RuClExample

March 21, 2024

1 RuCl Full Scale Example

The search for Kitaev Spin Liquid Materials is a very active area of Material Science Research. One example of a candidate material that demonstrates promise as a Kitaev Spin Liquid is α-RuCl$_3$. In this notebook, we explore the necessary steps to simulate the dynamics of this material on a Fault Tolerant Quantum Computer, assuming a gate-based architecture. For a good introduction to the properties of the material see the review article here. The Hamiltonian for α-RuCl$_3$ can be most generically represented in the form

$$\begin{aligned}
H_{material} = {} & K_x \sum_{ij} S_i^x S_j^x + K_y \sum_{ij} S_i^y S_j^y + K_z \sum_{ij} S_i^z S_j^z + J \sum_{ij} \mathbf{S_i} \cdot \mathbf{S_j} \\
& + \Gamma_z \sum_{ij} (S_i^x S_j^y + S_i^y S_j^x) + \Gamma_y \sum_{ij} (S_i^z S_j^x + S_i^x S_j^z) + \Gamma_x \sum_{ij} (S_i^y S_j^z + S_i^z S_j^y) \\
& + \Gamma'_z \sum_{ij} (S_i^x S_j^z + S_i^z S_j^x) + (S_i^x S_j^y + S_i^y S_j^x) \\
& + \Gamma'_y \sum_{ij} (S_i^y S_j^x + S_i^x S_j^y) + (S_i^y S_j^z + S_i^z S_j^y) \\
& + \Gamma'_x \sum_{ij} (S_i^z S_j^x + S_i^x S_j^z) + (S_i^z S_j^y + S_i^y S_j^z) \\
& + A \sum_{i} (S_i^z)^2
\end{aligned} \tag{1}$$

The terms S_i^x, S_i^y, and S_i^z represent the Pauli operators acting on site i. The terms K_x, K_y, and K_z represent the strength of the Kitaev interaction between two sites in a given direction. The bold terms $\mathbf{S_i}$ \$ = [S^x_i, S^y_i, S^z_i]\$ are useful for defining the Heisenberg interaction terms with strength J. The terms Γ represent the strength of off-diagonal symmetric exchange interactions between nearest neighboring sites, and the terms Γ' represent the effect of trigonal distortion. Lastly, the term A represents the effect of single-ion anisotropy.

The connectivity of the Hamiltonian is defined directionally, as shown in Figure 1 (obtained from [1]). As a result of the experiment being performed on the material, we need to include a time-varying Hamiltonian component, corresponding to the time-varying Zeeman terms operating on the material. This Hamiltonian is

$$H_{field}(t) = f_x(t) \sum_i S_i^x + f_y(t) \sum_i S_i^y + f_z(t) \sum_i S_i^z \tag{2}$$

We can then construct the complete time-varying Hamiltonian we want to simulate with

$$H(t) = H_{material} + H_{field}(t) \tag{3}$$

Figure 1

```
[1]: import os
     import time
     import pandas as pd
     import numpy as np
     from math import sqrt

     import networkx as nx
     from networkx.generators.lattice import hexagonal_lattice_graph

     import cirq

     from pyLIQTR.utils.Hamiltonian import Hamiltonian as pyH
     from qca.utils.algo_utils import estimate_qsp, estimate_trotter
     from pyLIQTR.gate_decomp.cirq_transforms import clifford_plus_t_direct_transform

     from qca.utils.utils import plot_histogram, gen_resource_estimate, re_as_json
     from qca.utils.hamiltonian_utils import flatten_nx_graph,␣
      ↪assign_hexagon_labels, pyliqtr_hamiltonian_to_openfermion_qubit_operator
```

```
/Users/jonhas/anaconda3/lib/python3.11/site-packages/attr/_make.py:918:
RuntimeWarning: Running interpreter doesn't sufficiently support code object
introspection.  Some features like bare super() or accessing __class__ will not
work with slotted classes.
  set_closure_cell(cell, cls)
```

First we will define the functions that we are going to use to estimate the resources for simulating the dynamics of the Hamiltonian, H described above. Since most of the common methods Hamiltonian simulation currently in use, for example Quantum Signal Processing (QSP) and Product formulas (e.g. Trotterization), we need to first break up the Hamiltonian into time independent steps using a Magnus expansion. Here, we only use the first order Magnus expansion, which ultimately bottlenecks our precision, but we allow the user to specify a number of steps to ensure that the simulation is accurate enough for practical purposes.

1.0.1 Magnus Expansion

The operation of the first order Magnus expansion is defined as follows:

$$U(s, s_0) \approx U_{mag_1} = \exp\left(-i\int_{t0}^{t1} H(s)ds\right) \tag{4}$$

implying that we can treat the Hamiltonian as the argument of the exponential:

$$H(s, s_0) \approx H_{mag_1} = \int_{t0}^{t1} H(s)ds \tag{5}$$

Note that when H has non-commuting component terms, $H = H_1 + H_2$ where $[H_1, H_2] \neq 0$, this serves only as a first order method and higher order terms can be introduced for more precision. For the sake of simplicity we choose to use the First order approximation. As a further approximating step, in order to reduce overheads of compiling multiple circuits, we currently treat the time varying portion as a constant, representing a value somewhere in the middle of the dynamics. In the future, we will likely improve these approximations to fully explore the time varying component and the potential improvement of using higher-order magnus expansions.

1.0.2 Simulation Algorithms

In this notebook, we use two different algorithms for simulating the dynamics of H: QSP and second order Trotterization. We explore both of these methods to better compare the resource requirements for each algorithm and see where tradeoffs, such as the parallelizability of T gates, exist. It should be noted that the bounds which we use for estimating the Trotter error are loose, and it has been shown that using a lower number of steps is often sufficient. For the implementation of QSP, we use pyLIQTR and for the implementation of Trotterization, we use openfermion. Because the term ordering used by default is almost antagonistic with respect to circuit depth for most Hamiltonians with a lattice structure, we also define our own term ordering based on edge coloring. While this may not be perfectly optimal, this coloring approach tends to get rather close empirically.

Due to the large circuit sizes required for implementing both of these algorithms, we take approaches to reduce the portion of the circuit directly stored in memory at any given time, and extrapolate the results for various subcircuits to obtain an approximate final estimate. In the case of QSP, the circuits are provided in the form of high level circuit abstractions which may be decomposed later. There is a high degree of repetition of the blocks in this high level abstraction, so we take one instance of each block, decompose it to its required clifford + T operations, and multiply that count by the number of occurences of the block. For Trotterization, since each step is iterated multiple times, we simply compute the resources required for a single step and multiply those resource estimates by the number of steps.

Next, we need to define the directional labels for the hexagonal lattice and add next-nearest and next-next-nearest neighbor interactions to the graph so we can construct the Hamiltonian from the connectivity graph later on. This should match the connectivity defined in figure 1 above. Note that we only consider cross-plaquette next-next-nearest neighbor interactions. For the sake of visualization, we use a 3x3 lattice. For the actual simulations, we will use a much larger lattice, defined below.

[2]:
```python
t_init = time.perf_counter()
def assign_hexagon_labels_rucl(g):

    assign_hexagon_labels(g, 'X1', 'Y1', 'Z1')

    #Adding next nearest and next-next nearest neighbor edges and labels
    for n in g.nodes:
        r,c = n

        #next nearest neighbors
        if (r, c+2) in g:
            g.add_edge(n, (r, c+2), label = 'Z2')
        if (r+1, c+1) in g:
            g.add_edge(n, (r+1, c+1), label = "Y2")
        if (r-1, c+1) in g:
            g.add_edge(n, (r-1, c+1), label = "X2")

        #next-next nearest neighbors
        if (r+1, c) in g and not ((n, (r+1, c)) in g.edges):
            g.add_edge(n, (r+1,c), label = "Z3")
        if (r+1, c+2) in g and (r + c)%2 == 0:
            g.add_edge(n, (r+1, c+2), label="X3")
        if (r-1, c+2) in g and (r + c)%2 == 1:
            g.add_edge(n, (r-1, c+2), label="Y3")

g_rucl = hexagonal_lattice_graph(3,3)
pos = nx.get_node_attributes(g_rucl, 'pos')
assign_hexagon_labels_rucl(g_rucl)
edge_labels = dict([((n1, n2), d['label']) for n1, n2, d in g_rucl.
 ↪edges(data=True)]);
nx.draw(g_rucl, pos, with_labels=True)
nx.draw_networkx_edge_labels(g_rucl, pos,edge_labels = edge_labels);
```

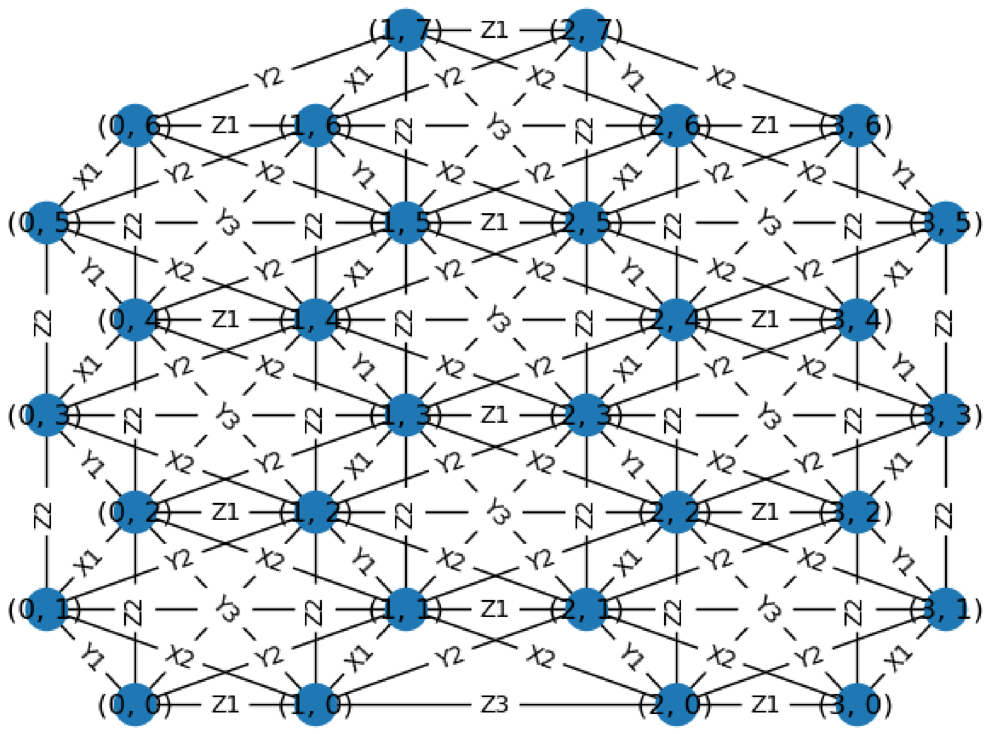


Figure 2 With the connectivity defined, we can construct a variety of Hamiltonians using the terms defined in [1]. Some of the models contain more terms and thus take longer to run. For

the purposes of demonstration, row 13 tends to have the shortest runtime, so we use that for demonstration purposes here. To run all rows, simply loop over all indices rather than just the 13th index. The number after the terms in the following graph indicate how far away the interactions are. For example J1 represents the nearest neighbor Heisenberg interactions, J2 represents the next-nearest neighbor Heisenberg interactions, and J3 represents the next-next-nearest neighbor Heisenberg interactions. Any terms not present in the table that appear in the generic Hamiltonian defined above are treated as 0.

```
[3]: rucl_references = ["Winter et al. PRB", "Winter et al. NC", "Wu et al.",␣
      ↪"Cookmeyer and Moore", "Kim and Kee", "Suzuki and Suga",
                   "Yadav et al.", "Ran et al.", "Hou et al.", "Wang et al.",␣
      ↪"Eichstaedt et al.", "Eichstaedt et al.",
                   "Eichstaedt et al.", "Banerjee et al.", "Kim et al.", "Kim and␣
      ↪Kee", "Winter et al.", "Ozel et al.", "Ozel et al."]

     rucl_methods = ["Ab initio (DFT + exact diag.)", "Ab initio-inspired (INS␣
      ↪fit)", "THz spectroscopy fit",
                     "Magnon thermal Hall (sign)", "DFT + t=U expansion", "Magnetic␣
      ↪specific heat", "Quantum chemistry (MRCI)",
                     "Spin wave fit to INS gap", "Constrained DFT + U", "DFT + t=U␣
      ↪expansion", "Fully ab initio (DFT + cRPA + t=U)",
                     "Neglecting non-local Coulomb", "Neglecting non-local SOC",␣
      ↪"Spin wave fit", "DFT + t=U expansion",
                     "DFT + t=U expansion", "Ab initio (DFT + exact diag.)", "Spin␣
      ↪wave fit/THz spectroscopy", "Spin wave fit/THz spectroscopy"]

     rucl_J1 = [-1.7, -0.5, -0.35, -0.5, -1.53, -1.53, 1.2, 0, -1.87, -0.3, -1.4, -0.
      ↪2, -1.3, -4.6, -12, -3.5, -5.5, -0.95, 0.46]
     rucl_K1 = [-6.7, -5.0, -2.8, -5.0, -6.55, -24.4, -5.6, -6.8, -10.7, -10.9, -14.
      ↪3, -4.5, -13.3, 7.0, 17., 4.6, 7.6, 1.15, -3.5]
     rucl_Gam1 = [6.6, 2.5, 2.4, 2.5, 5.25, 5.25, 1.2, 9.5, 3.8, 6.1, 9.8, 3.0, 9.4,␣
      ↪0, 12., 6.42, 8.4, 3.8, 2.35]
     rucl_Gam_prime1 = [-0.9, 0, 0, 0, -0.95, -0.95, -0.7, 0, 0, 0, -2.23, -0.73, -2.
      ↪3, 0, 0, -0.04, 0.2, 0, 0]
     rucl_J2 = [0, 0, 0, 0, 0, 0, 0.25, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
     rucl_K2 = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -0.63, -0.33, -0.67, 0, 0, 0, 0, 0, 0]
     rucl_J3 = [2.7, 0.5, 0.34, 0.1125, 0, 0, 0.25, 0, 1.27, 0.03, 1.0, 0.7, 1.0, 0,␣
      ↪0, 0, 2.3, 0, 0]
     rucl_K3 = [0, 0, 0, 0, 0, 0, 0, 0, 0.63, 0, 0.03, 0.1, 0.1, 0, 0, 0, 0, 0, 0]

     d_rucl = {'reference': rucl_references, 'method': rucl_methods, 'J1': rucl_J1,␣
      ↪'K1': rucl_K1,
               'Gam1': rucl_Gam1, 'Gam_prime1': rucl_Gam_prime1,
               'J2': rucl_J2, 'K2': rucl_K2, 'J3': rucl_J3, 'K3': rucl_K3}
     df_rucl = pd.DataFrame(d_rucl)
     df_rucl
```

```
[3]:                 reference                                  method     J1     K1  \
     0     Winter et al. PRB       Ab initio (DFT + exact diag.)  -1.70  -6.70
     1      Winter et al. NC      Ab initio-inspired (INS fit)  -0.50  -5.00
     2             Wu et al.              THz spectroscopy fit  -0.35  -2.80
     3   Cookmeyer and Moore         Magnon thermal Hall (sign)  -0.50  -5.00
     4          Kim and Kee              DFT + t=U expansion  -1.53  -6.55
     5        Suzuki and Suga           Magnetic specific heat  -1.53 -24.40
     6          Yadav et al.          Quantum chemistry (MRCI)   1.20  -5.60
     7            Ran et al.           Spin wave fit to INS gap   0.00  -6.80
     8            Hou et al.             Constrained DFT + U  -1.87 -10.70
     9           Wang et al.              DFT + t=U expansion  -0.30 -10.90
     10    Eichstaedt et al.  Fully ab initio (DFT + cRPA + t=U)  -1.40 -14.30
     11    Eichstaedt et al.      Neglecting non-local Coulomb  -0.20  -4.50
     12    Eichstaedt et al.          Neglecting non-local SOC  -1.30 -13.30
     13      Banerjee et al.                     Spin wave fit  -4.60   7.00
     14           Kim et al.              DFT + t=U expansion -12.00  17.00
     15          Kim and Kee              DFT + t=U expansion  -3.50   4.60
     16        Winter et al.       Ab initio (DFT + exact diag.)  -5.50   7.60
     17          Ozel et al.      Spin wave fit/THz spectroscopy  -0.95   1.15
     18          Ozel et al.      Spin wave fit/THz spectroscopy   0.46  -3.50

          Gam1  Gam_prime1    J2    K2      J3    K3
     0    6.60       -0.90  0.00  0.00  2.7000  0.00
     1    2.50        0.00  0.00  0.00  0.5000  0.00
     2    2.40        0.00  0.00  0.00  0.3400  0.00
     3    2.50        0.00  0.00  0.00  0.1125  0.00
     4    5.25       -0.95  0.00  0.00  0.0000  0.00
     5    5.25       -0.95  0.00  0.00  0.0000  0.00
     6    1.20       -0.70  0.25  0.00  0.2500  0.00
     7    9.50        0.00  0.00  0.00  0.0000  0.00
     8    3.80        0.00  0.00  0.00  1.2700  0.63
     9    6.10        0.00  0.00  0.00  0.0300  0.00
     10   9.80       -2.23  0.00 -0.63  1.0000  0.03
     11   3.00       -0.73  0.00 -0.33  0.7000  0.10
     12   9.40       -2.30  0.00 -0.67  1.0000  0.10
     13   0.00        0.00  0.00  0.00  0.0000  0.00
     14  12.00        0.00  0.00  0.00  0.0000  0.00
     15   6.42       -0.04  0.00  0.00  0.0000  0.00
     16   8.40        0.20  0.00  0.00  2.3000  0.00
     17   3.80        0.00  0.00  0.00  0.0000  0.00
     18   2.35        0.00  0.00  0.00  0.0000  0.00
```

Table 1 Now we may construct the Hamiltonian in a form usable by pyLIQTR using the values defined in the table above. We also allow for an additional time varying portion to represent the longitudinal field term. In the future, we would like to extend the capability of this notebook to further explore implementing this term in a more precise manner with higher order Magnus expansions and selecting multiple time-slices for a more accurate approximation.

```
[4]: def nx_rucl_terms(g, data_series):
         H = []
         n = len(g.nodes)
         for (n1,n2,d) in g.edges(data=True):
             label = d['label'][0]
             distance = int(d['label'][1])

             #Heisenberg and Kitaev terms
             if distance == 1:
                 weight_J = data_series.J1
             elif distance == 2:
                 weight_J = data_series.J2
             else:
                 weight_J = data_series.J3

             if distance == 1:
                 weight_K = data_series.K1
             elif distance == 2:
                 weight_K = data_series.K2
             else:
                 weight_K = data_series.K3

             if not (weight_J == 0 and weight_K == 0):
                 string_x = n*'I'
                 string_y = n*'I'
                 string_z = n*'I'

                 weight_x = weight_J
                 weight_y = weight_J
                 weight_z = weight_J
                 for i in [n1,n2]:
                     string_x = string_x[:i] + 'X' + string_x[i+1:]
                     string_y = string_y[:i] + 'Y' + string_y[i+1:]
                     string_z = string_z[:i] + 'Z' + string_z[i+1:]
                 if label == 'X':
                     weight_x += weight_K
                 elif label == 'Y':
                     weight_y += weight_K
                 else:
                     weight_z += weight_K
                 if weight_x != 0:
                     H.append((string_x, weight_x))
                 if weight_y != 0:
                     H.append((string_y, weight_y))
                 if weight_z != 0:
                     H.append((string_z, weight_z))
```

```
        #Gamma Terms
        if distance == 1 and data_series.Gam1 != 0:
            string_gam1_1 = n*'I'
            string_gam1_2 = n*'I'
            #unwrapping loop since there is no ordering guarantee
            labels=['X', 'Y', 'Z']
            labels.remove(label)
            l1,l2 = labels
            string_gam1_1 = string_gam1_1[:n1] + l1 + string_gam1_1[n1+1:]
            string_gam1_1 = string_gam1_1[:n2] + l2 + string_gam1_1[n2+1:]

            string_gam1_2 = string_gam1_2[:n1] + l2 + string_gam1_2[n1+1:]
            string_gam1_2 = string_gam1_2[:n2] + l1 + string_gam1_2[n2+1:]

            H.append((string_gam1_1, data_series.Gam1))
            H.append((string_gam1_2, data_series.Gam1))

        #Gamma' Terms
        if distance == 1 and data_series.Gam_prime1 != 0:
            #unwrapping inner loop since there is no ordering guarantee
            labels=['X', 'Y', 'Z']
            labels.remove(label)
            for label_offset in labels:
                string_gam1_1 = n*'I'
                string_gam1_2 = n*'I'
                l1 = label
                l2 = label_offset

                string_gam1_1 = string_gam1_1[:n1] + l1 + string_gam1_1[n1+1:]
                string_gam1_1 = string_gam1_1[:n2] + l2 + string_gam1_1[n2+1:]
                string_gam1_2 = string_gam1_2[:n1] + l2 + string_gam1_2[n1+1:]
                string_gam1_2 = string_gam1_2[:n2] + l1 + string_gam1_2[n2+1:]
                H.append((string_gam1_1, data_series.Gam_prime1))
                H.append((string_gam1_2, data_series.Gam_prime1))
    return H

def generate_time_varying_terms(g, s, x = lambda s: 0, y = lambda s: 0, z =␣
 ↪lambda s: 0):
    assert callable(x)
    assert callable(y)
    assert callable(z)

    weight_x, weight_y, weight_z = x(s), y(s), z(s)
    n = len(g)
    H = []
    if not (weight_x == 0):
        for node in g.nodes:
            string_x = n*'I'
            string_x = string_x[:node] + 'X' + string_x[node+1:]
            H.append((string_x, weight_x))
    if not (weight_y == 0):
        for node in g.nodes:
            string_y = n*'I'
            string_y = string_y[:node] + 'Y' + string_y[node+1:]
            H.append((string_y, weight_y))
    if not (weight_z == 0):
        for node in g.nodes:
            string_z = n*'I'
            string_z = string_z[:node] + 'Z' + string_z[node+1:]
            H.append((string_z, weight_z))
    return H

#using normalized time s = t_current / t_total to allow for more variation of␣
 ↪total time
def generate_rucl_hamiltonian(lattice_size, data_series, s=0, field_x=lambda s:␣
 ↪0, field_y=lambda s: 0, field_z=lambda s: 0):
    g = hexagonal_lattice_graph(lattice_size,lattice_size)
    assign_hexagon_labels_rucl(g)
    g = flatten_nx_graph(g)
    H_constant = nx_rucl_terms(g, data_series)
    H_time_varied = generate_time_varying_terms(g, s, x=field_x, y = field_y, z␣
 ↪= field_z)
    H = H_constant + H_time_varied
    return H
```

Next, we need to implement a circuit to prepare an initial estimate for the ground state of RuCl with a high degree of overlap with the actual ground state. It has been observed experimentally that at low temperatures $\alpha - RuCl_3$ corresponding to a zig-zag spin structure, seen in Figure 3. In the presence of an external magnetic field, this state is non-degenerate and can be prepared by implementing local rotations on individual sites.

The change of basis matrix from the $\hat{x}$, $\hat{y}$, $\hat{z}$ axis to the $\hat{a}$, $\hat{b}$, $\hat{c}$ axis is as follows:

$$\begin{bmatrix} \hat{a} \\ \hat{b} \\ \hat{c} \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{6}} & -\frac{2}{\sqrt{6}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} \end{bmatrix} \begin{bmatrix} \hat{x} \\ \hat{y} \\ \hat{z} \end{bmatrix} \tag{6}$$

This state may be prepared by initializing the spins in the $\hat{z}$ eigenbasis $|0...0\rangle$ then applying σ^x operations on the appropriate sites to form the zig-zag pattern. Next, the sites are rotated about the $\hat{c}$ axis until the state has maximum overlap with the $\hat{a}$ direction. The angle of rotation required in this instance is a rotation of $\theta = \pi$ about the $\hat{c}$ axis. This rotation can be achieved by performing a sequence of rotations evaluated from right to left): $R_i^z(-\pi/4) \cdot R_i^x(-\pi/4) \cdot \sigma_i^z \cdot R_i^x(\pi/4) \cdot R_i^z(\pi/4)$. We show below how one may construct this initial state below. This ground state is biased in the presence of a uniform magnetic field, representing the Zeeman terms in the $\hat{a}$ direction, making the

Zeeman terms

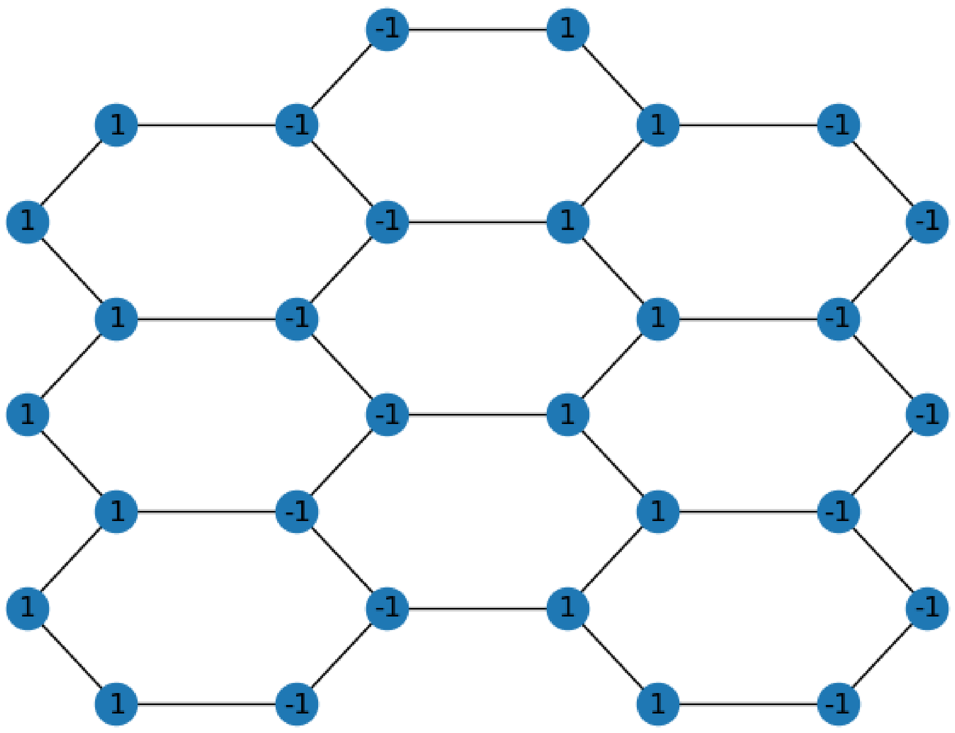
$$H_{field}(t) = t\sum_i \left(\frac{1}{\sqrt{6}}\sigma_i^x + \frac{1}{\sqrt{6}}\sigma_i^y - \frac{2}{\sqrt{6}}\sigma_i^z\right) \tag{7}$$

Once all of this has been done, the state has been prepared and the dynamics may be executed.

Figure 3

```
[5]: def assign_spin_labels_rucl(lattice_size):
         #We may omit NN and NNN couplings for the purposes of clarity in state prep
         g = hexagonal_lattice_graph(lattice_size, lattice_size)
         spin_labels = dict([(node, pow(-1, node[0])) for node in g])
         nx.set_node_attributes(g, spin_labels, name="spin")
         return g

     def prepare_initial_state_rucl(g):
         #site labels are assumed to be the same as for hamiltonian with NN and NNN␣
      ↪couplings
         #because they come from the same base generator
         g = flatten_nx_graph(g)
         spins = nx.get_node_attributes(g,"spin")
         qubits = [cirq.LineQubit(i) for i in g.nodes]

         #generating layers of operations to initialize state
         layer_zig_zag = [cirq.X(qubits[i]) for i in range(len(g)) if spins[i] == -1]
         layer_rz = [cirq.Rz(rads = np.pi/4).on(qubit) for qubit in qubits]
         layer_rx = [cirq.Rx(rads = np.pi/4).on(qubit) for qubit in qubits]
         layer_z = [cirq.Z(qubit) for qubit in qubits]
         layer_rx_neg = [cirq.Rx(rads=-np.pi/4).on(qubit) for qubit in qubits]
         layer_rz_neg = [cirq.Rz(rads=-np.pi/4).on(qubit) for qubit in qubits]

         #appending layers to a circuit to return
         circuit = cirq.Circuit()
         circuit.append(layer_zig_zag)
         circuit.append(layer_rz)
         circuit.append(layer_rx)
         circuit.append(layer_z)
         circuit.append(layer_rx_neg)
         circuit.append(layer_rz_neg)
         return circuit

     g_spins = assign_spin_labels_rucl(3)
     circuit_state_prep = prepare_initial_state_rucl(g_spins)
     pos = nx.get_node_attributes(g_spins, 'pos')
     spin_labels = nx.get_node_attributes(g_spins, "spin")
     nx.draw(g_spins, labels = spin_labels, pos=pos, with_labels=True)
     print(circuit_state_prep)
```

```
0:
  Rz(0.25 )  Rx(0.25 )  Z      Rx(-0.25 )  Rz(-0.25 )

1:
  Rz(0.25 )  Rx(0.25 )  Z      Rx(-0.25 )  Rz(-0.25 )

2:
  Rz(0.25 )  Rx(0.25 )  Z      Rx(-0.25 )  Rz(-0.25 )

3:
  Rz(0.25 )  Rx(0.25 )  Z      Rx(-0.25 )  Rz(-0.25 )

4:
  Rz(0.25 )  Rx(0.25 )  Z      Rx(-0.25 )  Rz(-0.25 )

5:
  Rz(0.25 )  Rx(0.25 )  Z      Rx(-0.25 )  Rz(-0.25 )

6:
  Rz(0.25 )  Rx(0.25 )  Z      Rx(-0.25 )  Rz(-0.25 )

7:
  X      Rz(0.25 )  Rx(0.25 )  Z       Rx(-0.25 )  Rz(-0.25 )

8:
  X      Rz(0.25 )  Rx(0.25 )  Z       Rx(-0.25 )  Rz(-0.25 )

9:
  X      Rz(0.25 )  Rx(0.25 )  Z       Rx(-0.25 )  Rz(-0.25 )

10:
  X      Rz(0.25 )  Rx(0.25 )  Z       Rx(-0.25 )  Rz(-0.25 )

11:
  X      Rz(0.25 )  Rx(0.25 )  Z       Rx(-0.25 )  Rz(-0.25 )

12:
  X      Rz(0.25 )  Rx(0.25 )  Z       Rx(-0.25 )  Rz(-0.25 )

13:
  X      Rz(0.25 )  Rx(0.25 )  Z       Rx(-0.25 )  Rz(-0.25 )

14:
  X      Rz(0.25 )  Rx(0.25 )  Z       Rx(-0.25 )  Rz(-0.25 )

15:
  Rz(0.25 )  Rx(0.25 )  Z      Rx(-0.25 )  Rz(-0.25 )

16:
 Rz(0.25 )  Rx(0.25 )  Z      Rx(-0.25 )  Rz(-0.25 )

17:
 Rz(0.25 )  Rx(0.25 )  Z      Rx(-0.25 )  Rz(-0.25 )

18:
 Rz(0.25 )  Rx(0.25 )  Z      Rx(-0.25 )  Rz(-0.25 )

19:
 Rz(0.25 )  Rx(0.25 )  Z      Rx(-0.25 )  Rz(-0.25 )

20:
 Rz(0.25 )  Rx(0.25 )  Z      Rx(-0.25 )  Rz(-0.25 )

21:
 Rz(0.25 )  Rx(0.25 )  Z      Rx(-0.25 )  Rz(-0.25 )

22:
 Rz(0.25 )  Rx(0.25 )  Z      Rx(-0.25 )  Rz(-0.25 )

23:
 X      Rz(0.25 )  Rx(0.25 )  Z      Rx(-0.25 )  Rz(-0.25 )

24:
 X      Rz(0.25 )  Rx(0.25 )  Z      Rx(-0.25 )  Rz(-0.25 )

25:
 X      Rz(0.25 )  Rx(0.25 )  Z      Rx(-0.25 )  Rz(-0.25 )

26:
 X      Rz(0.25 )  Rx(0.25 )  Z      Rx(-0.25 )  Rz(-0.25 )

27:
 X      Rz(0.25 )  Rx(0.25 )  Z      Rx(-0.25 )  Rz(-0.25 )

28:
 X      Rz(0.25 )  Rx(0.25 )  Z      Rx(-0.25 )  Rz(-0.25 )

29:
 X      Rz(0.25 )  Rx(0.25 )  Z      Rx(-0.25 )  Rz(-0.25 )
```

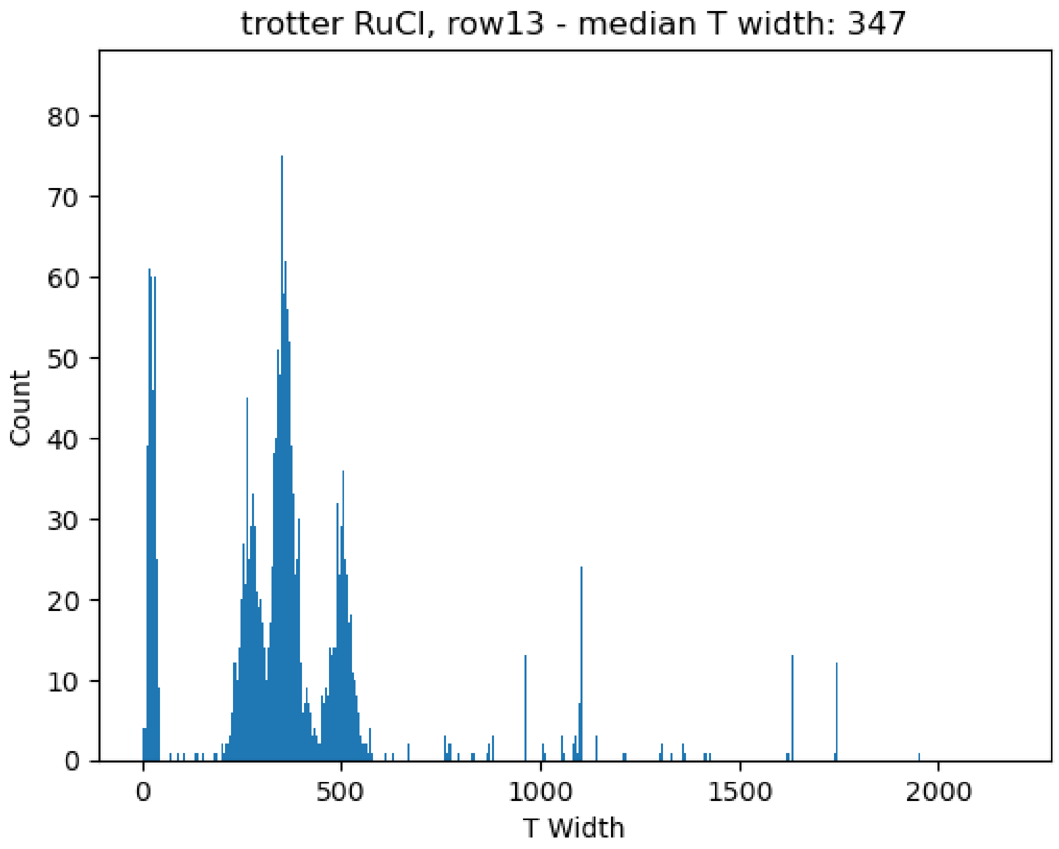


Finally, it is time to implement the circuit for simulating the time dynamics of H, parameterized by the rows of Table 1. First we perform the simulation using the second order Trotter expansion. We choose an energy precision of 1e-3, which is believed to be sufficiently low to allow for lower circuit depths while still getting empirically accurate results. We construct a lattice of size 32x32 honeycomb cells to hopefully have a sufficiently large mass to mitigate finite size effects. This is believed to be the lower end of what lattice size would be useful, and larger may be better in some cases. Finally, we plot the a histogram of the T-widths that occur on each layer of the circuit. The x-axis representing the number of parallel T-gates and the y-axis representing the number of layers where that T-width is found. This is done to determine the value of the parallelizability of T factories on a theoretical Fault Tolerant Device using a Clifford + T gateset.

[6]:
```
#Commenting out range, to avoid long runtimes.  This still takes ~20-30 minutes␣
 ↪as written
for i in [13]:
#for i in range(19):
    #defining precision required for the trotterized circuit
    energy_precision = 1e-3

    figdir="Trotter/Figures/"
    if not os.path.exists(figdir):
        os.makedirs(figdir)
    widthdir = "Trotter/Widths/"
    if not os.path.exists(widthdir):
        os.makedirs(widthdir)

    timesteps=1000
    H_rucl = generate_rucl_hamiltonian(32, df_rucl.iloc[i], field_x=lambda s: 1/
  ↪sqrt(6), field_y=lambda s: 1/sqrt(6), field_z=lambda s: -2/sqrt(6))
    H_rucl_pyliqtr = pyH(H_rucl)
    openfermion_hamiltonian_rucl =␣
  ↪pyliqtr_hamiltonian_to_openfermion_qubit_operator(H_rucl_pyliqtr)

    print("Estimating RuCl row " + str(i) + " using Trotterization")
    t0 = time.perf_counter()
    cpt_trotter_rucl = estimate_trotter(openfermion_hamiltonian_rucl,␣
  ↪timesteps, energy_precision, "Trotter/RuCl_circuits/",␣
  ↪hamiltonian_name="rucl_" + str(i), write_circuits=True)
    t1 = time.perf_counter()
    elapsed = t1 - t0
    print("Total time to estimate RuCl row " + str(i) + ": " + str(elapsed) + "␣
  ↪seconds\n")

    plot_histogram(cpt_trotter_rucl,
                   f'trotter RuCl, row{i}',
                   figdir,
                   widthdir,
                   0)
```

```
Estimating RuCl row 13 using Trotterization
Time to estimate number of trotter steps required (215647636): 43.20067066699994
seconds
Time to find term ordering: 0.026525416000036784 seconds
Time to generate trotter circuit from openfermion: 1.4170000213198364e-06
seconds
Time to generate a clifford + T circuit from trotter circuit: 178.4242624159997
seconds
   Time to decompose high level <class 'cirq.ops.common_gates.HPowGate circuit:
0.00010033299986389466 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.HPowGate circuit
to Clifford+T: 1.8624999993876554e-05 seconds
   Time to decompose high level <class 'cirq.ops.common_gates.Rz circuit:
5.6416000006720424e-05 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.Rz circuit to
Clifford+T: 0.008796791998975095 seconds
   Time to decompose high level <class 'cirq.ops.common_gates.Rx circuit:
6.0416999986045994e-05 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.Rx circuit to
Clifford+T: 7.654199907847214e-05 seconds
```

```
   Time to decompose high level <class 'cirq.ops.common_gates.CXPowGate circuit:
9.479099935560953e-05 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.CXPowGate circuit
to Clifford+T: 0.00013829199997417163 seconds
Total time to estimate RuCl row 13: 229.6507289580004 seconds
```



Finally we perform the same experiment for the Quantum Signal Processing algorithm. We omit plotting in this case because we perform estimation of the circuit by decomposing sub-circuits of the overall algorithm. This results in some sub-circuits where parallelizability may appear deceptively high, or no T gates may even be present. Emperically, for the portions of the circuit which compose the bulk of the circuit (SELECT and REFLECT operators), the T width tends to be near 1.

```
[7]: #Commenting out range, to avoid long runtimes.  This still takes ~20-30 minutes␣
      ↪as written
     for i in [13]:
     #for i in range(19):
         #defining precision required for the trotterized circuit
         energy_precision = 1e-3
         figdir="QSP/Figures/"
```

```
    if not os.path.exists(figdir):
        os.makedirs(figdir)
    widthdir = "QSP/Widths/"
    if not os.path.exists(widthdir):
        os.makedirs(widthdir)
    timesteps=1000
    H_rucl = generate_rucl_hamiltonian(32, df_rucl.iloc[i], field_x=lambda s: 1/
↪sqrt(6), field_y=lambda s: 1/sqrt(6), field_z=lambda s: -2/sqrt(6))
    H_rucl_pyliqtr = pyH(H_rucl)

    print(f'Estimating RuCl row {i} using QSP')
    t0 = time.perf_counter()
    # change to Clifford+T dictionary in future revision
    qsp_high_level_rucl = estimate_qsp(H_rucl_pyliqtr,
                                       timesteps,
                                       energy_precision,
                                       'QSP/RuCl_circuits/',
                                       hamiltonian_name="rucl_" + str(i),
                                       write_circuits=True)
    t1 = time.perf_counter()
    elapsed = t1 - t0
    print(f'Time to estimate RuCl row {i}: {elapsed} seconds\n')
    print()
```

```
Estimating RuCl row 13 using QSP
Time to generate high level QSP circuit: 112.33004700000129 seconds
   Time to decompose high level <class 'cirq.ops.pauli_gates._PauliX circuit:
9.120899994741194e-05 seconds
   Time to transform decomposed <class 'cirq.ops.pauli_gates._PauliX circuit to
Clifford+T: 1.9832999896607362e-05 seconds
   Time to decompose high level <class 'cirq.ops.common_gates.Rx circuit:
6.0166999901412055e-05 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.Rx circuit to
Clifford+T: 0.008287042001029477 seconds
   Time to decompose high level <class
'pyLIQTR.circuits.operators.hamiltonian_encodings.UnitaryBlockEncode circuit:
0.49925824999991164 seconds
   Time to transform decomposed <class
'pyLIQTR.circuits.operators.hamiltonian_encodings.UnitaryBlockEncode circuit to
Clifford+T: 121.69983987500018 seconds
   Time to decompose high level <class 'cirq.ops.common_gates.Ry circuit:
0.0043826250002894085 seconds
   Time to transform decomposed <class 'cirq.ops.common_gates.Ry circuit to
Clifford+T: 0.0054800420002720784 seconds
   Time to decompose high level <class 'cirq.ops.raw_types._InverseCompositeGate
circuit: 17.243875832999038 seconds
   Time to transform decomposed <class 'cirq.ops.raw_types._InverseCompositeGate
circuit to Clifford+T: 112.83102729199891 seconds
```

```
/Users/jonhas/anaconda3/lib/python3.11/site-
packages/numpy/linalg/linalg.py:2154: RuntimeWarning: divide by zero encountered
in det
  r = _umath_linalg.det(a, signature=signature)
/Users/jonhas/anaconda3/lib/python3.11/site-
packages/numpy/linalg/linalg.py:2154: RuntimeWarning: invalid value encountered
in det
  r = _umath_linalg.det(a, signature=signature)
```

```
   Time to decompose high level <class
'pyLIQTR.circuits.operators.reflect.Reflect circuit: 0.023523791998741217
seconds
   Time to transform decomposed <class
'pyLIQTR.circuits.operators.reflect.Reflect circuit to Clifford+T:
0.009446083000511862 seconds
Time to estimate RuCl row 13: 513.2348777919997 seconds
```

The final observable being measured is net magnetization, which can be obtained by simply measuring all of the qubits in the desired basis (in this case, the $\hat{a}$, $\hat{b}$, $\hat{c}$ basis), and summing the spins on all of the sites. Since the computational basis is in the $\hat{z}$ direction, we will need to implement a change of basis operation. For the purposes of this demonstration, we will use the measure in the $\hat{c}$ direction, for which the transformation $R_i^x(\pi/4) \cdot R_i^z(\pi/4)$ applied on each qubit before measurement is sufficient. This resource requirement, like the requirement for running the state preparation is constant, both in the sense that it needs to be performed for every circuit and in the sense that depth added to the circuit is $\mathcal{O}(1)$. For this reason, we perform the resource estimates for the state preparation and observable measurement separately from the dynamic circuits which are implemented based on Hamiltonian parameterizations. Below, we provide the resource estimates for both the state preparation and measurement (the state preparation circuit generator is defined above)

[8]:
```python
def prepare_measurement_rucl(g):
    #site labels are assumed to be the same as for hamiltonian with NN and NNN␣
 ↪couplings
    #because they come from the same base generator
    g = flatten_nx_graph(g)
    qubits = [cirq.LineQubit(i) for i in g.nodes]

    #generating layers of operations to initialize state
    layer_rz = [cirq.Rz(rads = np.pi/4).on(qubit) for qubit in qubits]
    layer_rx = [cirq.Rx(rads = np.pi/4).on(qubit) for qubit in qubits]
    layer_measurement = [cirq.measure(qubit) for qubit in qubits]
    #appending layers to a circuit to return
    circuit = cirq.Circuit()
    circuit.append(layer_rz)
    circuit.append(layer_rx)
    circuit.append(layer_measurement)
    return circuit

#example instance for clear printing and verification of the correct operators
circuit_measurement = prepare_measurement_rucl(g_spins)
print(circuit_measurement)
```

```
0:   Rz(0.25 )  Rx(0.25 )  M

1:   Rz(0.25 )  Rx(0.25 )  M

2:   Rz(0.25 )  Rx(0.25 )  M

3:   Rz(0.25 )  Rx(0.25 )  M

4:   Rz(0.25 )  Rx(0.25 )  M

5:   Rz(0.25 )  Rx(0.25 )  M

6:   Rz(0.25 )  Rx(0.25 )  M

7:   Rz(0.25 )  Rx(0.25 )  M

8:   Rz(0.25 )  Rx(0.25 )  M

9:   Rz(0.25 )  Rx(0.25 )  M

10:   Rz(0.25 )  Rx(0.25 )  M

11:   Rz(0.25 )  Rx(0.25 )  M

12:   Rz(0.25 )  Rx(0.25 )  M

13:   Rz(0.25 )  Rx(0.25 )  M

14:   Rz(0.25 )  Rx(0.25 )  M

15:   Rz(0.25 )  Rx(0.25 )  M

16:   Rz(0.25 )  Rx(0.25 )  M

17:   Rz(0.25 )  Rx(0.25 )  M

18:   Rz(0.25 )  Rx(0.25 )  M

19:   Rz(0.25 )  Rx(0.25 )  M

20:   Rz(0.25 )  Rx(0.25 )  M
```

```
21:   Rz(0.25 )  Rx(0.25 )  M

22:   Rz(0.25 )  Rx(0.25 )  M

23:   Rz(0.25 )  Rx(0.25 )  M

24:   Rz(0.25 )  Rx(0.25 )  M

25:   Rz(0.25 )  Rx(0.25 )  M

26:   Rz(0.25 )  Rx(0.25 )  M

27:   Rz(0.25 )  Rx(0.25 )  M

28:   Rz(0.25 )  Rx(0.25 )  M

29:   Rz(0.25 )  Rx(0.25 )  M
```

[9]:
```
g_spins = assign_spin_labels_rucl(32)
circuit_state_prep_and_measurement = prepare_initial_state_rucl(g_spins) +␣
 ↪prepare_measurement_rucl(g_spins)
circuit_state_prep_and_measurement_cpt =␣
 ↪clifford_plus_t_direct_transform(circuit_state_prep_and_measurement)
```

[10]:
```
state_prep_estimate =␣
 ↪gen_resource_estimate(circuit_state_prep_and_measurement_cpt)
re_as_json(state_prep_estimate, [], file_name =␣
 ↪'state_preparation_and_measurement.json')
```

[11]:
```
t_end = time.perf_counter()
print("Total time to run notebook (only using the fastest row): " + str(t_end -␣
 ↪t_init))
```

```
Total time to run notebook (only using the fastest row): 750.3481668330005
```

1.0.3 References

[1] Laurell, P., Okamoto, S. Dynamical and thermal magnetic properties of the Kitaev spin liquid candidate -RuCl3. npj Quantum Mater. 5, 2 (2020). https://doi.org/10.1038/s41535-019-0203-y