = Union[Characters[syllables]];
            syllablesok = SubsetQ[{"1","2","3","4","5","6","7","8","9","a","b","c
            cliffordcharacterlist = Union[Characters[clifford]];
            cliffordpossible = (StringLength[clifford]) <= 5 && SubsetQ[{"0","1",
            cliffordok = If[
                cliffordpossible,
                    1 <= FromDigits[clifford,16] <= 92160,
                    False
            ];
            syllablesok && cliffordok,
            False
    ];
    syllablescliffordok
];
ValidHexDecQ[U_] := False

PhaseSet = {Id,MatrixPower[W,4],W,MatrixPower[W,5]};
L1Set = {Id,H1,S1.H1.MatrixPower[W,7]};
L2Set = {Id,H2,S2.H2.MatrixPower[W,7]};
CZSet = {Id,CZ.MatrixPower[W,7]};
S1Set = {Id,S1.MatrixPower[W,7]};
P1Set = {Id,X1,X1.Z1,Z1};
S2Set = {Id,S2.MatrixPower[W,7]};
P2Set = {Id,X2,X2.Z2,Z2};
SWAPSet = {Id,EX.MatrixPower[W,5]};
ISet = {Id,MatrixPower[W,2]};

CliffordFromNumber[cliffnum_] := Module[{digits,cz,l1,l2,index},
    digits = PadLeft[IntegerDigits[cliffnum-1,MixedRadix[{4,10,3,3,2,4,2,4,2,2}]]
    cz = Unitize[digits[[2]]];
    l2 = Mod[digits[[2]]-cz,3];
    l1 = (digits[[2]]-l2-cz)/3;
    index = Join[{digits[[1]],l1,l2,cz},digits[[3;;-1]]] + 1;
    PhaseSet[[index[[1]]]] . L1Set[[index[[2]]]] . L2Set[[index[[3]]]] . CZSet[[i
        L1Set[[index[[5]]]] . L2Set[[index[[6]]]] . S1Set[[index[[7]]]] . P1Set[[
        S2Set[[index[[9]]]] . P2Set[[index[[10]]]] . SWAPSet[[index[[11]]]] . ISe
];
```

```
FromHexDec[str_String/;ValidHexDecQ[str]] := Module[{syllablelist,split,syllables
    syllablelist = If[StringStartsQ[str,"-"],SyllableListAsymmetric[[All,1]],Sylla
    split = StringSplit[str,{"00000","-"}];
    {syllablestr,cliffstr} = If[Length[split] > 1,split,{"",split[[1]]}];
    cliffnum = FromDigits[cliffstr,16];
    cliff = CliffordFromNumber[cliffnum];
    syllables = Map[syllablelist[[FromDigits[#,16]]]&,Characters[syllablestr]];
    Simplify[Dot @@ (Append[syllables,cliff])]
];
FromHexDec[str_] := (
    Message[FromHexDec::invalidnumber,str];
    $Failed
);
```

The Two-Qubit Clifford Group

Here we develop some basic functions for the two-qubit Clifford group.

Identifying if an operator is a Clifford and if two operators are Clifford-similar

Boolean functions for checking if an operator is a Clifford or if two operators are Clifford-similar.

```
CliffordQ[U_/;U4Q[U]] := (Det[U]^2 == 1) && AllTrue[Flatten[U4ToSO6[U]],IntegerQ]
CliffordQ[U_/;SO6Q[U]] := AllTrue[Flatten[U],IntegerQ];
CliffordQ[U_String/;ValidHexDecQ[U]] := CliffordQ[FromHexDec[U]];
CliffordQ[U_String] := CliffordQ[FromSequence[U]];
CliffordQ[U_] := (
    Message[CliffordQ::notacircuit];
    $Failed
);

RightCliffordSimilar[U_String/;ValidHexDecQ[U],V_String/;ValidHexDecQ[V]] := Righ
RightCliffordSimilar[U_String,V_String] := RightCliffordSimilar[FromSequence[U],Fr
RightCliffordSimilar[U_,V_] := CliffordQ[ConjugateTranspose[U].V];

LeftCliffordSimilar[U_String/;ValidHexDecQ[U],V_String/;ValidHexDecQ[V]] := LeftCl
LeftCliffordSimilar[U_String,V_String] := LeftCliffordSimilar[FromSequence[U],From
LeftCliffordSimilar[U_,V_] := CliffordQ[V.ConjugateTranspose[U]];
```

Synthesis of Clifford Circuits (with at most 1 CZ gate and 1 SWAP gate)

Rather than carry around an explicit lookup table with 92160 elements in the U(4) case and 23040 elements in the SO(6) case, we instead will use a synthesis algorithm based on the special representation; this takes advantage of the sparsity of Cliffords in this representation. We implicitly define our regular Clifford synthesis algorithm, CliffordSynth, in terms of this other algorithm, to be defined below.

```
PhaseFixU4[{str_,power_,SUcliffnum_},U_] := Module[{unitary,phase,Wpower,diff,Wco
    unitary = FromSequence[str];
    phase = Simplify[(U. ConjugateTranspose[unitary])[[1,1]]];
    Wpower = Mod[Log[(1+I)/Sqrt[2],phase],8];
    diff = Mod[Wpower - power,8];
    Wcosetnumber = Which[
        (diff == 0) || (diff == 2),0,
        (diff == 4) || (diff == 6),1,
        (diff == 1) || (diff == 3),2,
        True,3
    ];
    {IntegerString[23040*Wcosetnumber + FromDigits[SUcliffnum,16],16],str<>String
];
PhaseFixSO6[{str_,power_,SUcliffnum_}] := {SUcliffnum,str<>StringRepeat[" W",powe

CliffordSynth[U_String/;ValidHexDecQ[U]] := Module[{unitary},
    unitary = FromSequence[U];
    CliffordSynth[unitary]
];
CliffordSynth[U_String] := Module[{unitary},
    unitary = FromSequence[U];
    CliffordSynth[unitary]
];
CliffordSynth[U_/;(U4Q[U] && CliffordQ[U])] := Module[{orthogonal,synth},
    orthogonal = U4ToSO6[U];
    synth = CliffordSynthSp[{0,SparseArray[orthogonal]}];
    PhaseFixU4[synth,U]
];
CliffordSynth[U_/;CliffordQ[U]] := PhaseFixSO6[CliffordSynthSp[{0,SparseArray[U]}
CliffordSynth[U_] := (
    Message[CliffordSynth::notaclifford];
    $Failed
);
```

Clifford Group in the Special representation

Explicit CliffordSynthSp algorithm. We perform the synthesis by decomposing an element of the signed permutation group in terms of a generating set which is constructed from basic Clifford operators in the SO(6) representation.

```
L1Cosets = {
    {IdSp,"",0,0},
```

```
    {H1Sp,"H1 ",0,1},
    {DotSp[S1Sp,H1Sp],"S1 H1 ",7,2}
};
L2Cosets = {
    {IdSp,"",0,0},
    {H2Sp,"H2 ",0,1},
    {DotSp[S2Sp,H2Sp],"S2 H2 ",7,2}
};
CZSets = {
    {IdSp,"",0,0},
    {CZSp,"CZ ",7,1}
};
S1Sets = {
    {IdSp,"",0,0},
    {S1Sp,"S1 ",7,1},
};
Pauli1Sets = {
    {IdSp,"",0,0},
    {X1Sp,"X1 ",0,1},
    {DotSp[X1Sp,Z1Sp],"X1 Z1 ",0,2},
    {Z1Sp,"Z1 ",0,3}
};
S2Sets = {
    {IdSp,"",0,0},
    {S2Sp,"S2 ",7,1},
};
Pauli2Sets = {
    {IdSp,"",0,0},
    {X2Sp,"X2 ",0,1},
    {DotSp[X2Sp,Z2Sp],"X2 Z2 ",0,2},
    {Z2Sp,"Z2 ",0,3}
};
SwapSets = {
    {IdSp,"",0,0},
    {EXSp,"EX ",5,1}
};
ISets = {
    {IdSp,"",0,0},
    {ISp,"",2,1}
};

CliffordQSp[U_] := Module[{k,M},
    {k,M} = KReduce[U];
```

```
    (k == 0) && OrthogonalMatrixQ[M] && (Det[M] == 1)
];

CliffordSynthSp[{_,M_}] := Module[
    {
        CliffordList,SwapTest,MUnSwapped,UpperLeft,LowerRight,RemovedLeftCosets,C
        UpperLeftCol,LowerRightCol,RemovedLeftCosetsNew,S1Test,S2Test,RemovedSSet
        RemovedPauli1Sets,d3new,d4,d5,d6,RemovedPauli2Sets,ITest,str,phase,list,f
    },

    CliffordList = ConstantArray[0,11];

    SwapTest = Total[Abs[M[[1;;3,1;;3]]],2];
    CliffordList[[10]] = If[SwapTest > 1,SwapSets[[1]],SwapSets[[2]]];
    MUnSwapped = M.Transpose[CliffordList[[10,1,2]]];

    UpperLeft = Total[Abs[MUnSwapped[[1;;3,1;;3]]],{2}];
    LowerRight = Total[Abs[MUnSwapped[[4;;6,4;;6]]],{2}];
    CliffordList[[1]] = Which[
        UpperLeft == {0,1,1},L1Cosets[[2]],
        UpperLeft == {1,0,1},L1Cosets[[3]],
        True,L1Cosets[[1]]
    ];
    CliffordList[[2]] = Which[
        LowerRight == {0,1,1},L2Cosets[[2]],
        LowerRight == {1,0,1},L2Cosets[[3]],
        True,L2Cosets[[1]]
    ];
    RemovedLeftCosets = Transpose[CliffordList[[1,1,2]].CliffordList[[2,1,2]]].MU

    CZRows = Total[Abs[RemovedLeftCosets[[3,1;;3]]],2];
    CliffordList[[3]] = If[CZRows == 1,CZSets[[1]],CZSets[[2]]];
    RemovedCZSet = Transpose[CliffordList[[3,1,2]]].RemovedLeftCosets;

    UpperLeftCol = Abs[RemovedCZSet[[1;;3,3]]];
    LowerRightCol = Abs[RemovedCZSet[[4;;6,6]]];
    CliffordList[[4]] = Which[
        UpperLeftCol == {0,0,1},L1Cosets[[1]],
        UpperLeftCol == {0,1,0},L1Cosets[[3]],
        True,L1Cosets[[2]]
    ];
    CliffordList[[5]] = Which[
        LowerRightCol == {0,0,1},L2Cosets[[1]],
```

```
    LowerRightCol == {0,1,0},L2Cosets[[3]],
    True,L2Cosets[[2]]
];
RemovedLeftCosetsNew = Transpose[CliffordList[[4,1,2]].CliffordList[[5,1,2]]]

S1Test = Abs[RemovedLeftCosetsNew[[1,1]]];
S2Test = Abs[RemovedLeftCosetsNew[[4,4]]];
CliffordList[[6]] = If[S1Test == 1,S1Sets[[1]],S1Sets[[2]]];
CliffordList[[8]] = If[S2Test == 1,S2Sets[[1]],S2Sets[[2]]];
RemovedSSets = Transpose[CliffordList[[6,1,2]].CliffordList[[8,1,2]]].Removed

{d1,d2,d3} = Diagonal[RemovedSSets[[1;;3,1;;3]]];
CliffordList[[7]] = Which[
    (d1 == d2) && (d2 == d3),Pauli1Sets[[1]],
    (d1 != d2) && (d2 == d3),Pauli1Sets[[2]],
    (d1 != d2) && (d2 != d3),Pauli1Sets[[3]],
    True,Pauli1Sets[[4]]
];
RemovedPauli1Sets = Transpose[CliffordList[[7,1,2]]].RemovedSSets;

{d3new,d4,d5,d6} = Diagonal[RemovedPauli1Sets[[3;;6,3;;6]]];
CliffordList[[9]] = Which[
    (d4 == d5) && (d5 == d6) && (d3new*d4 == 1),Pauli2Sets[[1]],
    (d4 != d5) && (d5 == d6),Pauli2Sets[[2]],
    (d4 != d5) && (d5 != d6),Pauli2Sets[[3]],
    True,Pauli2Sets[[4]]
];
RemovedPauli2Sets = Transpose[CliffordList[[9,1,2]]].RemovedPauli1Sets;

ITest = RemovedPauli2Sets[[1,1]];
CliffordList[[11]] = If[ITest == 1,ISets[[1]],ISets[[2]]];

{str,phase,list} = Fold[{
        #1[[1]]<>#2[[2]],
        Mod[#1[[2]]+#2[[3]],8],
        Append[#1[[3]],#2[[4]]]
    }&,
    {"",0,{}},
    CliffordList
];
firstdigit = list[[3]] * (list[[1]]*3 + list[[2]] + 1);
cliffordnumber = FromDigits[Prepend[list[[4;;-1]],firstdigit],MixedRadix[{10,
```

```
    {str,phase,IntegerString[cliffordnumber,16]}
];
```

The Two-Qubit Clifford + CS Group

Here we develop some basic constructors for Clifford + CS circuits. We also provide a function for checking if an operator corresponds to a Clifford + CS circuit.

Check If an Operator Is a Gaussian Clifford + T Matrix

We define a function for determining if an operator is a representation for a Gaussian Clifford + T matrix, i.e. a Clifford + CS operator.

```
DyadicQ[n_] := Module[{numerator,denominator},
    {numerator,denominator} = NumeratorDenominator[n];
    IntegerQ[numerator] && IntegerQ[Log2[denominator]]
];
GaussianDyadicQ[n_]:= AllTrue[ReIm[n],DyadicQ];

GaussianQ[U_/;U4Q[U]] := Module[{UGaussDyadicQ,UWGuassDyadicQ},
    UGaussDyadicQ = AllTrue[Flatten[U],GaussianDyadicQ];
    UWGuassDyadicQ = AllTrue[Flatten[W.U],GaussianDyadicQ];
    UGaussDyadicQ || UWGuassDyadicQ
];
GaussianQ[U_/;SO6Q[U]] := Module[{sp},
    sp = SO6ToSpecialRep[U];
    (Det[U] == 1) && IntegerQ[sp[[1]]] && AllTrue[Flatten[sp[[2]]],IntegerQ]
];
GaussianQ[U_/;ValidHexDecQ[U]] := True;
GaussianQ[U_String] := GaussianQ[FromSequence[U]];
GaussianQ[U_] := (Message[GaussianQ::notacircuit];
    $Failed
);
```

Least Denominator Exponents and CS-counts

These functions can be used to find denominator exponents and optimal-CS counts.

```
LDE[U_/;(GaussianQ[U] && U4Q[U])] := Module[{reimlist},
    reimdenomlist = Denominator @ Flatten @ ReIm[U];
    FullSimplify[Max[Map[Simplify[Log[Sqrt[2],#]]&,reimdenomlist]]]
];
LDE[U_/;(GaussianQ[U] && SO6Q[U])] := Module[{sp},
    sp = SO6ToSpecialRep[U];
    sp[[1]]
];
LDE[_] := (
    Message[LDE::notamatrix];
    $Failed
);

OptimalCSCount[U_/;(GaussianQ[U] && U4Q[U])] := LDE[U4ToSO6[U]];
OptimalCSCount[O_/;(GaussianQ[O] && SO6Q[O])] := LDE[O]
OptimalCSCount[hexdec_String/;ValidHexDecQ[hexdec]] := OptimalCSCount[FromHexDec[
OptimalCSCount[str_String] := OptimalCSCount[FromSequence[str]];
OptimalCSCount[U_] := (
    Message[OptimalCSCount::notacircuit,U];
    $Failed
);
```

Syllable Lists

For the exported versions:

```
SyllableList = {
    {R["XI","IX"],RSO6["XI","IX"],"R[XI,IX] "},
    {R["YI","IY"],RSO6["YI","IY"],"R[YI,IY] "},
    {R["ZI","IZ"],RSO6["ZI","IZ"],"R[ZI,IZ] "},
    {R["YI","IZ"],RSO6["YI","IZ"],"R[YI,IZ] "},
    {R["ZI","IY"],RSO6["ZI","IY"],"R[ZI,IY] "},
    {R["ZI","IX"],RSO6["ZI","IX"],"R[ZI,IX] "},
    {R["XI","IZ"],RSO6["XI","IZ"],"R[XI,IZ] "},
    {R["XI","IY"],RSO6["XI","IY"],"R[XI,IY] "},
    {R["YI","IX"],RSO6["YI","IX"],"R[YI,IX] "},
    {R["XX","YY"],RSO6["XX","YY"],"R[XX,YY] "},
    {R["XX","ZY"],RSO6["XX","ZY"],"R[XX,ZY] "},
    {R["ZX","YY"],RSO6["ZX","YY"],"R[ZX,YY] "},
    {R["YX","XY"],RSO6["YX","XY"],"R[YX,XY] "},
    {R["ZX","XY"],RSO6["ZX","XY"],"R[ZX,XY] "},
    {R["YX","ZY"],RSO6["YX","ZY"],"R[YX,ZY] "}
};

SyllableListAsymmetric = {
    {H1.H2.CS,H1SO6.H2SO6.CSSO6,"H1 H2 CS "},
    {S1.H1.S2.H2.CS,S1SO6.H1SO6.S2SO6.H2SO6.CSSO6,"S1 H1 S2 H2 CS "},
    {CS,CSSO6,"CS "},
    {S1.H1.CS,S1SO6.H1SO6.CSSO6,"S1 H1 CS "},
    {S2.H2.CS,S2SO6.H2SO6.CSSO6,"S2 H2 CS "},
    {H2.CS,H2SO6.CSSO6,"H2 CS "},
    {H1.CS,H1SO6.CSSO6,"H1 CS "},
    {H1.S2.H2.CS,H1SO6.S2SO6.H2SO6.CSSO6,"H1 S2 H2 CS "},
    {S1.H1.H2.CS,S1SO6.H1SO6.H2SO6.CSSO6,"S1 H1 H2 CS "},
    {CNOT12.H1.CS,CNOT12SO6.H1SO6.CSSO6,"H2 CZ H1 H2 CS "},
    {CZ.S1.H1.S2.H2.CS,CZSO6.S1SO6.H1SO6.S2SO6.H2SO6.CSSO6,"CZ S1 H1 S2 H2 CS "},
    {CZ.H1.H2.CS,CZSO6.H1SO6.H2SO6.CSSO6,"CZ H1 H2 CS "},
    {CNOT12.S1.H1.CS,CNOT12SO6.S1SO6.H1SO6.CSSO6,"H2 CZ S1 H1 H2 CS "},
    {CZ.S1.H1.H2.CS,CZSO6.S1SO6.H1SO6.H2SO6.CSSO6,"CZ S1 H1 H2 CS "},
    {CZ.H1.S2.H2.CS,CZSO6.H1SO6.S2SO6.H2SO6.CSSO6,"CZ H1 S2 H2 CS "}
};
```

And the internal data structure:

```
SyllableListSp = MapIndexed[
    {SO6ToSpecialRep[#1[[2]]],#1[[3]],IntegerString[#2,16]}&,
    SyllableList
];
SyllableListAsymmetricSp = MapIndexed[
    {SO6ToSpecialRep[#1[[2]]],#1[[3]],IntegerString[#2,16]}&,
    SyllableListAsymmetric
];
```

Normalization

In this section we develop a function for normalizing a circuit in any of our representations. This algorithm is described in detail in [2].

Earliest Generator Ordering and associated row pairings

We develop here an association which matches up each syllable to their row pairings under EGO.

```
TwoFourPaired = Map[
    {#,Complement[Range[1,6],#]}&,
    Subsets[Range[1,6],{4}]
];
TwoTwoTwoPaired = Flatten[Map[
    Table[{#[[1]],{#[[2,1]],#[[2,k]]},Complement[#[[2]],{#[[2,1]],#[[2,k]]}]},{k,
    Table[{{1,j},Complement[Range[1,6],{1,j}]},{j,2,6}]
],1];
PairList = Map[Sort,Join[TwoTwoTwoPaired,TwoFourPaired]];

MatchingPairings[list_] := {
    list,
    {list[[3]],Union[list[[1]],list[[2]]]},
    {list[[2]],Union[list[[1]],list[[3]]]},
    {list[[1]],Union[list[[2]],list[[3]]]}
};
PossibleSyllableListPairings = Map[
    MatchingPairings[RowPairs[First[#]]]&,
    SyllableListSp
];
SyllableListPairings = Map[
    # -> First[FirstPosition[PossibleSyllableListPairings,#]]&,
    PairList
];
PairingKey = Association @@ SyllableListPairings;
```

Finding a leftmost syllable and normalizing in the Special representation

We develop separate synthesis algorithms here based on whether we want to synthesize a circuit using the symmetric or asymmetric syllables.

```
LeftmostSyllable[U_] := SyllableListSp[[PairingKey[RowPairs[U]]]];
RemoveLeftmost[{{k_,M_},str_,numberstr_}] := Module[{leftmost,clifford},
    Which[
        k > 0,
            leftmost = LeftmostSyllable[{k,M}];
            {Dot2Sp[InvSp[leftmost[[1]]],{k,M}],str<>leftmost[[2]],numberstr<>lef
        k == 0,
            clifford = CliffordSynthSp[{k,M}];
            {{-1,SparseArray[ConstantArray[0,{6,6}]]},str<>clifford[[1]]<>StringR
        True,
            {{k,M},str,numberstr}
    ]
];

LeftmostSyllableAsymmetric[U_] := SyllableListAsymmetricSp[[PairingKey[RowPairs[U
RemoveLeftmostAsymmetric[{{k_,M_},str_,numberstr_}] := Module[{leftmost,clifford}
    Which[
        k > 0,
            leftmost = LeftmostSyllableAsymmetric[{k,M}];
            {Dot2Sp[InvSp[leftmost[[1]]],{k,M}],str<>leftmost[[2]],numberstr<>lef
        k == 0,
            clifford = CliffordSynthSp[{k,M}];
            {{-1,SparseArray[ConstantArray[0,{6,6}]]},str<>clifford[[1]]<>StringR
        True,
            {{k,M},str,numberstr}
    ]
];

test[NormItSp,"SyllableType"] := MemberQ[{"Normal","Asymmetric"},#]&;

Options[NormItSp] = {"SyllableType" -> "Normal"};
NormItSp[U_,OptionsPattern[]]?optsCheck := Module[{type},
    type = If[OptionValue["SyllableType"] == "Normal",RemoveLeftmost,RemoveLeftmo
    FixedPoint[type,{U,"",If[OptionValue["SyllableType"] == "Normal","","-"]}][[2
];
```

Normalizing from a string, a hexadecimal, U(4), or SO(6).

We provide the function NormIt for normalizing operators in any of our forms.

```
Options[FixPhase] = {"UpToPhase" -> True};
FixPhase[{str_,numberstr_},U_,OptionsPattern[]] := Module[{syllablesplit,syllablen
    If[
        OptionValue["UpToPhase"] == True,
            {str,numberstr},
            syllablesplit = StringSplit[numberstr,"00000"];
            syllablenum = If[Length[syllablesplit] > 1, syllablesplit[[1]],""];
            syllablepart = FromHexDec[syllablenum<>"000001"];
            clifford = Simplify[ConjugateTranspose[syllablepart] . U];
            {cliffnum,cliffstr} = CliffordSynth[clifford];
            numberstrfixed = syllablenum<>"00000"<>cliffnum;
            nophasestr = StringDelete[str,"W "];
            phasecount = StringCount[cliffstr,"W "];
            strfixed = nophasestr<>StringRepeat["W ",phasecount];
            {strfixed,numberstrfixed}
    ]
];

test[NormIt,"SyllableType"] := MemberQ[{"Normal","Asymmetric"},#]&;
test[NormIt,"OutputType"] := MemberQ[{"String","HexDec"},#]&;
test[NormIt,"UpToPhase"] := BooleanQ;

Options[NormIt] = {"OutputType" -> "String","SyllableType" -> "Normal","UpToPhase'
NormIt[U_/;(GaussianQ[U] && U4Q[U]),OptionsPattern[]]?optsCheck := Module[{index}
    index = If[OptionValue["OutputType"] == "String",1,2];
    FixPhase[NormItSp[SO6ToSpecialRep[U4ToSO6[U]],"SyllableType" -> OptionValue["
];
NormIt[U_/;(GaussianQ[U] && SO6Q[U]),OptionsPattern[]]?optsCheck := Module[{index
    index = If[OptionValue["OutputType"] == "String",1,2];
    NormItSp[SO6ToSpecialRep[U],"SyllableType" -> OptionValue["SyllableType"]][[i
];
NormIt[hexdec_String/;ValidHexDecQ[hexdec],opts:OptionsPattern[]] := NormIt[FromHe
NormIt[str_String,opts:OptionsPattern[]] := NormIt[FromSequence[str],opts];
NormIt[U_] := (
    Message[NormIt::notacircuit,U];
    $Failed
);
```

ϵ-Approximating Pauli Rotations

This section describes algorithms used for finding approximations to Pauli Rotations

Frobenius Distance of Matrices

Computes the Frobenius distance between two matrices A and B (i.e. the Frobenius norm of the difference between A and B)

```
FrobeniusDistance[A_/;MatrixQ[A],B_/;MatrixQ[B]]/;(Dimensions[A] == Dimensions[B]
FrobeniusDistance[_,_] := (
    Message[FrobeniusDistance::notequidimensionalmatrices];
    $Failed
);
```

Continued fractions and affine transformation

We describe a scheme for finding an appropriate affine transformation based off of continued fractions in the rationals.

```
NextIterate[{aN_,rN_,pN_,qN_,pNm1_,qNm1_}] := Module[{aNp1},
    aNp1 = IntegerPart[1/rN];
    {aNp1,1/rN - aNp1,aNp1*pN + pNm1,aNp1*qN + qNm1,pN,qN}
];
ContinuedFractionToPrecision[α_,tol_] := Module[{a0,iterate},
    a0 = IntegerPart[α];
    iterate = NestWhile[NextIterate,{a0,N[α - a0,2*Log10[1/tol]],a0,1,1,0},#[[4]]
    If[iterate[[2]] == 0 && iterate[[4]] <= 1/tol,
        iterate[[3;;4]],
        iterate[[5;;6]]
    ]
];

AffineMatrix[φ_,ϵ_] := Module[{α,q,r,s,t},
    α = Tan[φ/2];
    {r,q} = ContinuedFractionToPrecision[α,Sqrt[ϵ/(8+ϵ*α)]*Sec[φ/2]];
    {t,s} = ExtendedGCD[q,-r][[2]];
    {{q,r},{s,t}}
];
```

Finding angle candidates for bounded and unbounded angles

For calculating a candidate solution, we first map every angle into the interval $[0,\pi/2]$. We then use our affine transformation to find solutions to the integer programming problem.

```
RealQ[x_] := Element[x,Reals];

CandidateFinder[φ_?RealQ,ϵ_?RealQ] := Module[{modφ},
    modφ=Mod[φ,4*Pi];
    Which[
```

```
        0<=modφ<=Pi/2,
            CandidateFinderBounded[modφ,∈],
        Pi/2<modφ<=Pi,
            MapAt[Reverse,CandidateFinderBounded[Pi-modφ,∈],2],
        Pi<modφ<=2*Pi,
            MapAt[{-#[[2]],#[[1]]}&,CandidateFinder[modφ-Pi,∈],2],
        True,
            MapAt[{-#[[1]],-#[[2]]}&,CandidateFinder[modφ-2*Pi,∈],2]
    ]
];
CandidateFinder[_,_] := (
    Message[CandidateFinder::notreals];
    $Failed
);

CandidateFinderBounded[φ_,∈_] := Module[{root2k,c,s,α,p,Δ,δ,A,invA,scaledp′,scalec
    root2k = 1;
    c = Cos[φ/2];
    s = Sin[φ/2];
    α = 1-∈^2/8;
    p = {c*α+s*Sqrt[1-α^2],s*α-c*Sqrt[1-α^2]};
    Δ = 2*Sqrt[1-α^2]*{-s,c};
    δ = ∈^2/8*{c,s};
    A = AffineMatrix[φ,∈];
    invA = Inverse[A];
    scaledp′ = A.p;
    scaledΔ′ = A.Δ;
    scaledδ′ = A.δ;
    θΔ′ = VectorAngle[scaledΔ′,{1,0}];
    Which[
        θΔ′ > Pi/2,
            m1 = scaledΔ′[[2]] / scaledΔ′[[1]];
            m2 = scaledδ′[[2]] / scaledδ′[[1]];
            valid = Catch[Do[
                {x1,x2,x3,x4} = {
                    Ceiling[scaledp′[[1]]+scaledΔ′[[1]]],
                    Floor[scaledp′[[1]]],
                    Floor[scaledp′[[1]]+scaledΔ′[[1]]+scaledδ′[[1]]],
                    Floor[scaledp′[[1]]+scaledδ′[[1]]]
                };

                Do[
                    y0 = If[x <= x2,
```

```
                    Ceiling[m1*(x - scaledp′[[1]]) + scaledp′[[2]]],
                    Ceiling[m2*(x - scaledp′[[1]]) + scaledp′[[2]]]
                ];
                y1 = If[x <= x3,
                    Floor[m2*(x - scaledp′[[1]]-scaled△′[[1]]) + scaledp′[[2]]
                    Floor[m1*(x - scaledp′[[1]]-scaledδ′[[1]]) + scaledp′[[2]]
                ];
                Do[
                    transformed = invA.{x,y};
                    If[Norm[transformed] <= Sqrt[2]^k,Throw[{k,transformed}]]
                    {y,y0,y1}
                ];,
                {x,x1,x4}
            ];
            {scaledp′,scaled△′,scaledδ′} *= Sqrt[2];,
            {k,0,Ceiling[4*Log2[1/ϵ]+6]}
        ];];,
    θ△′ == Pi/2,
        m1 = scaledδ′[[2]] / scaledδ′[[1]];
        valid = Catch[Do[
            {x1,x2} = {
                Ceiling[scaledp′[[1]]],
                Floor[scaledp′[[1]]+scaledδ′[[1]]]
            };
            Do[
                y0 = Ceiling[m1*(x - scaledp′[[1]]) + scaledp′[[2]]];
                y1 = Floor[m1*(x - scaledp′[[1]]-scaled△′[[1]]) + scaledp′[[2]
                Do[
                    transformed = invA.{x,y};
                    If[Norm[transformed] <= Sqrt[2]^k,Throw[{k,transformed}]]
                    {y,Floor[(y0+y1)/2],y1}
                ];,
                {x,x1,x2}
            ];
            {scaledp′,scaled△′,scaledδ′} *= Sqrt[2];,
            {k,0,Ceiling[4*Log2[1/ϵ]+6]}
        ];];,
    True,
        m1 = scaledδ′[[2]] / scaledδ′[[1]];
        m2 = scaled△′[[2]] / scaled△′[[1]];
        valid = Catch[Do[
            {x1,x2,x3,x4} = {
                Ceiling[scaledp′[[1]]],
```

```
                    Floor[scaledp′[[1]] + scaledδ′[[1]]],
                    Floor[scaledp′[[1]] + scaledΔ′[[1]]],
                    Floor[scaledp′[[1]] + scaledδ′[[1]] + scaledΔ′[[1]]]
                };
                Do[
                    y0 = If[x <= x2,
                        Ceiling[m1*(x - scaledp′[[1]]) + scaledp′[[2]]],
                        Ceiling[m2*(x - scaledp′[[1]] - scaledδ′[[1]]) + scaledp′[
                    ];
                    y1 = If[x <= x3,
                        Floor[m2*(x - scaledp′[[1]]) + scaledp′[[2]]],
                        Floor[m1*(x - scaledp′[[1]] - scaledΔ′[[1]]) + scaledp′[[2
                    ];
                    Do[
                        transformed = invA.{x,y};
                        If[Norm[transformed] <= Sqrt[2]^k,Throw[{k,transformed}]]
                        {y,y0,y1}
                    ];,
                    {x,x1,x4}
                ];
                {scaledp′,scaledΔ′,scaledδ′} *= Sqrt[2];,
                {k,0,Ceiling[4*Log2[1/ϵ]+6]}
            ];];
    ];
    valid
];
```

Solving Lagrange Four-Squares

Rather than implement our own solver based off of well known algorithms, we instead use a basic Mathematica function as the inputs for this problem never get too big.

```
Lagrange4[n_]:=Module[{x1,x2,x3,x4},({x1,x2,x3,x4}/.FindInstance[x1^2+x2^2+x3^2+x
```

SU(4) Approximations Using a Candidate Solution

We provide an algorithm for finding a rotation by angle φ up to error ϵ for any of the 15 Pauli matrices.

```
SU4Z1Finder[φ_,ϵ_]:=Module[{k,x,y,a,b,c,d,Invroot2k,α,β,χ},
    {k,{x,y}} = CandidateFinder[φ,ϵ];
    {a,b,c,d} = Lagrange4[2^k-x^2-y^2];
    Invroot2k = 1/Sqrt[2]^k;
    α = Invroot2k*(x+I*y);
    β = Invroot2k*(a+I*b);
```

```
    χ = Invroot2k*(c+I*d);
    {
        {α,0,-Conjugate[β],-Conjugate[χ]},
        {0,α,χ,-β},
        {β,-Conjugate[χ],Conjugate[α],0},
        {χ,Conjugate[β],0,Conjugate[α]}
    }
];

PauliRotation[φ_/;Not[RealQ[φ]],ϵ_/;Not[RealQ[ϵ]],_] := (
    Message[PauliRotation::notreals];
    $Failed
);
PauliRotation[φ_,ϵ_,"ZI"] := SU4Z1Finder[φ,ϵ];
PauliRotation[φ_,ϵ_,"XI"] := H1 . SU4Z1Finder[φ,ϵ] . H1;
PauliRotation[φ_,ϵ_,"YI"] := S1 . H1 . SU4Z1Finder[φ,ϵ] . H1 . ConjugateTranspose
PauliRotation[φ_,ϵ_,"IZ"] := EX . SU4Z1Finder[φ,ϵ] . EX;
PauliRotation[φ_,ϵ_,"IX"] := EX . H1 . SU4Z1Finder[φ,ϵ] . H1 . EX;
PauliRotation[φ_,ϵ_,"IY"] := EX . S1 . H1 . SU4Z1Finder[φ,ϵ] . H1 . ConjugateTrans
PauliRotation[φ_,ϵ_,"ZZ"] := CNOT21 . SU4Z1Finder[φ,ϵ] . CNOT21;
PauliRotation[φ_,ϵ_,"XZ"] := H1 . CNOT21 . SU4Z1Finder[φ,ϵ] . CNOT21 . H1;
PauliRotation[φ_,ϵ_,"YZ"] := S1 . H1 . CNOT21 . SU4Z1Finder[φ,ϵ] . CNOT21 . H1 . (
PauliRotation[φ_,ϵ_,"ZX"] := H2 . CNOT21 . SU4Z1Finder[φ,ϵ] . CNOT21 . H2;
PauliRotation[φ_,ϵ_,"XX"] := H1 . H2 . CNOT21 . SU4Z1Finder[φ,ϵ] . CNOT21 . H2 . I
PauliRotation[φ_,ϵ_,"YX"] := S1 . H1 . H2 . CNOT21 . SU4Z1Finder[φ,ϵ] . CNOT21 . I
PauliRotation[φ_,ϵ_,"ZY"] := S2 . H2 . CNOT21 . SU4Z1Finder[φ,ϵ] . CNOT21 . H2 . (
PauliRotation[φ_,ϵ_,"XY"] := H1 . S2 . H2 . CNOT21 . SU4Z1Finder[φ,ϵ] . CNOT21 . I
PauliRotation[φ_,ϵ_,"YY"] := S1 . H1 . S2 . H2 . CNOT21 . SU4Z1Finder[φ,ϵ] . CNOT:
PauliRotation[_,_,_] := (
    Message[PauliRotation::invldstring];
    $Failed
);

test[PauliRotationSequence,"SyllableType"] := MemberQ[{"Normal","Asymmetric"},#]&
test[PauliRotationSequence,"OutputType"] := MemberQ[{"String","HexDec"},#]&;

Options[PauliRotationSequence] = {"OutputType" -> "String","SyllableType" -> "Norr

PauliRotationSequence[φ_,ϵ_,pauli_,opts:OptionsPattern[]]?optsCheck := NormIt[Pau
```

General Unitary Decomposition

Here we develop an algorithm for the approximation of any U(4) matrix using the Clifford + CS gate set.

Angles of rotation in the Pauli decomposition

First, we develop a method for finding the Pauli decomposition of an operator in terms of its 15 Pauli-rotation angles.

```
SafeArcTan[a_,b_] := If[(a == 0) && (b == 0),0,ArcTan[a,b]];
AngleFind[{{a_,_},{b_,_}}] := SafeArcTan[a,b];
SmallPauliDecomposition[o_] := Module[{v1,v2,v3,φ1,φ2,φ3,M1,M2,o′,o′′},
    v1 = o[[1;;2,3]];
    φ1 = -SafeArcTan @@ Reverse[v1];
    M1 = {{Cos[φ1],Sin[φ1],0},{-Sin[φ1],Cos[φ1],0},{0,0,1}};
    o′ = Simplify[M1 . o];
    v2 = o′[[2;;3,3]];
    φ2 = -SafeArcTan @@ Reverse[v2];
    M2 = {{1,0,0},{0,Cos[φ2],Sin[φ2]},{0,-Sin[φ2],Cos[φ2]}};
    o′′ = Simplify[M2 . o′];
    v3 = o′′[[1;;2,1]];
    φ3 = SafeArcTan @@ v3;
    {φ1,φ2,φ3}
];

PauliDecomposition[U_?U4Q] := PauliDecomposition @ U4ToSO6 @ U
PauliDecomposition[U_?SO6Q] := Module[{a,b,c,d,U1a′,Da,U2a′,U1d′,Dd,U2d′,U1a,U2a,U1
    a = U[[1;;3,1;;3]];
    b = U[[4;;6,1;;3]];
    c = U[[1;;3,4;;6]];
    d = U[[4;;6,4;;6]];
    {U1a′,Da,U2a′} = SingularValueDecomposition[a];
    {U1d′,Dd,U2d′} = SingularValueDecomposition[d];
    {U1a,U2a,U1d,U2d} = {
        FullSimplify[Det[U1a′]] * U1a′,
        FullSimplify[Det[U2a′]] * U2a′,
        FullSimplify[Det[U1d′]] * U1d′,
        FullSimplify[Det[U2d′]] * U2d′
    };
    Db = Simplify[Transpose[U1d] . b . U2a];
    Dc = Simplify[Transpose[U1a] . c . U2d];
    blocks = Map[{{Da[[#,#]],Db[[#,#]]},{Dc[[#,#]],Dd[[#,#]]}}&,Range[1,3]];
    {φXX,φYY,φZZ} = -Map[AngleFind,blocks];
    {φZI1,φXI1,φZI2} = -SmallPauliDecomposition[U1a];
    {φZI3,φXI2,φZI4} = -SmallPauliDecomposition[Transpose[U2a]];
    {φIZ1,φIX1,φIZ2} = -SmallPauliDecomposition[U1d];
    {φIZ3,φIX2,φIZ4} = -SmallPauliDecomposition[Transpose[U2d]];
    {
```

```
        {φZI1,"ZI"},{φXI1,"XI"},{φZI2,"ZI"},
        {φIZ1,"IZ"},{φIX1,"IX"},{φIZ2,"IZ"},
        {φXX,"XX"},{φYY,"YY"},{φZZ,"ZZ"},
        {φZI3,"ZI"},{φXI2,"XI"},{φZI4,"ZI"},
        {φIZ3,"IZ"},{φIX2,"IX"},{φIZ4,"IZ"}
    }
];
PauliDecomposition[_] := (
    Message[PauliDecomposition::notanoperator];
    $Failed
);
```

Reconstruction using Pauli-rotation angles

```
ApproximateOp[_,ϵ_/;Not[RealQ[ϵ]]] := (
    Message[Approximate::notreal];
    $Failed
);
ApproximateOp[U_?U4Q,ϵ_] := Module[{anglesaxes,pauliapproximations},
    anglesaxes = PauliDecomposition[U];
    pauliapproximations = Map[PauliRotation[#[[1]],ϵ/15,#[[2]]]&,anglesaxes];
    Simplify[Dot @@ pauliapproximations]
];
ApproximateOp[U_?SO6Q,ϵ_] := Module[{anglesaxes,pauliapproximations},
    anglesaxes = PauliDecomposition[U];
    pauliapproximations = Map[PauliRotation[#[[1]],ϵ/15,#[[2]]]&,anglesaxes];
    U4ToSO6 @ Simplify @ Dot @@ pauliapproximations
];
ApproximateOp[_,_] := (
    Message[Approximate::notanoperator];
    $Failed
);

test[ApproximateSequence,"IfGaussianDoExact"] := BooleanQ;
test[ApproximateSequence,"SyllableType"] := MemberQ[{"Normal","Asymmetric"},#]&;
test[ApproximateSequence,"OutputType"] := MemberQ[{"String","HexDec"},#]&;

Options[ApproximateSequence] = {"IfGaussianDoExact" -> True,"OutputType" -> "Stri

ApproximateSequence[_,ϵ_/;Not[RealQ[ϵ]],OptionsPattern[]] := (
    Message[ApproximateSequence::notreal];
    $Failed
);
ApproximateSequence[U_,ϵ_,opts:OptionsPattern[]] := Module[{normitops},
    normitops = FilterRules[{opts},Options[NormIt]];
    If[
        OptionValue["IfGaussianDoExact"] && GaussianQ[U],
            NormIt[U,Append[normitops,"UpToPhase"->False]],
            NormIt[ApproximateOp[U,ϵ],normitops]
    ]
];
```

Pseudorandom Circuit Generator

In this section, we develop a pseudo-random Clifford + CS circuit generator using our unique normal form. These circuits are random in the sense that given some integer n, we pseudorandomly sample uniformly from the set of all Clifford + CS operators whose optimal CS-count is at most n. In the limit that n goes to infinity, this is uniformly random. Note that the overwhelming majority of operators in this set have an optimal CS-count around n.

Adjacency Matrices of Automata Graphs

The adjacency matrices of the graphs corresponding to the automata in [2].

```
adjacency3 = SparseArray[{
    {0,1,1},
    {1,0,1},
    {1,1,0}
}];
adjacency9 = SparseArray[{
    {0,1,1,1,1,0,0,0,0},
    {1,0,1,0,0,1,1,0,0},
    {1,1,0,0,0,0,0,1,1},
    {1,0,0,0,1,1,0,1,0},
    {1,0,0,1,0,0,1,0,1},
    {0,1,0,1,0,0,1,1,0},
    {0,1,0,0,1,1,0,0,1},
    {0,0,1,1,0,1,0,0,1},
    {0,0,1,0,1,0,1,1,0}
}];
adjacency15 = SparseArray[{
    {0,1,1,1,1,0,0,0,0,0,0,1,1,1,1},
    {1,0,1,0,0,1,1,0,0,0,1,0,1,1,1},
    {1,1,0,0,0,0,0,1,1,0,1,1,0,1,1},
    {1,0,0,0,1,1,0,1,0,1,0,1,1,0,1},
    {1,0,0,1,0,0,1,0,1,1,0,1,1,1,0},
    {0,1,0,1,0,0,1,1,0,1,1,0,1,0,1},
    {0,1,0,0,1,1,0,0,1,1,1,0,1,1,0},
    {0,0,1,1,0,1,0,0,1,1,1,1,0,0,1},
    {0,0,1,0,1,0,1,1,0,1,1,1,0,1,0},
    {0,0,0,1,1,1,1,1,1,0,0,0,0,1,1},
    {0,1,1,0,0,1,1,1,1,0,0,1,1,0,0},
    {1,0,1,1,1,0,0,1,1,0,1,0,1,0,0},
    {1,1,0,1,1,1,1,0,0,0,1,1,0,0,0},
    {1,1,1,0,1,0,1,0,1,1,0,0,0,0,1},
    {1,1,1,1,0,1,0,1,0,1,0,0,0,1,0}
}];
```

Next Vertex Keys

Association keys which make random walks on the automata graphs easy to compute.

```
S3NextVertexList = GatherBy[adjacency3["NonzeroPositions"],First][[All,All,2]];
S3VertexKey = Association @@ MapIndexed[First[#2] -> #1&,S3NextVertexList];

S9NextVertexList = GatherBy[adjacency9["NonzeroPositions"],First][[All,All,2]];
S9VertexKey = Association @@ MapIndexed[First[#2] -> #1&,S9NextVertexList];

S15NextVertexList = GatherBy[adjacency15["NonzeroPositions"],First][[All,All,2]];
S15VertexKey = Association @@ MapIndexed[First[#2] -> #1&,S15NextVertexList];
```

Random Walk Vertex Lists

Functions to generate a list of vertices for random walks of a given length on our automata.

```
S13Random[0] = {};
S13Random[n_] := Module[{seed,tail,sylnumlist},
    seed = RandomInteger[{1,3},1];
    tail = RandomInteger[{1,2},n-1];
    sylnumlist = Fold[Append[#1,S3VertexKey[#1[[-1]]][[#2]]]&,seed,tail];
    sylnumlist
];

S49Random[0] = {};
S49Random[n_] := Module[{seed,tail,sylnumlist},
    seed = RandomInteger[{4,9},1];
    tail = RandomInteger[{1,4},n-1];
    sylnumlist = Fold[Append[#1,S9VertexKey[#1[[-1]]][[#2]]]&,seed,tail];
    sylnumlist
];

S1015Random[0] = {};
S1015Random[n_] := Module[{seed,tail,sylnumlist},
    seed = RandomInteger[{10,15},1];
    tail = RandomInteger[{1,8},n-1];
    sylnumlist = Fold[Append[#1,S15VertexKey[#1[[-1]]][[#2]]]&,seed,tail];
    sylnumlist
];
```

Sampling from the probability distribution of sub-walk lengths given a total walk length

Because Mathematica doesn't support sampling form most custom multivariate probability density functions, we use the marginal probability for sub-walks having a certain length given that the total walk length is at most n.

```
marginalpdfk[n_,k_] := (2/3)^(DiscreteDelta[k])*3*2^k*(15*8^(n-k)-7*4^(n-k)-1)/(4
marginalpdfl[n_,k_,l_] := (2/3)^(DiscreteDelta[l])*3/2*4^l*(6*8^(n-k-l)+1)/(15*8^
marginalpdfm[n_,k_,l_,m_]:= (4/3)^(DiscreteDelta[m])*21/4*8^m/(6*8^(n-k-l)+1);
distributionk[n_] := ProbabilityDistribution[marginalpdfk[n,k],{k,0,n,1}];
distributionl[n_,k_] := ProbabilityDistribution[marginalpdfl[n,k,l],{l,0,n-k,1}];
distributionm[n_,k_,l_] := ProbabilityDistribution[marginalpdfm[n,k,l,m],{m,0,n-k

randklm[n_] := Module[{k,l,m},
    k = RandomVariate[distributionk[n]];
    l = RandomVariate[distributionl[n,k]];
    m = RandomVariate[distributionm[n,k,l]];
    {k,l,m}
];
```

Pseudorandom Circuits

Using our pseudorandom walk-length generator, our pseudorandom walk generator, our syllable list, a random Clifford generator, and our unique normal form from [2], we develop a function to generate uniformly pseudorandom Clifford + CS unitaries whose optimal CS-count is at most some integer n.

```
randsylnumlist[n_] := Module[{k,l,m},
    {k,l,m} = randklm[n];
    Join[S13Random[k],S49Random[l],S1015Random[m]]
];

RandomCliffCS[n_/;(IntegerQ[n] && n > 0)] := Module[{sylnumlist,cliffnum,Rmult,cl
    sylnumlist = randsylnumlist[n];
    cliffnum = RandomInteger[{1,92160}];
    Rmult = Fold[Dot[#1,SyllableList[[#2,1]]]&,Id,sylnumlist];
    clifford = CliffordFromNumber[cliffnum];
    Rmult.clifford
];
RandomCliffCS[_] := (
    Message[RandomCliffCS::notanatural];
    $Failed
);
```

End of functions in the private context

```
End[];
```

Exported Functions

```
Protect[Id,X1,X2,Z1,Z2,W,H1,H2,S1,S2,CZ,CNOT12,CNOT21,EX,CS,R,U4ToSO6,
    IdSO6,X1SO6,X2SO6,Z1SO6,Z2SO6,ISO6,H1SO6,H2SO6,S1SO6,S2SO6,CZSO6,CNOT12SO6,CN
    FromSequence,FromHexDec,CliffordQ,CliffordSynth,RightCliffordSimilar,LeftClif
    CandidateFinder,PauliRotation,PauliRotationSequence,PauliDecomposition,Approx
```

```
EndPackage[];
```