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 21, 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

1 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.

2 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.

2.1 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

$$H = \sum_{j=0}^{L-1} -J\sigma_{j+1}^z\sigma_j^z - h\sigma_j^x, \tag{1}$$

where the nearest-neighbour (nn) spin interaction is J, h denotes the transverse field, and σ_j^α are the Pauli spin-1/2 matrices. We use periodic boundary conditions and label the L lattice sites $0,\ldots,L-1$ 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]:

$$c_i = \prod_{j<i}\sigma_j^z\sigma_i^-, \qquad c_i^\dagger = \prod_{j<i}\sigma_j^z\sigma_i^+, \tag{2}$$

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.

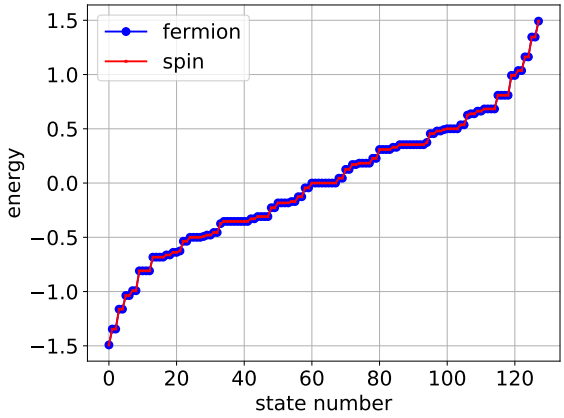
Code Analysis—We begin by loading the QuSpin operator and basis constructors, as well as some standard Python libraries.

```
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
```

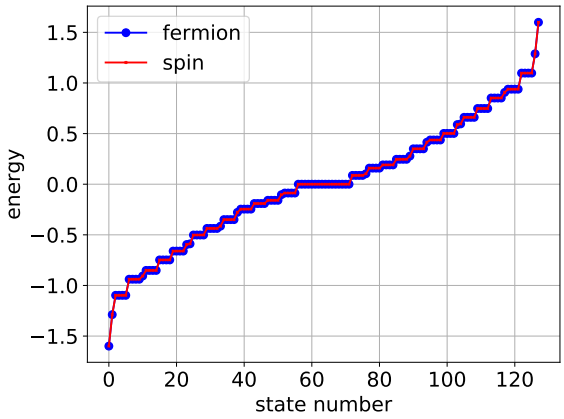
First, we define the models parameters.

```
##### 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



(a) negative spin inversion/PBC sector



(b) positive spin inversion/APBC 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 $\mathcal{O} = g\sum_{j=0}^{L-1}\sigma_j^\mu$ 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`→`g[j]`. Similarly, any two-body operator $\mathcal{O} = J_{zz}\sum_{j=0}^{L-1}\sigma_j^\mu\sigma_{j+1}^\nu$ 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 $\mathcal{O} = g\sum_{j=0}^{L-1}\sigma_j^x$ 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.

2.2 Free Particle Systems: the Fermionic SSH Chain

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

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

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

$$H = \sum_{k \in \mathrm{BZ}'} (a_k^\dagger, b_k^\dagger) \begin{pmatrix} \Delta & -(J+\delta J)\mathrm{e}^{-ik} - (J-\delta J)\mathrm{e}^{+ik} \\ -(J+\delta J)\mathrm{e}^{+ik} - (J-\delta J)\mathrm{e}^{-ik} & -\Delta \end{pmatrix} \begin{pmatrix} a_k \\ b_k \end{pmatrix}, \tag{5}$$

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

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

To evaluate the correlator numerically, we shall use the right-hand side of this equation.

As we are studying free particles, it is enough to work with the single-particle states. For instance the Fermi sea is given by $|\mathrm{FS}\rangle = \prod_{k\leq k_F} c_k^\dagger|0\rangle$. Denting the density operator in momentum space by $\hat{n}_i$, one can cast the correlator in momentum space in the following form:

$$C_{ij}(t) = \sum_{k\leq k_F} \langle k|\hat{n}_i(t)\hat{n}_j(0)|k\rangle. \tag{7}$$

If we want to consider finite-temperature β^{-1}, the above formula generalises to

$$C_{ij}(t,\beta) = \sum_{k\leq k_F} n_{\mathrm{FD}}(k,\beta)\langle k|\hat{n}_i(t)\hat{n}_j(0)|k\rangle, \tag{8}$$

where $n_{\mathrm{FD}}(k,\beta) = 1/(\exp(\beta E)+1)$ is the Fermi-Dirac distribution at temperature β^{-1}. Figure ??? shows the time evolution of $C_{ij}(t,\beta)$ for two sites, separated by the maximal distance on the ring: $L/2$.

Code Analysis.—Let us explain how one can do all this quickly and efficiently using QuSpin. As always, we start by loading the required packages and libraries.

```
from quspin.operators import hamiltonian,exp_op # Hamiltonians and operators
from quspin.basis import fermion_basis_1d # Hilbert space fermion basis
from quspin.tools.block_tools import block_diag_hamiltonian # block diagonalisation
import numpy as np # generic math functions
import matplotlib.pyplot as plt # plotting library
try: # import python 3 zip function in python 2 and pass if using python 3
    import itertools.izip as zip
except ImportError:
    pass
```

After that, we define the model parameters

```
##### define model parameters #####
L=100 # system size
J=1.0 # uniform hopping contribution
deltaJ=0.1 # bond dimerisation
Delta=0.5 # staggered potential
beta=100.0 # set inverse temperature for Fermi-Dirac distribution
```

In the following, we construct the fermionic SSH Hamiltonian first in real space. We then show how one can also construct it in momentum space where, provided we use periodic boundary conditions, it appear bock-diagonal. Let us define the fermionic site-coupling lists. Once again, we mention that fermion systems require special care in defining the hopping terms: Eq. (4) is conveniently cast in the form where all site indices on the operators grow to the right, and all signs due to the fermion statistics are explicitly spelt out.

```
##### construct single-particle Hamiltonian #####
# define site-coupling lists
hop_pm=[[-J-deltaJ*(-1)**i,i,(i+1)%L] for i in range(L)] # PBC
hop_mp=[[+J+deltaJ*(-1)**i,i,(i+1)%L] for i in range(L)] # PBC
stagg_pot=[[Delta*(-1)**i,i] for i in range(L)]
```

Defining the static list assigns the specific SSH operator structure. Since our problem does not possess any explicit time dependence, we leave the `dynamic` list empty.

```
# define static and dynamic lists
static=[["+-",hop_pm],["-+",hop_mp],['n',stagg_pot]]
dynamic=[]
```

Defining the fermion basis with the help of the constructor `fermion_basis_1d` proceeds as smoothly as in Sec. 2.1. Notice a cheap trick: by specifying a total of `Nf=1` fermion in the lattice, QuSpin actually allows to define single-particle models, as a special case of the more general many-body Hamiltonians. Unlike many body models, however, due to the exponentially reduced Hilbert space size, this allows us to scale up the system size `L`.

```
# define basis
basis=fermion_basis_1d(L,Nf=1)
```

We then build the real-valued SSH Hamiltonian in real space by passing the static and dynamic lists, as well as the basis ad the data type, and diagonalise it.

```
# build real-space Hamiltonian
H=hamiltonian(static,dynamic,basis=basis,dtype=np.float64)
# diagonalise real-space Hamiltonian
E,V=H.eigh()
```

For translation invariant single-particle models, however, the user might prefer to use momentum space, where the Hamiltonian becomes block diagonal, as we showed above. This can be achieved using QuSpin's `block_tools`. The idea behind it is simple: the main purpose is to create the full Hamiltonian in block-diagonal form, where the blocks correspond to predefined quantum numbers. In our case, we would like to use momentum or `kblock`'s. Note that the unit cell in the SSH model contains two sites, which we encode using the variable `a=2`. Thus, we can create a list of dictionaries `blocks`, each element of which defines a single symmetry block. If we combine all blocks, we exhaust the full Hilbert space. All other basis arguments, such as the system size, we store in the variable `basis_args`. We mention in passing that this procedure is independent of the symmetry, and can be done using all symmetries supported by QuSpin, not only translation.

```
##### compute Fourier transform and momentum-space Hamiltonian #####
# define basis blocks and arguments
blocks=[dict(Nf=1,kblock=i,a=2) for i in range(L//2)] # only L//2 distinct momenta
basis_args = (L,)
```

To create the block-diagonal Hamiltonian, we invoke the `block_diag_hamiltonian` method. It takes both requires and optional arguments, and returns the transformation which block-diagonalises the Hamiltonian (in our case the Fourier transform) and the block-diagonal Hamiltonian object. Required arguments, in order of appearance, are the `blocks`, the `static` and `dynamic` lists, the basis constructor, `basis_args`, and the data type. Since we expect the Hamiltonian to contain the Fourier factors $\exp(-ik)$, we know to choose a complex data type.

`block_diag_hamiltonian` also accepts some optional arguments, such as the flags for disabling the automatic built-in symmetry checks. More about this function can be found in the documentation, cf. App. C.

```
# construct block-diagonal Hamiltonian
FT,Hblock = block_diag_hamiltonian(blocks,static,dynamic,fermion_basis_1d,
                        basis_args,np.complex128,get_proj_kwargs=dict(pcon=True))
```

We can then use all functions and methods of the `hamiltonian` class, for instance the diagonalisation routines:

```
# diagonalise momentum-space Hamiltonian
Eblock,Vblock=Hblock.eigh()
```

We now proceed to calculate the correlation function from Eq. (6). To this end, we shall split the correlator according to the RHS of Eq. (6). Thus, the strategy is to evolve both the Fermi sea $|\mathrm{FS}(t)\rangle$ and the auxiliary state $|\mathrm{nFS(t)}\rangle$ in time, and compute the expectation value of the time-independent operator $n_i(0)$ in between the two states as a function of time. Last, keep in mind that we do not need to construct the Fermi sea as a many-body state explicitly, so we rather work with single-particle states.

The first step is to collect all momentum eigenstates into the array `psi`. We then build the operators $n_{j=0}$ and $n_{j=L/2}$ in real space.

```
##### prepare the density observables and initial states #####
# grab single-particle states and treat them as initial states
psi0=Vblock
# construct operator n_1 = $n_{j=0}$
n_1_static=[['n',[[1.0,0]]]]
n_1=hamiltonian(n_1_static,[],basis=basis,dtype=np.float64,
                check_herm=False,check_pcon=False)
# construct operator n_2 = $n_{j=L/2}$
n_2_static=[['n',[[1.0,L//2]]]]
n_2=hamiltonian(n_2_static,[],basis=basis,dtype=np.float64,
                check_herm=False,check_pcon=False)
```

We then transform these two operators to momentum space using the method `rotate_by()`. Setting the flag `generator=False` treats the Fourier transform `FT` as a unitary, rather than a generator of a unitary.

```
# transform n_j operators to momentum space
n_1=n_1.rotate_by(FT,generator=False)
n_2=n_2.rotate_by(FT,generator=False)
```

Let us proceed with the time-evolution. We first define the time vector `t` and the state `n_psi0`.

```
##### evaluate nonequal time correlator <FS|n_2(t) n_1(0)|FS> #####
# define time vector
t=np.linspace(0.0,90.0,901)
# calcualte state acted an by n_1
n_psi0=n_1.dot(psi0)
```

We can perform the time evolution in two ways: (i) we calculate the time-evolution operator `U` using the `exp_op` class, and apply it to the momentum states `psi0` and `n_psi0`. The `exp_op` class calculates the matrix exponential $\exp(aH)$ of an operator H multiplied by a complex number a. One can also conveniently compute a series of matrix exponentials $\exp(aHt)$ for

every time t by specifying the stating point `start`, endpoint `stop` and the number of elements `num` which define the time array t via `t=np.linspace(start,stop,num)`. Last, by parsing the flag `iterate=True` we create a generator – an object evaluated only at the time it is called, i.e. not pre-computed, which can save both time and memory.

```
# construct time-evolution operator using exp_op class (sometimes faster)
U = exp_op(Hblock,a=-1j,start=t.min(),stop=t.max(),num=len(t),iterate=True)
# evolve states
psi_t=U.dot(psi0)
n_psi_t = U.dot(n_psi0)
```

Another way of doing the time evolution, (ii), is to use `evolve()` method of the Hamiltonian class. The idea here is that every Hamiltonian defines a generator of time translations. This method solves Schrödinger equation using SciPy's ODE integration routines, see App. C for more details. The required aruments, in order of appearance, are the initial state, the initial time, and the time vector. The `evolve()` method also supports the option to create the output as a generator using the flag `iterate=True`. Both ways (i) and (ii) time-evolve all momentum states `psi` at once.

```
# alternative method for time evolution using Hamiltonian class
#psi_t=Hblock.evolve(psi0,0.0,t,iterate=True)
#n_psi_t=Hblock.evolve(n_psi0,0.0,t,iterate=True)
```

To evaluate the correlator, we first preallocate memory by defining the empty array `correlators`, which contains the correlator in every single-particle momentum mode $|k\rangle$. Using generators then allows us to loop only once over the time-evolved states `psi_t` and `n_psi_t`. In doing so, we evaluate the expectation value $\langle \mathrm{FS}(t)|n_i(0)|\mathrm{nFS}(t)\rangle$ using the `matrix_ele()` method of the `hamiltonian` class. The flag `diagonal=True` makes sure only the diagonal matrix elements are calculated[1] and returned as an one-dimensional array.

```
# preallocate variable
correlators=np.zeros(t.shape+psi0.shape[1:])
# loop over the time-evolved states
for i, (psi,n_psi) in enumerate( zip(psi_t,n_psi_t) ):
    correlators[i,:]=n_2.matrix_ele(psi,n_psi,diagonal=True).real
```

Finally, we weigh all singe-state correlators by the Fermi-Dirac distribution to obtain the finite-temperature non-equal time correlation function $C_{0,L/2}(t,\beta)$.

```
# evaluate correlator at finite temperature
n_FD=1.0/(np.exp(beta*E)+1.0)
correlator = (n_FD*correlators).sum(axis=-1)
```

The complete code including the lines that produce Fig. **??** is available in Example Code 2.

2.3 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.

[1] Recall that `psi_t` and `n_psi_t` contain many time-evolved states, and if one uses the default `diagonal=False`, all off-diagonal matrix elements will be computed as well, so the result will be an array

- 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:

$$H = -J \sum_{i=0,\sigma}^{L-2} \left(c^\dagger_{i\sigma} c_{i+1,\sigma} - c_{i\sigma} c^\dagger_{i+1,\sigma} \right) + U \sum_{i=0}^{L-1} n_{i\uparrow} n_{i\downarrow} + \sum_{i=0,\sigma}^{L-1} V_i n_{i\sigma} \tag{9}$$

where $c_{\sigma i}$ and $c^\dagger_{\sigma i}$ is a fermionic 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:

$$I = (N_A - N_B)/N_{\text{tot}} \tag{10}$$

where A and B refer to the different sublattices of the chain and N is the particle number operator. evolving with Hamiltonian (9) we will calculate its value which will tell us something 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— Once again we start out by loading a set of libraries we will need to procede with the calculation of the MBL fermions.

```
from quspin.operators import hamiltonian,exp_op,ops_dict # operators
from quspin.basis import tensor_basis,fermion_basis_1d # Hilbert spaces
from quspin.tools.measurements import obs_vs_time # calculating dynamics
import numpy as np # general math functions
from numpy.random import uniform,choice # tools for doing random sampling
from time import time # tool for calculating computation time
```

Some of these libraries and functions we have seen before but in this example we introduce the `ops_dict` which defines an operator which can be parameterized by many parameters as opposed to the Hamiltonian which is only parameterized by time. Also, in this script we will be doing many different disordered realisations in one simulations, therefore we will use `NumPy`'s random number library to produce the random sampling. We load `uniform` to generate the uniformly distributed random potential as well as `choice` which we will use to estimate the

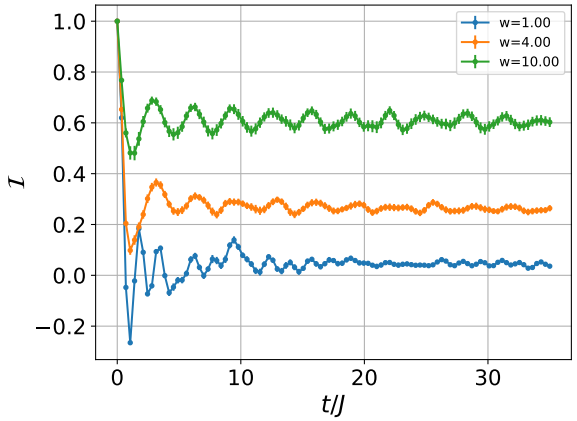


Figure 2: Sublattice imbalance I as a function of time averaged over 100 disorder realizations for different disorder strengths. This data was taken on a chain of length $L = 8$.

uncertainties of the disorder averages using a bootstrap re-sampling procedure which we will explain later. In order to time how long each realization takes we will use the `time` function from python's `time` library. After importing all the required libraries and function we set up the parameters for the simulation including number of realizations, and physical parameters `J`, U, number of up and down fermions etc.

```
##### setting parameters for simulation ####
# simulation parameters
n_real = 100 # number of realizations
n_boot = 100 # number of bootstrap samples to calculate error
# physical parameters
L = 8 # system size
N = L//2 # number of particles
w_list = [1.0,4.0,10.0] # disorder strength
J = 1.0 # hopping strength
U = 5.0 # interaction strength
# range to evolve system
start,stop,num=0.0,35.0,101
times = np.linspace(start,stop,num=num,endpoint=True)
# 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
```

Next we can set up the basis, introducing here the `tensor_basis` class. The `tensor_basis` class takes n basis objects which it then uses to construct the matrix elements in

$$\mathcal{H} = \mathcal{H}_1 \otimes \mathcal{H}_2 \otimes \cdots \otimes \mathcal{H}_n. \tag{11}$$

Here we will consider the case where we have two Hilbert spaces, one for fermions with spin up and one for fermions with spin down[2].

```
##### building the two basis to tensor together
basis_up = fermion_basis_1d(L,Nf=N_up) # up basis
```

[2] to construct a `tensor_basis` object in general: `t_basis = tensor_basis(basis_1,basis_2,...,basis_n)`

```
basis_down = fermion_basis_1d(L,Nf=N_down) # down basis
basis = tensor_basis(basis_up,basis_down) # spinful fermions
```

Next we can set up the coupling and operator lists

```
##### setting up physical problem
# creating coupling lists
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
# coupling list to create the sublattice imbalance observable
sublat_list = [[(-1)**i/N,i] for i in range(0,L)]
# create static lists
operator_list_0 = [
            ["+-|", hop_left], # up hop left
            ["-+|", hop_right], # up hop right
            ["|+-", hop_left], # down hop left
            ["|-+", hop_right], # down hop right
            ["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
# add key for Hubbard hamiltonian
operator_dict=dict(H0=operator_list_0)
# add keys for local potential in each site
for i in range(L):
    operator_dict["n"+str(i)] = [["n|",[[1.0,i]]],["|n",[[1.0,i]]]]
###### setting up operators
```

Notice here that the "|" characters is used to separate the operators which belong to the up (left side of tensor product) and down (right side of tensor product) Hilbert spaces in the tensor product. If no string in present the operator is assumed to be the identity. The index list, on the other hand, does not have to have anything separating the two sides of the tensor product it is assumed that the character lines up with the correct site index when the "|" characters are removed. The last lines of the code above we define a python **dictionary** object for which we add static operator lists as values in the dictionary indexed by a particular string or key. This dictionary is what gets passed into the **ops_dict** class which for each key in the dictioanry constructs the operator listed in the state list for that key. Then when one would like to evaluate this operator at a particular set of parameters one uses another dictionary where the same keys index a number which multiples the operator stored at that same key. In this way one can now parameterize the many-body operator in more complicated ways. In this case we define a key for the Hubbard hamiltonian and then keys for the local density operator for both up and down spins on each site. In this way we can then construct any disordered Hamiltonian by specify the disorder at each site.

```
# set up hamiltonian dictionary and observable
no_checks = dict(check_pcon=False,check_symm=False,check_herm=False)
H_dict = ops_dict(operator_dict,basis=basis,**no_checks)
I = hamiltonian([["n|",sublat_list],["|n",sublat_list]],[],basis=basis,**no_checks)
# strings which represent the initial state
```

Here the **ops_dict** class is constructed in almost an identical manner as a **hamiltonian** object with the excpetion that there is no dynamic operators. Next we must construct our initial

state with the fermions dispersed over the lattice on every other site. To get the index of the state which this corresponds to one can use the **index** function of the tensor basis. This function takes a string or integer representing the product state for each of the hilbert spaces and then searches to find the total product state and returns the index which corresponds to this state.

```
s_up = "".join("1000" for i in range(N_up))
s_down = "".join("0010" for i in range(N_down))
# basis.index accepts strings and returns the index
# which corresponds to that state in the basis list
i_0 = basis.index(s_up,s_down) # find index of product state
psi_0 = np.zeros(basis.Ns) # allocate space for state
psi_0[i_0] = 1.0 # set state to be in the given product state
```

Now that the operators are all set up, next we define a function which given a disorder realization of the local potential will calculate the dynamics of $\mathcal{I}$. The syntax to define a function is as follows:

```
def real(H_dict,I,psi_0,w,times,i):
    # body of function goes below
    ti = time() # start timing function
    # create a parameter list which specifies the onsite potential with disorder
```

The first task is to construct a hamiltonian from the disorder list **disorder** which is as simple as:

```
    # create a parameter list which specifies the onsite potential with disorder
    params=dict(H0=1)
    for i in range(L):
        params["n"+str(i)] = uniform(-w,w)
    # using the parameters dictionary construct a hamiltonian object with those
    # parameters defined in the list
    H = H_dict.tohamiltonian(params)
```

using the **tohamiltonian** function of **H_dict**. Once the hamiltonian has been constructed we can construct the **exp_op** object out of it and use it to create the generator of time dependent states

```
    # use exp_op to get the evolution operator
    U = exp_op(H,a=-1j,start=times.min(),stop=times.max(),num=len(times),iterate=
    True)
    psi_t = U.dot(psi_0) # get generator of time evolution
    # use obs_vs_time to evaluate the dynamics
    times = U.grid # time grid stored in U
    obs_t = obs_vs_time(psi_t,times,dict(I=I))
```

The function ends by printing the time of executing and returing the value for I as a function of time for this realization

```
    # print reporting the computation time for realization
    print("realization {}/{} completed in {:.2f} s".format(i+1,n_real,time()-ti))
    # return observable values.
    return obs_t["I"]
```

Now we are all set to run the disorder realizations for the different disorder strengths which in principle can be spilt up over multiple simulations but for completeness we do all of the calculations in one script.

```
###### looping over differnt disorder strengths
for w in w_list:
    I_data = np.vstack([real(H_dict,I,psi_0,w,times,i) for i in range(n_real)])
```

Which after this we then calculate the average and the error using bootstrap re-sampling (See Appendix??)

```
    ##### averaging and error estimation
    I_avg = I_data.mean(axis=0) # get mean value of I for all time points
    # generate bootstrap samples
    bootstrap_gen = (I_data[choice(n_real,size=n_real)].mean(axis=0) for i in range(
    n_boot))
    # generate the fluctuations about the mean of I
    sq_fluc_gen = ((bootstrap-I_avg)**2 for bootstrap in bootstrap_gen)
    I_error = np.sqrt(sum(sq_fluc_gen)/n_boot)
```

The complete code including the lines that produce Fig. ?? is available in Example Code 2.

2.4 Bose-Hubbard Model on Translationally Invariant Ladder

This example shows how to:

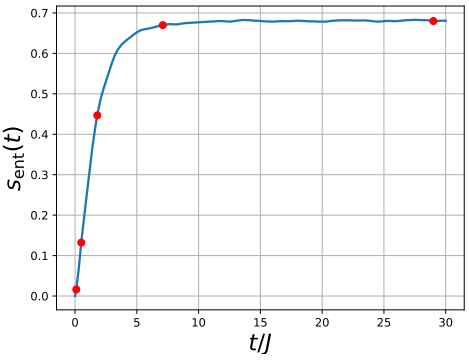
- construct Hamiltonians for bosonic systems.
- construct ladder Hamiltonians.
- using block_ops class to evolve over several symmetry sectors at once.
- measure entanglement entropy of ladder.

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:

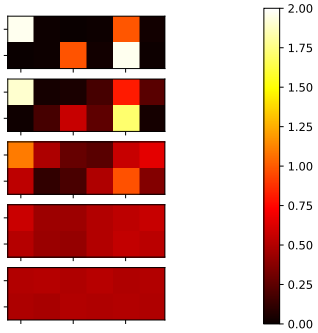
$$H_{\text{BHM}} = -J \sum_{\langle ij \rangle} a_i^\dagger a_j + \text{h.c.} + U \sum_i n_i(n_i - 1) \tag{12}$$

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 (12). 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



(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 $U = 10$.

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
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.

```
# 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`

```
# 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 $-U\sum_i n_i$:

```
# 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

```
# 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 `list_1.extend(list_2)` method to concatenate two lists together[3]. 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 which we will not use to construct a `hamiltonian` object but instead we will use the `block_ops` class.

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

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 space-time correlation functions of local operators in a translationally invariant system, while in this section we explicitly start out with a state which does not obey translational invariance. To construct the `block_ops` object we use the follow set of code

```
# creating block_ops object
blocks=[dict(kblock=kblock) for kblock in range(L)] # blocks to project on to
baisis_args = (N,) # boson_basis_1d manditory arguments
```

[3] Note that the `extend` function is done inplace so if you try to do `new_list=list_1.extend(list_2)`, `new_list` will be `None` and `list_1` will have all of the elements of `list_2` appended to it.

```
basis_kwargs = dict(nb=nb,sps=sps,a=2) # boson_basis_1d optional args
get_proj_kwargs = dict(pcon=True) # set projection to full particle basis
U_block = block_ops(blocks,static,dynamic,boson_basis_1d,baisis_args,np.complex128,
                    basis_kwargs=basis_kwargs,get_proj_kwargs=get_proj_kwargs)
```

Firstly, `blocks` is a list of dictionaries which define the different symmetry sectors to evolve the initial state over[4]. Secondly `basis_args` and `basis_kwargs` define optional arguments which apply to every symmetry sector. Finally `get_proj_kwargs` contains the optional arguments to construct the projectors[5]. For more information about this class we refer the user to the Documentation. Finally we define the initial state and the local density operators using the `boson_basis_1d` class.

```
# setting up basis for local fock basis
basis = boson_basis_1d(N,nb=nb,sps=sps)
# setting up observables
sub_sys_A = range(0,N,2) # bottom side of ladder
no_checks = dict(check_herm=False,check_symm=False,check_pcon=False)
n_list = [hamiltonian([["n",[[1.0,i]]]],[],basis=basis,dtype=np.float64,**no_checks)
     for i in range(N)]
# set up initial state
i0 = np.random.randint(basis.Ns) # pick random state from basis set
psi = np.zeros(basis.Ns,dtype=np.float64)
psi[i0] = 1.0
# print info about setup
state_str = "".join(str(int((basis[i0]//basis.sps**i)%basis.sps)) for i in range(N))
print("total H-space size: {}, initial state: |{}>".format(basis.Ns,state_str))
```

Next we can calculate the time dependent states by using the `expm` function of the `block_ops` class

```
# calculating the evolved states
n_jobs = 1 # increase this to see if calculation runs faster!
psi_t = U_block.expm(psi,start=start,stop=stop,num=num,block_diag=False,n_jobs=
    n_jobs)
```

We define this function to have almost identical arguments as that of the `exp_op` class, but with some major exceptions. For one, because the hamiltonian factorizes the evolution over each block can be done separately (e.g. just trivially loop through them sequentially), but in some cases where there are a lot of small blocks like in a single particle Hamiltonian, then it actually makes sense to calculate the matrix exponential in block diagonal form which is why we added the optional argument `block_diag` which toggles between these two options. Another option we have put in allows the user to spawn multiply python processes to do split up the work of doing the calculations for different blocks simultaneously, by setting `n_jobs` to be greater than 1. On most systems these processes will be distributed over multiple CPUs which can speed up the calculations if the resources are there. This also works on conjunction with the `block_diag` where each process creates its own block diagonal matrix for the calculation. Once all the calculations for each block are completed the results are then combined and projected to the local Fock basis. Finally, with the time dependent states calculated we can calculate the expectation values and entanglement entropy.

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

[5] In this case setting `pcon=True` means that the projector takes the state from the symmetry reduced basis to the fixed particle number basis

```
# calculating entanglement entropy
gen = (basis.ent_entropy(psi,sub_sys_A=sub_sys_A)["Sent_A"]/L for psi in psi_t.T[:])
ent_t = np.fromiter(gen,dtype=np.float64,count=num)
# calculating the local densities as a function of time
expt_n_t = np.vstack([n.expt_value(psi_t).real for n in n_list]).T
```

In the newer versions of QuSpin we have moved the entanglement entropy calculations to the basis classes themselves (keeping of course backwards compatible functions from older versions), the rational being that this calculation is highly dependent on the type of system one is studying. Here we show how to use to use the basis function to calculate the entanglement entropy but this function can also calculate the reduced density matrix and its eigenvalues for pure and mixed states. Once again we refer the reader to the Documentation to learn more about how to use this function.

2.5 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:

$$\begin{aligned} i\partial_t\psi_j(t) &= -J\left(\psi_{j-1}(t)+\psi_{j+1}(t)\right)+\frac{1}{2}\omega_{\rm trap}(t)(j-j_0)^2\psi_j(t)+U|\psi_j(t)|^2\psi_j(t), \\ \omega_{\rm trap}(t) &= (\omega_f-\omega_i)t/t_{\rm ramp}+\omega_i \end{aligned} \tag{13}$$

where J is the hopping matrix element, $\omega_{\rm trap}(t)$ – the slowly-varying time-dependent harmonic trap 'frequency' over a time scale $t_{\rm ramp}$, and U – the interaction 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. (13) in the following generic form

$$i\partial_t\vec{\psi}(t)=H_{\rm sp}(t)\vec{\psi}(t)+U\vec{\psi}^*(t)\circ\vec{\psi}(t)\circ\vec{\psi}(t), \tag{14}$$

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, $H_{\rm sp}(t)$, 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:

$$\begin{aligned}\vec{\psi}_{\rm GS} &= \inf_{\vec{\psi}}\left(\vec{\psi}^t H_{\rm sp}(0)\vec{\psi}+\frac{U}{2}\sum_{j=0}^{L-1}|\psi_j|^4\right),\\ &= \inf_{\psi_j}\left(\sum_{j=0}^{L-1}-J(\psi^*_{j+1}\psi_j+\text{c.c.})+\frac{1}{2}\omega_{\rm trap}(0)|\psi_j|^2+\frac{U}{2}|\psi_j|^4\right).\end{aligned} \tag{15}$$

One way to find the configuration $\vec{\psi}_{\rm GS}$, 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:

$$\begin{aligned}\partial_\tau\vec{\varphi}(\tau) &= -\Big[H_{\rm sp}(0)\vec{\varphi}(\tau)+U\vec{\varphi}^*(\tau)\circ\vec{\varphi}(\tau)\circ\vec{\varphi}(\tau)\Big], \qquad ||\vec{\varphi}(\tau)||=\text{const.},\\ \vec{\psi}_{\rm GS} &= \lim_{\tau\to\infty}\vec{\varphi}(\tau)\end{aligned} \tag{16}$$

Once we have the initial state $\vec{\psi}_{\rm GS}$, we evolve it according to the time-dependent GPE, Eq. (13), and track down the time evolution of the condensate density $\rho_j(t)=|\psi_j(t)|^2$. Fig. ??? shows the result.

Code Analysis—In the following, we demonstrate how one can code the above physics using QuSpin. As usual, we begin by loading the necessary packages

```
from quspin.operators import hamiltonian # Hamiltonians and operators
from quspin.basis import boson_basis_1d # Hilbert space boson basis
from quspin.tools.measurements import evolve
import numpy as np # generic math functions
#import scipy.sparse as sp # sparse matrices library
import matplotlib.pyplot as plt # plot library
```

Next, we define the model parameters. We distinguish between static parameters and dynamic parameters – those involved in the trap widening.

```
##### define model parameters #####
L=300 # system size
# calculate centre of chain
if L%2==0:
    j0 = L//2-0.5 # centre of chain
else:
    j0 = L//2 # centre of chain
sites=np.arange(L)-j0
# static parameters
J=1.0 # hopping
U=1.0 # Bose-Hubbard interaction strength
# dynamic parameters
omega_trap_i=0.001 # initial chemical potential
omega_trap_f=0.0001 # final chemical potential
t_ramp=40.0/J # set total ramp time
```

In order to do time evolution, we define the trap widening protocol from Eq. (13). Since we want to make use of QuSpin's time-dependent operator features, the first argument must be the time vector, followed by all protocol parameters. These same parameters are then listed in the variable `ramp_args`.

```
# ramp protocol
def ramp(t,omega_trap_i,omega_trap_f,t_ramp):
    return  (omega_trap_f - omega_trap_i)*t/t_ramp + omega_trap_i
# ramp protocol parameters
ramp_args=[omega_trap_i,omega_trap_f,t_ramp]
```

With this, we are ready to construct the single-particle Hamiltonian. The first step is to define the site-coupling lists, and the static and dynamic lists. Note that the dynamic list, which defines the harmonic potential in the single-particle Hamiltonian, contains four elements: apart from the operator string and the corresponding site-coupling list, the third and fourth elements are the time-dependent function `ramp` and its arguments `ramp_args` in this order.

```
##### construct single-particle Hamiltonian #####
# define site-coupling lists
hopping=[[-J,i,(i+1)%L] for i in range(L-1)]
trap=[[0.5*(i-j0)**2,i] for i in range(L)]
# define static and dynamic lists
static=[["+-",hopping],["-+",hopping]]
dynamic=[['n',trap,ramp,ramp_args]]
```

To create a single-particle Hamiltonian, we choose to use the bosonic basis constructor `boson_basis_1d` specifying the number sector to `Nb=1` boson for the entire lattice, and a local Hilbert space of `sps=2` states per site (empty and filled).

```
# define basis
basis = boson_basis_1d(L,Nb=1,sps=2)
```

Then we call the `hamiltonian` constructor to build the single-particle matrix. We can obtain the single-particle ground state without fully diagonalising the matrix by using the sparse diagonalisation attribute `Hsp.eigsh()`, with the options `k=1` and `'which'='SA'` which specify taking only one state starting from the bottom of the spectrum, i.e. the ground state.

```
# build Hamiltonian
Hsp=hamiltonian(static,dynamic,basis=basis,dtype=np.float64)
E,V=Hsp.eigsh(time=0.0,k=1,which='SA')
```

The next step in the program is to compute the ground state of the GPE using imaginary time evolution from Eq. (16). To this end, we first define the function `GPE_imag_time` which evaluate the RHS. It is required that the first argument for this function is (imaginary) time `tau`, followed by the state `phi`. All other arguments, such as the single-particle Hamiltonian and the interaction strength are listed last. Similar to before, we store these optional arguments in a list which we call `GPE_params`.

```
##### imaginary-time evolution to compute GS of GPE #####
def GPE_imag_time(tau,phi,Hsp,U):
    """
    This function solves the real-valued GPE in imaginary time:
    $$ -\dot\phi(\tau) = Hsp(t=0)\phi(\tau) + U |\phi(\tau)|^2 \phi(\tau) $$
    """
    return -( Hsp.dot(phi,time=0) + U*np.abs(phi)**2*phi )
```

```
# define ODE parameters
GPE_params = (Hsp,U)
```

Any initial value problem requires us to pick an initial state. In the case of imaginary evolution, this state can often be arbitrary, but needs to possess the same symmetries as the true GPE ground state. Here, we choose the ground state of the single-particle Hamiltonian for an initial state, and normalise it to one particle per site. We also define the imaginary time vector `tau`. This array has to contain sufficiently long times so that we make sure we obtain the long imaginary time limit $\tau \to \infty$, as required by Eq. (16). SInce imaginary time evolution is not unitary, QuSpin will be normalising the vector every τ-step. Thus, one also needs to make sure these steps are small enough to avoid convergence problems of the ODE solver.

```
# define initial state to flow to GS from
phi0=V[:,0]*np.sqrt(L) # initial state normalised to 1 particle per site
# define imaginary time vector
tau=np.linspace(0.0,35.0,71)
```

Performing Imaginary time evolution is done using the `evolve()` method of the `measurements` tool. This function accepts an initial state `phi0`, initial time `tau[0]`, and a time vector `tau` and solves the user-defined ODE `GPE_imag_time`. The parameters of the ODE are passed using the keyword argument `f_params=GPE_params`. To ensure the normalisation of the state at each τ-step we use the flag `imag_time=True`. Real-valued output can be specified by `real=True`. Last, we request `evolve()` to create a generator object using the keyword argument `iterate=True`. Many of the keyword arguments of `evolve()` are the same as in the `H.evolve()` method of the `hamiltonian class`: for instance, one can choose a specific SciPy solver and its arguments, or the solver's absolute and relative tolerance. We refer the interested reader to the documentation, cf. App. C.

```
# evolve state in imaginary time
psi_tau = evolve(phi0,tau[0],tau,GPE_imag_time,f_params=GPE_params,
                             imag_time=True,real=True,iterate=True)
```

Looping over the generator `phi_tau` we have access to the solution, which we display in a form of a movie:

```
# display state evolution
for i,psi0 in enumerate(psi_tau):
    # compute energy
    E_GS=(Hsp.matrix_ele(psi0,psi0,time=0) + 0.5*U*np.sum(np.abs(psi0)**4) ).real
    # plot wave function
    plt.plot(sites, abs(phi0)**2, color='r',marker='s',alpha=0.2,
                                      label='$|\\phi_j(0)|^2$')
    plt.plot(sites, abs(psi0)**2, color='b',marker='o',
                                label='$|\\phi_j(\\tau)|^2$' )
    plt.xlabel('$\\mathrm{lattice\\ sites}$',fontsize=14)
    plt.title('$J\\tau=%0.2f,\\ E_\\mathrm{GS}(\\tau)=%0.4fJ$'%(tau[i],E_GS)
                                                          ,fontsize=14)
    plt.ylim([-0.01,max(abs(phi0)**2)+0.01])
    plt.legend(fontsize=14)
    plt.draw() # draw frame
    plt.pause(0.005) # pause frame
    plt.clf() # clear figure
plt.close()
```

Last, we use our GPE ground state, to time-evolve it in real time according to the trap widening protocol hard-coded into the single-particle Hamiltonian. We proceed analogously – first we define the real-time GPE and the time vector. In defining the GPE function, we split the ODE into a time-independent static part and a time-dependent dynamic part. The single-particle Hamiltonian for the former is accessed using the `hamiltonian` attribute `Hsp.static` which returns a sparse matrix. We can then manually add the non-linear cubic mean-field interaction term. In order to access the time-dependent part of the Hamiltonian, and evaluate it, we loop over the dynamic list `Hsp.dynamic`, reading off the orresponding operator `Hd` together with the time-dependent function `f` which multiplies it, and its arguments `f_args`. Last, we multiply the final ouput vector by the Schrödinger $-i$, which ensures the unitarity of for real-time evolution.

```
##### real-time evolution of GPE #####
def GPE(time,psi):
    """
    This function solves the complex-valued time-dependent GPE:
    $$ i\dot\psi(t) = Hsp(t)\psi(t) + U |\psi(t)|^2 \psi(t) $$
    """
    # solve static part of GPE
    psi_dot = Hsp.static.dot(psi) + U*np.abs(psi)**2*psi
    # solve dynamic part of GPE
    for Hd,f,f_args in Hsp.dynamic:
        psi_dot += f(time,*f_args)*Hd.dot(psi)
    return -1j*psi_dot
# define real time vector
t=np.linspace(0.0,t_ramp,101)
```

To perform real-time evolution we once again use the `evolve()` function. This time, however, since the solution of the GPE is anticipated to be complex-valued, and because we do not do imaginary time, we do not need to pass the flags `real` and `imag_time`. Instead, we decided to show the flags for the relative and absolute tolerance of the solver.

```
# time-evolve state according to GPE
psi_t = evolve(psi0,t[0],t,GPE,iterate=True,atol=1E-12,rtol=1E-12)
```

Finally, we can enjoy the movie displaying real-time evolution

```
# display state evolution
for i,psi in enumerate(psi_t):
    # compute energy
    E=(Hsp.matrix_ele(psi,psi,time=t[i]) + 0.5*U*np.sum(np.abs(psi)**4) ).real
    # compute trap
    omega_trap=ramp(t[i],omega_trap_i,omega_trap_f,t_ramp)*(sites)**2
    # plot wave function
    plt.plot(sites, abs(psi0)**2, color='r',marker='s',alpha=0.2
                                ,label='$|\\psi_{\\mathrm{GS},j}|^2$')
    plt.plot(sites, abs(psi)**2, color='b',marker='o',label='$|\\psi_j(t)|^2$')
    plt.plot(sites, omega_trap,'--',color='g',label='$\\mathrm{trap}$')
    plt.ylim([-0.01,max(abs(psi0)**2)+0.01])
    plt.xlabel('$\\mathrm{lattice\\ sites}$',fontsize=14)
    plt.title('$Jt=%0.2f,\\ E(t)-E_\\mathrm{GS}=%0.4fJ$'%(t[i],E-E_GS),fontsize=14)
    plt.legend(loc='upper right',fontsize=14)
    plt.draw() # draw frame
    plt.pause(0.00005) # pause frame
```

```
    plt.clf() # clear figure
plt.close()
```

The complete code including the lines that produce Fig. **??** is available in Example Code 3.

2.6 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

$$H_{zz} = -\sum_{i=0}^{L-1} S_i^z S_{i+1}^z, \qquad H_x = -\sum_{i=0}^{L-1} S_i^x \tag{17}$$

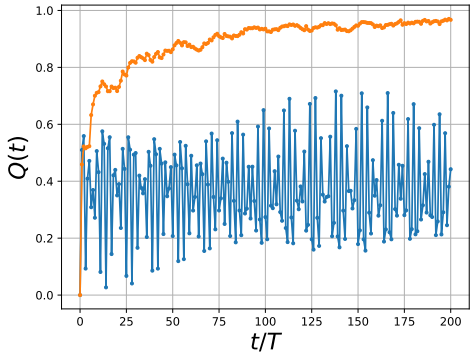
and evolve the ferromagnetic ground state of H_{zz} with the following piecewise periodic Hamiltonian:

$$H(t) = H_{zz} - \Omega \,\mathrm{sgn}\left(\cos(\Omega t)\right) H_x \tag{18}$$

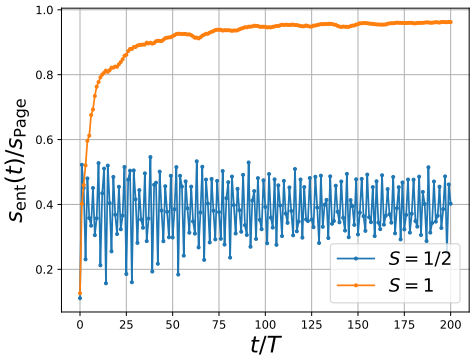
where Ω is the driving frequency and T is the period. As the Hamiltonian obeys translation,parity and spin-inversion symmetries we will use this to speed up the evolution by working in the symmetry sector which contains the ground state.

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:

$$Q(t) = \left\langle \psi(t) \left| \frac{2\left(H_{zz} - E_{\min}\right)}{E_{\max} - E_{\min}} \right| \psi(t) \right\rangle = \left\langle \psi(t) \left| \frac{H_{zz} - E_{\min}}{-E_{\min}} \right| \psi(t) \right\rangle \tag{19}$$



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



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

Figure 4: Comparing the dynamics of $Q(t)$ (a) and $s_{\rm ent}(t)$ (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. $s_{\rm ent}$ is normalized by the Page entropy per site[CITE Page]. Note that for both systems $\Omega = 4$.

where the last equality comes from the symmetry of the spectrum: $E_{\rm max} = -E_{\rm min}$. 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

$$s_{\rm ent}(t) = -\frac{1}{|\mathrm{A}|}\mathrm{tr}_{\mathrm{A}}\left[\rho_{\mathrm{A}}(t)\log\rho_{\mathrm{A}}(t)\right], \qquad \rho_{\mathrm{A}}(t) = \mathrm{tr}_{\mathrm{A^c}}|\psi(t)\rangle\langle\psi(t)| \tag{20}$$

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 $\mathrm{A^c}$ is the complement of A.

Code Analysis—...

```
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
###### setting up basis ######
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)
U2_2 = exp_op(Hzz_2-omega*Hx_2,a=-1j*T/2)
# get generator objects to get time dependent states
psi_1_t = evolve_gen(psi0_1,nT,U1_1,U2_1,U1_1)
psi_2_t = evolve_gen(psi0_2,nT,U1_2,U2_2,U1_2)
```

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

3 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 ???

A 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.

A.1 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.
```

A.2 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.

B 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.

B.1 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!

B.2 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 "..".

B.3 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!

C 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

D 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

```
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()
```

QuSpin *Example Code* 2: Free Particle Systems: the Fermionic SSH Chain

```
from quspin.operators import hamiltonian,exp_op # Hamiltonians and operators
from quspin.basis import fermion_basis_1d # Hilbert space fermion basis
from quspin.tools.block_tools import block_diag_hamiltonian # block diagonalisation
import numpy as np # generic math functions
import matplotlib.pyplot as plt # plotting library
try: # import python 3 zip function in python 2 and pass if using python 3
    import itertools.izip as zip
except ImportError:
    pass
##### define model parameters #####
L=100 # system size
J=1.0 # uniform hopping contribution
deltaJ=0.1 # bond dimerisation
Delta=0.5 # staggered potential
beta=100.0 # set inverse temperature for Fermi-Dirac distribution
##### construct single-particle Hamiltonian #####
# define site-coupling lists
hop_pm=[[-J-deltaJ*(-1)**i,i,(i+1)%L] for i in range(L)] # PBC
hop_mp=[[+J+deltaJ*(-1)**i,i,(i+1)%L] for i in range(L)] # PBC
stagg_pot=[[Delta*(-1)**i,i] for i in range(L)]
# define static and dynamic lists
static=[["+-",hop_pm],["-+",hop_mp],['n',stagg_pot]]
dynamic=[]
# define basis
basis=fermion_basis_1d(L,Nf=1)
# build real-space Hamiltonian
H=hamiltonian(static,dynamic,basis=basis,dtype=np.float64)
# diagonalise real-space Hamiltonian
E,V=H.eigh()
##### compute Fourier transform and momentum-space Hamiltonian #####
# define basis blocks and arguments
blocks=[dict(Nf=1,kblock=i,a=2) for i in range(L//2)] # only L//2 distinct momenta
basis_args = (L,)
# construct block-diagonal Hamiltonian
FT,Hblock = block_diag_hamiltonian(blocks,static,dynamic,fermion_basis_1d,
                        basis_args,np.complex128,get_proj_kwargs=dict(pcon=True))
# diagonalise momentum-space Hamiltonian
Eblock,Vblock=Hblock.eigh()
##### prepare the density observables and initial states #####
# grab single-particle states and treat them as initial states
psi0=Vblock
# construct operator n_1 = $n_{j=0}$
n_1_static=[['n',[[1.0,0]]]]
n_1=hamiltonian(n_1_static,[],basis=basis,dtype=np.float64,
                check_herm=False,check_pcon=False)
# construct operator n_2 = $n_{j=L/2}$
n_2_static=[['n',[[1.0,L//2]]]]
n_2=hamiltonian(n_2_static,[],basis=basis,dtype=np.float64,
                check_herm=False,check_pcon=False)
# transform n_j operators to momentum space
n_1=n_1.rotate_by(FT,generator=False)
```

```
n_2=n_2.rotate_by(FT,generator=False)
##### evaluate nonequal time correlator <FS|n_2(t) n_1(0)|FS> #####
# define time vector
t=np.linspace(0.0,90.0,901)
# calcualte state acted an by n_1
n_psi0=n_1.dot(psi0)
# construct time-evolution operator using exp_op class (sometimes faster)
U = exp_op(Hblock,a=-1j,start=t.min(),stop=t.max(),num=len(t),iterate=True)
# evolve states
psi_t=U.dot(psi0)
n_psi_t = U.dot(n_psi0)
# alternative method for time evolution using Hamiltonian class
#psi_t=Hblock.evolve(psi0,0.0,t,iterate=True)
#n_psi_t=Hblock.evolve(n_psi0,0.0,t,iterate=True)
# preallocate variable
correlators=np.zeros(t.shape+psi0.shape[1:])
# loop over the time-evolved states
for i, (psi,n_psi) in enumerate( zip(psi_t,n_psi_t) ):
    correlators[i,:]=n_2.matrix_ele(psi,n_psi,diagonal=True).real
# evaluate correlator at finite temperature
n_FD=1.0/(np.exp(beta*E)+1.0)
correlator = (n_FD*correlators).sum(axis=-1)
##### plot spectra
plt.plot(np.arange(H.Ns),E/L,
                    marker='o',color='b',label='real space')
plt.plot(np.arange(Hblock.Ns),Eblock/L,
                    marker='x',color='r',markersize=2,label='momentum space')
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()
##### plot correlator
plt.plot(t,correlator,linewidth=2)
plt.xlabel('$t$',fontsize=16)
plt.ylabel('$C_{0,L/2}(t,\\beta)$',fontsize=16)
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)
plt.grid()
plt.tight_layout()
plt.show()
```

QuSpin *Example Code* 3: The Gross-Pitaevskii Equation and Nonlinear Time Evolution

```
from quspin.operators import hamiltonian # Hamiltonians and operators
from quspin.basis import boson_basis_1d # Hilbert space boson basis
from quspin.tools.measurements import evolve
import numpy as np # generic math functions
#import scipy.sparse as sp # sparse matrices library
import matplotlib.pyplot as plt # plot library
##### define model parameters #####
L=300 # system size
# calculate centre of chain
if L%2==0:
    j0 = L//2-0.5 # centre of chain
else:
    j0 = L//2 # centre of chain
sites=np.arange(L)-j0
# static parameters
J=1.0 # hopping
U=1.0 # Bose-Hubbard interaction strength
# dynamic parameters
omega_trap_i=0.001 # initial chemical potential
omega_trap_f=0.0001 # final chemical potential
t_ramp=40.0/J # set total ramp time
# ramp protocol
def ramp(t,omega_trap_i,omega_trap_f,t_ramp):
    return  (omega_trap_f - omega_trap_i)*t/t_ramp + omega_trap_i
# ramp protocol parameters
ramp_args=[omega_trap_i,omega_trap_f,t_ramp]
##### construct single-particle Hamiltonian #####
# define site-coupling lists
hopping=[[-J,i,(i+1)%L] for i in range(L-1)]
trap=[[0.5*(i-j0)**2,i] for i in range(L)]
# define static and dynamic lists
static=[["+-",hopping],["-+",hopping]]
dynamic=[['n',trap,ramp,ramp_args]]
# define basis
basis = boson_basis_1d(L,Nb=1,sps=2)
# build Hamiltonian
Hsp=hamiltonian(static,dynamic,basis=basis,dtype=np.float64)
E,V=Hsp.eigsh(time=0.0,k=1,which='SA')
##### imaginary-time evolution to compute GS of GPE #####
def GPE_imag_time(tau,phi,Hsp,U):
    """
    This function solves the real-valued GPE in imaginary time:
    $$ -\dot\phi(\tau) = Hsp(t=0)\phi(\tau) + U |\phi(\tau)|^2 \phi(\tau) $$
    """
    return -( Hsp.dot(phi,time=0) + U*np.abs(phi)**2*phi )
# define ODE parameters
GPE_params = (Hsp,U)
# define initial state to flow to GS from
phi0=V[:,0]*np.sqrt(L) # initial state normalised to 1 particle per site
# define imaginary time vector
tau=np.linspace(0.0,35.0,71)
```

```
# evolve state in imaginary time
psi_tau = evolve(phi0,tau[0],tau,GPE_imag_time,f_params=GPE_params,
                            imag_time=True,real=True,iterate=True)
# display state evolution
for i,psi0 in enumerate(psi_tau):
    # compute energy
    E_GS=(Hsp.matrix_ele(psi0,psi0,time=0) + 0.5*U*np.sum(np.abs(psi0)**4) ).real
    # plot wave function
    plt.plot(sites, abs(phi0)**2, color='r',marker='s',alpha=0.2,
                                        label='$|\\phi_j(0)|^2$')
    plt.plot(sites, abs(psi0)**2, color='b',marker='o',
                                label='$|\\phi_j(\\tau)|^2$' )
    plt.xlabel('$\\mathrm{lattice\\ sites}$',fontsize=14)
    plt.title('$J\\tau=%0.2f,\\ E_\\mathrm{GS}(\\tau)=%0.4fJ$'%(tau[i],E_GS)
                                                              ,fontsize=14)
    plt.ylim([-0.01,max(abs(phi0)**2)+0.01])
    plt.legend(fontsize=14)
    plt.draw() # draw frame
    plt.pause(0.005) # pause frame
    plt.clf() # clear figure
plt.close()
##### real-time evolution of GPE #####
def GPE(time,psi):
    """
    This function solves the complex-valued time-dependent GPE:
    $$ i\dot\psi(t) = Hsp(t)\psi(t) + U |\psi(t)|^2 \psi(t) $$
    """
    # solve static part of GPE
    psi_dot = Hsp.static.dot(psi) + U*np.abs(psi)**2*psi
    # solve dynamic part of GPE
    for Hd,f,f_args in Hsp.dynamic:
        psi_dot += f(time,*f_args)*Hd.dot(psi)
    return -1j*psi_dot
# define real time vector
t=np.linspace(0.0,t_ramp,101)
# time-evolve state according to GPE
psi_t = evolve(psi0,t[0],t,GPE,iterate=True,atol=1E-12,rtol=1E-12)
# display state evolution
for i,psi in enumerate(psi_t):
    # compute energy
    E=(Hsp.matrix_ele(psi,psi,time=t[i]) + 0.5*U*np.sum(np.abs(psi)**4) ).real
    # compute trap
    omega_trap=ramp(t[i],omega_trap_i,omega_trap_f,t_ramp)*(sites)**2
    # plot wave function
    plt.plot(sites, abs(psi0)**2, color='r',marker='s',alpha=0.2
                                ,label='$|\\psi_{\\mathrm{GS},j}|^2$')
    plt.plot(sites, abs(psi)**2, color='b',marker='o',label='$|\\psi_j(t)|^2$')
    plt.plot(sites, omega_trap,'--',color='g',label='$\\mathrm{trap}$')
    plt.ylim([-0.01,max(abs(psi0)**2)+0.01])
    plt.xlabel('$\\mathrm{lattice\\ sites}$',fontsize=14)
    plt.title('$Jt=%0.2f,\\ E(t)-E_\\mathrm{GS}=%0.4fJ$'%(t[i],E-E_GS),fontsize=14)
    plt.legend(loc='upper right',fontsize=14)
    plt.draw() # draw frame
```

```
    plt.pause(0.00005) # pause frame
    plt.clf() # clear figure
plt.close()
```

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.