Numerical Methods for Manu Body Physics, Assignment #1

Yoav Zack, ID 211677398

February 12, 2024

In this assignment we will work on the (J_1, J_2) spin-$\frac{1}{2}$ extension to the Heisenberg model, which is sometimes called the J1-J2 Model:

$$H = J_1 \sum_i \vec{S}_i \cdot \vec{S}_{i+1} + J_2 \sum_i \vec{S}_i \cdot \vec{S}_{i+2}$$

We assume $J_1 > 0, J_2 \geq 0$ and denote $g = J_2/J_1$.

1 Question 0: Warm Up

We assume $J_2 = 0$ as required, thus the Hamiltonian is the Heisenberg one:

$$H = J_1 \sum_i \vec{S}_i \cdot \vec{S}_{i+1} = J_1 \sum_{\langle i,j \rangle} \left[\frac{1}{2}(\sigma_i^+ \sigma_j^- + \sigma_i^- \sigma_j^+) + \frac{1}{4}\sigma_i^z \sigma_j^z \right]$$

This means the for each pair of adjacent spins i, j, the state $|\psi\rangle$ turns into a sum of two elements:

1. A state with switched places for all pairs of opposite spins, with coefficient $-\frac{J_1}{2}$.
2. The same state as $|\psi\rangle$, with a coefficient $\frac{J_1}{4}$ and also a sum over all pairs of adjacent spins, $+1$ for $++/--$ and -1 for $+-/-+$.

Note that this means that the state $|\psi\rangle$ of the system must be represented as an array of real numbers, each on corresponding to the coefficient of a single pure state in the superposition, while each pure state is still a single `UInt`.

But first, some imports and global definitions:

[1]:
```
using LinearAlgebra, SparseArrays, Arpack, Random
using Plots, Printf, LaTeXStrings
using LsqFit

theme(:default)
default(background_color=:transparent, dpi=300)

rng = MersenneTwister(42)

Nmax = 10;
```

We start by creating two functions which we will use a lot later:

[2]:
```
function index2state(stateind::Integer, N::Integer)
  return digits(stateind, base=2, pad=N)
end

function flipspins(x,i,j) # takes i=(1,N) and j=(1,N)
    f = typeof(x)(1)<<(i-1) | typeof(x)(1)<<(j-1)
    return x   f
end

function ground_state_energy(H)
  return eigs(H; nev=1, which=:SR, ritzvec=false)[1][1]
end
```

[2]:
```
ground_state_energy (generic function with 1 method)
```

And now that we have that, we will use it to calculate the effect of the Hamiltonian on a general wave function:

[3]:
```
function multiply_heisenberg( ::Vector{<:Number}, J::Real)
  @assert abs(norm( ) - 1) < 1e-9 "Input state is not normalized"
  D = length( )
  @assert ispow2(D) "Input state has invalid number of elements"
  N = Int(log2(D))
  @assert mod(N,2)==0 "Only even number of spins is supported"

   out = zeros(D)
  for stateind in range(0, length=D)
    for i in range(1, length=N)
      j = mod(i,N)+1
      si = ( stateind & 1<<(i-1) ) >> (i-1)
      sj = ( stateind & 1<<(j-1) ) >> (j-1)

      if si == sj
         out[stateind+1] += J/4* [stateind+1]
      else
         out[stateind+1] -= J/4* [stateind+1]
        stateind_flipped = flipspins(stateind, i, j)
         out[stateind_flipped+1] += J/2* [stateind+1]
      end
    end
  end
  return  out
end
```

[3]:
```
multiply_heisenberg (generic function with 1 method)
```

In order to get the energy of the ground state, we will also need the Hamiltonian matrix from class:

[4]:
```
function heisenberg_hamiltonian(N)
    H = spzeros(2^N,2^N)

    for stateind in range(0, length = 2^N)
        for i in range(1, length = N)
            j = mod(i,N)+1
            si = ( stateind & 1<<(i-1) ) >> (i-1)
            sj = ( stateind & 1<<(j-1) ) >> (j-1)

            if si == sj
                H[stateind+1,stateind+1] += 1/4
            else
                H[stateind+1,stateind+1] -= 1/4
                stateind_flipped = flipspins(stateind,i,j)
                H[stateind+1,stateind_flipped+1] += 1/2
            end
        end
    end

    return H
end
```

[4]:
```
heisenberg_hamiltonian (generic function with 1 method)
```

and a function which calculates the energy of a given state:

[5]:
```
function heisenberg_energy( ::Vector{<:Number}, J::Real)
  return  multiply_heisenberg( , J)
end
```

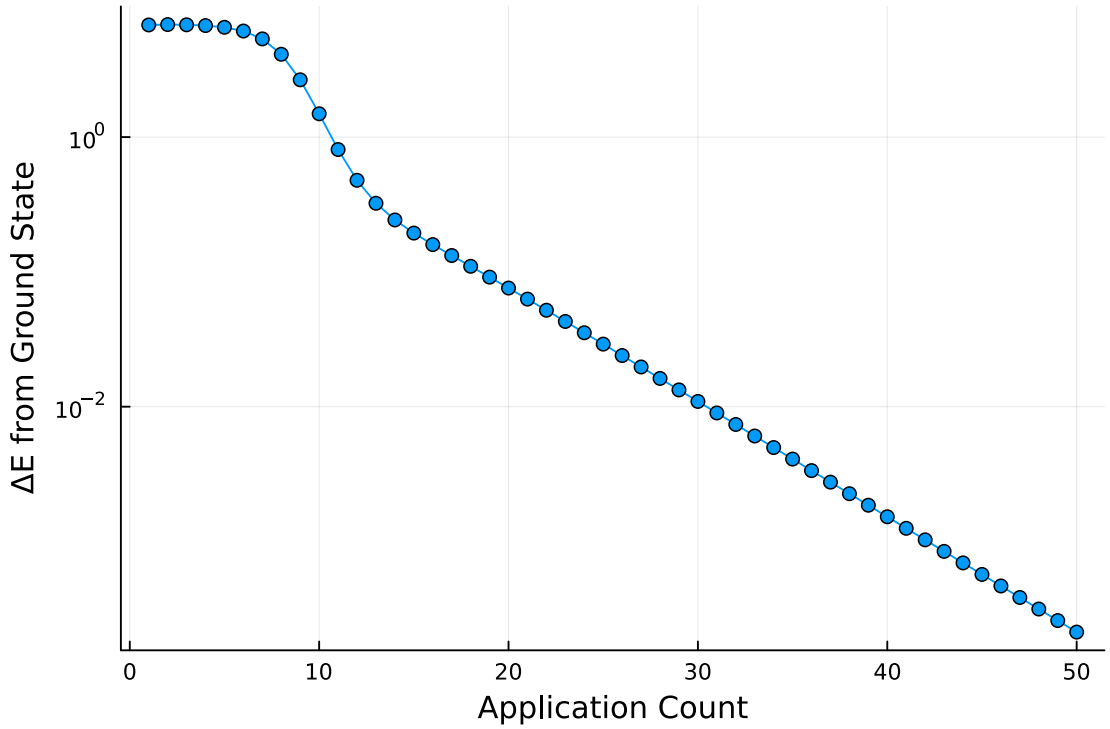
[5]:
```
heisenberg_energy (generic function with 1 method)
```

Using it we can calculate the energy after repeatedly applying the Hamiltonian and see that it converges to the ground state energy:

[6]:
```
J = 1.0
N = Nmax
  = normalize(rand(rng,2^N))

H = heisenberg_hamiltonian(N)
Emin = ground_state_energy(H)

iternum = 50
Erng = zeros(iternum)
Eground = zeros(iternum)
for ind in range(1, length=iternum)
    = multiply_heisenberg( , J)
  normalize!( )
```

```
  Erng[ind] = heisenberg_energy( , J)
  Eground[ind] = Emin
end

plot(Erng .- Eground , label=nothing, yaxis=:log, marker=:circle)
xlabel!("Application Count")
ylabel!("ΔE from Ground State")
```

[6]:



And we can see that the energy of the state goes towards the energy of the Ground State, as expected.

2 Question 1: Hamiltonian for $g \neq 0$

To extend the given Hamiltonian, first we need to define the same fixed S_z basis as in the tutorial:

[7]:
```
struct fixed_sz_basis
    N::Int64
    Nup::Int64
    states::Vector{Int64}

    function fixed_sz_basis(N::Int, Nup::Int)
      @assert mod(N, 2) == 0 "Number of spins most be even."
      Ndown = N - Nup
```

```
        D = binomial(N, Nup)
        states = zeros(Int, D)
        k=1
        for a in range(0, length = 2^N) # loop over all basis states
            if count_ones(a) == Nup
                states[k] = a
                k += 1
            end
        end
        new(N, Nup, states)
    end
end
```

It also requires helper functions:

[8]:
```
import Base.length
function length(b::fixed_sz_basis)
    return length(b.states)
end

function Sz(b::fixed_sz_basis)
    Ndown = b.N - b.Nup
    return (b.Nup-Ndown)/2
end
```

[8]:
```
Sz (generic function with 1 method)
```

Using this basis, we can create a function which generates a Hamiltonian Matrix for $g \neq 0$:

[9]:
```
function construct_g_hamiltonian(basis::fixed_sz_basis, J1::Real, J2::Real)
    D = length(basis)
    H = spzeros(D,D)

    for k in range(1, length = D)
        stateind = basis.states[k]
        for i in range(1, length = basis.N)
            j = mod(i, basis.N)+1
            h = mod(j, basis.N)+1
            si = ( stateind & 1<<(i-1) ) >> (i-1)
            sj = ( stateind & 1<<(j-1) ) >> (j-1)
            sh = ( stateind & 1<<(h-1) ) >> (h-1)

            if si == sj
                H[k,k] += J1/4
            else
                H[k,k] -= J1/4
                stateind_flipped = flipspins(stateind,i,j)
                l = searchsortedfirst(basis.states,stateind_flipped)
```

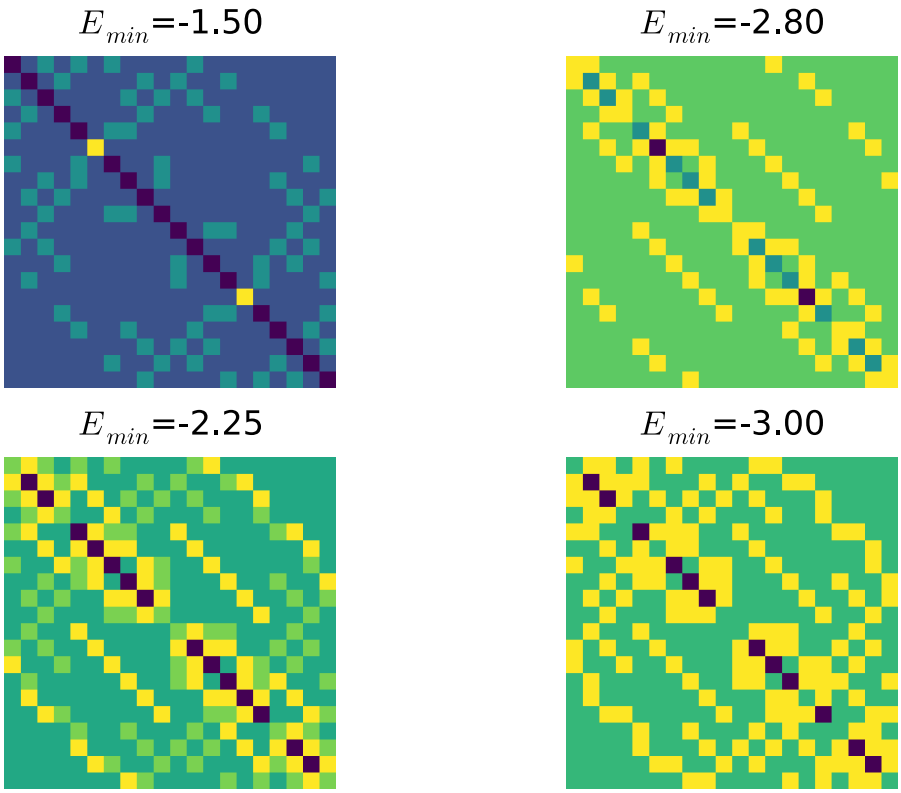
```
                @assert (l<=D) && (basis.states[l] == stateind_flipped)
                  "Invalid basis state generated by flipspins"
                H[k,l] += J1/2
            end

            if si == sh
                H[k,k] += J2/4
            else
                H[k,k] -= J2/4
                stateind_flipped = flipspins(stateind,i,h)
                l = searchsortedfirst(basis.states,stateind_flipped)
                @assert (l<=D) && (basis.states[l] == stateind_flipped)
                  "Invalid basis state generated by flipspins"
                H[k,l] += J2/2
            end
        end
    end

    return H
end
```

[9]:
```
construct_g_hamiltonian (generic function with 1 method)
```

Let's test it on a few simple cases spins with given $N, N_{\uparrow}$ pairs:

[10]:
```
N = 6
Nup = 3
b = fixed_sz_basis(N, Nup)
Harr = []
push!(Harr, construct_g_hamiltonian(b, 0.0, 1.0))
push!(Harr, construct_g_hamiltonian(b, 1.0, 0.0))
push!(Harr, construct_g_hamiltonian(b, 1.0, 0.5))
push!(Harr, construct_g_hamiltonian(b, 1.0, 1.0))

plt_list = []
l = @layout[a b; c d]
for H in Harr
    plt = heatmap(H, size=(170, 200), legend=false, aspect_ratio=:equal,␣
 ↪axis=([], false), yflip = true, title=L"E_{\rm min}"*@sprintf("=%.2f",␣
 ↪ground_state_energy(H)), c=:viridis)
    push!(plt_list, plt)
end # TODO fix color scheme
plot(plt_list[1], plt_list[2], plt_list[3], plt_list[4], layout=l, size=(700,␣
 ↪500))
```

[10]:



3 Question 2: Triplet Gap

In the following questions we will work on the following cases:

[11]:
```
gc = 0.241
garr = [0, gc, 0.49, 0.5]
```

[11]:
```
4-element Vector{Float64}:
 0.0
 0.241
 0.49
 0.5
```

To perform the triplet gap test we will calculate the energy of the ground state for $S^z = 0$ and the lowest energy for $S^z = 1$:

[12]:
```
Narr = 4:2:Nmax
J1 = 1.0

E0 = zeros(length(Narr), length(garr))
E1 = zeros(length(Narr), length(garr))
for (Nind, N) in enumerate(Narr)
```

```
  b0 = fixed_sz_basis(N, Int(N/2))
  b1 = fixed_sz_basis(N, Int(N/2)+1)
  for (gind, g) in enumerate(garr)
    J2 = g*J1
    H0 = construct_g_hamiltonian(b0, J1, J2)
    H1 = construct_g_hamiltonian(b1, J1, J2)

    res0 = eigs(H0; nev=1, which=:SR, ritzvec=false)[1]
    res1 = eigs(H1; nev=1, which=:SR, ritzvec=false)[1]
    E0[Nind, gind] = res0[1]
    E1[Nind, gind] = res1[1]
  end
end
ΔE = E1 .- E0
```

[12]:
```
4×4 Matrix{Float64}:
 1.0       1.0       1.0       1.0
 0.684742  0.63317   0.507243  0.5
 0.522674  0.468812  0.404902  0.404517
 0.423239  0.372804  0.345978  0.350212
```

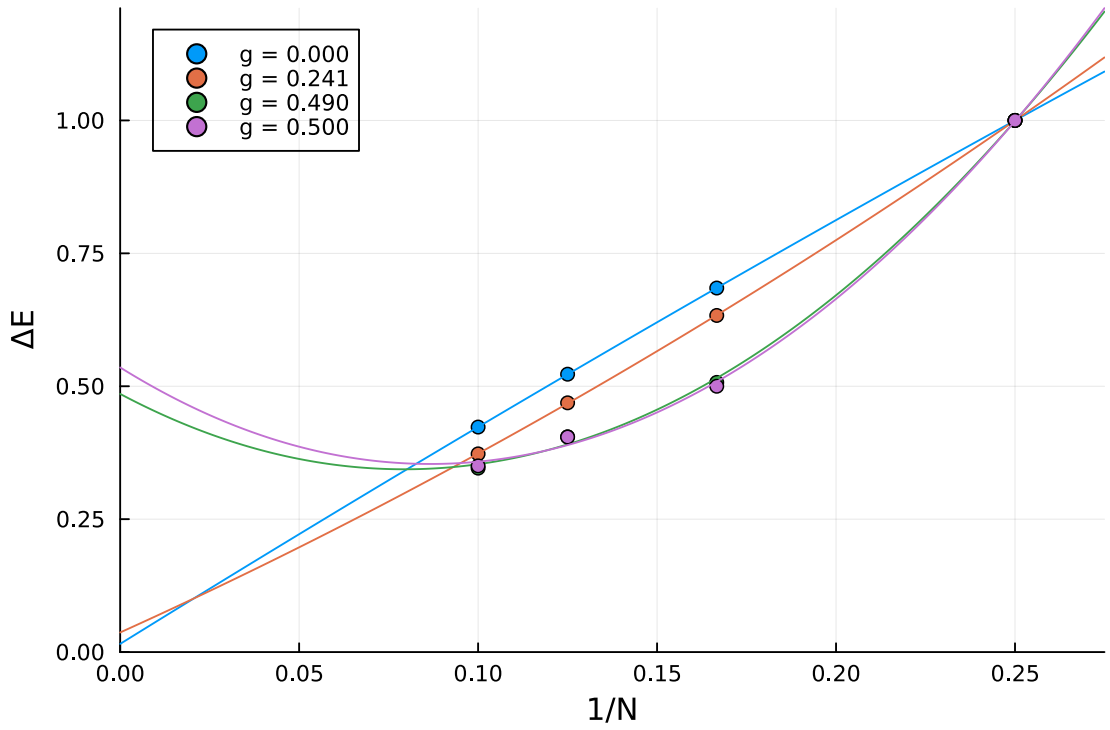
To verify that the gap is correct, we will plot it as a function of $\frac{1}{N}$ and verify that it is:

1. Goes to zero for $g \leq g_c$
2. Does not go to zero for $g > g_c$

We also perform a linear fit to each case to verify that it actually goes to zero at $N \to \infty$:

[13]:
```
@. model(x, a) = a[1] + a[2]*x + a[3]*x^2
x = 1 ./ Narr
xh = LinRange(0, 1.1/minimum(Narr), 1000)

intercept = []

plt = plot()
for (gind, g) in enumerate(garr)
  y = ΔE[:, gind]
  fitobj = curve_fit(model, x, y, [0.0,0.0,0.0])
  scatter!(x, y, label=@sprintf("g = %1.3f", g), color=palette(:default)[gind])
  plot!(xh, model(xh, coef(fitobj)), label=nothing, color=palette(:
 ↪default)[gind])
  push!(intercept, coef(fitobj)[1])
end
plot!(xlims=(0, Inf), ylims=(0, Inf))
xlabel!("1/N")
ylabel!("ΔE")
```

[13]:



As can be seen, for $g < g_c$ the intercept is basically 0, while for $g > g_c$ the intercept is finite and non-zero:

[14]:
```
for (gind, g) in enumerate(garr)
    @printf("For g=%.3f, Intercept is %.3f\n", g, intercept[gind])
end
```

```
For g=0.000, Intercept is 0.015
For g=0.241, Intercept is 0.037
For g=0.490, Intercept is 0.486
For g=0.500, Intercept is 0.535
```

4 Question 3: Singlet Gap

Now we want to perform a similar experiment, but for the gap in the singlet state. We use almost the same formalism:

[15]:
```
Narr = 4:2:Nmax
J1 = 1.0

E0 = zeros(length(Narr), length(garr))
E1 = zeros(length(Narr), length(garr))
for (Nind, N) in enumerate(Narr)
  b = fixed_sz_basis(N, Int(N/2))
```

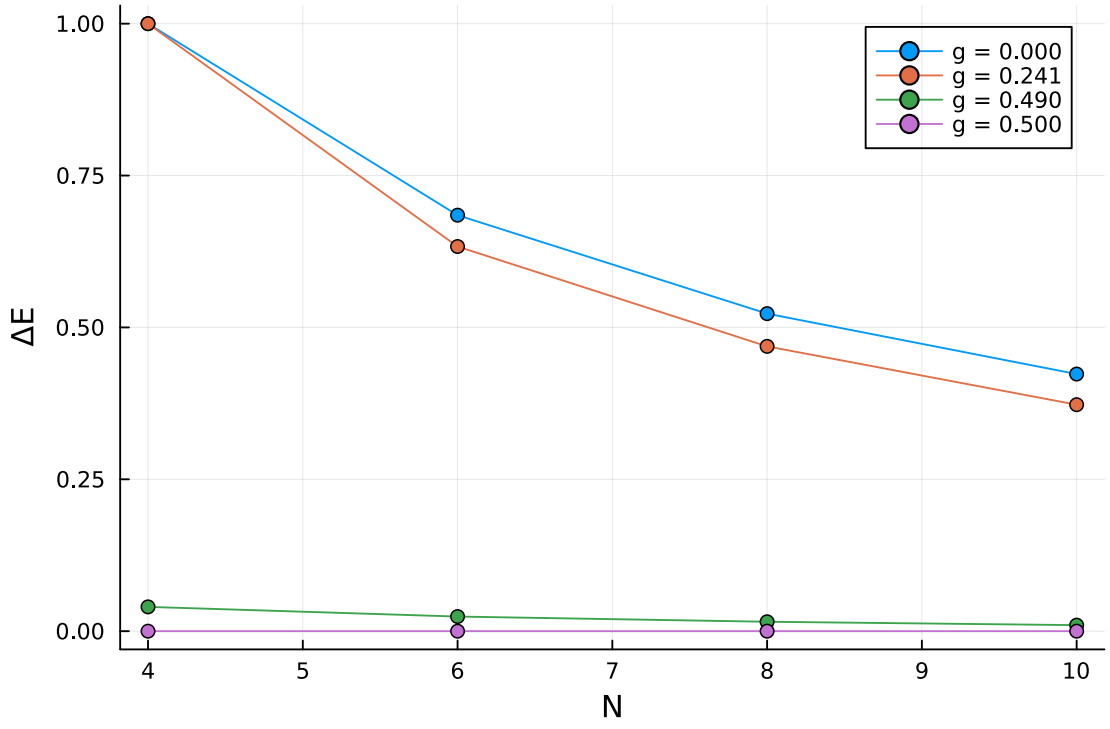
```
  for (gind, g) in enumerate(garr)
    J2 = g*J1
    H = construct_g_hamiltonian(b, J1, J2)

    Elist = eigs(H; nev=2, which=:SR, ritzvec=false)[1]
    E0[Nind, gind] = Elist[1]
    E1[Nind, gind] = Elist[2]
  end
end
ΔE = E1 .- E0
```

[15]:
```
4×4 Matrix{Float64}:
 1.0       1.0       0.04        8.88178e-16
 0.684742  0.63317   0.0240572   4.44089e-16
 0.522674  0.468812  0.0155083   4.88498e-15
 0.423239  0.372804  0.00994236  6.21725e-15
```

And plot it, but now with a separate plot for each values of g:

[16]:
```
plt = plot()
for (gind, g) in enumerate(garr)
  plot!(Narr, ΔE[:, gind], marker=:circle, label=@sprintf("g = %.3f", g) )
end
xlabel!("N")
ylabel!("ΔE")
```

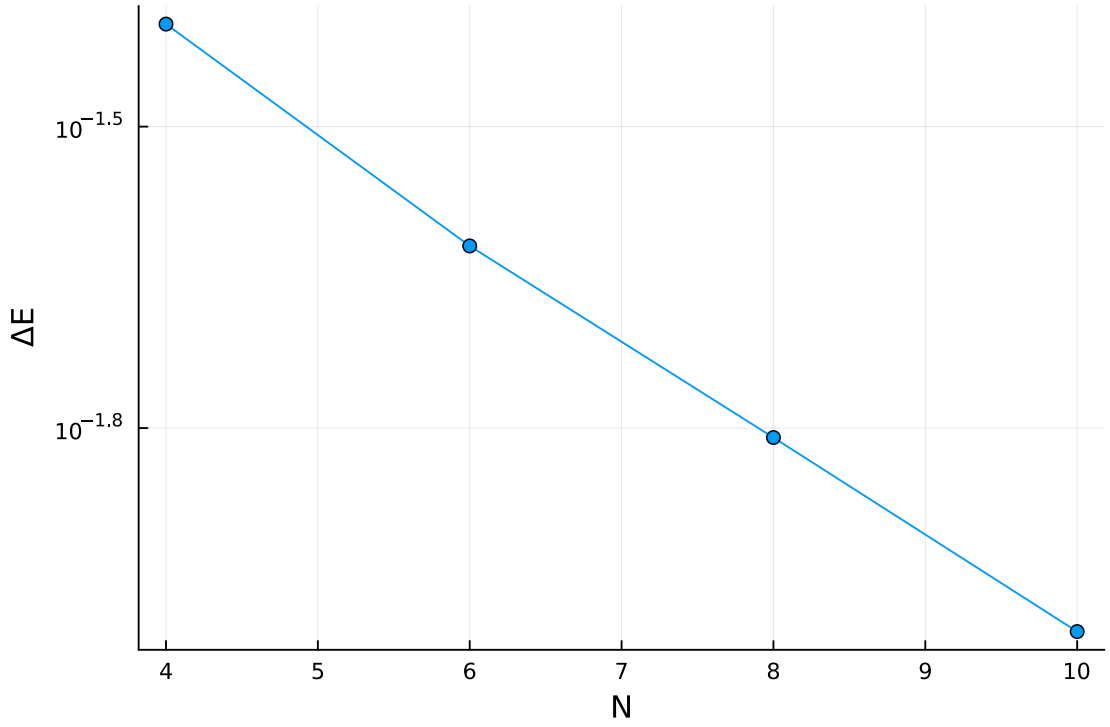
[16]:



To verify that in the case $g > g_c$ the relation is exponential we will plot it specifically in a semilog-y plot:

[17]:
```
plot(Narr, ΔE[:,3], marker=:circle, yaxis=:log, label=nothing)
xlabel!("N")
ylabel!("ΔE")
```

[17]:



And as we can see, this is exactly a linear relaition as expected.

5 Question 4: Spin-Spin Correlations

We first write a function which takes a wavefunction $|\psi\rangle$ and a basis b and returns the spin-spin correlation $\langle S_i S_{1+x} \rangle$:

[18]:
```
function spinspin_correlation( ::Vector{<:Number}, b::fixed_sz_basis, N::
 ↪Integer, x::Integer)
 @assert abs(norm( ) - 1) < 1e-9 "Input state is not normalized"
 @assert mod(N,2) == 0 && N > 0 "Number of spins must be positive and even"

 corr = 0
 for (substateind, coeff) in enumerate( )
```

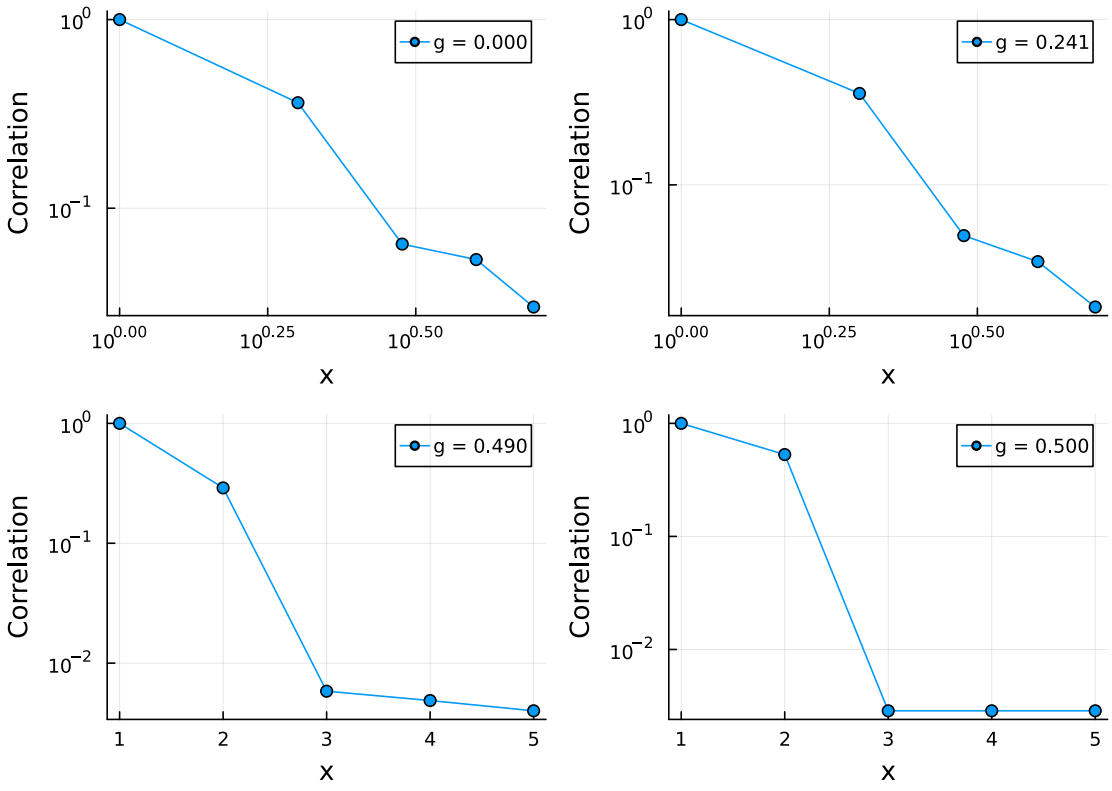
```
    stateind = b.states[substateind] # <1>
    state = 2 .* index2state(stateind, N) .- 1
    corr += abs(coeff)^2 * state[1] * state[mod(x,N)+1]
  end
  return corr
end
```

[18]:
```
spinspin_correlation (generic function with 1 method)
```

We will test the Spin-Spin correlation on the ground state of several Hamiltonians, each for a different value of g. To get the ground state, we will just take the 2nd output of the `eigs` function from `Arpack`:

[19]:
```
N = Nmax
D = 2^N
J1 = 1.0

xarr = range(0, length=Int(N/2))
corr = zeros(length(garr), Int(N/2))

plt_list = []
l = @layout [a b; c d]
for (gind, g) in enumerate(garr)
  J2 = g*J1
  b = fixed_sz_basis(N, Int(N/2))
  H = construct_g_hamiltonian(b, J1, J2)

  _,   = eigs(H; nev=1, which=:SR)
    = vec( )

  for x in xarr
    corr[gind, x+1] = spinspin_correlation( , b, N, x)
  end

  plt = plot(xarr.+1, corr[gind, :].^2, label=@sprintf("g = %1.3f", g),
    marker=:circle, xlabel="x", ylabel="Correlation")
  plot!(yaxis=:log, size=(250,200))

  if g <= gc
    plot!(xaxis=:log)
  end

  push!(plt_list, plt)
end
plot(plt_list[1], plt_list[2], plt_list[3], plt_list[4], layout=l, size=(700,␣
↪500))
```

[19]:



We can see that indeed the correlation decays as a power law for $g \leq g_c$ a expected, but for $g > g_c$ and $g = 1/2$ we get unexpected behavior: for $g = 1/2$ the correlation stops decaying and saturates to a given level (what should only happen when talking about bond-bond correlations, I think), and the in the $1/2 > g > g_c$ we see a middle state between a power law and saturation.

6 Question 5: Bond-Bond Correlations

Next we want to calculate the bond-bind correlation on the ground state in each case. First, we write a function which calculates the correlation, similar to the one from Question 4:

[20]:
```
function bondbond_correlation( ::Vector{<:Number}, b::fixed_sz_basis, N::
 ↪Integer, x::Integer)
 @assert abs(norm( ) - 1) < 1e-9 "Input state is not normalized"
 @assert mod(N,2) == 0 && N > 0 "Number of spins must be positive and even"

 corr = 0
 mean1 = 0
 mean2 = 0
 for (substateind, coeff) in enumerate( )
   stateind = b.states[substateind]
   state = 2 .* index2state(stateind, N) .- 1
```

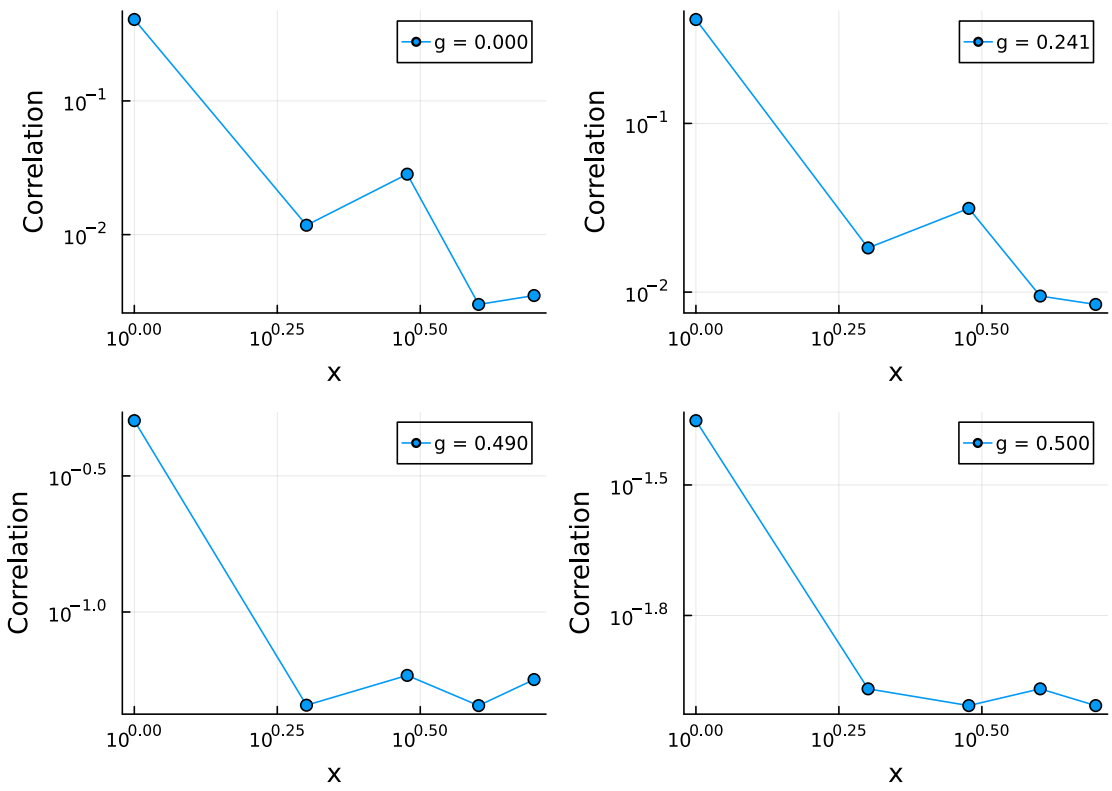
```
    bond1 = state[1] * state[2]
    bond2 = state[mod(x,N)+1] * state[mod(x+1,N)+1]
    corr += abs(coeff)^2 * bond1 * bond2
    mean1 += abs(coeff)^2 * bond1
    mean2 += abs(coeff)^2 * bond2
  end
  var = corr - mean1 * mean2
  return var
end
```

[20]:
```
bondbond_correlation (generic function with 1 method)
```

We test this function in the same manner we did in Question 4:

[21]:
```
N = Nmax
D = 2^N
J1 = 1.0

xarr = range(0, length=Int(N/2))
corr = zeros(length(garr), Int(N/2))

plt_list = []
l = @layout [a b; c d]
for (gind, g) in enumerate(garr)
  J2 = g*J1
  b = fixed_sz_basis(N, Int(N/2))
  H = construct_g_hamiltonian(b, J1, J2)

  _,   = eigs(H; nev=1, which=:SR)
    = vec( )

  for x in xarr
    corr[gind, x+1] = bondbond_correlation( , b, N, x)
  end
  plt = plot(xarr .+ 1, corr[gind, :].^2, label=@sprintf("g = %1.3f", g),␣
 ↪marker=:circle, xlabel="x", ylabel="Correlation")
  plot!(xaxis=:log, yaxis=:log, size=(250,200))
  push!(plt_list, plt)
end
plot(plt_list[1], plt_list[2], plt_list[3], plt_list[4], layout=l, size=(700,␣
 ↪500))
```

[21]:



We can see that for $g < g_c$ the correlation decays roughly as a power law, while for $g > g_c$ the correlation decays fast but saturates to a constant value for large x values, just as expected.