

1D Spin Chain Part 2

Christina Lee

July 11, 2017

Category: Grad

Tags: Quantum, ED, Magnet

Prerequisites: Many-Body Quantum Mechanics, 1D Spin Chain Part 1

Check out part 1 for the theoretical background, or [1] for an extensive explanation. Today is all programming.

Remember to keep a reasonable number of spins, n . The number of states goes as 2^n , and the size of the Hamiltonian will go as $2^n \times 2^n$. A 10-spin chain will have over a million entries, before taking into account any memory reducing tricks.

Here's the details of the largest spin-chains that fit on my machine which has 16GB of RAM and a 3.2 GHz Intel i5 processor:

n	m_z	t (min)	Memory (Gb)
16	8	1.5	2
18	9	?	>16
18	8	32.1	15.5

I have included a file in this directory, ED.jl, that is just the necessary executable parts of this Jupyter notebook. For large n , I recommend running ED.jl.

Here, we input one parameter n , the number of spins in our chain.

The program automatically calculates the parameter $nstates$.

```
In [1]: n=4
        nstates=2^n
```

```
Out[1]: 16
```

Now, let's write out all of our possible states in the S^z basis.

```
In [2]: psi=collect(0:(nstates-1))
        for p in psi
            println(bin(p,n), ' ', p)
        end
```

```
0000 0
0001 1
0010 2
0011 3
```

```

0100 4
0101 5
0110 6
0111 7
1000 8
1001 9
1010 10
1011 11
1100 12
1101 13
1110 14
1111 15

```

As in Part 1, we will be using the powers of 2 to compute magnetization, and masks to flip spins. To not have to calculate them each time, we just store them in memory.

```

In [3]: powers2=collect(0:(n-1));
        powers2=2.^powers2;

        mask=[0;powers2]+[powers2;0];
        mask=[mask[2:end-1];[1+2^(n-1)]]
        for m in mask
            println(bin(m,n))
        end

```

```

0011
0110
1100
1001

```

In Part 1, I used the number of up-spins as a proxy for magnetization. Here, we need the actual magnetization, not a proxy. An up-spin is $+1/2$ and a down-spin is $-1/2$. We modify our magnetization by

$$m = \frac{1}{2} (n_{\uparrow} - n_{\downarrow}) = \frac{1}{2} (n_{\uparrow} - (n - n_{\uparrow})), \quad (0.1)$$

$$m = n_{\uparrow} - \frac{n}{2}. \quad (0.2)$$

```

In [4]: m=zeros(psi)
        for i in 1:nstates
            m[i]=sum((psi[i]&powers2)./(powers2))
        end
        m=m-n/2

```

```

Out[4]: 16-element Array{Float64,1}:
  -2.0
  -1.0
  -1.0
   0.0
  -1.0
   0.0
   0.0
   1.0
  -1.0
   0.0
   0.0
   1.0
   0.0
   1.0
   1.0
   2.0

```

Grouped by Magnetization

Now that we have the magnetizations corresponding to each state, we perform some trickery that allows us to reduce the difficulty of our problem dramatically.

Magnetization is a conserved quantity. By Noether's theorem, we know that the Hamiltonian is not going to mix states of different magnetizations. We only deal with one magnetization at a time, which is a much smaller problem.

```

In [5]: # The possible values for magnetization
ma=collect(0:1:n)-n/2

```

```

Out[5]: 5-element Array{Float64,1}:
  -2.0
  -1.0
   0.0
   1.0
   2.0

```

Now let's just pick out a single magnetization quantum number mz and only work with that matrix for the rest of the post.

```

In [6]: # The magnetic quantum number
mz=3

# An array of states with the correct magnetization
psi_mz=psi[m.==ma[mz]]

[psi_mz bin.(psi_mz,n) m[psi_mz+1]]

```

```

Out[6]: 6×3 Array{Any,2}:
 3 "0011" 0.0

```

```
5 "0101" 0.0
6 "0110" 0.0
9 "1001" 0.0
10 "1010" 0.0
12 "1100" 0.0
```

And now creating the matrix.
Stuff goes here

```
In [7]: dim=length(psi_mz)
```

```
M=ma[mz]*(ma[mz]+1)*eye(Float64,dim,dim)
#M=zeros(Float64,dim,dim); use this for XY model
```

Though we have significantly reduced the size of the matrix by restricting to one magnetization, we no longer have our states in 1,2,3,4... order. Their position in an array no longer corresponds to their value. Therefore, we need a function to determine their index once we know their value.

We can find the index of the flipped state multiple different ways, but the simplest is by the Midpoint method. We split the interval in half, and see if the value we are looking for is higher or lower than the middle point. Then we get a new interval.

```
In [8]: function findstate(state::Int,set::Array{Int})
```

```
    #Lower bound of interval
    imin=1
    #Upper bound of interval
    imax=length(set)

    # checking if the lower bound is what we are looking for
    if set[imin] == state
        return imin
    end
    # checking if the upper bound is what we are looking for
    if set[imax] == state
        return imax
    end

    # Initializing variables
    # looking to see if we've found it yet
    found=false
    # how many times we've gone around the while loop
    count=0

    while found==false && count < length(set)

        count+=1
        tester=floor(Int,imin+(imax-imin)/2)
```

```

        if state < set[tester]
            imax=tester-1
        elseif state > set[tester]
            imin=tester+1
        else
            found=true
            return tester
        end
    end
end

if found == false
    println("findstate never found a match")
    println("Are you sure the state is in that Array?")
end

return 0
end

```

Out[8]: findstate (generic function with 1 method)

Now time to generate the matrix.

For each state and for each pair of adjacent spins within that state, we apply the operator that flips adjacent spins, \$mask. Sometimes the adjacent spins will take on the same value, 00 or 11. In this circumstance, the off-diagonal part of the Hamiltonian would not act on those spins. The state generated by the operator would have a different magnetization, and we can neglect that pair.

If the new state produced by this process has the same magnetization, we know that the flip exists in the Hamiltonian, and add the entry to the matrix.

In this algorithm, we do end up going over each pair twice, but I have yet to figure out how to take advantage of the degeneracy to cut the calculation in half. Let me know if you have a better way to write this.

In [11]: mp=sum(psi_mz[1]&powers2./powers2)

```

for ii in 1:length(psi_mz)
    p=psi_mz[ii]
    for jj in 1:n
        flipped=p$mask[jj]
        if sum((flipped&powers2)./powers2) == mp
            tester=findstate(flipped,psi_mz)
            M[ii,tester]=.5
            M[tester,ii]=.5
            println(bin(p,n),'\t',bin(flipped,n))
        end
    end
end
end

```

```

0011      0101
0011      1010

```

```
0101      0110
0101      0011
0101      1001
0101      1100
0110      0101
0110      1010
1001      1010
1001      0101
1010      1001
1010      1100
1010      0110
1010      0011
1100      1010
1100      0101
```

```
In [12]: M
```

```
Out[12]: 6×6 Array{Float64,2}:
 0.0  0.5  0.0  0.0  0.5  0.0
 0.5  0.0  0.5  0.5  0.0  0.5
 0.0  0.5  0.0  0.0  0.5  0.0
 0.0  0.5  0.0  0.0  0.5  0.0
 0.5  0.0  0.5  0.5  0.0  0.5
 0.0  0.5  0.0  0.0  0.5  0.0
```

```
In [13]: F=eigfact(M)
         display(F[:values])
         display(F[:vectors])
```

```
6-element Array{Float64,1}:
-1.41421
 0.0
 0.0
 0.0
 1.9984e-15
 1.41421
```

```
6×6 Array{Float64,2}:
 0.353553  0.0      -0.211325  0.788675  -0.288675  -0.353553
-0.5      0.707107  0.0      0.0      -5.1279e-16  -0.5
 0.353553  0.0      0.788675  -0.211325  -0.288675  -0.353553
 0.353553  0.0     -0.57735  -0.57735  -0.288675  -0.353553
-0.5     -0.707107  0.0      0.0      -5.1279e-16  -0.5
 0.353553  0.0      0.0      0.0      0.866025  -0.353553
```

Now we have eigenvalues and eigenvectors! You just solved the Heisenberg Spin Chain!

In my next post, I will analyze what this tells us about the system and what we can do with the information.

References

- [1] Anders W. Sandvik, Adolfo Avella, and Ferdinando Mancini. Computational Studies of Quantum Spin Systems. volume 135, pages 135–338, 2010.