Operator Module API

Linear Operator and Matrix representation

Consider some unitary operator U, where U has matrix representation:

$$U = \begin{bmatrix} U_{00} & U_{01} \\ U_{10} & U_{11} \end{bmatrix}$$

U imparts a certain transition amplitude to an initial state $|\psi\rangle$. Specifically we can think of each element U_{ij} of U as the transition amplitude imparted on state $|j\rangle$ as it transitions to $|i\rangle$. Another way to think about this is that is each U_{ij} takes state $|j\rangle$ into $|i\rangle$ with probability U_{ij}

Constructing matrix representations using outproducts

In FQAM, we want a method of defining these matrix representation of unitary operators in a manner that is flexible. Simply tabulating all the mappings a unitary operator performs as a matrix can be cumbersome especially for large state spaces. That is we would rather not have to explicitly write out the unitary matrix for each operator we want to execute.

Instead it is desirable to have a framework that allows the specification of operators by only defining the states that 'matter' in a transition.

Hence, in FQAM, for now, all operators are specified by their outer product representation in the following manner:

$$\sqrt{\mathbf{NOT}} = \begin{cases} U_{00} := \frac{1}{\sqrt{2}} |0\rangle \langle 0| \\ U_{01} := \frac{i}{\sqrt{2}} |0\rangle \langle 1| \\ U_{10} := \frac{i}{\sqrt{2}} |1\rangle \langle 0| \\ U_{11} := \frac{1}{\sqrt{2}} |1\rangle \langle 1| \end{cases}$$

Which corresponds to the following in the codebase:

```
FQAM_Operator_add (FQAM_op *operator, Complex alpha, FQAM_op *op_term);
```

Args:

- FQAM_op *operator: A pointer to an operator object to add to.
- Complex alpha: Complex scalar coefficient.
- FQAM_op op_term: Operator object representing the term.

Notes:

- Semantically equivalent to adding individual terms in a summation outer product decomposition of an operator:

$$U = \sum_{i=1}^{n} |i\rangle \langle i|$$

Invariant: Every call of this function must satisfy an invariant to ensure valid unitary operators.

Relation to Precondition Calling append: Prior to calling stage append, we precondition the unitary must be a valid unitary.

Aside: Remark on manipulating multi-qubit states from the perspective of local qubit operations

We have a system of qubits represented as a contiguous array (i.e., the state vector).

- In FQAM, multi-qubit operators are applied locally to a neighborhood of qubits.
- The neighborhood can be varied/dialed to logically define locality in terms of an n-dimensional lattice.
- That is, we can define different views into the contiguous array which expose the array elements in different ways.

In terms of DLA, we can think of this as just being an 'embedding' of a 1D rank 1 flattened buffer/array into a dimension m rank m tensor.

FLA_repart ();

- A neiborhood is defined wich defines meaning of locality

Example:

|0><0|

```
FQAM_Operator_add_term(FQAM_op operator, 1/sqrt(2), |0><0|);
FQAM_Operator_add_term(FQAM_op operator, i/sqrt(2), |1><0|);
FQAM_Operator_add_term(FQAM_op operator, i/sqrt(2), |0><1|);
FQAM_Operator_add_term(FQAM_op operator, 1/sqrt(2), |1><1|);
```

where each $|0\rangle$ and $|1\rangle$ are part of some orthonormal basis.

Aside:

In general, I am not too sure yet about the theory of operator decomposition, but in the future, I would like to be able to add functionality such that any unitary operator A can be specified in terms of a decomposition of other operators. The outer product case above can be thought of as a special case of this.

More formally, the user should be able to define an operator A like so:

$$A = \sum_{i=1}^{n} \alpha_i P_i, \text{ where } P_i \text{ is an operator}$$

Generating Orthonormal Basis

An orthonormal basis in Hilbert space C^2 is parameterized by:

$$\left\{ \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix}, \begin{bmatrix} \sin(\theta) \\ -\cos(\theta) \end{bmatrix} \right\}$$

Hence in the code base one can generate any

```
FQAM_Operator_outer_product (FQAM_Op *op, int theta);
```

Args:

- FQAM_Op *op: Operator store result into
- int theta: Angle in [0, 2pi].

Summary of the API

The FQAM API need to be universal insofar any unitary operator describing any evolution can be constructed by only the following commands:

Public:

```
void FQAM_Basis_state   (FQAM_Basis basis, int eigenstate, double angle);
void FQAM_Basis_state_N (FQAM_Basis basis, int eigenstate, double angle, dim_t dim);
void FQAM_Op_outer      (FQAM_Op outer, FQAM_Basis ket0, FQAM_Basis ket1);
void FQAM_Op_add        (FQAM_Op outer, Complex alpha, FQAM_Op term);
bool FQAM_Op_check      (FQAM_Op op); // Asserts 'boundary conditions'
```

Internal:

```
void tensor_product (); // To generate
```

API Examples

```
typedef struct {
  double
} FQAM_Basis;

FQAM_Basis_standard ();
FQAM_Basis_hadamard ();

// Stores the result of the outer product of given operators ket0 and ket1
FQAM_Op_outer (ket0, ket1, outer2);
```

```
// Returns orthonormal eigenstate of the basis states specified by angle
FQAM_Basis FQAM_Basis_state (int eigenstate, double angle);

FQAM_Basis_standard ();

// Example of creating a not using NOT
void example_not ()
{
  FQAM_op ket0;
  FQAM_op ket1;
  FQAM_op outer1, outer 2;
  FQAM_op res;

  FQAM_Op_create (op);

  ket0 = FQAM_Basis_standard (0);
  ket1 = FQAM_Basis_standard (1);

  // |0><1|
  FQAM_Op_outer (ket1, ket0, outer);

  // |1><0|
  FQAM_Op_outer (ket0, ket1, outer2);

  FQAM_Op_add (1, outer1, res);
  FQAM_Op_add (1, outer2, res);
}

// Example of creating a not using NOT
void example_identity ()
{
  FQAM_op ket0;
  FQAM_op ket1;
  FQAM_op outer1, outer 2;
  FQAM_op res;

  FQAM_Op_create (op);

  ket0 = FQAM_Basis_standard (0);
  ket1 = FQAM_Basis_standard (1);

  // |0><0|
  FQAM_Op_outer (ket0, ket0, outer);

  // |1><1|
  FQAM_Op_outer (ket1, ket1, outer2);
```

```
  FQAM_Op_add (1, outer1, res);
  FQAM_Op_add (1, outer2, res);
}

void example_CNOT (void)
{
  FQAM_Basis ket0;
  FQAM_Op ket1;
  FQAM_Op outer00, outer11;
  FQAM_Op res;

  ket0 = FQAM_Basis_standard (0);
  ket1 = FQAM_Basis_standard (1);

  outer00 = FQAM_Op_outer (ket0, ket0)
  outer11 = FQAM_Op_outer (ket1, ket1)



  outer00 = FQAM_Op_outer (ket0, ket0)
  outer11 = FQAM_Op_outer (ket1, ket1)

  Op = |0><0|I + |1><1|X
  FQAM_Op_add (res, 1, outer00, res);
  FQAM_Op_add (1, outer11, res);
}

void example_phase_not (void)
{
  |0><0|0><I|I
}

void example_root_not (void)
{
  |0><0|

  FQAM_Op_add (1, , res);
  FQAM_Op_add (1, outer11, res);
}
```

Aside: Ensuring closure. Ensuring Norm preservation

How can we ensure

Aside: Hierarchical/Recursive creation of operators

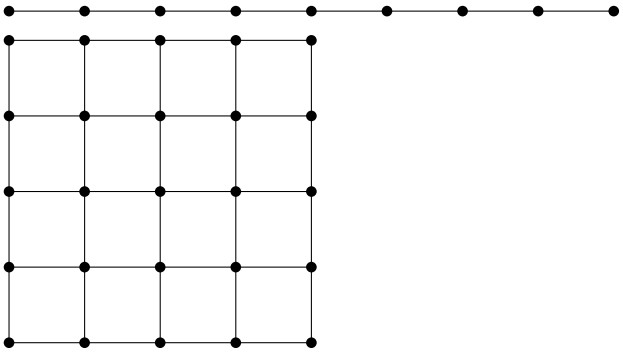
Suppose we have a three qubit system $|\psi\rangle = |000\rangle$. An example of how we can create a new operator A using FQAM_Op_create () called and use the it to generate B.

$$A = |0\rangle \langle 0| X |0\rangle \langle 0|$$

$$B = |0\rangle \langle 0| |0\rangle \langle 0| A$$

A in this case is an operator that itself hierarchically applies X on the center qubit. We can then create B an operator that hierarchically applies n the third qubit.

Aside: Embeding Views



Aside: Thinking FQAM and it's relation to conservative logic

The issue I am currently thinking about is how formalisms for quantum mechanics can be used formalisms for conservative logic. In general it seems to me that all these different formalism like:

- Feynman path integral formulation (Quantum field theory)
- Dirac notation
- Heisenberg QM formulation

are all forms of conservative system representation.

This leads me to the following question:

What is a universal conservative logic formalism that lets us easily and efficiently implement the following:

- Cellular Automata

- Apoetic systems
- QIS applications:
  - Algorithms
  - QEC/Stabilizer circuits

By universal I mean these can be fully be described using tools only from the formalism. In other words: Any arbitrary orchestration of conservative logic wether ,it be quantum or classical, can be described using this formalism.