Assignment 3

Due date: Wednesday, March 14

Working with qubits

In this exercise you develop the computational machinery needed for reconstructing a quantum state from a set of measurements, or "quantum state tomography". The measurement matrices you construct here will be the starting point for the actual tomography in the next assignment.

Your quantum system is comprised of four qubits. One way to represent a general state[1] is as a $2 \times 2 \times 2 \times 2$ tensor x. For example, $x[1, 2, 2, 1]$ is the amplitude that qubits 1 and 4 are spin-up, and qubits 2 and 3 are spin-down.

The tensor representation is what you want to use when acting on a state with spin operators. You will be using measurements that involve single qubits as well as pairs of qubits, such as

$$\sigma_x^1 \, \sigma_y^3. \tag{1}$$

These are the standard Pauli spin operators and the superscript tells you which qubit is being acted upon. Write a program that acts on an arbitrary state (tensor) with operators such as (1). Test your program by computing the 16×16 matrix for (1) in the *standard basis*. In basis state 1 all the elements of x are zero except $x[1, 1, 1, 1]$, which is 1. In the next basis states (2, 3, 4 ...), the single 1-element is located at $x[1, 1, 1, 2]$, $x[1, 1, 2, 1]$, $x[1, 1, 2, 2]$, etc. The 16×16 matrix for (1) is zero except for a single $\pm i$ in each row and column[2]. Here are the $[\mathrm{row}, \mathrm{col}]$ indices of the $+i$ entries:

$$[1, 11] \quad [2, 12] \quad [5, 15] \quad [6, 16] \quad [9, 3] \quad [10, 4] \quad [13, 7] \quad [14, 8].$$

The $-i$ entries are uniquely determined from these by hermiticity of (1). Be sure to write your code in such a way that you can construct matrices for all $6 \times 3 \times 3$ operators of the form (1), i.e. all distinct pairs of spins and all combinations of Paulis.

[1] The more conventional symbol would be Ψ, but we're keeping with the course convention of the-entity-being-reconstructed-shall-be-called-x.

[2] Your matrix may differ by the exchange $i \leftrightarrow -i$, which is just a matter of convention.

Pizza for (up to) five

You've invited four people over for pizza, so the maximum number of people sharing the pie is five. Your task is to divide the pizza into p pieces in advance, so that no matter how many guests show up you will be able to combine the pieces in such a way that everyone gets the same amount. The smallest p for which a solution exists is $p = 9$.

Use RRR to find at least two different solutions. The unknown, as explained in lecture, is a $3 \times 4 \times 5$ tensor x on which we impose two constraints:

EQUIPARTITION: This is a system of linear equations:

$$\forall i_1 \neq i_2 : \quad \sum_{j,k} x[i_1, j, k] = \sum_{j,k} x[i_2, j, k],$$

and similarly for the other two dimensions of the tensor. Use the simple algorithm described in lecture to project to this constraint.

SPARSITY-p: Only p elements of x may be non-zero, and these must be positive.

In "small" applications of RRR, such as this one, there is a higher probability of the dynamical system falling into an unproductive cycle. It is therefore likely that you will have to make several runs (from random starts) in order to find two true solutions. When you get convergence, verify the solution by rounding the nonzero elements of x to candidate rational numbers. *Hint*: first multiply by $\mathrm{LCM}(3, 4, 5)$.

Latin squares by divide and concur

A *latin square* of order n is an $n \times n$ array of n symbols arranged so that each symbol occurs exactly once in each row and column.

A nice way to represent a latin square is as an $n \times n \times n$ tensor x with only 0 and 1 elements. Think of x as a "building", where the 1's on floor k give the locations of the symbol k on that floor. Clearly on each floor we must have exactly a single 1 in each (horizontal) row and (horizontal) column. But because the latin square has a unique symbol in each array element, the building must also have a single 1 in each vertical stack as well. Summarizing: the 0/1 tensor x must have exactly n^2 1's arranged so that in each of the n^2 horizontal rows, and each of the n^2 horizontal columns, and in each of the n^2 vertical stacks, there is exactly a single 1.

Latin square completion, related to sudoku, is where $g < n^2$ of the symbols are given and you need to consistently fill in the rest — or prove this is impossible. This is equivalent to being given the positions of g 1's in the tensor x.

There are many ways of applying RRR to latin square completion. In this exercise you will try one of the easiest implementations, called *divide and concur*. The idea

is to work with a larger tensor y, of size $3 \times n \times n \times n$. Think of $y[1, \cdots]$, $y[2, \cdots]$, $y[3, \cdots]$ as three copies of the tensor x on which we are free to impose independent constraints, corresponding to the rows, columns, and vertical stacks in the "building". We still end up with just two meta-constraints for which there are easy projections in the RRR scheme:

DIVIDE:

$$\begin{aligned} \forall\, i, j: &\quad y[1, i, j, k] \text{ is zero except at a single } k \text{ where it equals } 1 \\ \forall\, j, k: &\quad y[2, i, j, k] \text{ is zero except at a single } i \text{ where it equals } 1 \\ \forall\, k, i: &\quad y[3, i, j, k] \text{ is zero except at a single } j \text{ where it equals } 1. \end{aligned}$$

CONCUR:

$$\forall\, i, j, k: \quad y[1, i, j, k] = y[2, i, j, k] = y[3, i, j, k].$$

The positions of the g given symbols (for latin square completion) can be introduced into either constraint with little overhead. In DIVIDE you would check if the row/column/stack intersected with the 1 of a given symbol and make that the choice. In CONCUR, in addition to the three copies being equal, the value equals 1 when $[i, j, k]$ is the location of a given symbol (optionally, set to 0 the other elements on the intersecting row, column, stack). Pick one of these (or both) and experiment to see which is better at finding solutions. Work with the incomplete latin square in the data folder of the course github site.

This RRR implementation of the latin square problem happens to have a very strong dependence on the time-step parameter β.