Searching for Minimal Universal Classical and Quantum Gates

Theoretical approaches to representing discrete operators as linear state space transformations, a proof of classical universality, and a proposed smallest 2→2 universal quantum gate set.

Introduction

Universal Classical Gates

Classical logic uses logical operators, called gates, that transform one or more boolean-valued inputs to one or more boolean-valued outputs. For example, the AND gate returns True only when both of it's inputs are True. AND is of the form 2→1, because it takes in two inputs and returns a single output.
An operation of two boolean values can be represented by a truth table, with each axis representing each input.

Implementation

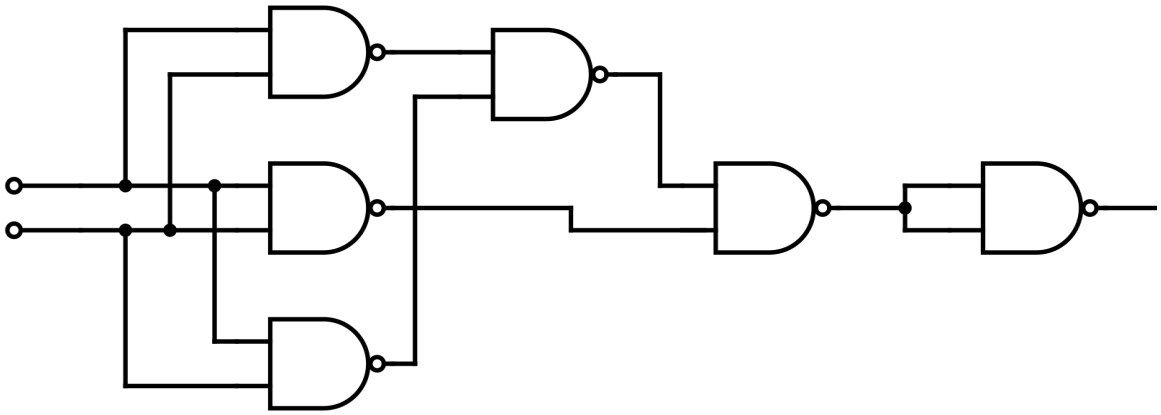
Example

In[•]:=
```
truthTable[And]
```

Out[•]=

And	0	1
0	0	0
1	0	1

Some gates are able to be composed with themselves to make any other possible gate. These gates are called universal gate.
There are only two universal, classical, 2→1 gates: NAND ($\overline{\wedge}$) and NOR ($\overline{\vee}$). For example, here is how the NAND gate can represent the XOR gate with a circuit diagram and code:



In[•]:=
```
(* Built-In XOR *)
xor1[a_, b_] := Xor[a, b];

(* XOR from NAND gates *)
xor2[a_, b_] := (((a ⊼ a) ⊼ (b ⊼ b)) ⊼ (a ⊼ b)) ⊼ (((a ⊼ a) ⊼ (b ⊼ b)) ⊼ (a ⊼ b));

(* Display Truth Tables *)
Row[{truthTable[xor1], Spacer[10], truthTable[xor2]}]
```

Out[•]=

xor1	0	1
0	0	1
1	1	0

xor2	0	1
0	0	1
1	1	0

Here is every gate represented with the built-in function, with only NANDs, and with only NORs:

Universal Function Definitions

Universal NAND Visualization

In[•]:=
```
Grid[
 Transpose@
  (truthTable[#] & /@ # & /@ {{and1, and2, and3}, {or1, or2, or3}, {xor1, xor2, xor3},
     {nand1, nand2, nand3}, {nor1, nor2, nor3}, {xnor1, xnor2, xnor3}}),
 Spacings → {3, 2}]
```

Out[•]=

and1	0	1
0	0	0
1	0	1

or1	0	1
0	0	1
1	1	1

xor1	0	1
0	0	1
1	1	0

nand1	0	1
0	1	1
1	1	0

nor1	0	1
0	1	0
1	0	0

xnor1	0	1
0	1	0
1	0	1

and2	0	1
0	0	0
1	0	1

or2	0	1
0	0	1
1	1	1

xor2	0	1
0	0	1
1	1	0

nand2	0	1
0	1	1
1	1	0

nor2	0	1
0	1	0
1	0	0

xnor2	0	1
0	1	0
1	0	1

and3	0	1
0	0	0
1	0	1

or3	0	1
0	0	1
1	1	1

xor3	0	1
0	0	1
1	1	0

nand3	0	1
0	1	1
1	1	0

nor3	0	1
0	1	0
1	0	0

xnor3	0	1
0	1	0
1	0	1

State Vector Representation of Operations

Classical 2→1 gates are usually considered to be a function with two inputs and one output. However, there is an alternative interpretation of gates as a linear operation on a unit basis vector representing the systems total state. This allows for all operations to be represented as matrix multiplications, and generalizes easily to quantum circuits.

State Vector

- Let $\mathbb{B}$ be the set of binary numbers $\{0, 1\}$.

An n-bit system can be represented with a vector in $\mathbb{B}^n$, such that each dimension of the vector corresponds to a bit in the state. A gate can be considered to be an association of each input state—of the total 2^n possible states—to a specific output state.
A vector can be constructed in $\mathbb{B}^{2^n}$ so that each dimension corresponds to a possible state. Thus, a unit basis vector of $\mathbb{B}^{2^n}$ represents one singular state. A matrix transforms the ith basis vector e_i to the ith column in the matrix V_i.

For example, in the 4-bit binary system 1011, there are $2^4 = 16$ possible states, so the 16-dimensional state vector representing this system is [0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0].
Note that the state vector of the binary number n (11 in this example) is the basis vector e_n, assuming the vector is indexed from 0.

Gate Application to a State Vector

A matrix M = [V_0 V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8 V_9 V_{10} V_{11} V_{12} V_{13} V_{14} V_{15}], where $V_i \in \mathbb{B}^{2^n}$, takes the ith basis vector to V_i (i.e., $M \bullet e_i = V_i$).

This means that M can associate each basis vector e_i to a specific output vector V_i, so, by our definition of a gate operation, this matrix can apply any arbitrary gate.

Motivation

The reason for representing gates as matrix multiplication is two-fold:
Firstly, in this representation, all gates are linear transformations, which can be easily composed and reasoned about. A whole circuit, made up of many gate operations, can be simplified to just a single matrix. Thus, the problem of determining whether a set of gates is universal is equivalent to the problem of whether every gate matrix can be decomposed into a product of elements in that set.
Secondly, quantum systems exhibit phenomena such as superposition and entanglement, in which a quantum state may be in a linear combination of multiple states at the same time, and in which a full state can not be represented as the tensor product of individual qubit states, respectively. This means that an n-bit quantum state can be represented by a vector on the complex unit hypersphere of $\mathbb{C}^{2^n}$.
The reason it must be on the unit hypersphere is because the magnitude of the vector must be equal to one, which is equivalent to the statement that the sum of the probabilities of the states of a random system is one.

Universal Quantum Gate Sets

Universal quantum gates should be able to recreate any possible other quantum gate. However, due to the fact that quantum state vectors are complex, there is a continuous space of possible gates, which cannot be described exactly by a finite universal set. For example, the arbitrary Z rotation gate $\begin{pmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{pmatrix}$ is a valid quantum gate for any choice of θ.
Instead, a universal quantum gate set is defined as one which can approximate any quantum gate to within some error factor ε. Formally, in order to represent a target gate T in terms of the finite gate set S_G,
$\| T - M_1 M_2 \ldots M_f \| \leq \varepsilon$ where $M_i \in S_G$, for some appropriate matrix norm, such as the maximum L2 norm of the columns.

Quantum Computer Simulator Implementation

In order to explore quantum logic gates and quantum circuits more effectively, I wrote a simulation of a quantum computer which can run any arbitrary quantum circuit, though it is of exponential complexity with respect to the number of wires, even for circuits which theoretically can be simulated efficiently.

Boolean State Vectors

Booleans

Here are a variety of helper functions which convert between bools (b), integers (i), and state vectors (v):

In[21]:=
```
b2i[b_] := If[b, 1, 0]
b2i[b_List] := b2i/@b
i2b[b_Integer] := b==1;
i2b[b_List] := i2b/@b

b2v[b_] := If[b, {0, 1}, {1, 0}]
i2v[b_Integer] := If[b==1, {0, 1}, {1, 0}]
v2i[{v1_Integer, v2_Integer}] := v1*1+v2*0
```

Tensors and State Construction

A state containing multiple bits is constructed by taking the tensor product of each consecutive bit's state. tensor takes the tensor product of vectors, and fullState constructs the state vector of a list of boolean values.

In[28]:=
```
tensor[v__] := Flatten[TensorProduct[v]]
fullState[b_List] := tensor@@i2v/@b
```

Matrix Definitions

Each matrix representing an operation has a different form. For example, the NOT matrix is {{0, 1}, {1, 0}}.
Some matrices are constant and worked out by hand beforehand.
Other matrix operations may need arguments, and so has to be computed. This is done by computing the desired output for each input, then representing the inputs and outputs as state vectors, then constructing the M matrix from each output V_i.

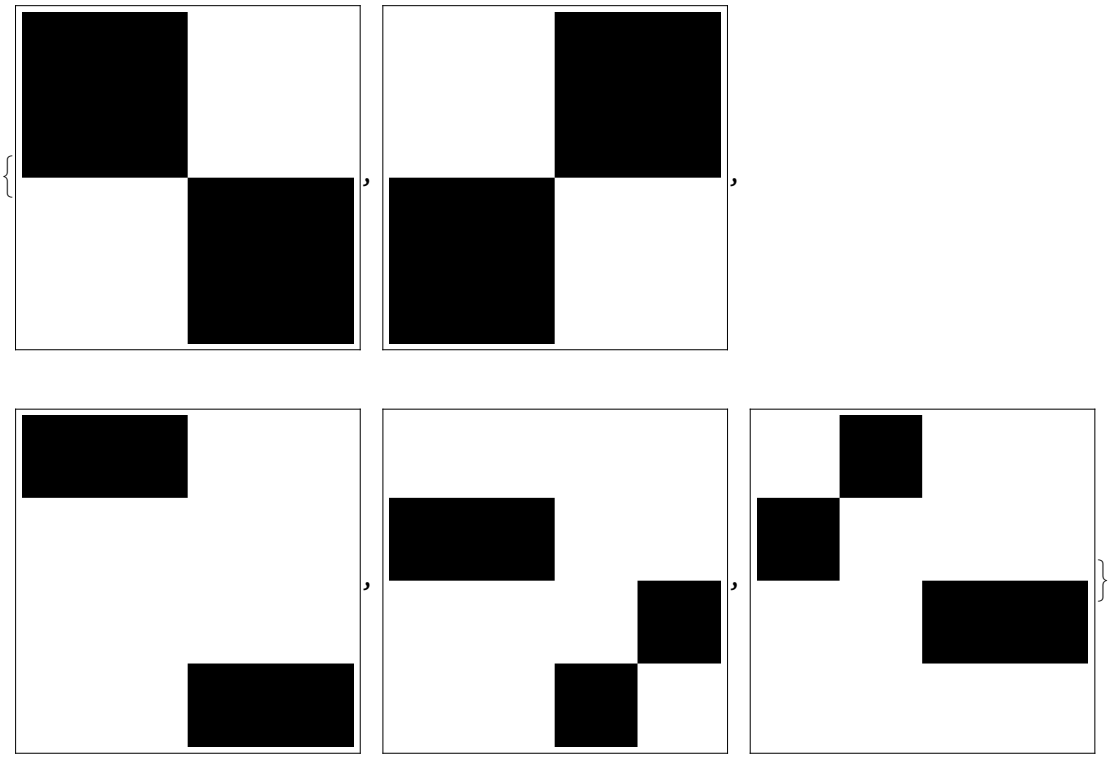
Constant Matrices

In[30]:=
```
I2 = {{1, 0}, {0, 1}}; (* Identity Matrix *)

matrixNot = {{0, 1}, {1, 0}};

matrixCopy = {{1, 1, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 1, 1}}; (* Only possible in classical computation due to the No Cloning Theorem *)

matrixNand = {{0, 0, 0, 0}, {1, 1, 0, 0}, {0, 0, 0, 1}, {0, 0, 1, 0}};

matrixNor= {{0, 1, 0, 0}, {1, 0, 0, 0}, {0, 0, 1, 1}, {0, 0, 0, 0}};
```

Here is a visualization of each of those matrices:

In[•]:=
```
ArrayPlot /@ {I2, matrixNot, matrixCopy, matrixNand, matrixNor}
```

Out[•]=



Computed Matrices

Swap switches two bits at positions s1 and s2.

In[35]:=
```
(* Generates the ith column of SWAP *)
ViSWAP[i_, n_, s1_, s2_] := Module[{bv, sbv},
 bv=IntegerDigits[i, 2, n];
 sbv=fullState[Table[bv⟦Switch[j, s1, s2, s2, s1, _, j]⟧, {j,n}]]]

(* Generates SWAP column-wise *)
matrixSwap[n_, s1_, s2_]:=Table[ViSWAP[i, n, s1, s2], {i, 0, 2^n-1}]
```

Return reduces the number of bits, thereby creating a smaller subset of the original bits to use as an output.

In[37]:=
```
(* Generates the ith column of Return *)
ViReturn[i_, n_, s1_, s2_]:=tensor@@i2v/@(IntegerDigits[i, 2, n])⟦Span[s1, s2]⟧

(* Generates Return column-wise *)
matrixReturn[n_, s1_, s2_]:=Transpose[ViReturn[#, n, s1, s2]&/@Range[0, 2^n-1]]
```

For example, here is how a state vector of length $2^5 = 32$ is reduced to a single bit state vector:

In[•]:=
```
matrixReturn[5, 1, 1] // MatrixForm
```

Out[•]//MatrixForm=

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

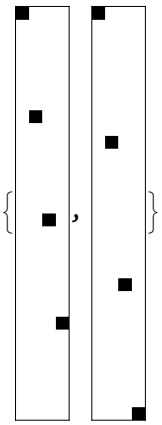
Expand increases the number of bits, initializing the circuit to a particular state, either of the form
matrixZeroExpand[a, b] → |ab0000 ...⟩
or
matrixRepeatExpand[a, b] → |ababab ...⟩.

In[39]:=
```
matrixZeroExpand[n_, s1_, s2_]:=Module[{ω1, ω2, Ω1, Ω2},
 ω1=s1-1; Ω1=2^ω1; (* ω1 is the number of bits of the input, Ω1 is the size of the state vector c
 ω2=s2; Ω2=2^ω2;   (* ω2 is the number of bits of the output, Ω2 is the size of the state vector
 Table[Module[{Vω1, Vω2, VΩ2},
  Vω1=IntegerDigits[i, 2, ω1];
  Vω2=Join[Vω1, ConstantArray[0, ω2-ω1]];
  VΩ2=fullState[Vω2];
  VΩ2], {i, 0, Ω1-1}]]//Transpose

matrixRepeatExpand[n_, s1_, s2_]:=Module[{ω1, ω2, Ω1, Ω2},
 ω1=s1-1; Ω1=2^ω1; (* ω1 is the number of bits of the input, Ω1 is the size of the state vector c
 ω2=s2; Ω2=2^ω2;   (* ω2 is the number of bits of the output, Ω2 is the size of the state vector
 Table[Module[{Vω1, Vω2, VΩ2},
  Vω1=IntegerDigits[i, 2, ω1];
  Vω2=Flatten[ConstantArray[Vω1, ⌈ω2/ω1⌉]][[;;ω2]];
  VΩ2=fullState[Vω2];
  VΩ2], {i, 0, Ω1-1}]]//Transpose
```

For example, here are the expansion matrices increasing a systems wire count from 2 to 5

In[•]:=
```
ArrayPlot /@ {matrixZeroExpand[5, 3, 5], matrixRepeatExpand[5, 3, 5]}
```

Out[•]=



Gates to Matrices

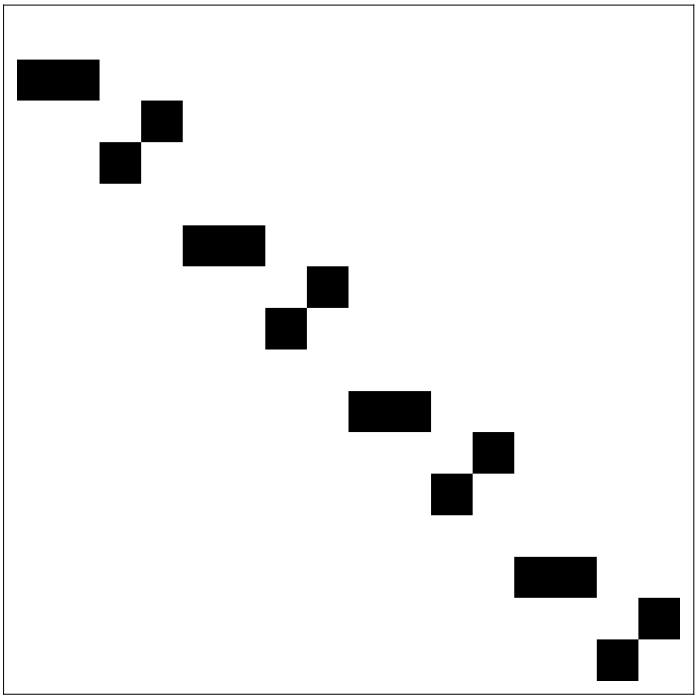
In order to apply a 2-bit gate to a >2-bit system, the matrix cannot be multiplied directly with the state vector of the entire system. Rather, the gate matrix needs to be composed with Identity matrices representing the lack of operations on other bits. The computed matrices, because they are in a shape other than $2^2 \times 2^2$, are made in the correct dimension, so they are able to apply to the whole state. Here is how each gate is converted to a matrix:

In[41]:=
```
matrixFormOfColumn[G_, n_] :=
  (* M = G[[2]] *)
  (* s1 = G[[1, 1]] *)
  (* s2 = G[[1, 2]] *)
  If[MatchQ[G[[2]], _List], (* For fixed matrices *)

    KroneckerProduct[IdentityMatrix[2^(G[[1,1]]-1)], G[[2]], IdentityMatrix[2^(n - G[[1, 2]])]],

   G[[2]][n, G[[1, 1]], G[[1, 2]]]] (* Computed matrix: uses Mtype[n, s1, s2] *)
```

For example, here is how the NAND gate is represented when applied to the last two bits of a 4-bit system. The repeating NAND pattern can be seen on the lease significant bits, and the diagonal identity pattern can be seen on the most significant bits.

In[42]:=
```
matrixFormOfColumn[{{3, 4}, matrixNand}, 4]//ArrayPlot
```

Out[42]=



Circuits Implementation

Circuit Application

A circuit is the composition of a set of gates applied to a set of wires. Because each gate is represented by a matrix, application of a gate is equivalent to multiplication of a state vector by a matrix. Because matrix multiplication is associative, all of the matrices of a circuit can be multiplied together into a single total circuit matrix. As most circuits require many more internal wires than input or output

wires, a circuit matrix can be precomputed (which will take a relatively large amount of time) and then applied to one or more input state vectors (which will take a relatively small amount of time).
Finally, we can convert apply circuit to a logical function which can be displayed.
Here is the code that does this, first computing each matrix for each gate, then composing them by multiplication, then applying them to an input state, then creating a logic function from that:

In[43]:=
```
(* Generates a matrix for each column of a circuit *)
circuitMatrices[circuit_]:= With[{n = Max@Flatten@circuit[[All, 1]]}, matrixFormOfColumn[#, n]& /@

(* Generates a combined matrix representing the whole circuit *)
circuitMatrix[circuit_]:= Dot @@ Reverse @ circuitMatrices[circuit]

(* Apples the combined matrix to a state vector, returning an output state vector *)
applyCircuit[circuit_, state_]:= circuitMatrix[circuit].state

(* Represents circuit application as a function in boolean logic *)
logicFunctionFromCircuit[circuit_]:=(applyCircuit[circuit, fullState[{#1, #2}]] // v2i)&  (* Onl
```

Circuit Visualization

Example XOR Circuits

Now that we can create, apply, and visualize a circuit to an input, here are some circuits that perform the XOR gate using only the NAND gate:

In[51]:=
```
XorFromNandGatesCircuit = {
   (* Makes use of SWAP, NAND, and COPY *)
 {{3, 5}, matrixZeroExpand},
 {{2, 3}, matrixCopy},
 {{2, 3}, matrixNand},
 {{1, 2}, matrixSwap},
 {{3, 4}, matrixSwap},
 {{2, 3}, matrixCopy},
 {{2, 3}, matrixNand},
 {{1, 2}, matrixNand},
 {{3, 4}, matrixNand},
 {{2, 3}, matrixSwap},
 {{3, 4}, matrixNand},
 {{4, 5}, matrixCopy},
 {{4, 5}, matrixNand},
 {{5, 5}, matrixReturn}
 };

XorFromNandGatesCircuitNoCopy = {
   (* Longer, but only makes use of SWAP and NAND *)
   {{3, 12}, matrixRepeatExpand},
   {{4, 5}, matrixSwap},
   {{10, 11}, matrixSwap},
   {{1, 2}, matrixNand},
   {{3, 4}, matrixNand},
   {{5, 6}, matrixNand},
   {{7, 8}, matrixNand},
   {{9, 10}, matrixNand},
   {{11, 12}, matrixNand},
   {{4, 5}, matrixSwap},
   {{10, 11}, matrixSwap},
   {{5, 6}, matrixNand},
   {{11, 12}, matrixNand},
   {{2, 5}, matrixSwap},
   {{8, 11}, matrixSwap},
   {{5, 6}, matrixNand},
   {{11, 12}, matrixNand},
   {{6, 11}, matrixSwap},
   {{11, 12}, matrixNand},
   {{12, 12}, matrixReturn}
    };
```

As you can see, if we run these two circuits, we get the same truth table as XOR:

```
numericalTruthTable[logicFunctionFromCircuit[XorFromNandGatesCircuit], "XorWithCopy"]

(* Takes a really long time to run,
because large circuits are extremely inefficient. Like literally
 >1000 seconds to run a single XOR gate. Run at your own risk. *)
numericalTruthTable[
 logicFunctionFromCircuit[XorFromNandGatesCircuitNoCopy], "XorWithoutCopy"]
```

Out[•]=

XorWithCopy	0	1
0	1	0
1	0	1

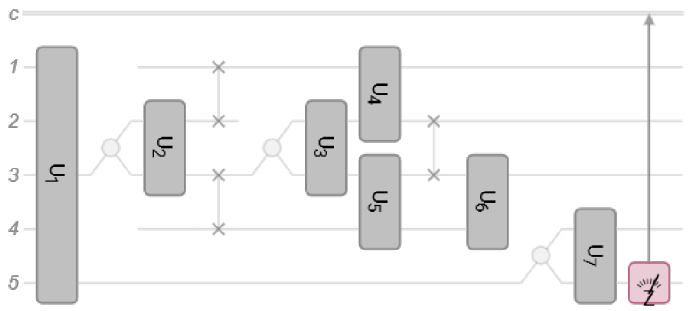
Out[•]=

```
$Aborted
```

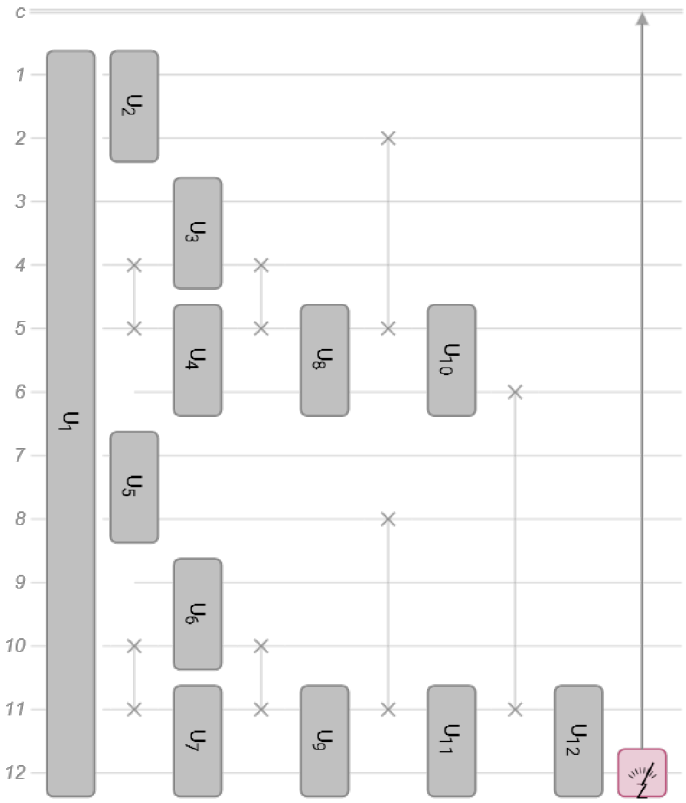
We can visualize these quantum circuits using the quantum circuit diagram from the Wolfram Quantum Framework paclet:

```
In[•]:= Row[{Labeled[circuitPlot[XorFromNandGatesCircuit], "XOR With Copy"], Spacer[50],
           Labeled[circuitPlot[XorFromNandGatesCircuitNoCopy], "XOR Without Copy"]}]
```

Out[•]=



XOR With Copy



XOR Without Copy

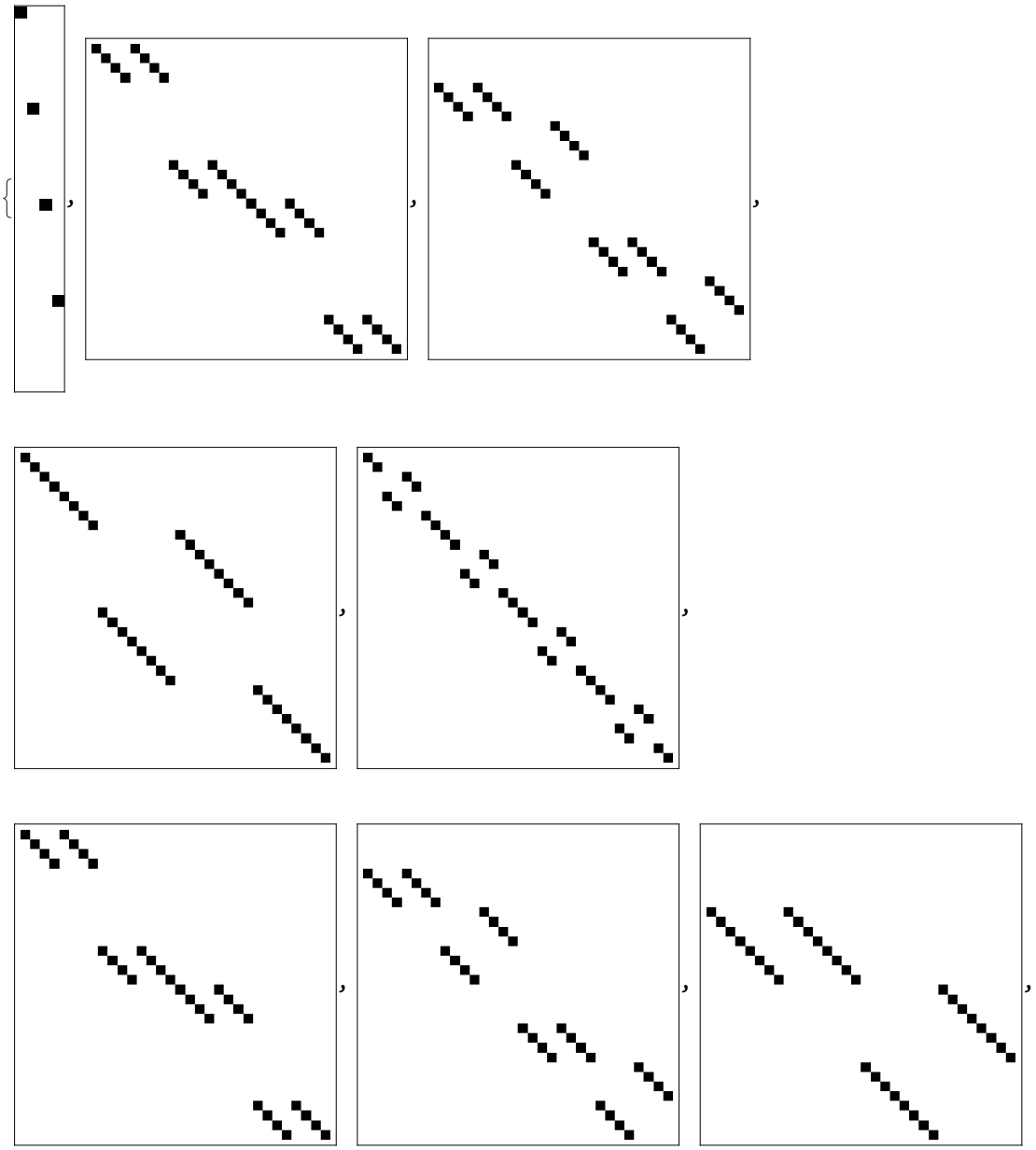
For the smaller XOR circuit which uses copy, we can easily visualize the matrices that make up the circuit:

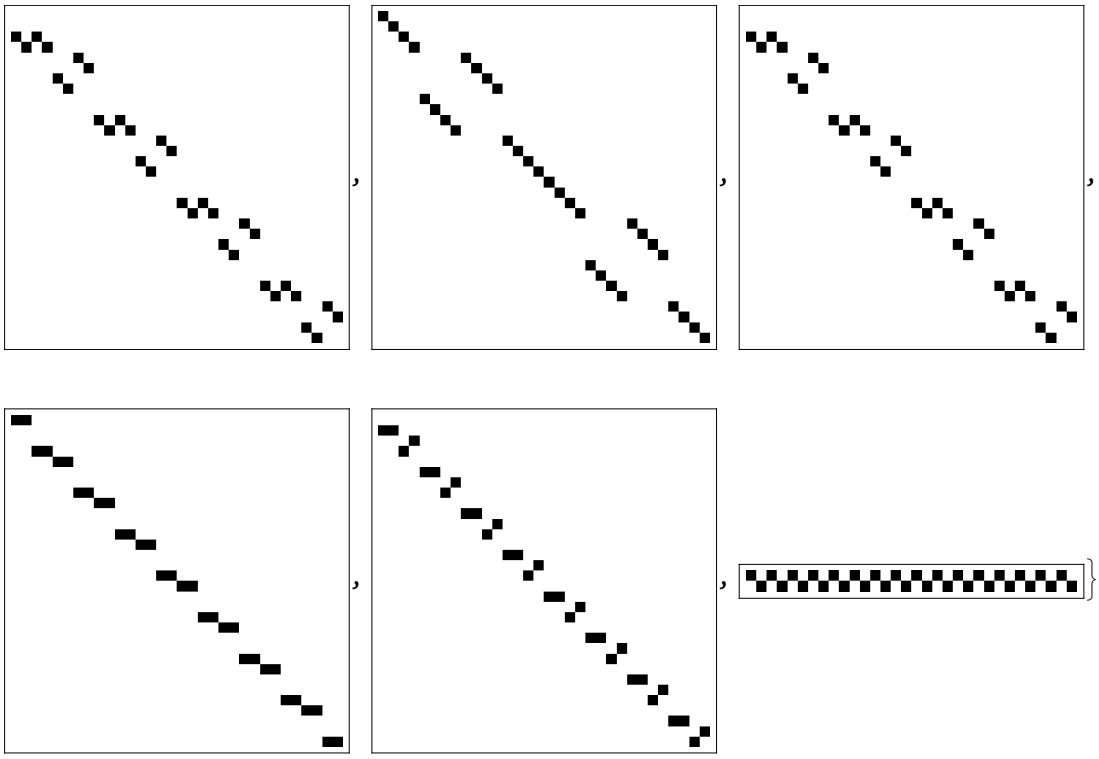
The first and last matrices are not square, because they setup and measure the state vector, respectively.

The interior gate matrices are usually mostly on the diagonal, because they are composed of many identity matrices but only one gate matric.

In[•]:= **ArrayPlot /@ circuitMatrices[XorFromNandGatesCircuit]**

Out[•]=

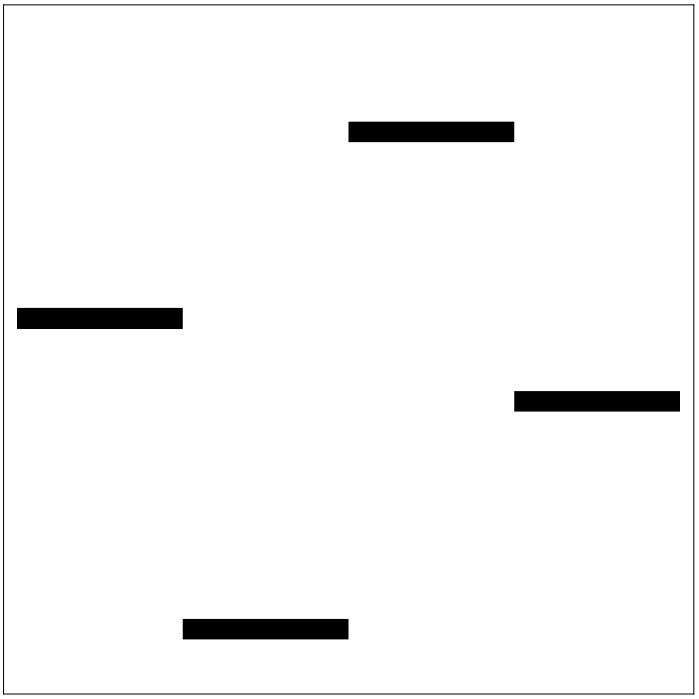




Here is the composition of all of the interior matrices:

```
In[•]:= circuitMatrix[XorFromNandGatesCircuit[[2 ;; -2]]] // ArrayPlot
```

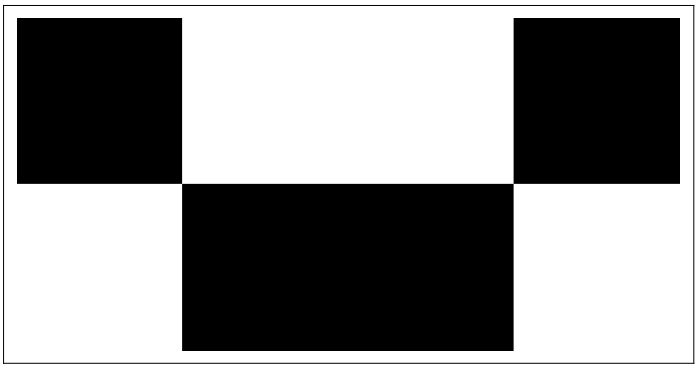
Out[•]=



And here is the final composition of all the matrices, which makes a matrix representation of XOR:

```
In[•]:= circuitMatrix[XorFromNandGatesCircuit] // ArrayPlot
```

Out[•]=



Universal Gates

Definition

A set of universal gates can be formulated as the set of gates that comprises matrix decompositions to all other possible gates.

Classical Definition

A set of gates is universal if, for all possible 2x2 gates T, there exists an equivalent circuit with some number of wires w and some number of gates m such that the columns c_i of the circuit apply a gate to wires λ_1 through λ_2. Formally,
S_g is a set of universal gates iff $\forall\, G \in S_G,\ \forall\, G_I \in G,$

$$G_i \in \{e_1,\ e_2,\ e_3,\ e_4\} \text{ of } \mathbb{B}^4,\ \forall \left\{T \in \mathbb{B}^{4\times 4} \;\middle|\; \forall\, T_i \in T,\ T_i \in \{e_1,\ e_2,\ e_3,\ e_4\} \text{ of } \mathbb{B}^4\right\} \exists\, w \in \mathbb{N},$$
$$m \in \mathbb{N} \;\middle|\; \left(P \in \mathbb{B}^{2^w \times 4} \;\middle|\; \forall\, P_i \in P,\ P_i \in \{e_1,\ e_2,\ e_3,\ e_4\} \text{ of } \mathbb{B}^4\right) \times$$
$$\left((c_1 \times c_2 \ldots c_m) \;\middle|\; c_i \in \left\{{I_2}^{\otimes \lambda_1} \otimes G \otimes {I_2}^{\otimes \lambda_2} \;\middle|\; \forall\, \lambda_1 \in \mathbb{N},\ \lambda_2 \in \mathbb{N} \mid G \in S_G \wedge \lambda_1 + 2 + \lambda_2 == w\right\}\right) \times$$
$$\left(R \in \mathbb{B}^{4\times 2^w} \;\middle|\; \forall\, R_i \in R,\ R_i \in \{e_1,\ e_2,\ e_3,\ e_4\} \text{ of } \mathbb{B}^4\right) == T,$$

where P and R are state preparation and reduction matrices, respectively.

Quantum Definition

Very similar to the classical definition, but with a few requirements for quantum gates and the possibility of complex unit vectors. Thus,
S_g is a set of quantum universal gates iff $\forall\, G \in S_G,\ G$ is unitary $\wedge\ \forall\, G_i \in G,\ \|G_i\| == 1\},$

$$\forall \left\{T \in \mathbb{C}^{4\times 4} \;\middle|\; T \text{ is unitary} \wedge \forall\, T_i \in T,\ \|T_i\| == 1\right\} \exists\, w \in \mathbb{N},$$
$$m \in \mathbb{N} \;\middle|\; \left(P \in \mathbb{B}^{2^w \times 4} \;\middle|\; \forall\, P_i \in P,\ P_i \in \{e_1,\ e_2,\ e_3,\ e_4\} \text{ of } \mathbb{B}^4\right) \times$$
$$\left((c_1 \times c_2 \ldots c_m) \;\middle|\; c_i \in \left\{{I_2}^{\otimes \lambda_1} \otimes G \otimes {I_2}^{\otimes \lambda_2} \;\middle|\; \forall\, \lambda_1 \in \mathbb{N},\ \lambda_2 \in \mathbb{N} \mid G \in S_G \wedge \lambda_1 + 2 + \lambda_2 == w\right\}\right) \times$$
$$\left(R \in \mathbb{B}^{4\times 2^w} \;\middle|\; \forall\, R_i \in R,\ R_i \in \{e_1,\ e_2,\ e_3,\ e_4\} \text{ of } \mathbb{B}^4\right) == T,$$

where P and R are state preparation and reduction matrices, respectively.

Classical Universality

Classical 2→1 Universality Proof

Theorem

A gate set containing at least one gate that satisfies each of the following properties is universal:

- G(a, a) = ¬a
- G(0, 0) + G(0, 1) + G(1, 0) + G(1, 1) is Odd

Lemma 1:

- In order to make a 2→1 NOT gate, a gate set must have a gate which satisfies the property G(a, a) = ¬a.

In order for a gate to be universal, it needs to be able to implement any 2-bit gate, including the 2-bit NOT gates: $\begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$ or $\begin{pmatrix} 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{pmatrix}$. The 2-bit NOT gates comprise a 1-bit NOT gate on a single wire, and a 1-bit NOT gate can be constructed by using only that single bit of the 2-bit version. Thus, the ability of a gate set to construct a 2-bit NOT gate is equivalent to it's ability to construct a 1-bit NOT gate.
The only way for a 1-bit NOT gate to be constructed from a 2→1 gate is if the gate satisfies the property that G(a, a) = ¬a, because the input wire to the gate can be duplicated, thus constructing a NOT gate.

Lemma 2:

- For 2→1 gates with two 1-valued outputs, the number of 1-valued outputs stays the same under NOT transformations.

Define the sum of a gate to be the parity of the sum of the digits in it's truth table. I.e., the number of 1-valued outputs.
The truth table of a gate constructed from applying a NOT gate to it's first wire then applying some other 2→1 gate is equivalent to the truth table of the other gate mirrored horizontally. The same is true for NOT gates applied to the second input being equivalent to a vertical mirroring of the original truth table. Moving the values of a truth table does not change the sum, so these transformations of a gate preserve sum.
For a gate constructed by applying a NOT gate to the output of some other gate, because all the outputs of 0 will turn to 1, and there are 4-n 0s in the truth table for a gate with n 1s, then the sum of the new gate will be 4 - (sum of the old gate).

For gates with a sum of 2, the value thus never changes under transformations of NOT.

Corollary to Lemma 2:

- 2→1 gates with a sum of 2 can be decomposed into 1-bit NOT gates.

The identity truth table represents a gate which has no effect on the wires of the circuit. Because the identity gate has a sum of 2, this means that it is equivalent to all other gates with a sum of 2 under NOT gate transformations.
As identity gates can be removed with no effect to the circuit, any 2-sum gate is equivalent to some transformation of 1-bit not gates. If a gate set already has the ability to implement NOT gates, which every universal set must by Lemma 1, then 2-sum gates are always decomposable into other gates of the set.

Proof:

By Lemma 1, all universal gate sets must have a gate with the property G(a, a) = ¬a.
By the Corollary to Lemma 2 gates with 2-sum are decomposable to NOT, and because the only other even-sum gates have a constant output, a universal gate set must contain a gate with odd parity.

Results:

Out of all of the possible 2→1 gates

```
In[•]:= truthTable[BooleanFunction[#, 2], "func" <> ToString[#]] & /@ Range[0, 15]
```

Out[•]=

func0	0	1
0	0	0
1	0	0

func1	0	1
0	1	0
1	0	0

func2	0	1
0	0	1
1	0	0

func3	0	1
0	1	1
1	0	0

func4	0	1
0	0	0
1	1	0

func5	0	1
0	1	0
1	1	0

func6	0	1
0	0	1
1	1	0

func7	0	1
0	1	1
1	1	0

func8	0	1
0	0	0
1	0	1

func9	0	1
0	1	0
1	0	1

func10	0	1
0	0	1
1	0	1

func11	0	1
0	1	1
1	0	1

func12	0	1
0	0	0
1	1	1

func13	0	1
0	1	0
1	1	1

func14	0	1
0	0	1
1	1	1

func15	0	1
0	1	1
1	1	1

there exist only two gates which individually satisfy both of the properties given above:

```
In[•]:= truthTable[BooleanFunction[#, 2], "func" <> ToString[#]] & /@ {7, 14}
```

Out[•]=

func7	0	1
0	1	1
1	1	0

func14	0	1
0	0	1
1	1	1

Thus, these two gates, the NAND and NOR, respectively, are the only two gates which are individually universal.

Classical 2→2 Universality Proof

Theorem:

Any 2→1 universal gate set can be made into a 2→2 universal gate set.

Lemma 1:

All 2→2 gates can be decomposed into two 2→1 gates ($G_\alpha(x,\ y),\ G_\beta(x,\ y)$).

Lemma 2:

Any of the following forms allow a 2→1 gate to be constructed by only using a single output bit of a 2→2 gate:
$(G(x,\ y),\ 0),\ (G(x,\ y),\ 1),\ (0,\ G(x,\ y)),\ (1,\ G(x,\ y)),$
$(G(x,\ y),\ x),\ (G(x,\ y),\ y),\ (x,\ G(x,\ y)),\ (y,\ G(x,\ y))$

Proof:

Thus, the gates of any 2→1 universal gate set can be transformed into 2→2 gates which themselves are universal to the 2→2 gates.
(Although there are also more ways to construct universal 2→2 gates, such as by combining a universal 2→1 gate set of length two into a single 2→2 gate.)

Quantum Universality

Quantum Matrix Definitions

Single Qubit Gates

Bloch Sphere Implementation

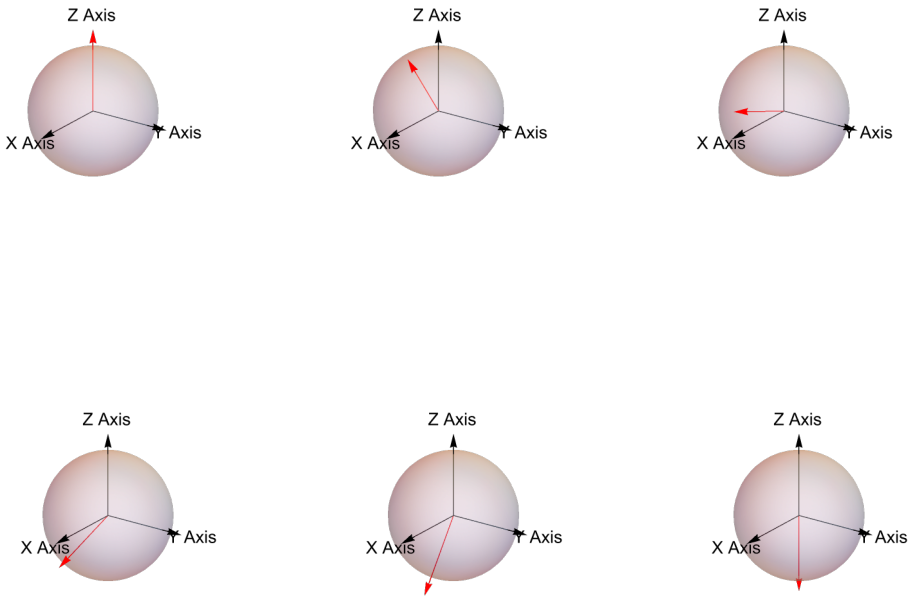
Bloch Sphere Visualization

A Bloch Sphere is a visualization of a qubit state $\alpha\,|0\rangle + \beta\,|1\rangle$, where $\alpha,\ \beta \in \mathbb{C}^2$ such that $|\alpha|^2 + |\beta|^2 = 1$. Although a space of two complex numbers is isomorphic to $\mathbb{R}^4$, not $\mathbb{R}^3$, the global phase of a state (i.e., the imaginary component of α) has no physical significance, and can be discarded, leaving only three dimensions necessary to be visualized.
The state vector projected onto the Z-axis corresponds to the qubit's probability of being measured 0 or 1 (technically, the Z-projection gives the square root of probability), and the rotation about the Z-axis corresponds to the relative phase between the $|0\rangle$ and $|1\rangle$ states.

For example, here is a Bloch sphere visualizing a rotation of $\frac{\pi}{3}$ about the Y-axis:

In[415]:=
```
θ = {0, Pi / 5, 2 Pi / 5, 3 Pi / 5, 4 Pi / 5, Pi};
Row[qubitStateGraphics[{1, 0}.RotationMatrix[-0.5 #], Small, False] & /@ θ]
```

Out[416]=



Arbitrary Rotation Gates

The most common quantum computing gates correspond to rotations around the principal axes.

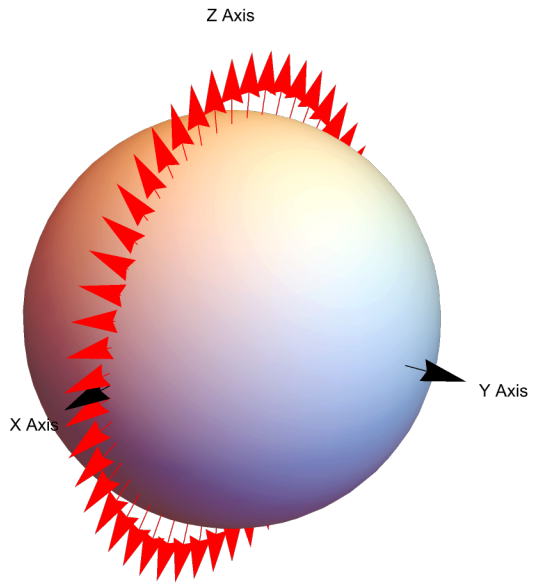
The arbitrary rotation gates are $R_x(\theta)$, $R_y(\theta)$, $R_z(\theta)$,
and the rotations with $\theta = \pi$ are denoted X, Y, and Z,
and the rotations with $\theta = \frac{\pi}{2}$ are denoted $X^{\frac{1}{2}}, Y^{\frac{1}{2}}, Z^{\frac{1}{2}}$.

In order for a gate set to be universal, it must be able to reach any point on the sphere, because the gates $R_x(\theta)$, $R_z(\varphi)$ are able to reach any point in spherical coordinates $(1, \theta, \varphi)$.
This presents a problem, as any arbitrary rotation gate $R_u(\theta)$ is technically an infinite set of gates for all choices of θ. This can be solved by only approximating arbitrary rotations with a smaller, finite set of rotations.

In[231]:=

```
Labeled[Show @@
  (qubitStateGraphics[{1, 0}.RotationMatrix[-#], Large, False] & /@ Range[0, 2 Pi, Pi/50]),
 "Approximating arbitrary Y-rotation \nwith a finite number of gates"]
```

Out[231]=



Approximating arbitrary Y-rotation
with a finite number of gates

By the Solovay-Kitaev theorem, an arbitrary rotation to within an accuracy of ε can be done efficiently in $O\left(\log^k\left(\frac{1}{\varepsilon}\right)\right)$ complexity for some number k. However, this approach may use multiple types of gates, and so may not give the minimal number of gates possible.
Instead, I developed my own algorithm for implementing arbitrary rotations by using a $\frac{\pi}{4}$ rotation about one axis, and a rotation of at most $\varphi = \sqrt{2} \times \varepsilon$ in a perpendicular axis.

(The $\varphi = \sqrt{2} \times \varepsilon$ comes from finding the nearest multiple of φ to an arbitrary angle θ. The angular distance is at most $\frac{1}{2}\varphi$ in 2D and $\frac{\sqrt{2}}{2}\varphi$ in 3D, in the case where θ is directly in between the multiples of φ. As ε goes to zero, because we are working on the unit sphere, the euclidian distance approaches the angular distance in radians. Until then, the maximum euclidian distance is less than the maximum angular distance. In the original definition, ε was defined in terms of the max euclidian distance of the columns, so using the formula $\frac{\sqrt{2}}{2}\varphi \leq \varepsilon$, we get that $\varphi = \sqrt{2} \times \varepsilon$.)

In-Depth Solution Example

Single-Qubit Gates

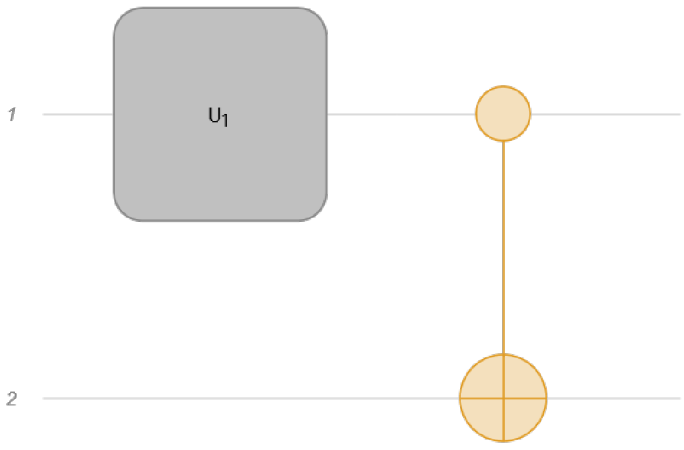
All single-qubit gates can be implemented with these simple rotations, so these two gates of $Y^{\frac{1}{2}}$ and $R_z(\varphi)$ make a universal set for single-bit gates. However, because they are both 1→1 gates, they cannot make different wires interact, and so are not themselves universal for 2→2 gates.

Multi-Qubit Gates

The simplest way to make a universal set of 2→2 gates from a universal set of 1→1 gates is to add on the CNOT gate. The CNOT, or controlled not, applies a $\frac{\pi}{2}$ flip in the X axis--equivalent to a bit flip in classical computing--only when the control qubit is in the one state. This allows two qubits to be entangled together very easily, such as in the following circuit:

In[437]:=
```
circuitPlot[{{{1, 1}, gateH}, {{1, 2}, gateCNOT}}]
```

Out[437]=



Which will transfer the state $|\psi\rangle|0\rangle$ to $|\psi\psi\rangle$. Whenever you measure the first qubit in the X-basis, you will know the value of the second qubit in the X-basis.

In order to make my gate set universal, I need to implement some type of controlled operation. I could add CNOT, which would make it universal, or I can replace my $R_z(\varphi)$ with a controlled arbitrary rota-

tion $CR_z(\varphi)$.

Proof of Universality

Now that a multi-qubit gate has been added, it is possible to prove that my gate set of $\left\{Y^{\frac{1}{2}},\ CR_z(\varphi)\right\}$ is universal.

Other Universal Sets

There are a variety of universal quantum gates that have already been found. Some of these, such as {Toffoli, H} or {$CCiR_x(a\pi)$ for an irrational a} use very few gates, but require 3→3 qubit gates (i.e., the Toffoli and $CCiR_x$), but the starting point for my formulation will rely on the universal set {H, S, CNOT, T}.
These four gates are a universal set, so if any set of gates that can create H, S, CNOT, and T, then it is also universal.