Exact diagonalization User Manual PNJPG

Juan Polo (joanpologomez@gmail.com)
Piero Naldesi (piero.naldesi@lpmmc.cnrs.fr)

LPMMC, Grenoble, France

https://github.com/JeanClaude87/BH_diagonalization

October 16, 2018

Exact diagonalization of interacting Bose gases

Contents

1 Quick User Guide

The PNJPG software package is a pure python code, consisting of two main scripts (*bose.py* and *Hamiltonian.py*) and several libraries (*function.py* - containing the main functions of the code. Being mainly a number-crunching code, it is currently designed to be ran directly from a terminal without a graphical user interface.

1.1 Installation

1.2 Running the code

2 Exact diagonalization principle

The first key element for implementing an Exact diagonalization algorithm is to find the best way to write the basis. In particular we need to index each state of the basis in a way that we can easily and distinctly identify each of the states of the basis. We will use the following table as an example:

Bose config.	**Configuration**	**Index**
(2,0,0)	(1,1,0,0)	0
(1,1,0)	(1,0,1,0)	1
(1,0,1)	(1,0,0,1)	2
(0,2,0)	(0,1,1,0)	3
(0,1,1)	(0,1,0,1)	4
(0,0,2)	(0,0,1,1)	5

Table 2.1: Table showing different representations of the states in the basis for $N = 2$ and $L = 3$.

Note that in Table **??** we use the following idea to write the states from the Bose-Hubbard configuration to the Configuration space of our code: (i) whenever we find a number of consecutive ones $n \geq 1$ in the list it means that there are n particles in that position, (ii) a zero means that we need to go to the next position. Thus, a $(1, 1, 0, 0)$ means two particles in the first position and 0 in the rest, whereas (0,1,0,1) means zero particles in the first position, one in the second, and 1 in the last.

However we have not still mention how to obtain the full bases. In order to do so more easily we will build all possible combinations of zeros and ones in the so called configuration space. Note that the dimension of our tupple will be $N + L - 1$.

```
def Base_prep(**args):

  ll = args.get("ll")
  nn = args.get("nn")

  base_bin = []
  base_num = []
  base_ind = []

# Max = int(sum([2**(ll+nn-2-x) for x in range(nn)]))
# TO_con_tab = [None] * Max

  for bits in itertools.combinations(range(nn+ll-1), nn):
    s = ['0'] * (nn+ll-1)
    for bit in bits:
      s[bit] = '1'
    bi = ''.join(s)
    base_bin.append(bi)

    bose = TO_bose_conf(s,ll)

    base_num.append(bose)

    in_bi=int(bi,2)
    base_ind.append(in_bi)

    #TO_con_tab[in_bi] = bi

  base_bose = np.asarray(base_num, dtype=np.int8)
```

```
  return base_bin, base_bose, base_ind
```

In this function we first we use "itertools.combinations(range(nn+ll-1),nn)" which gives all possible combinations with no repetitions in the following way: combinations(range(4),2) will give all possible combinations of the elements $(0,1,2,3)$ in chunks of 2, thus $(0,1)$, $(0,2)$, $(0,3)$, $(1,2)$, $(1,3)$, $(2,3)$. The second for puts one in the positions that are chosen in the previous intertools functions. Example $(0,1)$ will put a 1 in the position 0 and 1 of the list s of dimension $N+L-1$. Finally .join put the list together in a single set of numbers like 1100. Finally we create "base_bose" and "base_bin". Note that our states are already in base_bin form, while for base_bose we need to use the "TO_bose_conf" function. In this same function we also save the indexes using "in_bi=int(bi,2)" that basically finds the decimal number of the binary form of 1100.

Next step is to prepare the possible hoppings that can occur in our system to then be able to write the Hamiltonian with its corresponding nondiagonal terms. This is done in a single line calculation. The dimensions of the hopping are $N+L-2$ and for each one of this values we calculate how can we jump to other states in binary and then we write that in configuration form. So for example we get 1100, 0110 and 0011. This will be the "states" that we will use to perform an XOR and calculate the hopping. For instance from 1100 we apply 1100 and leds to the same state, but 1100 with 0110 leads to 1010 which is the action of $a^\dagger_{i+1}a_i$.

```
def Hop_prep(**args):

  X = args.get("ll")
  Y = args.get("nn")

  Hop_dim=X+Y-2

  return [TO_con(2**i+2**((i+1)%(X+Y-1)),X+Y-1) for i in range(Hop_dim)]
```

Finally we create the Hamiltonian, ideally in a sparse form. The function to create the Hamiltonian relies on evaluate_ham. In this function we start by taking each element of the base and the calculate the hopping with respect to that element. As it is an element of the base, we can calculate directly the effect of the interactions by easily applying TO_bose_conf in the state and then getting the onsite interaction for the population in each site. Note that in all this process we append the values and indexes in three vectors in such a way that we never save positions with zero weight in the Hamiltonian. Next we apply the hopping in each state for all the list of possible hoppings obtained before using the XOR operation (this calculation is done in decimal base). It is necessary to check that we have not increased or decreased the number of particles. We transform the state in binary form and calculate its positon in the base.

Note that the index is obtained by calculating the binary value in reverse form. We take the Hilbert dimensions and subtract the value of the Hilbert dimensions that we would have if we were in a corresponding state. As an example for state 0011 we calculate the Hilbert dimension of $n=2$, $L=5$ that is $D(N,M)=(N+M-1)!/(N!(M-1)!)$. So for the first zero its 3 and the second 0 in the state is gives dim 2. The total result is $6-(3-2)=5$ which is the index corresponding to state $0011=(0,0,2)$.

```
def get_index(state,**args):

  ll           = args.get("ll")
  nn           = args.get("nn")
  DIM_H        = args.get("DIM_H")
  tab_fact     = args.get("tab_fact")
  hilb_dim_tab = args.get("hilb_dim_tab")
```

```
  size   = int(nn+ll-1)
  r_par  = int(nn)    #remaining_particles
  r_sit  = int(ll)  #remaining_sites
  result = DIM_H

  for jj in range(size):
    action_i = int(state[jj]);

    if r_par==0:
      break
    if action_i == 0:
      #print(jj,r_par)

      result -= hilb_dim_tab[r_par-1][r_sit]
      #result -= binomial_table[r_par-1][r_sit]
      r_sit  -= 1;

    else:
      r_par -= 1;
      #print('else',jj,r_par)

  return DIM_H-result
```

We also need to calculate the action of the hopping, which is done in the usual way in the Bose configuration using the formula $a_1^\dagger a_0^\dagger |n,m\rangle = \sqrt{n(m+1)}|n-1,m+1\rangle$.

If periodic boundary conditions are considered ($BC = 0$) we need to calculate some extra hoppings

3 Results

4 Data workflow

4.1 Main scripts and flow-charts

4.2 File handling