厳密対角化

1222031 北野志

2025 年 7 月 24 日

1　量子力学の基本的な計算を目的としたコード

コードの構成

```
module ExactDiag
const _dim = Ref(1)
const _site = Ref(1)
const _trans = Ref(Vector{Vector{Int}}())
const _reverse = Ref(Dict{Vector{Int}, Int}())
export # ここに外でも使う関数を列挙する
end
```

基本的には上のように module を定義してその中で関数を定義していく。(毎回関数に各サイトの次元_dim とサイト数_site と Fock 状態の番号を渡すのは面倒なため)

用いるパッケージ

```
using LinearAlgebra
using PrettyTables
using SparseArrays
using UnionFind
```

厳密対角化ではいくつかの線形代数の関数を用いるため LinearAlgebra を、メモリの消費量を減らすのを目的として行列のスパース表現を用いるため SparseArrays を、行列の表示をきれいにするため PrettyTables を、UnionFind はブロック対角化のために用いる。

初期化の関数

```
function init(dim::Int, site::Int)
  _dim[] = dim
  _site[] = site
```

```
  dim_tot = dim^site
  _trans[]=Vector{Vector{UInt8}}(undef, dim_tot)
  row = zeros(Int, site)
  _trans[][1] = copy(row)
  @inbounds for i in 1:dim_tot-1
    j = site
    while true
      v = row[j] + 1
      if v == dim
        row[j] = 0
        j -= 1
      else
        row[j] = v
        break
      end
    end
    _trans[][i+1] = copy(row)
  end
  _reverse[] = Dict{Vector{Int}, Int}(_trans[][i] => i for i in eachindex(_trans
    []))
end
```

各サイトの次元_dim(スピンの大きさ s として、$2s+1$) とサイト数_site を指定して初期化する。
さらに加えて、Fock 状態の番号付けを行うための配列_ reverse とその逆変換を行うための配列_trans とを定義する。(メモしておくことで計算量を削減できる)

Fock 状態の番号付け

```
function Nary_reverse(n::Vector{Int})
  if haskey(_reverse[], n)
    return copy(_reverse[][n])
  elseif length(n)!=_site[]
    throw(ArgumentError("n must be a vector of length site"))
  else
    throw(ArgumentError("n is not a valid state"))
  end
end
```

この関数は、Fock 状態を_site 進数で表現したときの整数値を 1-index で返すことにより番号付けしている。
たとえば、サイト数が 3 で次元が 2 のとき、状態 $|001\rangle$ は $1+0\cdot 2^2+0\cdot 2^1+1\cdot 2^0=2$、$|101\rangle$ は $1+1\cdot 2^2+0\cdot 2^1+1\cdot 2^0=6$ となる。

```
function Nary_trans(t::Int)
  if t == 0
    return [-1 for _ in 1:_site[]]
  elseif t < 1 || t > _dim[]^_site[]
    throw(ArgumentError("t must be in the range [0, dim^site - 1]"))
  else
    return copy(_trans[][t])
  end
end
```

逆に整数値から Fock 状態の表示に戻す。
引数が 0 だとすべてのサイトが-1 を返す仕様である。(変更の可能性あり)

関数の定義とその演算

```
struct Op
  op::Vector{Tuple{ComplexF64,Vector{Function}}}
  function Op(op::Vector{Tuple{ComplexF64,Vector{Function}}})
    new(op)
  end
  function Op(op1::Function)
    vecf = Vector{Function}([op1])
    Op([(1.0 + 0.0im, vecf)])
  end
end
```

このように演算子を定義する。Vector{Function}はその要素である Function の積を表しており、Tuple{ComplexF64,Vector{Function}}の ComplexF64 が Vector{Function}の係数を表しており、Vector{Tuple{ComplexF64,Vector{Function}}}の要素である Tuple{ComplexF64,Vector{Function}}の総和を表している。
例えば、Function として A, B, C, D, E を定義すると $\{(1.0 + 0.0i, \{A, B\}), (1.5 + 0.0i, \{C, D\}), (0.5 + 0.0i, \{E\})\}$ は $AB + 1.5CD + 0.5E$ を表している。

```
import Base: *, +, -, show, sum
```

演算子に用いる記号のオーバーロードを行う。

```
function +(ops::Op...)
  k = Vector{Tuple{ComplexF64,Vector{Function}}}()
  for op1 in ops
    k = vcat(k, op1.op)
  end
  Op(k)
end
```

演算子同士の足し算は演算子のベクトルを連結することで定義する。

```
function *(op1::Op)
  op1
end
function *(op1::Op, op2::Op...)
  op3 = *(op2...)
  k = Vector{Tuple{ComplexF64,Vector{Function}}}()
  for op11 in op1.op
    for op31 in op3.op
      push!(k, (op11[1] * op31[1], vcat(op11[2], op31[2])))
    end
  end
  Op(k)
end
```

演算子同士の掛け算は分配法則を用いて帰納的に定義されている。

```
function *(coeff::Union{ComplexF64,Float64}, op1::Op...)
  op2 = *(op1...)
  k = Vector{Tuple{ComplexF64,Vector{Function}}}()
  for op21 in op2.op
    push!(k, (op21[1] * ComplexF64(coeff), op21[2]))
  end
  Op(k)
end
```

通常の数 (自前で書くときに面倒なので複素数と実数どちらでも良くしている) と演算子の掛け算は係数にその数を掛けることによって定義されている。

```
function *(op1::Op, t::Int)
  t1 = copy(t)
  for op11 in op1.op[1][2]
    t1 = op11(t1)[2]
  end
  sum = 0.0 + 0.0im
  for op11 in op1.op
    product = op11[1]
    t2 = copy(t)
    for op12 in op11[2]
      t2 = op12(t2)[2]
      product *= op12(t2)[1]
    end
    if t2 != t1
      throw(ArgumentError("The operator does not preserve the state."))
    end
    sum += product
  end
  return (sum, t1)
end
```

整数で表される Fock 状態に対して演算子を作用させる。
この場合は、作用させた後に単一の Fock 状態に戻る場合のみに用いることができる。(そうでなければエラーが出る仕様)

```
function -(op1::Op)
  (-1.0 + 0.0im) * op1
end
function -(op1::Op, op2::Op)
  op1 + (-op2)
end
function -(op1::Op, op2::Op...)
  op3 = +(op2...)
  op1 - op3
end
```

演算子同士の引き算は通常の数との積を用いて帰納的に定義されている。

```
function sum(mats::Op...)
  ans = mats[1]
  for mat in mats[2:end]
    ans += mat
  end
  ans
end
function sum(f::Function, k::Int=0)
  sum_j(Tuple(f(i) for i in 1:(_site[]-k))...)
```

```
end
```

演算子の和を計算する関数であり、引数の演算子の総和を求められる。
下の関数の適用例として

```
sum_j(i->f(i))
```

上のように表記することで、サイト数で総和の範囲を指定することもできる。

```
sum_j(i->f(i), k)
```

上のように表記することで、サイト数から k を引いた値までの総和を求めることもできる。(開放端条件などのために用いられる)

具体的な演算子の定義

作用させた後に単一の Fock 状態に戻る演算子のみを定義する。

```
function id(i::Int)
  if i < 1 || i > _dim[]^_site[]
    throw(ArgumentError("i must be in the range [1, dim^site]"))
  end
  return (1.0 + 0.0im, i)
end
id() = Op(id)
```

恒等演算子は Fock 状態をそのまま係数 1 で返すと定義されている。(範囲外の場合はエラーが出る)

```
function shift(k::Int=1)
  function shift1(t::Int)
    n = Nary_trans(t)
    n1 = circshift(n, k)
    return (1.0 + 0.0im, Nary_reverse(n1))
  end
  return Op(shift1)
end
```

並進演算子は Fock 状態を k だけ右にずらすと定義されている。(デフォルトでは 1 だけずらす)

```
function site_flip(t::Int)
  n = Nary_trans(t)
  n1 = reverse(n)
  return (1.0 + 0.0im, Nary_reverse(n1))
```

```
end
site_flip() = Op(site_flip)
```

サイト反転演算子は Fock 状態を反転させると定義されている。

```
function spin_flip(t::Int)
  n = Nary_trans(t)
  n1 = Vector{Int}(undef, _site[])
  for i in 1:_site[]
    n1[i] = _dim[]/2.0 - n[i]
  end
  return (1.0 + 0.0im, Nary_reverse(n1))
end
spin_flip() = Op(spin_flip)
```

スピン反転演算子は Fock 状態の各サイトのスピンを反転させると定義されている。(スピンの大きさが 1/2 の場合のみ定義されている)

```
function spin(kind::Char, site::Int)
  site_number = _site[]
  idx = (site - 1) % site_number + 1
  if kind == '+'
    plus = function (t::Int)
      n = Nary_trans(t)
      n[idx] += 1
      return n[idx] < _dim[] ? (1.0 + 0im, Nary_reverse(n)) : (0.0 + 0im, t)
    end
    return Op(plus)
  elseif kind == '-'
    minus = function (t::Int)
      n = Nary_trans(t)
      n[idx] -= 1
      return n[idx] > -1 ? (1.0 + 0im, Nary_reverse(n)) : (0.0 + 0im, t)
    end
    return Op(minus)
  elseif kind == 'x'
    plus = function (t::Int)
      n = Nary_trans(t)
      n[idx] += 1
      n[idx] < _dim[] ? (1.0 + 0im, Nary_reverse(n)) : (0.0 + 0im, t)
    end
    minus = function (t::Int)
      n = Nary_trans(t)
      n[idx] -= 1
      n[idx] > -1 ? (1.0 + 0im, Nary_reverse(n)) : (0.0 + 0im, t)
    end
    return 0.5 * (Op(plus) + Op(minus))
  elseif kind == 'y'
    plus = function (t::Int)
      n = Nary_trans(t)
      n[idx] += 1
      n[idx] < _dim[] ? (1.0 + 0im, Nary_reverse(n)) : (0.0 + 0im, t)
    end
    minus = function (t::Int)
      n = Nary_trans(t)
      n[idx] -= 1
      n[idx] > -1 ? (1.0 + 0im, Nary_reverse(n)) : (0.0 + 0im, t)
    end
    return (-0.5im) * (Op(plus) - Op(minus))
  elseif kind == 'z'
    z = function (t::Int)
      n = Nary_trans(t)
      ((_dim[] - 1.0) / 2.0 - n[idx] + 0.0im, t)
```

```
    end
    return Op(z)
  else
    throw(ArgumentError("kind must be '+', '-', 'x', 'y', or 'z'"))
  end
end
```

スピン演算子はスピンの種類とサイト番号を指定して定義される。
スピンの種類は $+$, $-$, z, x, y の 5 種類がある。
スピンの種類 $+$ はスピンを 1 つ上げる演算子、$-$ はスピンを 1 つ下げる演算子、z はスピンの z 成分を返す演算子、x はスピンを 1 つ上げる演算子と 1 つ下げる演算子の和の半分、y はスピンを 1 つ上げる演算子と 1 つ下げる演算子の差の虚数倍の半分である。(普通に関数を定義するとエラーが出るので、無名関数を用いて定義しておく)

```
function num(site::Int)
  return ((_dim[]-1.0)/2.0)*id()-spin('z',site)
end
```

Jordan-Wigner 変換を用いて、スピンの z 成分を数演算子に変換するための関数である。

```
function S_z()
  return sum(j->spin('z', j))
end
```

スピンの z 成分を全てのサイトに対して和を取った演算子である。

ハミルトニアンの行列とその表示

```
function matrix(op1::Union{Matrix{ComplexF64},Op})
  if isa(op1, Matrix{ComplexF64})
    return op1
  end
  dimention = _dim[]
  site_number = _site[]
  dim_tot = dimention^site_number
  mat = zeros(ComplexF64, dim_tot, dim_tot)
  for t in 1:dim_tot
    for (coeff, op11) in op1.op
      t1 = t
      coeff1 = coeff
      for op12 in op11
        (v1, t1) = op12(t1)
        coeff1 *= v1
      end
      mat[t, t1] += coeff1
    end
  end
  return mat
end
```

ハミルトニアンの行列を計算する関数であり、それぞれの演算子要素ごとに関数を作用させていってゼロ行列に加えていくことで行列が生成される。

```
function complex_formatter(; digits::Int=1)
  return (v, i, j) -> begin
    if v == 0 + 0im
      @sprintf("%.*f", digits, 0.0)
    elseif isa(v, Complex)
      rea = round(real(v), digits=digits)
      image = round(imag(v), digits=digits)
      if image == 0
        @sprintf("%.*f", digits, rea)
      elseif rea == 0
        @sprintf("%.*fim", digits, image)
      else
        sign = image > 0 ? "+" : "-"
        @sprintf("%.*f%s%.*fim", digits, rea, sign, digits, abs(image))
      end
    elseif isa(v, Number)
      @sprintf("%.*f", digits, v)
    else
      string(v)
    end
  end
end
```

複素数を指定した桁数で表示させるためのフォーマットであり、行列の表示に用いられる。中身はよくわかっていない。

```
function show(op1::Op, digit::Int=1)
  mat1 = matrix(op1)
  pretty_table(mat1, header=([join(string.(Nary_trans(i)), "") for i in 1:size(
      mat1, 1)]), row_labels=([join(string.(Nary_trans(i)), "") for i in 1:size(
      mat1, 2)]), formatters=complex_formatter(digits=digit))
end
```

Fock 状態による行列を表示できる関数である。

代表的なハミルトニアンの表示

　これまでの演算子の関数を用いて代表的なハミルトニアンを表示してみる。
まず、初期化をしておく。

```
init(2, 4)
```

スピン $\frac{1}{2}$ の 4 サイトの系を考える。
横磁場イジング模型 (周期境界条件)

$$H = \sum_{j=1}^{L} \left(S_j^z S_{j+1}^z - h S_j^x \right)$$

このハミルトニアンは J を用いて無次元化して

```
hj = 2.0
H1 = sum(j -> spin('z', j) * spin('z', j + 1) - hj * spin('x', j))
```

と表せる。(hj は縦磁場の強さを表す $\frac{h}{J}$ であるがどんな値でもよい)
開放端条件ならば

```
H1 = sum(j -> spin('z', j) * spin('z', j + 1), 1) - sum(j -> hj * spin('x', j))
```

と表される。
XXZ 模型 (周期境界条件)

$$H = \sum_{j=1}^{L} \left(S_j^x S_{j+1}^x + S_j^y S_{j+1}^y + \Delta S_j^z S_{j+1}^z \right)$$

```
Δ = 2.0
H2 = sum(j -> spin('x', j) * spin('x', j + 1) + spin('y', j) * spin('y', j + 1)
     + Δ * spin('z', j) * spin('z', j + 1))
```

と表せる。
開放端条件ならば

```
H2 = sum(j -> spin('x', j) * spin('x', j + 1) + spin('y', j) * spin('y', j + 1)
     + Δ * spin('z', j) * spin('z', j + 1), 1)
```

と表される。
Bose-Hubbard 模型 (開放端条件)

$$H = -J \sum_{j=1}^{L-1} \left(a_j^\dagger a_{j+1} + a_{j+1}^\dagger a_j \right) + \frac{U}{2} \sum_{j=1}^{L} n_j (n_j - 1)$$

```
Uj = 2.0
H3 = sum(j -> -spin('+', j) * spin('-', j + 1) + -spin('-', j) * spin('+', j + 1
     ), 1) + sum(j -> (Uj / 2.0) * num(j) * (num(j) - 1))
```

と表せる。(Jordan-Wigner 変換を用いている)
Bose-Hubbard 模型においては周期境界条件で Jordan-Wigner 変換を用いると非自明な項が出てくるため、開放端条件で定義している。

2　固有状態とそれを基底としたハミルトニアンの計算方法

ここでは少し一般の系を扱う。

2.1　無限の場合の固有状態

$\hat{A}$ と任意の状態 $|\varphi\rangle$ について
固有値 α の固有状態 $|\alpha\rangle$ は

$$|\alpha\rangle = C \sum_{n=-\infty}^{\infty} \left(\frac{\hat{A}}{\alpha}\right)^n |\varphi\rangle$$

と表せる。(C は規格化係数)
ここで、$\hat{A}^{-n}|\varphi\rangle (n \geq 0)$ は $|\varphi\rangle = \hat{A}^0|\varphi\rangle$ であり、$n = k$ として $\hat{A}^{-k}|\varphi\rangle$ が定義された場合に状態 $\hat{A}^{-(k+1)}|\varphi\rangle$ は、$\hat{A}^{-k}|\varphi\rangle = \hat{A}(\hat{A}^{-(k+1)}|\phi\rangle)$ を満たすとして、再帰的に定義する。(一意に決まるかどうかはわからないが、定義を満たす状態を用いればよい)
ただし、$\||\alpha\rangle\| < \infty$ とする。

証明

$$\begin{aligned}
\hat{A}|\alpha\rangle &= C \sum_{n=-\infty}^{\infty} \alpha^{-n} \hat{A}^{n+1} |\varphi\rangle \\
&= C \sum_{n=-\infty}^{\infty} \alpha^{-n+1} \hat{A}^{n} |\varphi\rangle \\
&= \alpha C \sum_{n=-\infty}^{\infty} \left(\frac{\hat{A}}{\alpha}\right)^n |\varphi\rangle \\
&= \alpha |\alpha\rangle
\end{aligned}$$

具体例

調和振動子について、消滅演算子 $\hat{A} = \hat{a}$ と状態 $|0\rangle$ を用いて

$$\begin{aligned}
|\alpha\rangle &= C \sum_{n=-\infty}^{\infty} \left(\frac{\hat{a}}{\alpha}\right)^n |0\rangle \\
&= C \sum_{n=-\infty}^{0} \left(\frac{\hat{a}}{\alpha}\right)^n |0\rangle \\
&= C \sum_{n=0}^{\infty} \alpha^n \hat{a}^{-n} |0\rangle \\
&= C \sum_{n=0}^{\infty} \frac{\alpha^n}{\sqrt{n!}} |n\rangle
\end{aligned}$$

これは収束する。

2.2 有限の場合の固有状態

ある n について、$\hat{A}^n |\varphi\rangle = \beta |\varphi\rangle$ を満たす演算子 $\hat{A}$ と状態 $|\varphi\rangle$ について
固有値は $\alpha(\beta$ の n 乗根) の固有状態 $|\alpha\rangle$ は

$$|\alpha\rangle = C \sum_{k=0}^{n-1} \left(\frac{\hat{A}}{\alpha}\right)^k |\varphi\rangle$$

証明

$$\begin{aligned}
\hat{A} |\alpha\rangle &= C \sum_{k=0}^{n-1} \alpha^{-k} \hat{A}^{k+1} |\varphi\rangle \\
&= C \sum_{k=1}^{n} \alpha^{-k+1} \hat{A}^k |\varphi\rangle \\
&= \alpha C \sum_{k=1}^{n} \left(\frac{\hat{A}}{\alpha}\right)^k |\varphi\rangle \\
&= \alpha C \sum_{k=1}^{n-1} \left(\frac{\hat{A}}{\alpha}\right)^k |\varphi\rangle + \left(\frac{\hat{A}}{\alpha}\right)^n |\varphi\rangle \\
&= \alpha C \sum_{k=0}^{n-1} \left(\frac{\hat{A}}{\alpha}\right)^k |\varphi\rangle \\
&= \alpha |\alpha\rangle
\end{aligned}$$

具体例

並進演算子 $\hat{A}=\hat{T}$ と状態 $|\varphi\rangle=|001\rangle$ を用いて、$\hat{T}^3|001\rangle=|001\rangle$ より
$x^3=1$ の解 $1, e^{\frac{2}{3}\pi i}, e^{\frac{4}{3}\pi i}$ であるため

$$
\begin{aligned}
|1\rangle &= \frac{1}{\sqrt{3}}\left(|001\rangle+|100\rangle+|010\rangle\right)\\
\left|e^{\frac{2}{3}\pi i}\right\rangle &= \frac{1}{\sqrt{3}}\left(|001\rangle+e^{\frac{4}{3}\pi i}|100\rangle+e^{\frac{2}{3}\pi i}|010\rangle\right)\\
\left|e^{\frac{4}{3}\pi i}\right\rangle &= \frac{1}{\sqrt{3}}\left(|001\rangle+e^{\frac{2}{3}\pi i}|100\rangle+e^{\frac{4}{3}\pi i}|010\rangle\right)
\end{aligned}
$$

2.3 ハミルトニアンの行列成分

ここまでで作った固有状態を用いてブロック対角化されたハミルトニアンの行列成分を計算する。
完全正規直交基底 $\{|\varphi_n\rangle\}_{n=1}^{N}$ を用いて、演算子 $\hat{A}$ はある自然数 m と任意の自然数 n に対して複素数 α_n が

$$\hat{A}^m|\varphi_n\rangle=\alpha_n|\varphi_n\rangle$$

を満たすとする。(m はこの条件を満たす自然数のうち最小の自然数とする)
この基底 $|\varphi_n\rangle$ と演算子 $\hat{A}$ を用いて固有状態が生成できるが、同じ固有状態を生成する基底を集めた集合を考え、そのうち n の値が最小の基底 $|\varphi_{\bar{n}}\rangle$ を代表元とする。
固有値 β($\alpha_{\bar{n}}$ の n 乗根) を持つ固有状態は

$$|\bar{n};\beta\rangle=\frac{1}{K_{\bar{n},\beta}}\sum_{k=0}^{m-1}\left(\frac{\hat{A}}{\beta}\right)^k|\varphi_{\bar{n}}\rangle$$

と表せる。
ただし、

$$K_{n,\beta}=\left\|\sum_{k=0}^{m-1}\left(\frac{\hat{A}}{\beta}\right)^k|\varphi_n\rangle\right\|$$

とする。
ここで、$\forall n:\hat{A}|\varphi_n\rangle=a|\varphi_k\rangle\,(a\in\mathbb{R})$ と表せることは仮定しておく。
したがって、ハミルトニアンを左からかけると

$$
\begin{aligned}
\hat{H}|\bar{n};\beta\rangle &= \frac{1}{K_{\bar{n},\beta}}\sum_{k=0}^{m-1}\left(\frac{\hat{A}}{\beta}\right)^k\hat{H}|\varphi_{\bar{n}}\rangle\\
&= \frac{1}{K_{\bar{n},\beta}}\sum_{k=0}^{m-1}\sum_{l=0}^{N}\left(\frac{\hat{A}}{\beta}\right)^k|\varphi_l\rangle\langle\varphi_l|\hat{H}|\varphi_{\bar{n}}\rangle\\
&= \frac{1}{K_{\bar{n},\beta}}\sum_{l=0}^{N}H_{l,\bar{n}}\sum_{k=0}^{m-1}\left(\frac{\hat{A}}{\beta}\right)^k|\varphi_l\rangle\\
&= \sum_{l=0}^{N}\frac{K_{l,\beta}}{K_{\bar{n},\beta}}\left(\frac{|\beta|}{\beta}\right)^{d(l)}H_{l,\bar{n}}|\bar{l};\beta\rangle
\end{aligned}
$$

ここで、$d(l)$ は

$$\hat{A}^{d(n)} |\varphi_n\rangle = b |\varphi_{\bar{n}}\rangle$$

を満たすとする。

3　エンタングルメントエントロピーの計算結果

縦磁場横磁場イジングモデル (開放端)—空間反転のみ

$$\hat{H} = J \sum_{i=1}^{N-1} \hat{S}_i^z \hat{S}_{i+1}^z - \frac{h}{2} \sum_{i=1}^{N} \left(\hat{S}_i^+ + \hat{S}_i^- \right) - v \sum_{i=1}^{N} \hat{S}_i^z$$

$$\frac{h}{J} = \sqrt{1.5}$$

$$\frac{v}{J} = \sqrt{2.0}$$

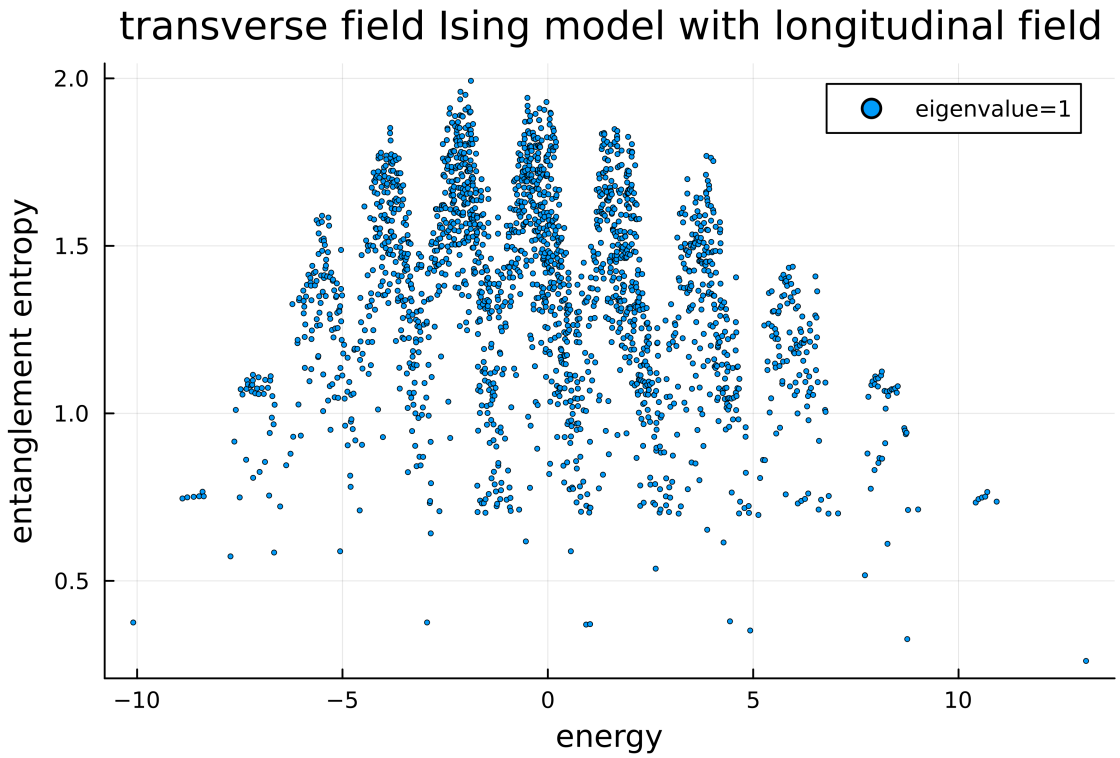

図 1　縦磁場を入れた横磁場イジングモデル

ここで、$\frac{h}{J} = 10000$ とすると。

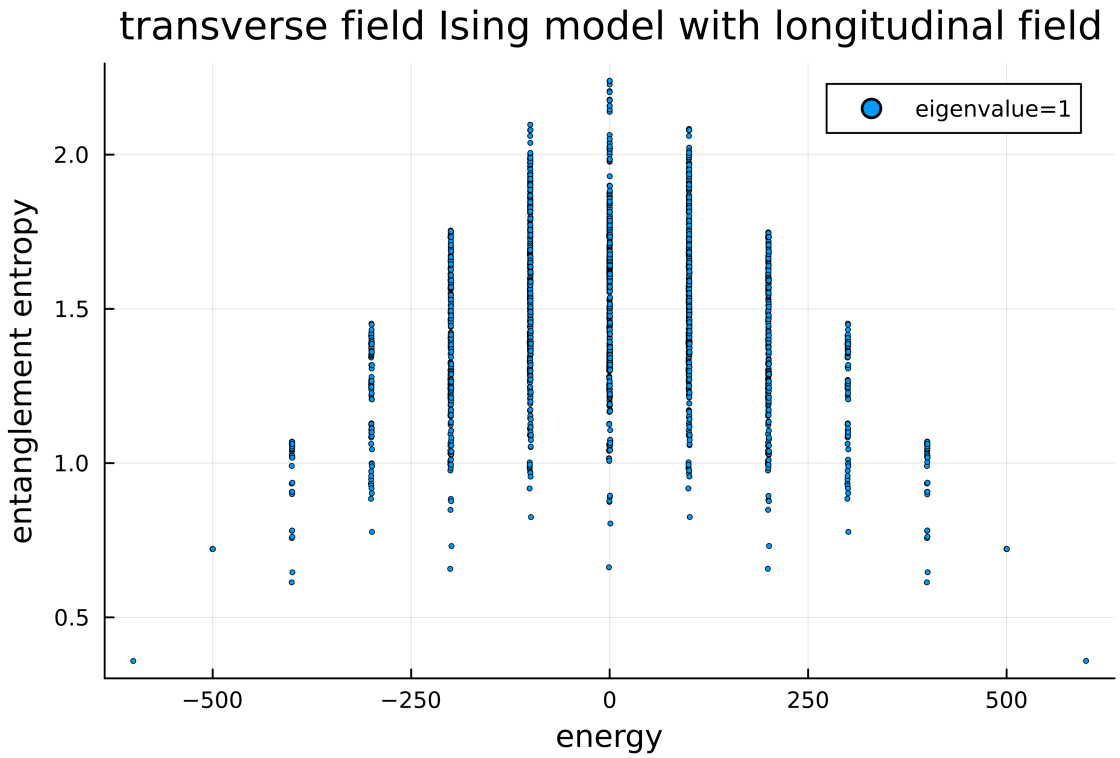

図 2　縦磁場を入れた横磁場イジングモデル

このようにほぼ縮退しており、x 方向のスピンにより分解されている。

横磁場イジングモデル (開放端)—空間反転のみ

$$\hat{H} = J\sum_{i=1}^{N-2}\hat{S}_i^z\hat{S}_{i+1}^z + k\hat{S}_{N-1}^z\hat{S}_N^z - \frac{h}{2}\sum_{i=1}^{N}\left(\hat{S}_i^+ + \hat{S}_i^-\right)$$

$$\frac{h}{J} = \sqrt{1.5}$$

$$\frac{k}{J} = \sqrt{2.5}$$

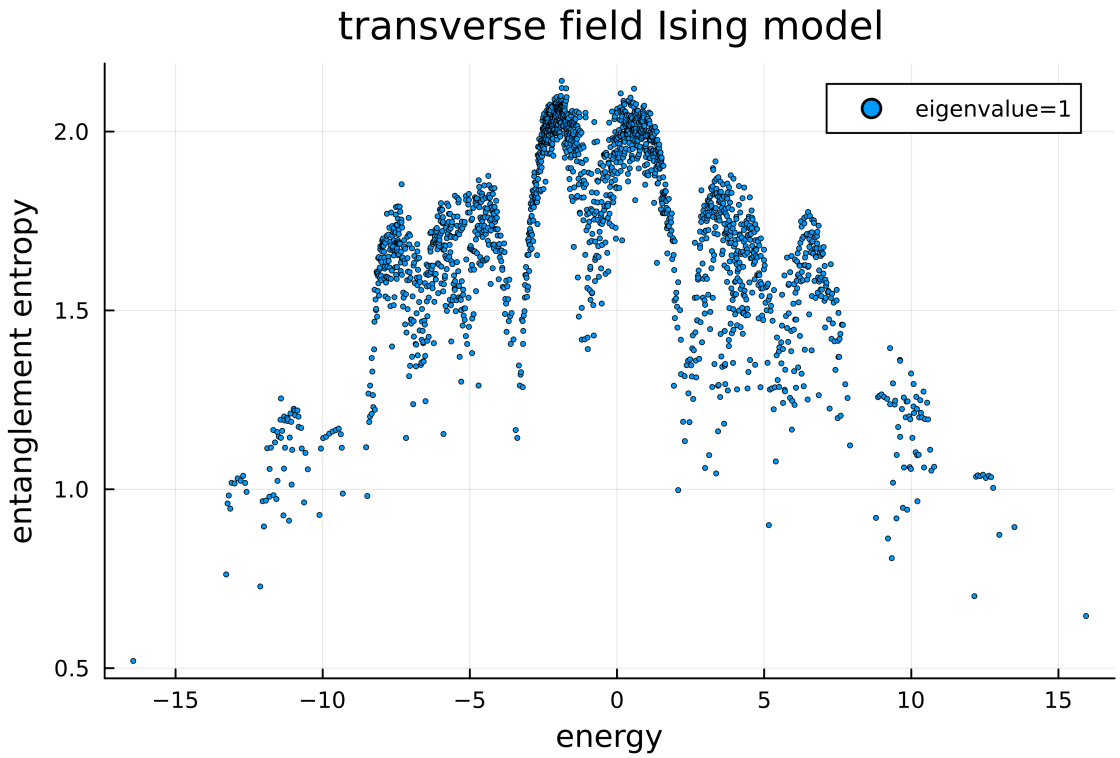

図 3　横磁場イジングモデル

サイン関数による縦磁場勾配を加えた XXZ 模型 (開放端)—U(1) 対称性のみ

$$\hat{H} = J \sum_{i=1}^{N-1} \left(\frac{1}{2} (\hat{S}_i^+ \hat{S}_{i+1}^- + \hat{S}_i^- \hat{S}_{i+1}^+) + \Delta \hat{S}_i^z \hat{S}_{i+1}^z \right) + v \sum_{i=1}^{N} \sin(i) \hat{S}_i^z$$

$$\Delta = \sqrt{1.3}$$

$$\frac{v}{J} = \sqrt{2}$$

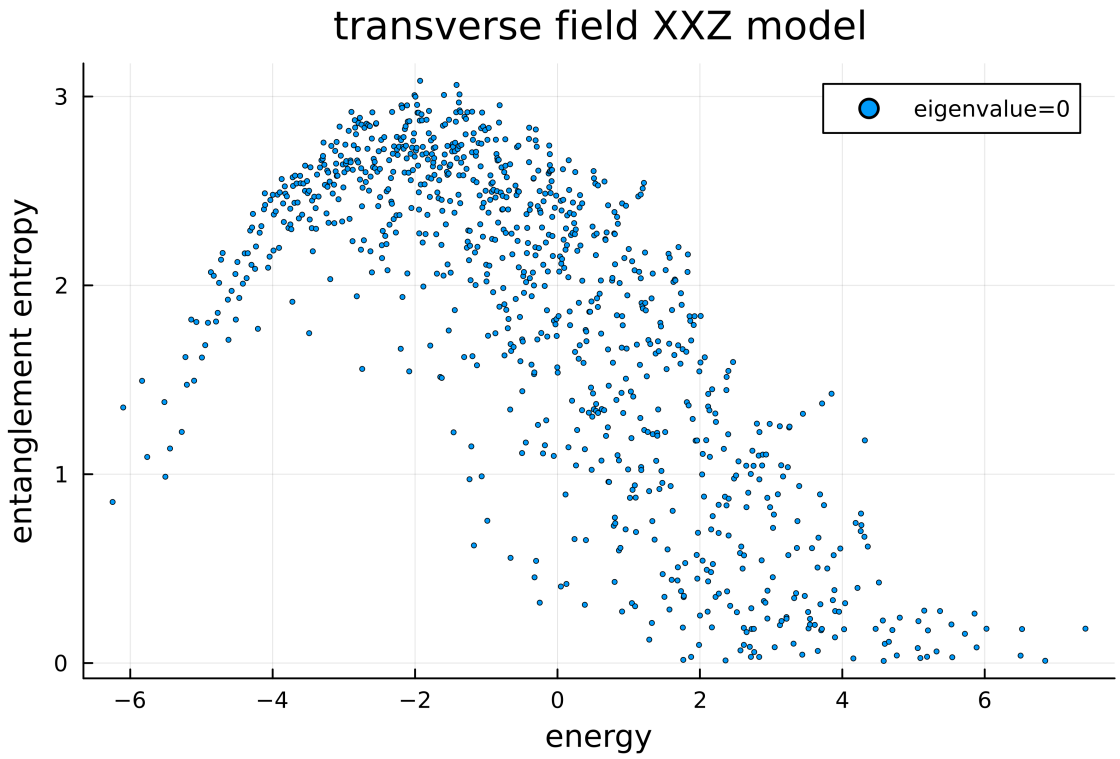

図 4　縦磁場を入れた XXZ 模型