Quantum Ising Phase Transition

Carsten Bauer, Katharine Hyatt

August 11, 2019

0.1 Introduction

In this tutorial we will consider a simple quantum mechanical system of spins sitting on a chain. Here, *quantum mechanical*, despite its pompous sound, simply means that our Hamiltonian matrix will have a non-trivial (i.e. non-diagonal) matrix structure.

We will then ask a couple of basic questions,

- What is the ground state of the system?
- What happens if we turn on a transverse magnetic field?
- Are there any phase transitions?

To get answers to the questions, we will solve the time-independent Schrödinger equation

$$H|\psi\rangle = E|\psi\rangle$$

in Julia by means of exact diagonalization of the Hamiltonian.

0.2 Transverse field quantum Ising chain

Let's start out by defining our system. The Hamiltonian is given by

$$\mathcal{H} = -\sum_{\langle i,j \rangle} \hat{\sigma}_i^z \otimes \hat{\sigma}_j^z - h \sum_i \hat{\sigma}_i^x$$

Here, $\hat{\sigma}^z$ and $\hat{\sigma}^x$ are two of the three Pauli matrices, representing our quantum spins, $\langle i, j \rangle$ indicates that only neighboring spins talk to each other, and h is the amplitude of the magnetic field.

```
σz = [1 0; 0 -1] # \sigma <TAB> followed by \^z <TAB>
```

```
2×2 Array{Int64,2}:
 1   0
 0  -1
```

```
σx = [0 1; 1 0]
```

```
2×2 Array{Int64,2}:
 0  1
 1  0
```

Labeling the eigensates of σ^z as $|\downarrow\rangle$ and $|\uparrow\rangle$, we interpret them as a spin pointing down or up (in z-direction), respectively.

Clearly, since being purely off-diagonal, the effect of σ^x on such a single spin is to flip it:

$$\hat{\sigma}^x |\downarrow\rangle = |\uparrow\rangle$$
$$\hat{\sigma}^x |\uparrow\rangle = |\downarrow\rangle$$

The idea behind the Hamiltonian above is as follows:

- The first term is diagonal in the σ^z eigenbasis. If there is no magnetic field, $h = 0$, our quantum model reduces to the well-known classical Ising model (diagonal = trivial matrix structure -> classical). In this case, we have a **finite temperature phase transition** from a paramagnetic ($T > T_c$) phase, where the spins are **disordered by thermal fluctuations**, to a ferromagnetic phase ($T < T_c$), where they all point into the z direction and, consequently, a ferromagnetic ground state at $T = 0$.
- Since this would be boring, we want to add quantum complications to this picture by making H non-diagonal. To this end, we expose the quantum spins to a transverse magnetic field h in x direction in the second term. Now, since σ^z and σ^x do not commute (check $\sigma^z\sigma^x - \sigma^x\sigma^z$ yourself), there is no common eigenbasis of the first and the second term and our Hamiltonian has a non-trivial matrix structure (It's quantum!). If there was *only the second term* the system would, again, be trivial, as it would be diagonal in the eigenbasis of σ^x: the quantum spins want to be in an eigenstate of σ^x, i.e. align to the magnetic field.
- We can see that if we have both terms we have a competition between the spins wanting to point in the z direction (first term) and at the same time being disturbed by the transverse magnetic field. We say that the magnetic field term adds **quantum fluctuations** to the system.

Let us explore the physics of this interplay.

0.3 Building the Hamiltonian matrix

We will choose the σ^z eigenbasis as our computation basis.

To build up our Hamiltonian matrix we need to take the kronecker product (tensor product) of spin matrices. Fortunately, Julia has a built-in function for this.

```
kron(σ̃,σ̃) # this is the matrix of the tensor product σ̃_i⊗ σ̃_j (⊗ = \otimes <TAB>)
```

```
4×4 Array{Int64,2}:
 1   0   0  0
 0  -1   0  0
 0   0  -1  0
 0   0   0  1
```

Let's be fancy (cause we can!) and make this look a bit cooler.

```
⊗(x,y) = kron(x,y)
```

```
⊗ (generic function with 1 method)
```

```
σ̃ ⊗ σ̃
```

```
4×4 Array{Int64,2}:
 1   0   0  0
 0  -1   0  0
 0   0  -1  0
 0   0   0  1
```

0.3.1 Explicit 4-site Hamiltonian

Imagine our spin chain consists of four sites. Writing out identity matrices (which were left implicit in H above) explicitly, our Hamiltonian reads

$$\mathcal{H}_4 = -\hat{\sigma}_1^z\hat{\sigma}_2^z\hat{I}_3\hat{I}_4 - \hat{I}_1\hat{\sigma}_2^z\hat{\sigma}_3^z\hat{I}_4 - \hat{I}_1\hat{I}_2\hat{\sigma}_3^z\hat{\sigma}_4^z - h\left(\hat{\sigma}_1^x\hat{I}_2\hat{I}_3\hat{I}_4 + \hat{I}_1\hat{\sigma}_2^x\hat{I}_3\hat{I}_4 + \hat{I}_1\hat{I}_2\hat{\sigma}_3^x\hat{I}_4 + \hat{I}_1\hat{I}_2\hat{I}_3\hat{\sigma}_4^x\right)$$

(Note that we are considering *open* boundary conditions here - the spin on site 4 doesn't interact with the one on the first site. For *periodic* boundary conditions we'd have to add a term $-\hat{\sigma}_1^z\hat{I}_2\hat{I}_3\hat{\sigma}_4^z$.)

Translating this expression to Julia is super easy. After defining the identity matrix

```
id = [1 0; 0 1] # identity matrix
```

```
2×2 Array{Int64,2}:
 1  0
 0  1
```

we can simply write

```
h = 1
H = - σˣ⊗σˣ⊗id⊗id - id⊗σˣ⊗σˣ⊗id - id⊗id⊗σˣ⊗σˣ
H -= h*(σᶻ⊗id⊗id⊗id + id⊗σᶻ⊗id⊗id + id⊗id⊗σᶻ⊗id + id⊗id⊗id⊗σᶻ)
```

```
16×16 Array{Int64,2}:
 -3  -1  -1   0  -1   0   0   0  -1   0   0   0   0   0   0   0
 -1  -1   0  -1   0  -1   0   0   0  -1   0   0   0   0   0   0
 -1   0   1  -1   0   0  -1   0   0   0  -1   0   0   0   0   0
  0  -1  -1  -1   0   0   0  -1   0   0   0  -1   0   0   0   0
 -1   0   0   0   1  -1  -1   0   0   0   0   0  -1   0   0   0
  0  -1   0   0  -1   3   0  -1   0   0   0   0   0  -1   0   0
  0   0  -1   0  -1   0   1  -1   0   0   0   0   0   0  -1   0
  0   0   0  -1   0  -1  -1  -1   0   0   0   0   0   0   0  -1
 -1   0   0   0   0   0   0   0  -1  -1  -1   0  -1   0   0   0
  0  -1   0   0   0   0   0   0  -1   1   0  -1   0  -1   0   0
  0   0  -1   0   0   0   0   0  -1   0   3  -1   0   0  -1   0
  0   0   0  -1   0   0   0   0   0  -1  -1   1   0   0   0  -1
  0   0   0   0  -1   0   0   0  -1   0   0   0  -1  -1  -1   0
  0   0   0   0   0  -1   0   0   0  -1   0   0  -1   1   0  -1
  0   0   0   0   0   0  -1   0   0   0  -1   0  -1   0  -1  -1
  0   0   0   0   0   0   0  -1   0   0   0  -1   0  -1  -1  -3
```

There it is.

As nice as it is to write those tensor products explicitly, we certainly wouldn't want to write out all the terms for, say, 100 sites.

Let's define a function that iteratively does the job for us.

```
function TransverseFieldIsing(;N,h)
    id = [1 0; 0 1]
    σˣ = [0 1; 1 0]
    σᶻ = [1 0; 0 -1]

    # vector of operators:  [σᶻ, σᶻ, id, ...]
    first_term_ops = fill(id, N)
    first_term_ops[1] = σᶻ
    first_term_ops[2] = σᶻ

    # vector of operators:  [σˣ, id, ...]
    second_term_ops = fill(id, N)
    second_term_ops[1] = σˣ
```

```
    H = zeros(Int, 2^N, 2^N)
    for i in 1:N-1
        # tensor multiply all operators
        H -= foldl(⊗, first_term_ops)
        # cyclic shift the operators
        first_term_ops = circshift(first_term_ops,1)
    end

    for i in 1:N
        H -= h*foldl(⊗, second_term_ops)
        second_term_ops = circshift(second_term_ops,1)
    end
    H
end
```

```
TransverseFieldIsing (generic function with 1 method)
```

```
TransverseFieldIsing(N=8, h=1)
```

```
256×256 Array{Int64,2}:
 -7  -1  -1   0  -1   0   0   0  -1  …   0   0   0   0   0   0   0   0   0
 -1  -5   0  -1   0  -1   0   0   0      0   0   0   0   0   0   0   0   0
 -1   0  -3  -1   0   0  -1   0   0      0   0   0   0   0   0   0   0   0
  0  -1  -1  -5   0   0   0  -1   0      0   0   0   0   0   0   0   0   0
 -1   0   0   0  -3  -1  -1   0   0      0   0   0   0   0   0   0   0   0
  0  -1   0   0  -1  -1   0  -1   0  …   0   0   0   0   0   0   0   0   0
  0   0  -1   0  -1   0  -3  -1   0      0   0   0   0   0   0   0   0   0
  0   0   0  -1   0  -1  -1  -5   0      0   0   0   0   0   0   0   0   0
 -1   0   0   0   0   0   0   0  -3      0   0   0   0   0   0   0   0   0
  0  -1   0   0   0   0   0   0  -1      0   0   0   0   0   0   0   0   0
  ⋮                   ⋮               ⋱                   ⋮                   ⋮
  0   0   0   0   0   0   0   0   0     -3   0   0   0   0   0   0   0  -1
  0   0   0   0   0   0   0   0   0      0  -5  -1  -1   0  -1   0   0   0
  0   0   0   0   0   0   0   0   0      0  -1  -3   0  -1   0  -1   0   0
  0   0   0   0   0   0   0   0   0  …   0  -1   0  -1  -1   0   0  -1   0
  0   0   0   0   0   0   0   0   0      0   0  -1  -1  -3   0   0   0  -1
  0   0   0   0   0   0   0   0   0      0  -1   0   0   0  -5  -1  -1   0
  0   0   0   0   0   0   0   0   0      0   0  -1   0   0  -1  -3   0  -1
  0   0   0   0   0   0   0   0   0      0   0   0  -1   0  -1   0  -5  -1
  0   0   0   0   0   0   0   0   0  …  -1   0   0   0  -1   0  -1  -1  -7
```

0.3.2 Many-particle basis

Beyond a single spin, we have to think how to encode our basis states.

We make the arbitrary choice: $0 = \text{false} = \downarrow$ and $1 = \text{true} = \uparrow$

This way, our many-spin basis states have nice a binary representations and we can efficiently store them in a Julia `BitArray`.

Example: $|0010\rangle = |\text{false}, \text{false}, \text{true}, \text{false}\rangle = |\downarrow\downarrow\uparrow\downarrow\rangle$ is a basis state of a 4-site system

We construct the full basis by binary counting.

```
"""Binary `BitArray` representation of the given integer `num`, padded to length `N`."""
bit_rep(num::Integer, N::Integer) = BitArray(parse(Bool, i) for i in string(num, base=2, pad=N))

"""generate_basis(N::Integer) -> basisGenerates a basis (`Vector{BitArray}`) spanning the Hilbert
    space of `N` spins."""
function generate_basis(N::Integer)
    nstates = 2^N
    basis = Vector{BitArray{1}}(undef, nstates)
    for i in 0:nstates-1
```

```
        basis[i+1] = bit_rep(i, N)
    end
    return basis
end
```

```
Main.WeaveSandBox1.generate_basis
```

```
generate_basis(4)
```

```
16-element Array{BitArray{1},1}:
 [false, false, false, false]
 [false, false, false, true]
 [false, false, true, false]
 [false, false, true, true]
 [false, true, false, false]
 [false, true, false, true]
 [false, true, true, false]
 [false, true, true, true]
 [true, false, false, false]
 [true, false, false, true]
 [true, false, true, false]
 [true, false, true, true]
 [true, true, false, false]
 [true, true, false, true]
 [true, true, true, false]
 [true, true, true, true]
```

0.3.3 Side remark: Iterative construction of H

It might not be obvious that this basis is indeed the basis underlying the Hamiltonian matrix constructed in `TransverseFieldIsing`. To convince ourselves that this is indeed the case, let's calculate the matrix elements of our Hamiltonian, $\langle\psi_1|H|\psi_2\rangle$, explicitly by applying H to our basis states and utilizing their orthonormality, $\langle\psi_i|\psi_j\rangle = \sigma_{i,j}$.

```
using LinearAlgebra
```

```
function TransverseFieldIsing_explicit(; N::Integer, h::T=0) where T<:Real
    basis = generate_basis(N)
    H = zeros(T, 2^N, 2^N)
    bonds = zip(collect(1:N-1), collect(2:N))
    for (i, bstate) in enumerate(basis)
        # diagonal part
        diag_term = 0.
        for (site_i, site_j) in bonds
            if bstate[site_i] == bstate[site_j]
                diag_term -= 1
            else
                diag_term += 1
            end
        end
        H[i, i] = diag_term

        # off diagonal part
        for site in 1:N
            new_bstate = copy(bstate)
            # flip the bit on the site (that's what σ⊗ does)
            new_bstate[site] = !new_bstate[site]
            # find corresponding single basis state with unity overlap (orthonormality)
            new_i = findfirst(isequal(new_bstate), basis)
            H[i, new_i] = -h
        end
    end
    return H
```

```
end
```

```
TransverseFieldIsing_explicit (generic function with 1 method)
```

```
TransverseFieldIsing_explicit(N=4, h=1) ≈ TransverseFieldIsing(N=4, h=1)
```

```
true
```

0.3.4 Full exact diagonalization

Alright. Let's solve the Schrödinger equation by diagonalizing H for a system with $N = 8$ and $h = 1$.

```
basis = generate_basis(8)
H = TransverseFieldIsing(N=8, h=1)
vals, vecs = eigen(H)
```

```
LinearAlgebra.Eigen{Float64,Float64,Array{Float64,2},Array{Float64,1}}
eigenvalues:
256-element Array{Float64,1}:
 -9.837951447459426
 -9.46887800960621
 -8.74329948717107
 -8.374226049317867
 -8.054998024353266
 -7.685924586500063
 -7.427412901942416
 -7.058339464089192
 -6.960346064064927
 -6.881915778576785
  ⋮
  6.960346064064934
  7.0583394640891886
  7.427412901942393
  7.685924586500062
  8.054998024353269
  8.374226049317883
  8.74329948717109
  9.468878009606211
  9.83795144745942
eigenvectors:
256×256 Array{Float64,2}:
 0.340927   -0.459362   -3.81639e-17  …   7.97973e-17   0.00885459
 0.173416   -0.210059    0.0826509        -0.00480955   -0.0133586
 0.109265   -0.126567    0.0600217         0.00990677   -0.0214075
 0.135509   -0.123533    0.182584         -0.00219934    0.0121301
 0.101521   -0.116488    0.0389137        -0.0115529    -0.0195355
 0.0603592  -0.0541307   0.0677375    …    0.0296823     0.0367401
 0.0754706  -0.0572934   0.103307          0.00222615    0.0173773
 0.120803   -0.0590509   0.260672         -0.00610295   -0.0128526
 0.0995642  -0.11394     0.0135226         0.0125492    -0.020245
 0.0532448  -0.04999     0.0381968        -0.00794715    0.0263806
 ⋮                                    ⋱                  ⋮
 0.0995642   0.11394    -0.0135226        -0.0125492    -0.020245
 0.120803    0.0590509  -0.260672          0.00610295   -0.0128526
 0.0754706   0.0572934  -0.103307         -0.00222615    0.0173773
 0.0603592   0.0541307  -0.0677375    …   -0.0296823     0.0367401
 0.101521    0.116488   -0.0389137         0.0115529    -0.0195355
 0.135509    0.123533   -0.182584          0.00219934    0.0121301
 0.109265    0.126567   -0.0600217        -0.00990677   -0.0214075
 0.173416    0.210059   -0.0826509         0.00480955   -0.0133586
 0.340927    0.459362    0.0           …   4.38622e-18   0.00885459
```

That's it. Here is our groundstate.

```
groundstate = vecs[:,1];
```

The absolute square of this wave function is the probability of finding the system in a particular basis state.

```
abs2.(groundstate)
```

```
256-element Array{Float64,1}:
 0.11623105759942885
 0.030073150814502212
 0.0119388989548912
 0.01836268922781065
 0.010306563749646199
 0.0036432311839576883
 0.005695810419718821
 0.014593393364127294
 0.009913022568277332
 0.002835013679521494
 ⋮
 0.009913022568277134
 0.014593393364126966
 0.005695810419718817
 0.003643231183957665
 0.010306563749646001
 0.018362689227810196
 0.01193889895489093
 0.030073150814501577
 0.11623105759942208
```

It's instructive to look at the extremal cases $h = 0$ and $h >> 1$.

```
H = TransverseFieldIsing(N=8, h=0)
vals, vecs = eigen(H)
groundstate = vecs[:,1]
abs2.(groundstate)
```

```
256-element Array{Float64,1}:
 1.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 ⋮
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
```

As we can see, for $h = 0$ the system is (with probability one) in the first basis state, where all spins point in $-z$ direction.

```
basis[1]
```

```
8-element BitArray{1}:
 false
 false
 false
 false
 false
 false
 false
 false
```

On the other hand, for $h = 100$, the system occupies all basis states with approximately equal probability (maximal superposition) - corresponding to eigenstates of σ^x, i.e. alignment to the magnetic field.

```
H = TransverseFieldIsing(N=8, h=100)
vals, vecs = eigen(H)
groundstate = vecs[:,1]
abs2.(groundstate)
```

```
256-element Array{Float64,1}:
 0.004045339592145809
 0.004004987329714098
 0.003965137205785807
 0.004004886702523801
 0.003965038080465978
 0.003925683101156531
 0.003964938454528805
 0.004004886199391184
 0.003965037585486796
 0.003925487314471259
 ⋮
 0.003965037585486885
 0.004004886199391219
 0.003964938454528845
 0.003925683101156275
 0.003965038080466088
 0.004004886702523875
 0.003965137205785895
 0.004004987329714019
 0.004045339592145809
```

1 Are you a magnet or what?

Let's vary h and see what happens. Since we're looking at quantum magnets we will compute the overall magnetization, defined by

$$M = \frac{1}{N}\sum_i \sigma_i^z$$

where σ_i^z is the value of the spin on site i when we measure.

```
function magnetization(state, basis)
    M = 0.
    for (i, bstate) in enumerate(basis)
        bstate_M = 0.
        for spin in bstate
            bstate_M += (state[i]^2 * (spin ? 1 : -1))/length(bstate)
        end
        @assert abs(bstate_M) <= 1
        M += abs(bstate_M)
    end
```

```
    return M
end
```

```
magnetization (generic function with 1 method)
```

```
magnetization(groundstate, basis)
```

```
0.2748106008973601
```

Now we would like to examine the effects of h. We will:

1. Find a variety of h to look at.
2. For each, compute the lowest energy eigenvector (groundstate) of the corresponding Hamiltonian.
3. For each groundstate, compute the overall magnetization M.
4. Plot $M(h)$ for a variety of system sizes, and see if anything cool happens.

```
using Plots
hs = 10 .^ range(-2., stop=2., length=10)
Ns = 2:10
p = plot()
for N in Ns
    M = zeros(length(hs))
    for (i,h) in enumerate(hs)
        basis = generate_basis(N)
        H = TransverseFieldIsing(N=N, h=h)
        vals, vecs = eigen(H)
        groundstate = vecs[:,1]
        M[i] = magnetization(groundstate, basis)
    end
    plot!(p, hs, M, xscale=:log10, marker=:circle, label="N = $N",
        xlab="h", ylab="M(h)")
    println(M)
end
```

```
[0.9999, 0.999228, 0.994111, 0.959187, 0.820255, 0.643561, 0.553551, 0.5193
42, 0.506956, 0.5025]
[0.999962, 0.999697, 0.997483, 0.977379, 0.835423, 0.620175, 0.538777, 0.51
3279, 0.504686, 0.501673]
[0.999969, 0.999757, 0.998091, 0.983463, 0.830993, 0.517856, 0.418746, 0.38
9914, 0.380267, 0.376881]
[0.999972, 0.999787, 0.998342, 0.986444, 0.849703, 0.500414, 0.410874, 0.38
703, 0.379226, 0.376507]
[0.999975, 0.999806, 0.998495, 0.987923, 0.861409, 0.444473, 0.349674, 0.32
4999, 0.316897, 0.314069]
[0.999977, 0.99982, 0.998603, 0.988861, 0.875535, 0.431583, 0.344792, 0.323
263, 0.316275, 0.313845]
[0.999978, 0.999831, 0.998684, 0.989526, 0.88554, 0.394698, 0.306229, 0.284
401, 0.277288, 0.274811]
[0.999979, 0.999839, 0.998746, 0.990035, 0.894378, 0.384896, 0.302852, 0.28
3213, 0.276864, 0.274659]
[0.99998, 0.999845, 0.998797, 0.99044, 0.900862, 0.358218, 0.275738, 0.2559
74, 0.249561, 0.24733]
```

```
p
```

This looks like a phase transition!

For small h, the magnetization is unity, corresponding to a ferromagnetic state. By increasing the magnetic field h we have a competition between the two terms in the Hamiltonian and eventually the system becomes paramagnetic with $M \approx 0$. Our plot suggests that this change of state happens around $h \sim 1$, which is in good agreement with the exact solution $h = 1$.

It is crucial to realize, that in our calculation we are inspecting the ground state of the system. Since $T = 0$, it is purely quantum fluctuations that drive the transition: a **quantum phase transition**! This is to be compared to increasing temperature in the classical Ising model, where it's thermal fluctuations that cause a classical phase transition from a ferromagnetic to a paramagnetic state. For this reason, the state that we observe at high magnetic field strengths is called a **quantum paramagnet**.

1.1 Hilbert space is a big space

So far, we have only inspected chains of length $N \leq 10$. As we see in our plot above, there are rather strong finite-size effects on the magnetization. To extract a numerical estimate for the critical magnetic field strength h_c of the transition we would have to consider much larger systems until we observe convergence as a function of N. Although this is clearly beyond the scope of this tutorial, let us at least pave the way.

Our calculation, in its current form, doesn't scale. The reason for this is simple, **Hilbert space is a big place!**

The number of basis states, and therefore the number of dimensions, grows **exponentially** with system size.

```
plot(N -> 2^N, 1, 20, legend=false, color=:black, xlab="N", ylab="# Hilbert space dimensions")
```

Our Hamiltonian matrix therefore will become huge(!) and is not going to fit into memory (apart from the fact that diagonalization would take forever).

```
using Test
@test_throws OutOfMemoryError TransverseFieldIsing(N=20, h=1)
```

```
Test Passed
      Thrown: OutOfMemoryError
```

So, what can we do about it? The answer is, **sparsity**.

Let's inspect the Hamiltonian a bit more closely.

```
H = TransverseFieldIsing(N=10, h=1)
```

```
1024×1024 Array{Int64,2}:
 -9  -1  -1   0  -1   0   0   0  -1  …   0   0   0   0   0   0   0   0   0
 -1  -7   0  -1   0  -1   0   0   0      0   0   0   0   0   0   0   0   0
 -1   0  -5  -1   0   0  -1   0   0      0   0   0   0   0   0   0   0   0
  0  -1  -1  -7   0   0   0  -1   0      0   0   0   0   0   0   0   0   0
 -1   0   0   0  -5  -1  -1   0   0      0   0   0   0   0   0   0   0   0
  0  -1   0   0  -1  -3   0  -1   0  …   0   0   0   0   0   0   0   0   0
  0   0  -1   0  -1   0  -5  -1   0      0   0   0   0   0   0   0   0   0
  0   0   0  -1   0  -1  -1  -7   0      0   0   0   0   0   0   0   0   0
 -1   0   0   0   0   0   0   0  -5      0   0   0   0   0   0   0   0   0
  0  -1   0   0   0   0   0   0  -1      0   0   0   0   0   0   0   0   0
  ⋮                   ⋮              ⋱   ⋮                   ⋮
  0   0   0   0   0   0   0   0   0  …  -5   0   0   0   0   0   0   0  -1
  0   0   0   0   0   0   0   0   0      0  -7  -1  -1   0  -1   0   0   0
  0   0   0   0   0   0   0   0   0      0  -1  -5   0  -1   0  -1   0   0
  0   0   0   0   0   0   0   0   0      0  -1   0  -3  -1   0   0  -1   0
```

```
 0   0   0   0   0   0   0   0   0      0   0  -1  -1  -5   0   0   0  -1
 0   0   0   0   0   0   0   0   0  ...   0  -1   0   0   0  -7  -1  -1   0
 0   0   0   0   0   0   0   0   0      0   0  -1   0   0  -1  -5   0  -1
 0   0   0   0   0   0   0   0   0      0   0   0  -1   0  -1   0  -7  -1
 0   0   0   0   0   0   0   0   0     -1   0   0   0  -1   0  -1  -1  -9
```

Noticably, there are a lot of zeros. How does this depend on N?

Let's plot the sparsity, i.e. ratio of zero entries.

```
sparsity(x) = count(isequal(0), x)/length(x)

Ns = 2:12
sparsities = Float64[]
for N in Ns
    H = TransverseFieldIsing(N=N, h=1)
    push!(sparsities, sparsity(H))
end
plot(Ns, sparsities, legend=false, xlab="chain length N", ylab="Hamiltonian
    sparsity", marker=:circle)
```

For $N \gtrsim 10$ almost all entries are zero! We should get rid of those and store H as a sparse matrix.

1.1.1 Building the sparse Hamiltonian

Generally, we can bring a dense matrix into a sparse matrix format using the function `sparse`.

```
using SparseArrays
H = TransverseFieldIsing(N=4,h=1)
H |> sparse
```

```
16×16 SparseArrays.SparseMatrixCSC{Int64,Int64} with 80 stored entries:
  [1 ,  1]  =  -3
  [2 ,  1]  =  -1
  [3 ,  1]  =  -1
  [5 ,  1]  =  -1
  [9 ,  1]  =  -1
  [1 ,  2]  =  -1
  [2 ,  2]  =  -1
  [4 ,  2]  =  -1
  [6 ,  2]  =  -1
   ⋮
  [11, 15]  =  -1
  [13, 15]  =  -1
  [15, 15]  =  -1
  [16, 15]  =  -1
  [8 , 16]  =  -1
  [12, 16]  =  -1
  [14, 16]  =  -1
  [15, 16]  =  -1
  [16, 16]  =  -3
```

Note that in this format, only the 80 non-zero entries are stored (rather than 256 elements).

So, how do we have to modify our function `TransverseFieldIsing` to only keep track of non-zero elements during the Hamiltonian construction?

It turns out it is as simple as initializing our Hamiltonian, identity, and pauli matrices as sparse matrices!

```
function TransverseFieldIsing_sparse(;N,h)
    id = [1 0; 0 1] |> sparse
```

```
    σ⨷ = [0 1; 1 0] |> sparse
    σ⨀ = [1 0; 0 -1] |> sparse

    first_term_ops = fill(id, N)
    first_term_ops[1] = σ⨀
    first_term_ops[2] = σ⨀

    second_term_ops = fill(id, N)
    second_term_ops[1] = σ⨷

    H = spzeros(Int, 2^N, 2^N) # note the spzeros instead of zeros here
    for i in 1:N-1
        H -= foldl(⊗, first_term_ops)
        first_term_ops = circshift(first_term_ops,1)
    end

    for i in 1:N
        H -= h*foldl(⊗, second_term_ops)
        second_term_ops = circshift(second_term_ops,1)
    end
    H
end
```

```
TransverseFieldIsing_sparse (generic function with 1 method)
```

We should check that apart from the new type `SparseMatrixCSC` this is still the same Hamiltonian.

```
H = TransverseFieldIsing_sparse(N=10, h=1);

H_dense = TransverseFieldIsing(N=10, h=1)
H ≈ H_dense
```

```
true
```

Great. But is it really faster?

```
@time TransverseFieldIsing(N=10,h=1);
```

```
0.201316 seconds (379 allocations: 442.688 MiB, 25.17% gc time)
```

```
@time TransverseFieldIsing_sparse(N=10,h=1);
```

```
0.000724 seconds (887 allocations: 2.621 MiB)
```

It is *a lot* faster!

Alright, let's try to go to larger N. While `TransverseFieldIsing` threw an `OutOfMemoryError` for `N=20`, our new function is more efficient:

```
@time H = TransverseFieldIsing_sparse(N=20,h=1)
```

```
5.657440 seconds (4.58 k allocations: 6.742 GiB, 49.12% gc time)
1048576×1048576 SparseArrays.SparseMatrixCSC{Int64,Int64} with 22020096 sto
red entries:
  [1      ,       1]  =  -19
  [2      ,       1]  =  -1
  [3      ,       1]  =  -1
  [5      ,       1]  =  -1
  [9      ,       1]  =  -1
  [17     ,       1]  =  -1
  [33     ,       1]  =  -1
  [65     ,       1]  =  -1
  [129    ,       1]  =  -1
  ⋮
```

```
  [1048448, 1048576]  =  -1
  [1048512, 1048576]  =  -1
  [1048544, 1048576]  =  -1
  [1048560, 1048576]  =  -1
  [1048568, 1048576]  =  -1
  [1048572, 1048576]  =  -1
  [1048574, 1048576]  =  -1
  [1048575, 1048576]  =  -1
  [1048576, 1048576]  =  -19
```

Note that this is matrix, formally, has **1,099,511,627,776** entries!

1.1.2 Diagonalizing sparse matrices

We have taken the first hurdle of constructing our large-system Hamiltonian as a sparse matrix. Unfortunately, if we try to diagonalize H, we realize that Julia's built-in eigensolver `eigen` doesn't support matrices.

```
eigen(A) not supported for sparse matrices. Use for example eigs(A) from the Arpack package i
```

Gladly it suggests a solution: ARPACK.jl. It provides a wrapper to the Fortran library ARPACK which implements iterative eigenvalue and singular value solvers for sparse matrices.

There are also a bunch of pure Julia implementations available in

- ArnoldiMethod.jl
- KrylovKit.jl
- IterativeSolvers.jl

Let us use the ArnoldiMethod.jl package.

```
using ArnoldiMethod

function eigen_sparse(x)
    decomp, history = partialschur(x, nev=1, which=SR()); # only solve for the ground state
    vals, vecs = partialeigen(decomp);
    return vals, vecs
end
```

```
eigen_sparse (generic function with 1 method)
```

Solving for the ground state takes less than a minute on an i5 desktop machine.

```
@time vals, vecs = eigen_sparse(H)
```

```
8.862459 seconds (52.81 k allocations: 338.700 MiB, 0.19% gc time)
([-25.1078], [-0.149161; -0.0748002; ... ; -0.0748002; -0.149161])
```

Voila. There we have the ground state energy and the ground state wave function for a $N = 20$ chain of quantum spins!

```
groundstate = vecs[:,1]
```

```
1048576-element Array{Float64,1}:
 -0.14916143684856045
 -0.07480019660537927
 -0.04680670393715045
 -0.056487648246642416
```

```
-0.04331037948297223
-0.025019354338629528
-0.030717722411135618
-0.04758869250743675
-0.04222144234078573
-0.022038589547061716
 ⋮
-0.042221442583761565
-0.04758869206240731
-0.030717722316464195
-0.02501935432807452
-0.04331037968987974
-0.05648764817798465
-0.04680670413717739
-0.07480019698519194
-0.14916143804710202
```

1.1.3 Magnetization once again

To measure the magnetization, we could use our function `magnetization(state, basis)` from above. However, the way we wrote it above, it depends on an explicit list of basis states which we do not want to construct for a large system explicitly.

Let's rewrite the function slightly such that bit representations of our basis states are calculated on the fly.

```
function magnetization(state)
    N = Int(log2(length(state)))
    M = 0.
    for i in 1:length(state)
        bstate = bit_rep(i-1,N)
        bstate_M = 0.
        for spin in bstate
            bstate_M += (state[i]^2 * (spin ? 1 : -1))/N
        end
        @assert abs(bstate_M) <= 1
        M += abs(bstate_M)
    end
    return M
end
```

```
magnetization (generic function with 2 methods)
```

```
magnetization(groundstate, basis)
```

```
0.05009671029999692
```

We are now able to recreate our magnetization vs magnetic field strength plotincluding larger systems (takes about 3 minutes on this i5 Desktop machine).

```
using Plots
hs = 10 .^ range(-2., stop=2., length=10)
Ns = 2:2:20
p = plot()
@time for N in Ns
    M = zeros(length(hs))
    for (i,h) in enumerate(hs)
        H = TransverseFieldIsing_sparse(N=N, h=h)
        vals, vecs = eigen_sparse(H)
        groundstate = @view vecs[:,1]
        M[i] = magnetization(groundstate)
    end
```

```
    plot!(p, hs, M, xscale=:log10, marker=:circle, label="N = $N",
        xlab="h", ylab="M(h)")
    println(M)
end
```

```
[0.9999, 0.999228, 0.994111, 0.959187, 0.820255, 0.643561, 0.553551, 0.5193
42, 0.506956, 0.5025]
[0.999969, 0.999757, 0.998091, 0.983463, 0.830993, 0.517856, 0.418746, 0.38
9914, 0.380267, 0.376881]
[0.999975, 0.999806, 0.998495, 0.987923, 0.861409, 0.444473, 0.349674, 0.32
4999, 0.316897, 0.314069]
[0.999978, 0.999831, 0.998684, 0.989526, 0.88554, 0.394698, 0.306229, 0.284
401, 0.277288, 0.274811]
[0.99998, 0.999845, 0.998797, 0.99044, 0.900862, 0.358218, 0.275738, 0.2559
74, 0.249561, 0.24733]
[0.999981, 0.999855, 0.998872, 0.991047, 0.910222, 0.330096, 0.252836, 0.23
465, 0.228765, 0.226719]
[0.999982, 0.999862, 0.998926, 0.991479, 0.916192, 0.307612, 0.234825, 0.21
7894, 0.212425, 0.210525]
[0.999983, 0.999867, 0.998966, 0.991804, 0.920281, 0.289126, 0.220182, 0.20
4279, 0.199149, 0.197367]
[0.999983, 0.999871, 0.998998, 0.992057, 0.92328, 0.273591, 0.207974, 0.192
933, 0.188086, 0.186402]
[0.999984, 0.999874, 0.999023, 0.992259, 0.925601, 0.260303, 0.197594, 0.18
3288, 0.178682, 0.177082]
147.884727 seconds (84.41 M allocations: 149.259 GiB, 27.79% gc time)
```

```
p
```