$\mathcal{QuSpiN}$: a Python Package for Dynamics and Exact Diagonalisation of Quantum Many Body Systems
part I: spin chains

Phillip Weinberg* and Marin Bukov

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

September 13, 2016

Abstract

We present a new open-source Python package for quantum dynamics of spin chains based on exact diagonalisation, called $\mathcal{QuSpiN}$. The package is well suited to study quantum quenches at finite and infinite times, the Eigenstate Thermalisation hypothesis, many-body localisation and other dynamical phase transitions, periodically-driven (Floquet) systems, adiabatic and counter-diabatic ramps, spin-photon interactions, etc. Moreover, $\mathcal{QuSpiN}$'s user-friendly interface can easily be used in combination with other Python packages which makes it highly amenable to customisation. We explain how to use $\mathcal{QuSpiN}$ using three detailed examples: (i) adiabatic ramping of parameters in the many-body localised Heisenberg model, (ii) heating in the periodically-driven transverse-field Ising model in a parallel field, and (iii) quantised light-atom interactions: recovering the periodically-driven atom in the semi-classical limit of a static Hamiltonian.

Contents

What Problems can I Solve with $\mathcal{Q}^{\text{u}}\mathcal{S}\rho\text{¡}\mathcal{N}$?

copy-check the examples codes

During the last several decades, interacting quantum systems have been the primary focus of research in the condensed matter community. Prominent examples include (high-T_c) superconductors, superfluids, spin liquids, fractional topological insulators, and various other strongly-correlated systems. With respect to their mathematical description, these systems can be divided into two categories: *integrable* systems typically possess an extensive number of local conserved quantities (integrals of motion) and, therefore, their properties can be described in a closed, exact mathematical form with the help of ingenious techniques, such as the celebrated Bethe ansatz, the Jordan-Wigner transformation, transfer matrices, conformal field theory, etc. The analytical understanding of *non-integrable* systems, on the other hand, is usually feasible only in limited parameter regimes, and relies predominantly on the development of sophisticated approximation schemes based on perturbation theory; notable examples include Landau's Fermi liquid theory, Bogoliubov's theory of superfluidity, the BCS theory of superconductivity, numerous other mean-field theories, the Renormalisation group, the Schrieffer-Wolff transformation, and may more.

It is widely accepted that generic quantum systems in dimensions $d > 1$ are non-integrable and, thus, their theoretical study poses a formidable challenge. To tackle these hard problems, various highly-sophisticated numerical techniques, of different applicability regimes, have been developed: Quantum Monte Carlo methods, Density Matrix Renormalisation Group, Matrix Product States, Dynamical Mean-Field Theory waht else?, just to name a few. Although all of these methods are known to suffer from some sort of limitations of their own, when considered as a whole they allow one to successfully investigate the low-energy physics of many condensed matter systems. The Algorithms and Libraries for Physics Simulations (ALPS), the C++ library ITensor, as well as the Quantum Toolbox in Python (QuTiP) provide open source, freely accessible packages which contribute to widespread the use of such numerical techniques among the physics community.

Often times, when it comes to the study of dynamical problems, e.g. dynamical phase transitions (e.g. many-body localisation), thermalising time evolution, long-time dynamics, what else?, most numerical methods originally developed to study low-energy physics fail and, thus, become unreliable. At the same time, in the dawn of scientific computing, a numerical technique based on linear algebra was developed, robust enough, to safely deal with such studies – Exact Diagonalisation (ED). Although ED can also be applied to study low-energy phenomena, it is largely outperformed by the aforementioned much more sophisticated numerical algorithms. This comes mainly due to the limited computational resources available as of today, which imposes an upper limit on the Hilbert space size of the quantum model under consideration. In turn, for many-body systems, this typically sets an upper bound on the manageable system sizes, and gives rise to (sometimes unwanted and challenging to

analyse) finite-size effects. However, when it comes to the study of dynamical problems at long times, ED still represents a cutting-edge investigation technique for 'driven' quantum many-body problems.

In this paper, we report on a newly developed optimised Python package for dynamics and Exact Diagonalisation of quantum many-body spin systems in 1 dimension, which we called $\mathcal{QuSpinNet}$. The computation of the quantum operator matrices is done through costum code written by the authors in Cython. Cython is an optimizing static compiler which takes code written with a syntax similar Python and compiles it into efficient C/C++ code which easily interfaces with Python and can run orders of magnitude faster then a pure Python code. The quantum operators are stored as memory efficient SciPy sparse matrices. This allows $\mathcal{QuSpinNet}$ to easily iterface with mature Python packages NumPy and SciPy as well as other Python packages. These packages provide reliable state-of-the-art tools for scientific computation as well as support from the Python community to improve and update these Python packages. Moreover, we have included specific functionality in $\mathcal{QuSpinNet}$ which uses NumPy and SciPy to do all of the standard ED calculations one mught want to do while making sure the user only has to make use of a few NumPy or SciPy functions directly. Last but not least, $\mathcal{QuSpinNet}$ has been designed such that the amount of code needed do an ED calculations is very short (typically less than 200 lines). This greatly reducing the amount of time to start a new ED study as well as allowing users with little to no programming experience to do state of the art ED calculations.

Let us describe in more detail some of the features that make $\mathcal{QuSpinNet}$ interesting and useful and which, we believe, can serve a countless number of different studies.

- A major representative feature of $\mathcal{QuSpinNet}$ is the construction of spin Hamiltonians containing arbitrary (possibly non-local in space) many-body operators. One example is the four-spin operator $\mathcal{O} = \sum_j \sigma^z_j \sigma^+_{j+1} \sigma^-_{j+2} \sigma^z_{j+3} + \text{h.c.}$. Such multi-spin operators are often times used to study engineered integrable systems designed to feature particular topological properties [CITE], like ???. They are also generated by the nested commutators typically appearing in higher-order terms of perturbative expansions, such as the Schrieffer-Wolff transformation[CITE] and the inverse-frequency expansion[CITE].

- Another important feature is the availability to use symmetries which, if present in a given model, induce conservation laws leading to selection rules between the many-body states. As a result, the Hilbert space reduces to a tensor product of the Hilbert spaces corresponding to the underlying symmetry blocks. Consequently, the presence of symmetries effectively reduces the relevant Hilbert space dimension which, in turn, allows one to study larger systems. Currently, $\mathcal{QuSpinNet}$ supports the following spin chain symmetries:

  - total magnetisation
  - parity (i.e. reflection w.r.t. the middle of the chain)
  - spin inversion (on the entire chain but also individually for sublattices A and B)
  - the joint application of parity and spin inversion (present e.g. when studying staggered or linear external potentials)
  - translation symmetry
  - all physically meaningful combinations of the above

As we shall see in Sec. 2, constructing Hamiltonians with given symmetries is done by specifying the desired symmetry block. By default $\mathcal{Q}^{\mathrm{u}}\mathcal{S}\rho\mathrm{¡}\mathcal{N}$ performs a check on the compatibility of the requested symmetry with the Hamiltonian: if the user tries to specify a symmetry which is not present in their model, it will raise an error message[1].

- As alluded to above, as of present date ED methods represent the most reliable way to safely study quantum dynamics in a generic way. In this respect, with $\mathcal{Q}^{\mathrm{u}}\mathcal{S}\rho\mathrm{¡}\mathcal{N}$ the user can build arbitrary time-dependent Hamiltonians. The package contains built-in routines to calculate the real (or imaginary) time evolution of any quantum state under a user-defined time-dependent Hamiltonian based on scipy's integration tool for ordinary differential equations.

- Besides spin chains, $\mathcal{Q}^{\mathrm{u}}\mathcal{S}\rho\mathrm{¡}\mathcal{N}$ also allows the user to couple an arbitrary spin chain to a single photon mode (i.e. quantum harmonic oscillator). In this case, the total magnetisation symmetry is replaced by the combined total photon and spin number conservation.

We close this brief introduction by mentioning a short list of 'hot' topics that can be successfully studied with the help of $\mathcal{Q}^{\mathrm{u}}\mathcal{S}\rho\mathrm{¡}\mathcal{N}$:

* quantum quenches and quantum dynamics at finite and infinite times
* adiabatic and counter-diabatic ramps
* periodically driven (Floquet) systems
* many-body localisation, Eigenstate Thermalisation hypothesis
* quantum information
* quantised photon-spin interactions and similar cavity QED related models
* dynamical phase transitions and critical phenomena
* machine learning with quantum many-body systems

This list is far from being complete, but it can serve as a useful guideline to the interested user. Let us now illustrate in detail how to use $\mathcal{Q}^{\mathrm{u}}\mathcal{S}\rho\mathrm{¡}\mathcal{N}$, addressing three exciting problems, which we believe cover a wide range of interesting topics, to exemplify some of the most common $\mathcal{Q}^{\mathrm{u}}\mathcal{S}\rho\mathrm{¡}\mathcal{N}$ tools.

How do I use $\mathcal{Q}^{\mathrm{u}}\mathcal{S}\rho\mathrm{¡}\mathcal{N}$?

One of the main advantages of $\mathcal{Q}^{\mathrm{u}}\mathcal{S}\rho\mathrm{¡}\mathcal{N}$ is its user-friendly interface. To demonstrate how the package works, we shall guide the reader step by step through a short Python code, explaining all details of a proper use of the package. In case the reader is unfamiliar with Python, we kindly invite them to accept the challenge of learning the Python basics, while enjoying the study of quantum many-body dynamics.

[1] This automatic check can also be disabled, see the documentation under Sec. C.

Installing $\mathcal{Q}^{\mathrm{u}}\mathcal{S}\rho_{\mathrm{i}}\mathcal{N}$ is quick and efficient; just follow the steps outlined in App. A.

Below, we stick to the following general guidelines: first, we define the problem containing the physical quantities of interest and show their behaviour in a few figures. After that, we present the $\mathcal{Q}^{\mathrm{u}}\mathcal{S}\rho_{\mathrm{i}}\mathcal{N}$ code used to generate them, broken up into its building blocks. We explain each step in great detail. The complete contiguous code, including the lines used to generate the figures shown below, is available in App. B. It is not our purpose in this paper to discuss in detail the interesting underlying physics of these systems; instead, we focus on setting up the Python code to study them with the help of $\mathcal{Q}^{\mathrm{u}}\mathcal{S}\rho_{\mathrm{i}}\mathcal{N}$.

Adiabatic Control of Parameters in Many-Body Localised Phases

Physics Setup—Strongly disordered many-body models have recently enjoyed a lot of attention in the theoretical condensed matter community. It has been shown that, beyond a critical disorder strength, these models undergo a dynamical phase transition from an delocalised ergodic (thermalising) phase to a many-body localised (MBL), i.e. non-conducting, non-thermalising phase, in which the system does not obey the Eigenstate Thermalisation hypothesis.

In our first $\mathcal{Q}^{\mathrm{u}}\mathcal{S}\rho_{\mathrm{i}}\mathcal{N}$ example, we show how one can study the adiabatic control of model parameters in many-body localised phases. It was recently argued that the adiabatic theorem does not apply to disordered systems [CITE]. On the other hand, controlling the system parameters in MBL phases is of crucial experimental[CITE] significance. Thus, the question as to whether there exists an adiabatic window for some, possibly intermediate, ramp speeds (as is the case for periodically-driven systems[CITE]), is of particular and increasing importance.

Let us consider the Heisenberg XXZ open chain in a disordered z-field with the Hamiltonian

$$H(t) = \sum_{j=0}^{L-2} \frac{J_{xy}}{2} \left(S^+_{j+1} S^-_j + \text{h.c.} \right) + J_{zz}(t) S^z_{j+1} S^z_j + \sum_{j=0}^{L-1} h_j S^z_j, \tag{1}$$

where J_{xy} is the spin-spin interaction in the xy-plane, disorder is modelled by a uniformly distributed random field $h_j \in [-h_0, h_0]$ of strength h_0 along the z-direction, and the spin-spin interaction along the z-direction, $J_{zz}(t)$ is the adiabatically-modulated parameter. Note that we enumerate the L sites of the chain by $j = 0, 1, \ldots, L-1$ to conform with Python's array indexing convention. It has been demonstrated that this model exhibits a transition to an MBL phase at the critical disorder strength $h_0^* = ???|$. In particular, for $h_0 = h_{MBL} = 3.9$ the system is in a many-body localised phase, while for $h_0 = h_{ETH} = 0.1$ the system is in the ergodic (ETH) delocalised phase. We now choose the ramp protocol $J_{zz}(t) = (1/2 + vt) J_{zz}(0)$ with the ramp speed v, and evolve the system with the Hamiltonian $H(t)$ from $t_i = 0$ to[2] $t_f = (2v)^{-1}$. We choose the initial state $|\psi_i\rangle = |\psi(t_i)\rangle$ from the middle of the spectrum of $H(t_i)$ to ensure typicality; more specifically we choose $|\psi_i\rangle$ to be that eigenstate of $H(t_i)$ whose energy is closest to the rum of middle of the spectrum of $H(t_i)$, where the density of states, and thus the thermodynamic entropy, is largest.

To measure whether the system can adiabatically follow the ramp, we use two different indicators: (i) we evolve the state up to time t_f and project it onto the eigenstates of $H(t_f)$.

[2] Notice that $t_f \to \infty$ as $v \to 0$ and thus, the total evolution time increases with decreasing the ramp speed v.

The corresponding diagonal entropy

$$S_d = -\mathrm{tr}\left[\rho_d \log \rho_d\right], \qquad \rho_d = \sum_n |\langle n|\psi(t_f)\rangle|^2 |n\rangle\langle n| \tag{2}$$

in the basis $\{|n\rangle\}$ of $H(t_f)$ at small enough ramp speeds v is a measure of the delocalisation of the time-evolved state $\psi(t_f)\rangle$ onto the energy eigenstates of $H(t_f)$. If, for instance, after the ramp the system still occupies a single eigenstate $|\tilde{n}\rangle$, then $S_d = 0$.

The second measure of adiabaticity we use is (ii) the entanglement entropy

$$S_{\mathrm{ent}}(t_f) = -\frac{1}{|\mathrm{A}|}\mathrm{tr}_{\mathrm{A}}\left[\rho_{\mathrm{A}}(t_f)\log\rho_{\mathrm{A}}(t_f)\right], \qquad \rho_{\mathrm{A}}(t_f) = \mathrm{tr}_{\mathrm{A}^c}|\psi(t_f)\rangle\langle\psi(t_f)| \tag{3}$$

of subsystem A, defined to contain the left half of the chain and $|\mathrm{A}| = L/2$. We denoted the reduced density matrix of subsystem A by ρ_{A}, and A^c is the complement of A. Phil, do we see area vs volume law in Sent for ETH vs MBL? Note that Sent in the code comes out normalised by the volume of subsystem A, unless the 'densities=False' flag is passed.

Code Analysis—Let us now explain how one can study this problem numerically using $\mathcal{QuSpy}\mathcal{N}$. First, we load the required Python packages.

```
from qspin.operators import hamiltonian # Hamiltonians and operators
from qspin.basis import spin_basis_1d # Hilbert space spin basis
from qspin.tools.measurements import ent_entropy, diag_ensemble # entropies
from numpy.random import ranf,seed # pseudo random numbers
from joblib import delayed,Parallel # parallelisation
import numpy as np # generic math functions
from time import time # timing package
```

Since we want to produce many realisations of the date and average over disorder, we specify the simulations parameters: `n_real` is the number of disorder realisations, while `n_jobs` is the `joblib` parallelisation parameter which determines how many Python processes to run simultaneously.

```
##### define simulation parameters #####
n_real=10 # number of disorder realisations
n_jobs=1 # number of CPU cores used for parallelisation
```

Next, we define the physical model parameters. In doing so it is advisable to use the floating point when the coupling is meant to be a non-integer real number in order to avoid problems with division: for example, `1` is the integer 1 while `1.0` – the corresponding float. For instance, in Python `0.5` is *not* equal to `1/2`, but rather to `1.0/2.0`.

```
##### define model parameters #####
L=10 # system size
Jxy=0.5 # xy interaction
Jzz_0=1.0 # zz interaction
h_MBL=3.9 # MBL disorder strength
h_ETH=0.1 # delocalised disorder strength
vs=np.logspace(-2.0,0.0,num=20,base=10) # log-spaced vector of ramp speeds
```

The time-dependent disordered Hamiltonian consists of two parts: the time-dependent XXZ model which is disorder-free, and the disorder field whose values differ from one realisation to another. We focus on the XXZ part first. First, we define the driving protocol $J_{zz}(t)$, which we call `ramp`. As already explained, our goal is to obtain the disorder-averaged entropies as

a function of the ramp speed. Hence, for each disorder realisation, we need to evolve the initial state many times, each corresponding to a different ramp speed v. However, defining the Hamiltonian from scratch every single time is not particularly efficient from the point of view of simulation runtime. We thus want to set up a family of Hamiltonians $H_v(t)$ at once. This can be achieved by using the fact that, in Python,re-assigning an already declared variable does not change its memory address; instead it records the new value to the same old address.Marin: memory address or not, in Python, variables initialized outside of functions are always assumed to be global and can be accessed inside any local function, If we really want to emphisize this point we can declare `v` as a global variable in the ramp function, in this way you don't have to decalare `v` before the function. To use this, notice that the drive speed `v` is *not* a parameter of the function `ramp`, but is declared beforehand. Once, `ramp` has been defined, reassigning `v` dynamically induces a change of `ramp` without directly modifying the latter. We shall comment on how this works later on in the code.

```
##### set up Heisenberg Hamiltonian with linearly varying zz-interaction #####
# define linear ramp function
v = 1.0 # declare ramp speed variable
def ramp(t):
        return (0.5 + v*t)
ramp_args=[]
```

To set up any Hamiltonian, we need to calculate the basis of the Hilbert space it lives in. Since the Hamiltonian (1) conserves the total magnetisation, we can reach larger system sizes by working in a fixed magnetisation sector. A natural choice is the zero-magnetisation sector which contains the ground state. All symmetries in $\mathcal{QuSpy}\mathcal{N}$ can be declared when the `basis` is being created. As shown below, the `basis` class has one required argument `L` – the system size. The user can declare a symmetry with the help of optional arguments: for instance, `Nup=L/2` defines the zero-magnetisation sector. In general, the magnetisation symmetry is defined by specifying the number of up spins in the chain. In Sec. 2.2 we shall show how to use two other symmetries – translation and parity (reflection). Last, since we work with spin operators here, it is required to pass the flag `pauli=False`; failure to do so will result in a Hamiltonian defined in terms of the Pauli spin matrices.

```
# compute basis in the 0-total magnetisation sector (requires L even)
basis = spin_basis_1d(L,Nup=L/2,pauli=False)
```

Setting up the spin-spin operators goes as follows. First, we need to define the site-coupling lists `J_zz` and `J_xy`. To uniquely specify a two-spin interaction, we need (i) the coupling, and (ii) – the labels of the sites the two operators act on. $\mathcal{QuSpy}\mathcal{N}$ uses Python's indexing convention meaning that the first lattice site is always $i = 0$, and the last one – $i = L - 1$. For example, for the zz-interaction, the coupling is `1.0`, while the two sites are the nearest neighbours `i,i+1`. Hence, the tuple `[1.0,i,i+1]` defines the bond operator $1 \times S_i^\mu S_{i+1}^\mu$. To define the total interaction energy $\sum_{i=0}^{L-2} S_i^\mu S_{i+1}^\mu$, all we need is to loop over the $L-2$ bonds of the open chain[3]. In the same way one can define boundary or single-site operators.

```
# define operators with OBC using site-coupling lists
J_zz = [[Jzz_0,i,i+1] for i in range(L-1)] # OBC
J_xy = [[Jxy,i,i+1] for i in range(L-1)] # OBC
```

[3] The Python expression `range(L-1)` produces all integers between `0` and `L-2` including.

The above lines of code specify the coupling but not yet which spin operators are being coupled. To do this, we need to create a `static` and/or `dynamic` operator list. As the name suggests, static lists define time-independent operators. Given the site-coupling list `J_xy` from above, it is easy to set the operator $1/2\sum_{i=0}^{L-2} S_i^+ S_{i+1}^-$ by specifying the spin operator type in the same order as the site indices appear in the corresponding site-coupling list: `[["+-",J_xy]]`. Similarly, one can set up the hermitian conjugate term $1/2\sum_{i=0}^{L-2} S_i^- S_{i+1}^+$ as `[["-+",J_xy]]`. In the end, one can concatenate these operator lists to produce the static part of the Hamiltonian.

```
# static and dynamic lists
static = [["+-",J_xy],["-+",J_xy]]
```

The time-dependent part of the Hamiltonian is defined using `dynamic` lists. Similar to their static counterparts, one needs to define an operator string, say `"zz"` to declare the specific operator our of a site-coupling list. Apart from the site-coupling list `J_zz`, however, a dynamic list also requires a time-dependent function and its arguments. If one desires to define a time-independent Hamiltonian, then one should set an empty dynamic list, `dynamic=[]`.In the linearly driven XXZ-Hamiltonian we are setting up here, the function arguments `ramp_args` is an empty list.

```
dynamic =[["zz",J_zz,ramp,ramp_args]]
```

Once the static and dynamic lists are set up, building up the corresponding Hamiltonian is a one-liner. In $\mathcal{QuSpi}\mathcal{N}$, this is done using the `hamiltonian` class. The first required argument is the `static` list, while the second one – the `dynamic` list. These two arguments necessarily must appear in this order. Another required argument is the `basis`, which carries the necessary info about symmetries. Yet whether a given Hamiltonian has these symmetries or not, depends on the operators defined in the static and dynamic lists. The `hamiltonian` class performs an automatic check on the Hamiltonian for hermiticity and the presence of magnetisation conservations and other symmetries. These checks are turned on by default, but can be disabled using the optional arguments `check_herm=False`, `check_pcon=False`, and `check_symm=False`.

```
# compute the time-dependent Heisenberg Hamiltonian
H_XXZ = hamiltonian(static,dynamic,basis=basis,dtype=np.float64)
```

Since the problem involves disorder, we will be generating multiple disorder realisations. In this case, it is recommended to reset the pseudo-random generator before any random numbers have been drawn.

```
##### calculate diagonal and entanglement entropies #####
def realization(vs,H_XXZ,basis,real):
```

To produce the entropies vs. ramp speed data over many disorder realisations, we define the function `realization` which returns a two-element NumPy array, `np.array([S_d,Sent])` with the values of the diagonal entropy S_d in the first element, and the values of the entanglement entropy S_{ent} – in the second element. The first argument is the vector of ramp speeds, `vs`, required for the dynamics. The second argument is the time-dependent XXZ Hamiltonian `H_XXZ` to which we shall add a uniformly drawn z-field for each disorder realisation. The third argument is the spin `basis` which is required to calculate the S_{ent}. The fourth and last argument is the number of realisations, which is only used to print a message about the duration of the single realisation in the end of `realization`.

```
41      This function computes the entropies for a single disorder realisation.
42      --- arguments ---
43      vs: vector of ramp speeds
44      H_XXZ: time-dep. Heisenberg Hamiltonian with driven zz-interactions
45      basis: spin_basis_1d object containing the spin basis
46      n_real: number of disorder realisations; used only for timing
47      """
48      seed() # for parallel jobs the random number needs to be seeded for each
        parallel process
49      global v, h_MBL, h_ETH # define global variables
50      ti = time() # start timer
```

Since we want to call the variables **v**, **h_MBL** and **h_ETH** inside **realization**, we declare them **global**. While **h_MBL** and **h_ETH** can also be parsed as additional arguments of **realization**, this cannot be done with the variable **v** because this would create a new local variable with its own new memory address, while we need the original address in order to change the drive function and thus the Hamiltonian dynamically (we shall see exactly how this works in a moment).

```
51      #
```

In order to time each realisation, we use the package **time**:

```
52      # draw random field uniformly from [-1.0,1.0] for each lattice site
```

Recall that the full disordered Hamiltonian of the problem is $H(t) = H_{\mathrm{XXZ}}(t) + \sum_j h_j S^z_j$. The random field h_j differs from one realisation to another. Hence, it has to be defined inside the **realisation** function. Recall that we want to compare the localised with the delocalised regimes, corresponding to the disordere strengths h_{MBL} and h_{ETH}, respectively. To this end, we first, for each lattice site i, draw a random number **unscaled_fields[i]** uniformly in the interval $[-1, 1]$, and store it in the variable **unscaled_fields**, see **line 55** below. Building the external z-field proceeds in exactly the same way as before: (i) we calculate the site-coupling list, **line 57**, (ii) we designate that the operator is along the z-axis by defining a static operator list, **line 59**, and (iii) we use the already computed spin basis to construct the operator matrix with the **hamiltonian** class, **lines 62**. Notice how we disabled the default checks on hermiticity, magnetisation (particle number) conservation, and symmetries using the auxiliary dictionary **no_checks** passed straight to **hamiltonian** as keyword arguments. Last, in **lines 64-65**, we define the MBL and ETH time-dependent Hamiltonians.

```
54      # define z-field operator site-coupling list
55      h_z=[[unscaled_fields[i],i] for i in range(basis.L)]
56      # static list
57      disorder_field = [["z",h_z]]
58      # compute disordered z-field Hamiltonian
59      no_checks={"check_herm":False,"check_pcon":False,"check_symm":False}
60      Hz=hamiltonian(disorder_field,[],basis=basis,dtype=np.float64,**no_checks)
61      # compute the MBL and ETH Hamiltonians for the same disorder realisation
62      H_MBL=H_XXZ+h_MBL*Hz
63      H_ETH=H_XXZ+h_ETH*Hz
64      #
65      ### ramp in MBL phase ###
```

Let us first focus on the MBL phase. First, we have to define the initial state. We want it to be as close as possible to an infinite-temperature state within the given symmetry sector. To this

end, we can first calculate the minimum and maximum energy, **Emin** and **Emax** of the spectrum of $H_{MBL}(t=0)$. This is achieved using the **hamiltonian** attribute method **eigsh**, see **line 69-70**, which is a wrapper for **scipy.sparse.linalg.eigsh**, cf. App. C. Then, by taking the average we obtain a number, **E_inf_temp**, which is as represents the infinite-temperature energy up to finite-size effects, **line 71**. The initial state **psi_0** is then that eigenstate of $H_{MBL}(t=0)$, whose energy is closest to **E_inf_temp**, cf. **lines 73-74**.

```
67    eigsh_args={"k":2,"which":"BE","maxiter":1E10,"return_eigenvectors":False}
68    Emin,Emax=H_MBL.eigsh(time=0.0,**eigsh_args)
69    E_inf_temp=(Emax+Emin)/2.0
70    # calculate nearest eigenstate to energy at infinite temperature
71    E,psi_0=H_MBL.eigsh(time=0.0,k=1,sigma=E_inf_temp,maxiter=1E10)
72    psi_0=psi_0.reshape((-1,))
73    # calculate the eigensystem of the final MBL hamiltonian
74    E_final,V_final=H_MBL.eigh(time=(0.5/vs[-1]))
```

The calculation of the diagonal entropy S_d requires the eigensystem of the Hamiltonian $H_{MBL}(t_f)$ at the end of the ramp $t_f = (2v_f)^{-2}$. The entire spectrum and the corresponding eigenstates are obtained using the **hamiltonian** method **eigh**. For time-dependent Hamiltonians, **eigh** accepts the argument **time** to specify the time variable. Unless explicitly specified, **time=0.0** by default.

```
75    # evolve states and calculate entropies in MBL phase
76    run_MBL=[_do_ramp(psi_0,H_MBL,basis,v,E_final,V_final) for v in vs]
```

To calculate the entropies for each ramp speed, we define the helper function **_do_ramp**, which first evolves the initial state according to the v-dependent Hamiltonian $H_{MBL}(t;v)$ for a fixed ramp speed v. In **line 78** we loop over the ramp speed greed **vs**. More importantly, however, the iteration index **v** carries the same name as the parameter in the drive function **ramp**. Thus, every time a new ramp speed is read off the vector **vs**, the parameter **v** changes its value. Because **v** is a global variable, this change induces a change into the function **ramp**, which in turn induces a change in the **dynamic** list. Thus, at the end of the day, a new member of the family of MBL Hamiltonians, $\{v : H_{MBL}(t;v)\}$, is picked and parsed to **_do_ramp** to do the time evolution with. Hence, we end up with a convenient and automatic way of generating the whole family $\{v : H_{MBL}(t;v)\}$, while having to calculate the operators in the Hamiltonian only once.

```
77    run_MBL=np.vstack(run_MBL).T
78    #
79    ###  ramp in ETH phase ###
```

It remains to discuss the helper function **_do_ramp**. It evolves the initial state **psi_0** with the **hamiltonian** object **H** and calculates the entropies at the end of the ramp. Given a ramp speed **v**, we first determine the total ramp time **t_f**. Evolving a quantum state under any Hamiltonian **H** is easily done with the **hamiltonian** method **evolve**, see **line 112**. **evolve** requires the initial state **psi_0**, the starting time – here **0.0**, and a vector of times to return the evolved state at, but since we are only interested in the state at the final time – we pass **t_f**. The **evolve** method has further interesting features which we discuss in Secs. 2.2 and 2.3.

```
109   # time-evolve state from time 0.0 to time t_f
110   psi = H.evolve(psi_0,0.0,t_f)
111   # calculate entanglement entropy
112   subsys = range(basis.L/2) # define subsystem
```

Once we have the state at the end of the ramp, we can obtain the entropies as follows. Calculating S_{ent} is done using the `measurements` function `ent_entropy` which we imported in `line 3`. It requires the quantum state (here the pure state `psi`), and the `basis` it is stored in[4]. Optionally, one can specify the site indices which define the subsystem of interest using the argument `chain_subsys`. Note that `ent_entropy` returns a dictionary in which the value of the entanglement entropy is stored under the key `"Sent"`. The function `ent_entropy` has a variety of interesting features, described in the documentation, see App. C.

```
113     Sent = ent_entropy(psi,basis,chain_subsys=subsys)["Sent"]
114     # calculate diagonal entropy in the basis of H(t_f)
115     S_d = diag_ensemble(basis.L,psi,E_final,V_final,Sd_Renyi=True)["Sd_pure"]
```

Similarly, there is a built-in function to calculate the diagonal entropy S_d of a state `psi` in a given basis (here `V_final`), called `diag_ensemble`. This function can calculate a variety of interesting quantities in the diagonal ensemble defined by the eigensystem arguments (here `E_final`, `V_final`). We again invite the interested reader to check out the documentation in App. C. This concludes the definition of `_do_ramp`.

```
116     #
117     return np.asarray([S_d,Sent])
```

Back to the function `realization`, we have already seen how to obtain the entropies in the MBL phase. We now do the same thing in the delocalised ETH phase. The only difference is that, before we start, we have to re-set the parameter `v` to unity, see `line 82`. This is required since the iteration over the ramp speeds `vs` in `line 78` changed dynamically not only the Hamiltonian `H_MBL` but also `H_ETH`. Apart from this subtleties, the code is the same as the MBL one.

```
81      # calculate the energy at infinite temperature for initial ETH hamiltonian
82      Emin,Emax=H_ETH.eigsh(time=0.0,**eigsh_args)
83      E_inf_temp=(Emax+Emin)/2.0
84      # calculate nearest eigenstate to energy at infinite temperature
85      E,psi_0=H_ETH.eigsh(time=0.0,k=1,sigma=E_inf_temp,maxiter=1E10)
86      psi_0=psi_0.reshape((-1,))
87      # calculate the eigensystem of the final ETH hamiltonian
88      E_final,V_final=H_ETH.eigh(time=(0.5/vs[-1]))
89      # evolve states and calculate entropies in ETH phase
90      run_ETH=[_do_ramp(psi_0,H_ETH,basis,v,E_final,V_final) for v in vs]
91      run_ETH=np.vstack(run_ETH).T # stack vertically elements of list run_ETH
92      # show time taken
93      print "realization {0}/{1} finished in {2:.2f} sec".format(real+1,n_real,time()-
        ti)
```

We can now display how long the single iteration took, and conclude the definition of `realization`.

```
94      return run_MBL,run_ETH
95  #
96  ##### evolve state and evaluate entropies #####
```

Now that we have written the `realization` function, we can call it `n_real` times to produce the data. The easiest way of doing this is to loop over the disorder realisation, as shown in `line 124`. However, a better way of doing this makes use of the `joblib` package which can distribute simultaneous function calls over `n_job` Python processes, see `line 126`. To learn

[4] The `basis` is required since the subsystem may not share the same symmetries as the entire chain.

more about how to do this, we invite the readers to check the documentation of the joblib package. Having produced and extracted the entropy vs. ramp speed data, we are ready to perform the disorder average by taking the mean over all realisations.

```
# alternative way without parallelisation
data = np.asarray([realization(vs,H_XXZ,basis,i) for i in xrange(n_real)])
"""
data = np.asarray(Parallel(n_jobs=n_jobs)(delayed(realization)(vs,H_XXZ,basis,i) for
    i in xrange(n_real)))
run_MBL,run_ETH = zip(*data) # extract MBL and data
# average over disorder
mean_MBL = np.mean(run_MBL,axis=0)
mean_ETH = np.mean(run_ETH,axis=0)
#
##### plot results #####
```

The complete code is available in Fig. 1.

Heating in Periodically Driven Spin Chains

Physics Setup—As a second example, we now show how one can easily study heating in the periodically-driven transverse-field Ising model with a parallel field[CITE David, Tomaz]. This model is non-integrable even without the time-dependent driving protocol. The time-periodic Hamiltonian is defined as a two-step protocol as follows:

$$\begin{aligned} H(t) &= \left\{ \begin{array}{ll} J\sum_{j=0}^{L-1}\sigma_j^z\sigma_{j+1}^z + h\sum_{j=0}^{L-1}\sigma^z, & t\in[-T/4,\ T/4] \\ g\sum_{j=0}^{L-1}\sigma_j^x, & t\in[\ T/4, 3T/4] \end{array} \right\} \text{mod } T, \\ &= \sum_{j=0}^{L-1}\frac{1}{2}\left(J\sigma_j^z\sigma_{j+1}^z + h\sigma^z + g\sigma_j^x\right) + \frac{1}{2}\text{sgn}\left[\cos\Omega t\right]\left(J\sigma_j^z\sigma_{j+1}^z + h\sigma^z - g\sigma_j^x\right). \end{aligned} \tag{4}$$

We assume that the spin chain is closed with a periodic boundary (i.e. a ring). The spin-spin interaction strength is denoted by J, the transverse field – by g, and the parallel field – by h. The period of the drive is T and, although the periodic step protocol contains infinitely many Fourier harmonics, we shall refer to $\Omega = 2\pi/T$ as *the* frequency of the drive.

Since the Hamiltonian is periodic, $H(t+T) = H(t)$, Floquet's theorem applies and postulates that the dynamics of the system at times lT, integer multiple of the driving period (a.k.a. stroboscopic times), is governed by the time-independent Floquet Hamiltonian[5] H_F. In other words, the evolution operator is stroboscopically given by

$$U(lT) = \mathcal{T}_t \exp\left(-i\int_0^{lT} H(t)\mathrm{d}t\right) = \exp(-ilTH_F). \tag{5}$$

While the Floquet Hamiltonian for this system cannot be calculated analytically, a suitable approximation can be found at high drive frequencies by means of the van Vleck inverse-frequency expansion[CITE]. However, this expansion is known to calculate the effective Floquet Hamiltonian H_{eff} in a different basis than the original stroboscopic one: $H_F = \exp[-iK_{\text{eff}}(0)]H_{\text{eff}}\exp[iK_{\text{eff}}(0)]$, which requires the additional calculation of the so-called Kick operator $K_{\text{eff}}(0)$ to 'rotate' to the original basis.

[5] One has to be careful when using the term 'Hamiltonian', as H_F need not be a local operator. In such cases there does not exist a static physically meaningful system described by H_F.

In the inverse-frequency expansion, we expand both $H_{\rm eff}$ and $K_{\rm eff}(0)$ in powers of the inverse frequency. Let us label these approximate objects by the superscript $^{(n)}$, suggesting that the corresponding object is of order $\mathcal{O}(\Omega^{-n})$:

$$\begin{aligned} H_F &= H_F^{(0)} + H_F^{(1)} + H_F^{(2)} + H_F^{(3)} + \mathcal{O}(\Omega^{-4}) = H_F^{(0+1+2+3)} + \mathcal{O}(\Omega^{-4}), \\ H_{\rm eff} &= H_{\rm eff}^{(0)} + H_{\rm eff}^{(1)} + H_{\rm eff}^{(2)} + H_{\rm eff}^{(3)} + \mathcal{O}(\Omega^{-4}), \\ K_{\rm eff} &= K_{\rm eff}^{(0)} + K_{\rm eff}^{(1)} + K_{\rm eff}^{(2)} + K_{\rm eff}^{(3)} + \mathcal{O}(\Omega^{-4}), \end{aligned}$$

Using the short-hand notation one can show that, for this problem, all odd-order terms in the van Vleck expansion vanish [see App. G of Ref.[CITE thesis]]

$$H_F^{(0+1+2+3)} = H_F^{(0+2)} \approx \mathrm{e}^{-iK_{\rm eff}^{(2)}(0)} \left(H_{\rm eff}^{(0)} + H_{\rm eff}^{(2)} \right) \mathrm{e}^{+iK_{\rm eff}^{(2)}(0)}, \tag{6}$$

while the first few even-order ones are given by

$$\begin{aligned} H_{\rm eff}^{(0)} &= \frac{1}{2} \sum_j J\sigma_j^z\sigma_{j+1}^z + h\sigma_j^z + g\sigma_j^x, \\ H_{\rm eff}^{(2)} &= -\frac{\pi^2}{12\Omega^2} \sum_j J^2 g \sigma_{j-1}^z \sigma_j^x \sigma_{j+1}^z + Jgh(\sigma_j^x\sigma_{j+1}^z + \sigma_j^z\sigma_{j+1}^x) + Jg^2(\sigma_j^y\sigma_{j+1}^y - \sigma_j^z\sigma_{j+1}^z) \\ &\qquad + \left(J^2 g + \frac{1}{2} h^2 g \right) \sigma_j^x + \frac{1}{2} h g^2 \sigma_j^z, \\ K_{\rm eff}^{(0)} &= \mathbf{0}, \\ K_{\rm eff}^{(2)}(0) &= -\frac{\pi^2}{8\Omega^2} \sum_j Jg \left(\sigma_j^z\sigma_{j+1}^y + \sigma_j^y\sigma_{j+1}^z \right) + hg\sigma_j^y, \end{aligned} \tag{7}$$

It was recently argued based on the aforementioned Floquet theorem[CITE] that, in a closed periodically driven system, stroboscopic dynamics is sufficient to completely quantify heating, and we shall make use of this fact in our little study here. We choose as the initial state the ground state of the approximate Hamiltonian $H_F^{(0+1+2+3)}$ and denote it by $|\psi_i\rangle$. Regimes of slow and fast heating can then be easily detected by looking at the energy density $\mathcal{E}$ absorbed by the system from the drive

$$\mathcal{E}(lT) = \frac{1}{L} \langle \psi_i | \mathrm{e}^{ilTH_F} H_F^{(0+1+2)} \mathrm{e}^{-ilTH_F} | \psi_i \rangle, \tag{8}$$

and the entanglement entropy of a subsystem. We call this subsystem A and define it to contain $L/2$ consecutive chain sites[6]:

$$S_{\rm ent}(lT) = -\frac{1}{L_{\rm A}} \mathrm{tr}_{\rm A} \left[\rho_{\rm A}(lT) \log \rho_{\rm A}(lT) \right], \quad \text{with} \quad \rho_{\rm A}(lT) = \mathrm{tr}_{\rm A^c} \left[\mathrm{e}^{-ilTH_F} |\psi_i\rangle\langle\psi_i| \mathrm{e}^{ilTH_F} \right], \tag{9}$$

where the partial trace in the definition of the reduced density matrix (DM) $\rho_{\rm A}$ is over the complement of A, denoted $\mathrm{A^c}$, and $L_{\rm A} = L/2$ denotes the length of subsystem A.

[6] Since we use periodic boundaries, it does not matter which consecutive sites we choose. In fact, in $\mathcal{Q}^{\rm u}\mathcal{S}\rho\mathcal{N}$ the user can choose any (possibly disconnected) subsystem to calculate the entanglement entropy and the reduced DM, see Sec. C.

Since heating can be exponentially slow at high frequencies[CITE], one might be interested in calculating also the infinite-time quantities

$$\overline{\mathcal{E}} = \lim_{N\to\infty}\frac{1}{N}\sum_{l=0}^{N}\mathcal{E}(lT), \qquad \overline{S}_{\rm rdm} = -\frac{1}{L_{\rm A}}{\rm tr}_{\rm A}\left[\overline{\rho}_{\rm A}\log\overline{\rho}_{\rm A}\right], \qquad S_d^F = -\frac{1}{L}{\rm tr}\left[\rho_d^F\log\rho_d^F\right], \quad (10)$$

where $\overline{\rho}_{\rm A}$ is the infinite-time reduced DM of subsystem A, and ρ_d^F is the DM of the Diagonal ensemble in the exact Floquet basis $\{|n_F\rangle\colon H_F|n_F\rangle = E_F|n_F\rangle\}$ [CITE TD review]:

$$\overline{\rho}_{\rm A} = \lim_{N\to\infty}\frac{1}{N}\sum_{l=0}^{N}\rho_{\rm A}(lT) = {\rm tr}_{\rm A^c}\left[\rho_d^F\right], \qquad \rho_d^F = \sum_n |\langle\psi_i|n_F\rangle|^2|n_F\rangle\langle n_F|$$

We note in passing that in general $\overline{S}_{\rm rdm} \neq \lim_{N\to\infty} N^{-1}\sum_{l=0}^{N} S_{\rm ent}(lT)$ due to interference terms, although the two may happen to be close.

In Fig.??? we show the time evolution of $\mathcal{E}(lT)$ and $S_{\rm ent}(lT)$ as a function of the number of driving cycle for a given drive frequency, together with their infinite-time values.

Code Analysis—Let us now discuss the $\mathcal{QUSpin}$ code for this problem in detail. First we load the required classes, methods and functions required for the computation:

```
from qspin.operators import hamiltonian # Hamiltonians and operators
from qspin.basis import spin_basis_1d # Hilbert space spin basis
from qspin.tools.measurements import obs_vs_time, diag_ensemble # t_dep measurements
from qspin.tools.Floquet import Floquet, Floquet_t_vec # Floquet Hamiltonian
import numpy as np # generic math functions
```

After that, we define the model parameters as

```
L=14 # system size
J=1.0 # spin interaction
g=0.809 # transverse field
h=0.9045 # parallel field
Omega=4.5 # drive frequency
```

The time-periodic step drive can easily be incorporated through the following function:

```
# define time-reversal symmetric periodic step drive
def drive(t,Omega):
    return np.sign(np.cos(Omega*t))
drive_args=[Omega]
```

Next, we define the basis, similar to the example in Sec. 2.1. One can convince oneself that the Hamiltonian in Eq. (4) possesses two symmetries at all times t which are, therefore, also inherited by the Floquet Hamiltonian. These are translation invariance and parity (i.e. reflection w.r.t. the centre of the chain). To incorporate them, one needs to specify the desired block for each symmetry: `kblock=int` selects the many-body states of total momentum $2\pi/L$*`int`, while `pblock`=±1 sets the parity sector. For all total momenta different from 0 and π, the translation operator does not commute with parity, in which case semi-momentum states resulting in a *real* Hamiltonian have been programmed [CITE Anders].

```
# compute basis in the 0-total momentum and +1-parity sector
basis=spin_basis_1d(L=L,kblock=0,pblock=1)
```

The definition of the operator lists proceeds similarly to the MBL example above. It is interesting to note how the periodic boundary condition is put in `line 25` using the modulo operator `%`. Compared to open boundaries, the PBC `J_nn` list now also has a total of `L` elements, as many as there are sites and bonds on the ring.

```
21 # define PBC site-coupling lists for operators
22 x_field_pos=[[+g,i] for i in range(L)]
23 x_field_neg=[[-g,i] for i in range(L)]
24 z_field=[[h,i]      for i in range(L)]
25 J_nn=[[J,i,(i+1)%L] for i in range(L)] # PBC
```

To program the full Hamiltonian $H(t)$, we use the second line of Eq. (4). The time-independent part is defined using the static operator list. For the time-dependent part, we need to pass the function `drive` and its arguments `drive_args`, defined in `line 15`, to all operators the drive couples to. In fact, $\mathcal{QuSpi}\mathcal{N}$ is smart enough to automatically sum up all operators multiplied by the same time-dependent function in any dynamic list created. Note that since we are dealing with a Hamiltonian defined by Pauli matrices and not the spin-1/2 operators, we drop the optional argument `pauli` for the `hamiltonian` class, since by default it is set to `pauli=True`.

```
26 # static and dynamic lists
27 static=[["zz",J_nn],["z",z_field],["x",x_field_pos]]
28 dynamic=[["zz",J_nn,drive,drive_args],
29          ["z",z_field,drive,drive_args],["x",x_field_neg,drive,drive_args]]
30 # compute Hamiltonians
31 H=0.5*hamiltonian(static,dynamic,dtype=np.float64,basis=basis)
```

The following lines define the approximate van Vleck Floquet Hamiltonian, cf. Eq. (7). Of particular interest is `line 37` where we define the operator list for the three-spin operator `"zxz"`. Apart from the coupling `J**2*g`, we now need to specify the *three* site indices `i,(i+1)%L,(i+2)%L` for each of the operators `"zxz"`, respectively. In a similar fashion, one can define any multi-spin operator.

```
33 ##### set up second-order van Vleck Floquet Hamiltonian #####
34 # zeroth-order term
35 Heff_0=0.5*hamiltonian(static,[],dtype=np.float64,basis=basis)
36 # second-order term: site-coupling lists
37 Heff2_term_1=[[+J**2*g,i,(i+1)%L,(i+2)%L] for i in range(L)] # PBC
38 Heff2_term_2=[[+J*g*h, i,(i+1)%L] for i in range(L)] # PBC
39 Heff2_term_3=[[-J*g**2,i,(i+1)%L] for i in range(L)] # PBC
40 Heff2_term_4=[[+J**2*g+0.5*h**2*g,i] for i in range(L)]
41 Heff2_term_5=[[0.5*h*g**2,        i] for i in range(L)]
42 # define static list
43 Heff_static=[["zxz",Heff2_term_1],
44              ["xz",Heff2_term_2],["zx",Heff2_term_2],
45              ["yy",Heff2_term_3],["zz",Heff2_term_2],
46              ["x",Heff2_term_4],
47              ["z",Heff2_term_5]                      ]
48 # compute van Vleck Hamiltonian
49 Heff_2=hamiltonian(Heff_static,[],dtype=np.float64,basis=basis)
50 Heff_2*=-np.pi**2/(12.0*Omega**2)
51 # zeroth + second order van Vleck Floquet Hamiltonian
52 Heff_02=Heff_0+Heff_2
```

In order to rotate the state from the van Vleck to the stroboscopic (Floquet-Magnus) picture, we also have to calculate the kick operator at time $t = 0$. While the procedure is the same as above, note that $K_{\text{eff}}(0)$ has imaginary matrix elements, whence the variable `dtype=np.complex128` is used (in fact this is the default `dtype` optional argument that the `hamiltonian` class assumes if one does not pass this argument explicitly). If the user tries to force define a real-valued Hamiltonian which, however, has complex matrix elements, $\mathcal{QuSpiN}$ will raise an error.

```
##### set up second-order van Vleck Kick operator #####
Keff2_term_1=[[J*g,i,(i+1)%L] for i in range(L)] # PBC
Keff2_term_2=[[h*g,i] for i in range(L)]
# define static list
Keff_static=[["zy",Keff2_term_1],["yz",Keff2_term_1],["y",Keff2_term_2]]
Keff_02=hamiltonian(Keff_static,[],dtype=np.complex128,basis=basis)
Keff_02*=-np.pi**2/(8.0*Omega**2)
```

Next, we need to find $H_F^{(0+2)} = \exp[-iK_{\text{eff}}^{(2)}(0)]H_{\text{eff}}^{(0+2)}\exp[iK_{\text{eff}}^{(2)}(0)]$. To this end, we make use of the `hamiltonian` class method `rotate_by` which conveniently provides a function for this purpose. In general, it uses a generator B to define a linear transformation 'rotate' a `hamiltonian` object A via $\exp(aB)A\exp(a^*B^\dagger)$ So where exactly is the † in `rotate_by`? for any complex-valued number a. To rotate using a generator, we set `generate=True`. Although we do not use it directly here, it is also useful for the user to become familiar with the `exp_op` class which provides the matrix exponential. This class has a variety of useful methods and we invite the user to check out the complete documentation, cf. App. C. For instance, $\exp(zB)A$ can be obtained using `exp_op(B,a=z).dot(A)`, while $A\exp(zB)$ is `A.dot(exp_op(B,a=z)`[7].

```
##### rotate Heff to stroboscopic basis #####
# e^{-1j*Keff_02} Heff_02 e^{+1j*Keff_02}
HF_02 = Heff_02.rotate_by(Keff_02,generator=True,a=-1j)
```

Now that we have concluded the initialisation of the approximate Floquet Hamiltonian, it is time to discuss how to study the dynamics of the system. We start by defining a vector of times `t`, particularly suitable for the study of periodically driven systems. We initialise this time vector as an object of the `Floquet_t_vec` class. The arguments we need are the drive frequency `Omega`, the number of periods (here `100`), and the number of time points per period `len_T` (here set to `1`). Once initialised, `t` has many useful attributes, such as the time values `t.vals`, the drive period `t.T`, the stroboscopic times `t.strobo.vals`, or their indices `t.strobo.inds`. The `Floquet_t_vec` class has further useful properties, described in the documentation in App. C.

```
##### define time vector of stroboscopic times with 100 cycles #####
t=Floquet_t_vec(Omega,100,len_T=1) # t.vals=times, t.i=init. time, t.T=drive period
```

To calculate the exact stroboscopic Floquet Hamiltonian H_F, one can conveniently make use of the `Floquet` class. Currently, it supports three different ways of obtaining the Floquet Hamiltonian: (i) passing an arbitrary time-periodic `hamiltonian` object it will evolve each basis eigenstate for one period to obtain the evolution operator $U(T)$. This calculation can be parallelised using the Python module `joblib`, activated by setting the optional argument `n_jobs`. (ii) one can pass a list of static `hamiltonian` objects, accompanied by a list of time steps to apply each of these Hamiltonians for. In this case, the `Floquet` class will make use

[7] One can also use the syntax `A.rdot(exp_op(a*B))` and `exp_op(z*B).rdot(A)`, respectively.

of the matrix exponential to find $U(T)$. Instead, here we choose, (iii), to use a single dynamic **hamiltonian** object $H(t)$, accompanied a list of times $\{t_i\}$ to evaluate it at, and a list of time steps $\{\delta t_i\}$ to compute the time-ordered matrix exponential as $\prod_i \exp(-iH(t_i)\delta t_i)$. The **Floquet** class calculates the quasienergies **EF** folded in the interval $[-\Omega/2, \Omega/2]$ by default. If required, the user may further request the set of Floquet states by setting **VF=True**, the Floquet Hamiltonian, **HF=True**, and Floquet phases – **thetaF=True**.

```
##### calculate exact Floquet eigensystem #####
t_list=np.array([0.0,t.T/4.0,3.0*t.T/4.0])+np.finfo(float).eps # times to evaluate H
dt_list=np.array([t.T/4.0,t.T/2.0,t.T/4.0]) # time step durations to apply H for
Floq=Floquet({'H':H,'t_list':t_list,'dt_list':dt_list},VF=True) # call Floquet class
VF=Floq.VF # read off Floquet states
EF=Floq.EF # read off quasienergies
```

As discussed in the main text, we choose for the initial state the ground state[8] of the approximate Hamiltonian $H_F^{(0+2)}$. Here, we demonstrate how to fully diagonalise a **hamiltonian** object using the function **eigh**. Note that to find only the ground state it is, in fact, much more efficient to use the sparse matrix Lanczos-based function **eigsh**, as in the MBL example.

```
##### calculate initial state (GS of HF_02) and its energy
EF_02, VF_02 = HF_02.eigh()
EF_02, psi_i = EF_02[0], VF_02[:,0]
```

Finally, we can calculate the time-dependence of the energy density $\mathcal{E}(t)$ and the entanglement entropy density $S_{\text{ent}}(t)$. This is done using the **measurements** function **obs_vs_time**. If one evolves with a constant Hamiltonian (which is effectively the case for stroboscopic time evolution), $\mathcal{Q}^{\text{u}}\mathcal{S}\rho\mathcal{N}$ offers one two different options, that we now show. (i) As a first required argument of **obs_vs_time** one passes a tuple **(psi_i,E,V)** with the initial state, the spectrum and the eigenbasis of the Hamiltonian to do the evolution with. The second argument is the time vector (here **t.vals**), and the third one – the operator one would like to measure (here the approximate energy density **1.0/L*HF_02**. If the observable is time-dependent, **obs_vs_time** will evaluate it at the appropriate times: $\langle\psi(t)|\mathcal{O}(t)|\psi(t)\rangle$. To obtain the entanglement entropy, **obs_vs_time** calls the **measurements** function **ent_entropy**, whose arguments are passed using the variable **Sent_args**. **ent_entropy** requires to pass the **basis**, and optionally – the subsystem **chain_subsys** which would otherwise be set to the first **L/2** sites of the chain. To learn more about how to obtain the reduced density matrix or other features of **ent_entropy**, consult the documentation, App. C.

```
##### time-dependent measurements
# calculate measurements
Sent_args = {"basis":basis,"chain_subsys":[j for j in range(L/2)]}
meas = obs_vs_time((psi_i,EF,VF),t.vals,[HF_02/L],Sent_args=Sent_args)
```

The other way to calculate a time-dependent observable (ii) is more generic and works for arbitrary time-dependent Hamiltonians. It makes use of Schrödinger evolution to find the time-dependent state using the **evolve** method of the **hamiltonian** class. This method typically works for larger system sizes, than the ones that allow full ED. One can then simply pass the generator **psi_t** into **obs_vs_time** instead of the initial tuple.

```
# alternative way by solving Schroedinger's eqn
```

[8] The approximate Floquet Hamiltonian is unfolded[CITE FAPT review] and, thus, the ground state is well-defined.

```
psi_t = H.evolve(psi_i,t.i,t.vals,iterate=True,rtol=1E-9,atol=1E-9)
meas = obs_vs_time(psi_t,t.vals,[HF_02/L],Sent_args=Sent_args)
```

The output of `obs_vs_time` is a dictionary. Extracting the energy density and entanglement entropy density values as a function of time, is as easy as:

```
# read off measurements
Energy_t = meas["Expt_time"]
Entropy_t = meas["Sent_time"]["Sent"]
```

Last, we compute the infinite-time values of the energy density, and the entropy of the infinite-time reduced density matrix, as well as the diagonal entropy. They are, in fact, closely related to the expectation values of the Diagonal Ensemble of the initial state in the Floquet basis. The `measurements` tool contains the function `diag_ensemble` specifically designed for this purpose. The required arguments are the system size `L`, the initial state `psi_i`, as well as the Floquet spectrum `EF` and states `VF`. The optional arguments are packed in the auxiliary dictionary `DE_args`, and contain the observable `Obs`, the diagonal entropy `Sd_Renyi`, and the entanglement entropy of the reduced DM `Srdm_Renyi` with its arguments `Srdm_args`. The additional label `_Renyi` is used since in general one can also compute the Renyi entropy with parameter α, if desired. The function `diag_ensemble` will automatically return the densities of the requested quantities, unless the flag `densities=False` is specified. It has more features which allow one to calculate the temporal and quantum fluctuations of an observable in the diagonal ensemble, and return the diagonal density matrix. Moreover, it can do additional averages of all diagonal ensemble quantities over a user-specified energy distribution, which may prove useful in calculating thermal expectations at infinite times, cf. App. C.

```
##### calculate diagonal ensemble measurements
DE_args = {"Obs":HF_02,"Sd_Renyi":True,"Srdm_Renyi":True,"Srdm_args":Sent_args}
DE = diag_ensemble(L,psi_i,EF,VF,**DE_args)
Ed = DE["Obs_pure"]
Sd = DE["Sd_pure"]
Srdm=DE["Srdm_pure"]
```

The complete code is available in Fig. 2.

Quantised Light-Atom Interactions in the Semi-classical Limit: Recovering the Periodically Driven Atom

Physics Setup—The last example we show deals with the quantisation of the (monochromatic) electromagnetic (EM) field. For the purpose of our little study, we take a two-level atom (i.e. a single-site spin chain) and couple it to a single photon mode (i.e. a quantum harmonic oscillator). The Hamiltonian reads

$$H = \Omega a^\dagger a + \frac{A}{2}\frac{1}{\sqrt{N_{\rm ph}}}\left(a^\dagger + a\right)\sigma^x + \Delta\sigma^z, \tag{11}$$

where the operator $a^\dagger/a$ creates/destroys a photon in the mode, and the atom is modelled by a two-level system described by the Pauli spin operators $\sigma^{x,y,z}$. The photon frequency is Ω, $N_{\rm ph}$ is the average number of photons in the mode, A – the coupling between the EM field $E = \sqrt{N_{\rm ph}^{-1}}\left(a^\dagger + a\right)$ and the dipole operator σ^x, and Δ measures the energy difference between the two atomic states.

An interesting question to ask is under what conditions the atom can be described[9] by the time-periodic semi-classical Hamiltonian:

$$H_{\rm sc}(t) = A\cos\Omega t\ \sigma^x + \Delta\sigma^z. \tag{12}$$

Curiously, despite its simple form, one cannot solve in a closed form for the dynamics generated by the semi-classical Hamiltonian $H_{\rm sc}(t)$.

To address the above question, we prepare the system such that the atom is in its ground state, while we put the photon mode in a coherent state with mean number of photons $N_{\rm ph}$, as required to by the semi-classical regime[CITE Haroche]:

$$|\psi_i\rangle = |{\rm coh}(N_{\rm ph})\rangle|\downarrow\rangle. \tag{13}$$

We then calculate the exact dynamics generated by the spin-photon Hamiltonian H, measure the Pauli spin matrix σ^z which represents the energy of the atom, σ^x – the dipole operator, and the photon number $n = a^\dagger a$:

$$\langle\mathcal{O}\rangle = \langle\psi_i|{\rm e}^{itH}\mathcal{O}\ {\rm e}^{-itH}|\psi_i\rangle, \qquad \mathcal{O} = n, \sigma^z, \sigma^y, \tag{14}$$

and compare these to the semi-classical expectation values

$$\langle\mathcal{O}\rangle_{\rm sc} = \langle\downarrow|\mathcal{T}_t{\rm e}^{i\int_0^t H_{\rm sc}(t')dt'}\mathcal{O}\ \mathcal{T}_t{\rm e}^{-i\int_0^t H_{\rm sc}(t')dt'}|\downarrow\rangle, \qquad \mathcal{O} = \sigma^z, \sigma^y. \tag{15}$$

Figure ??? a shows a comparison between the quantum and the semi-classical time evolution of all observables $\mathcal{O}$ as defined above.

Code Analysis—We used the following compact $\mathcal{QS}\rho\mathcal{N}$ code to produce these results. First we load the required classes, methods and functions to do the calculation:

```
from qspin.basis import spin_basis_1d,photon_basis # Hilbert space bases
from qspin.operators import hamiltonian # Hamiltonian and observables
from qspin.tools.measurements import obs_vs_time # t_dep measurements
from qspin.tools.Floquet import Floquet,Floquet_t_vec # Floquet Hamiltonian
from qspin.basis.photon import coherent_state # HO coherent state
import numpy as np # generic math functions
```

Next, we define the model parameters as follows:

```
##### define model parameters #####
Nph_tot=60 # total number of photon states
Nph=Nph_tot/2 # mean number of photons in initial coherent state
Omega=3.5 # drive frequency
A=0.8 # spin-photon coupling strength (drive amplitude)
```

To set up the spin-photon Hamiltonian, we first build the operator lists. The **ph_energy** list does not require the specification of a lattice site index, since the latter is not defined for the photon sector. The **at_energy** list, on the other hand, requires the input of the lattice site for the σ^z operator: since we consider a single two-level system or, equivalently – a single-site chain, this index is **0**. The spin-photon coupling lists **absorb** and **emit** also require the site index which refers to the corresponding Pauli matrices: in this model – **0** again due to dimensional constraints.

[9] Strictly speaking the Hamiltonian $H_{\rm sc}(t)$ describes the spin dynamics in the rotating frame, defined by $a \to ae^{-i\Omega t}$; however, all three observables of interest: $a^\dagger a$, and $\sigma^{y,z}$ are invariant under this transformation.

```
# define operator site-coupling lists
ph_energy=[[Omega]]
at_energy=[[Delta,0]]
absorb=[[A/(2.0*np.sqrt(Nph)),0]]
emit=[[A/(2.0*np.sqrt(Nph)),0]]
```

To build the static operator list, we use the `|` symbol in the operator string to distinguish the spin and photon operators: spin operators always come to the left of the `|`-symbol, while photon operators – to the right. For convenience, the identity operator `I` can be omitted, such that `I|n` is the same as `|n`, and `z|I` is equivalent to `z|`, respectively. The dynamic list is empty since the spin-photon Hamiltonian is time-independent.

```
# define static and dynamics lists
static=[["|n",ph_energy],["x|-",absorb],["x|+",emit],["z|",at_energy]]
dynamic=[]
```

To build the spin-photon basis, we call the function `photon_basis` and use `spin_basis_1d` as the first argument. We need to specify the number of spin lattice sites, and the total number of harmonic oscillator states. Building the Hamiltonian works as before.

```
# compute atom-photon basis
basis=photon_basis(spin_basis_1d,L=1,Nph=Nph_tot)
# compute atom-photon Hamiltonian H
H=hamiltonian(static,dynamic,dtype=np.float64,basis=basis)
```

We now define the time-periodic semi-classical Hamiltonian which is defined on the spin Hilbert space only; thus we use a `spin_basis_1d` basis object. Note the existence of a non-empty dynamic list to define the time-dependence.

```
# define operators
dipole_op=[[A,0]]
# define periodic drive and its parameters
def drive(t,Omega):
    return np.cos(Omega*t)
drive_args=[Omega]
# define semi-classical static and dynamic lists
static_sc=[["z",at_energy]]
dynamic_sc=[["x",dipole_op,drive,drive_args]]
# compute semi-classical basis
basis_sc=spin_basis_1d(L=1)
# compute semi-classical Hamiltonian H_{sc}(t)
H_sc=hamiltonian(static_sc,dynamic_sc,dtype=np.float64,basis=basis_sc)
```

Next, we define the initial state as a product state, see Eq. (13). Notice that in the $\mathcal{Q}^{\mathrm{u}}\mathcal{S}\rho\mathcal{N}$ `spin_basis_1d` basis convention the state $|\downarrow\rangle = (1,0)^t$. This is because the spin basis states are coded using their bit representations and the state of all spins pointing down is assigned the integer `0`. To define the oscillator coherent state with mean photon number N_{ph}, we use the function `coherent_state`: its first argument is the eigenvalue of the annihilation operator a, while the second argument is the total number of oscillator states[10].

```
# define atom ground state
psi_at_i=np.array([1.0,0.0]) # spin-down eigenstate of \sigma^z
```

[10] Since the oscillator ground state is denoted by $|0\rangle$, the state $|N_{\mathrm{ph}}\rangle$ is the $(N_{\mathrm{ph}}+1)$–st state of the oscillator basis.

```
47 # define photon coherent state with mean photon number Nph
48 psi_ph_i=coherent_state(np.sqrt(Nph),Nph_tot+1)
49 # compute atom-photon initial state as a tensor product
50 psi_i=np.kron(psi_at_i,psi_ph_i)
```

The next step is to define a vector of stroboscopic times, using the class **Floquet_t_vec**. Unlike in Sec. 2.2, here we are also interested in the non-stroboscopic times in between the perfect periods lT. Thus, we omit the optional argument **len_T** since by default it is set to **len_T=100**, meaning that there are now 100 time points within each period. The initial time can be conveniently called by **t.i**, while the drive period – by **t.T** making use of the attributes of the **Floquet_t_vec** class. For more information on **Floquet_t_vec**, the user is advised to consult the package documentation, see App. C.

```
53 # define time vector over 30 driving cycles with 100 points per period
54 t=Floquet_t_vec(Omega,30) # t.i = initial time, t.T = driving period
```

We now time evolve the initial state both in the atom-photon, and the semi-classical cases using the **hamiltonian** class method **evolve**, as before. Here we define the solution **psi_t** as a generator expression.

```
53 # evolve atom-photon state with Hamiltonian H
54 psi_t=H.evolve(psi_i,t.i,t.vals,iterate=True,rtol=1E-9,atol=1E-9)
55 # evolve atom GS with semi-classical Hamiltonian H_sc
56 psi_sc_t=H_sc.evolve(psi_at_i,t.i,t.vals,iterate=True,rtol=1E-9,atol=1E-9)
```

Last, we define the observables of interest, using the **hamiltonian** class with unit coupling constants. since each observable represents a single operator, we refrain from defining operator lists and set the observables in-line. Note that the main difference below (apart from the **|** notation) in defining the Pauli operators in the atom-photon and the semi-classical cases is the basis argument. The Python dictionaries **obs_args** and **obs_args_sc** represent another way of passing optional keyword arguments to the hamiltonian function. Here we also show how one can disable the automatic symmetry and hermiticity checks.

```
61 # define observables parameters
62 obs_args={"basis":basis,"check_herm":False,"check_symm":False}
63 obs_args_sc={"basis":basis_sc,"check_herm":False,"check_symm":False}
64 # in atom-photon Hilbert space
65 n=hamiltonian([["|n", [[1.0  ]] ]],[],dtype=np.float64,**obs_args)
66 sz=hamiltonian([["z|",[[1.0,0]] ]],[],dtype=np.float64,**obs_args)
67 sy=hamiltonian([["y|",  [[1.0,0]] ]],[],dtype=np.complex128,**obs_args)
68 # in the semi-classical Hilbert space
69 sz_sc=hamiltonian([["z",[[1.0,0]] ]],[],dtype=np.float64,**obs_args_sc)
70 sy_sc=hamiltonian([["y",[[1.0,0]] ]],[],dtype=np.complex128,**obs_args_sc)
```

Finally, we calculate the time-dependent expectation values using the **measurements** tool function **obs_vs_time**. Its arguments are the time-dependent state **psi_t**, the vector of times **t.vals**, and a tuple of all observables of interest. **obs_vs_time** returns a dictionary with all time-dependent expectations stored under the key **"Expt_time"**. They can be accessed by array slicing in the order in which the observables appear in the tuple argument, as shown in lines **75** and **78**, respectively.

```
73 # in atom-photon Hilbert space
74 Obs_t = obs_vs_time(psi_t,t.vals,(n,sz,sy))["Expt_time"]
75 O_n, O_sz, O_sy = Obs_t[:,0], Obs_t[:,1], Obs_t[:,2]
```

```
# in the semi-classical Hilbert space
Obs_sc_t = obs_vs_time(psi_sc_t,t.vals,(sz_sc,sy_sc))["Expt_time"]
O_sz_sc, O_sy_sc = Obs_sc_t[:,0], Obs_sc_t[:,1]
```

The complete code is available in Fig. 3.

Future Perspectives for $\mathcal{Q}^{\mathrm{u}}\mathcal{S}\rho\mathrm{¡}\mathcal{N}$

Fermions and Bosons to come; optimisation for hermitian operators.
2d
mention tensoring bases and refer to documentation

Although $\mathcal{Q}^{\mathrm{u}}\mathcal{S}\rho\mathrm{¡}\mathcal{N}$ passed all tests we could think of so far, there may still be some bugs lurking out there. Therefore, we would much appreciate it, if the users could report these by sending the relevant piece of their code to P. Weinberg. As a rule of thumb, bug reports are most useful when the script sent is as short as possible but contains the necessary annotations and comments so it can be followed and, at the same time, the Hamiltonian used is the simplest one which displays the bug.

Last but not least, we would like to mention that $\mathcal{Q}^{\mathrm{u}}\mathcal{S}\rho\mathrm{¡}\mathcal{N}$ can prove excellent for various teaching purposes. For instance it can be integrated into homework assignments of classes and tutorials. At the same time, the package provides a wonderful opportunity for undergrad and graduate students to gain intuition while working on a research project.

Acknowledgements

Let's also mention the SCC here – I like them and we've been using the cluster for testing. I think they need to report what papers have been produced using thh cluster so that they get more funding to update it. IF u're fine just copy the sentences from the FAPT paper.

Author contributions This is optional. If desired, contributions should be succinctly described in a single short paragraph, using author initials.

Funding information Authors are required to provide funding information, including relevant agencies and grant numbers with linked author's initials.

Installation Guide in a Few Steps

$\mathcal{Q}^{\mathrm{u}}\mathcal{S}\rho\mathrm{¡}\mathcal{N}$ is currently only being supported for Python 2.7 and so one must make sure to install this version of Python. The Authors recommend the use of Anaconda to install Python and manage your Python packages. It is free to download at , or for a lighter installation you can use miniconda which can be found at .

Mac OS X/Linux

To install Anaconda/miniconda all one has to do is execute the installation script with administrative prevlige. To do this open up the terminal and go to the folder conataining 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.

Installing $\mathcal{Q}^{u}\mathcal{S}\rho_{¡}\mathcal{N}$ via Anaconda Once you have Anaconda/miniconda installed all you have to do to install $\mathcal{Q}^{u}\mathcal{S}\rho_{¡}\mathcal{N}$ is to execute the following command into the terminal: conda install -c weinbe58 qspin. If asked to install new packages just say yes. To update the code just run this command periodically.

Installing $\mathcal{Q}^{u}\mathcal{S}\rho_{¡}\mathcal{N}$ manually Installing the package manually is not recomended unless the above method failed. Note that you must have NumPy, SciPy, and Joblib installed before installing $\mathcal{Q}^{u}\mathcal{S}\rho_{¡}\mathcal{N}$Once all the prerequisit packages are installed can then download the source code from github and then extract the code to whichever directory you please. 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. The install_file.txt can be used to remove the package by running: cat install_file.txt | xargs rm -rf.

Windows

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

Installing $\mathcal{Q}^{u}\mathcal{S}\rho_{¡}\mathcal{N}$ via Anaconda Once you have Anaconda/miniconda installed all you have to do to install $\mathcal{Q}^{u}\mathcal{S}\rho_{¡}\mathcal{N}$ is to execute the following command into the terminal: conda install -c weinbe58 qspin. If asked to install new packages just say yes. To update the code just run this command periodically.

Installing $\mathcal{Q}^{u}\mathcal{S}\rho_{¡}\mathcal{N}$ manually Installing the package manually is not recomended unless the above method failed. Note that you must have NumPy, SciPy, and Joblib installed before installing $\mathcal{Q}^{u}\mathcal{S}\rho_{¡}\mathcal{N}$Once all the prerequisit packages are installed can then download the source code from github and then extract the code to whichever directory you please. 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.

Complete Example Codes

$\mathcal{Q}^{u}\mathcal{S}\rho\mathcal{N}$ *Example Code* 1: Adiabatic Control of Parameters in MBL Phases

```
from qspin.operators import hamiltonian # Hamiltonians and operators
from qspin.basis import spin_basis_1d # Hilbert space spin basis
from qspin.tools.measurements import ent_entropy, diag_ensemble # entropies
from numpy.random import ranf,seed # pseudo random numbers
from joblib import delayed,Parallel # parallelisation
import numpy as np # generic math functions
from time import time # timing package
#
##### define simulation parameters #####
n_real=10 # number of disorder realisations
n_jobs=1 # number of CPU cores used for parallelisation
#
##### define model parameters #####
L=10 # system size
Jxy=0.5 # xy interaction
Jzz_0=1.0 # zz interaction
h_MBL=3.9 # MBL disorder strength
h_ETH=0.1 # delocalised disorder strength
vs=np.logspace(-2.0,0.0,num=20,base=10) # log-spaced vector of ramp speeds
#
##### set up Heisenberg Hamiltonian with linearly varying zz-interaction #####
# define linear ramp function
v = 1.0 # declare ramp speed variable
def ramp(t):
        return (0.5 + v*t)
ramp_args=[]
# compute basis in the 0-total magnetisation sector (requires L even)
basis = spin_basis_1d(L,Nup=L/2,pauli=False)
# define operators with OBC using site-coupling lists
J_zz = [[Jzz_0,i,i+1] for i in range(L-1)] # OBC
J_xy = [[Jxy,i,i+1] for i in range(L-1)] # OBC
# static and dynamic lists
static = [["+-",J_xy],["-+",J_xy]]
dynamic =[["zz",J_zz,ramp,ramp_args]]
# compute the time-dependent Heisenberg Hamiltonian
H_XXZ = hamiltonian(static,dynamic,basis=basis,dtype=np.float64)
#
##### calculate diagonal and entanglement entropies #####
def realization(vs,H_XXZ,basis,real):
    """
    This function computes the entropies for a single disorder realisation.
    --- arguments ---
    vs: vector of ramp speeds
    H_XXZ: time-dep. Heisenberg Hamiltonian with driven zz-interactions
    basis: spin_basis_1d object containing the spin basis
    n_real: number of disorder realisations; used only for timing
    """
    seed() # for parallel jobs the random number needs to be seeded for each
    parallel process
```

```
    global v, h_MBL, h_ETH # define global variables
    ti = time() # start timer
    #
    # draw random field uniformly from [-1.0,1.0] for each lattice site
    unscaled_fields=-1+2*ranf((basis.L,))
    # define z-field operator site-coupling list
    h_z=[[unscaled_fields[i],i] for i in range(basis.L)]
    # static list
    disorder_field = [["z",h_z]]
    # compute disordered z-field Hamiltonian
    no_checks={"check_herm":False,"check_pcon":False,"check_symm":False}
    Hz=hamiltonian(disorder_field,[],basis=basis,dtype=np.float64,**no_checks)
    # compute the MBL and ETH Hamiltonians for the same disorder realisation
    H_MBL=H_XXZ+h_MBL*Hz
    H_ETH=H_XXZ+h_ETH*Hz
    #
    ### ramp in MBL phase ###
    # calculate the energy at infinite temperature for initial MBL Hamiltonian
    eigsh_args={"k":2,"which":"BE","maxiter":1E10,"return_eigenvectors":False}
    Emin,Emax=H_MBL.eigsh(time=0.0,**eigsh_args)
    E_inf_temp=(Emax+Emin)/2.0
    # calculate nearest eigenstate to energy at infinite temperature
    E,psi_0=H_MBL.eigsh(time=0.0,k=1,sigma=E_inf_temp,maxiter=1E10)
    psi_0=psi_0.reshape((-1,))
    # calculate the eigensystem of the final MBL hamiltonian
    E_final,V_final=H_MBL.eigh(time=(0.5/vs[-1]))
    # evolve states and calculate entropies in MBL phase
    run_MBL=[_do_ramp(psi_0,H_MBL,basis,v,E_final,V_final) for v in vs]
    run_MBL=np.vstack(run_MBL).T
    #
    ###  ramp in ETH phase ###
    v=1.0 # reset ramp speed to unity
    # calculate the energy at infinite temperature for initial ETH hamiltonian
    Emin,Emax=H_ETH.eigsh(time=0.0,**eigsh_args)
    E_inf_temp=(Emax+Emin)/2.0
    # calculate nearest eigenstate to energy at infinite temperature
    E,psi_0=H_ETH.eigsh(time=0.0,k=1,sigma=E_inf_temp,maxiter=1E10)
    psi_0=psi_0.reshape((-1,))
    # calculate the eigensystem of the final ETH hamiltonian
    E_final,V_final=H_ETH.eigh(time=(0.5/vs[-1]))
    # evolve states and calculate entropies in ETH phase
    run_ETH=[_do_ramp(psi_0,H_ETH,basis,v,E_final,V_final) for v in vs]
    run_ETH=np.vstack(run_ETH).T # stack vertically elements of list run_ETH
    # show time taken
    print "realization {0}/{1} finished in {2:.2f} sec".format(real+1,n_real,time()-
    ti)
    return run_MBL,run_ETH
#
##### evolve state and evaluate entropies #####
def _do_ramp(psi_0,H,basis,v,E_final,V_final):
    """
    Auxiliary function to evolve the state and calculate the entropies after the
    ramp.
```

```
    --- arguments ---
    psi_0: initial state
    H: time-dependent Hamiltonian
    basis: spin_basis_1d object containing the spin basis (required for Sent)
    E_final, V_final: eigensystem of H(t_f) at the end of the ramp t_f=1/(2v)
    """
    # determine total ramp time
    t_f = 0.5/v
    # time-evolve state from time 0.0 to time t_f
    psi = H.evolve(psi_0,0.0,t_f)
    # calculate entanglement entropy
    subsys = range(basis.L/2) # define subsystem
    Sent = ent_entropy(psi,basis,chain_subsys=subsys)["Sent"]
    # calculate diagonal entropy in the basis of H(t_f)
    S_d = diag_ensemble(basis.L,psi,E_final,V_final,Sd_Renyi=True)["Sd_pure"]
    #
    return np.asarray([S_d,Sent])
#
##### produce data for n_real disorder realisations #####
"""
# alternative way without parallelisation
data = np.asarray([realization(vs,H_XXZ,basis,i) for i in xrange(n_real)])
"""
data = np.asarray(Parallel(n_jobs=n_jobs)(delayed(realization)(vs,H_XXZ,basis,i) for
     i in xrange(n_real)))
run_MBL,run_ETH = zip(*data) # extract MBL and data
# average over disorder
mean_MBL = np.mean(run_MBL,axis=0)
mean_ETH = np.mean(run_ETH,axis=0)
#
##### plot results #####
import matplotlib.pyplot as plt
### MBL plot ###
f, pltarr1 = plt.subplots(2,sharex=True) # define subplot panel
pltarr1[0].set_title("MBL phase") # set title
# subplot 1: diag enetropy vs ramp speed
pltarr1[0].plot(vs,mean_MBL[0],label="MBL",marker=".",color="blue") # plot data
pltarr1[0].set_ylabel("Diagonal Entropy") # label y-axis
pltarr1[0].set_xlabel("Velocity") # label x-axis
pltarr1[0].set_xscale("log") # set log scale on x-axis
pltarr1[0].grid(True,which='both') # plot grid
# subplot 2: entanglement entropy vs ramp speed
pltarr1[1].plot(vs,mean_MBL[1],marker=".",color="blue") # plot data
pltarr1[1].set_ylabel("Entanglement Entropy") # label y-axis
pltarr1[1].set_xlabel("Velocity") # label x-axis
pltarr1[1].set_xscale("log") # set log scale on x-axis
pltarr1[1].grid(True,which='both') # plot grid
#
### ETH plot ###
f, pltarr2 = plt.subplots(2,sharex=True) # define subplot panel
pltarr2[0].set_title("ETH phase") # set title
# subplot 1: diag enetropy vs ramp speed
pltarr2[0].plot(vs,mean_ETH[0],marker=".",color="green") # plot data
```

```
pltarr2[0].set_ylabel("Diagonal Entropy") # label y-axis
pltarr2[0].set_xlabel("Velocity") # label x-axis
pltarr2[0].set_xscale("log") # set log scale on x-axis
pltarr2[0].grid(True,which='both') # plot grid
# subplot 2: entanglement entropy vs ramp speed
pltarr2[1].plot(vs,mean_ETH[1],marker=".",color="green") # plot data
pltarr2[1].set_ylabel("Entanglement Entropy") # label y-axis
pltarr2[1].set_xlabel("Velocity") # label x-axis
pltarr2[1].set_xscale("log") # set log scale on x-axis
pltarr2[1].grid(True,which='both') # plot grid
#
plt.show() # show plots
```

$\mathcal{Q}^{u}\mathcal{S}\rho_{\dot{c}}\mathcal{N}$ *Example Code* 2: Heating in Periodically Driven Spin Chains

```
from qspin.operators import hamiltonian # Hamiltonians and operators
from qspin.basis import spin_basis_1d # Hilbert space spin basis
from qspin.tools.measurements import obs_vs_time, diag_ensemble # t_dep measurements
from qspin.tools.Floquet import Floquet, Floquet_t_vec # Floquet Hamiltonian
import numpy as np # generic math functions
#
##### define model parameters #####
L=14 # system size
J=1.0 # spin interaction
g=0.809 # transverse field
h=0.9045 # parallel field
Omega=4.5 # drive frequency
#
##### set up alternating Hamiltonians #####
# define time-reversal symmetric periodic step drive
def drive(t,Omega):
    return np.sign(np.cos(Omega*t))
drive_args=[Omega]
# compute basis in the 0-total momentum and +1-parity sector
basis=spin_basis_1d(L=L,kblock=0,pblock=1)
# define PBC site-coupling lists for operators
x_field_pos=[[+g,i] for i in range(L)]
x_field_neg=[[-g,i] for i in range(L)]
z_field=[[h,i]      for i in range(L)]
J_nn=[[J,i,(i+1)%L] for i in range(L)] # PBC
# static and dynamic lists
static=[["zz",J_nn],["z",z_field],["x",x_field_pos]]
dynamic=[["zz",J_nn,drive,drive_args],
         ["z",z_field,drive,drive_args],["x",x_field_neg,drive,drive_args]]
# compute Hamiltonians
H=0.5*hamiltonian(static,dynamic,dtype=np.float64,basis=basis)
#
##### set up second-order van Vleck Floquet Hamiltonian #####
# zeroth-order term
Heff_0=0.5*hamiltonian(static,[],dtype=np.float64,basis=basis)
# second-order term: site-coupling lists
Heff2_term_1=[[+J**2*g,i,(i+1)%L,(i+2)%L] for i in range(L)] # PBC
Heff2_term_2=[[+J*g*h, i,(i+1)%L] for i in range(L)] # PBC
Heff2_term_3=[[-J*g**2,i,(i+1)%L] for i in range(L)] # PBC
Heff2_term_4=[[+J**2*g+0.5*h**2*g,i] for i in range(L)]
Heff2_term_5=[[0.5*h*g**2,        i] for i in range(L)]
# define static list
Heff_static=[["zxz",Heff2_term_1],
             ["xz",Heff2_term_2],["zx",Heff2_term_2],
             ["yy",Heff2_term_3],["zz",Heff2_term_2],
             ["x",Heff2_term_4],
             ["z",Heff2_term_5]                      ]
# compute van Vleck Hamiltonian
Heff_2=hamiltonian(Heff_static,[],dtype=np.float64,basis=basis)
Heff_2*=-np.pi**2/(12.0*Omega**2)
# zeroth + second order van Vleck Floquet Hamiltonian
Heff_02=Heff_0+Heff_2
```

```
#
##### set up second-order van Vleck Kick operator #####
Keff2_term_1=[[J*g,i,(i+1)%L] for i in range(L)] # PBC
Keff2_term_2=[[h*g,i] for i in range(L)]
# define static list
Keff_static=[["zy",Keff2_term_1],["yz",Keff2_term_1],["y",Keff2_term_2]]
Keff_02=hamiltonian(Keff_static,[],dtype=np.complex128,basis=basis)
Keff_02*=-np.pi**2/(8.0*Omega**2)
#
##### rotate Heff to stroboscopic basis #####
# e^{-1j*Keff_02} Heff_02 e^{+1j*Keff_02}
HF_02 = Heff_02.rotate_by(Keff_02,generator=True,a=-1j)
#
##### define time vector of stroboscopic times with 100 cycles #####
t=Floquet_t_vec(Omega,100,len_T=1) # t.vals=times, t.i=init. time, t.T=drive period
#
##### calculate exact Floquet eigensystem #####
t_list=np.array([0.0,t.T/4.0,3.0*t.T/4.0])+np.finfo(float).eps # times to evaluate H
dt_list=np.array([t.T/4.0,t.T/2.0,t.T/4.0]) # time step durations to apply H for
Floq=Floquet({'H':H,'t_list':t_list,'dt_list':dt_list},VF=True) # call Floquet class
VF=Floq.VF # read off Floquet states
EF=Floq.EF # read off quasienergies
#
##### calculate initial state (GS of HF_02) and its energy
EF_02, VF_02 = HF_02.eigh()
EF_02, psi_i = EF_02[0], VF_02[:,0]
#
##### time-dependent measurements
# calculate measurements
Sent_args = {"basis":basis,"chain_subsys":[j for j in range(L/2)]}
meas = obs_vs_time((psi_i,EF,VF),t.vals,[HF_02/L],Sent_args=Sent_args)
"""
# alternative way by solving Schroedinger's eqn
psi_t = H.evolve(psi_i,t.i,t.vals,iterate=True,rtol=1E-9,atol=1E-9)
meas = obs_vs_time(psi_t,t.vals,[HF_02/L],Sent_args=Sent_args)
"""
# read off measurements
Energy_t = meas["Expt_time"]
Entropy_t = meas["Sent_time"]["Sent"]
#
##### calculate diagonal ensemble measurements
DE_args = {"Obs":HF_02,"Sd_Renyi":True,"Srdm_Renyi":True,"Srdm_args":Sent_args}
DE = diag_ensemble(L,psi_i,EF,VF,**DE_args)
Ed = DE["Obs_pure"]
Sd = DE["Sd_pure"]
Srdm=DE["Srdm_pure"]
#
##### plot results #####
import matplotlib.pyplot as plt
import pylab
# define legend labels
str_E_t = "$\\mathcal{E}(lT)$"
str_Sent_t = "$\\mathcal{S}_\mathrm{ent}(lT)$"
```

```
str_Ed = "$\\overline{\mathcal{E}}$"
str_Srdm = "$\\overline{\mathcal{S}}_\mathrm{rdm}$"
str_Sd = "$\\mathcal{S}_d^F$"
# plot time-dependent data
plt.plot(t.vals/t.T,Energy_t,"b-o",linewidth=1,label=str_E_t,markersize=3.0)
plt.plot(t.vals/t.T,Entropy_t,"r-s",linewidth=1,label=str_Sent_t,markersize=3.0)
# plot infinite-time data
plt.plot(t.vals/t.T,Ed*np.ones(t.vals.shape),"b--",linewidth=1,label=str_Ed)
plt.plot(t.vals/t.T,Srdm*np.ones(t.vals.shape),"r--",linewidth=1,label=str_Srdm)
plt.plot(t.vals/t.T,Sd*np.ones(t.vals.shape),"g--",linewidth=1,label=str_Sd)
# label axes
plt.xlabel("$\\#\ \\mathrm{periods}\\ l$",fontsize=18)
# set y axis limits
plt.ylim([-0.8,0.7])
# display legend
plt.legend(loc="lower right")
# update axis font size
plt.tick_params(labelsize=16)
# turn on grid
plt.grid(True)
# show plot
plt.show()
```

$\mathcal{Q}^{u}\mathcal{S}\rho\mathcal{N}$ *Example Code* 3: Quantised Light-Atom Interactions in the Semi-classical Limit

```
from qspin.basis import spin_basis_1d,photon_basis # Hilbert space bases
from qspin.operators import hamiltonian # Hamiltonian and observables
from qspin.tools.measurements import obs_vs_time # t_dep measurements
from qspin.tools.Floquet import Floquet,Floquet_t_vec # Floquet Hamiltonian
from qspin.basis.photon import coherent_state # HO coherent state
import numpy as np # generic math functions
#
##### define model parameters #####
Nph_tot=60 # total number of photon states
Nph=Nph_tot/2 # mean number of photons in initial coherent state
Omega=3.5 # drive frequency
A=0.8 # spin-photon coupling strength (drive amplitude)
Delta=1.0 # difference between atom energy levels
#
##### set up photon-atom Hamiltonian #####
# define operator site-coupling lists
ph_energy=[[Omega]]
at_energy=[[Delta,0]]
absorb=[[A/(2.0*np.sqrt(Nph)),0]]
emit=[[A/(2.0*np.sqrt(Nph)),0]]
# define static and dynamics lists
static=[["|n",ph_energy],["x|-",absorb],["x|+",emit],["z|",at_energy]]
dynamic=[]
# compute atom-photon basis
basis=photon_basis(spin_basis_1d,L=1,Nph=Nph_tot)
# compute atom-photon Hamiltonian H
H=hamiltonian(static,dynamic,dtype=np.float64,basis=basis)
#
##### set up semi-classical Hamiltonian #####
# define operators
dipole_op=[[A,0]]
# define periodic drive and its parameters
def drive(t,Omega):
    return np.cos(Omega*t)
drive_args=[Omega]
# define semi-classical static and dynamic lists
static_sc=[["z",at_energy]]
dynamic_sc=[["x",dipole_op,drive,drive_args]]
# compute semi-classical basis
basis_sc=spin_basis_1d(L=1)
# compute semi-classical Hamiltonian H_{sc}(t)
H_sc=hamiltonian(static_sc,dynamic_sc,dtype=np.float64,basis=basis_sc)
#
##### define initial state #####
# define atom ground state
psi_at_i=np.array([1.0,0.0]) # spin-down eigenstate of \sigma^z
# define photon coherent state with mean photon number Nph
psi_ph_i=coherent_state(np.sqrt(Nph),Nph_tot+1)
# compute atom-photon initial state as a tensor product
psi_i=np.kron(psi_at_i,psi_ph_i)
#
##### calculate time evolution #####
```

```
# define time vector over 30 driving cycles with 100 points per period
t=Floquet_t_vec(Omega,30) # t.i = initial time, t.T = driving period
# evolve atom-photon state with Hamiltonian H
psi_t=H.evolve(psi_i,t.i,t.vals,iterate=True,rtol=1E-9,atol=1E-9)
# evolve atom GS with semi-classical Hamiltonian H_sc
psi_sc_t=H_sc.evolve(psi_at_i,t.i,t.vals,iterate=True,rtol=1E-9,atol=1E-9)
#
##### define observables #####
# define observables parameters
obs_args={"basis":basis,"check_herm":False,"check_symm":False}
obs_args_sc={"basis":basis_sc,"check_herm":False,"check_symm":False}
# in atom-photon Hilbert space
n=hamiltonian([["|n", [[1.0  ]] ]],[],dtype=np.float64,**obs_args)
sz=hamiltonian([["z|",[[1.0,0]] ]],[],dtype=np.float64,**obs_args)
sy=hamiltonian([["y|",  [[1.0,0]] ]],[],dtype=np.complex128,**obs_args)
# in the semi-classical Hilbert space
sz_sc=hamiltonian([["z",[[1.0,0]] ]],[],dtype=np.float64,**obs_args_sc)
sy_sc=hamiltonian([["y",[[1.0,0]] ]],[],dtype=np.complex128,**obs_args_sc)
#
##### calculate expectation values #####
# in atom-photon Hilbert space
Obs_t = obs_vs_time(psi_t,t.vals,(n,sz,sy))["Expt_time"]
O_n, O_sz, O_sy = Obs_t[:,0], Obs_t[:,1], Obs_t[:,2]
# in the semi-classical Hilbert space
Obs_sc_t = obs_vs_time(psi_sc_t,t.vals,(sz_sc,sy_sc))["Expt_time"]
O_sz_sc, O_sy_sc = Obs_sc_t[:,0], Obs_sc_t[:,1]
##### plot results #####
import matplotlib.pyplot as plt
import pylab
# define legend labels
str_n = "$\\langle n\\rangle$"
str_z = "$\\langle\\sigma^z\\rangle$"
str_x = "$\\langle\\sigma^x\\rangle$"
# plot spin-photon data
plt.plot(t.vals/t.T,O_n/Nph,"k",linewidth=1,label=str_n)
plt.plot(t.vals/t.T,O_sz,"b",linewidth=1,label=str_z)
plt.plot(t.vals/t.T,O_sy,"r",linewidth=1,label=str_x)
# plot semi-classical data
plt.plot(t.vals/t.T,O_sz_sc,"b.",marker=".",markersize=1.8)
plt.plot(t.vals/t.T,O_sy_sc,"r.",marker=".",markersize=2.0)
# label axes
plt.xlabel("$t/T$",fontsize=18)
# set y axis limits
plt.ylim([-1.1,1.3])
# display legend
plt.legend(loc="upper left")
# update axis font size
plt.tick_params(labelsize=16)
# turn on grid
plt.grid(True)
# show plot
plt.show()
```

Package Documentation

The complete up-to-date documentation for the package is available under:

https://github.com/weinbe58/qspin

References