QuSpin: a Python Package for Dynamics and Exact Diagonalisation of Quantum Many Body Systems. Part II: bosons, fermions and higher spins

Phillip Weinberg* and Marin Bukov

Department of Physics, Boston University,
590 Commonwealth Ave., Boston, MA 02215, USA
* weinbe58@bu.edu

May 16, 2017

Abstract

We present a major update to QuSpin, *SciPostPhys.2.1.003*, – an open-source Python package for exact diagonalization and quantum dynamics of boson, fermion and spin many-body systems, supporting the use of various symmetries in 1-dimension and (imaginary) time evolution. We explain how to use the new features of QuSpin using six detailed examples of various complexity: (i)... This easily accessible package can serve various purposes, including educational and cutting-edge experimental and theoretical research.

Contents

What can QuSpin be Useful for?

- re-label numbering of examples to match order in text.

Understanding the physics of many-body quantum condensed matter systems often involves a great deal of numerical simulations, be it to gain intuition about the complicated problem of interest, or because the latter does not admit an analytical solution which can be expressed in a closed form. This motivated the development of open-source packages [CITE], the purpose of which is to facilitate the study of condensed matter systems without the need to understand and implement complicated numerical methods which required years to develop. Here, we report a major upgrade to QuSpin [1] – a Python library for exact diagonalisation (ED) and simulation of the dynamics of quantum many-body systems.

Although ED methods are vastly outperformed by more sophisticated numerical techniques in the study of equilibrium systems [CITE], as of present date ED remains essential for most dynamical non-equilibrium problems. The reason for this often times relies on the fact that the underlying physics of these problems cannot be explained without taking into consideration the contribution from high-energy states excited during the evolution. Some prominent examples of such problems include the study of many-body localisation (MBL) [CITE], the Eigenstate Thermalisation hypothesis [CITE], quantum quench dynamics [CITE], periodically-driven systems [CITE], adiabatic and counter-diabatic state preparation, applications of Machine Learning to non-equilibrium physics [CITE], and many more **did I forget smth important?**.

It is, thus, arguably useful to have a toolbox available which allows one to quickly simulate and study these and related nonequilibrium problems. As such, QuSpin offers easy access to performing numerical simulations, which can facilitate the development and inspiration of new ideas and the discovery of novel phenomena, eliminating the cost of spending time to develop a reliable code. Besides theorists, the new version of QuSpin will hopefully even prove valuable to experimentalists working on problems containing dynamical setups, as it can help students and researchers focus on making the experiment run, rather than worrying about writing the supporting simulation code. Last but not least, with the computational processing power growing higher than ever before, the role played by simulations for theoretical research becomes increasingly more important too. It can, therefore, be expected that in the near future quantum simulations become an integral part of the standard physics university curriculum, and having easily accessible toolboxes, such as QuSpin, is one of the required prerequisites.

How do I use the New Features of QuSpin?

New in QuSpin 2.0, we have added the following features and toolboxes:

- ...

Installing QuSpin is quick and efficient; just follow the steps outlined in App. A.

Before we carry on, we refer the interested reader to examples (i)-(iv) from the original QuSpin paper [1]. The examples below focus predominantly on the newly introduced features, and are thus to be considered complementary. We emphasize that, since they serve the purpose of explaining how to use QuSpin, for the sake of brevity we shall not discuss the interesting physics related to the interpretation of the results.

The Spectrum of the Transverse Field Ising Model and the Jordan-Wigner Transformation

This example shows how to

- construct fermionic hopping, p-wave pairing and on-site potential terms, and spin$-1/2$ interactions and transverse fields,
- implement periodic and anti-periodic boundary conditions with translation and parity (reflection) symmetries,
- use particle conservation modulo 2, spin inversion, reflection, and translation symmetries,
- handle the default built-in particle conservation and symmetry checks,
- obtain the spectrum of a QuSpin Hamiltonian.

Physics Setup—The transverse field Ising (TFI) chain is paradigmatic in our understanding of quantum phase transitions, since it represents an exactly solvable model[CITE Sachdev]. The Hamiltonian is given my

$$H = \sum_{j=0}^{L-1} -J\sigma^z_{j+1}\sigma^z_j - h\sigma^x_j, \tag{1}$$

where the nearest-neighbour (nn) spin interaction is J, h denotes the transverse field, and σ^α_j are the Pauli spin-1/2 matrices. We use periodic boundary conditions and label the L lattice sites $0, \dots, L-1$ to conform with Python's convention. This model has gapped, fermionic elementary excitations, and exhibits a phase transition from an antiferromagnet to a paramagnet at $(h/J)_c = 1$ CHECK!. This Hamiltonian possesses the symmetries: magnetisation conservation, parity (reflection about the centre of the chain), spin inversion, and (many-body) momentum conservation.

In one dimension, the TFI Hamiltonian can be mapped to spinless p-wave superconducting fermions via the Jordan-Wigner (JW) transformation[CITE Sachdev, other paper]:

$$c_i = \prod_{j<i} \sigma^z_j \sigma^-_i, \qquad c^\dagger_i = \prod_{j<i} \sigma^z_j \sigma^+_i, \tag{2}$$

where the fermionic operators satisfy $\{c_i, c_j^\dagger\} = \delta_{ij}$. The Hamiltonian is readily shown to take the form

$$H = \sum_{j=0}^{L-1} J\left(-c_j^\dagger c_{j+1} + c_j c_{j+1}^\dagger\right) + J\left(-c_j^\dagger c_{j+1}^\dagger + c_j c_{j+1}\right) + 2h\left(n_j - \frac{1}{2}\right). \tag{3}$$

In the fermionic representation, the spin zz-interaction maps to nn hopping and a p-wave pairing term with coupling constant J, while the transverse field translates to an on-site potential shift of magnitude h. In view of the QuSpin implementation of the model, we have ordered the terms such that the site index is growing to the right which comes at the cost of a few negative signs due to the fermion statistics. The fermion Hamiltonian posses the symmetries: particle conservation modulo 2, parity and (many-body) "momentum" conservation.

Here, we are interested in studying the spectrum of the TFI model in both the spin and fermion representation. However, if one naively carries out the JW transformation, and computes the spectra of Eqs. (1) and (3), one might be surprised that they do not match exactly. The reason lies in the form boundary condition required to make the JW mapping exact – a subtle issue often left aside in favour of discussing the interesting physics of the TFI model.

Recall that the starting point is the periodic boundary condition imposed on the spin Hamiltonian 1. Due to the symmetries of the spin Hamiltonian (1), we can define the JW transformation on every symmetry sector separately. To make the JW mapping exact, we supplement Eq. (2) with the following boundary conditions: (i) the negative spin-inversion symmetry sector maps to the fermion Hamiltonian (3) with *periodic* boundary conditions (PBC) and odd total number of fermions; (ii) the positive spin-inversion symmetry sector maps to the fermion Hamiltonian (3) with *anti-periodic* boundary conditions (APBC) and even total number of fermions. Anti-periodic boundary conditions differ from PBC by a negative sign attached to all coupling constants that cross a single, fixed lattice bond (the bond itself is arbitrary as all bonds are equal for PBC). APBC and PBC are special cases of the more general, twisted boundary conditions, where instead of a negative sign, one attaches a phase factor.

In the following, we show how to compute the spectra of the Hamiltonians in Eqs. (1) and (3) with the correct boundary conditions using QuSpin. Figure 1 shows that they match exactly in both the PBC and APBC cases discussed above.

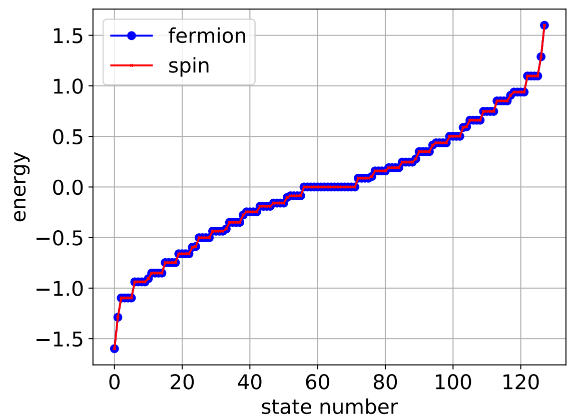
Code Analysis—We begin by loading the QuSpin operator and basis constructors, as well as some standard Python libraries.

```python
from quspin.operators import hamiltonian # Hamiltonians and operators
from quspin.basis import spin_basis_1d, fermion_basis_1d # Hilbert space spin basis
import numpy as np # generic math functions
import matplotlib.pyplot as plt # figure/plot library
```

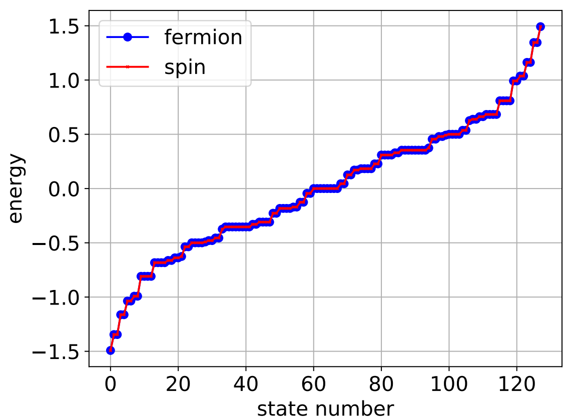
First, we define the models parameters.

```python
##### define model parameters #####
L=8 # system size
J=1.0 # spin zz interaction
h=np.sqrt(2) # z magnetic field strength
```

We have to consider two cases when computing the spectrum, as discussed in the theory section above. In one case, the fermionic system has PBC, while the spins are constrained



(b) positive spin inversion/APBC sector



(a) negative spin inversion/PBC sector

Figure 1: Comparison of the spectra of the spin and fermion representation of the transverse field Ising Hamiltonian in the spin (1) and fermion (3) representations. The degeneracy is the spectrum is due to the remaining parity and momentum conservations which are not taken into account (see text). The parameters are $J = 1.0$, $h = \sqrt{2}$, and $L = 8$.

to the negative spin inversion symmetry sector, while in the second – the fermion model has APBC and the spin model is considered in the positive spin inversion sector. To this end, we introduce the variables `zblock` $\in \{\pm 1\}$ and `PBC` $\in \{\pm 1\}$, where `PBC` $= -1$ denotes APBC. Note than the only meaningful combinations are (`zblock`, `PBC`) $= (-1, 1), (1, -1)$.

```
# loop over spin inversion symmetry block variable and boundary conditions
for zblock,PBC in zip([-1,1],[1,-1]):
```

Within this loop, the code is divided to two independent parts: first, we compute the spectrum of the TFI system, and then – that of the equivalent fermionic model. Let us discuss the spins.

```
    ##### define spin model
```

In QuSpin, operators are stored as sparse lists. These lists contain two parts: (i) the lattice sites on which the operator acts together with the coupling strength, which we call a *site-coupling* list, and (ii) the types of the operators involved, i.e. *operator-string*. For example, the operator $\mathcal{O} = g\sum_{j=0}^{L-1}\sigma^\mu_j$ can be uniquely represented by the site-coupling list `[[g,0],[g,1],...,[g,L-1]]`, and the information that it is the Pauli matrix μ. The components lists are nothing but the tuples of the field strength and the site index `[g,j]`. It is straightforward to generalise this to non-uniform fields `g`$\to$`g[j]`. Similarly, any two-body operator $\mathcal{O} = J_{zz}\sum_{j=0}^{L-1}\sigma^\mu_j\sigma^\nu_{j+1}$ can be fully represented by the two sites it acts on, and its coupling strength: `[J,j,j+1]`. We then stack up these elementary lists together into a the site-coupling list: `[[J,0,1],[J,1,2],...,[J,L-2,L-1],[J,L-1,0]]`.

```
# site-coupling lists (PBC for both spin inversion sectors)
h_field=[[-h,i] for i in range(L)]
J_zz=[[-J,i,(i+1)%L] for i in range(L)] # PBC
```

Notice the way we defined the periodic boundary condition for the spin-spin interaction using the modulo operator %, which effectively puts a coupling between sites $L-1$ and 0. We mention in passing that the above procedure generalises so one can define any multi-body local and nonlocal operator using QuSpin.

In order to specify the types of the on-site single-particle operators, we use operator strings. For instance, the transverse field operator $\mathcal{O} = g\sum_{j=0}^{L-1}\sigma^x_j$ becomes `['x',h_field]`, while the two-body interaction is `['zz',J_zz]`. It is important to notice that the order of the letters in the operator string corresponds to the order the operators are listed in the site-coupling lists. Putting everything into one final list yields:

```
# define spin static and dynamic lists
static_spin =[["zz",J_zz],["x",h_field]] # static part of H
```

In QuSpin, the user can define both static and dynamic operators. Since this example does not require any time evolution, we postpone the explanation of how to use dynamic lists to Sec. 2.5, and use an empty list instead.

```
dynamic_spin=[] # time-dependent part of H
```

The last step before we can construct the Hamiltonian is to build the basis for it. This is done using the basis constructors. For spin systems, we use `spin_basis_1d` which allows to use the operator strings `'z','+','-'`, and for spin-1/2 additionally `'x','y'`. The first and required argument is the number of sites L. Optional arguments are used to parse symmetry sectors. For instance, if we want to construct an operator in the spin-inversion block with quantum number $+1$, we can conveniently do this using the flag `zblock=1`.

```
# construct spin basis in pos/neg spin inversion sector depending on APBC/PBC
basis_spin = spin_basis_1d(L=L,zblock=zblock)
```

Having specified the static and dynamic lists, as well as the basis, building up the Hamiltonian is a one-liner, using the `hamiltonian` constructor. The first and second compulsory arguments are the static and dynamic list, respectively. Optional arguments include the `basis`, and the precision or data type `dtype`. If no basis is passed, the constructor uses `spin_basis_1d` by default. The default data type is `np.complex128`.

```
# build spin Hamiltonians
H_spin=hamiltonian(static_spin,dynamic_spin,basis=basis_spin,dtype=np.float64)
```

The Hamiltonian is stored as a sparse matrix for efficiency. It can be cast to a full array for a more convenient inspection using the attribute `H.toarray()`. To calculate its spectrum, we use the attribute $H.eigenvalsh()$, which returns all eigenvalues. Other attributes for diagonalisation were discussed in Example 0 [PUT link to github], c.f. Ref. [1].

```
# calculate spin energy levels
E_spin=H_spin.eigvalsh()
```

Let us now move to the second part of the loop which defines the fermionic p-wave superconductor. We start by defining the site-coupling list local potential

```
##### define fermion model
# define site-coupling lists for external field
h_pot=[[2.0*h,i] for i in range(L)]
```

Let us ficus on the case of periodic boundary conditions `PBC=1` first.

```
    if PBC==1: # periodic BC: odd particle number subspace only
```

In the fermion model, we have two types of two-body terms: hopping terms $c_i^\dagger c_{i+1} - c_i c_{i+1}^\dagger$, and pairing terms $c_i^\dagger c_{i+1}^\dagger - c_i c_{i+1}$. While QuSpin allows any ordering of the operators, for te sake of completeness we set a convention: the site indices grows to the right. Do to the opposite signs in the terms resulting from the fermion statistics, we have to code the site-coupling lists for all four terms separately. This is analogous to the spin-spin interaction above:

```
        # define site-coupling lists (including boudary couplings)
        J_pm=[[-J,i,(i+1)%L] for i in range(L)] # PBC
        J_mp=[[+J,i,(i+1)%L] for i in range(L)] # PBC
        J_pp=[[-J,i,(i+1)%L] for i in range(L)] # PBC
        J_mm=[[+J,i,(i+1)%L] for i in range(L)] # PBC
```

To construct a fermionic operator, we make use of the fermion basis constructor `fermion_basis_1d`. This Once again, we pass the number of sites `L`. As we explained in the analysis above, we need to consider all odd particle number sectors in the case of PBC. This is done by specifying the particle number sector `Nf`.

```
        # construct fermion basis in the odd particle number subsector
        basis_fermion = fermion_basis_1d(L=L,Nf=range(1,L+1,2))
```

In the case of APBC, we first construct all two-body site-coupling lists as if the boundaries were open, and supplement the APBC links in the end:

```
    elif PBC==-1: # anti-periodic BC: even particle number subspace only
        # define bulk site coupling lists
        J_pm=[[-J,i,i+1] for i in range(L-1)]
        J_mp=[[+J,i,i+1] for i in range(L-1)]
        J_pp=[[-J,i,i+1] for i in range(L-1)]
        J_mm=[[+J,i,i+1] for i in range(L-1)]
        # add boundary coupling between sites (L-1,0)
        J_pm.append([+J,L-1,0]) # APBC
        J_mp.append([-J,L-1,0]) # APBC
        J_pp.append([+J,L-1,0]) # APBC
        J_mm.append([-J,L-1,0]) # APBC
```

The definition of the basis is the same, except that this time, we need all even particle number sectors:

```
        # construct fermion basis in the even particle number subsector
        basis_fermion = fermion_basis_1d(L=L,Nf=range(0,L+1,2))
```

As before, we need to specify the type of operators that go in the Hamiltonian using operator string lists. The `fermion_basis_1d` class accepts the following strings `'+'`,`'-'`,`'n'`, and additionally the particle-hole symmetrised density operator `'z'` $= n - 1/2$. The static and dynamic lists read as

```
    # define fermionic static and dynamic lists
    static_fermion =[["+-",J_pm],["-+",J_mp],["++",J_pp],["--",J_mm],['z',h_pot]]
    dynamic_fermion=[]
```

Constructing and diagonalising the fermion Hamiltonian is the same as for the spin-1/2 system. Note that one can disable the automatic built-in checks for particle conservation check_pcon=False and all other symmetries check_symm=False if one wishes to suppress the checks.

```
# build fermionic Hamiltonian
H_fermion=hamiltonian(static_fermion,dynamic_fermion,basis=basis_fermion,
                        dtype=np.float64,check_pcon=False,check_symm=False)
# calculate fermionic energy levels
E_fermion=H_fermion.eigvalsh()
```

The complete code including the lines that produce Fig. 1 is available in Example Code 1.

Free Particle Systems: the SSH Model

This example shows how to

- construct free-particle Hamiltonians in real space,
- implement translation invariance with a two-site unit cell and construct the single-particle Hamiltonian in momentum space in block-diagonal form,
- compute non-equal time correlation functions,
- ...

Physics Setup—The Su-Schrieffer-Heeger (SSH) model of a free-particle on a dimerised chain is widely used to introduce the concept of edge states, topology, Berry phase, etc., in one spatial dimension. The Hamiltonian is given by

$$H=\sum_{j=0}^{L-1}-(J+(-1)^j\delta J)\left(c_jc_{j+1}^\dagger-c_j^\dagger c_{j+1}\right)+\Delta(-1)^jn_j, \tag{4}$$

where $\{c_i,c_j^\dagger\}=\delta_{ij}$ obey fermionic commutation relations. The uniform part of the hopping matrix element is J, δJ defines the bond dimerisation, and Δ is the staggered potential. We assume periodic boundary conditions.

Below, we show how one can use QuSpin to study the physics of free fermions in the SSH chain. One way of doing this would be to work in the many-body (Fock space) basis, see Sec. ???. However, whenever the particles are non-interacting, the exponential scaling of the Hilbert space dimension with the number of lattice sites imposes an artificial limitation on the system sizes one can do. Luckily, with no interactions present, the many-body wave functions factorise in a product of single-particle states. Hence, it is possible to study the behaviour of many free bosons and fermions by simulating the physics of a single particle.

Making use of translation invariance, a straightforward Fourier transformation to momentum space, $a_k=\sqrt{2/L}\sum_{j\text{ even}}^{L-1}\mathrm{e}^{-ikj}c_j$ and $b_k=\sqrt{2/L}\sum_{j\text{ odd}}^{L-1}\mathrm{e}^{-ikj}c_j$, casts the SSH Hamiltonian in the following form

$$H=\sum_{k\in\mathrm{BZ}'}(a_k^\dagger,b_k^\dagger)^t\begin{pmatrix}\Delta & -(J+\delta J)\mathrm{e}^{-ik}-(J-\delta J)\mathrm{e}^{+ik}\\ -(J+\delta J)\mathrm{e}^{+ik}-(J-\delta J)\mathrm{e}^{-ik} & -\Delta\end{pmatrix}\begin{pmatrix}a_k\\ b_k\end{pmatrix}, \tag{5}$$

where the reduced Brillouin zone is defined as $\mathrm{BZ}'=[-\pi/2,\pi/2)$. We thus see that the Hamiltonian reduces further to a set of independent 2×2 matrices. The spectrum of the SSH model is gapped, see Fig. ???.

Since we are dealing with free fermions, the ground state is the Fermi sea, $|\mathrm{FS}\rangle$, defined by filling up the lowest band completely. We are interested in measuring the real-space non-equal time correlation function

$$C_{ij}(t) = \langle \mathrm{FS}|n_i(t)n_j(0)|\mathrm{FS}\rangle = \langle \mathrm{FS}(t)|n_i(0)\underbrace{U(t,0)n_j(0)|\mathrm{FS}\rangle}_{|\mathrm{nFS}(t)\rangle}. \tag{6}$$

For simplicity, let us focus on a single unit cell. Figure ??? shows the time evolution of $C_{AA}(t)$ and $C_{AB}(t)$.
Code Analysis—...

```
from quspin.operators import hamiltonian # Hamiltonians and operators
from quspin.basis import spin_basis_1d, fermion_basis_1d # Hilbert space spin basis
import numpy as np # generic math functions
import matplotlib.pyplot as plt # figure/plot library
##### define model parameters #####
L=8 # system size
J=1.0 # spin zz interaction
h=np.sqrt(2) # z magnetic field strength
# loop over spin inversion symmetry block variable and boundary conditions
for zblock,PBC in zip([-1,1],[1,-1]):
    ##### define spin model
    # site-coupling lists (PBC for both spin inversion sectors)
    h_field=[[-h,i] for i in range(L)]
    J_zz=[[-J,i,(i+1)%L] for i in range(L)] # PBC
    # define spin static and dynamic lists
    static_spin =[["zz",J_zz],["x",h_field]] # static part of H
    dynamic_spin=[] # time-dependent part of H
    # construct spin basis in pos/neg spin inversion sector depending on APBC/PBC
    basis_spin = spin_basis_1d(L=L,zblock=zblock)
    # build spin Hamiltonians
    H_spin=hamiltonian(static_spin,dynamic_spin,basis=basis_spin,dtype=np.float64)
    # calculate spin energy levels
    E_spin=H_spin.eigvalsh()
    ##### define fermion model
    # define site-coupling lists for external field
    h_pot=[[2.0*h,i] for i in range(L)]
    if PBC==1: # periodic BC: odd particle number subspace only
        # define site-coupling lists (including boudary couplings)
        J_pm=[[-J,i,(i+1)%L] for i in range(L)] # PBC
        J_mp=[[+J,i,(i+1)%L] for i in range(L)] # PBC
        J_pp=[[-J,i,(i+1)%L] for i in range(L)] # PBC
        J_mm=[[+J,i,(i+1)%L] for i in range(L)] # PBC
        # construct fermion basis in the odd particle number subsector
        basis_fermion = fermion_basis_1d(L=L,Nf=range(1,L+1,2))
    elif PBC==-1: # anti-periodic BC: even particle number subspace only
        # define bulk site coupling lists
        J_pm=[[-J,i,i+1] for i in range(L-1)]
        J_mp=[[+J,i,i+1] for i in range(L-1)]
        J_pp=[[-J,i,i+1] for i in range(L-1)]
        J_mm=[[+J,i,i+1] for i in range(L-1)]
        # add boundary coupling between sites (L-1,0)
        J_pm.append([+J,L-1,0]) # APBC
```

```
        J_mp.append([-J,L-1,0]) # APBC
        J_pp.append([+J,L-1,0]) # APBC
        J_mm.append([-J,L-1,0]) # APBC
        # construct fermion basis in the even particle number subsector
        basis_fermion = fermion_basis_1d(L=L,Nf=range(0,L+1,2))
    # define fermionic static and dynamic lists
    static_fermion =[["+-",J_pm],["-+",J_mp],["++",J_pp],["--",J_mm],['z',h_pot]]
    dynamic_fermion=[]
    # build fermionic Hamiltonian
    H_fermion=hamiltonian(static_fermion,dynamic_fermion,basis=basis_fermion,
                            dtype=np.float64,check_pcon=False,check_symm=False)
    # calculate fermionic energy levels
    E_fermion=H_fermion.eigvalsh()
```

The complete code including the lines that produce Fig. **??** is available in Example Code **??**.

Fermionic Many-body Localization

This example shows how to:

- construct Hamiltonians for spinful fermions using the tensor_basis class.
- how to use ops_dict class to construct Hamiltonians with varying parameters.
- use new basis functionality to construct simple product states
- use obs_vs_time functionality to measure observables as a function of time

A class of exciting new problems in the field of non-equalibrium physics are that of the Many-body localzation (MBL) transition. The MBL transition is a dynamical phase transition in the eigenstates of a many-body Hamiltonian. Driven primarily by quenched disorder, the transition occurs is distinguished by ergodic eigenstates in the weak disorder limit and non-ergodic eigenstates in the strong disorder limit. The MBL phase is reminiscent of integrable systems as one can construct quasi-local integrals of motion in the MBL phase, but these integrals of motion are much more robust in the sense that they are not sensitive to small perturbations are is the case in many classes of integrable systems[CITE MBL misc].

Motivated by some recent experiments in cold atomic gasses [CITE Bloch MBL exp] we explore MBL in the context of fermions using QuSpin. The model we will consider is the Fermi-Hubbard model with quenched random disorder which has the following Hamiltonian:

$$H = -J\sum_{\sigma,i=0}^{L-1} c^\dagger_{\sigma i}c_{\sigma i+1} - c_{\sigma i}c^\dagger_{\sigma i+1} + U\sum_{i=0}^{L} n_{\uparrow i}n_{\downarrow i} + \sum_{\sigma,i=0}^{L} V_i n_{\sigma i} \tag{7}$$

where $c_{\sigma i}$ and $c^\dagger_{\sigma i}$ is a ferminoic creation and annihilation operators on site i for spin σ respectively. We will work in the sector of $1/4$ filling for both up and down spins. Preparing an initial configuration of fermions of alternating spin on every other site, we will then measure the sublattice imbalance:

$$I = (N_A - N_B)/N_{\text{tot}} \tag{8}$$

where A and B refer to the different sublattices of the chain and N is the particle number operator. evolving with Hamiltonian (7) we will calculate its value which will tell us something

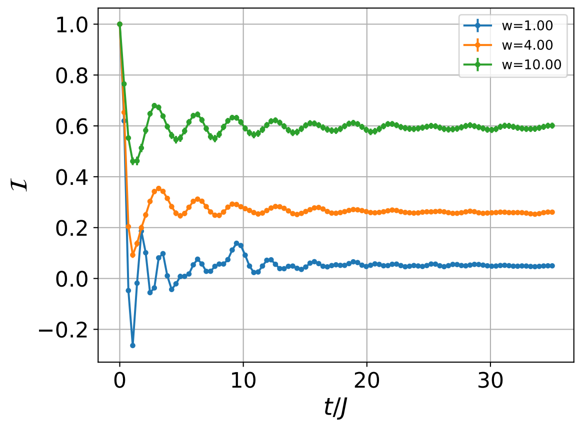


Figure 2: Sublattice imbalance I as a function of time averaged over 100 disorder realizations for different disorder strengths. This data was taken on a chain of length $L = 8$.

about the ergodicity(or lack there of) of the Hamiltonian. If the Hamiltonian is ergodic then this quantity will decay to 0 in the limit $t \to \infty$ as one would expect in equilibrium while if the Hamiltonian is MBL then some memory of its initial condition will be retained, and therefore this quantity will be non-zero even at infinite times.

Because the Hilbert space dimension grows so quickly for this Hamiltonian we will only consider the dynamics after a finite amount of time, and even with this it is a fairly long calculation to do $L > 10$.

Code Analysis—...

```
from __future__ import print_function, division

from quspin.operators import hamiltonian,exp_op,ops_dict # Hamiltonians and other operators
from quspin.basis import tensor_basis,fermion_basis_1d # fermion and tensor Hilbert spaces
from quspin.tools.measurements import obs_vs_time # function to calculating dynamics
import numpy as np # general math functions
from numpy.random import uniform,choice # tools for doing random sampling
from joblib import Parallel,delayed # tools for doing
from time import time # tool for calculating computation time
import matplotlib.pyplot as plt # plotting

# setting parameters for simulation
n_jobs = 2 # number of cores to use in calculating realizations
n_real = 1000 # number of realizations
n_boot = 100*n_real # number of bootstrap realizations to calculate error
# physical parameters
L = 8 # system size
N = L//2 # number of particles
w1 = 1.0 # disorder strength
w2 = 4.0
w3 = 10.0
J = 1.0 # hopping strength
U = 5.0 # interaction strength
```

```python
k = 0.1 # trap stiffness
# range to evolve system
start,stop,num=0.0,35.0,101
# setting up basis
N_up = N//2 + N % 2 # number of fermions with spin up
N_down = N//2 # number of fermions with spin down
# building the two basis to tensor together
basis_up = fermion_basis_1d(L,Nf=N_up) # up basis
basis_down = fermion_basis_1d(L,Nf=N_down) # down basis
basis = tensor_basis(basis_up,basis_down) # spinful fermsions
# creating coupling lists
i_mid = (L//2+1 if L%2 else L//2+0.5) # mid point on lattice
hop_right = [[-J,i,i+1] for i in range(L-1)] # hopping to the right OBC
hop_left = [[J,i,i+1] for i in range(L-1)] # hopping to the left OBC
int_list = [[U,i,i] for i in range(L)] # onsite interaction
trap_list = [[0.5*k*(i-i_mid)**2,i] for i in range(L)] # harmonic trap
# coupling list to create the sublattice imbalance observable
sublat_list = [[(-1)**i,i] for i in range(0,L)]
# create static lists
operator_list_0 = [
            ["+-|", hop_left], # up hop left
            ["-+|", hop_right], # up hop right
            ["n|", trap_list], # up trap potential
            ["|+-", hop_left], # down hop left
            ["|-+", hop_right], # down hop right
            ["|n", trap_list], # down trap potential
            ["n|n", int_list], # onsite interaction
         ]
# create operator dictionary for ops_dict class
# creates a dictioanry with keys h0,h1,h2,...,hL for local potential
operator_dict = {"h"+str(i):[["n|",[[1.0,i]]],["|n",[[1.0,i]]]] for i in range(L)}
operator_dict["H0"]=operator_list_0
```

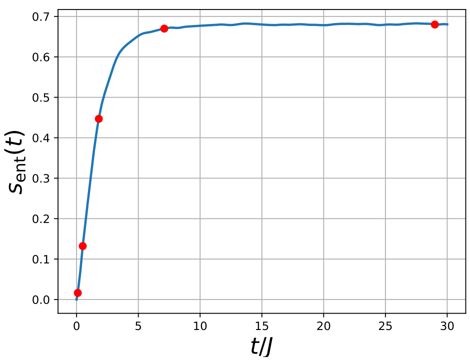
Bose-Hubbard Model on Translationally Invariant Ladder

This example shows how to:

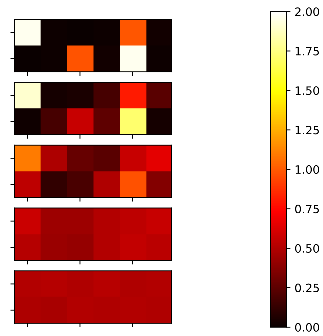
- construct Hamiltonians for bosonic systems.
- construct ladder Hamiltonians.
- using block_ops class to evolve over several symmetry sectors at once.
- measure entanglement entropy of ladder.

Physics Setup—In this example we will use QuSpin to solve the dynamics of the Bose-Hubbard model (BHM) on a ladder geometry. The BHM is a minimal model of interacting bosons which is experimentally realizable in cold atom experiments [CITE]. The Hamiltonian is given by:

$$H_{\mathrm{BHM}} = -J\sum_{\langle ij\rangle} a_i^\dagger a_j + \mathrm{h.c.} + U\sum_i n_i(n_i-1) \tag{9}$$



(a) Entanglement entropy density between the two legs of the ladder as a function of time.



(b) local density of bosons as a function of time. Later times appearing farther down the page.

Figure 3: Showing results from the quench in the BHM. plot (a) shows the half-ladder entanglement entropy density and plot (b) shows the local density on each site as a function of time. The red dots on the entanglement plot shows the time points where the density plots are taken. For this data was taken with $J = 1$ and $U = 10$.

where a_i and $a_i^\dagger$ are bosonic creation and annihilation operators on site i respectively and the sum $\langle ij \rangle$ is a sum over nearest neighbors on Ladder. We will consider a half filled ladder of length L with $N = 2L$ sites. We will perform a quench where the system starts out in a random product state and let it evolve with Hamiltonian (9). We will restrict the local Hilbert space to allow at most 2-particles on a site which is valid in the large U limit. This model is not integrable and so we expect that the system will eventually thermalize so that the occupation is roughly uniform over the entire system. On top of measuring the local density we will also measure the entanglement entropy between the legs of the ladder.

If we consider a translational invariant ladder that implies the Hamiltonian factorizes into different many-body momentum blocks similar to what was discussed in Sec. 2.2 but slightly different as we consider translations of the many-body fock states as opposed to the single particle states[CITE Anders review]. In this section instead of projecting the operators to momentum space as was done in Sec. 2.2, we will project the wavefunction to the different symmetry sectors and evolve each of the projections separately under the Hamiltonian for that symmetry sector. Then each of the Block wavefunctions are projected back to the local Fock space basis and summed to recover the evolve state which one can then look at local quantities like the density and entanglement.

Code Analysis— Just as in the previous examples we start out our python script by loading some essential modules needed for the calculation

```
from quspin.operators import hamiltonian # Hamiltonians and operators
from quspin.basis import boson_basis_1d # bosonic Hilbert space
```

```python
from quspin.tools.block_tools import block_ops # tool for doing dynamics over
    symmetry blocks
import numpy as np # general math functions
```

Next we set up the model parameters defining the length of the ladder `L` chain number of sites `N=2*L` as well as filling factor for the bosons `nb`, and the maximum number of states per site `sps`. The hopping matrix elements $J_\perp$, $J_{\parallel,1}$, and $J_{\parallel,2}$ (see Fig. ??create figure showing ladder and couplings etc...), correspond to the python script are variables `J_perp`, `J_par_1`, and `J_par_2` respectively.

```python
# setting up parameters of simulation
L = 6 # length of chain
N = 2*L # number of sites
nb = 0.5 # density of bosons
sps = 3 # number of states per site
J_par_1 = 1.0 # top side of ladder hopping
J_par_2 = 1.0 # bottom side of ladder hopping
J_perp =  0.5 # rung hopping
U = 10.0 # Hubbard interaction
```

Next we set up the times at which we would like to solve the Schrödinger equation. Due to the same restrictions on the exponential solver discussed Sec . 2.3 we will only consider time points which are linearly spaced defined by the variables `start`,`stop`, and `num`

```python
# setting up parameters for evolution
start,stop,num = 0,30,301 # 0.1 equally spaced points
times = np.linspace(start,stop,num)
```

For bosonic systems we have `'+'`,`'-'`, and `'n'` as possible operators to use. In order to set up the Hubbard local interaction we must define two coupling lists for the $U\sum_i n_i^2$ and the $-U\sum_i n_i$:

```python
# U n_i(n_i-1) interaction
int_list_2 = [[U,i,i] for i in range(N)] # U n_i^2
int_list_1 = [[-U,i] for i in range(N)] # -U n_i
```

We also define the hopping lists

```python
# setting up hopping lists
hop_list = [[-J_par_1,i,(i+2)%N] for i in range(0,N,2)] # PBC bottom
hop_list.extend([[-J_par_2,i,(i+2)%N] for i in range(1,N,2)]) # PBC top
hop_list.extend([[-J_perp,i,i+1] for i in range(0,N,2)]) # perp hopping
hop_list_hc = [[J.conjugate(),i,j] for J,i,j in hop_list]
```

where we use the `[...].extend([...])` method to concatenate two lists together[1]. The way the ladder is defined, the even sites correspond to the bottom side while the odds sites are the top part of the ladder, therefore the hopping on the bottom/top runge is defined by `[J_par_...,i,(i+2)%N]`, while hopping from top to bottom is defined by `[J_perp,i,(i+1)%N]`. Finally we can define the static and dynamic lists

```python
# setting up static list
static = [
            ["+-",hop_list], # hopping
```

[1]Note that the `extend` function is done inplace so if you try to do `new_list=list.extend(other_list)`, `new_list` will be `None` and `list` will have all of the elements of `other_list` appended to it.

```
            ["-+",hop_list_hc], # hopping h.c.
            ["nn",int_list_2], # U n_i^2
            ["n",int_list_1] # -U n_i
        ]
dynamic = [] # no dynamic operators
```

which we will not use to construct a `hamiltonian` object but instead we will use the `block_ops` class. The purpose of `block_ops` is to provide a simple interface for solving the Schrödinger equation when an initial state may not obey the symmetries of the Hamiltonian which is generating the dynamics. We have seen an example of this in Sec. 2.2 when trying to measure non-equal time correlation functions of local operators, while in this section we explicitly start out with a state which does not obey translational invariance. Before we construct the `block_ops` let us first define a few things which will be used to construct the `block_ops` object

```
# creating list which contains the blocks
# which the initial state will get projected on to.
blocks=[dict(kblock=kblock) for kblock in range(L)]
# tuple containing manditory arguments for boson_basis_1d
baisis_args = (N,)
# optional arguements for boson_basis_1d which are the same for all blocks
basis_kwargs = dict(nb=nb,sps=sps,a=2)
# optional arguments for boson_basis_1d.get_proj
# the function which calculates the projector from the
# symmetry reduced basis to the non-symmetry reduced basis
# this option is to set it so that th
```

Firstly, `blocks` is a list of dictionaries which define the different symmetry sectors to evolve the initial state over[2].

The Gross-Pitaevskii Equation and Nonlinear Time Evolution

This example shows how to

- simulate time-dependent nonlinear equations of motion
- use imaginary time dynamics to find a lowest energy configuration
- ...

Physics Setup—The Gross-Pitaevskii wave equation (GPE) has been shown to govern the physics of weakly-interacting bosonic systems. It constitutes the starting point for studying Bose-Einstein condensates, but can also appear in non-linear optics, and represents the natural description of Hamiltonian mechanics in the wave picture. One of its characteristic features is that it exhibits chaotic classical dynamics, a physical manifestation of the presence of a cubic non-linear term.

Here, we study the time-dependent GPE on a one-dimensional lattice:

$$i\partial_t\psi_j(t) = -J\left(\psi_{j-1}(t)+\psi_{j+1}(t)\right) + \frac{1}{2}\omega_{\mathrm{trap}}(t)(j-j_0)^2\psi_j(t) + U|\psi_j(t)|^2\psi_j(t), \tag{10}$$

where J is the hopping matrix element, $\omega_{\mathrm{trap}}(t) = (\omega_f - \omega_i)t/t_{\mathrm{ramp}} + \omega_i$ – the slowly-varying time-dependent harmonic trap frequency over a time scale t_{ramp}, and U – the interaction

[2] `block_ops` will not evolve a particular symmetry sector if the projection is 0.

strength. The lattice sites are labelled by $j = 0, \ldots, L-1$, and j_0 is the centre of the 1d chain. We set the lattice constant to unity, and use open boundary conditions.

Whenever $U = 0$, the system is non-interacting and the GPE reduces to the Heisenberg EOM for the bosonic field operator $\hat{\psi}_j(t)$. Thus, for the purposes of using QuSpin to simulate the GPE, it is instructive to cast Eq. (10) in the following generic form

$$i\partial_t \vec{\psi}(t) = H_{\mathrm{sp}}(t)\vec{\psi}(t) + U\vec{\psi}^*(t) \circ \vec{\psi}(t) \circ \vec{\psi}(t), \tag{11}$$

where $[\vec{\psi}(t)]_j = \psi_j(t)$, and $\circ$ represents the element-wise multiplication

$$\vec{\psi}(t) \circ \vec{\phi}(t) = \Big(\psi_0(t)\phi_0(t), \psi_1(t)\phi_1(t), \ldots, \psi_{L-1}(t)\phi_{L-1}(t)\Big)^t.$$

The time-dependent single-particle Hamiltonian in real space is represented as an $L \times L$ matrix, $H_{\mathrm{sp}}(t)$, which comprises the hopping term, and the harmonic trap.

We want to initiate the time-evolution of the system at $t = 0$ in its lowest energy state. To this end, we can define a 'ground state' for the GPE equation, in terms of the configuration which minimises the energy of the system:

$$\begin{aligned} \vec{\psi}_{\mathrm{GS}} &= \inf_{\vec{\psi}} \left(\vec{\psi}^t H_{\mathrm{sp}}(0)\vec{\psi} + \frac{U}{2}\sum_{j=0}^{L-1}|\psi_j|^4\right), \\ &= \inf_{\psi_j} \left(\sum_{j=0}^{L-1} -J(\psi^*_{j+1}\psi_j + \mathrm{c.c.}) + \frac{1}{2}\omega_{\mathrm{trap}}(0)|\psi_j|^2 + \frac{U}{2}|\psi_j|^4\right). \end{aligned} \tag{12}$$

One way to find the configuration $\vec{\psi}_{\mathrm{GS}}$, is to solve the GPE in imaginary time ($it \to \tau$), which induces exponential decay in all modes of the system, except for the lowest-energy state. In doing so, we keep the norm of the solution fixed:

$$\begin{aligned} \partial_\tau \vec{\varphi}(\tau) &= -\Big[H_{\mathrm{sp}}(0)\vec{\varphi}(\tau) + U\vec{\varphi}^*(\tau) \circ \vec{\varphi}(\tau) \circ \vec{\varphi}(\tau)\Big], \qquad ||\vec{\varphi}(\tau)|| = \mathrm{const.}, \\ \vec{\psi}_{\mathrm{GS}} &= \lim_{\tau\to\infty} \vec{\varphi}(\tau) \end{aligned} \tag{13}$$

Once we have the initial state $\vec{\psi}_{\mathrm{GS}}$, we evolve it according to the time-dependent GPE, Eq. (10), and track down the time evolution of the condensate density $\rho_j(t) = |\psi_j(t)|^2$. Fig. ??? shows the result.

Code Analysis—...

```python
from quspin.operators import hamiltonian # Hamiltonians and operators
from quspin.basis import spin_basis_1d, fermion_basis_1d # Hilbert space spin basis
import numpy as np # generic math functions
import matplotlib.pyplot as plt # figure/plot library
##### define model parameters #####
L=8 # system size
J=1.0 # spin zz interaction
h=np.sqrt(2) # z magnetic field strength
# loop over spin inversion symmetry block variable and boundary conditions
for zblock,PBC in zip([-1,1],[1,-1]):
    ##### define spin model
```

```python
    # site-coupling lists (PBC for both spin inversion sectors)
    h_field=[[-h,i] for i in range(L)]
    J_zz=[[-J,i,(i+1)%L] for i in range(L)] # PBC
    # define spin static and dynamic lists
    static_spin =[["zz",J_zz],["x",h_field]] # static part of H
    dynamic_spin=[] # time-dependent part of H
    # construct spin basis in pos/neg spin inversion sector depending on APBC/PBC
    basis_spin = spin_basis_1d(L=L,zblock=zblock)
    # build spin Hamiltonians
    H_spin=hamiltonian(static_spin,dynamic_spin,basis=basis_spin,dtype=np.float64)
    # calculate spin energy levels
    E_spin=H_spin.eigvalsh()
    ##### define fermion model
    # define site-coupling lists for external field
    h_pot=[[2.0*h,i] for i in range(L)]
    if PBC==1: # periodic BC: odd particle number subspace only
        # define site-coupling lists (including boudary couplings)
        J_pm=[[-J,i,(i+1)%L] for i in range(L)] # PBC
        J_mp=[[+J,i,(i+1)%L] for i in range(L)] # PBC
        J_pp=[[-J,i,(i+1)%L] for i in range(L)] # PBC
        J_mm=[[+J,i,(i+1)%L] for i in range(L)] # PBC
        # construct fermion basis in the odd particle number subsector
        basis_fermion = fermion_basis_1d(L=L,Nf=range(1,L+1,2))
    elif PBC==-1: # anti-periodic BC: even particle number subspace only
        # define bulk site coupling lists
        J_pm=[[-J,i,i+1] for i in range(L-1)]
        J_mp=[[+J,i,i+1] for i in range(L-1)]
        J_pp=[[-J,i,i+1] for i in range(L-1)]
        J_mm=[[+J,i,i+1] for i in range(L-1)]
        # add boundary coupling between sites (L-1,0)
        J_pm.append([+J,L-1,0]) # APBC
        J_mp.append([-J,L-1,0]) # APBC
        J_pp.append([+J,L-1,0]) # APBC
        J_mm.append([-J,L-1,0]) # APBC
        # construct fermion basis in the even particle number subsector
        basis_fermion = fermion_basis_1d(L=L,Nf=range(0,L+1,2))
    # define fermionic static and dynamic lists
    static_fermion =[["+-",J_pm],["-+",J_mp],["++",J_pp],["--",J_mm],['z',h_pot]]
    dynamic_fermion=[]
    # build fermionic Hamiltonian
    H_fermion=hamiltonian(static_fermion,dynamic_fermion,basis=basis_fermion,
                        dtype=np.float64,check_pcon=False,check_symm=False)
    # calculate fermionic energy levels
    E_fermion=H_fermion.eigvalsh()
```

The complete code including the lines that produce Fig. ?? is available in Example Code ??.

Integrability Breaking in Higher spin TFI model

This example shows how to:

- construct Hamiltonians for Higher spin operators.
- find ground state of a Hamiltonian

- use obs_vs_time function with costume user defined generator to calculate the expectation value of operators as a function of time.
- use the new functionality of the basis class to calculate the entanglement entropy for higher spin.

Physics Setup— In the previous section we introduced the TFI model and showed how one can solve the problem using the Jordan-Wigner transformation. This transformation allows one to get an exact analytic solution to the Hamiltonian (when the system obeys translational invariance). The fact that this solution exists in deeply connected to the notion of Integrability which has implications of how the system responds to a periodic modulation [CITE]. For non-integrable system when periodic driving, energy is no longer conserved and so generically one would expect that the system will heat up to infinite temperature, while in an integrable system, even though energy is not conserved, there are an extensive number of other static conserved quantities which may be conserved under the drive. If this is the case, then the system will not heat up at long times, but instead reach some steady state. By simply taking the transverse field ising model and promoting the spin-1/2 operators to spin-1, there is no longer a simple mapping to a quadratic Hamiltonian and therefore the model is no longer integrable. Here we will show this explicitly by driving the two different systems and checking it they heat or not. To do this we will define two Hamiltonians

$$H_{zz} = -\sum_{i=0}^{L-1} S^z_i S^z_{i+1}, \qquad H_x = -\sum_{i=0}^{L-1} S^x_i \tag{14}$$

and evolve the ferromagnetic ground state of H_{zz} with the following piecewise periodic Hamiltonian:

$$H(t) = H_{zz} - \Omega\, \mathrm{sgn}\left(\cos(\Omega t)\right) H_x \tag{15}$$

where Ω is the driving frequency and T is the period. As the Hamiltonian obeys translation,parity and spin-inversion symmetries we will use this to speed up the evolution by working in the symmetry sector which contains the ground state.

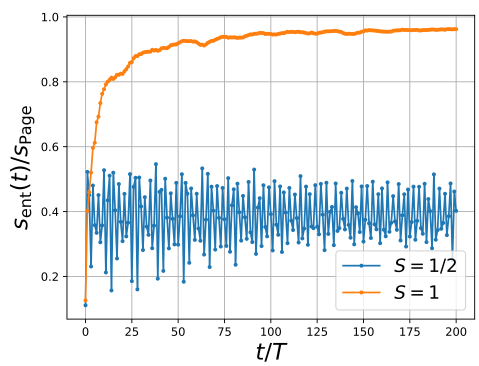
In order to measure the difference in heating between spin-1 and spin-1/2 we measure the expectation value of H_{zz} as a function of time. This operator has a symmetric spectrum and so following ref.[CITE heading papers] we define Q:

$$Q(t) = \left\langle \psi(t) \left| \frac{2\,(H_{zz} - E_{\mathrm{min}})}{E_{\mathrm{max}} - E_{\mathrm{min}}} \right| \psi(t) \right\rangle = \left\langle \psi(t) \left| \frac{H_{zz} - E_{\mathrm{min}}}{-E_{\mathrm{min}}} \right| \psi(t) \right\rangle \tag{16}$$

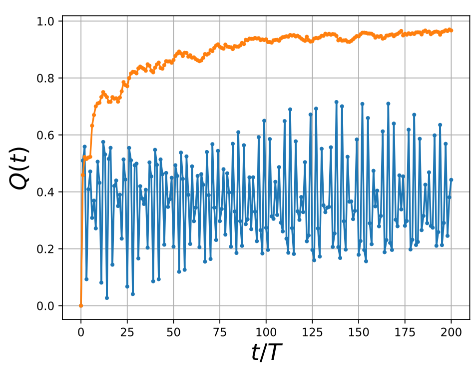
where the last equality comes from the symmetry of the spectrum: $E_{\mathrm{max}} = -E_{\mathrm{min}}$. This quantity is defined such that an infinite temperature state has $Q = 1$. Another measure of heating we will use is the entanglement entropy density

$$s_{\mathrm{ent}}(t) = -\frac{1}{|\mathrm{A}|}\mathrm{tr}_{\mathrm{A}}\left[\rho_{\mathrm{A}}(t)\log\rho_{\mathrm{A}}(t)\right], \qquad \rho_{\mathrm{A}}(t) = \mathrm{tr}_{\mathrm{A}^c}|\psi(t)\rangle\langle\psi(t)| \tag{17}$$

of subsystem A, defined to contain the left half of the chain and $|\mathrm{A}| = L/2$. We denoted the reduced density matrix of subsystem A by ρ_{A}, and A^c is the complement of A.
Code Analysis—...



(b) $s_{\mathrm{ent}}(t)/s_{\mathrm{Page}}$ for $t = nT$



(a) $Q(t)$ for $t = nT$

Figure 4: Comparing the dynamics of $Q(t)$ (a) and $s_{\mathrm{ent}}(t)$ (b) for $S = 1$ (orange) and $S = 1/2$ (blue) at stroboscopic times ($t = nT$). For $S = 1$ and $S = 1/2$ we take $L = 11$ and 18 respectively as to make sure the many-body Hilbert spaces have roughly the same number of state. s_{ent} is normalized by the Page entropy per site[CITE Page]. Note that for both systems $\Omega = 4$.

```
from __future__ import print_function, division

import numpy as np
from quspin.operators import ops_dict,hamiltonian,exp_op
from quspin.basis import spin_basis_1d
from quspin.tools.measurements import obs_vs_time
import matplotlib.pyplot as plt
import sys,os

# user defined generator
# generates stroboscopic dynamics
def evolve_gen(psi0,nT,*U_list):
    yield psi0
    for i in range(nT): # loop over number of periods
        for U in U_list: # loop over unitaries
            psi0 = U.dot(psi0)
        yield psi0

# frequency and period for driving.
omega = 2
T = 2*np.pi/omega
nT = 200 # number of periods to evolve to.
times = np.arange(0,nT+1,1)*T

L_1 = 18 # length of chain for spin 1/2
L_2 = 11 # length of chain for spin 1

basis_1 = spin_basis_1d(L_1,S="1/2",kblock=0,pblock=1,zblock=1) # spin 1/2 basis
basis_2 = spin_basis_1d(L_2,S="1"  ,kblock=0,pblock=1,zblock=1) # spin 1 basis
# print information about the basis
print("S = {S:3s}, L = {L:2d}, Size of H-space: {Ns:d}".format(S="1/2",L=L_1,Ns=
    basis_1.Ns))
print("S = {S:3s}, L = {L:2d}, Size of H-space: {Ns:d}".format(S="1"  ,L=L_2,Ns=
    basis_2.Ns))

# setting up coupling lists
Jzz_1 = [[-1.0,i,(i+1)%L_1] for i in range(L_1)]
hx_1  = [[-1.0,i] for i in range(L_1)]
Jzz_2 = [[-1.0,i,(i+1)%L_2] for i in range(L_2)]
hx_2  = [[-1.0,i] for i in range(L_2)]
# dictioanry to turn off checks
no_checks = dict(check_symm=False,check_herm=False)
# setting up hamiltonians
Hzz_1 = hamiltonian([["zz",Jzz_1]],[],basis=basis_1,dtype=np.float64)
Hx_1  = hamiltonian([["+",hx_1],["-",hx_1]],[],basis=basis_1,dtype=np.float64)
Hzz_2 = hamiltonian([["zz",Jzz_2]],[],basis=basis_2,dtype=np.float64,**no_checks)
Hx_2  = hamiltonian([["+",hx_2],["-",hx_2]],[],basis=basis_2,dtype=np.float64,**
    no_checks)
# calculating bandwidth for non-driven hamiltonian
[E_1_min],psi_1 = Hzz_1.eigsh(k=1,which="SA")
[E_2_min],psi_2 = Hzz_2.eigsh(k=1,which="SA")
# setting up initial states
psi0_1 = psi_1.ravel()
```

```
psi0_2 = psi_2.ravel()
# creating generators of time evolution
U1_1 = exp_op(Hzz_1+omega*Hx_1,a=-1j*T/4)
U2_1 = exp_op(Hzz_1-omega*Hx_1,a=-1j*T/2)
U1_2 = exp_op(Hzz_2+omega*Hx_2,a=-1j*T/4)
```

The complete code including the lines that produce Fig. **??** is available in Example Code **??**.

New Horizons for QuSpin

- 2D lattices
- single-particle Hamiltonian class
- Liouville dynamics

We would much appreciate it if the users could report bugs using the issues forum in the QuSpin online repository.

Acknowledgements

We would like to thank L. Pollet, M. Kolodrubetz, S. Capponi ... for various stimulating discussions and for providing comments on the draft. The authors are pleased to acknowledge that the computational work reported on in this paper was performed on the Shared Computing Cluster which is administered by Boston University's Research Computing Services. The authors also acknowledge the Research Computing Services group for providing consulting support which has contributed to the results reported within this paper. We would also like to thank Github for providing the online resources to help develop and maintain this project.

Funding information This work was supported by ???

Installation Guide in a Few Steps

QuSpin is currently only being supported for Python 2.7 and Python 3.5 and so one must make sure to install this version of Python. We recommend the use of the free package manager Anaconda which installs Python and manages its packages. For a lighter installation, one can use miniconda.

Mac OS X/Linux

To install Anaconda/miniconda all one has to do is execute the installation script with administrative privilege. To do this, open up the terminal and go to the folder containing the downloaded installation file and execute the following command:

```
$ sudo bash <installation_file>
```

You will be prompted to enter your password. Follow the prompts of the installation. We recommend that you allow the installer to prepend the installation directory to your PATH variable which will make sure this installation of Python will be called when executing a Python script in the terminal. If this is not done then you will have to do this manually in your bash profile file:

```
$ export PATH="path_to/anaconda/bin:$PATH"
```

Installing via Anaconda.—Once you have Anaconda/miniconda installed, all you have to do to install QuSpin is to execute the following command into the terminal:

```
$ conda install -c weinbe58 quspin
```

If asked to install new packages just say 'yes'. To keep the code up-to-date, just run this command regularly.

Installing Manually.—Installing the package manually is not recommended unless the above method failed. Note that you must have the Python packages NumPy, SciPy, and Joblib installed before installing QuSpin. Once all the prerequisite packages are installed, one can download the source code from github and then extract the code to whichever directory one desires. Open the terminal and go to the top level directory of the source code and execute:

```
$ python setup.py install --record install_file.txt
```

This will compile the source code and copy it to the installation directory of Python recording the installation location to `install_file.txt`. To update the code, you must first completely remove the current version installed and then install the new code. The `install_file.txt` can be used to remove the package by running:

```
$ cat install_file.txt | xargs rm -rf.
```

Windows

To install Anaconda/miniconda on Windows, download the installer and execute it to install the program. Once Anaconda/miniconda is installed open the conda terminal and do one of the following to install the package:

Installing via Anaconda.—Once you have Anaconda/miniconda installed all you have to do to install QuSpin is to execute the following command into the terminal:

```
> conda install -c weinbe58 quspin
```

If asked to install new packages just say 'yes'. To update the code just run this command regularly.

Installing Manually.—Installing the package manually is not recommended unless the above method failed. Note that you must have NumPy, SciPy, and Joblib installed before installing QuSpin. Once all the prerequisite packages are installed, one can download the source code from github and then extract the code to whichever directory one desires. Open the terminal and go to the top level directory of the source code and then execute:

```
> python setup.py install --record install_file.txt
```

This will compile the source code and copy it to the installation directory of Python and record the installation location to `install_file.txt`. To update the code you must first completely remove the current version installed and then install the new code.

Basic Use of Command Line to Run Python

In this appendix we will review how to use the command line for Windows and OS X/Linux to navigate your computer's folders/directories and run the Python scripts.

Mac OS X/Linux

Some basic commands:

- change directory:

  ```
  $ cd < path_to_directory >
  ```

- list files in current directory:

  ```
  $ ls
  ```

  list files in another directory:

  ```
  $ ls < path_to_directory >
  ```

- make new directory:

  ```
  $ mkdir <path>/< directory_name >
  ```

- copy file:

  ```
  $ cp < path >/< file_name > < new_path >/< new_file_name >
  ```

- move file or change file name:

  ```
  $ mv < path >/< file_name > < new_path >/< new_file_name >
  ```

- remove file:

  ```
  $ rm < path_to_file >/< file_name >
  ```

Unix also has an auto complete feature if one hits the TAB key. It will complete a word or stop when it matches more than one file/folder name. The current directory is denoted by "." and the directory above is "..". Now, to execute a Python script all one has to do is open your terminal and navigate to the directory which contains the python script. To execute the script just use the following command:

```
$ python script.py
```

It's that simple!

Windows

Some basic commands:

- change directory:

```
> cd < path_to_directory >
```

- list files in current directory:

```
> dir
```

  list files in another directory:

```
> dir < path_to_directory >
```

- make new directory:

```
> mkdir <path>\< directory_name >
```

- copy file:

```
> copy < path >\< file_name > < new_path >\< new_file_name >
```

- move file or change file name:

```
> move < path >\< file_name > < new_path >\< new_file_name >
```

- remove file:

```
> erase < path >\< file_name >
```

Windows also has a auto complete feature using the TAB key but instead of stopping when there multiple files/folders with the same name, it will complete it with the first file alphabetically. The current directory is denoted by "." and the directory above is "..".

Execute Python Script (any operating system)

To execute a Python script all one has to do is open up a terminal and navigate to the directory which contains the Python script. Python can be recognised by the extension `.py`. To execute the script just use the following command:

```
python script.py
```

It's that simple!

Package Documentation

In QuSpin quantum many-body operators are represented as matrices. The computation of these matrices are done through custom code written in Cython. Cython is an optimizing static compiler which takes code written in a syntax similar to Python, and compiles it into a highly efficient C/C++ shared library. These libraries are then easily interfaced with Python, but can run orders of magnitude faster than pure Python code [2]. The matrices are stored in a sparse matrix format using the sparse matrix library of SciPy [3]. This allows QuSpin to easily interface with mature Python packages, such as NumPy, SciPy, any many others. These packages provide reliable state-of-the-art tools for scientific computation as well as support from the Python community to regularly improve and update them [4, 5, 6, 3]. Moreover, we have included specific functionality in QuSpin which uses NumPy and SciPy to do many desired calculations common to ED studies, while making sure the user only has to call a few NumPy or SciPy functions directly. The complete up-to-date documentation for the package is available online under:

https://github.com/weinbe58/QuSpin/#quspin

Complete Example Codes

In this appendix, we give the complete python scripts for the dix examples discussed in Sec. 2. In case the reader has trouble with the TAB spaces when copying from the code environments below, the scripts can be downloaded from github at:

https://github.com/weinbe58/QuSpin/tree/master/examples

QuSpin *Example Code* 1: The Spectrum of the Transverse Field Ising Model and the Jordan-Wigner Transformation

```python
from quspin.operators import hamiltonian # Hamiltonians and operators
from quspin.basis import spin_basis_1d, fermion_basis_1d # Hilbert space spin basis
import numpy as np # generic math functions
import matplotlib.pyplot as plt # figure/plot library
##### define model parameters #####
L=8 # system size
J=1.0 # spin zz interaction
h=np.sqrt(2) # z magnetic field strength
# loop over spin inversion symmetry block variable and boundary conditions
for zblock,PBC in zip([-1,1],[1,-1]):
    ##### define spin model
    # site-coupling lists (PBC for both spin inversion sectors)
    h_field=[[-h,i] for i in range(L)]
    J_zz=[[-J,i,(i+1)%L] for i in range(L)] # PBC
    # define spin static and dynamic lists
    static_spin =[["zz",J_zz],["x",h_field]] # static part of H
    dynamic_spin=[] # time-dependent part of H
    # construct spin basis in pos/neg spin inversion sector depending on APBC/PBC
    basis_spin = spin_basis_1d(L=L,zblock=zblock)
    # build spin Hamiltonians
    H_spin=hamiltonian(static_spin,dynamic_spin,basis=basis_spin,dtype=np.float64)
    # calculate spin energy levels
    E_spin=H_spin.eigvalsh()
    ##### define fermion model
    # define site-coupling lists for external field
    h_pot=[[2.0*h,i] for i in range(L)]
    if PBC==1: # periodic BC: odd particle number subspace only
        # define site-coupling lists (including boudary couplings)
        J_pm=[[-J,i,(i+1)%L] for i in range(L)] # PBC
        J_mp=[[+J,i,(i+1)%L] for i in range(L)] # PBC
        J_pp=[[-J,i,(i+1)%L] for i in range(L)] # PBC
        J_mm=[[+J,i,(i+1)%L] for i in range(L)] # PBC
        # construct fermion basis in the odd particle number subsector
        basis_fermion = fermion_basis_1d(L=L,Nf=range(1,L+1,2))
    elif PBC==-1: # anti-periodic BC: even particle number subspace only
        # define bulk site coupling lists
        J_pm=[[-J,i,i+1] for i in range(L-1)]
        J_mp=[[+J,i,i+1] for i in range(L-1)]
        J_pp=[[-J,i,i+1] for i in range(L-1)]
        J_mm=[[+J,i,i+1] for i in range(L-1)]
        # add boundary coupling between sites (L-1,0)
        J_pm.append([+J,L-1,0]) # APBC
```

```
        J_mp.append([-J,L-1,0]) # APBC
        J_pp.append([+J,L-1,0]) # APBC
        J_mm.append([-J,L-1,0]) # APBC
        # construct fermion basis in the even particle number subsector
        basis_fermion = fermion_basis_1d(L=L,Nf=range(0,L+1,2))
    # define fermionic static and dynamic lists
    static_fermion =[["+-",J_pm],["-+",J_mp],["++",J_pp],["--",J_mm],['z',h_pot]]
    dynamic_fermion=[]
    # build fermionic Hamiltonian
    H_fermion=hamiltonian(static_fermion,dynamic_fermion,basis=basis_fermion,
                            dtype=np.float64,check_pcon=False,check_symm=False)
    # calculate fermionic energy levels
    E_fermion=H_fermion.eigvalsh()
    ##### plot spectra
    plt.plot(np.arange(H_fermion.Ns),E_fermion/L,marker='o'
                                    ,color='b',label='fermion')
    plt.plot(np.arange(H_spin.Ns),E_spin/L,marker='x'
                                    ,color='r',markersize=2,label='spin')
    plt.xlabel('state number',fontsize=16)
    plt.ylabel('energy',fontsize=16)
    plt.xticks(fontsize=16)
    plt.yticks(fontsize=16)
    plt.legend(fontsize=16)
    plt.grid()
    plt.tight_layout()
    plt.show()
```

References

[1] P. Weinberg and M. Bukov, *QuSpin: a Python Package for Dynamics and Exact Diagonalisation of Quantum Many Body Systems part I: spin chains*, SciPost Phys. **2**, 003 (2017), doi:10.21468/SciPostPhys.2.1.003.

[2] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn and K. Smith, *Cython: The best of both worlds*, Computing in Science & Engineering **13**(2), 31 (2011), doi:http://dx.doi.org/10.1109/MCSE.2010.118.

[3] E. Jones, T. Oliphant, P. Peterson *et al.*, *SciPy: Open source scientific tools for Python* (2001–).

[4] S. v. d. Walt, S. C. Colbert and G. Varoquaux, *The numpy array: A structure for efficient numerical computation*, Computing in Science & Engineering **13**(2), 22 (2011), doi:http://dx.doi.org/10.1109/MCSE.2011.37.

[5] T. E. Oliphant, *Python for scientific computing*, Computing in Science & Engineering **9**(3), 10 (2007), doi:http://dx.doi.org/10.1109/MCSE.2007.58.

[6] K. J. Millman and M. Aivazis, *Python for scientists and engineers*, Computing in Science & Engineering **13**(2), 9 (2011), doi:http://dx.doi.org/10.1109/MCSE.2011.36.