Fermi-Hubbard Model simulation

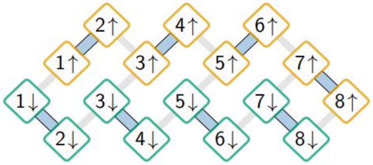
A. Foroni, G. Manganelli

16/05/2022

Table of contents

Introduction

The Google Quantum AI group employed a fast calibration method to simulate 8 fermions in 1-D, on a superconducting quantum processor, in a highly excited regime. They observed a phenomenon known as spin-charge separation.



Our aim is reproducing those results computationally.

Fermi-Hubbard Model

The Fermi-Hubbard model on a one-dimensional lattice with open boundary conditions is defined by the Hamiltonian:

$$H = -J \sum_{j=1}^{L-1} \sum_{\nu=\uparrow,\downarrow} a^{\dagger}_{j,\nu} a_{j+1,\nu} + h.c.$$

$$+U \sum_{j=1}^{L} n_{j,\uparrow} n_{j,\downarrow} + \sum_{j=1}^{L} \sum_{\nu=\uparrow\downarrow} \epsilon_{j,\nu} n_{j,\nu}$$

where $a_{j,\nu}$ and $a^{\dagger}_{j,\nu}$ are the fermionic annihilation and creation operators associated to site number j and spin state, $n_{j,\nu} = a^{\dagger}_{j,\nu} a_{j,\nu}$ are the number operators.

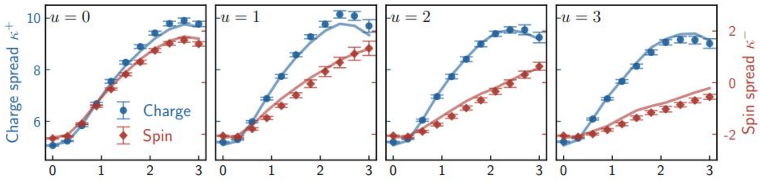
Fermi-Hubbard Model

A remarkable property of the one-dimensional Fermi-Hubbard model is spin-charge separation. In order to quantify the degree that charge and spin densities spread from the middle of the chain it is used

$$\kappa_t^{\pm} = \sum_{j=1}^{L} |j - \frac{L+1}{2}| \rho_{j,t}^{\pm}$$

where $\rho_{j,t}^{\pm}$ are the charge and spin densities at site j and time t, defined as

$$\rho_j^{\pm} = \langle n_{j,\uparrow} \rangle \pm \langle n_{j,\downarrow} \rangle$$



Qubit Mappings

- N fermions can be represented with anticommuting fermionic operators $\{a_j\}_{j=1,\dots,N}$ satisfying the canonical anticommutation relations

$$\{a_p, a_q\} = 0$$

$$\{a_p, a_q^\dagger\} = \delta_{p,q}$$

- Qubit operators are written in terms of Pauli matrices X, Y, Z

One qubit $\rightarrow \{(1,0)^T, (0,1)^T\} \implies a = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$

$$\frac{1}{2}(X + iY) = a$$

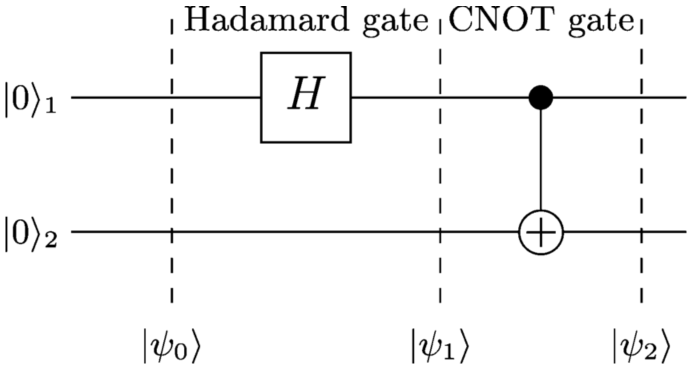
Generalizing to more than one qubit: parity is needed to reproduce anticommutation relations

$$Z|1\rangle = -|1\rangle \rightarrow a_j = Z_{(1)} \otimes Z_{(2)} \cdots \otimes a \otimes \mathbb{I}_{(2N-1)} \cdots \otimes \mathbb{I}_{(2N)}$$

counting how many qubits precede the $j - th$ one.

$\uparrow, \downarrow$ spin on a site: 2 qubits per site needed

Quantum Circuits



Computation modelled as a sequence of gates acting on qubits

Implementing the operators

$$c_j = Z_{(1)} \otimes Z_{(2)} \cdots \otimes \frac{X_j + iY_j}{2} \otimes \mathbb{I}_{(2N-1)} \cdots \otimes \mathbb{I}_{(2N)}$$

$$n_j = \mathbb{I}_{(1)} \otimes \cdots \otimes \frac{\mathbb{I} - Z}{2}_{(j)} \otimes \cdots \otimes \mathbb{I}_{(2N)}$$

In terms of a matrix representation, $a = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$ and $n = a^\dagger a = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$

Brief Hardware Overview

Programs written in Cirq can run on quantum computers in Google's labs in Santa Barbara, CA.

The experiment ran on a processor called Google Rainbow.

Gates

Time evolution is implemented with

- CPHASE(ϕ), $\phi = \frac{\tau U}{\hbar}$

$$e^{-i\phi} = \begin{cases} e^{-i\frac{U}{\hbar}\tau \cdot n_j n_{j+1}} \text{ if } n_j = n_{j+1} = 1 \\ 1 \text{ otherwise} \end{cases}$$

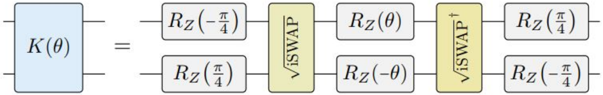
- K(θ), custom gate to implement hopping
- CPHASE(φ), $\varphi \approx (0.138 \pm 0.015)$ *rad*, to implement a parasitic phase $Vn_{j,\nu}n_{j+1,\nu}$ between first neighbours.

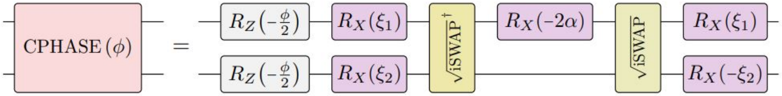
Gates decomposition for the experiment

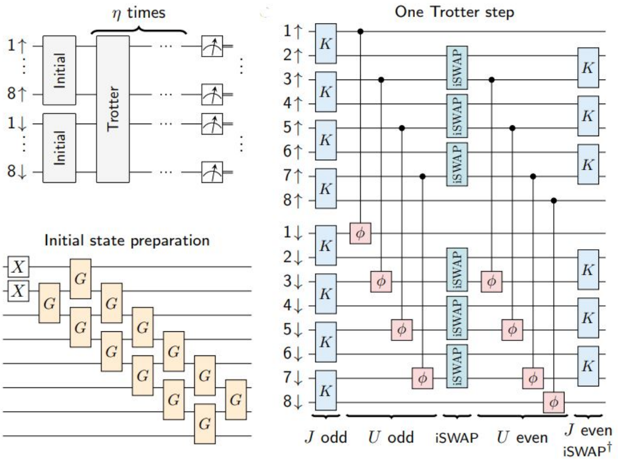
$$K = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -i\sin\theta & 0 \\ 0 & -i\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \phi = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{-i\phi} \end{pmatrix}$$

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \text{iSWAP} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & i & 0 \\ 0 & i & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Implemented with the native processor operations







1. Hopping and on-site interaction for odd sites
2. iSWAP changing odd and even qubits
3. hopping interaction as before and swap the qubits back

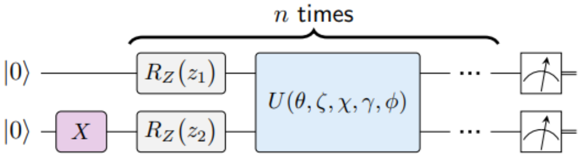
Calibration

Calibrating quantum circuits

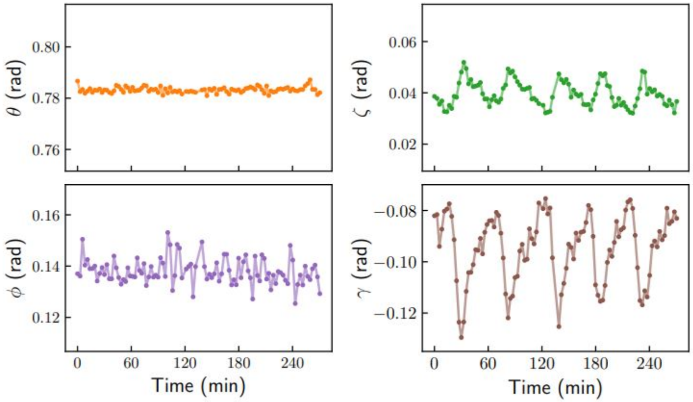
- high accuracy (system errors add up quickly)
- faster than drifts
- have the same structure as the circuit to calibrate, to capture crosstalking between gates

Floquet Calibration

Very fast calibration, $O(1min)$ per calibrated moment



The gate to calibrate, $U(\theta, \zeta, \chi, \gamma, \phi)$, is repeated many times to add up even tiny errors reaching $\approx 10^{-3}$*rad* precision on gate parameters.

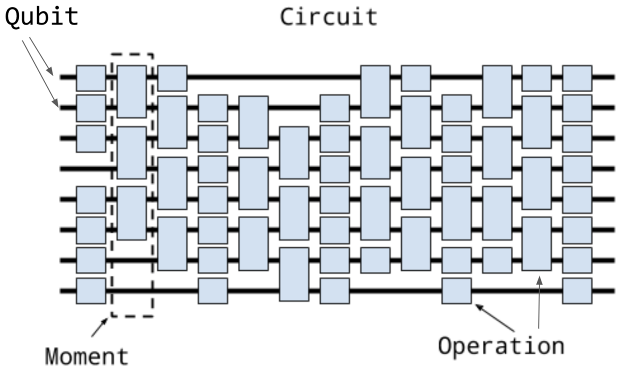


Cirq

Circuits are represented as a *Circuit* object, which is a set of *Moments*:

1. *Qubits*
2. *Operation*: an effect on a subset of *Qubits*
3. *Moment*: set of *Operation*s acting on the same time slice

*Gate*s acting on *Qubits* define an *Operation*.



Openfermion

- *FermionOperator* Specified in the basis of $a^\dagger, a$, encoded in a *tuple* of 2-*tuple*s

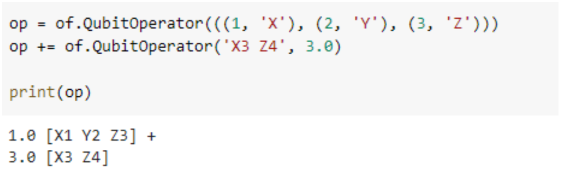
$$(j \in \mathbb{N}, \gamma \in \mathbb{Z}_2) \rightarrow \begin{cases} a_j^\dagger \text{ if } \gamma = 1 \\ a_j \text{ if } \gamma = 0 \end{cases}$$

$$((4,1),(8,0),(2,0)) \rightarrow a_4^\dagger a_8 a_2$$

- *get_sparse_operator*: converts a FermionOperator or QubitOperator to a *scipy.sparse.csc* matrix.

Openfermion

- Qubit operator: same as FermionOperator, but the possible actions are 'X', 'Y', and 'Z' instead of 1 and 0



```
op = of.QubitOperator(((1, 'X'), (2, 'Y'), (3, 'Z')))
op += of.QubitOperator('X3 Z4', 3.0)

print(op)
```

```
1.0 [X1 Y2 Z3] +
3.0 [X3 Z4]
```

FQE

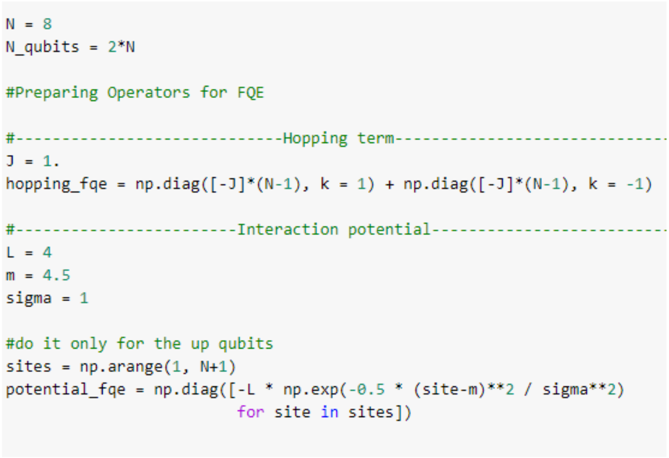
More efficient than Cirq's *Simulator* because it exploits symmetries.

- *Wavefunction*: Direct sum of fixed N, S_z sectors, specified with a *tuple* of 3-*tuple* $[N_{el},\ S_z,\ N_{orbitals}]$
  The quantum state $|0111\rangle_\uparrow |0111\rangle_\downarrow$ would be in [6, 0, 4]
- *evolve_fqe_givens_sector*: evolves a *Wavefunction* through a unitary $N \times N$ *np.ndarray*

α, β are $\uparrow, \downarrow$ qubits, respectively. Theory and more details can be found in [4].

Code

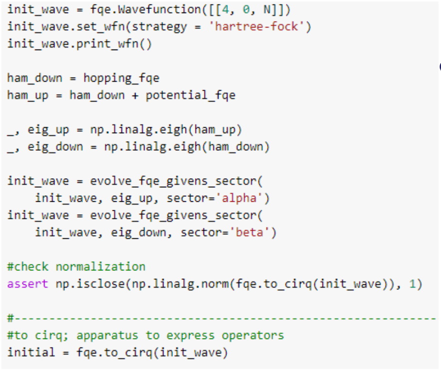
1. Experiment parameters



```
N = 8
N_qubits = 2*N

#Preparing Operators for FQE

#-----------------------------Hopping term-----------------------------
J = 1.
hopping_fqe = np.diag([-J]*(N-1), k = 1) + np.diag([-J]*(N-1), k = -1)

#------------------------Interaction potential------------------------
L = 4
m = 4.5
sigma = 1

#do it only for the up qubits
sites = np.arange(1, N+1)
potential_fqe = np.diag([-L * np.exp(-0.5 * (site-m)**2 / sigma**2)
                         for site in sites])
```

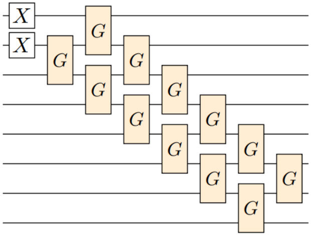
Code-initialization

2. Wave function initialization



```
init_wave = fqe.Wavefunction([[4, 0, N]])
init_wave.set_wfn(strategy = 'hartree-fock')
init_wave.print_wfn()

ham_down = hopping_fqe
ham_up = ham_down + potential_fqe

_, eig_up = np.linalg.eigh(ham_up)
_, eig_down = np.linalg.eigh(ham_down)

init_wave = evolve_fqe_givens_sector(
    init_wave, eig_up, sector='alpha')
init_wave = evolve_fqe_givens_sector(
    init_wave, eig_down, sector='beta')

#check normalization
assert np.isclose(np.linalg.norm(fqe.to_cirq(init_wave)), 1)

#---------------------------------------------------------------
#to cirq; apparatus to express operators
initial = fqe.to_cirq(init_wave)
```

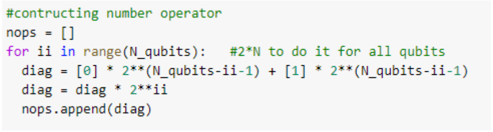
- The state lies in the sector $S_z = 0, Q = 4$: $N \uparrow = N \downarrow = 2$
- FQE prepares the state filling particle orbitals;



Code-Simulation

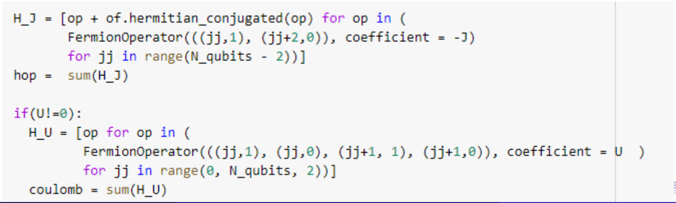
3. Number operator and Hamiltonian initialization

$$\mathbb{I} \otimes n = diag(01|01) : \mathbb{I}\otimes \text{ repeats the diagonal}$$
$$n \otimes \mathbb{I} = diag(00|11); \otimes\mathbb{I} \text{ doubles the half diagonals.}$$



```
#contructing number operator
nops = []
for ii in range(N_qubits):   #2*N to do it for all qubits
  diag = [0] * 2**(N_qubits-ii-1) + [1] * 2**(N_qubits-ii-1)
  diag = diag * 2**ii
  nops.append(diag)
```

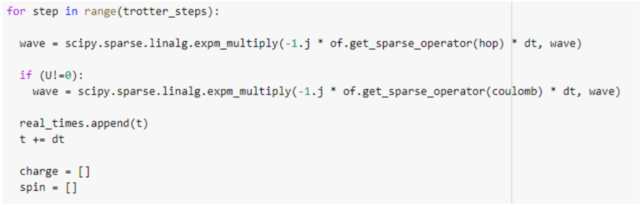
Then Hamiltonian is defined by the hopping term and interaction term separately:



```
H_J = [op + of.hermitian_conjugated(op) for op in (
       FermionOperator(((jj,1), (jj+2,0)), coefficient = -J)
       for jj in range(N_qubits - 2))]
hop =  sum(H_J)

if(U!=0):
  H_U = [op for op in (
         FermionOperator(((jj,1), (jj,0), (jj+1, 1), (jj+1,0)), coefficient = U  )
         for jj in range(0, N_qubits, 2))]
  coulomb = sum(H_U)
```

Trotter step

3. Temporal evolution and computation of expectation values



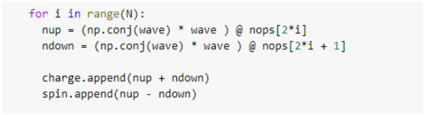
```
for step in range(trotter_steps):

  wave = scipy.sparse.linalg.expm_multiply(-1.j * of.get_sparse_operator(hop) * dt, wave)

  if (U!=0):
    wave = scipy.sparse.linalg.expm_multiply(-1.j * of.get_sparse_operator(coulomb) * dt, wave)

  real_times.append(t)
  t += dt

  charge = []
  spin = []
```

Hopping (H_1) and Coulomb (H_2) time evolutions are implemented separadetely at each time step,

$$H = H_1 + H_2 \rightarrow exp(-iH\epsilon) \approx \prod_{j=1}^{2} exp(-iH_j\epsilon)$$

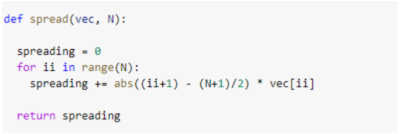
Expected values computation

Charge and spin densities computation:

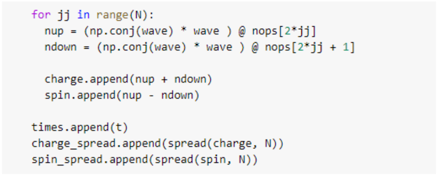


```
for i in range(N):
  nup = (np.conj(wave) * wave ) @ nops[2*i]
  ndown = (np.conj(wave) * wave ) @ nops[2*i + 1]

  charge.append(nup + ndown)
  spin.append(nup - ndown)
```

Charge and spin densities spread function and computation:

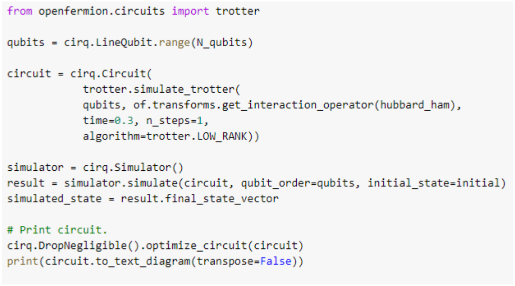


```
def spread(vec, N):

  spreading = 0
  for ii in range(N):
    spreading += abs((ii+1) - (N+1)/2) * vec[ii]

  return spreading
```



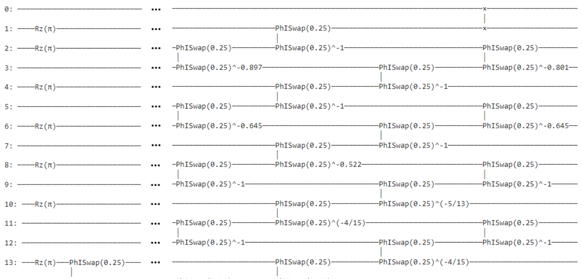
```
for jj in range(N):
  nup = (np.conj(wave) * wave ) @ nops[2*jj]
  ndown = (np.conj(wave) * wave ) @ nops[2*jj + 1]

  charge.append(nup + ndown)
  spin.append(nup - ndown)

times.append(t)
charge_spread.append(spread(charge, N))
spin_spread.append(spread(spin, N))
```

Trotter Steps as a Quantum Circuit



```python
from openfermion.circuits import trotter

qubits = cirq.LineQubit.range(N_qubits)

circuit = cirq.Circuit(
            trotter.simulate_trotter(
            qubits, of.transforms.get_interaction_operator(hubbard_ham),
            time=0.3, n_steps=1,
            algorithm=trotter.LOW_RANK))

simulator = cirq.Simulator()
result = simulator.simulate(circuit, qubit_order=qubits, initial_state=initial)
simulated_state = result.final_state_vector

# Print circuit.
cirq.DropNegligible().optimize_circuit(circuit)
print(circuit.to_text_diagram(transpose=False))
```

Trotter step circuit[2], simulated through Cirq

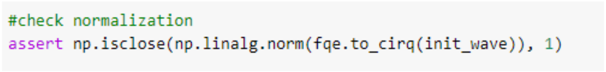


```
0:  ────────────────────────────  ...  ─────────────────────────────────────────────────────────────────────────x──────────────────────
                                                                                                                │
1:  ───Rz(π)────────────────────  ...  ─────────────────────────PhISwap(0.25)───────────────────────────────────x──────────────────────
                                                                │
2:  ───Rz(π)────────────────────  ...  ─PhISwap(0.25)───────────PhISwap(0.25)^-1────────────────────────────────PhISwap(0.25)──────────
                                        │                                                                       │
3:  ────────────────────────────  ...  ─PhISwap(0.25)^-0.897────────────────────────────PhISwap(0.25)───────────PhISwap(0.25)^-0.801───
                                                                                        │
4:  ───Rz(π)────────────────────  ...  ─────────────────────────PhISwap(0.25)───────────PhISwap(0.25)^-1───────────────────────────────
                                                                │
5:  ────────────────────────────  ...  ─PhISwap(0.25)───────────PhISwap(0.25)^-1────────────────────────────────PhISwap(0.25)──────────
                                        │                                                                       │
6:  ───Rz(π)────────────────────  ...  ─PhISwap(0.25)^-0.645────────────────────────────PhISwap(0.25)───────────PhISwap(0.25)^-0.645───
                                                                                        │
7:  ────────────────────────────  ...  ─────────────────────────PhISwap(0.25)───────────PhISwap(0.25)^-1───────────────────────────────
                                                                │
8:  ───Rz(π)────────────────────  ...  ─PhISwap(0.25)───────────PhISwap(0.25)^-0.522────────────────────────────PhISwap(0.25)──────────
                                        │                                                                       │
9:  ────────────────────────────  ...  ─PhISwap(0.25)^-1────────────────────────────────PhISwap(0.25)───────────PhISwap(0.25)^-1───────
                                                                                        │
10: ───Rz(π)────────────────────  ...  ─────────────────────────PhISwap(0.25)───────────PhISwap(0.25)^(-5/13)──────────────────────────
                                                                │
11: ────────────────────────────  ...  ─PhISwap(0.25)───────────PhISwap(0.25)^(-4/15)───────────────────────────PhISwap(0.25)──────────
                                        │                                                                       │
12: ────────────────────────────  ...  ─PhISwap(0.25)^-1────────────────────────────────PhISwap(0.25)───────────PhISwap(0.25)^-1───────
                                                                                        │
13: ──Rz(π)───PhISwap(0.25)─────  ...  ─────────────────────────PhISwap(0.25)───────────PhISwap(0.25)^(-4/15)──────────────────────────
              │                                                 │
```

Code checks

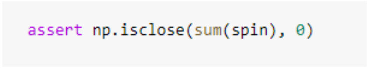
Three main checks are implemented inside the code:

1. Wavefunction normalization



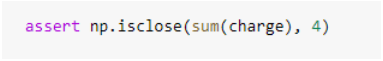
```
#check normalization
assert np.isclose(np.linalg.norm(fqe.to_cirq(init_wave)), 1)
```

2. Spin Conservation ($S_z = 0$)



```
assert np.isclose(sum(spin), 0)
```

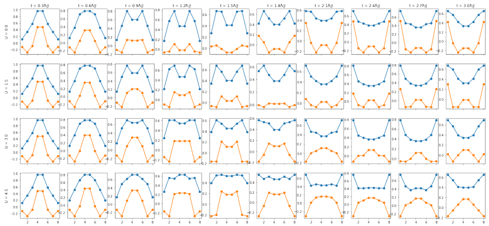
3. Charge conservation ($Q = 4$)



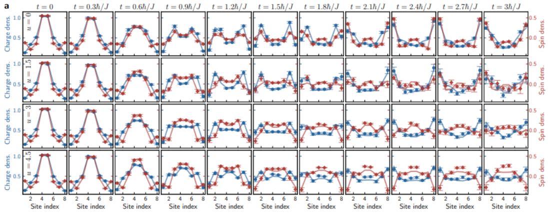
```
assert np.isclose(sum(charge), 4)
```

Results

Charge and spin densities time evolution, varying U:

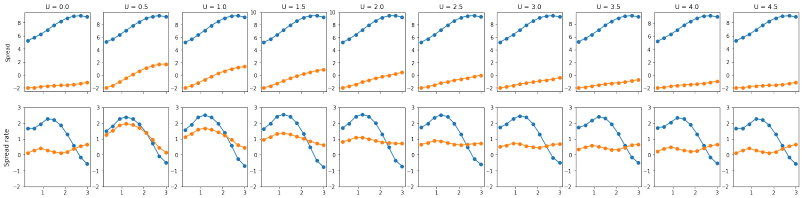


Paper results:

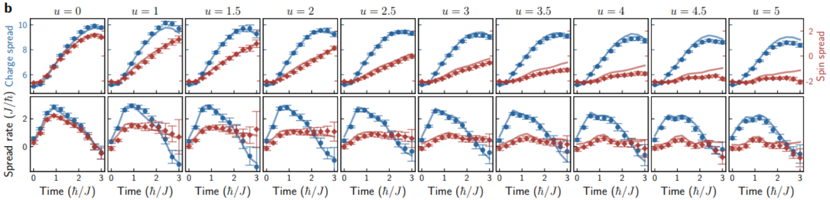


Results

Charge and spin spread time derivatives, varying U:

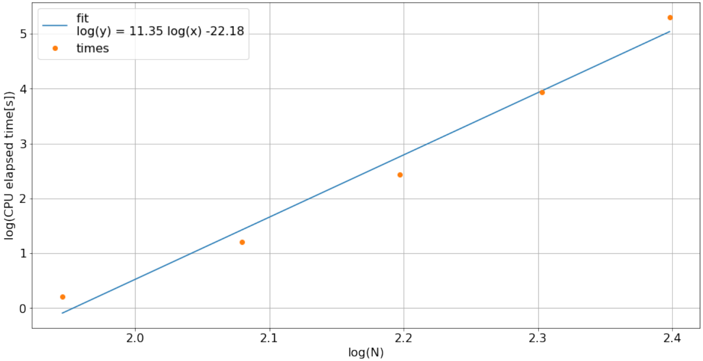


Paper results:



Efficiency analysis

Algorithm efficiency is analyzed by performing one trotter step for different values of N.



Conclusions

- Experimental results correctly reproduced.
- We employed only our classical PC: the experiment doesn't prove quantum supremacy.

Thank you
for your attention

Givens Rotations

Initialization into GS of non-interacting fermionic Hamiltonian

- Prepare slater determinant state $\psi_{SD} = \prod_{j=1}^{N_{el}} a_j^\dagger |0\rangle$
- Apply various $R_{i,j}(\theta) = e^{\theta c_i^\dagger c_j - h.c.}$

$\mathcal{O}(N)$ to prepare the ground state of a non-interacting Hamiltonian[1].

Givens Rotations - FQE

In *evolve_fqe_givens_sector*, the unitary matrix gets decomposed as

$$U = G_k \dots G_1$$

where each rotation acts on the sub-space spanned by two coordinate axes. It has the form (relative to the proper axes)

$$\begin{pmatrix} \cos\theta & -e^{i\phi}\sin\theta \\ \sin\theta & e^{i\phi}\cos\theta \end{pmatrix}$$

Number Conserving 2 qubit Gate

$$U(\theta,\zeta,\chi,\gamma,\phi) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & e^{-i(\gamma+\zeta)}\cos\theta & -ie^{-i(\gamma-\chi)}\sin\theta & 0 \\ 0 & -ie^{-i(\gamma-\chi)}\sin\theta & e^{-i(\gamma-\zeta)}\cos\theta & 0 \\ 0 & 0 & 0 & e^{-i(2\gamma+\phi)} \end{pmatrix}$$

$$U = R_Z(-\gamma,-\gamma)R_Z(\beta,-\beta)U(\theta,0,0,0,\phi)R_Z(\alpha,-\alpha)$$

with $\alpha = \frac{(\chi+\zeta)}{2}, \beta = \frac{\zeta-\chi}{2}$

How FQE implements Time Evolution

FQE evolves states through (spatial orbitals indexed by i, j and S_z values by σ, ρ

$$H = \epsilon \left(\hat{g} + \hat{g}^\dagger\right)$$

$$\hat{g} = g \prod_{p=1}^{\#ops} \hat{a}^\dagger_{i_p \sigma_p} \prod_{q=1}^{\#ops} \hat{a}_{j_q \rho_q}$$

using the identity[3]

$$\begin{aligned} e^{-i\epsilon\left(\hat{g}+\hat{g}^\dagger\right)} =& \mathbb{I} + \left[\cos(\epsilon|g|) - 1 - i\hat{g}^\dagger \frac{\sin \epsilon|g|}{|g|}\right] \hat{\mathcal{P}}_{\hat{g}\hat{g}^\dagger} \\ & + \left[\cos(\epsilon|g|) - 1 - i\hat{g} \frac{\sin \epsilon|g|}{|g|}\right] \hat{\mathcal{P}}_{\hat{g}^\dagger\hat{g}} \end{aligned}$$

where $\hat{\mathcal{P}}_{\hat{g}\hat{g}^\dagger}$ projects onto the basis of determinants not annihilated by $\hat{g}\hat{g}^\dagger$.

Jordan-Wigner transformation

Jordan-Wigner transform (JWT): fermionic transform mapping FermionOperators to QubitOperators in a way that preserves the canonical anticommutation relations.

For example the hopping term is mapped as follows:

$$a^\dagger_{j,\nu} a_{j+1,\nu} + h.c. \longrightarrow \frac{1}{2}(X_{j,\nu} X_{j+1,\nu} + Y_{j,\nu} Y_{j+1,\nu})$$

References I

I. D. Kivlichan, J. McClean, N. Wiebe, C. Gidney, A. Aspuru-Guzik, G. K.-L. Chan, and R. Babbush.
Quantum simulation of electronic structure with linear depth and connectivity.
Phys. Rev. Lett., 120:110501, Mar 2018.

M. Motta, E. Ye, J. R. McClean, Z. Li, A. J. Minnich, R. Babbush, and G. K.-L. Chan.
Low rank representations for quantum simulation of electronic structure, 2018.

N. C. Rubin, K. Gunst, A. White, L. Freitag, K. Throssell, G. K.-L. Chan, R. Babbush, and T. Shiozaki.
The fermionic quantum emulator.
Quantum, 5:568, oct 2021.

References II

D. Wecker, M. B. Hastings, N. Wiebe, B. K. Clark, C. Nayak, and M. Troyer.
Solving strongly correlated electron models on a quantum computer.
Physical Review A, 92(6), dec 2015.