Entanglement Studies in the Affleck–Kennedy–Lieb–Tasaki (AKLT) Spin 1 Biquadratic Model

Dissertation submitted to
Visvesvaraya National Institute of Technology, Nagpur
in partial fulfilment of the requirements for the award of degree of

Master of Science
In
Physics

By

Kaushal Joshi
(MS23PHY018)

under the supervision of

Dr. M. S. Ramkarthik
Assistant Professor
Department of Physics,
VNIT Nagpur

Department of Physics
Visvesvaraya National Institute of Technology, Nagpur
May, 2025.

To my guide

Dr. M. S. Ramkarthik,

for showing me the real

beauty of Physics.

&

To my ***parents*** *and* ***guardians***

for supporting me with

my decisions.

VISVESVARAYA NATIONAL INSTITUTE OF TECHNOLOGY, NAGPUR

Non-Plagiarism Certificate

Certified that M.Sc. Dissertation titled **"Entanglement Studies in the Affleck–Kennedy–Lieb–Tasaki (AKLT) Spin 1 Biquadratic Model"** submitted by **Kaushal Joshi**, Enrollment No. MS23PHY018, ID No. 30645, is original in nature and not copied from any literature material. Due credit has been given to the original researcher wherever referred.

Signature:

Name of the student: **Kaushal Joshi**

Enrollment No.: **MS23PHY018**

Signature:

Dr. M. S. Ramkarthik (Asst. Prof., Dept. of Physics, VNIT)

ACKNOWLEDGMENTS

At first, I would like to express my sincere gratitude to my professor and guide, ***Dr. M. S. Ramkarthik****, for allowing me to do this project work under his supervision. His command in several topics proves to me the fact that how important reading is for understanding and doing Physics. The way of understanding and questioning everything until you prove it is the best thing that I have learnt from him. His dedication towards his work and his proficiency in mentoring students give me inspiration on how to become an ideal teacher.*

I want to acknowledge my parents, ***Mrs. Mamta Joshi*** *and* ***Mr. Naveen Chandra Joshi*** *and my guardians,* ***Mrs. Vimla Pant****,* ***Mr. Kamlesh Pant*** *and* ***Mrs. Rekha Pant****, for their unconditional support and for having faith in my decision. Next, I would like to acknowledge my friend* ***Ankan Bhattacharyya*** *for helping me with the basics, whenever I felt stuck, I came to you, and it gave me the clarity on how to proceed further. He used to suggest to me the approaches that I can use while working on this project. I appreciate his support and the insightful discussions that we used to have during these two years of M.Sc.*

I want to acknowledge my group members, Gargee Arun Tammadwar, Divyani Vilas Bhagat, Durgesh Thail Vasant, Om Sanjay Shah, and Atul. The discussions that we all used to have together are priceless. Without them, the journey of Saturday talks and the project would have been flavourless. They're the flavours of the journey.

I also want to express my heartfelt gratitude to all of my friends for being a part of this short but beautiful journey.

ABSTRACT

Entanglement is one of the most fascinating concepts in quantum mechanics. It describes a situation where two or more particles become correlated in such a way that measuring the state of one particle instantly affects the state of the other without even measuring the other particle, no matter how far apart they are. This remarkable phenomenon has numerous applications, including teleportation, quantum cryptography etc..

In this project work, we have studied the entanglement in a class of one-dimensional Heisenberg chains known as the AKLT Model, which is a bilinear biquadratic spin chain model, named after Affleck-Kennedy-Lieb-Tasaki, who first gave the concept of the valence bond states in antiferromagnetic materials as an extension of the famous Majumdar-Ghosh model. We started with the Ising Model in a magnetic field (transverse and longitudinal) to develop an understanding of spin chains, and then studied Heisenberg model and Majumdar-Ghosh model to develop the foundations of the AKLT Model, Then we used the concept of valence bond states to derive our AKLT Hamiltonian using projectors. Due to the presence of biquadratic terms in the Hamiltonian, it became difficult to build the Hamiltonian for the higher number of qubits, so we used some advanced approaches like, (DSYEVX-LAPACK, Sparsification techniques, Parallel Programming using both CPU and GPU). On optimizing the computational approaches, we studied entanglement properties using concurrence (two-qubit measure) and von Neumann Entropy (multi-qubit measure) of the ground state by varying interaction energy and magnetic field, as these are the crucial parameters in the problem. The study of these two measures were done to see the trade off between the two-qubit and multi-qubit measures.

We have numerically studied entanglement by plotting concurrence and entropy for different numbers of qubits and analyzed it in detail, the results of which showed interesting saturation behaviour of entanglement with the increase in the number of qubits. We have found some interesting patterns in the behaviour of concurrence and entropy in the AKLT model.

List of Figures

List of Tables

Contents

Chapter 1

Introduction to Ising Model

1.1 History and Motivation

The Ising model is a mathematical model defined by Ernst Ising to explain the spontaneous magnetisation in a ferromagnetic material [1]. The motivation for defining this model is that in ferromagnets, all the magnetic moments are aligned in the same direction, which leads to strong magnetic properties. The ground state of this model is also the same as all the spins half particles are either up (i.e. $|\uparrow\uparrow .... \uparrow\rangle$) or down (i.e. $|\downarrow\downarrow .... \downarrow\rangle$) in that state. However, this model was not successful, but this model played a significant role in explaining a plethora of phenomena such as phase transitions, frustrated spin chains, etc, and in fact, it became one of the paradigms in physics.

1.2 The Hamiltonian

Let us consider a linear 1-D lattice with N sites consisting of spin half particles (i.e. either spin up (s = + $\frac{1}{2}$) or spin down (s = - $\frac{1}{2}$)), then the Hamiltonian ($\mathcal{H}$) is defined as,

$$\mathcal{H} = -J\sum_i \sigma_i^z \sigma_{i+1}^z - B\sum_i \sigma_i^\alpha, \tag{1.1}$$

where J is the interaction energy between each spin, B is the applied constant magnetic field and $\alpha = \{x,y,z\}$.

Case I: If $\alpha = Z$, then the magnetic field will be in the longitudinal direction [2].

Case II: If $\alpha = \{X,Y\}$, then the magnetic field will be in transverse direction [3].

Since we know the spin for each particle, $s = \frac{1}{2}$, we can calculate the number of basis states as

$$N = 2s + 1, \tag{1.2}$$

substitute the value of s in Eq. [1.2], the total number of basis states will be 2. The basis states for this system are $|0\rangle$ and $|1\rangle$, which are the eigenvectors of σ_z with eigenvalues 1 and -1 respectively [4].

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \tag{1.3}$$

In the Hamiltonian, for placing the lattice along different directions we can use the fact that σ_i^α are spin half operators acting on i^{th} spin, where σ^x, σ^y and σ^z are Pauli matrices,

$$\sigma^x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \sigma^y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \sigma^z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}. \tag{1.4}$$

And these Pauli matrices satisfy,

$$[\sigma_i^a, \sigma_j^b] = 2i\epsilon_{abc}\sigma_j^c\delta_{ij}, \tag{1.5}$$

where, ϵ_{abc} is the Levi Civita symbol

$$\epsilon_{abc} = \begin{cases} +1 & \text{if (a,b,c)} \equiv (1,2,3),(2,3,1),(3,1,2) \\ -1 & \text{if (a,b,c)} \equiv (3,2,1),(1,3,2),(2,1,3) \\ 0 & \text{if a = b, b = c, c = a} \end{cases}, \tag{1.6}$$

and, i and j represent the position of spin, and a,b, and c are the Pauli matrices (i.e. x, y, z).

This Hamiltonian can be used to define two types of systems based on boundary conditions.

Case I: Open Chain System

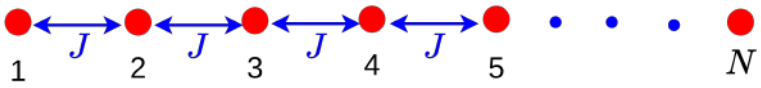


Figure 1.1: Open chain system

In this type of system, the spins present at the boundary of the lattice interact only with a single spin, while the rest of the spins interact with two corresponding neighbours. This case is not translational invariant. So the Hamiltonian can be written as,

$$\mathcal{H} = -J \sum_{i=1}^{N-1} \sigma_i^z \sigma_{i+1}^z - B \sum_{i=1}^{N-1} \sigma_i^\alpha. \tag{1.7}$$

Case II: Closed Chain System (periodic boundary condition or wrap up model)

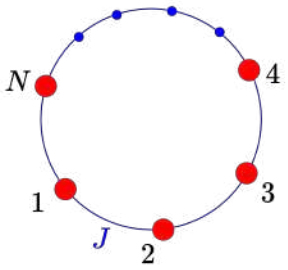


Figure 1.2: Closed chain system / periodic boundary condition

In this type of system, all the spins interact with two corresponding neighbours. So the Hamiltonian can be written as,

$$\mathcal{H} = -J \sum_{i=1}^{N} \sigma_i^z \sigma_{i+1}^z - B \sum_{i=1}^{N} \sigma_i^\alpha. \tag{1.8}$$

1.3 Building the Hamiltonian and Calculating Eigenvalues

We will discuss two approaches for computing the Hamiltonian and its eigenvalues and eigenvectors. One method is quite old and tedious and requires so much memory allocation and computational time, which makes it difficult for us to compute the Hamiltonian for the higher number of qubits/spins. Due to this, it becomes more important to get an alternative method to compute the Hamiltonian and its eigenvalues and eigenvectors, which forms the backbone in analyzing physical properties of the system.

1.3.1 Tensor Product Approach

This is an old and tedious method to compute the Hamiltonian, which requires more memory allocation and computational time, which makes it difficult to compute the Hamiltonian for higher qubit/spin systems. Let us expand the Hamiltonian and see how it looks for i^{th} qubit,

$$\begin{aligned}\mathcal{H} =&\langle a_1 \cdots a_i a_{i+1} \cdots a_n | I_1 \otimes \cdots \sigma_i^z \otimes \sigma_{i+1}^z \otimes \cdots I_n | a_1 \cdots a_i a_{i+1} \cdots a_n \rangle \\ &+ \langle a_1 \cdots a_i \cdots a_n | I_1 \otimes \cdots \sigma_i^z \otimes \cdots I_n | a_1 \cdots a_i \cdots a_n \rangle.\end{aligned} \tag{1.9}$$

It can be clearly seen that the σ_i^z and σ_{i+1}^z will only act on i^{th} and $(i+1)^{th}$ qubit respectively and also σ_i^α will act on i^{th} qubit only. Now, if we look at the complete Hamiltonian, we have to calculate such 2^N terms, which is a horrible task for higher qubit systems. Let us show a Fortran program that will help you to imagine the difficulty of this method to build the Hamiltonian for higher qubit systems. In the program given below, we have built the Hamiltonian and computed eigenvalues and eigenvectors for the 3-qubit system.

```
1 program ising
2     implicit none
3     integer :: k, l, g
```

```
    real :: s_x(2,2), s_z(2,2), I(2,2), D_1(8,8), D_2(8,8), D_3(8,8)
    real :: D_4(8,8), D_5(8,8), D_6(8,8)
    real :: J = 1.0, B = 1.0, a1, a2, a3
    real :: D_7(8,8), D_8(8,8), D_9(8,8), D_10(8,8)
    character*1 :: JOBZ, UPLO
    integer, parameter :: N = 8
    integer ::  LDA = 8, LWORK = 3*N, INFO
    double precision :: W(N), WORK(3*N), H(N,N)

    ! Initialize H to zero
    H = 0.0

    ! Defining Pauli Matrix sigma_0
    I(1, 1) = 1.0
    I(1, 2) = 0.0
    I(2, 1) = 0.0
    I(2, 2) = 1.0

    ! Defining Pauli Matrix sigma_x
    s_x(1, 1) = 0.0
    s_x(1, 2) = 1.0
    s_x(2, 1) = 1.0
    s_x(2, 2) = 0.0

    ! Defining Pauli Matrix sigma_z
    s_z(1, 1) = 1.0
    s_z(1, 2) = 0.0
    s_z(2, 1) = 0.0
    s_z(2, 2) = -1.0

    !Input a1, a2 and a3
    print *, 'Enter value of a1:'
    read(*, *) a1
    print *, 'Enter value of a2:'
    read(*, *) a2
    print *, 'Enter value of a3:'
    read(*, *) a3

```

```
42      ! Calling function
43      call tensorprod(s_z, s_z, I, D_1)
44      write(45, *) 'D_1 = '
45      write(45, *) D_1
46
47      call tensorprod(I, s_z, s_z, D_2)
48      write(45, *) 'D_2 = '
49      write(45, *) D_2
50
51      call tensorprod(s_z, I, s_z, D_3)
52      write(45, *) 'D_3 = '
53      write(45, *) D_3
54
55      call tensorprod(s_z, I, I, D_4)
56      write(45, *) 'D_4 = '
57      write(45, *) D_4
58
59      call tensorprod(I, s_z, I, D_5)
60      write(45, *) 'D_5 = '
61      write(45, *) D_5
62
63      call tensorprod(I, I, s_z, D_6)
64      write(45, *) 'D_6 = '
65      write(45, *) D_6
66
67      call tensorprod(s_x, I, I, D_7)
68      write(45, *) 'D_7 = '
69      write(45, *) D_7
70
71      call tensorprod(I, s_x, I, D_8)
72      write(45, *) 'D_8 = '
73      write(45, *) D_8
74
75      call tensorprod(I, I, s_x, D_9)
76      write(45, *) 'D_9 = '
77      write(45, *) D_9
78
79      D_10 = D_7 + D_8 + D_9
```

```
80      write(45, *) 'D_10 = '
81      write(45, *) D_10
82
83      ! Calculate Hamiltonian H
84      write(45,*) 'Hamiltonian Matrix :'
85      do k = 1, 8
86          do l = 1, 8
87              H(k, l) = a1*J*(D_1(k, l) + D_2(k, l) + D_3(k, l)) +&
88                        a2*B*(D_4(k, l) + D_5(k, l) + D_6(k, l)) +&
89                        a3*B*(D_7(k, l) + D_8(k, l) + D_9(k, l))
90              write(45, *) 'H(', k, ',', l, ') =', H(k, l)
91          end do
92      end do
93
94      ! Define parameters for DSYEV
95      JOBZ = 'V'  ! Compute eigenvalues and eigenvectors
96      UPLO = 'U'  ! Upper triangular part of A is stored
97
98      ! Call DSYEV to compute eigenvalues and eigenvectors
99      call DSYEV(JOBZ, UPLO, N, H, LDA, W, WORK, LWORK, INFO)
100
101     ! Check for successful execution
102     if (INFO == 0) then
103         write(45,*) 'Eigenvalues are:'
104         do g = 1, N
105             write(45,*) W(g)
106         end do
107     else
108         print*, 'Error in DSYEV, info =', INFO
109     end if
110     write(45,*) 'Eigen Vectors are :'
111     do k = 1, 8
112         write(45,*) 'Eigen Vector ', k, ':'
113         do l = 1, 8
114             write(45,*) H(l,k)
115         end do
116     end do
117 contains
```

```
118 subroutine tensorprod(A, B, C, D)
119     implicit none
120     real, intent(in) :: A(2, 2), B(2, 2), C(2, 2)
121     real, intent(out) :: D(8, 8)
122     real :: temp(4, 4)
123     integer :: p, q, k, l
124 
125     ! Initialize the result matrix
126     D = 0.0
127 
128     ! Calculate the tensor product of A and B first
129     do p = 1, 2
130         do q = 1, 2
131             do k = 1, 2
132                 do l = 1, 2
133                     temp((p-1)*2 + k,(q-1)*2 + l) = A(p, q)*B(k, l)
134                 end do
135             end do
136         end do
137     end do
138     ! Now calculate the tensor product of temp and C
139     do p = 1, 4
140         do q = 1, 4
141             do k = 1, 2
142                 do l = 1, 2
143                     D((p-1)*2 + k,(q-1)*2 + l) = temp(p, q)*C(k, l)
144                 end do
145             end do
146         end do
147     end do
148 end subroutine tensorprod
149 end program ising
150 
```

In this program, we have taken a_1, a_2, and a_3 as input from the user, which will give the user the flexibility to alter the direction of the applied magnetic field (i.e. if $a_2 = 1$ and $a_3 = 0$, the applied magnetic field will be in the longitudinal direction

(z-direction) and if $a_2 = 0$ and $a_3 = 1$, then the applied magnetic field will be in the transverse direction (x-direction)). So a_1, a_2, and a_3 in the program can be considered as a switch which allows us to play with the applied magnetic field.
Now you have got an idea about how complex the process is to compute each matrix first and then add them all to get the Hamiltonian, and then compute the eigenvalues and eigenvectors. To get rid of this complexity, let us introduce an alternative way for computing the Hamiltonian.

1.3.2 Bit Operation Approach

Before learning the method, let us look into the operation of Pauli matrices on the qubits. One can simply calculate these operations using matrix multiplication. Let us see, using a Python program, how Pauli matrices act on qubits,

```
# Importing Libraries
import numpy as np
from scipy import linalg
import math
from IPython.display import display, Math
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore")

# Pauli Matrices for Qubits
I = np.eye(2)
X = np.array([[0, 1], [1, 0]])         # Pauli X (sigma_x)
Y = np.array([[0, -1j], [1j, 0]])      # Pauli Y (sigma_y)
Z = np.array([[1, 0], [0, -1]])        # Pauli Z (sigma_z)

# Qubit Basis States |0> and |1>
zero = np.array([[1], [0]])
one  = np.array([[0], [1]])

# Function to Print Matrices in LaTeX Format
def print_matrix(array):
    matrix = ''
```

```
    for row in array:
        for number in row:
            matrix += f'{np.round(number, 3)}&'
        matrix = matrix[:-1] + r'\\'
    display(Math(r'\begin{bmatrix}'+matrix+r'\end{bmatrix}'))

# Display matrix elements <i|X|j> for all i, j in {0,1}
for i, ket_i in enumerate([zero, one]):
    for j, ket_j in enumerate([zero, one]):
        expr = rf"\langle {i}|X|{j} \rangle ="
        display(Math(expr))
        print_matrix(np.matmul(ket_i.T.conj(), np.matmul(X, ket_j)))

# Display matrix elements <i|Y|j> for all i, j in {0,1}
for i, ket_i in enumerate([zero, one]):
    for j, ket_j in enumerate([zero, one]):
        expr = rf"\langle {i}|Y|{j} \rangle ="
        display(Math(expr))
        print_matrix(np.matmul(ket_i.T.conj(), np.matmul(Y, ket_j)))

# Display matrix elements <i|Z|j> for all i, j in {0,1}
for i, ket_i in enumerate([zero, one]):
    for j, ket_j in enumerate([zero, one]):
        expr = rf"\langle {i}|Z|{j} \rangle ="
        display(Math(expr))
        print_matrix(np.matmul(ket_i.T.conj(), np.matmul(Z, ket_j)))

```

So, here are the observations from this Python exercise,

$$\sigma^x|0\rangle = |1\rangle, \qquad \sigma^x|1\rangle = |0\rangle, \tag{1.10}$$

$$\sigma^y|0\rangle = i|1\rangle, \qquad \sigma^y|1\rangle = -i|0\rangle, \tag{1.11}$$

$$\sigma^z|0\rangle = |0\rangle, \qquad \sigma^z|1\rangle = -|1\rangle. \tag{1.12}$$

So, if we want to apply σ^x or σ^y or σ^z on i^{th} qubit, we have to generalize the operation, which gives us for $a = 0$ or 1 as,

$$\sigma^x|a\rangle = |(1-a)\rangle, \tag{1.13}$$

$$\sigma^y|a\rangle = (-1)^a i|(1-a)\rangle, \tag{1.14}$$

$$\sigma^z|a\rangle = (1-2a)|a\rangle, \tag{1.15}$$

where a represents the qubit in which the Pauli matrices are operating. Now let us understand this operation by expanding the i^{th} term of the Hamiltonian for the applied magnetic field along the longitudinal direction,

$$\begin{aligned}\mathcal{H} &= \langle a_1 \cdots a_i a_{i+1} \cdots a_n | I_1 \otimes \cdots \sigma_i^z \otimes \sigma_{i+1}^z \otimes \cdots I_n | a_1 \cdots a_i a_{i+1} \cdots a_n \rangle \\ &\quad + \langle a_1 \cdots a_i \cdots a_n | I_1 \otimes \cdots \sigma_i^z \otimes \cdots I_n | a_1 \cdots a_i \cdots a_n \rangle \\ &= (1-2a_i)(1-2a_{i+1}) \langle a_1 \cdots a_i a_{i+1} \cdots a_n | a_1 \cdots a_i a_{i+1} \cdots a_n \rangle \\ &\quad + (1-2a_i) \langle a_1 \cdots a_i \cdots a_n | a_1 \cdots a_i \cdots a_n \rangle.\end{aligned} \tag{1.16}$$

Similarly, this can be implemented for the transverse magnetic field. This method allows us to compute the Hamiltonian for higher qubit systems with less memory usage and less computational time. So let us implement this in a Fortran program as below,

```
program bit_op_for_sigma_z
    implicit none
    integer :: i, j, k, s, num, l, N, g
    real*8 :: P, B
    character(len=:), allocatable :: state(:)
    integer, allocatable :: digits(:), decimal(:)
    double precision, allocatable :: D(:,:), C(:,:), H(:,:), W(:),
   WORK(:)
    character*1 :: JOBZ, UPLO
    integer ::  LDA , LWORK , INFO

    print *, "Enter total number of bits: "
    read *, s
    num = 2**s
```

```
    N = num
    LDA = num
    LWORK = 3*N

    ! Allocate arrays after determining the value of num
    allocate(character(len=s) :: state(num))
    allocate(integer :: decimal(num))
    allocate(integer :: digits(s))
    allocate(real(8) :: C(num, num))
    allocate(real(8) :: D(num, num))
    allocate(real(8) :: H(num , num))
    allocate(real(8) :: W(LDA), WORK(LWORK))

    !print *, "The states are:"
    do i = 0, num - 1
        call integer_binary(i, state(i+1), s)
        !print *, state(i)
    end do
    do i = 1, num
        decimal(i) = btod(state(i))
        !print*, ' Decimal equivalent of ', state(i) ,' = ', decimal
   (i)
    end do
    D = 0.0
        do i = 1, num
            do j = 1, num
                call sd(state(i), digits)
                    if (decimal(i) .ne. decimal(j)) then
                        D(i, j) =  0.0
                        !print*, 'D(',i, ',', j,') =', D(i,j)
                    else
                        D(i,j) = 0.0
                        do l = 1, s - 1
                            D(i, j) = D(i, j) + (1 - 2 * digits(l))
                                        * (1 - 2 * digits(l+1))
                        end do
                        D(i, j) = D(i, j) + (1 - 2 * digits(1))
                                        * (1 - 2 * digits(s))
```

```
                    end if
                end do
            end do
        !write(57,*) D
        C = 0.0
            do i = 1, num
                do j = 1, num
                    call sd(state(i), digits)
                        if (decimal(i) .ne. decimal(j)) then
                            C(i, j) =  0.0
                        else
                            C(i,j) = 0.0
                            do l = 1, s
                                C(i, j) = C(i, j) + (1 - 2 * digits(l))
                            end do
                        end if
                end do
            end do
        !write(57,*) C
        ! Initialize H matrix if needed
        H = 0.0
        P = 1
        B = 1
        !write(57,*) ' J = ', P
        H = P * D + B * C
        !write(46,*) 'H(',i,',',j,') = ', H
        !write(57,*) ' H = '
        !do i = 1, num
            !do j = 1, num
                !write(57,*) i,j, H(i,j)
            !end do
        !end do
        ! Define parameters for DSYEV
        JOBZ = 'V'  ! Compute eigenvalues and eigenvectors
        UPLO = 'U'  ! Upper triangular part of A is stored
        ! Call DSYEV to compute eigenvalues and eigenvectors
        call DSYEV(JOBZ, UPLO, N, H, LDA, W, WORK, LWORK, INFO)
        ! Check for successful execution
```

```
89      if (INFO == 0) then
90          !write(57,*) 'Eigenvalues are:'
91          do g = 1, N
92              write(57,*) W(g)
93          end do
94      else
95          print*, 'Error in DSYEV, info =', INFO
96      end if
97      !write(57,*) 'Eigen Vectors are :'
98      !do k = 1, num
99          !write(57,*) 'Eigen Vector ', k, ':'
100         !do l = 1, num
101             !write(57,*) H(l,k)
102         !end do
103     !end do
104     ! Deallocate arrays
105     deallocate(state)
106     deallocate(decimal)
107     deallocate(digits)
108     deallocate(C)
109     deallocate(D)
110     deallocate(H)
111     deallocate(W)
112     deallocate(WORK)
113
114 contains
115
116 function btod(binaryString) result(decimalValue)
117     implicit none
118     character(len=*), intent(in) :: binaryString
119     integer :: decimalValue
120     integer :: i, length
121     decimalValue = 0
122     length = len_trim(binaryString)  ! Get the length of the binary
    string
123     ! Convert binary string to decimal
124     do i = 1, length
125         if (binaryString(i:i) == '1') then
```

```
126             decimalValue = decimalValue + 2**(length - i)
127         else if (binaryString(i:i) /= '0') then
128             print *, "Invalid character in binary string."
129             decimalValue = -1  ! Indicate an error with -1
130             return
131         end if
132     end do
133 end function btod
134 subroutine sd(binaryString, digits)
135     character(len=*), intent(in) :: binaryString
136     integer, allocatable, intent(out) :: digits(:)
137     integer :: i, len
138     len = len_trim(binaryString)  ! Get the length of the binary
    string
139     allocate(digits(len))          ! Allocate array to hold digits
140     ! Convert each character to an integer
141     do i = 1, len
142         if (binaryString(i:i) == '1') then
143             digits(i) = 1
144         else if (binaryString(i:i) == '0') then
145                 digits(i) = 0
146             else
147             print *, "Invalid character in binary string."
148             digits = 0  ! Set to zero if invalid character is found
149             return
150         end if
151     end do
152 end subroutine sd
153 subroutine integer_binary(num, binary_string, n)
154     implicit none
155     integer :: num
156     integer, intent(in) :: n
157     integer :: i, temp_num
158     character(len=n), intent(out) :: binary_string
159     temp_num = num
160     ! Initialize binary_string to all zeros
161     binary_string = repeat('0', n)
162     ! Convert the integer to binary
```

```
    do i = 0, n - 1
        if (mod(temp_num, 2) == 1) then
            binary_string(n-i:n-i) = '1'  ! Set the bit to '1'
        else
            binary_string(n-i:n-i) = '0'  ! Set the bit to '0'
        end if
        temp_num = temp_num / 2  ! Divide num by 2 to shift right
    end do
end subroutine integer_binary
end program
```

1.3.3 Energy Eigenvalue Spectrum for Variable J and B

Let us study the energy eigenvalue spectrum of the Hamiltonian with varying interaction energy from J = 0 to 1 in the stepsize of 0.01 keeping the applied magnetic field constant as 1, and also for the applied magnetic field, B varying from 0 to 1 in the stepsize of 0.01 keeping interaction energy constant as 1 for the system considering spin-spin interaction along the same direction (i.e. along the z-direction.), So the Hamiltonian can be given as [2],

$$\mathcal{H} = -J\sum_{i=1}^{n}\sigma_i^z\sigma_{i+1}^z - B\sum_{i=1}^{n}\sigma_i^z. \tag{1.17}$$

Implementation of this in a Fortran program as below,

```
program bit_op_for_sigma_z
    implicit none
    integer :: i, j, k, s, num, l, N, g
    real*8 :: P, B
    character(len=:), allocatable :: state(:)
    integer, allocatable :: digits(:), decimal(:)
    double precision, allocatable :: D(:,:), C(:,:), H(:,:), W(:),
   WORK(:)
    character*1 :: JOBZ, UPLO
    integer ::  LDA , LWORK , INFO
```

```
10
11      print *, "Enter total number of bits: "
12      read *, s
13      num = 2**s
14
15      N = num
16      LDA = num
17      LWORK = 3*N
18
19      ! Allocate arrays after determining the value of num
20      allocate(character(len=s) :: state(num))
21      allocate(integer :: decimal(num))
22      allocate(integer :: digits(s))
23      allocate(real(8) :: C(num, num))
24      allocate(real(8) :: D(num, num))
25      allocate(real(8) :: H(num , num))
26      allocate(real(8) :: W(LDA), WORK(LWORK))
27
28      !print *, "The states are:"
29      do i = 0, num - 1
30          call integer_binary(i, state(i+1), s)
31          !print *, state(i)
32      end do
33      do i = 1, num
34          decimal(i) = btod(state(i))
35          !print*, ' Decimal equivalent of ', state(i) ,' = ', decimal
    (i)
36      end do
37      D = 0.0
38          do i = 1, num
39              do j = 1, num
40                  call sd(state(i), digits)
41                      if (decimal(i) .ne. decimal(j)) then
42                          D(i, j) =  0.0
43                          !print*, 'D(',i, ',', j,') =', D(i,j)
44                      else
45                          D(i,j) = 0.0
46                          do l = 1, s - 1
```

```
                            D(i, j) = D(i, j) + (1 - 2 * digits(l))
                                              * (1 - 2 * digits(l+1))
                        end do
                        D(i, j) = D(i, j) + (1 - 2 * digits(1))
                                          * (1 - 2 * digits(s))
                    end if
                end do
            end do
        !write(57,*) D
        C = 0.0
            do i = 1, num
                do j = 1, num
                    call sd(state(i), digits)
                        if (decimal(i) .ne. decimal(j)) then
                            C(i, j) =  0.0
                        else
                            C(i,j) = 0.0
                            do l = 1, s
                                C(i, j) = C(i, j) + (1 - 2 * digits(l))
                            end do
                        end if
                end do
            end do
        !write(57,*) C
        ! Initialize H matrix if needed
        H = 0.0
        !Varying value of J
        do l = 0, 100 , 1
            P = 0.01 * l
            B = 1
            !write(57,*) ' J = ', P
            H = P * D + B * C
            !write(46,*) 'H(',i,',',j,') = ', H
            !write(57,*) ' H = '
            !do i = 1, num
                !do j = 1, num
                    !write(57,*) i,j, H(i,j)
                !end do
```

```
85          !end do
86
87          ! Define parameters for DSYEV
88          JOBZ = 'V'  ! Compute eigenvalues and eigenvectors
89          UPLO = 'U'  ! Upper triangular part of A is stored
90          ! Call DSYEV to compute eigenvalues and eigenvectors
91          call DSYEV(JOBZ, UPLO, N, H, LDA, W, WORK, LWORK, INFO)
92          ! Check for successful execution
93          if (INFO == 0) then
94              !write(57,*) 'Eigenvalues are:'
95              do g = 1, N
96                  write(57,*) P, W(g)
97              end do
98          else
99              print*, 'Error in DSYEV, info =', INFO
100         end if
101
102         !write(57,*) 'Eigen Vectors are :'
103         !do k = 1, num
104             !write(57,*) 'Eigen Vector ', k, ':'
105             !do l = 1, num
106                 !write(57,*) H(l,k)
107             !end do
108         !end do
109     end do
110     !Varying value of B
111     do l = 0, 100 , 1
112         B = 0.01 * l
113         P = 1
114         !write(58,*) ' B = ', B
115         H =  P * D + B * C
116         !write(58,*) 'H(',i,',',j,') = ', H
117         !write(58,*) ' H = '
118         !do i = 1, num
119             !do j = 1, num
120                 !write(58,*) i,j, H(i,j)
121             !end do
122         !end do
```

```
123         ! Define parameters for DSYEV
124         JOBZ = 'V'  ! Compute eigenvalues and eigenvectors
125         UPLO = 'U'  ! Upper triangular part of A is stored
126         ! Call DSYEV to compute eigenvalues and eigenvectors
127         call DSYEV(JOBZ, UPLO, N, H, LDA, W, WORK, LWORK, INFO)
128         ! Check for successful execution
129         if (INFO == 0) then
130             !write(58,*) 'Eigenvalues are:'
131             do g = 1, N
132                 write(58,*) B, W(g)
133             end do
134         else
135             print*, 'Error in DSYEV, info =', INFO
136         end if
137         !write(58,*) 'Eigen Vectors are :'
138         !do k = 1, num
139             !write(58,*) 'Eigen Vector ', k, ':'
140             !do l = 1, num
141                 !write(58,*) H(l,k)
142             !end do
143         !end do
144     end do
145     ! Deallocate arrays
146     deallocate(state)
147     deallocate(decimal)
148     deallocate(digits)
149     deallocate(C)
150     deallocate(D)
151     deallocate(H)
152     deallocate(W)
153     deallocate(WORK)
154
155 contains
156
157 function btod(binaryString) result(decimalValue)
158     implicit none
159     character(len=*), intent(in) :: binaryString
160     integer :: decimalValue
```

```
    integer :: i, length
    decimalValue = 0
    length = len_trim(binaryString)  ! Get the length of the binary
   string
    ! Convert binary string to decimal
    do i = 1, length
        if (binaryString(i:i) == '1') then
            decimalValue = decimalValue + 2**(length - i)
        else if (binaryString(i:i) /= '0') then
            print *, "Invalid character in binary string."
            decimalValue = -1  ! Indicate an error with -1
            return
        end if
    end do
end function btod
subroutine sd(binaryString, digits)
    character(len=*), intent(in) :: binaryString
    integer, allocatable, intent(out) :: digits(:)
    integer :: i, len
    len = len_trim(binaryString)  ! Get the length of the binary
   string
    allocate(digits(len))          ! Allocate array to hold digits
    ! Convert each character to an integer
    do i = 1, len
        if (binaryString(i:i) == '1') then
            digits(i) = 1
        else if (binaryString(i:i) == '0') then
                digits(i) = 0
            else
            print *, "Invalid character in binary string."
            digits = 0  ! Set to zero if invalid character is found
            return
        end if
    end do
end subroutine sd
subroutine integer_binary(num, binary_string, n)
    implicit none
    integer :: num
```

```
197     integer, intent(in) :: n
198     integer :: i, temp_num
199     character(len=n), intent(out) :: binary_string
200     temp_num = num
201     ! Initialize binary_string to all zeros
202     binary_string = repeat('0', n)
203     ! Convert the integer to binary
204     do i = 0, n - 1
205         if (mod(temp_num, 2) == 1) then
206             binary_string(n-i:n-i) = '1'  ! Set the bit to '1'
207         else
208             binary_string(n-i:n-i) = '0'  ! Set the bit to '0'
209         end if
210         temp_num = temp_num / 2  ! Divide num by 2 to shift right
211     end do
212 end subroutine integer_binary
213 end program
214
```

So let us see the plots of eigenvalues vs J and B and try to conclude some important information from that.

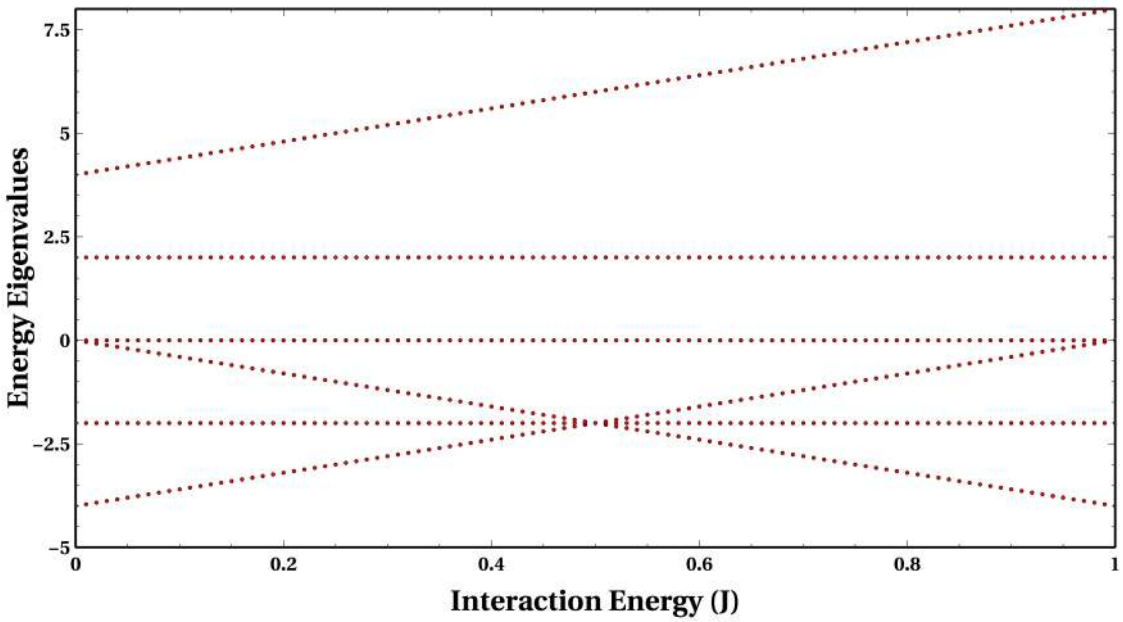


Figure 1.3: Energy eigenvalues vs interaction energy (J) for 4 qubits for both J and B along z-direction

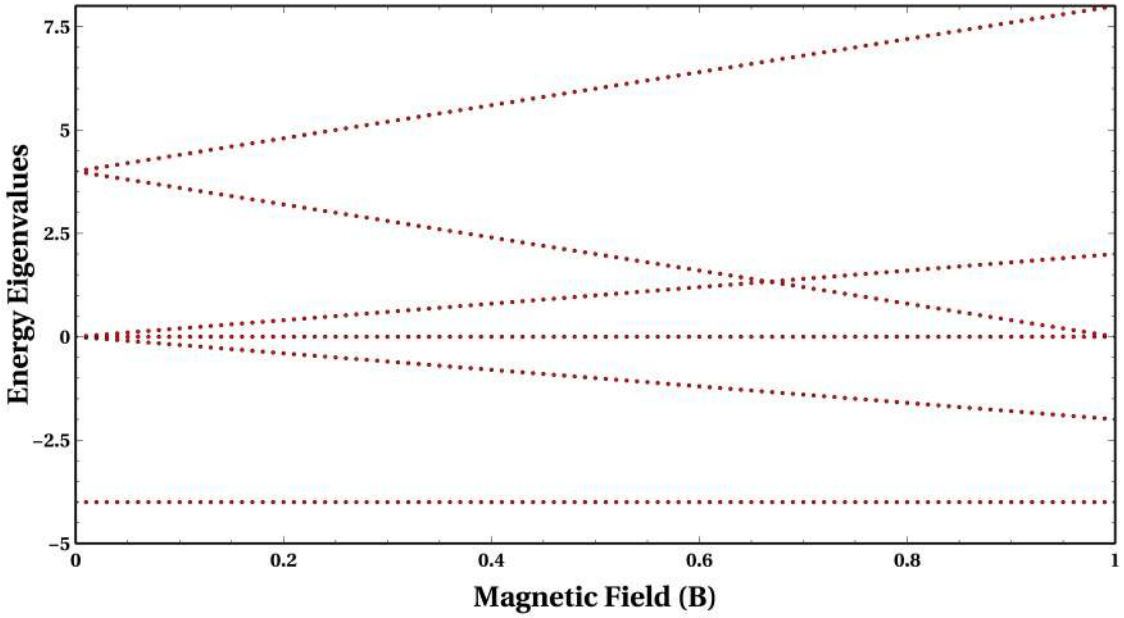


Figure 1.4: Energy eigenvalues vs magnetic field (B) for 4 qubits for both J and B along z-direction

Now if we observe the variation of energy eigenvalue w.r.t. J and B, we get a straight line for both, which is obvious because the Hamiltonian is made of σ^z terms only, so we can easily get a set of common eigenvectors that will diagonalize both the terms in Hamiltonian [4], which can also be represented mathematically by considering $|\psi\rangle$ as an arbitrary eigenvector of $\mathcal{H}$ as,

$$\langle\psi|\mathcal{H}|\psi\rangle = -J\underbrace{\sum_i \langle\psi|\sigma_i^z\sigma_{i+1}^z|\psi\rangle}_{\alpha} - B\underbrace{\sum_i \langle\psi|\sigma_i^z|\psi\rangle}_{\beta} \implies \mathcal{E} = -J\alpha - B\beta. \tag{1.18}$$

where $\mathcal{E}$ is the energy eigenvalue corresponding to that eigenvector of $\mathcal{H}$ and α and β are some constants or numbers. So this equation is of a straight line y = mx + c.

Now consider the system having spin-spin interaction along the z-direction and the magnetic field along the x-direction. So the Hamiltonian can be given as [3],

$$\mathcal{H} = -J\sum_{i=1}^{n}\sigma_i^z\sigma_{i+1}^z - B\sum_{i=1}^{n}\sigma_i^x. \tag{1.19}$$

Implementation of this in a Fortran program is given by,

```
program bit_op_for_sigma_x
    implicit none
    integer :: i, j, k, s, num, l, N, g
    real*8 :: P, B
    character(len=:), allocatable :: state(:)
    integer, allocatable :: digits1(:), digits2(:), decimal(:)
    double precision, allocatable :: D(:,:), C(:,:), H(:,:), W(:),
   WORK(:)
    character*1 :: JOBZ, UPLO
    integer ::  LDA , LWORK , INFO, flag
    print *, "Enter total number of bits: "
    read *, s
    num = 2**s
    N = num
    LDA = num
    LWORK = 3*N
    ! Allocate arrays after determining the value of num
    allocate(character(len=s) :: state(num))
    allocate(integer :: decimal(num))
    allocate(integer :: digits1(s), digits2(s))
    allocate(real(8) :: C(num, num))
    allocate(real(8) :: D(num, num))
    allocate(real(8) :: H(num , num))
    allocate(real(8) :: W(LDA), WORK(LWORK))
    !print *, "The states are:"
    do i = 0, num - 1
        call integer_binary(i, state(i+1), s)  ! Adjusted for 1-
   based indexing
        !print *, state(i)
    end do
    do i = 1, num
        decimal(i) = btod(state(i))
        !print*, ' Decimal equivalent of ', state(i) ,' = ', decimal
   (i)
    end do

    D = 0.0
```

```
35          do i = 1, num
36              do j = 1, num
37                  call sd(state(i), digits1)
38                      if (decimal(i) .ne. decimal(j)) then
39                          D(i, j) =  0.0
40                          !print*, 'D(',i, ',', j,') =', D(i,j)
41                      else
42                          D(i,j) = 0.0
43                          do l = 1, s - 1
44                              D(i, j) = D(i, j) + (1 - 2 * digits1(l))
    * (1 - 2 * digits1(l+1))
45                              !print*, 'D(',i, ',', j,') =', D(i,j)
46                          end do
47                          D(i, j) = D(i, j) + (1 - 2 * digits1(1)) *
   (1 - 2 * digits1(s))
48                      end if
49              end do
50          end do
51      ! write(57,*) D
52      !print *, 'D = '
53      !do i = 1, num
54          !print *, D(i, :)
55      !end do
56      C = 0.0
57      flag = 0
58      do k = 1, s
59          !print* ,"k =", k
60          do i = 1, num
61              call sd(state(i), digits1)
62              !print* , digits1
63              do j = 1, num
64                  flag = 0
65                  call sd(state(j), digits2)
66                  !print* , digits2
67                  if (i == j) then
68                      C(i, j) = C(i,j) + 0.0
69                      !print*, 'C(',i, ',', j,') =', C(i,j)
70                  else
```

```
                    C(i,j) = C(i,j) + 0.0
                    do l = 1, s
                        if (l == k) then
                            continue
                        else if (digits1(l) .ne. digits2(l)) then
                            flag = flag + 1
                            !print*, flag
                        end if
                    end do
                    if (flag .ne. 0) then
                        C(i,j) = C(i,j) + 0.0
                        !print*, 'C(',i, ',', j,') =', C(i,j)
                    else
                        C(i,j) = C(i,j) + 1
                        !print*, 'C(',i, ',', j,') =', C(i,j)
                    end if
                end if
            end do
        end do
    end do
    !print* , C
    write(57,*) C+D
    ! Initialize H matrix if needed

    H = 0.0
    !Varying value of J
    do l = 0, 100 , 1
        P = 0.01 * l
        B = 1
        !write(57,*) ' J = ', P
        H = P * D + B * C
        !write(46,*) 'H(',i,',',j,') = ', H
        !write(57,*) ' H = '
        !do i = 1, num
            !do j = 1, num
                !write(57,*) i,j, H(i,j)
            !end do
        !end do
```

```
109         ! Define parameters for DSYEV
110         JOBZ = 'V'  ! Compute eigenvalues and eigenvectors
111         UPLO = 'U'  ! Upper triangular part of A is store
112         ! Call DSYEV to compute eigenvalues and eigenvectors
113         call DSYEV(JOBZ, UPLO, N, H, LDA, W, WORK, LWORK, INFO)
114         ! Check for successful execution
115         if (INFO == 0) then
116             ! write(57,*) 'Eigenvalues are:'
117             do g = 1, N
118                 write(59,*) P, W(g)
119             end do
120         else
121             print*, 'Error in DSYEV, info =', INFO
122         end if
123 !        write(57,*) 'Eigen Vectors are :'
124 !        do k = 1, num
125 !            write(57,*) 'Eigen Vector ', k, ':'
126 !            do l = 1, num
127 !                write(57,*) H(l,k)
128             !end do
129         !end do
130     end do
131     !Varying value of B
132     !do k = 0, 100 , 1
133         do l = 0, 100 , 1
134         B = 0.01 * l
135         P = 1
136         !write(58,*) ' B = ', B
137         H =  P * D + B * C
138         !write(58,*) 'H(',i,',',j,') = ', H
139         !write(58,*) ' H = '
140         !do i = 1, num
141             !do j = 1, num
142                 !write(58,*) i,j, H(i,j)
143             !end do
144         !end do
145         ! Define parameters for DSYEV
146         JOBZ = 'V'  ! Compute eigenvalues and eigenvectors
```

```
147         UPLO = 'U'  ! Upper triangular part of A is stored
148         ! Call DSYEV to compute eigenvalues and eigenvectors
149         call DSYEV(JOBZ, UPLO, N, H, LDA, W, WORK, LWORK, INFO)
150         ! Check for successful execution
151         if (INFO == 0) then
152             !write(58,*) 'Eigenvalues are:'
153             do g = 1, N
154                 write(60,*) B, W(g)
155             end do
156         else
157             print*, 'Error in DSYEV, info =', INFO
158         end if
159         !write(58,*) 'Eigen Vectors are :'
160         !do k = 1, num
161             !write(58,*) 'Eigen Vector ', k, ':'
162             !do l = 1, num
163                 !write(58,*) H(l,k)
164             !end do
165         !end do
166         end do
167     !end do
168     ! Deallocate arrays
169     deallocate(state)
170     deallocate(decimal)
171     deallocate(digits1)
172     deallocate(digits2)
173     deallocate(C)
174     deallocate(D)
175     deallocate(H)
176     deallocate(W)
177     deallocate(WORK)
178 
179 contains
180 
181 function btod(binaryString) result(decimalValue)
182     implicit none
183     character(len=*), intent(in) :: binaryString
184     integer :: decimalValue
```

```
185     integer :: i, length
186     decimalValue = 0
187     length = len_trim(binaryString)  ! Get the length of the binary
    string
188     ! Convert binary string to decimal
189     do i = 1, length
190         if (binaryString(i:i) == '1') then
191             decimalValue = decimalValue + 2**(length - i)
192         else if (binaryString(i:i) /= '0') then
193             print *, "Invalid character in binary string."
194             decimalValue = -1  ! Indicate an error with -1
195             return
196         end if
197     end do
198 end function btod
199 subroutine sd(binaryString, digits)
200     character(len=*), intent(in) :: binaryString
201     integer, allocatable, intent(out) :: digits(:)
202     integer :: i, len
203     len = len_trim(binaryString)  ! Get the length of the binary
    string
204     allocate(digits(len))          ! Allocate array to hold digits
205     ! Convert each character to an integer
206     do i = 1, len
207         if (binaryString(i:i) == '1') then
208             digits(i) = 1
209         else if (binaryString(i:i) == '0') then
210             digits(i) = 0
211         else
212             print *, "Invalid character in binary string."
213             digits = 0  ! Set to zero if invalid character is found
214             return
215         end if
216     end do
217 end subroutine sd
218 subroutine integer_binary(num, binary_string, n)
219     implicit none
220     integer :: num
```

```
221     integer, intent(in) :: n
222     integer :: i, temp_num
223     character(len=n), intent(out) :: binary_string
224     temp_num = num
225     ! Initialize binary_string to all zeros
226     binary_string = repeat('0', n)
227     ! Convert the integer to binary
228     do i = 0, n - 1
229         if (mod(temp_num, 2) == 1) then
230             binary_string(n-i:n-i) = '1'  ! Set the bit to '1'
231         else
232             binary_string(n-i:n-i) = '0'  ! Set the bit to '0'
233         end if
234         temp_num = temp_num / 2  ! Divide num by 2 to shift right
235     end do
236 end subroutine integer_binary
237 end program
238
```

So let us see the plots of eigenvalues vs J and B and try to conclude some important information from that,

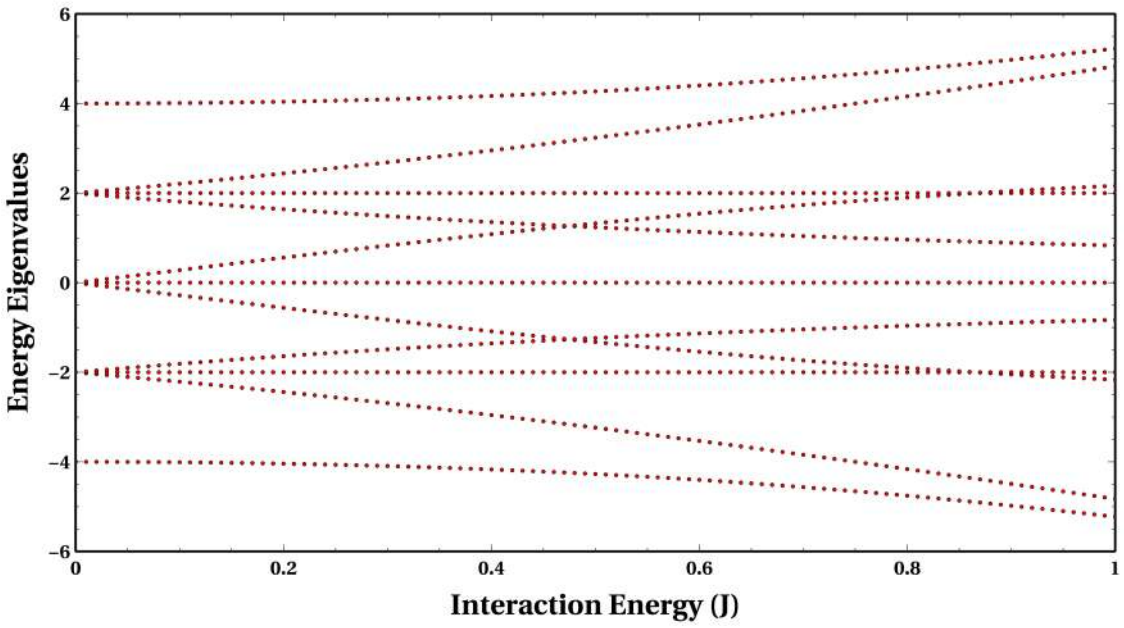


Figure 1.5: Energy eigenvalues vs interaction energy (J) for 4 qubits for J along z-direction and B along x-direction

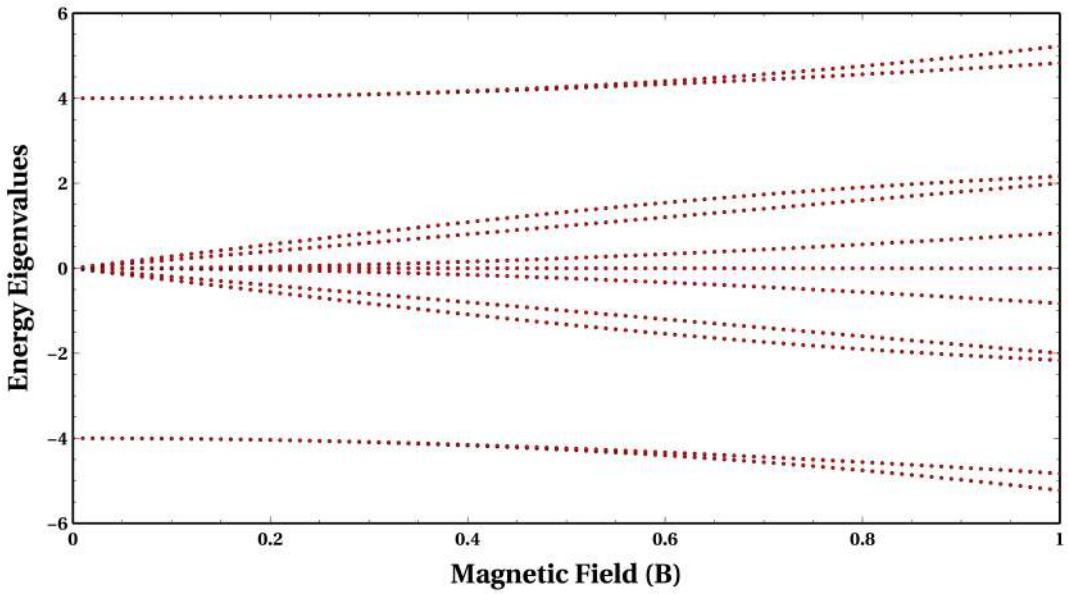


Figure 1.6: Energy eigenvalues vs magnetic field (B) for 4 qubits for J along z-direction and B along x-direction

Now if we observe the variation of energy eigenvalue w.r.t. J and B, we do not get a straight line for both, this is because the Hamiltonian is made of σ^z and σ^x terms which do not commute [4], so we cannot get a set of common eigenvectors that will diagonalize both the terms in Hamiltonian, since we are using numerical approach (DSYEV) so it will give us the approximate eigenvectors.
Now let us see the plots of Ground State Energy vs Interaction Energy and Magnetic Field for different numbers of qubits.

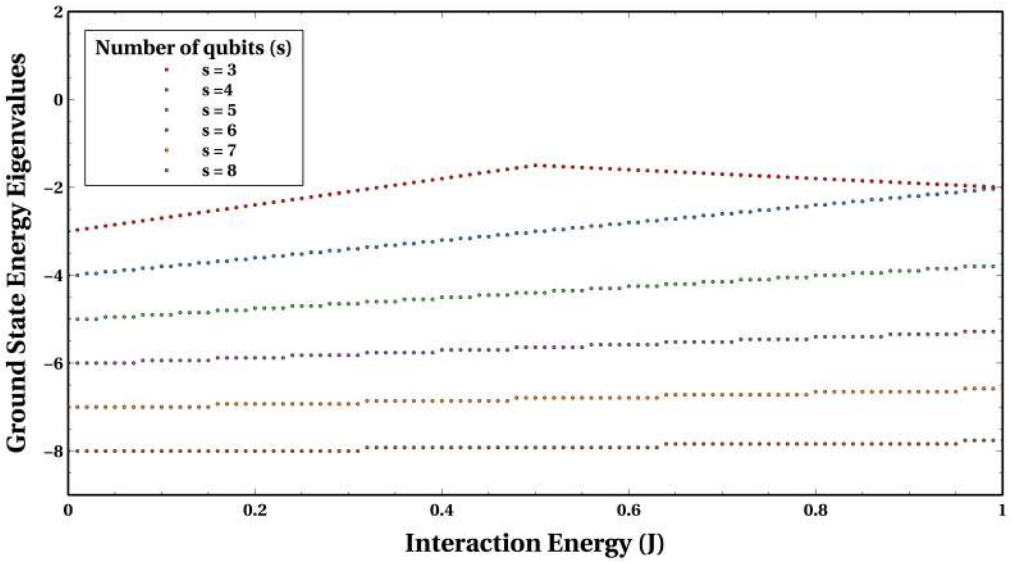


Figure 1.7: Ground state energy vs interaction energy (J) for different numbers of qubits.

So it is quite evident from the plots that as the number of qubits increases, the ground state energy decreases. With the increase in the value of interaction energy, the ground state energy increases in steps, depending on the number of qubits.

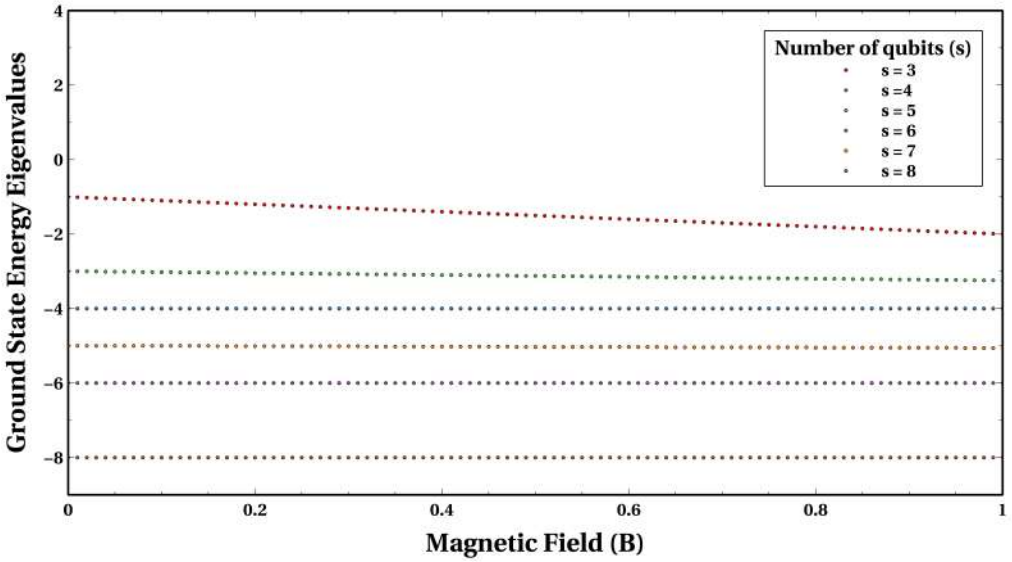


Figure 1.8: Ground state energy vs magnetic field (B) for different numbers of qubits. For an increase in the number of qubits, the ground state energy vs magnetic field shows alternating behaviour, for odd numbers of qubits, energy is higher than the even numbers of qubits. With the increase in the magnetic Field, the ground state energy remains almost constant for all the number of qubits except for 3 qubits, where it decreases slowly.

Chapter 2

Introduction to the AKLT Model

2.1 History and Motivation

Having understood the basics of spin chains in the last chapter by considering the spin half 1D Ising model, we move on to explore more difficult models like the AKLT (Affleck-Kennedy-Lieb-Tasaki) Model, which is defined by Ian Affleck, Tom Kennedy, Elliott H. Lieb, and Hal Tasaki to explain the valence-bond ground state of antiferromagnetic materials [5]. This model is an extension of the one-dimensional Heisenberg spin chain model [6].

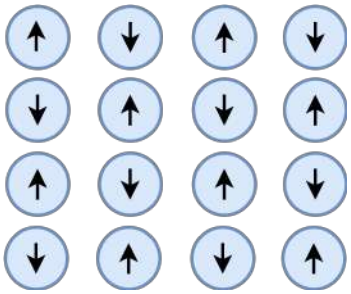


Figure 2.1: Alignment of Magnetic Moments in Antiferromagnets.

The motivation for defining this model is that in antiferromagnets, half of the magnetic moments are aligned in one direction and the rest are aligned in the opposite direction, due to which the net magnetic moment is zero. The ground state of this model is the same as they considered two out of every set of three neighbouring sites,

which are paired to form a valence bond or singlet. This leads to two spin halves for every site (s = 1); thus, the solution must be the wave function of the spin 1 system.

However, Hulthén [7] further modified this model to find a solution for antiferromagnets. Still, this model is important in understanding several areas, such as Quantum states in extreme environments (e.g., Neutron Stars [8] and Quark Matter [9]), magnetar spin dynamics [10], etc.

Let us understand the Heisenberg Model for developing the basics for our AKLT Model.

2.2 Heisenberg Model

Heisenberg spin chain systems can be represented by the Hamiltonian in the form as follows,

$$\mathcal{H} = J \sum_{i=1}^{N} \vec{\sigma_i}.\vec{\sigma_{i+1}}. \tag{2.1}$$

Here, σ_i's are the Pauli spin matrices and J is the interaction energies between the nearest neighbouring spins.

It looks quite interesting about the structure of the Hamiltonian Eq. [2.1], and it is evident that this particular structure of the Hamiltonian originates because of the two types of spin configuration of the spin -$\frac{1}{2}$ particles as singlets and triplets. If we do a rigorous calculation to obtain the energies corresponding to the singlet and triplet configurations, then we can able to see that the difference between the energies for the two configurations is nothing but twice that of the nearest spin interaction(i.e., $2J$ as shown in Fig. [2.2]).

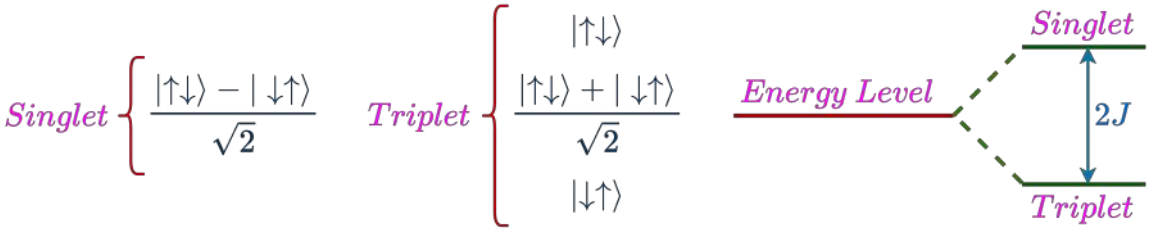


Figure 2.2: Energy level scheme in Heisenberg model

Now, we can think that the Hamiltonian should be,

$$\mathcal{H} = 2J \sum_{i,j} \vec{\sigma_i}.\vec{\sigma_j}. \tag{2.2}$$

But to avoid the double counting as well as self-interaction, we used to consider,

$$\mathcal{H} = J \sum_{i \neq j} \vec{\sigma_i}.\vec{\sigma_j}. \tag{2.3}$$

2.3 The AKLT Hamiltonian

Now, let us understand the concept of Valence bonds (or a singlet) using the Majumdar-Ghosh Model.

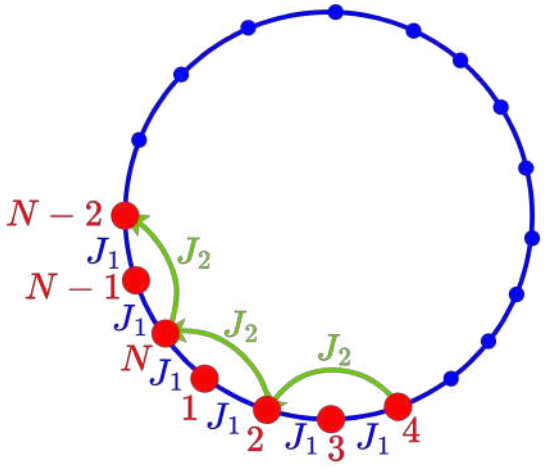


Figure 2.3: Majumdar Ghosh Model with Periodic Boundary Conditions.

The second nearest neighbour Heisenberg model, which is called the Majumdar–Ghosh Model [11], is represented as,

$$\mathcal{H} = J_1 \sum_i \vec{S}_i \cdot \vec{S}_{i+1} + J_2 \sum_i \vec{S}_i \cdot \vec{S}_{i+2}, \tag{2.4}$$

where $\vec{S}$ is quantum spin operator with quantum number $S = \frac{1}{2}$. This model has an analytical solution only at the MG point (i.e. $J_2 = \frac{J_1}{2}$). The ground states of the

MG Model are the tensor product of singlet states, which are given as,

$$|R_N\rangle = [1\ 2][3\ 4][5\ 6]\cdots[N-1\ N], \tag{2.5}$$

$$|C_N\rangle = [2\ 3][4\ 5][5\ 6]\cdots[N\ 1], \tag{2.6}$$

where,

$$[1\ 2] = \frac{1}{\sqrt{2}}[|\uparrow\downarrow\rangle_{12} - |\downarrow\uparrow\rangle_{12}]. \tag{2.7}$$

The two-qubit state in Eq. [2.7] is a singlet state, let us verify this,

$$\begin{aligned} S^2_{tot}[1\ 2] &= (\vec{S_1} + \vec{S_2})^2[1\ 2] = S(S+1)[1\ 2] \\ &= 0(0+1)[1\ 2] = 0[1\ 2]. \\ S^2_{tot}[1\ 2] &= 0[1\ 2]. \end{aligned} \tag{2.8}$$

Since the Eq. [2.8] is an eigenvalue equation for the S^2 operator, the eigenvalue corresponding to it is 0, so it is a singlet state.

The states in Eq. [2.5] and Eq. [2.6] are represented as,

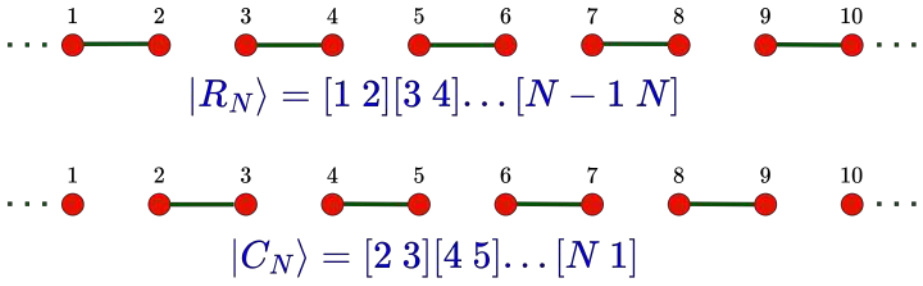


Figure 2.4: Ground State of the MG Model at ($J_2 = \frac{J_1}{2}$)

The bonds shown in green colour in Fig. [2.4] are called valence bonds or a singlet.

Now, let us extend the concept of valence bonds for higher spins. If a particle has spin S, then this spin can be constructed by $2S$ spin-$\frac{1}{2}$ particles. For example, if we have spin $S = 1$ particle, that can be constructed by $2 \times 1 = 2$ spin-$\frac{1}{2}$ particles, as $\frac{1}{2} + \frac{1}{2} = 1$ by angular momentum algebra of quantum mechanics.

Let us consider a linear 1-D lattice consisting of N sites. Between each pair of sites,

there is a valence bond. So each site will have two spin-half particles, leading to a pair of spin-1 particles. So if we have N number of spin-1 sites, that can be constructed using $2N$ number of spin-half particles.

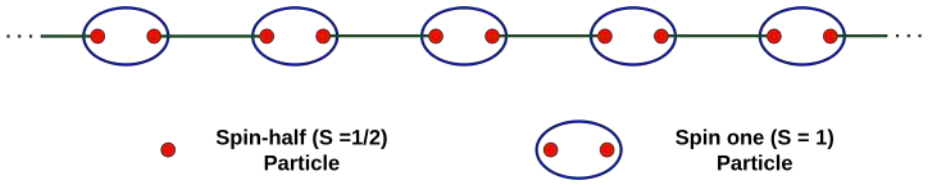


Figure 2.5: Ground State of AKLT Model

Unlike the dimerized state in Fig. [2.4], the state in Fig. [2.5] has unbroken translational symmetry.

Now we can construct the Hamiltonian for which this can be the ground state, also the presence of a valence bond between each neighbouring pair implies that the total spin of such pairs cannot be 2.

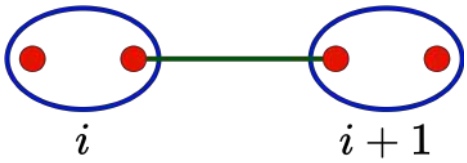


Figure 2.6: Neighboring site interaction

Let us try to understand the previous statement using the neighbouring pair i and $i+1$ as shown in Fig. [2.6]. Two of the spin-half particles are in the singlet state (particles connected by a green line). So the remaining two spin-half particles will combine to form the total spin S, as,

$$\begin{aligned} S &= |s_1 + s_2|, \cdots, |s_1 - s_2| \\ &= \left|\frac{1}{2} + \frac{1}{2}\right|, \cdots, \left|\frac{1}{2} - \frac{1}{2}\right| \end{aligned} \tag{2.9}$$

$$S = 1 \text{ and } 0. \tag{2.10}$$

Hence, the pair of sites i and $i+1$ cannot have total spin $S = 2$. Thus, we can choose our Hamiltonian to be the sum of projection operators onto Spin $S = 2$ for

each neighbouring pair as,

$$\mathcal{H} = \sum_i P_2(\vec{S}_i + \vec{S}_{i+1}), \tag{2.11}$$

where P_2 is a projection operator on spin $S = 2$. Now, one may think of a question: why we did not choose projection operators onto spin S=$\frac{3}{2}$?

It is because the neighboring sites i and $i+1$ are $S = 1$ particles, so, $S_i = 1$ and $S_{i+1} = 1$. Hence, the outcomes for total spin are 2, 1, and 0. So the ground state, which is represented in Fig. [2.4] and Fig. [2.5], has to have a Hamiltonian made of projectors onto spin $S = 2$. The eigenvalues of the projection operator P_2 are 0 and 1.

Now we try to find the expression of the projection operator P_2 in Eq. [2.11]. As we know total spin operator for two neighbouring spin sites ($S_i = 1$) can be written as,

$$\vec{S}_{tot} = \vec{S}_i + \vec{S}_{i+1}. \tag{2.12}$$

$$X = \vec{S}_{tot}.\vec{S}_{tot} = S^2_{tot} = \left(\vec{S}_i + \vec{S}_{i+1}\right)^2.$$

$$X = S^2_{tot} = \left(\vec{S}_i + \vec{S}_{i+1}\right)^2. \tag{2.13}$$

Now, we represent $S = 0$ state as $|0\rangle$, $S = 1$ state as $|1\rangle$ and $S = 2$ state as $|2\rangle$. If operator $X(S^2_{tot})$ operates on spin state $|S\rangle$, then we observe,

$$X|S\rangle = S(S+1)|S\rangle. \tag{2.14}$$

Using Eq. [2.14], we can calculate the following as,

$$X|0\rangle = 0(0+1)|0\rangle = 0|0\rangle, \tag{2.15}$$

$$X|1\rangle = 1(1+1)|1\rangle = 2|1\rangle, \tag{2.16}$$

$$X|2\rangle = 2(2+1)|2\rangle = 6|2\rangle. \tag{2.17}$$

Hence, the matrix representation of X operator can be written as,

$$X = \begin{bmatrix} \langle 0|X|0\rangle & \langle 0|X|1\rangle & \langle 0|X|2\rangle \\ \langle 1|X|0\rangle & \langle 1|X|1\rangle & \langle 1|X|2\rangle \\ \langle 2|X|0\rangle & \langle 2|X|1\rangle & \langle 2|X|2\rangle \end{bmatrix},$$

$$X = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 6 \end{bmatrix}. \tag{2.18}$$

Let us calculate the term $X(X-2I)$,

$$X(X-2I) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 6 \end{bmatrix} \begin{bmatrix} -2 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 4 \end{bmatrix},$$

$$X(X-2I) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 24 \end{bmatrix}. \tag{2.19}$$

From Eq. [2.19] we can write P_2 as,

$$P_2 = \frac{1}{24}X(X-2I) = \frac{1}{24}|2\rangle\langle 2|. \tag{2.20}$$

Similarly, we can also calculate $(X-2I)(X-6I)$ as,

$$(X-2I)(X-6I) = \begin{bmatrix} -2 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 4 \end{bmatrix} \begin{bmatrix} -6 & 0 & 0 \\ 0 & -4 & 0 \\ 0 & 0 & 0 \end{bmatrix},$$

$$(X-2I)(X-6I) = \begin{bmatrix} 12 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}. \tag{2.21}$$

From Eq. [2.21] we can write P_0 as,

$$P_0 = \frac{1}{12}(X-2I)(X-6I) = \frac{1}{12}|0\rangle\langle 0|. \tag{2.22}$$

Similarly, we can also calculate $X(X-6I)$ as,

$$X(X-6I) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 6 \end{bmatrix} \begin{bmatrix} -6 & 0 & 0 \\ 0 & -4 & 0 \\ 0 & 0 & 0 \end{bmatrix},$$

$$X(X-6I) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & -8 & 0 \\ 0 & 0 & 0 \end{bmatrix}. \tag{2.23}$$

From Eq. [2.23] we can write P_1 as,

$$P_1 = -\frac{1}{8}X(X-6I) = -\frac{1}{8}|1\rangle\langle 1|. \tag{2.24}$$

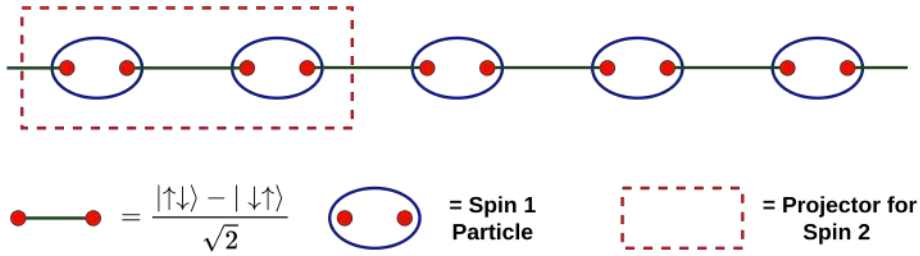


Figure 2.7: Ground State of AKLT Model

One can imagine the P_2 state as shown in Fig. [2.7]. Now we calculate the algebraic expression of Eq. [2.20] as,

$$\begin{aligned} P_2 &= \frac{1}{24}[X^2 - 2X], \\ &= \frac{1}{24}[(\vec{S}_i + \vec{S}_j)^2(\vec{S}_i + \vec{S}_j)^2 - 2(\vec{S}_i + \vec{S}_j)^2], \\ &= \frac{1}{24}[(S_i^2 + S_j^2 + 2\vec{S}_i.\vec{S}_j)(S_i^2 + S_j^2 + 2\vec{S}_i.\vec{S}_j) - 2(S_i^2 + S_j^2 + 2\vec{S}_i.\vec{S}_j)]. \end{aligned} \tag{2.25}$$

As we are working with spin $S = 1$, so $S_i^2 = 1(1+1) = 2$. So,

$$\begin{aligned} P_2 &= \frac{1}{24}[(2 + 2 + 2\vec{S}_i.\vec{S}_j)(2 + 2 + 2\vec{S}_i.\vec{S}_j) - 2(2 + 2 + 2\vec{S}_i.\vec{S}_j)], \\ &= \frac{1}{24}[(4 + 2\vec{S}_i.\vec{S}_j)(4 + 2\vec{S}_i.\vec{S}_j) - 2(4 + 2\vec{S}_i.\vec{S}_j)], \end{aligned}$$

$$
\begin{aligned}
P_2 &= \frac{1}{24}[(16 + 16\vec{S}_i.\vec{S}_j + 4(\vec{S}_i.\vec{S}_j)^2) - (8 + 4\vec{S}_i.\vec{S}_j)], \\
&= \frac{1}{24}[8 + 12\vec{S}_i.\vec{S}_j + 4(\vec{S}_i.\vec{S}_j)^2], \\
&= \left[\frac{1}{3} + \frac{1}{2}(\vec{S}_i.\vec{S}_j) + \frac{1}{6}(\vec{S}_i.\vec{S}_j)^2\right].
\end{aligned} \tag{2.26}
$$

So, The Hamiltonian ($\mathcal{H}$) can be defined as,

$$
\mathcal{H} = \sum_i \left[\frac{1}{3} + \frac{1}{2}\vec{S}_i.\vec{S}_{i+1} + \frac{1}{6}(\vec{S}_i.\vec{S}_{i+1})^2\right]. \tag{2.27}
$$

where $\vec{S}_i$ is a spin half operator acting on i^{th} spin, where $\vec{S}_i$ is defined as,

$$
\vec{S}_i = S_i^x\hat{x} + S_i^y\hat{y} + S_i^z\hat{z}. \tag{2.28}
$$

where S_i^x, S_i^y and S_i^z are spin components as usual.
The constant term in the Hamiltonian will not affect the properties of the Hamiltonian; it will just shift the properties by some range, so that we can take this term as a reference point. So, the Hamiltonian can be written as [5],

$$
\mathcal{H} = \sum_i \left[\frac{1}{2}\vec{S}_i.\vec{S}_{i+1} + \frac{1}{6}(\vec{S}_i.\vec{S}_{i+1})^2\right]. \tag{2.29}
$$

Now, if we see the above expression, there is a common factor of $\frac{1}{2}$ in both the terms of the Hamiltonian, which will just scale the things and again not going to change the nature of the Hamiltonian. So we can again take it common and neglect it. So the Hamiltonian will finally boil down to,

$$
\mathcal{H} = \sum_i \left[\vec{S}_i.\vec{S}_{i+1} + \frac{1}{3}(\vec{S}_i.\vec{S}_{i+1})^2\right]. \tag{2.30}
$$

Now, if each qubit has a spin angular momentum of half (i.e., $S = \frac{1}{2}$), this leads to a total spin angular momentum quantum number [12],

$$
S = \begin{cases} 0 & \text{(singlet)} \\ 1 & \text{(triplet)} \end{cases},
$$

We know,

$$\vec{S_1} \cdot \vec{S_2} = \frac{(S^2 - S_1^2 - S_2^2)}{2}. \tag{2.31}$$

For the triplet case ($S = 1$),

$$\vec{S_1} \cdot \vec{S_2} = \left[\frac{1(1+1) - \frac{1}{2}(1+\frac{1}{2}) - \frac{1}{2}(1+\frac{1}{2})}{2}\right]\hbar^2 = \frac{1}{4}\hbar^2. \tag{2.32}$$

For the singlet case ($S = 0$),

$$\vec{S_1} \cdot \vec{S_2} = \left[\frac{0(0+1) - \frac{1}{2}(1+\frac{1}{2}) - \frac{1}{2}(1+\frac{1}{2})}{2}\right]\hbar^2 = -\frac{3}{4}\hbar^2. \tag{2.33}$$

Again, in terms of the Pauli spin matrix, we can write $\vec{S_1} = \frac{\hbar}{2}\vec{\sigma_1}$ and $\vec{S_2} = \frac{\hbar}{2}\vec{\sigma_2}$ [13]. This gives us,

$$\vec{\sigma_1} \cdot \vec{\sigma_2} = \frac{4}{\hbar^2}\vec{S_1} \cdot \vec{S_2} = \frac{4}{\hbar^2}.\frac{1}{4}\hbar^2 = 1, \qquad \text{(for triplet)} \tag{2.34}$$

$$\vec{\sigma_1} \cdot \vec{\sigma_2} = \frac{4}{\hbar^2}\vec{S_1} \cdot \vec{S_2} = -\frac{4}{\hbar^2}.\frac{3}{4}\hbar^2 = -3, \qquad \text{(for singlet)} \tag{2.35}$$

So let us rewrite our Hamiltonian in terms of Pauli Matrices,

$$\mathcal{H} = \frac{\hbar^2}{4}\sum_i \vec{\sigma}_i.\vec{\sigma}_{i+1} + \frac{1}{3}\left(\frac{\hbar^2}{4}\right)^2 \sum_i (\vec{\sigma}_i.\vec{\sigma}_{i+1})^2. \tag{2.36}$$

Now we can choose $\hbar = 1$ for convenience, then our Hamiltonian becomes,

$$\mathcal{H} = \frac{1}{4}\sum_i \vec{\sigma}_i.\vec{\sigma}_{i+1} + \frac{1}{48}\sum_i (\vec{\sigma}_i.\vec{\sigma}_{i+1})^2. \tag{2.37}$$

Since we know the spin for each particle, $\mathbf{s} = \frac{1}{2}$, we can calculate the number of basis states as

$$N = 2s + 1, \tag{2.38}$$

Substitute the value of $\mathbf{s}$ in Eq. [2.38], the total number of basis states will be 2. The basis states for this system are $|0\rangle$ and $|1\rangle$, which are the eigenvectors of σ_z with

eigenvalues 1 and -1, respectively.

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \tag{2.39}$$

In the Hamiltonian, for considering the interaction along different directions, we can use the fact that σ_i^α with $\alpha = \{x, y, z\}$ are spin half operators acting on i^{th} spin, where σ^x, σ^y and σ^z are Pauli matrices about which we have discussed in Sec. [1.2].

Now, if we simplify the Hamiltonian in Eq. [2.37], it will look like below,

$$\mathcal{H} = \frac{1}{4}\sum_i(\sigma_i^x\sigma_{i+1}^x + \sigma_i^y\sigma_{i+1}^y + \sigma_i^z\sigma_{i+1}^z) + \frac{1}{48}\sum_i(\sigma_i^x\sigma_{i+1}^x + \sigma_i^y\sigma_{i+1}^y + \sigma_i^z\sigma_{i+1}^z)^2. \tag{2.40}$$

We can place an external parameter called Interaction Energy (J) and vary that parameter to study the system's behaviour. This parameter defines the energy by which the particles interact with each other via quantum exchange interaction. So the Hamiltonian becomes,

$$\mathcal{H} = \frac{J}{4}\sum_i(\sigma_i^x\sigma_{i+1}^x + \sigma_i^y\sigma_{i+1}^y + \sigma_i^z\sigma_{i+1}^z) + \frac{J}{48}\sum_i(\sigma_i^x\sigma_{i+1}^x + \sigma_i^y\sigma_{i+1}^y + \sigma_i^z\sigma_{i+1}^z)^2. \tag{2.41}$$

Now we place another term, interaction due to magnetic field (B), we can study the system's behaviour with a varying external magnetic field. So the Hamiltonian will become,

$$\mathcal{H} = \frac{J}{4}\sum_i(\sigma_i^x\sigma_{i+1}^x + \sigma_i^y\sigma_{i+1}^y + \sigma_i^z\sigma_{i+1}^z) + \frac{J}{48}\sum_i(\sigma_i^x\sigma_{i+1}^x + \sigma_i^y\sigma_{i+1}^y + \sigma_i^z\sigma_{i+1}^z)^2 + B\sum_i\sigma_i^z\sigma_{i+1}^z. \tag{2.42}$$

Let us see how to build the Hamiltonian numerically for a general number of qubits, so that we can study the behaviour of the system for varying interaction energy (J) and external magnetic field (B).

2.4 Building the Hamiltonian and Computing Eigenvalues and Eigenvectors

Similar to the Ising model, we will discuss two approaches for computing the Hamiltonian and its eigenvalues and eigenvectors. Again, the tensor product method is quite old and tedious and requires so much memory allocation and computational time, we use an alternative method to build the Hamiltonian and compute its eigenvalues and eigenvectors.

2.4.1 Tensor Product Approach

This is the old and tedious method to compute the Hamiltonian, which requires more memory allocation and computation time, making it difficult to compute the Hamiltonian for higher qubit systems. Let us expand the Hamiltonian and see how it looks for the i^{th} qubit,

$$\begin{aligned}\mathcal{A} =&\langle a_1 \cdots a_i a_{i+1} \cdots a_n | I_1 \otimes \cdots S_i^x \otimes S_{i+1}^x \otimes \cdots I_n | a_1 \cdots a_i a_{i+1} \cdots a_n \rangle \\ &+ \langle a_1 \cdots a_i a_{i+1} \cdots a_n | I_1 \otimes \cdots S_i^y \otimes S_{i+1}^y \otimes \cdots I_n | a_1 \cdots a_i a_{i+1} \cdots a_n \rangle \\ &+ \langle a_1 \cdots a_i a_{i+1} \cdots a_n | I_1 \otimes \cdots S_i^z \otimes S_{i+1}^z \otimes \cdots I_n | a_1 \cdots a_i a_{i+1} \cdots a_n \rangle.\end{aligned} \tag{2.43}$$

Here $\mathcal{A}$ is an energy term, and is the expansion of the first term of the Hamiltonian as defined in Eq. [2.30], which we have calculated earlier, to calculate the complete Hamiltonian easily. Then we have to compute the square term for each i^{th} qubit. Now let us write the complete Hamiltonian,

$$\mathcal{H} = \mathcal{A} + \frac{1}{3}\mathcal{A}^2. \tag{2.44}$$

Now, if you notice how difficult it is to write a complete Hamiltonian even for an i^{th} term, you can estimate the robustness of this method for computing a complete Hamiltonian. It can be clearly seen that the $\vec{S}_i$ and $\vec{S}_{i+1}$ will only act on i^{th} and $(i+1)^{th}$ qubit respectively. Now, if we look for a complete Hamiltonian, we have to calculate such 2^N terms for linear and quadratic terms, which is a horrible task for

higher qubit systems. Let us show a Fortran program that will help you to imagine the difficulty of this method to build the Hamiltonian for higher qubit systems. In the program given below, we have built the Hamiltonian and computed eigenvalues and eigenvectors for a 4-qubit system.

```
program spin1tensorp
    implicit none
    integer, parameter :: s = 4     !Total number of qubits
    integer, parameter :: num = 2**s        !Total number of states
    integer :: k, l, g
    complex :: I(2,2), S_x(2,2), S_z(2,2), S_y(2,2)
    complex :: D_1(num,num),D_2(num,num),D_3(num,num),D_4(num,num)
    complex :: D_5(num,num),D_6(num,num),D_7(num,num),D_8(num,num)
    complex :: D_9(num,num),D_10(num,num),D_11(num,num)
    complex :: D_13(num,num),D_14(num,num),D_15(num,num)
    complex :: D_17(num,num),D_12(num,num),D_16(num,num)
    real(8) :: D_18(num,num)
    double precision :: W(num), WORK(3*num), H(num,num), D(num,num)
    integer ::  LDA = num, LWORK = 3*num, INFO
    character*1 :: JOBZ, UPLO
    real*8 :: sum
    ! Initialize H to zero
    H = 0.0
    ! Defining Pauli Matrix sigma_0
    I(1, 1) = (1.0,0.0)
    I(1, 2) = (0.0,0.0)
    I(2, 1) = (0.0,0.0)
    I(2, 2) = (1.0,0.0)

    ! Defining Pauli Matrix sigma_x
    S_x(1, 1) = (0.0,0.0)
    S_x(1, 2) = (1.0,0.0)
    S_x(2, 1) = (1.0,0.0)
    S_x(2, 2) = (0.0,0.0)

    ! Defining Pauli Matrix sigma_x
    S_y(1, 1) = (0.0,0.0)
```

```
33      S_y(1, 2) = (0.0,-1.0)
34      S_y(2, 1) = (0.0,1.0)
35      S_y(2, 2) = (0.0,0.0)
36
37      ! Defining Pauli Matrix sigma_z
38      S_z(1, 1) = (1.0,0.0)
39      S_z(1, 2) = (0.0,0.0)
40      S_z(2, 1) = (0.0,0.0)
41      S_z(2, 2) = (-1.0,0.0)
42      ! Calling function
43      call tensorprod(S_x, S_x, I, I, D_1)
44      ! write(2, *) 'D_1 = '
45      ! do k = 1, num
46      !     do l = 1, num
47      !         write(2,*) k, l, D_1(k,l)
48      !     end do
49      ! end do
50      !write(2, *) D_1
51      call tensorprod(I, S_x, S_x, I, D_2)
52      ! write(2, *) 'D_2 = '
53      ! write(2, *) D_2
54      call tensorprod(I, I, S_x, S_x, D_3)
55      ! write(2, *) 'D_3 = '
56      ! write(2, *) D_3
57      call tensorprod(S_x, I, I, S_x, D_4)
58      ! Calling function
59      call tensorprod(S_y, S_y, I, I, D_5)
60      ! write(2, *) 'D_4 = '
61      ! write(2, *) D_4
62      call tensorprod(I, S_y, S_y, I, D_6)
63      ! write(2, *) 'D_5 = '
64      ! write(2, *) D_5
65      call tensorprod(I, I, S_y, S_y, D_7)
66      ! write(2, *) 'D_6 = '
67      ! write(2, *) D_6
68      ! Calling function
69      call tensorprod(S_y, I, I, S_y, D_8)
70      ! write(2, *) 'D_7 = '
```

```
71      ! write(2, *) D_7
72      call tensorprod(S_z, S_z, I, I, D_9)
73      ! write(2, *) 'D_8 = '
74      ! write(2, *) D_8
75      call tensorprod(I, S_z, S_z, I, D_10)
76      ! write(2, *) 'D_9 = '
77      ! write(2, *) D_9
78      call tensorprod(I, I, S_z, S_z, D_11)
79      call tensorprod(S_z, I, I, S_z, D_12)
80      D_13 = D_1 + D_5 + D_9
81      D_14 = D_2 + D_6 + D_10
82      D_15 = D_3 + D_7 + D_11
83      D_16 = D_4 + D_8 + D_12
84      D_17 = matmul(D_13,D_13) + matmul(D_14,D_14) + matmul(D_15,D_15)
        + matmul(D_16,D_16)
85      ! do k = 1, num
86      !     do l = 1, num
87      !         D_18(k,l) = D_8(k,l) !+ D_14(k,l) + D_15(k,l) + D_16(k
        ,l)
88      !         write(2,*) k, l, D_18(k,l)
89      !     end do
90      ! end do
91      ! write(2, *) D_13 + D_14 + D_15 + D_16
92      ! Calculate Hamiltonian H
93      !write(2,*) 'Hamiltonian Matrix :'
94      do k = 1, num
95          do l = 1, num
96              H(k, l) = (D_13(k,l) + D_14(k,l) + D_15(k,l) + D_16(k,l)
        ) + (1.0d0/3.0d0)*(D_17(k,l))
97              !print*, "H(", k, ",", l, ") =", H(k, l)
98              write(2, *) k, l, H(k, l)
99          end do
100     end do
101
102     ! Define parameters for DSYEV
103     JOBZ = 'V'  ! Compute eigenvalues and eigenvectors
104     UPLO = 'U'  ! Upper triangular part of A is stored
105     ! Call DSYEV to compute eigenvalues and eigenvectors
```

```
106     call DSYEV(JOBZ, UPLO, num, H, LDA, W, WORK, LWORK, INFO)
107     ! Check for successful execution
108     sum = 0.0d0
109     if (INFO == 0) then
110         write(2,*) 'Eigenvalues are:'
111         do g = 1, num
112             write(2,*) W(g)
113             sum = sum + W(g)
114         end do
115     else
116         print*, 'Error in DSYEV, info =', INFO
117     end if
118     !print*, sum
119     !write(2,*) 'Eigen Vectors are :'
120     do k = 1, num
121         !write(2,*) 'Eigen Vector ', k, ':'
122         do l = 1, num
123             write(4,*) k, l, H(l,k)
124         end do
125     end do
126 contains
127 
128     subroutine tensorprod(A, B, C, D, E)
129         implicit none
130         complex, intent(in) :: A(2, 2), B(2, 2), C(2, 2), D(2,2)
131         complex, intent(out) :: E(16, 16)
132         complex :: temp1(4, 4), temp2(8,8)
133         integer :: p, q, k, l
134         ! Initialize the result matrix
135         E = (0.0,0.0)
136         ! Calculate the tensor product of A and B first
137         do p = 1, 2
138             do q = 1, 2
139                 do k = 1, 2
140                     do l = 1, 2
141                         temp1((p-1)*2 + k, (q-1)*2 + l) = A(p, q) *
    B(k, l)
142                     end do
```

```
                end do
            end do
        end do
        ! Now calculate the tensor product of temp and C
        do p = 1, 4
            do q = 1, 4
                do k = 1, 2
                    do l = 1, 2
                        temp2((p-1)*2 + k, (q-1)*2 + l) = temp1(p, q
    ) * C(k, l)
                    end do
                end do
            end do
        end do
        ! Now calculate the tensor product of temp and C
        do p = 1, 8
            do q = 1, 8
                do k = 1, 2
                    do l = 1, 2
                        E((p-1)*2 + k, (q-1)*2 + l) = temp2(p, q) *
    D(k, l)
                    end do
                end do
            end do
        end do
     end subroutine tensorprod
end program

```

Now, if you look into the program for building the Hamiltonian for the 4-qubit system itself, it requires many matrices and has dimensions $2^4 \times 2^4 = 16 \times 16$, which increases exponentially if we use this method for building the Hamiltonian for higher-order qubit systems. As the memory allocation increases, the computation time also increases. So it became our necessity to find an alternative method, as we have seen in the case of the Ising model, to compute the Hamiltonian for higher qubit systems.

2.4.2 Bit Operation Approach

We have found a general pattern for the operation of Pauli matrices on qubits in the Ising model (see Sec. [1.3.2]), which we can use to build the Hamiltonian of the AKLT model for the general particle system. Let us implement this in a Fortran program.

```
program spin1bitop
implicit none
integer, parameter :: d =2, s = 4, s1 = 2
integer, parameter :: num = d**s, N = num, di = d**s1
integer :: i, j, k, l, t, u, flagx, flagy
integer, allocatable :: digits1(:), digits2(:), decimal(:)
character(len=:), allocatable :: state(:)
double precision, allocatable :: E(:,:), C(:,:), F(:,:), G(:,:), H
    (:,:)
character*1 :: JOBZ, UPLO
integer ::  LDA = num, LWORK = 3*num, INFO
double precision :: W(num), WORK(3*num)
real*8 :: sum, Jo, B
! Allocate arrays after determining the value of num
allocate(character(len=s) :: state(num))
allocate(integer :: decimal(num))
allocate(integer :: digits1(s), digits2(s))
allocate(real(8) :: C(num, num))
allocate(real(8) :: E(num, num))
allocate(real(8) :: F(num, num))
allocate(real(8) :: G(num, num))
allocate(real(8) :: H(0:num-1 , 0:num-1))
do i = 0, num - 1
   call integer_binary(i, state(i+1), s)  ! Adjusted for 1-based
   indexing
   !print *, state(i)
end do
do i = 1, num
    decimal(i) = btod(state(i))
    !print*, ' Decimal equivalent of ', state(i) ,' = ', decimal(i)
```

```
end do
G = 0.0d0
do k =1, s
    E = 0.0d0
    do i = 1, num
        do j = 1, num
        ! Computing Sigma_i^x * Sigma_i+1^x for each site
        flagx = 0
        if   (decimal(i) .ne. decimal(j)) then
                    call sd(state(i), digits1)
                    call sd(state(j), digits2)
        if (k < s) then
            if (digits1(k) .ne. digits2(k) .and. digits1(k+1) .ne.
   digits2(k+1)) then
            do l = 1, k - 1
                if (digits1(l) .ne. digits2(l)) then
                    flagx  = flagx + 1
                end if
            end do
            do l = k + 2, s
                if (digits1(l) .ne. digits2(l)) then
                    flagx  = flagx + 1
                end if
            end do
                if (flagx == 0) then
                    E(i, j) = E(i, j) + 1.0
                    else
                    E(i, j) = E(i, j) + 0.0
                end if
                else
                    E(i, j) = E(i, j) + 0.0
            end if
            else
            if (digits1(1) .ne. digits2(1) .and. digits1(s) .ne.
   digits2(s)) then
                do l = 2, s - 1
                    if (digits1(l) .ne. digits2(l)) then
                        flagx  = flagx + 1
```

```
65                      end if
66                  end do
67                      if (flagx == 0) then
68                          E(i, j) = E(i, j) + 1.0
69                      else
70                          E(i, j) = E(i, j) + 0.0
71                      end if
72              else
73                  E(i, j) = E(i, j) + 0.0
74              end if
75          end if
76          else
77              E(i, j) = E(i, j) + 0.0
78      end if
79                  ! Computing Sigma_i^y * Sigma_i+1^y for each site
80                  flagy = 0
81                  if (decimal(i) .ne. decimal(j)) then
82                      call sd(state(i), digits1)
83                      call sd(state(j), digits2)
84                      if (k < s) then
85                          if (digits1(k) .ne. digits2(k) .and. digits1
    (k+1) .ne. digits2(k+1)) then
86                              do l = 1, k - 1
87                                  if (digits1(l) .ne. digits2(l)) then
88                                      flagy  = flagy + 1
89                                  end if
90                              end do
91                              do l = k + 2, s
92                                  if (digits1(l) .ne. digits2(l)) then
93                                      flagy  = flagy + 1
94                                  end if
95                              end do
96                              if (flagy == 0) then
97                                  E(i, j) = E(i, j) + (-1.0 * (-1.0)
    **(digits2(k) + digits2(k+1)))
98                              else
99                                  E(i, j) = E(i, j) + 0.0
100                             end if
```

```
                        else
                            E(i, j) = E(i, j) + 0.0
                        end if
                    else
                        if (digits1(1) .ne. digits2(1) .and. digits1
(s) .ne. digits2(s)) then
                            do l = 2, s - 1
                                if (digits1(l) .ne. digits2(l)) then
                                    flagy  = flagy + 1
                                end if
                            end do
                            if (flagy == 0) then
                                E(i, j) = E(i, j)  + (-1.0 * (-1.0)
**(digits2(s) + digits2(1)))
                            else
                                E(i, j) = E(i, j) + 0.0
                            end if
                        else
                            E(i, j) = E(i, j) + 0.0
                        end if
                    end if
                else
                    E(i, j) = E(i, j) + 0.0
                end if
                ! Computing Sigma_i^z * Sigma_i+1^z for each site
                call sd(state(i), digits1)
                if (decimal(i) .ne. decimal(j)) then
                    E(i, j) =  E(i,j) + 0.0
                else
                    if (k < s) then
                        E(i, j) = E(i, j) + (1 - 2 * digits1(k)) *
(1 - 2 * digits1(k+1))
                    else
                        E(i, j) = E(i, j) + (1 - 2 * digits1(1)) *
(1 - 2 * digits1(s))
                    end if
                end if
                ! Computing Sigma_i^z for each site
```

```
135                 call sd(state(i), digits1)
136                 if (decimal(i) .ne. decimal(j)) then
137                     G(i, j) =  G(i,j) + 0.0
138                 else
139                     G(i, j) = G(i, j) + (1 - 2 * digits1(k))
140                 end if
141             end do
142         end do
143      C = C + (1.0d0/4.0d0)*E
144      F = F + matmul((1.0d0/4.0d0)*E,(1.0d0/4.0d0)*E)
145      end do
146      ! write(58,*) C
147      do u = 0, 200
148          Jo = 0.01 * u
149          ! do t = 0, 200                                                    !
    Loop for varying B
150          !     B =  0.01 * t
151              B = 0.0d0
152              H = 0.0d0
153              do i = 1, num
154                  do j = 1, num
155                      H(i-1,j-1) = Jo * (C(i,j) + (1.0d0/3.0d0)* F(i,j
    )) + B * (1.0d0/2.0d0)* G(i,j)
156                      !if (i .le. j) then
157                          ! write(57,*) i, j, H(i-1,j-1)
158                      !end if
159                  end do
160              end do
161              ! Define parameters for DSYEV
162              JOBZ = 'V'  ! Compute eigenvalues and eigenvectors
163              UPLO = 'U'  ! Upper triangular part of A is stored
164              ! Call DSYEV to compute eigenvalues and eigenvectors
165              call DSYEV(JOBZ, UPLO, num, H, LDA, W, WORK, LWORK, INFO
    )
166              ! Check for successful execution
167              sum = 0.0d0
168              if (INFO == 0) then
169                  ! write(4,*) 'Eigenvalues are:'
```

```
                do i = 1, num
                    write(3,*) Jo, W(i)
                    sum = sum + W(i)
                end do
            else
                print*, 'Error in DSYEV, info =', INFO
            end if
            ! print*, sum
            ! write(2,*) 'Eigen Vectors for J =', Jo, 'are :'
            ! do k = 0, num -1
            !     write(2,*) 'Eigen Vector ', k, ':'
            !     do l = 0, num -1
            !         write(2,*) k, l, H(l,k)
            !     end do
            ! end do
        end do
    ! end do
contains
    subroutine integer_binary(num, binary_string, n)
        implicit none
        integer :: num
        integer, intent(in) :: n
        integer :: i, temp_num
        character(len=n), intent(out) :: binary_string
        temp_num = num
        ! Initialize binary_string to all zeros
        binary_string = repeat('0', n)
        ! Convert the integer to binary
        do i = 0, n - 1
            if (mod(temp_num, 2) == 1) then
                binary_string(n-i:n-i) = '1'  ! Set the bit to '1'
            else
                binary_string(n-i:n-i) = '0'  ! Set the bit to '0'
            end if
            temp_num = temp_num / 2  ! Divide num by 2 to shift
    right
        end do
    end subroutine integer_binary
```

```
207     subroutine sd(binaryString, digits)
208         character(len=*), intent(in) :: binaryString
209         integer, allocatable, intent(out) :: digits(:)
210         integer :: i, len
211         len = len_trim(binaryString)
212         allocate(digits(len))
213         ! Convert each character to an integer
214         do i = 1, len
215             if (binaryString(i:i) == '1') then
216                 digits(i) = 1
217             else if (binaryString(i:i) == '0') then
218                 digits(i) = 0
219             else
220                 print *, "Invalid character in binary string."
221                 digits = 0
222                 return
223             end if
224         end do
225     end subroutine sd
226     function btod(binaryString) result(decimalValue)
227         implicit none
228         character(len=*), intent(in) :: binaryString
229         integer :: decimalValue
230         integer :: i, length
231         decimalValue = 0
232         length = len_trim(binaryString)
233         do i = 1, length
234             if (binaryString(i:i) == '1') then
235                 decimalValue = decimalValue + 2**(length - i)
236             else if (binaryString(i:i) /= '0') then
237                 print *, "Invalid character in binary string."
238                 decimalValue = -1  ! Indicate an error with -1
239                 return
240             end if
241         end do
242     end function btod
243 end program
244
```

In this program, we have defined s as a parameter that gives us the freedom to change the number of qubits and allows us to compute the Hamiltonian for a desired system. However, using this approach, we can compute up to s = 8 (i.e., 8-qubit system), but the computation time is still more (however, it is less when compared to the tensor product approach), which requires more optimization in the program to make it faster. Also, we are using the DSYEV subroutine, which comes under the LAPACK package, which allows us to compute all the eigenvalues and orthogonal eigenvectors that are not of interest to us till now. We only require the ground state, a few low-lying excited states for our further calculations, so using DSYEV is also not a good option. So we will see further approaches to make the program more faster with a modest computing power machine to compute the Hamiltonian and eigenvalues, as well as eigenvectors for higher qubit systems, with less computation time. But before that, let us study the complete energy eigenvalue spectrum of the system and the variation of the spectrum by varying the interaction energy (J) and the external magnetic field (B).

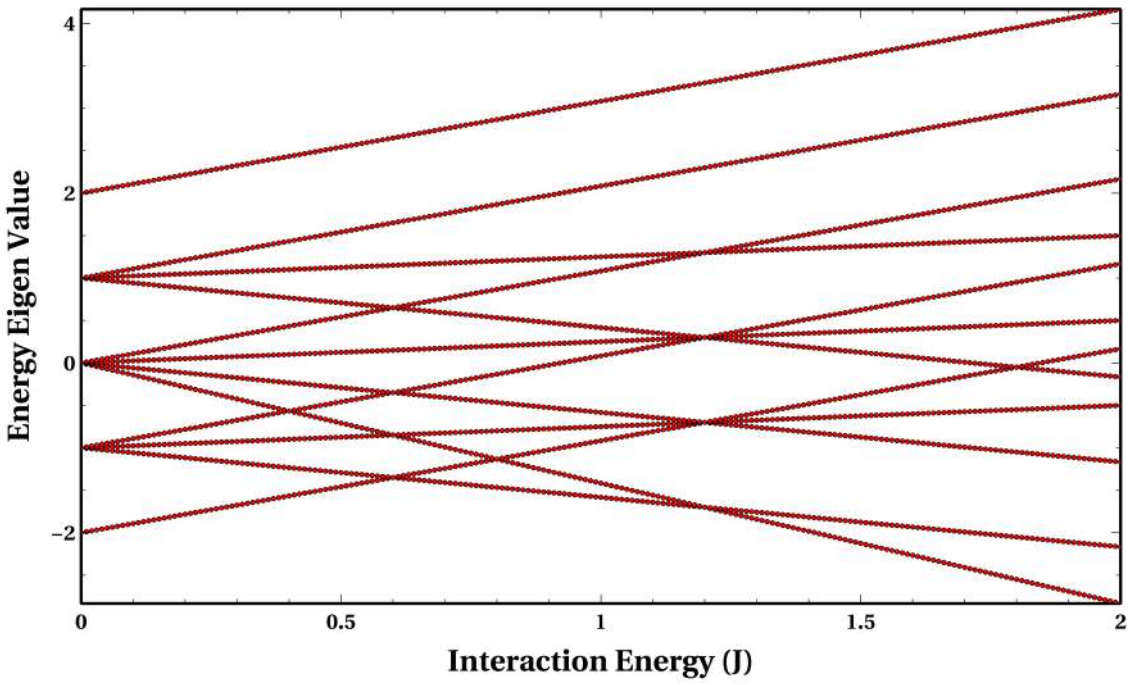


Figure 2.8: Energy eigenvalues vs interaction energy (J) for 4 qubits

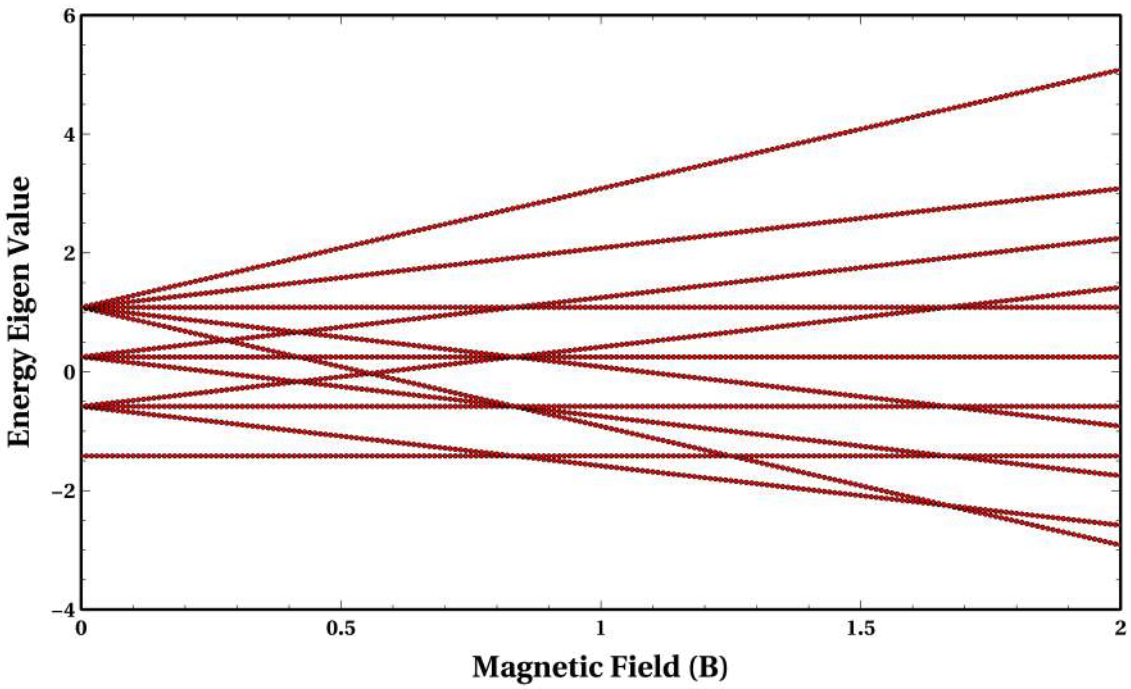


Figure 2.9: Energy eigenvalues vs magnetic field (B) for 4 qubits

For a 4-qubit periodic chain system, the eigenvalue spectrum corresponding to the change in the Interaction energy (for fixed $B = 1$) is shown in Fig. [2.8] and corresponding to the magnetic field (for fixed $J = 1$) is shown in Fig. [2.9]. In our Hamiltonian, as per equation Eq. [2.42], the magnetic field was assumed to be along the z-direction. We can see clearly from the plots that the variation of the energy eigenvalues with interaction energy, as well as the magnetic field, is a straight line in nature. This is kind of obvious because the Hamiltonian of the system involves σ^2 and σ_z terms, which commute with each other as [4],

$$[\sigma^2, \sigma_z] = [\sigma_x^2 + \sigma_y^2 + \sigma_z^2, \sigma_z] = [\sigma_x^2, \sigma_z] + [\sigma_y^2, \sigma_z] + [\sigma_z^2, \sigma_z]. \tag{2.45}$$

We know that,

$$[\sigma_x, \sigma_y] = i\sigma_z, \qquad [\sigma_y, \sigma_z] = i\sigma_x, \qquad [\sigma_z, \sigma_x] = i\sigma_y. \tag{2.46}$$

Now,

$$[\sigma_z^2, \sigma_z] = 0. \tag{2.47}$$

and

$$\begin{aligned}[\sigma_x^2, \sigma_z] &= \sigma_x[\sigma_x, \sigma_z] + [\sigma_x, \sigma_z]\sigma_x \\ &= \sigma_x(-i\sigma_y) + (-i\sigma_y)\sigma_x \\ &= -i\sigma_x\sigma_y - i\sigma_y\sigma_x. \end{aligned} \tag{2.48}$$

similarly,

$$\begin{aligned}[\sigma_y^2, \sigma_z] &= \sigma_y[\sigma_y, \sigma_z] + [\sigma_y, \sigma_z]\sigma_y \\ &= \sigma_y(i\sigma_x) + (i\sigma_x)\sigma_y \\ &= i\sigma_y\sigma_x + i\sigma_x\sigma_y. \end{aligned} \tag{2.49}$$

Substituting the values of all these separate commutations in Eq. [2.45], we can get,

$$[\sigma^2, \sigma_z] = [\sigma_x^2, \sigma_z] + [\sigma_y^2, \sigma_z] + [\sigma_z^2, \sigma_z] = 0. \tag{2.50}$$

If we consider ψ to be the eigenstate of the system, then we can get,

$$\langle\psi|\mathcal{H}|\psi\rangle = J \underbrace{\langle\psi| \sum_i \vec{\sigma}_i.\vec{\sigma}_{i+1}|\psi\rangle}_{a} + B \underbrace{\langle\psi_g| \sum_i \sigma_i^z|\psi_g\rangle}_{u}. \tag{2.51}$$

which turns out to be,

$$\mathcal{E} = aJ + uB. \tag{2.52}$$

where $\mathcal{E}$ is the energy eigenvalue of that particular state, and a and u are constants.

Now let us see the plots of ground state energy vs interaction energy and magnetic field for different numbers of qubits,

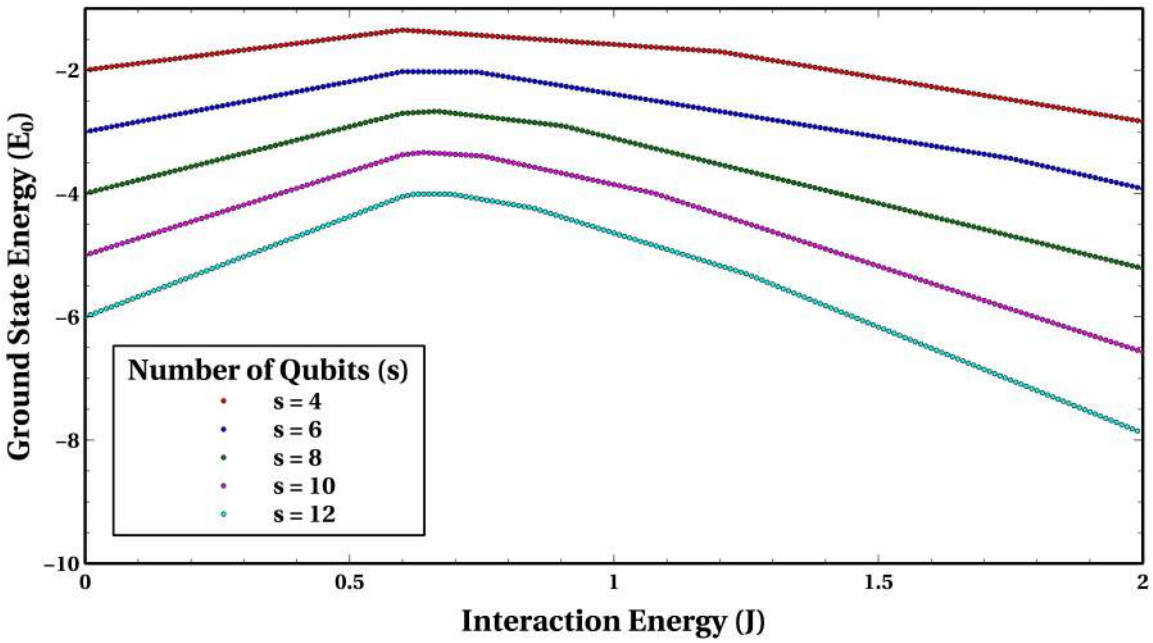


Figure 2.10: Ground state energy vs interaction energy (J) for different numbers of qubits (from $s = 4$ to $s = 12$).

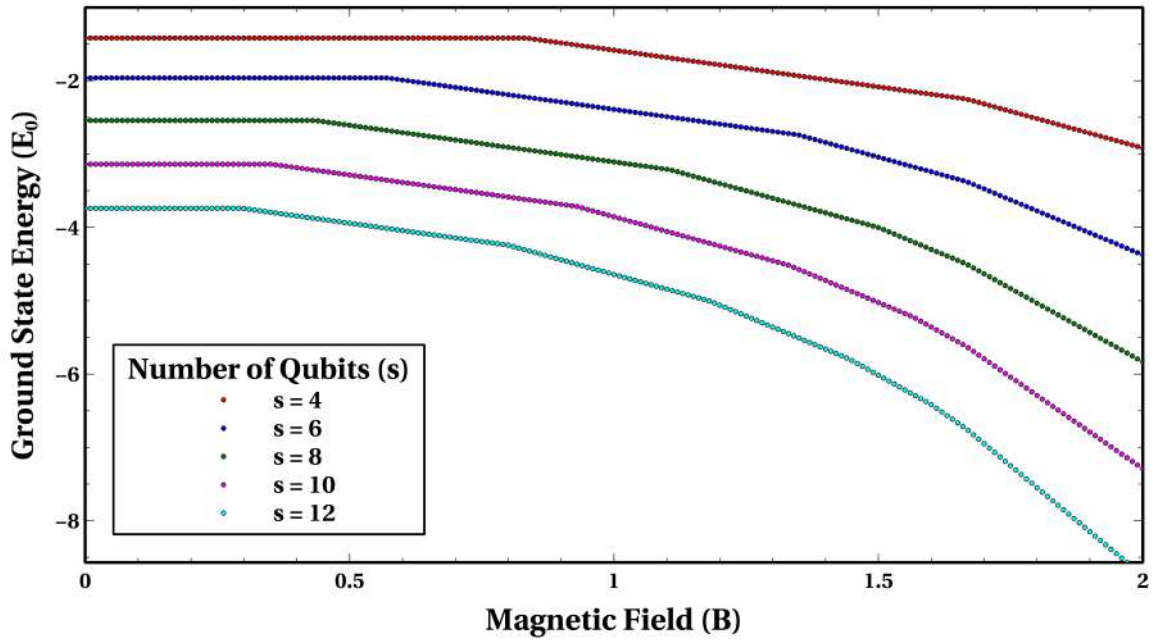


Figure 2.11: Ground state energy vs magnetic field (B) for different numbers of qubits (from $s = 4$ to $s = 12$).

So it is quite evident from the plots that as the number of qubits increases, the ground state energy decreases. With the increase in interaction energy, first the ground state energy increases and then it again decreases. With the increase in the magnetic field,

first the ground state energy remains constant, but then it starts to decrease however, that transition point (the point at which energy starts to decrease) is not fixed.

Chapter 3

Computational Approaches

3.1 Motivation

As we have discussed in the previous chapter, the bit operation approach is more efficient than the tensor product approach. However, it still takes more computation time for higher qubit systems, and also, we can compute only up to 8 qubits using this approach due to the huge memory allocation. So, if we want to build Hamiltonian, and compute eigenvalues, and eigenvectors for a higher qubit system, we have to think of further compatibility approaches without affecting the desired result. In this chapter, we are going to discuss such approaches that can help us to build Hamiltonian and compute eigenvalues and eigenvectors with an average machine power laptop. Also, we are going to give a time comparison of the approaches.

3.2 Using DSYEVX Subroutine

We have already mentioned the fact that we only require ground state, and a few low-lying excited states and their corresponding eigenvalues and eigenvectors for our further
calculations. So instead of using the DSYEV subroutine from the LAPACK package

[14], which calculates the full spectrum, we can use the DSYEVX subroutine, which gives us the ability to compute only the desired subset of eigenvalues and eigenvectors out of the full spectrum, which drastically reduces our computation time. Let us look into a Fortran program that we have written to build the Hamiltonian using our bit test approach, and then compute the ground state eigenvalues and eigenvectors using DSYEVX.

```
program spin1DSYEVX
implicit none
integer, parameter :: d =2, s = 4, s1 = 2
integer, parameter :: num = d**s, N = num, di = d**s1
integer :: i, j, k, l, t, flagx, flagy, pos, u
integer, allocatable :: digits1(:), digits2(:), decimal(:), site(:)
character(len=:), allocatable :: state(:)
double precision, allocatable :: E(:,:), C(:,:), F(:,:), G(:,:), H
    (:,:), Z(:,:)
character*1 :: JOBZ, UPLO, RANGE
nteger ::  LDA = num, LWORK = 8*num, INFO, IL, IU, M, LDZ, IWORK(5*N
    ), IFAIL(N)
double precision :: W(num), WORK(8*num), VL, VU, ABSTOL
real*8 :: sum, trace, con, Jo, B
real*8, allocatable :: psi(:,:), rho(:,:), SF(:,:), RT(:,:), RRT
    (:,:)
real*8, allocatable :: rdm(:,:), RW(:), RWORK(:) !,red_rho(:,:)
integer :: RLDA = di, RINFO, RLWORK = 3*di -1

! Allocate arrays after determining the value of num
allocate(character(len=s) :: state(num))
allocate(integer :: decimal(num))
allocate(integer :: digits1(s), digits2(s))
allocate(real(8) :: C(num, num))
allocate(real(8) :: E(num, num))
allocate(real(8) :: F(num, num))
allocate(real(8) :: G(num, num))
allocate(real(8) :: H(0:num-1 , 0:num-1))
allocate(real(8) :: Z(0:num-1 , 0:num-1))
```

```
allocate(psi(0:num-1,0:0))
allocate(rho(0:num-1, 0:num-1))
allocate(rdm(di, di))
! allocate(red_rho(0:di-1,0:di-1))
allocate(site(0:s1-1))
allocate(SF(di,di))
allocate(RT(di,di))
allocate(RRT(di,di))
allocate(RW(di))
allocate(RWORK(RLWORK))

do i = 0, num - 1
    call integer_binary(i, state(i+1), s)  ! Adjusted for 1-based
   indexing
    !print *, state(i)
end do

do i = 1, num
    decimal(i) = btod(state(i))
    !print*, ' Decimal equivalent of ', state(i) ,' = ', decimal(i)
end do

G = 0.0d0
do k =1 , s
    E = 0.0d0
    do i = 1, num
        do j = 1, num
            ! Computing Sigma_i^x * Sigma_i+1^x for each site
            flagx = 0
            if (decimal(i) .ne. decimal(j)) then
                call sd(state(i), digits1)
                call sd(state(j), digits2)
                if (k < s) then
                    if (digits1(k) .ne. digits2(k) .and. digits1(k
   +1) .ne. digits2(k+1)) then
                        do l = 1, k - 1
                            if (digits1(l) .ne. digits2(l)) then
                                flagx  = flagx + 1
```

```
63                              end if
64                          end do
65                          do l = k + 2, s
66                              if (digits1(l) .ne. digits2(l)) then
67                                  flagx  = flagx + 1
68                              end if
69                          end do
70                          if (flagx == 0) then
71                              E(i, j) = E(i, j) + 1.0
72                          else
73                              E(i, j) = E(i, j) + 0.0
74                          end if
75                      else
76                          E(i, j) = E(i, j) + 0.0
77                      end if
78                  else
79                      if (digits1(1) .ne. digits2(1) .and. digits1(s)
    .ne. digits2(s)) then
80                          do l = 2, s - 1
81                              if (digits1(l) .ne. digits2(l)) then
82                                  flagx  = flagx + 1
83                              end if
84                          end do
85                          if (flagx == 0) then
86                              E(i, j) = E(i, j) + 1.0
87                          else
88                              E(i, j) = E(i, j) + 0.0
89                          end if
90                      else
91                          E(i, j) = E(i, j) + 0.0
92                      end if
93                  end if
94              else
95                  E(i, j) = E(i, j) + 0.0
96              end if
97
98              ! Computing Sigma_i^y * Sigma_i+1^y for each site
99              flagy = 0
```

```
100             if (decimal(i) .ne. decimal(j)) then
101                 call sd(state(i), digits1)
102                 call sd(state(j), digits2)
103                 if (k < s) then
104                     if (digits1(k) .ne. digits2(k) .and. digits1(k
    +1) .ne. digits2(k+1)) then
105                         do l = 1, k - 1
106                             if (digits1(l) .ne. digits2(l)) then
107                                 flagy  = flagy + 1
108                             end if
109                         end do
110                         do l = k + 2, s
111                             if (digits1(l) .ne. digits2(l)) then
112                                 flagy  = flagy + 1
113                             end if
114                         end do
115                         if (flagy == 0) then
116                             E(i, j) = E(i, j) + (-1.0 * (-1.0)**(
    digits2(k) + digits2(k+1)))
117                         else
118                             E(i, j) = E(i, j) + 0.0
119                         end if
120                     else
121                         E(i, j) = E(i, j) + 0.0
122                     end if
123                 else
124                     if (digits1(1) .ne. digits2(1) .and. digits1(s)
    .ne. digits2(s)) then
125                         do l = 2, s - 1
126                             if (digits1(l) .ne. digits2(l)) then
127                                 flagy  = flagy + 1
128                             end if
129                         end do
130                         if (flagy == 0) then
131                             E(i, j) = E(i, j)  + (-1.0 * (-1.0)**(
    digits2(s) + digits2(1)))
132                         else
133                             E(i, j) = E(i, j) + 0.0
```

```
                        end if
                    else
                        E(i, j) = E(i, j) + 0.0
                    end if
                end if
            else
                E(i, j) = E(i, j) + 0.0
            end if

            ! Computing Sigma_i^z * Sigma_i+1^z for each site
            call sd(state(i), digits1)
            if (decimal(i) .ne. decimal(j)) then
                E(i, j) =  E(i,j) + 0.0
            else
                if (k < s) then
                    E(i, j) = E(i, j) + (1 - 2 * digits1(k)) * (1 -
   2 * digits1(k+1))
                else
                    E(i, j) = E(i, j) + (1 - 2 * digits1(1)) * (1 -
   2 * digits1(s))
                end if
            end if


            ! Computing Sigma_i^z for each site
            call sd(state(i), digits1)
            if (decimal(i) .ne. decimal(j)) then
                G(i, j) =  G(i,j) + 0.0
            else
                G(i, j) = G(i, j) + (1 - 2 * digits1(k))
            end if
        end do
    end do
C = C + (1.0d0/4.0d0)*E
F = F + matmul((1.0d0/4.0d0)*E,(1.0d0/4.0d0)*E)
end do
! write(58,*) C

```

```
do k = 0, s-2
    do l = k+1, s-1
        site = (/k, l/)
        ! print*, site
        ! do u = 0, 200
        !   Jo = 0.01 * u
            do t = 0, 200
!Loop for varying B
                Jo =  0.01 * t
                B = 1.0d0
                H = 0.0d0
                do i = 1, num
                    do j = 1, num
                        H(i-1,j-1) = Jo * (C(i,j) + (1.0d0/3.0d0)* F
(i,j)) + B * (1.0d0/2.0d0)* G(i,j)
                        !if (i .le. j) then
                            ! write(57,*) i, j, H(i-1,j-1)
                        !end if
                    end do
                end do

                ! Define parameters for DSYEVX
                JOBZ = 'V'  ! Compute eigenvalues and eigenvectors
                RANGE = 'I' ! Compute selective eigenvalues and
eigenvectors
                UPLO = 'U'  ! Upper triangular part of A is stored
                ABSTOL = 0.0d0
                IL = 1
                IU = 1
                M = IU - IL + 1
                LDZ = N

                ! Call DSYEVX to compute ground state eigenvalue and
 eigen vector
                call DSYEVX (JOBZ, RANGE, UPLO, N, H, LDA, VL, VU,
IL, IU, ABSTOL &
                , M, W, Z, LDZ, WORK, LWORK, IWORK, IFAIL, INFO)

```

```
203                 ! Check for successful execution
204                 ! sum = 0.0d0
205                 ! if (INFO == 0) then
206                 !     do i = 1, M
207                 !         write(3,*) B, W(i)
208                 !     end do
209                 ! else
210                 !     print*, 'Error in DSYEVX, info =', INFO
211                 ! end if
212
213                 do i = 0, M-1
214                     do j = 0, num - 1
215                         if (abs(Z(j,i)) .le. 0.000000000001 .and.
    abs(Z(j,i)) .ge. 0) then
216                             psi(j,i) = 0
217                         else
218                             psi(j,i) = Z(j,i)
219                         end if
220                         ! write(7,*) j, i, psi(j,i)
221                     end do
222                 end do
223
224                 rho = matmul(psi, transpose(psi))
225                 ! do i = 0, num-1
226                 !     do j = 0, num-1
227                 !         if(i .eq. j) then
228                 !             trace = trace + rho(i,j)
229                 !         end if
230                 !         write(7,*) i, j, rho(i, j)
231                 !     end do
232                 ! end do
233                 ! print*, "Trace = ", trace
234
235                 call ptrace(rho, d, s, s1, site, rdm)
236                 ! trace = 0.0d0
237                 ! do i = 0, di-1
238                 !     do j = 0, di-1
239                 !         ! rdm(i+1,j+1) = red_rho(i,j)
```

```
240                 !          if (i .eq. j) then
241                 !              trace = trace + rdm(i+1,j+1)
242                 !          end if
243                 !          ! write(7,*) i, j, rdm(i+1, j+1)
244                 !      end do
245                 ! end do
246                 ! print*, "Trace = ", trace
247
248                 call spin_flip_matrix(d,SF)                          !
    Generating Spin Flip Matrix
249                 RT = matmul(SF, matmul(rdm, SF))
250                 RRT = matmul(rdm, RT)
251
252                 ! Define parameters for DSYEVX
253                 JOBZ = 'N'  ! Compute eigenvalues only
254                 ! Call DSYEV to compute eigenvalues and eigenvectors
255                 call DSYEV(JOBZ, UPLO, di, RRT, RLDA, RW, RWORK,
    RLWORK, RINFO)
256                 sum = 0.0d0
257                 ! Check for successful execution
258                 if (RINFO == 0) then
259                     call sort_dec(RW, di)
260                     do i = 2, di
261                         sum = sum + sqrt(abs(RW(i)))
262                     end do
263                 else
264                     print*, 'Error in DSYEV, info =', RINFO
265                 end if
266                 con = max(0.0d0, (sqrt(abs(RW(1)))- sum))
267                 pos = k*10 + l
268                 write(400+pos,*) Jo, con
269             ! end do
270         end do
271     end do
272 end do
273
274 contains
275
```

```
276 subroutine integer_binary(num, binary_string, n)
277     implicit none
278     integer :: num
279     integer, intent(in) :: n
280     integer :: i, temp_num
281     character(len=n), intent(out) :: binary_string
282     temp_num = num
283     ! Initialize binary_string to all zeros
284     binary_string = repeat('0', n)
285     ! Convert the integer to binary
286     do i = 0, n - 1
287         if (mod(temp_num, 2) == 1) then
288             binary_string(n-i:n-i) = '1'  ! Set the bit to '1'
289         else
290             binary_string(n-i:n-i) = '0'  ! Set the bit to '0'
291         end if
292         temp_num = temp_num / 2  ! Divide num by 2 to shift right
293     end do
294 end subroutine integer_binary
295
296 subroutine sd(binaryString, digits)
297     character(len=*), intent(in) :: binaryString
298     integer, allocatable, intent(out) :: digits(:)
299     integer :: i, len
300     len = len_trim(binaryString)  ! Get the length of the binary
    string
301     allocate(digits(len))          ! Allocate array to hold digits
302     ! Convert each character to an integer
303     do i = 1, len
304         if (binaryString(i:i) == '1') then
305             digits(i) = 1
306         else if (binaryString(i:i) == '0') then
307             digits(i) = 0
308         else
309             print *, "Invalid character in binary string."
310             digits = 0  ! Set to zero if invalid character is found
311             return
312         end if
```

```
313     end do
314 end subroutine sd
315 
316 function btod(binaryString) result(decimalValue)
317     implicit none
318     character(len=*), intent(in) :: binaryString
319     integer :: decimalValue
320     integer :: i, length
321 
322     decimalValue = 0
323     length = len_trim(binaryString)  ! Get the length of the binary
    string
324 
325     ! Convert binary string to decimal
326     do i = 1, length
327         if (binaryString(i:i) == '1') then
328             decimalValue = decimalValue + 2**(length - i)
329         else if (binaryString(i:i) /= '0') then
330             print *, "Invalid character in binary string."
331             decimalValue = -1  ! Indicate an error with -1
332             return
333         end if
334     end do
335 end function btod
336 
337 subroutine ptrace(rho, d, s, s1, site, rdm)
338         implicit none
339         integer, intent(in) :: d, s, s1
340         integer, intent(in) :: site(0:s1-1)
341         real*8, intent(in) :: rho(0:d**s-1, 0:d**s-1)
342         real*8, intent(out) :: rdm(d**s1, d**s1)
343         integer, allocatable :: idits(:), jdits(:), ridits(:),
    rjdits(:), to(:)
344         integer :: i, j, k, p, flag, ri, rj
345         real*8 :: a
346 
347         allocate(idits(0:s-1), jdits(0:s-1))
348         allocate(ridits(0:s1-1), rjdits(0:s1-1))
```

```
349         allocate(to(0:s-s1-1))
350
351         ! Compute the indices to trace out
352         call TOI(s, site, s1, to)
353
354         rdm = 0.0d0
355         do i = 0, d**s - 1
356             do j = 0, d**s - 1
357                 if (rho(i,j) .ne. 0.0d0) then
358                     a = rho(i,j)
359                     call DECTOD(i, d, s, idits)
360                     call DECTOD(j, d, s, jdits)
361                     flag = 0
362                     do k = 0, s - s1 - 1
363                         if (idits(to(k)) .ne. jdits(to(k))) then
364                             flag = 1
365                             exit
366                         end if
367                     end do
368                     if (flag .eq. 0) then
369                         do k = 0, s1 - 1
370                             p = site(k)
371                             ridits(k) = idits(p)
372                             rjdits(k) = jdits(p)
373                         end do
374                         call DTODEC(d, s1, ridits, ri)
375                         call DTODEC(d, s1, rjdits, rj)
376                         rdm(ri+1, rj+1) = rdm(ri+1, rj+1) + a
377                     end if
378                 end if
379             end do
380         end do
381
382         deallocate(idits, jdits, ridits, rjdits, to)
383     end subroutine ptrace
384
385     subroutine DECTOD(dec, d, s, dits)
386         implicit none
```

```
387         integer, intent(in) :: dec, d, s
388         integer, allocatable, intent(out) :: dits(:)
389         integer :: i, temp_dec
390 
391         allocate(dits(0:s-1))
392 
393         temp_dec = dec
394         do i = s-1, 0, -1
395             dits(i) = mod(temp_dec, d)
396             temp_dec = temp_dec / d
397         end do
398     end subroutine DECTOD
399 
400     subroutine DTODEC(d, s, dits, dec)
401         implicit none
402         integer, intent(in) :: d, s
403         integer, dimension(0:s-1), intent(in) :: dits
404         integer, intent(out) :: dec
405         integer :: i
406 
407         dec = 0
408         do i = 0, s - 1
409             dec = dec * d + dits(i)
410         end do
411     end subroutine DTODEC
412 
413     subroutine TOI(s, site, s1, to)
414         implicit none
415         integer, intent(in) :: s, s1
416         integer, intent(in) :: site(0:s1-1)
417         integer, intent(out) :: to(0:s-s1-1)
418         integer :: i, j, k, found
419 
420         k = 0
421         do i = 0, s-1
422             found = 0
423             do j = 0, s1 - 1
424                 if (i == site(j)) then
```

```
                    found = 1
                    exit
                end if
            end do
            if (found == 0) then
                to(k) = i
                k = k + 1
            end if
        end do
    end subroutine TOI

    subroutine spin_flip_matrix(n, SFM)
        implicit none
        integer, intent(in) :: n        ! The size of the level system (dimension of the system)
        real*8, intent(out) :: SFM(n**2, n**2)  ! Output matrix

        integer :: size, i, j

        ! Calculate the size of the matrix (n^2)
        size = n**2

        ! Initialize the matrix with zeros
        SFM = 0.0d0

        ! Fill the matrix based on the alternating pattern
        do i = 1, size
            if (mod(i - 1, n + 1) == 0) then
                SFM(i, size - i + 1) = -1  ! Place -1 every (n+1)-th index (i=0, n+1, 2*(n+1), ...)
            else
                SFM(i, size - i + 1) = 1   ! Place 1 for all other indices
            end if
        end do
    end subroutine spin_flip_matrix

    subroutine sort_dec(arr, n)
```

```
460         implicit none
461         integer, intent(in) :: n
462         real*8, intent(inout) :: arr(n)
463         integer :: i, j
464         real*8 :: temp
465 
466         ! Insertion sort (Descending Order)
467         do i = 2, n
468             temp = arr(i)
469             j = i - 1
470             do while (j > 0 .and. arr(j) < temp)
471                 arr(j + 1) = arr(j)
472                 j = j - 1
473             end do
474             arr(j + 1) = temp
475         end do
476     end subroutine sort_dec
477 end program
478 
```

Using this approach, we can build the Hamiltonian and eigenvalues and eigenvectors up to 10 qubits (i.e. matrix dimension $2^{10} \times 2^{10}$), which we were able to compute up to 8 qubits only earlier using the bit operation approach and DSYEV subroutine. Since we are focusing only on computing specific eigenvalues and eigenvectors now, we can reduce the computation time by almost 3-4 times compared to DSYEV. Still, we are not able to compute the Hamiltonian and eigenvalues and eigenvectors for 12 qubits, which involves 2^{12} basis states, so we have to look for further smart approaches.

3.3 Sparsification

As we have explained earlier, the urge for further approaches to accelerate computational power, Sparsification [15] is another method that helps us to achieve more efficiency. So, before applying the method, let us understand the preliminaries and the method first.

3.3.1 Sparse Matrix and Sparsity

Dense Matrix: A matrix in which the majority of elements are non-zero.

Sparse Matrix: A matrix in which many of the elements are zero.

Sparsity: Sparsity is defined as the percentage of zero elements present in the matrix. Let us understand sparsity using an example. Let us consider two matrices A and B having dimension 3×3,

$$A = \begin{bmatrix} 5 & 3 & 0 \\ -2 & 0 & 0 \\ 1 & 0 & i \end{bmatrix}, B = \begin{bmatrix} 0 & -1 & 3 \\ 2i & 5 & 7i \\ 0 & 2 & -3 \end{bmatrix}. \tag{3.1}$$

One can see that the number of non-zero elements in B is more than in A, so B is denser and less sparse than A. One can also calculate the sparsity of A as 44.4% and B as 22.2%, where sparsity is calculated as,

$$\text{Sparsity} = \frac{\text{Number of zero elements}}{\text{Number of total elements}} \times 100\%. \tag{3.2}$$

3.3.2 Motivation for Sparsification

Since most of the real-world systems are sparse, so is our Hamiltonian; we do not want to deal with the zero elements in the matrix. Since we do not have to store zero elements, we do not need to allocate space for zeros, thus it saves a huge amount of memory. Also, we do not need to perform operations with zeros thus it saves, thus it saves computation time. When we do not want to deal with zero elements means we do not need to load these zeros, thus it saves memory bandwidth very much. We will discuss the sparsity of our Hamiltonian further in this chapter. First, let us understand the sparsification format that we are using. There are several sparse matrix storage formats, such as:

- Coordinate Format (COO)
- Compressed Sparse Row (CSR)

- ELLPACK Format (ELL)
- Jagged Diagonal Storage (JDS)

and so on...

3.3.3 Coordinate Format (COO)

One can ask the basic question why are we using COO format? [16] The reasons behind this are space efficiency, ease of adding/ reordering elements, ease of finding desired data, and minimising central divergence.
In this format, we store every non-zero element along with its row index and column index.
Let us understand using an example, consider a 4×4 matrix,

$$M = \begin{bmatrix} 1 & 0 & 7 & 0 \\ 9 & 3 & 0 & 5 \\ 0 & 2 & 0 & 0 \\ 6 & 0 & 1 & 0 \end{bmatrix}, \tag{3.3}$$

Now mark the non-zero elements and store them in a 1-D array named value,

$$\text{value} = \begin{bmatrix} 1 & 7 & 9 & 3 & 5 & 2 & 6 & 1 \end{bmatrix},$$

next, store the corresponding index of the row in another 1-D array named row index,

$$\text{row index} = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 2 & 3 & 3 \end{bmatrix},$$

then, store the corresponding index of the column in another 1-D array named column index,

$$\text{column index} = \begin{bmatrix} 0 & 2 & 0 & 1 & 3 & 1 & 0 & 2 \end{bmatrix}.$$

Now, this sparse matrix can be used to compute the eigenvalues and eigenvectors of our matrix. A simple pictorial representation of this method is shown below,

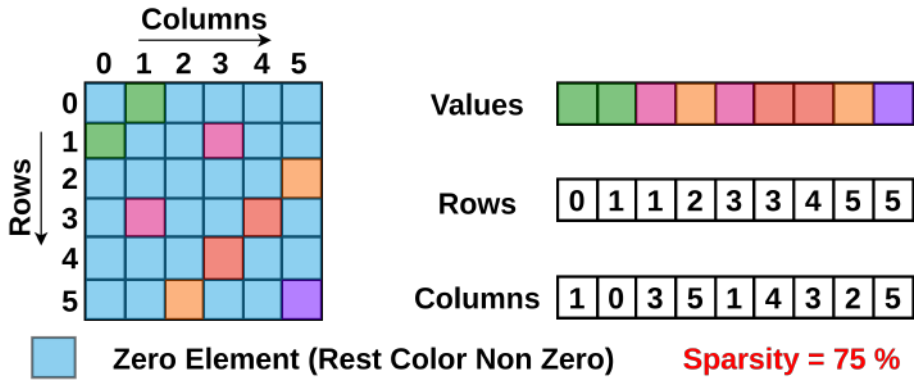


Figure 3.1: Coordinate format method (COO)

Let us see a simple Fortran program to do sparsification for a matrix in Eq. [3.3].

```
program sparse
    implicit none
    integer, parameter :: s = 4     ! Dimension of the matrix
    real*8, allocatable :: D(:,:), value_nz(:)
    integer, allocatable :: row_i(:), column_i(:)
    integer :: nnz, i, j
    real*8 :: sparsity

    ! Allocating arrays
    allocate(real(8) :: D(0:s-1, 0:s-1))

    ! Initialize matrix
    D = 0.0d0
    D(0,0) = 1.0
    D(0,2) = 7.0
    D(1,0) = 9.0
    D(1,1) = 3.0
    D(1,3) = 5.0
    D(2,1) = 2.0
    D(3,0) = 6.0
    D(3,2) = 1.0

    ! Initialize nnz = 0
    nnz = 0
```

```
25      do i = 0, s-1
26          do j = 0, s-1
27              if(D(i,j) .ne. 0) then
28                  nnz = nnz + 1
29              end if
30          end do
31      end do
32      print*, nnz
33
34      ! Allocating
35      allocate(real(8) :: value_nz(0:nnz-1))
36      allocate(integer :: row_i(0:nnz-1))
37      allocate(integer :: column_i(0:nnz-1))
38      nnz = 0
39
40      do i = 0, s-1
41          do j = 0, s-1
42              if(D(i,j) .ne. 0) then
43                  row_i(nnz) = i
44                  column_i(nnz) = j
45                  value_nz(nnz) = D(i,j)
46                  nnz = nnz + 1
47              end if
48          end do
49      end do
50
51      do i = 0, nnz-1
52          write(11,*) row_i(i), column_i(i), value_nz(i)
53      end do
54
55      sparcity = 100 - (nnz*100/s**2)
56      print*, "Sparsity of the given matrix is: ", sparsity, "%."
57  end program sparse
58
```

This program will store the non-zero elements, their row index, and column index in a different 1-D array, thus giving us all the valuable information in a highly compact

form. Also, we can calculate the sparsity of the matrix in this program. Now let us implement this for our Hamiltonian Matrix in Eq. [2.42],

```
program spin1sparse
    implicit none
    integer, parameter :: d =2, s = 12, s1 = 2
    integer, parameter :: num = d**s, N = num, di = d**s1
    integer :: i, j, k, l, t, u, flagx, flagy, nnz
    integer, allocatable :: digits1(:), digits2(:), decimal(:)
    character(len=:), allocatable :: state(:)
    double precision, allocatable :: E(:,:), C(:,:), F(:,:), G(:,:),
    H(:,:), Z(:,:)
    character*1 :: JOBZ, UPLO, RANGE
    integer ::  LDA = num, LWORK = 8*num, INFO, IL, IU, M, LDZ,
   IWORK(5*N), IFAIL(N)
    double precision :: W(num), WORK(8*num), VL, VU, ABSTOL
    real*8 :: sum, Jo, B
    integer, allocatable :: row_i(:), column_i(:)
    real*8 :: sparcity, nz
    real*8, allocatable :: value_nz(:)

    ! Allocate arrays after determining the value of num
    allocate(character(len=s) :: state(num))
    allocate(integer :: decimal(num))
    allocate(integer :: digits1(s), digits2(s))
    allocate(real(8) :: C(num, num))
    allocate(real(8) :: E(num, num))
    allocate(real(8) :: F(num, num))
    allocate(real(8) :: G(num, num))
    allocate(real(8) :: H(0:num-1 , 0:num-1))

    do i = 0, num - 1
        call integer_binary(i, state(i+1), s)  ! Adjusted for 1-
   based indexing
        !print *, state(i)
    end do

```

```
32      do i = 1, num
33          decimal(i) = btod(state(i))
34          !print*, ' Decimal equivalent of ', state(i) ,' = ', decimal
    (i)
35      end do
36
37      G = 0.0d0
38      do k =1 , s
39
40          E = 0.0d0
41          do i = 1, num
42              do j = 1, num
43                  ! Computing Sigma_i^x * Sigma_i+1^x for each site
44                  flagx = 0
45                  if (decimal(i) .ne. decimal(j)) then
46                      call sd(state(i), digits1)
47                      call sd(state(j), digits2)
48                      if (k < s) then
49                          if (digits1(k) .ne. digits2(k) .and. digits1
    (k+1) .ne. digits2(k+1)) then
50                              do l = 1, k - 1
51                                  if (digits1(l) .ne. digits2(l)) then
52                                      flagx  = flagx + 1
53                                  end if
54                              end do
55                              do l = k + 2, s
56                                  if (digits1(l) .ne. digits2(l)) then
57                                      flagx  = flagx + 1
58                                  end if
59                              end do
60                              if (flagx == 0) then
61                                  E(i, j) = E(i, j) + 1.0
62                              else
63                                  E(i, j) = E(i, j) + 0.0
64                              end if
65                          else
66                              E(i, j) = E(i, j) + 0.0
67                          end if
```

```
68                      else
69                          if (digits1(1) .ne. digits2(1) .and. digits1
    (s) .ne. digits2(s)) then
70                              do l = 2, s - 1
71                                  if (digits1(l) .ne. digits2(l)) then
72                                      flagx  = flagx + 1
73                                  end if
74                              end do
75                              if (flagx == 0) then
76                                  E(i, j) = E(i, j) + 1.0
77                              else
78                                  E(i, j) = E(i, j) + 0.0
79                              end if
80                          else
81                              E(i, j) = E(i, j) + 0.0
82                          end if
83                      end if
84                  else
85                      E(i, j) = E(i, j) + 0.0
86                  end if
87
88                  ! Computing Sigma_i^y * Sigma_i+1^y for each site
89                  flagy = 0
90                  if (decimal(i) .ne. decimal(j)) then
91                      call sd(state(i), digits1)
92                      call sd(state(j), digits2)
93                      if (k < s) then
94                          if (digits1(k) .ne. digits2(k) .and. digits1
    (k+1) .ne. digits2(k+1)) then
95                              do l = 1, k - 1
96                                  if (digits1(l) .ne. digits2(l)) then
97                                      flagy  = flagy + 1
98                                  end if
99                              end do
100                             do l = k + 2, s
101                                 if (digits1(l) .ne. digits2(l)) then
102                                     flagy  = flagy + 1
103                                 end if
```

```
104                             end do
105                             if (flagy == 0) then
106                                 E(i, j) = E(i, j) + (-1.0 * (-1.0)
    **(digits2(k) + digits2(k+1)))
107                             else
108                                 E(i, j) = E(i, j) + 0.0
109                             end if
110                         else
111                             E(i, j) = E(i, j) + 0.0
112                         end if
113                     else
114                         if (digits1(1) .ne. digits2(1) .and. digits1
    (s) .ne. digits2(s)) then
115                             do l = 2, s - 1
116                                 if (digits1(l) .ne. digits2(l)) then
117                                     flagy  = flagy + 1
118                                 end if
119                             end do
120                             if (flagy == 0) then
121                                 E(i, j) = E(i, j)  + (-1.0 * (-1.0)
    **(digits2(s) + digits2(1)))
122                             else
123                                 E(i, j) = E(i, j) + 0.0
124                             end if
125                         else
126                             E(i, j) = E(i, j) + 0.0
127                         end if
128                     end if
129                 else
130                     E(i, j) = E(i, j) + 0.0
131                 end if
132
133                 ! Computing Sigma_i^z * Sigma_i+1^z for each site
134                 call sd(state(i), digits1)
135                 if (decimal(i) .ne. decimal(j)) then
136                     E(i, j) =  E(i,j) + 0.0
137                 else
138                     if (k < s) then
```

```
                        E(i, j) = E(i, j) + (1 - 2 * digits1(k)) *
    (1 - 2 * digits1(k+1))
                    else
                        E(i, j) = E(i, j) + (1 - 2 * digits1(1)) *
    (1 - 2 * digits1(s))
                    end if
                end if

                ! Computing Sigma_i^z for each site
                call sd(state(i), digits1)
                if (decimal(i) .ne. decimal(j)) then
                    G(i, j) =  G(i,j) + 0.0
                else
                    G(i, j) = G(i, j) + (1 - 2 * digits1(k))
                end if
            end do
        end do
    C = C + (1.0d0/4.0d0)*E
    F = F + matmul((1.0d0/4.0d0)*E,(1.0d0/4.0d0)*E)
    end do
    ! write(58,*) C

    B = 1.0d0
    Jo = 1.0d0
    H = 0.0d0
    do i = 1, num
        do j = 1, num
            H(i-1,j-1) = Jo * (C(i,j) + (1.0d0/3.0d0)* F(i,j)) + B *
    (1.0d0/2.0d0)* G(i,j)
            if(H(i-1,j-1) .ne. 0) then
                nnz = nnz + 1
            end if
        end do
    end do

    !Deallocating Matrices
    deallocate(state)
    deallocate(decimal)
```

```
174     deallocate(digits1)
175     deallocate(digits2)
176     deallocate(C)
177     deallocate(G)
178
179     !Allocating
180     allocate(real(8) :: value_nz(0:nnz-1))
181     allocate(integer :: row_i(0:nnz-1))
182     allocate(integer :: column_i(0:nnz-1))
183
184     nnz = 0
185     do i = 0, num-1
186         do j = 0, num-1
187             if(H(i,j) .ne. 0) then
188                 row_i(nnz) = i
189                 column_i(nnz) = j
190                 value_nz(nnz) = H(i,j)
191                 nnz = nnz + 1
192             end if
193         end do
194     end do
195     !print*, nnz
196
197     do i = 0, nnz-1
198         write(3,*) row_i(i), column_i(i), value_nz(i)
199     end do
200
201     nz = num**2 - nnz
202     sparcity = (nz*100/num**2)
203     print*, "Sparcity of the given matrix is: ", sparcity, "%."
204
205     ! Define parameters for DSYEVX
206     JOBZ = 'V'  ! Compute eigenvalues and eigenvectors
207     RANGE = 'I' ! Compute selective eigenvalues and eigenvectors
208     UPLO = 'U'  ! Upper triangular part of A is stored
209     ABSTOL = 0.0d0
210     IL = 1
211     IU = 1
```

```
212     M = IU - IL + 1
213     LDZ = N
214
215     ! Call DSYEVX to compute selective eigenvalues and eigenvectors
216     call DSYEVX (JOBZ, RANGE, UPLO, N, H, LDA, VL, VU, IL, IU,
        ABSTOL, M, W, Z,LDZ, WORK, LWORK, IWORK, IFAIL, INFO)
217
218     ! Check for successful execution
219     sum = 0.0d0
220     if (INFO == 0) then
221         write(3,*) 'Eigenvalues are:'
222         do l = 1, M
223             write(3,*) W(l)
224             ! sum = sum + W(l)
225         end do
226     else
227         print*, 'Error in DSYEV, info =', INFO
228     end if
229     !print*, sum
230
231     write(7,*) 'Eigen Vectors are :'
232     do k = 1, M
233         write(7,*) 'Eigen Vector ', k, ':'
234         do l = 1, num
235             write(7,*) k, l, Z(l-1,k-1)
236         end do
237     end do
238
239 contains
240
241     subroutine integer_binary(num, binary_string, n)
242         implicit none
243         integer :: num
244         integer, intent(in) :: n
245         integer :: i, temp_num
246         character(len=n), intent(out) :: binary_string
247         temp_num = num
248         ! Initialize binary_string to all zeros
```

```
249         binary_string = repeat('0', n)
250         ! Convert the integer to binary
251         do i = 0, n - 1
252             if (mod(temp_num, 2) == 1) then
253                 binary_string(n-i:n-i) = '1'  ! Set the bit to '1'
254             else
255                 binary_string(n-i:n-i) = '0'  ! Set the bit to '0'
256             end if
257             temp_num = temp_num / 2  ! Divide num by 2 to shift
    right
258         end do
259     end subroutine integer_binary
260
261     subroutine sd(binaryString, digits)
262         character(len=*), intent(in) :: binaryString
263         integer, allocatable, intent(out) :: digits(:)
264         integer :: i, len
265         len = len_trim(binaryString)  ! Get the length of the binary
     string
266         allocate(digits(len))          ! Allocate array to hold
    digits
267         ! Convert each character to an integer
268         do i = 1, len
269             if (binaryString(i:i) == '1') then
270                 digits(i) = 1
271             else if (binaryString(i:i) == '0') then
272                 digits(i) = 0
273             else
274                 print *, "Invalid character in binary string."
275                 digits = 0  ! Set to zero if invalid character is
    found
276                 return
277             end if
278         end do
279     end subroutine sd
280
281     function btod(binaryString) result(decimalValue)
282         implicit none
```

```
283         character(len=*), intent(in) :: binaryString
284         integer :: decimalValue
285         integer :: i, length
286         decimalValue = 0
287         length = len_trim(binaryString)  ! Get the length of the
    binary string
288         ! Convert binary string to decimal
289         do i = 1, length
290             if (binaryString(i:i) == '1') then
291                 decimalValue = decimalValue + 2**(length - i)
292             else if (binaryString(i:i) /= '0') then
293                 print *, "Invalid character in binary string."
294                 decimalValue = -1  ! Indicate an error with -1
295                 return
296             end if
297         end do
298     end function btod
299 end program
300
```

3.3.4 Sparsity of Hamiltonian Matrix

Now let us compare the Sparsity for our Hamiltonian Matrix with the increasing number of qubits,

Number of Qubits (s)	**Dimension of Hilbert Space ($2^s \times 2^s$)**	**Sparsity of Hamiltonian Matrix (%)**
4	16×16	81.25
6	64×64	93.75
8	256×256	98.05
10	1024×1024	99.41
12	4096×4096	99.83

Table 3.1: Sparsity of the Hamiltonian matrix for different numbers of qubits from s = 4 to 12.

One can see that even the Hamiltonian Matrix for the 4-qubit system is very sparse i.e. 81.25%, and as we increase the number of qubits, the sparsity even increases more and more till it reaches to 99.83% for 12 qubits. Let us plot the graph to have a better picture.

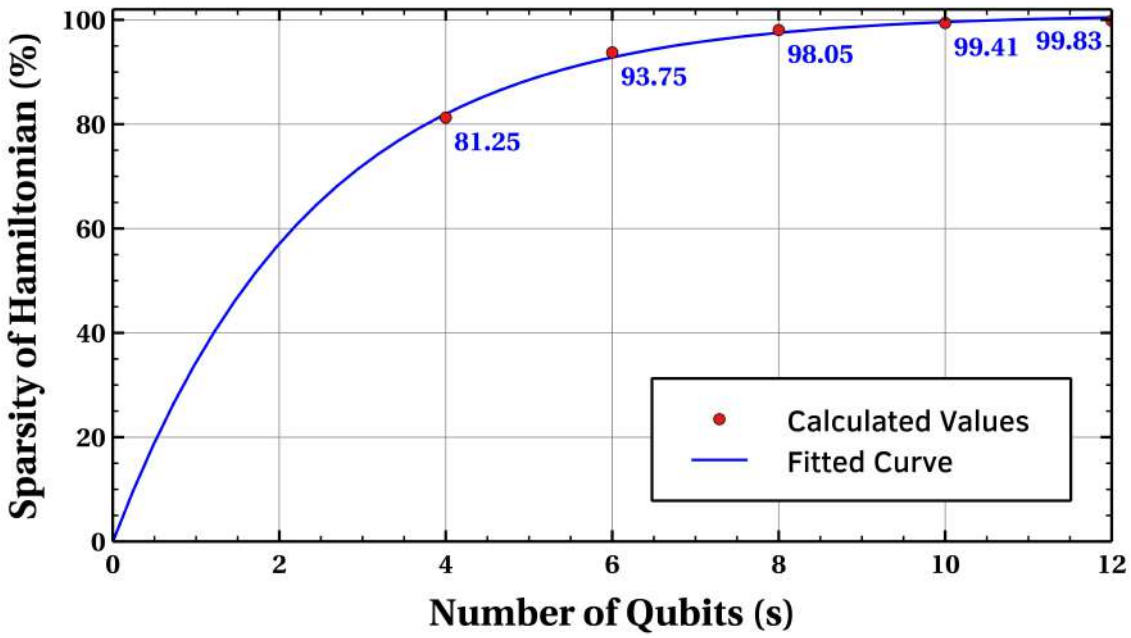


Figure 3.2: Trend of sparsity of Hamiltonian matrix vs a few of the qubits

3.3.5 Computational Time for the AKLT Hamiltonian

Now let us compare the computation time for all the methods for different numbers of qubits we have used to make the computation faster till now, from the bit operation approach to sparsification.

Number of Qubits (s)	Computation Time Using DSYEV	Computation Time Using DSYEVX
4	11**ms**	10**ms**
6	35**ms**	22**ms**
8	462**ms**	386**ms**
10	11**s** 947**ms**	9**s** 237**ms**
12	8**m** 10**s** 790**ms**	5**m** 26**s** 951**ms**

Table 3.2: Computational time of Hamiltonian matrix and eigenvalues and eigenvectors for different numbers of qubits using DSYEV and DSYEVX.

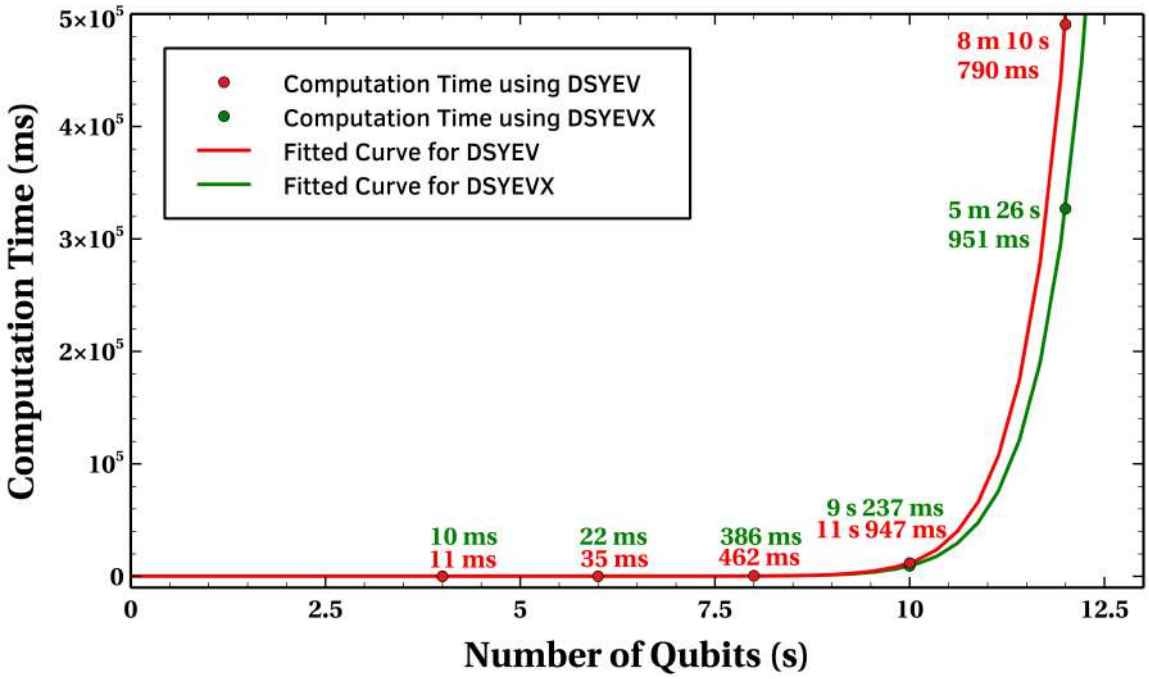


Figure 3.3: Computational time of Hamiltonian matrix and eigenvalues and eigenvectors for different numbers of qubits using DSYEV and DSYEVX.

First, let us analyse building the Hamiltonian using the bit operation approach and the DSYEV subroutine to compute eigenvalues and eigenvectors for different numbers of qubits. Since our Hamiltonian is symmetric, we can compute only the upper triangular matrix and then use that to compute eigenvalues and eigenvectors. So, from the graph Fig. [3.3], you can see that the computational time increases slowly up to 10 qubits, but as we took a 12-qubit system, it increased drastically, approximately 50 times that of 10 qubits, so it gives a clear understanding of why we are required to make the program more compatible. We are only writing the upper triangular matrix, but that does not significantly decrease the computational time.

Now, let us analyse the computational time for the bit operation approach and the DSYEVX subroutine to compute eigenvalues and eigenvectors for different numbers of qubits. The graph Fig. [3.3] shows that the computation time using DSYEVX has been reduced to 2-3 times that of DSYEV.

Now, let us see the computational time for the bit operation approach, the DSYEV subroutine and the sparse Hamiltonian matrix (COO Format) to compute eigenvalues and eigenvectors for different numbers of qubits.

Number of Qubits (s)	Computation Time using DSYEV	Computation Time using COO Format
4	11**ms**	9**ms**
6	35**ms**	17**ms**
8	462**ms**	254**ms**
10	11**s** 947**ms**	5**s** 786**ms**
12	8**m** 10**s** 790**ms**	2**m** 12**s** 621**ms**

Table 3.3: Computational time of Hamiltonian matrix and eigenvalues and eigenvectors for different numbers of qubits using DSYEV and COO format.

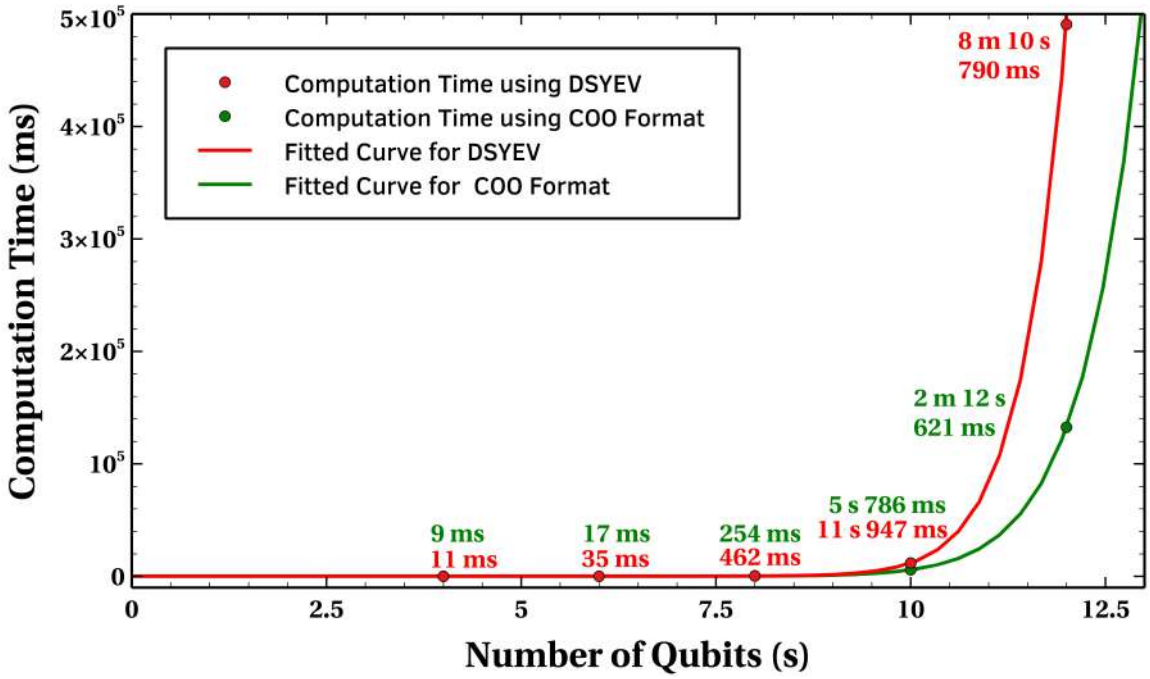


Figure 3.4: Computational time of Hamiltonian matrix and eigenvalues and eigenvectors for different numbers of qubits using DSYEV and COO format.

It can be seen from the graph that the computation time has significantly reduced (almost 5-6 times) for computing the Hamiltonian matrix and ground state eigenvalue, and eigenvector, but we further want to do the entanglement study of the system, for which we want further optimization of the program. To optimize the program further, we used parallel programming. Earlier, we were using only a single core of the CPU for running the program, which takes more computation time; instead of this we can use multiple CPU cores at a time for a single process, which will significantly reduce the time and for more complicated computations we can further use GPU cores which will considerably accelerate the computation. Before implementing

that first let us understand some basics about it.

3.4 Parallel Programming

Let us understand the concept of parallel programming with a simple example. If we have to make a 10 ft. high wall and we hire a single labourer for this, it will take 3 to 4 days for him, because he has to arrange bricks, make cement, and then make the wall by himself. But if we hire 4-5 labourers for this work, few of them can arrange bricks, few of them can make the cement, and the rest can make the wall simultaneously, which will significantly reduce the time to 1 day, meaning delegating the work reduces the total time. It is because for 1 labourer, he has to do tasks serially, firstly he has to complete the first task, then he can proceed further, but for 4-5 labourers, they can divide the tasks among themselves and do the tasks in parallel.

This is the central principle behind parallel programming. One can significantly reduce the computation time by splitting one large task into smaller ones that can be performed by multiple processors at the same time. With parallel programming, a task that will take several days can be reduced to several hours or even lesser.

3.4.1 Serial and Parallel Processes

A serial process is a process entirely run by a single processor. It takes more time because it runs the process line by line as written in the code. It is analogous to making a wall by single labourer.

A parallel process is a process that is run over multiple cores of several processors (both CPU and GPU). Each subprocess has its own memory, and the memories of all the subprocesses are shared. It is analogous to making a wall by 4-5 labourers. For doing this in Fortran, we have several libraries, such as:

- OpenMP
- OpenACC

- MPI (Message Passing Interface)
- NVIDIA HPC SDK

etc.

3.4.2 Message Passing Interface (MPI)

MPI is the standard library for handling parallel processing [17]. We choose this because it gives more flexibility in handling memory for individual processes, and it is also compatible with the multi-node structures. We are using the advanced version of MPI, known as OpenMPI, which is easier to learn in comparison to MPI, but it is based on MPI only.

We are not going to give details on getting started with OpenMPI. One can search for their official website and get the documentation there. So, before implementing parallel programming in our system under study, let us understand some basic commands and functions in the OpenMPI library.

Command	**Function**
`MPI_INIT`	Initializes the MPI execution environment. This must be called before any other MPI function.
`MPI_COMM_RANK`	Determines the rank (ID) of the calling process within the communicator.
`MPI_COMM_SIZE`	Determines the total number of processes in the communicator.
`MPI_FINALIZE`	Terminates the MPI execution environment. This should be the last MPI call.

This table gives the basic function of some commands in the MPI library. Now, let us learn the calling statements of the commands.

Command	Calling Statements
MPI_INIT	call MPI_INIT(ierr)
MPI_COMM_RANK	call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
MPI_COMM_SIZE	call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
MPI_FINALIZE	call MPI_FINALIZE(ierr)

These are the calling statements for the given functions. Now, let us learn how to compile and run the program.

```
!For compiling the program using multiple CPU cores with the name of
!the file as "programname.f90", the command is:
mpif90 -O3 programname.f90 -llapack
```

Let us see the breakup of each phrase in the above command as,

Command	Description
mpif90	Invokes the MPI-enabled Fortran compiler wrapper, which automatically links necessary MPI libraries.
-O3	Enables high-level optimization during compilation, including loop unrolling and inlining for better performance.
programname.f90	Specifies the source file name to be compiled.
-llapack	Links the LAPACK library, which provides routines for solving linear algebra problems.

```
!For running the program using multiple CPU cores, the command is:
time mpirun -np 6 ./a.out
```

Command	Description
time	Measures and displays the time taken by the program to execute, including real, user, and system time.
mpirun	Executes the MPI program using the specified number of processes.
-np 6	Indicates that the program should run using 6 parallel processes.
./a.out	Specifies the compiled executable to run. This is typically the output of the compilation step.

Now let us see a few more commands which can be useful for data transfer from one core to another.

`MPI_SEND(buf, count, type, dest, tag, comm, ierr)`

The `MPI_SEND` command is a standard blocking point-to-point communication routine in MPI, used to send a message from one process to another.

Argument	IN/OUT	Description
`buf`	OUT	Starting address of the send buffer.
`count`	IN	Number of elements in the send buffer.
`type`	IN	`MPI_DATATYPE` of each buffer element.
`dest`	IN	Rank of the destination process.
`tag`	IN	Message tag to identify the message.
`comm`	IN	Communicator that defines the group of processes involved.

If there is a message sending command from one process to another, there must be a receiving command also; let us understand that.

`MPI_RECV(buf, count, type, src, tag, comm, status, ierr)`

The `MPI_RECV` function in MPI (Message Passing Interface) is used for receiving a message from another process.

Argument	IN/OUT	Description
`buf`	OUT	Starting address of the receive buffer.
`count`	IN	Number of elements that can be received into the buffer.
`type`	IN	`MPI_DATATYPE` of each element in the buffer.
`src`	IN	Rank of the source process from which the message is received.
`tag`	IN	Message tag used to identify the message.
`comm`	IN	Communicator defining the group of processes involved.
`status`	OUT	Status object that provides details about the received message.

But the `MPI_SEND` and `MPI_RECV` are useful only when we are sending and receiving

the message from individual cores. If we want to send and receive the message to all cores, then we have different commands as below,

---

```
MPI_BCAST(buf, count, type, root,MPI_COMM_WORLD)
```

The MPI_BCAST function in MPI is used for broadcasting a message from one process to all other processes in a communicator.

```
MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
recvcount, recvtype, source, recvtag, MPI_COMM_WORLD, status)
```

The MPI_SENDRECV function in MPI (Message Passing Interface) is used to send and receive messages between two processes in one combined operation. It allows a process to send a message to one process while simultaneously receiving a message from another process.

Now we have enough basics for implementing parallel programming for building our Hamiltonian and computing eigenvalues and eigenvectors. Let us implement it,

```
1  program aklt_gsev_mpi
2      use mpi
3      implicit none
4
5      ! -------------------- MPI Variables --------------------
6      integer :: ierr, rank, size
7      integer :: tstart, tend, chunksize
8
9      ! -------------------- System Parameters --------------------
10     integer, parameter :: d = 2, s = 10, s1 = 2
11     integer, parameter :: num = d**s, N = num, di = d**s1
12     integer :: i, j, k, l, t, flagx, flagy, pos
13     real*8 :: Jo, B, be
14
15     ! -------------------- Array Declarations --------------------
16     integer, allocatable :: digits1(:), digits2(:), decimal(:), site
```

```
    (:)
17      character(len=:), allocatable :: state(:)
18      double precision, allocatable :: E(:,:), C(:,:), F(:,:), G(:,:),
        H(:,:), Z(:,:)
19      real*8, allocatable :: psi(:,:)
20
21      ! DSYEVX-related
22      character*1 :: JOBZ, UPLO, RANGE
23      integer :: LDA = num, LWORK = 8*num, INFO, IL, IU, M, LDZ
24      integer :: IWORK(5*N), IFAIL(N)
25      double precision :: W(num), WORK(8*num), VL, VU, ABSTOL
26
27      ! -------------------- Initialize MPI --------------------
28      call MPI_Init(ierr)
29      call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
30      call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
31
32      ! Allocate memory
33      allocate(character(len=s) :: state(num))
34      allocate(integer :: decimal(num))
35      allocate(integer :: digits1(s), digits2(s))
36      allocate(real(8) :: C(num,num), E(num,num), F(num,num), G(num,
        num))
37      allocate(real(8) :: H(0:num-1,0:num-1), Z(0:num-1,0:num-1), psi
        (0:num-1, 0:0))
38      allocate(site(0:s1-1))
39
40      ! -------------------- Create state and decimal arrays
        --------------------
41      do i = 0, num - 1
42          call integer_binary(i, state(i+1), s)
43      end do
44      do i = 1, num
45          decimal(i) = btod(state(i))
46      end do
47
48      ! -------------------- Parallel section  --------------------
49      chunksize = num / size
```

```
50      tstart = rank * chunksize + 1
51      tend = (rank + 1) * chunksize
52      if (rank == size - 1) tend = num
53
54      ! -------------------- Build full Hamiltonian terms C, F, G
    --------------------
55      G = 0.0d0
56      C = 0.0d0
57      F = 0.0d0
58
59      do k = 1, s
60          E = 0.0d0
61          do i = tstart, tend
62              do j = tstart, tend
63                  ! Computing Sigma_i^x * Sigma_i+1^x for each site
64                  flagx = 0
65                  if (decimal(i) .ne. decimal(j)) then
66                      call sd(state(i), digits1)
67                      call sd(state(j), digits2)
68                      if (k < s) then
69                          if (digits1(k) .ne. digits2(k) .and. digits1
    (k+1) .ne. digits2(k+1)) then
70                              do l = 1, k - 1
71                                  if (digits1(l) .ne. digits2(l)) then
72                                      flagx  = flagx + 1
73                                  end if
74                              end do
75                              do l = k + 2, s
76                                  if (digits1(l) .ne. digits2(l)) then
77                                      flagx  = flagx + 1
78                                  end if
79                              end do
80                              if (flagx == 0) then
81                                  E(i, j) = E(i, j) + 1.0
82                              end if
83                          end if
84                      else
85                          if (digits1(1) .ne. digits2(1) .and. digits1
```

```
    (s) .ne. digits2(s)) then
                            do l = 2, s - 1
                                if (digits1(l) .ne. digits2(l)) then
                                    flagx  = flagx + 1
                                end if
                            end do
                            if (flagx == 0) then
                                E(i, j) = E(i, j) + 1.0
                            end if
                        end if
                    end if
                end if

                ! Computing Sigma_i^y * Sigma_i+1^y for each site
                flagy = 0
                if (decimal(i) .ne. decimal(j)) then
                    call sd(state(i), digits1)
                    call sd(state(j), digits2)
                    if (k < s) then
                        if (digits1(k) .ne. digits2(k) .and. digits1
    (k+1) .ne. digits2(k+1)) then
                            do l = 1, k - 1
                                if (digits1(l) .ne. digits2(l)) then
                                    flagy  = flagy + 1
                                end if
                            end do
                            do l = k + 2, s
                                if (digits1(l) .ne. digits2(l)) then
                                    flagy  = flagy + 1
                                end if
                            end do
                            if (flagy == 0) then
                                E(i, j) = E(i, j) + (-1.0 * (-1.0)
    **(digits2(k) + digits2(k+1)))
                            end if
                        end if
                    else
                        if (digits1(1) .ne. digits2(1) .and. digits1
```

```
    (s) .ne. digits2(s)) then
121                             do l = 2, s - 1
122                                 if (digits1(l) .ne. digits2(l)) then
123                                     flagy  = flagy + 1
124                                 end if
125                             end do
126                             if (flagy == 0) then
127                                 E(i, j) = E(i, j)  + (-1.0 * (-1.0)
    **(digits2(s) + digits2(1)))
128                             end if
129                         end if
130                     end if
131                 end if
132
133                 ! Computing Sigma_i^z * Sigma_i+1^z for each site
134                 call sd(state(i), digits1)
135                 if (decimal(i) .ne. decimal(j)) then
136                     E(i, j) =  E(i,j) + 0.0
137                 else
138                     if (k < s) then
139                         E(i, j) = E(i, j) + (1 - 2 * digits1(k)) *
    (1 - 2 * digits1(k+1))
140                     else
141                         E(i, j) = E(i, j) + (1 - 2 * digits1(1)) *
    (1 - 2 * digits1(s))
142                     end if
143                 end if
144
145                 ! Computing Sigma_i^z for each site
146                 call sd(state(i), digits1)
147                 if (decimal(i) == decimal(j)) then
148                     G(i, j) = G(i, j) + (1 - 2 * digits1(k))
149                 end if
150             end do
151         end do
152         C = C + (1.0d0/4.0d0)*E
153         F = F + matmul((1.0d0/4.0d0)*E, (1.0d0/4.0d0)*E)
154     end do
```

```
155
156
157     do t = 0, 200
158         B = 0.01d0 * t
159         Jo = 1.0d0
160         H = 0.0d0
161
162         do i = tstart, tend
163             do j = tstart, tend
164                 H(i-1, j-1) = Jo * (C(i,j) + (1.0d0/3.0d0)*F(i,j)) +
    B * (1.0d0/2.0d0)*G(i,j)
165             end do
166         end do
167
168         ! Setup for LAPACK call
169         JOBZ = 'V'; RANGE = 'I'; UPLO = 'U'
170         ABSTOL = 0.0d0; IL = 1; IU = 1
171         M = IU - IL + 1; LDZ = N
172
173         call DSYEVX(JOBZ, RANGE, UPLO, N, H, LDA, VL, VU, IL, IU,
    ABSTOL, M, W, Z, LDZ, WORK, LWORK, IWORK, IFAIL, INFO)
174         write(10+s,*) B, W(1)
175     end do
176
177     call MPI_Finalize(ierr)
178
179 contains
180
181     subroutine integer_binary(num, binary_string, n)
182         implicit none
183         integer :: num
184         integer, intent(in) :: n
185         integer :: i, temp_num
186         character(len=n), intent(out) :: binary_string
187         temp_num = num
188         ! Initialize binary_string to all zeros
189         binary_string = repeat('0', n)
190         ! Convert the integer to binary
```

```
191         do i = 0, n - 1
192             if (mod(temp_num, 2) == 1) then
193                 binary_string(n-i:n-i) = '1'  ! Set the bit to '1'
194             else
195                 binary_string(n-i:n-i) = '0'  ! Set the bit to '0'
196             end if
197             temp_num = temp_num / 2  ! Divide num by 2 to shift
    right
198         end do
199     end subroutine integer_binary
200
201     subroutine sd(binaryString, digits)
202         character(len=*), intent(in) :: binaryString
203         integer, allocatable, intent(out) :: digits(:)
204         integer :: i, len
205         len = len_trim(binaryString)  ! Get the length of the binary
     string
206         allocate(digits(len))           ! Allocate array to hold
    digits
207         ! Convert each character to an integer
208         do i = 1, len
209             if (binaryString(i:i) == '1') then
210                 digits(i) = 1
211             else if (binaryString(i:i) == '0') then
212                 digits(i) = 0
213             else
214                 print *, "Invalid character in binary string."
215                 digits = 0  ! Set to zero if invalid character is
    found
216                 return
217             end if
218         end do
219     end subroutine sd
220
221     function btod(binaryString) result(decimalValue)
222         implicit none
223         character(len=*), intent(in) :: binaryString
224         integer :: decimalValue
```

```
225         integer :: i, length
226
227         decimalValue = 0
228         length = len_trim(binaryString)  ! Get the length of the
    binary string
229
230         ! Convert binary string to decimal
231         do i = 1, length
232             if (binaryString(i:i) == '1') then
233                 decimalValue = decimalValue + 2**(length - i)
234             else if (binaryString(i:i) /= '0') then
235                 print *, "Invalid character in binary string."
236                 decimalValue = -1  ! Indicate an error with -1
237                 return
238             end if
239         end do
240     end function btod
241 end program aklt_gsev_mpi
```

In this program, we have used parallel programming to first build/form the Hamiltonian and then calculate the ground state eigenvalue and eigenvector for varying J or B. But still, we are using only the multiple cores of the CPU [18]. Now, if we want to use a GPU (if available), then we have to modify the program considerably, and we have to use the OpenACC environment in FORTRAN and if we are using Python, we will use Numba, so let us first understand some basics of OpenACC and why we use this and we cannot do GPU programming using OpenMPI?

3.4.3 Open Accelerated Computing (OPENACC)

OpenACC [19] is a directive-based programming model designed to provide a simple yet powerful approach to accelerate programs with minimal programming effort. With OpenACC, a single version of the source code will deliver performance portability across the platforms.

By inserting compiler "hints" or directives into their C11, C++17 or Fortran 2003

code, with the NVIDIA OpenACC compiler, one can offload and run the code on the GPU and CPU. In addition to the NVIDIA OpenACC compilers, the HPC SDK includes GPU-enabled libraries and developer tools to help with the GPU acceleration effort.

Let us see a few commands and their use to apply them to our program,

---

```
!$acc data copyin(C,F,G) copy(state,decimal)
```

This directive defines a data region, allocating memory on the GPU and controlling host-device data movement.

- `copyin(C,F,G)`: Copies matrices `C`, `F`, and `G` from the CPU to the GPU at the beginning of the region.

- `copy(state,decimal)`: Copies `state` and `decimal` from CPU to GPU at the beginning, and copies them back to CPU after the region ends.

This helps minimise data transfer overhead during nested loops.

```
!$acc parallel loop collapse(2) present(H, C, F, G)
```

This is a compute region that offloads the nested loop that constructs the Hamiltonian matrix `H` to the GPU.

- `collapse(2)`: Flattens the nested loops (`do i, do j`) into a single loop for better parallelization.

- `present(...)`: Specifies that the variables already exist on the GPU, avoiding redundant transfers.

```
!$acc parallel loop collapse(2) present(Z, psi)
```

This directive offloads the post-processing of eigenvectors to the GPU.

- Filters out small entries from the matrix `Z` and stores the result in `psi`.

- Again, `collapse(2)` maximizes GPU utilization.

```
!$acc end data
```

Ends the `data` region. This:

- Deallocates GPU memory used by the region.
- Automatically transfers back any variables marked as `copy`.

Now that we have understood the use of basic commands that we have used in our program, let us implement them in our program to build the Hamiltonian and compute eigenvalues and eigenvectors of the AKLT Model.

```
program akltmpigpu
    use mpi
    implicit none

    integer, parameter :: d = 2, s = 4, s1 = 2
    integer, parameter :: num = d**s, N = num, di = d**s1
    integer :: i, j, k, l, t, flagx, flagy, pos, u
    integer :: myrank, numprocs, ierr
    integer, allocatable :: digits1(:), digits2(:), decimal(:), site
   (:)
    character(len=:), allocatable :: state(:)
    double precision, allocatable :: E(:,:), C(:,:), F(:,:), G(:,:),
    H(:,:), Z(:,:)
    character*1 :: JOBZ, UPLO, RANGE
    integer ::  LDA = num, LWORK = 8*num, INFO, IL, IU, M, LDZ,
   IWORK(5*N), IFAIL(N)
    double precision :: W(num), WORK(8*num), VL, VU, ABSTOL
    real*8 :: sum, trace, Jo, B, be
    real*8, allocatable :: psi(:,:)
    real*8 :: startTime, endTime

    call MPI_Init(ierr)
    call MPI_Comm_rank(MPI_COMM_WORLD, myrank, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, numprocs, ierr)

    if (myrank < 3) then
        call acc_set_device_num(0, acc_device_nvidia)
    else
        call acc_set_device_num(1, acc_device_nvidia)
```

```
27      end if
28
29      call cpu_time(startTime)
30
31      ! Allocate arrays
32      allocate(character(len=s) :: state(num))
33      allocate(integer :: decimal(num))
34      allocate(integer :: digits1(s), digits2(s))
35      allocate(real(8) :: C(num, num))
36      allocate(real(8) :: E(num, num))
37      allocate(real(8) :: F(num, num))
38      allocate(real(8) :: G(num, num))
39      allocate(real(8) :: H(0:num-1 , 0:num-1))
40      allocate(real(8) :: Z(0:num-1 , 0:num-1))
41      allocate(psi(0:num-1,0:0))
42      allocate(site(0:s1-1))
43
44      C = 0.0d0
45      F = 0.0d0
46      G = 0.0d0
47
48      !$acc data copyin(C,F,G) copy(state,decimal)
49
50      do i = 0, num - 1
51          call integer_binary(i, state(i+1), s)
52      end do
53
54      do i = 1, num
55          decimal(i) = btod(state(i))
56      end do
57
58
59      ! -------------------- Build full Hamiltonian terms C, F, G
    --------------------
60      G = 0.0d0
61      C = 0.0d0
62      F = 0.0d0
63
```

```
64      do k = 1, s
65           E = 0.0d0
66           do i = 1, num
67                do j = 1, num
68                     ! Computing Sigma_i^x * Sigma_i+1^x for each site
69                     flagx = 0
70                     if (decimal(i) .ne. decimal(j)) then
71                          call sd(state(i), digits1)
72                          call sd(state(j), digits2)
73                          if (k < s) then
74                               if (digits1(k) .ne. digits2(k) .and. digits1
    (k+1) .ne. digits2(k+1)) then
75                                    do l = 1, k - 1
76                                         if (digits1(l) .ne. digits2(l)) then
77                                              flagx  = flagx + 1
78                                         end if
79                                    end do
80                                    do l = k + 2, s
81                                         if (digits1(l) .ne. digits2(l)) then
82                                              flagx  = flagx + 1
83                                         end if
84                                    end do
85                                    if (flagx == 0) then
86                                         E(i, j) = E(i, j) + 1.0
87                                    else
88                                         E(i, j) = E(i, j) + 0.0
89                                    end if
90                               else
91                                    E(i, j) = E(i, j) + 0.0
92                               end if
93                          else
94                               if (digits1(1) .ne. digits2(1) .and. digits1
    (s) .ne. digits2(s)) then
95                                    do l = 2, s - 1
96                                         if (digits1(l) .ne. digits2(l)) then
97                                              flagx  = flagx + 1
98                                         end if
99                                    end do
```

```
100                             if (flagx == 0) then
101                                 E(i, j) = E(i, j) + 1.0
102                             else
103                                 E(i, j) = E(i, j) + 0.0
104                             end if
105                         else
106                             E(i, j) = E(i, j) + 0.0
107                         end if
108                     end if
109                 else
110                     E(i, j) = E(i, j) + 0.0
111                 end if
112 
113                 ! Computing Sigma_i^y * Sigma_i+1^y for each site
114                 flagy = 0
115                 if (decimal(i) .ne. decimal(j)) then
116                     call sd(state(i), digits1)
117                     call sd(state(j), digits2)
118                     if (k < s) then
119                         if (digits1(k) .ne. digits2(k) .and. digits1
    (k+1) .ne. digits2(k+1)) then
120                             do l = 1, k - 1
121                                 if (digits1(l) .ne. digits2(l)) then
122                                     flagy  = flagy + 1
123                                 end if
124                             end do
125                             do l = k + 2, s
126                                 if (digits1(l) .ne. digits2(l)) then
127                                     flagy  = flagy + 1
128                                 end if
129                             end do
130                             if (flagy == 0) then
131                                 E(i, j) = E(i, j) + (-1.0 * (-1.0)
    **(digits2(k) + digits2(k+1)))
132                             else
133                                 E(i, j) = E(i, j) + 0.0
134                             end if
135                         else
```

```
136                             E(i, j) = E(i, j) + 0.0
137                         end if
138                     else
139                         if (digits1(1) .ne. digits2(1) .and. digits1
    (s) .ne. digits2(s)) then
140                             do l = 2, s - 1
141                                 if (digits1(l) .ne. digits2(l)) then
142                                     flagy  = flagy + 1
143                                 end if
144                             end do
145                             if (flagy == 0) then
146                                 E(i, j) = E(i, j)  + (-1.0 * (-1.0)
    **(digits2(s) + digits2(1)))
147                             else
148                                 E(i, j) = E(i, j) + 0.0
149                             end if
150                         else
151                             E(i, j) = E(i, j) + 0.0
152                         end if
153                     end if
154                 else
155                     E(i, j) = E(i, j) + 0.0
156                 end if
157
158                 ! Computing Sigma_i^z * Sigma_i+1^z for each site
159                 call sd(state(i), digits1)
160                 if (decimal(i) .ne. decimal(j)) then
161                     E(i, j) =  E(i,j) + 0.0
162                 else
163                     if (k < s) then
164                         E(i, j) = E(i, j) + (1 - 2 * digits1(k)) *
    (1 - 2 * digits1(k+1))
165                     else
166                         E(i, j) = E(i, j) + (1 - 2 * digits1(1)) *
    (1 - 2 * digits1(s))
167                     end if
168                 end if
169
```

```
                ! Computing Sigma_i^z for each site
                call sd(state(i), digits1)
                if (decimal(i) .ne. decimal(j)) then
                    G(i, j) =  G(i,j) + 0.0
                else
                    G(i, j) = G(i, j) + (1 - 2 * digits1(k))
                end if
            end do
        end do
        C = C + (1.0d0/4.0d0)*E
        F = F + matmul((1.0d0/4.0d0)*E, (1.0d0/4.0d0)*E)
    end do

!$acc data copyin(C, F, G) create(H, Z, W, psi)
    do t = 0, 300
        B =  0.01 * t
        Jo = 1.0d0

        !$acc parallel loop collapse(2) present(H, C, F, G)
        do i = 1, num
            do j = 1, num
                H(i-1,j-1) = Jo * (C(i,j) + (1.0d0/3.0d0) * F(i,j))
   + B * (1.0d0/2.0d0) * G(i,j)
            end do
        end do

        ! Diagonalize
        call DSYEVX(JOBZ, RANGE, UPLO, N, H, LDA, VL, VU, IL, IU,
   ABSTOL, M, W, Z, LDZ, WORK, LWORK, IWORK, IFAIL, INFO)

        !$acc parallel loop collapse(2) present(Z, psi)
        do i = 0, M-1
            do j = 0, num - 1
                if (abs(Z(j,i)) <= 1.0d-12) then
                    psi(j,i) = 0.0d0
                else
                    psi(j,i) = Z(j,i)
                end if
```

```
206             end do
207         end do
208     end do
209 !$acc end data
210 
211     call cpu_time(endTime)
212     if (myrank == 0) then
213         print *, "Execution Time (s): ", endTime - startTime
214     end if
215 
216     call MPI_Finalize(ierr)
217 
218 contains
219 
220     subroutine integer_binary(num, binary_string, n)
221         implicit none
222         integer :: num
223         integer, intent(in) :: n
224         integer :: i, temp_num
225         character(len=n), intent(out) :: binary_string
226         temp_num = num
227         ! Initialize binary_string to all zeros
228         binary_string = repeat('0', n)
229         ! Convert the integer to binary
230         do i = 0, n - 1
231             if (mod(temp_num, 2) == 1) then
232                 binary_string(n-i:n-i) = '1'  ! Set the bit to '1'
233             else
234                 binary_string(n-i:n-i) = '0'  ! Set the bit to '0'
235             end if
236             temp_num = temp_num / 2  ! Divide num by 2 to shift
    right
237         end do
238     end subroutine integer_binary
239 
240     subroutine sd(binaryString, digits)
241         character(len=*), intent(in) :: binaryString
242         integer, allocatable, intent(out) :: digits(:)
```

```
243         integer :: i, len
244         len = len_trim(binaryString)  ! Get the length of the binary
     string
245         allocate(digits(len))          ! Allocate array to hold
    digits
246         ! Convert each character to an integer
247         do i = 1, len
248             if (binaryString(i:i) == '1') then
249                 digits(i) = 1
250             else if (binaryString(i:i) == '0') then
251                 digits(i) = 0
252             else
253                 print *, "Invalid character in binary string."
254                 digits = 0  ! Set to zero if invalid character is
    found
255                 return
256             end if
257         end do
258     end subroutine sd
259
260     function btod(binaryString) result(decimalValue)
261         implicit none
262         character(len=*), intent(in) :: binaryString
263         integer :: decimalValue
264         integer :: i, length
265
266         decimalValue = 0
267         length = len_trim(binaryString)  ! Get the length of the
    binary string
268
269         ! Convert binary string to decimal
270         do i = 1, length
271             if (binaryString(i:i) == '1') then
272                 decimalValue = decimalValue + 2**(length - i)
273             else if (binaryString(i:i) /= '0') then
274                 print *, "Invalid character in binary string."
275                 decimalValue = -1  ! Indicate an error with -1
276                 return
```

```
277             end if
278         end do
279     end function btod
280
281     subroutine BERPVR(s,s1,site,vin,be)
282         implicit none
283         integer::s, s1,site(0:s1-1)
284         real*8::be,betemp,vin(0:2**s-1),rdm(0:2**s1-1,0:2**s1-1)
285         integer::i,INFO
286         real*8::WORK(0:3*(2**s1)-1),W(0:2**s1-1)
287
288         call PTRVR(s,s1,site,vin,rdm)
289         call DSYEV('N','U', 2**s1, rdm, 2**s1, W, WORK,3*(2**s1)-1,
    INFO)
290         be=0
291         betemp=0
292         do i=0,2**s1-1,1
293             if (W(i)>0.0000000000001) then
294                 betemp=-(dlog(abs(W(i)))*abs(W(i)))/dlog(2.0d0)
295                 be=be+betemp
296             end if
297         end do
298     end subroutine
299
300     subroutine PTRVR(s,s1,site,vin,rdm)
301         implicit none
302         integer::s, s1,s2
303         real*8::trace
304         real*8,dimension(0:2**s-1)::vin
305         real*8,dimension(0:2**s1-1,0:2**s1-1)::rdm
306         integer::ii,i1,a,i0,ia,j1,oo,k1,x1,y1,j,t1,t2,z1,i
307         integer,dimension(0:s1-1)::site,site2
308         integer,dimension(0:s1-1)::bin
309         integer,dimension(0:(2**(s-s1))*(2**s1)-1)::ind
310         s2=s-s1
311         x1=0
312         y1=0
313         ind=0
```

```
314         do j1=2**s1-1,0,-1
315             do ii=0,2**s-1
316                 a=ii
317                 do i1=0,s1-1,1
318                     i0=site(i1)
319                     call DTOBONEBIT(a,ia,i0,s)
320                     site2(i1)=ia
321                 enddo
322                 call DTOB(j1,bin,s1)
323                 oo=1
324                 do k1=0,s1-1,1
325                     oo=oo*(bin(k1)-site2(k1))
326                 end do
327                 if(abs(oo)==1) then
328                     ind(x1)=ii
329                     x1=x1+1
330                 end if
331             end do
332         end do
333         rdm=0.0d0
334         do t1=0,2**s1-1,1
335             do t2=0,2**s1-1,1
336                 do z1=0,2**s2-1,1
337                     rdm(t1,t2)=rdm(t1,t2)+vin(ind((2**s2)*t1+z1)) &
338                     *vin(ind((2**s2)*t2+z1))
339                 enddo
340             end do
341         end do
342     end subroutine
343
344     subroutine DTOB(m,tt,s)
345         implicit none
346         integer::s
347         integer,dimension(0:s-1)::tt
348         integer::m,k,a2
349         tt=0
350         a2=m
351         do k = 0,s-1,1
```

```
352             tt(s-k-1) = mod(a2,2)
353             a2 = a2/2
354             if (a2== 0) then
355                 exit
356             end if
357         end do
358     end subroutine
359
360     subroutine DTOBONEBIT(m,ia,i0,s)
361         implicit none
362         integer*4::s
363         integer,dimension(0:s-1)::tt
364         integer::m,ia,i0
365         call DTOB(m,tt,s)
366         ia=tt(i0)
367     end subroutine
368 end program
```

Now, to compile this code, we have to modify our command as,

```
!For compiling the program using multiple GPU cores with the name of
!the file as "programname.f90", the command is:
nvfortran -acc -Minfo=accel -fast -ta=tesla:ccXX programname.f90 -
    llapack

!ccXX represents GPU compatibility; update it according to your GPU.
! E.g. for RTX 2080, XX should be replaced by 75.
```

For running this program, the command remains the same as earlier,

```
!For running the program using multiple CPU cores, the command is:
time mpirun -np 6 ./a.out
```

Chapter 4

Entanglement Studies in the AKLT Model

4.1 Entanglement

Entanglement is the most phenomenal concept in quantum mechanics. Two or more particles can be entangled in such a way that the measurement on any one part of the state of a combined system affects the state of the other part, regardless of how far they are. This fact was first stated by Schrödinger in his paper titled *"The Characteristic Trait of Quantum Mechanics"* [21] as a reply to Einstein, Podolsky, and Rosen's paper titled *"Can Quantum-Mechanical Description of Physical Reality Be Considered Complete?"* [20], questioning the formulation of quantum mechanics. This phenomenal concept can be used in many applications, like teleportation [22], super-dense coding, and quantum cryptography [23]. So it is very important to measure entanglement (check whether the parties concerned are entangled or not?) and quantify entanglement [24] therein (if they are entangled) to study the behaviour of the system.

In quantum mechanics, a physical system is associated with a corresponding abstract vector in Hilbert space, which can be used to determine all its properties if it is a pure state. Systems consisting of spin $\frac{1}{2}$-particles or photons (i.e systems with two states) can be represented by a qubit. Any system in quantum mechanics can be in

a pure or a mixed state. If a system is in a single state with a probability equal to 1, then we say the system is in a pure state represented by $|\psi\rangle$ (can also be written in the form of a density matrix $\rho = |\psi\rangle\langle\psi|$), however, if the system is in a probabilistic (probabilities add to 1) mixture of pure states, then we say the system is in a mixed state represented as

$$\rho = \sum_i p_i |\psi_i\rangle\langle\psi_i|, \tag{4.1}$$

where $|\psi_i\rangle$'s are the pure states and p_i are the corresponding probabilities which forms an ensemble $\{p_i, |\psi_i\rangle\}$ such that $\sum_i p_i = 1$ and $0 \leqslant p_i \leqslant 1$ [23].

Entanglement can exist for pure and mixed states. A pure state is entangled if it is not possible to write the state as a tensor product state (i.e. $\rho^{AB} \neq \rho^A \otimes \rho^B$). A mixed state is entangled if it cannot be represented as a mixture of unentangled pure states (i.e. $\rho^{AB} \neq \sum_i p_i \rho_i^A \otimes \rho_i^B$) [25].

For quantifying the entanglement, there are various measures for pure states as well as mixed states [24], such as von Neumann entropy, logarithmic negativity, entanglement of formation, concurrence, etc., out of which we are using concurrence and von Neumann entropy for studying our AKLT system.

4.2 Concurrence

Concurrence is a well-defined method by William K. Wootters for quantifying entanglement for a 2-qubit system [26, 27]. For a 2-qubit pure state $|\psi\rangle$, concurrence is defined as,

$$\mathcal{C} = |\langle\psi|\tilde{\psi}\rangle|, \tag{4.2}$$

where,

$$|\tilde{\psi}\rangle = (\sigma_y \otimes \sigma_y)|\psi^*\rangle, \tag{4.3}$$

where $|\tilde{\psi}\rangle$ is the "spin flip" transformation for a 2 qubit pure state.

For a 2 qubit mixed state ρ, concurrence is defined as,

$$\mathcal{C}(\rho) = \max\{0, \lambda_1 - \lambda_2 - \lambda_3 - \lambda_4\}, \tag{4.4}$$

where $\lambda_1, \lambda_2, \lambda_3, \lambda_4$ are square root of the eigenvalues of $\rho\tilde{\rho}$ in non increasing order and $\tilde{\rho}$ is defined as,

$$\tilde{\rho} = (\sigma_y \otimes \sigma_y)\rho^*(\sigma_y \otimes \sigma_y), \tag{4.5}$$

where $\tilde{\rho}$ is the "spin flip" transformation for 2 qubit density matrix. One can visualise $(\sigma_y \otimes \sigma_y)$ as,

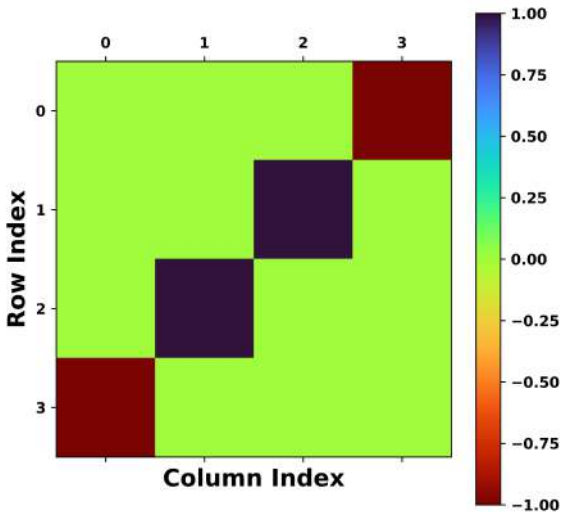


Figure 4.1: Spin-Flip Matrix

- For a **separable state**, **concurrence** comes to be **zero** (i.e. $\mathcal{C} = 0$).
- For a **maximally entangled state**, **concurrence** comes to be **one** (i.e. $\mathcal{C} = 1$).

Now, let us study the entanglement properties of our system using concurrence. Here is the Fortran code implementation of the concurrence on our AKLT system.

```
program spin1qubit
    implicit none
    integer, parameter :: d =2, s = 4, s1 = 2
    integer, parameter :: num = d**s, N = num, di = d**s1
    integer :: i, j, k, l, t, flagx, flagy, pos, u
    integer, allocatable :: digits1(:), digits2(:), decimal(:), site
   (:)
    character(len=:), allocatable :: state(:)
    double precision, allocatable :: E(:,:), C(:,:), F(:,:), G(:,:),
```

```
    H(:,:), Z(:,:)
    character*1 :: JOBZ, UPLO, RANGE
    integer ::  LDA = num, LWORK = 8*num, INFO, IL, IU, M, LDZ,
   IWORK(5*N), IFAIL(N)
    double precision :: W(num), WORK(8*num), VL, VU, ABSTOL
    real*8 :: sum, trace, con, Jo, B
    real*8, allocatable :: psi(:,:), rho(:,:), SF(:,:), RT(:,:), RRT
   (:,:)
    real*8, allocatable :: rdm(:,:), RW(:), RWORK(:) !,red_rho(:,:)
    integer :: RLDA = di, RINFO, RLWORK = 3*di -1

    ! Allocate arrays after determining the value of num
    allocate(character(len=s) :: state(num))
    allocate(integer :: decimal(num))
    allocate(integer :: digits1(s), digits2(s))
    allocate(real(8) :: C(num, num))
    allocate(real(8) :: E(num, num))
    allocate(real(8) :: F(num, num))
    allocate(real(8) :: G(num, num))
    allocate(real(8) :: H(0:num-1 , 0:num-1))
    allocate(real(8) :: Z(0:num-1 , 0:num-1))
    allocate(psi(0:num-1,0:0))
    allocate(rho(0:num-1, 0:num-1))
    allocate(rdm(di, di))
    ! allocate(red_rho(0:di-1,0:di-1))
    allocate(site(0:s1-1))
    allocate(SF(di,di))
    allocate(RT(di,di))
    allocate(RRT(di,di))
    allocate(RW(di))
    allocate(RWORK(RLWORK))

    do i = 0, num - 1
        call integer_binary(i, state(i+1), s)  ! Adjusted for 1-
   based indexing
        !print *, state(i)
    end do
    do i = 1, num
```

```
43          decimal(i) = btod(state(i))
44          !print*, ' Decimal equivalent of ', state(i) ,' = ', decimal
    (i)
45       end do
46       G = 0.0d0
47       do k =1 , s
48          E = 0.0d0
49          do i = 1, num
50              do j = 1, num
51                  ! Computing Sigma_i^x * Sigma_i+1^x for each site
52                  flagx = 0
53                  if (decimal(i) .ne. decimal(j)) then
54                      call sd(state(i), digits1)
55                      call sd(state(j), digits2)
56                      if (k < s) then
57                          if (digits1(k) .ne. digits2(k) .and. digits1
    (k+1) .ne. digits2(k+1)) then
58                              do l = 1, k - 1
59                                  if (digits1(l) .ne. digits2(l)) then
60                                      flagx  = flagx + 1
61                                  end if
62                              end do
63                              do l = k + 2, s
64                                  if (digits1(l) .ne. digits2(l)) then
65                                      flagx  = flagx + 1
66                                  end if
67                              end do
68                              if (flagx == 0) then
69                                  E(i, j) = E(i, j) + 1.0
70                              else
71                                  E(i, j) = E(i, j) + 0.0
72                              end if
73                          else
74                              E(i, j) = E(i, j) + 0.0
75                          end if
76                      else
77                          if (digits1(1) .ne. digits2(1) .and. digits1
    (s) .ne. digits2(s)) then
```

```
                            do l = 2, s - 1
                                if (digits1(l) .ne. digits2(l)) then
                                    flagx  = flagx + 1
                                end if
                            end do
                            if (flagx == 0) then
                                E(i, j) = E(i, j) + 1.0
                            else
                                E(i, j) = E(i, j) + 0.0
                            end if
                        else
                            E(i, j) = E(i, j) + 0.0
                        end if
                    end if
                else
                    E(i, j) = E(i, j) + 0.0
                end if
                ! Computing Sigma_i^y * Sigma_i+1^y for each site
                flagy = 0
                if (decimal(i) .ne. decimal(j)) then
                    call sd(state(i), digits1)
                    call sd(state(j), digits2)
                    if (k < s) then
                        if (digits1(k) .ne. digits2(k) .and. digits1
(k+1) .ne. digits2(k+1)) then
                            do l = 1, k - 1
                                if (digits1(l) .ne. digits2(l)) then
                                    flagy  = flagy + 1
                                end if
                            end do
                            do l = k + 2, s
                                if (digits1(l) .ne. digits2(l)) then
                                    flagy  = flagy + 1
                                end if
                            end do
                            if (flagy == 0) then
                                E(i, j) = E(i, j) + (-1.0 * (-1.0)
**(digits2(k) + digits2(k+1)))
```

```
                            else
                                E(i, j) = E(i, j) + 0.0
                            end if
                        else
                            E(i, j) = E(i, j) + 0.0
                        end if
                    else
                        if (digits1(1) .ne. digits2(1) .and. digits1
(s) .ne. digits2(s)) then
                            do l = 2, s - 1
                                if (digits1(l) .ne. digits2(l)) then
                                    flagy  = flagy + 1
                                end if
                            end do
                            if (flagy == 0) then
                                E(i, j) = E(i, j)  + (-1.0 * (-1.0)
**(digits2(s) + digits2(1)))
                            else
                                E(i, j) = E(i, j) + 0.0
                            end if
                        else
                            E(i, j) = E(i, j) + 0.0
                        end if
                    end if
                else
                    E(i, j) = E(i, j) + 0.0
                end if
                ! Computing Sigma_i^z * Sigma_i+1^z for each site
                call sd(state(i), digits1)
                if (decimal(i) .ne. decimal(j)) then
                    E(i, j) =  E(i,j) + 0.0
                else
                    if (k < s) then
                        E(i, j) = E(i, j) + (1 - 2 * digits1(k)) *
(1 - 2 * digits1(k+1))
                    else
                        E(i, j) = E(i, j) + (1 - 2 * digits1(1)) *
(1 - 2 * digits1(s))
```

```
148                     end if
149                 end if
150                 ! Computing Sigma_i^z for each site
151                 call sd(state(i), digits1)
152                 if (decimal(i) .ne. decimal(j)) then
153                     G(i, j) =  G(i,j) + 0.0
154                 else
155                     G(i, j) = G(i, j) + (1 - 2 * digits1(k))
156                 end if
157             end do
158         end do
159     C = C + (1.0d0/4.0d0)*E
160     F = F + matmul((1.0d0/4.0d0)*E,(1.0d0/4.0d0)*E)
161     end do
162     ! write(58,*) C
163
164     do k = 0, s-2
165         do l = k+1, s-1
166             site = (/k, l/)
167                 do t = 0, 200
168                     Jo =  0.01 * t
169                     B = 1.0d0
170                     H = 0.0d0
171                     do i = 1, num
172                         do j = 1, num
173                             H(i-1,j-1) = Jo * (C(i,j) + (1.0d0/3.0d0
    )* F(i,j)) + B * (1.0d0/2.0d0)* G(i,j)
174                             !if (i .le. j) then
175                                 ! write(57,*) i, j, H(i-1,j-1)
176                             !end if
177                         end do
178                     end do
179
180                     ! Define parameters for DSYEVX
181                     JOBZ = 'V'  ! Compute eigenvalues and
    eigenvectors
182                     RANGE = 'I' ! Compute selective eigenvalues and
    eigenvectors
```

```
183                     UPLO = 'U'  ! Upper triangular part of A is
    stored
184                     ABSTOL = 0.0d0
185                     IL = 1
186                     IU = 1
187                     M = IU - IL + 1
188                     LDZ = N
189                     ! Call DSYEVX to compute ground state eigenvalue
     and eigen vector
190                     call DSYEVX (JOBZ, RANGE, UPLO, N, H, LDA, VL,
    VU, IL, IU, ABSTOL &
191                     , M, W, Z, LDZ, WORK, LWORK, IWORK, IFAIL, INFO)
192                     ! Check for successful execution
193                     ! sum = 0.0d0
194                     ! if (INFO == 0) then
195                     !     do i = 1, M
196                     !         write(3,*) B, W(i)
197                     !     end do
198                     ! else
199                     !     print*, 'Error in DSYEVX, info =', INFO
200                     ! end if
201
202                     do i = 0, M-1
203                         do j = 0, num - 1
204                             if (abs(Z(j,i)) .le. 0.000000000001 .and
    . abs(Z(j,i)) .ge. 0) then
205                                 psi(j,i) = 0
206                             else
207                                 psi(j,i) = Z(j,i)
208                             end if
209                             ! write(7,*) j, i, psi(j,i)
210                         end do
211                     end do
212
213                     rho = matmul(psi, transpose(psi))
214                     ! do i = 0, num-1
215                     !     do j = 0, num-1
216                     !         if(i .eq. j) then
```

```
217                     !             trace = trace + rho(i,j)
218                     !         end if
219                     !         write(7,*) i, j, rho(i, j)
220                     !     end do
221                     ! end do
222                     ! print*, "Trace = ", trace
223
224                     call ptrace(rho, d, s, s1, site, rdm)
225                     ! trace = 0.0d0
226                     ! do i = 0, di-1
227                     !     do j = 0, di-1
228                     !         ! rdm(i+1,j+1) = red_rho(i,j)
229                     !         if (i .eq. j) then
230                     !             trace = trace + rdm(i+1,j+1)
231                     !         end if
232                     !         ! write(7,*) i, j, rdm(i+1, j+1)
233                     !     end do
234                     ! end do
235                     ! print*, "Trace = ", trace
236
237                     call spin_flip_matrix(d,SF)              !
    Generating Spin Flip Matrix
238                     RT = matmul(SF, matmul(rdm, SF))
239                     RRT = matmul(rdm, RT)
240
241                     ! Define parameters for DSYEVX
242                     JOBZ = 'N'  ! Compute eigenvalues only
243                     ! Call DSYEV to compute eigenvalues and
    eigenvectors
244                     call DSYEV(JOBZ, UPLO, di, RRT, RLDA, RW, RWORK,
     RLWORK, RINFO)
245                     sum = 0.0d0
246                     ! Check for successful execution
247                     if (RINFO == 0) then
248                         call sort_dec(RW, di)
249                         do i = 2, di
250                             sum = sum + sqrt(abs(RW(i)))
251                         end do
```

```
252                     else
253                         print*, 'Error in DSYEV, info =', RINFO
254                     end if
255                     con = max(0.0d0, (sqrt(abs(RW(1)))- sum))
256                     pos = k*10 + l
257                     write(400+pos,*) Jo, con
258                 ! end do
259             end do
260         end do
261     end do
262 
263 contains
264 
265     subroutine integer_binary(num, binary_string, n)
266         implicit none
267         integer :: num
268         integer, intent(in) :: n
269         integer :: i, temp_num
270         character(len=n), intent(out) :: binary_string
271         temp_num = num
272         ! Initialize binary_string to all zeros
273         binary_string = repeat('0', n)
274         ! Convert the integer to binary
275         do i = 0, n - 1
276             if (mod(temp_num, 2) == 1) then
277                 binary_string(n-i:n-i) = '1'  ! Set the bit to '1'
278             else
279                 binary_string(n-i:n-i) = '0'  ! Set the bit to '0'
280             end if
281             temp_num = temp_num / 2  ! Divide num by 2 to shift
    right
282         end do
283     end subroutine integer_binary
284 
285     subroutine sd(binaryString, digits)
286         character(len=*), intent(in) :: binaryString
287         integer, allocatable, intent(out) :: digits(:)
288         integer :: i, len
```

```
289         len = len_trim(binaryString)  ! Get the length of the binary
     string
290         allocate(digits(len))          ! Allocate array to hold
    digits
291         ! Convert each character to an integer
292         do i = 1, len
293             if (binaryString(i:i) == '1') then
294                 digits(i) = 1
295             else if (binaryString(i:i) == '0') then
296                 digits(i) = 0
297             else
298                 print *, "Invalid character in binary string."
299                 digits = 0  ! Set to zero if invalid character is
    found
300                 return
301             end if
302         end do
303     end subroutine sd
304
305     function btod(binaryString) result(decimalValue)
306         implicit none
307         character(len=*), intent(in) :: binaryString
308         integer :: decimalValue
309         integer :: i, length
310
311         decimalValue = 0
312         length = len_trim(binaryString)  ! Get the length of the
    binary string
313
314         ! Convert binary string to decimal
315         do i = 1, length
316             if (binaryString(i:i) == '1') then
317                 decimalValue = decimalValue + 2**(length - i)
318             else if (binaryString(i:i) /= '0') then
319                 print *, "Invalid character in binary string."
320                 decimalValue = -1  ! Indicate an error with -1
321                 return
322             end if
```

```
323         end do
324     end function btod
325
326     subroutine ptrace(rho, d, s, s1, site, rdm)
327         implicit none
328         integer, intent(in) :: d, s, s1
329         integer, intent(in) :: site(0:s1-1)
330         real*8, intent(in) :: rho(0:d**s-1, 0:d**s-1)
331         real*8, intent(out) :: rdm(d**s1, d**s1)
332         integer, allocatable :: idits(:), jdits(:), ridits(:),
    rjdits(:), to(:)
333         integer :: i, j, k, p, flag, ri, rj
334         real*8 :: a
335
336         allocate(idits(0:s-1), jdits(0:s-1))
337         allocate(ridits(0:s1-1), rjdits(0:s1-1))
338         allocate(to(0:s-s1-1))
339
340         ! Compute the indices to trace out
341         call TOI(s, site, s1, to)
342
343         rdm = 0.0d0
344         do i = 0, d**s - 1
345             do j = 0, d**s - 1
346                 if (rho(i,j) .ne. 0.0d0) then
347                     a = rho(i,j)
348                     call DECTOD(i, d, s, idits)
349                     call DECTOD(j, d, s, jdits)
350                     flag = 0
351                     do k = 0, s - s1 - 1
352                         if (idits(to(k)) .ne. jdits(to(k))) then
353                             flag = 1
354                             exit
355                         end if
356                     end do
357                     if (flag .eq. 0) then
358                         do k = 0, s1 - 1
359                             p = site(k)
```

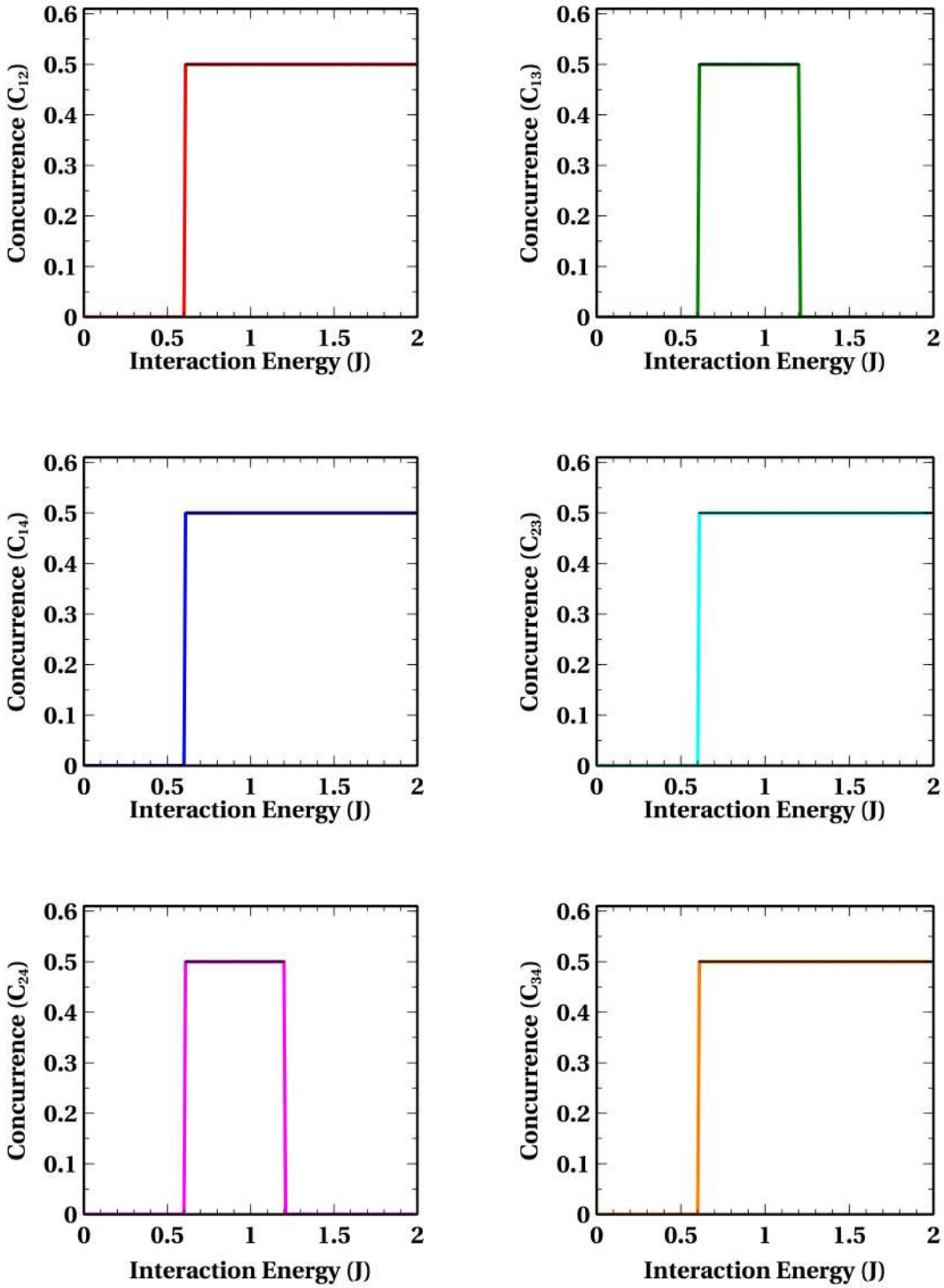
```
360                             ridits(k) = idits(p)
361                             rjdits(k) = jdits(p)
362                         end do
363                         call DTODEC(d, s1, ridits, ri)
364                         call DTODEC(d, s1, rjdits, rj)
365                         rdm(ri+1, rj+1) = rdm(ri+1, rj+1) + a
366                     end if
367                 end if
368             end do
369         end do
370
371         deallocate(idits, jdits, ridits, rjdits, to)
372     end subroutine ptrace
373
374     subroutine DECTOD(dec, d, s, dits)
375         implicit none
376         integer, intent(in) :: dec, d, s
377         integer, allocatable, intent(out) :: dits(:)
378         integer :: i, temp_dec
379
380         allocate(dits(0:s-1))
381
382         temp_dec = dec
383         do i = s-1, 0, -1
384             dits(i) = mod(temp_dec, d)
385             temp_dec = temp_dec / d
386         end do
387     end subroutine DECTOD
388
389     subroutine DTODEC(d, s, dits, dec)
390         implicit none
391         integer, intent(in) :: d, s
392         integer, dimension(0:s-1), intent(in) :: dits
393         integer, intent(out) :: dec
394         integer :: i
395
396         dec = 0
397         do i = 0, s - 1
```

```
398             dec = dec * d + dits(i)
399         end do
400     end subroutine DTODEC
401
402     subroutine TOI(s, site, s1, to)
403         implicit none
404         integer, intent(in) :: s, s1
405         integer, intent(in) :: site(0:s1-1)
406         integer, intent(out) :: to(0:s-s1-1)
407         integer :: i, j, k, found
408
409         k = 0
410         do i = 0, s-1
411             found = 0
412             do j = 0, s1 - 1
413                 if (i == site(j)) then
414                     found = 1
415                     exit
416                 end if
417             end do
418             if (found == 0) then
419                 to(k) = i
420                 k = k + 1
421             end if
422         end do
423     end subroutine TOI
424
425     subroutine spin_flip_matrix(n, SFM)
426         implicit none
427         integer, intent(in) :: n
428         real*8, intent(out) :: SFM(n**2, n**2)  ! Output matrix
429         integer :: size, i, j
430         ! Calculate the size of the matrix (n^2)
431         size = n**2
432         ! Initialize the matrix with zeros
433         SFM = 0.0d0
434         ! Fill the matrix based on the alternating pattern
435         do i = 1, size
```

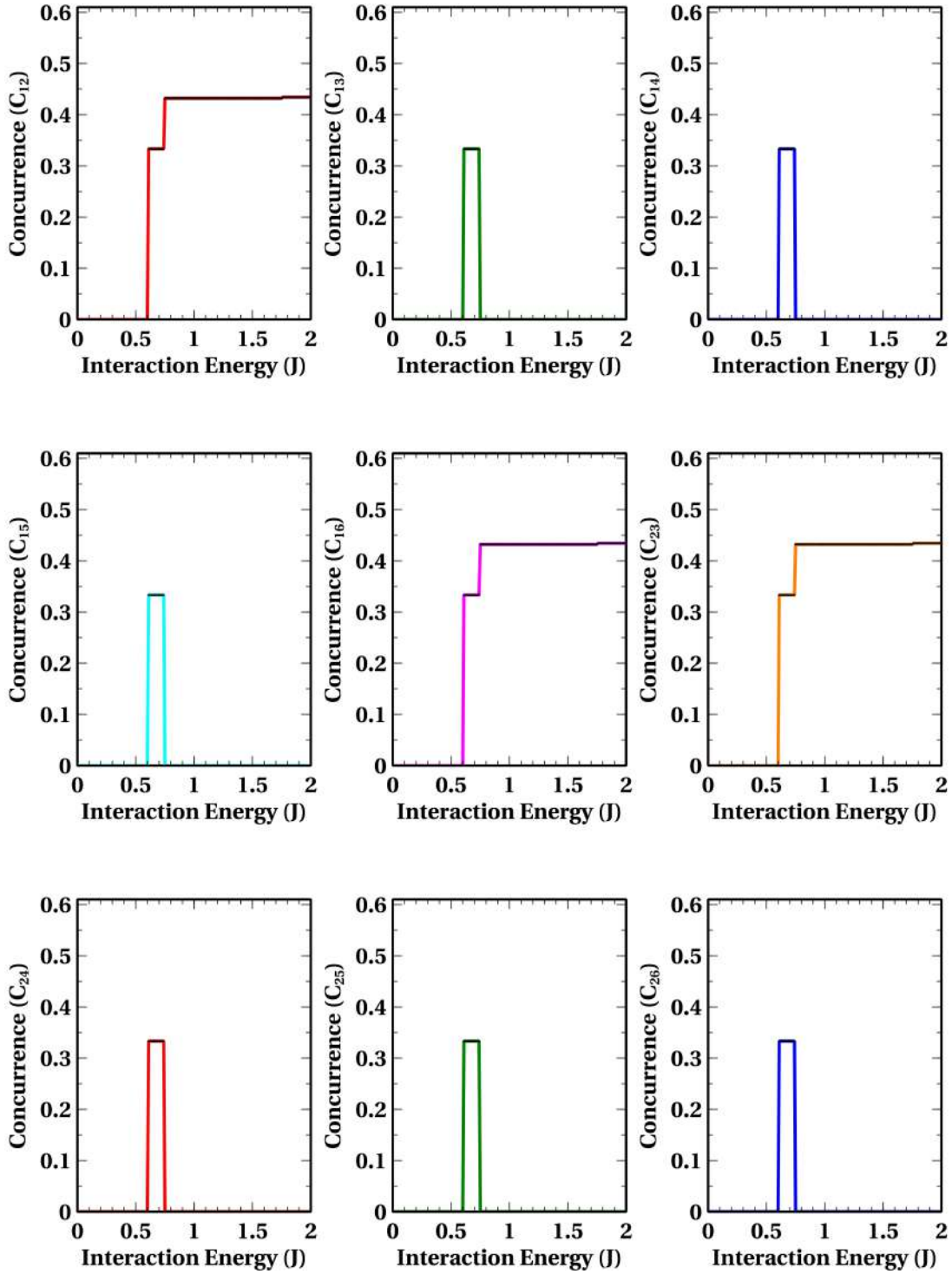
```
436             if (mod(i - 1, n + 1) == 0) then
437                 SFM(i, size - i + 1) = -1
438             else
439                 SFM(i, size - i + 1) = 1
440             end if
441         end do
442     end subroutine spin_flip_matrix
443
444     subroutine sort_dec(arr, n)
445         implicit none
446         integer, intent(in) :: n
447         real*8, intent(inout) :: arr(n)
448         integer :: i, j
449         real*8 :: temp
450         ! Insertion sort (Descending Order)
451         do i = 2, n
452             temp = arr(i)
453             j = i - 1
454             do while (j > 0 .and. arr(j) < temp)
455                 arr(j + 1) = arr(j)
456                 j = j - 1
457             end do
458             arr(j + 1) = temp
459         end do
460     end subroutine sort_dec
461 end program
```

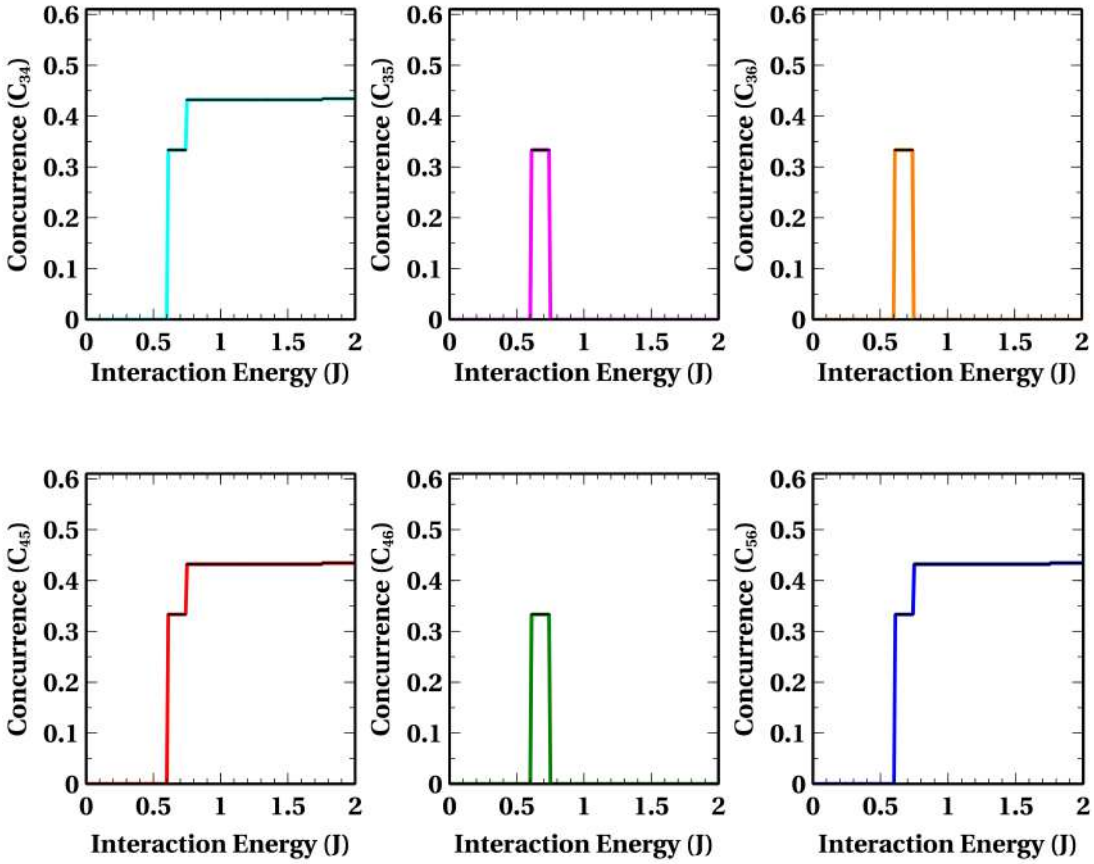
This is the general program to calculate the concurrence for all the combinations of qubits by varying interaction energy (J) and keeping the magnetic field constant as ($B = 1$), where one can change the number of qubits by changing the value of s. Let us see the concurrence plots for different combinations of qubits.

4.2.1 Concurrence vs Interaction Energy (J) for a 4 qubit AKLT Model

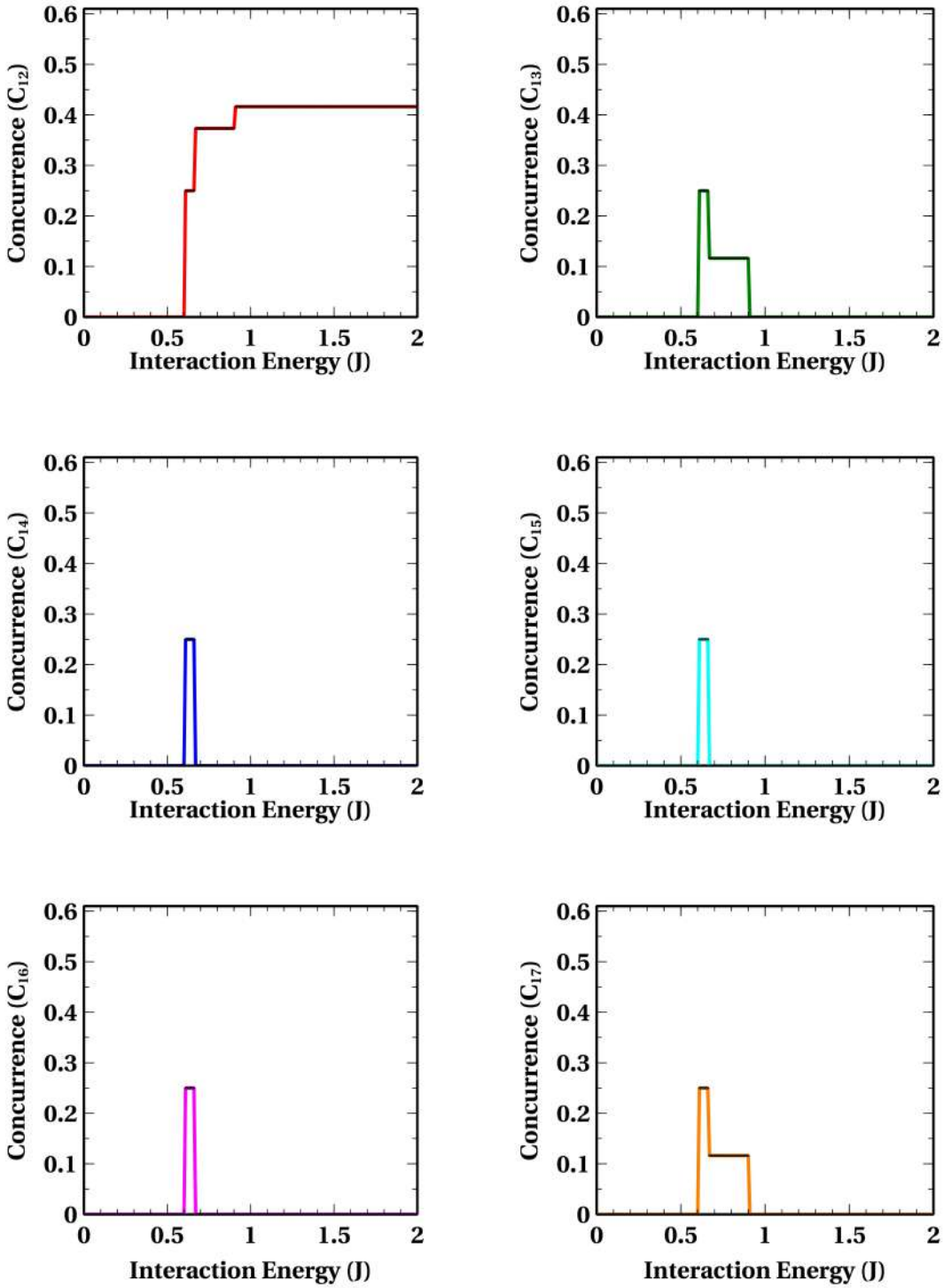


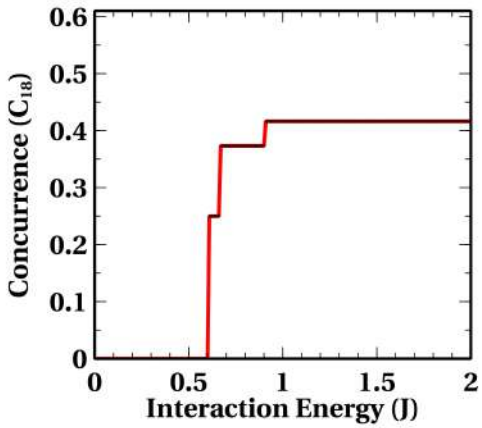
4.2.2 Concurrence vs Interaction Energy (J) for a 6 qubit AKLT Model

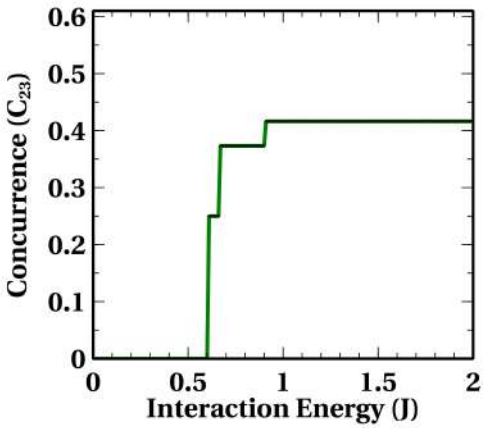


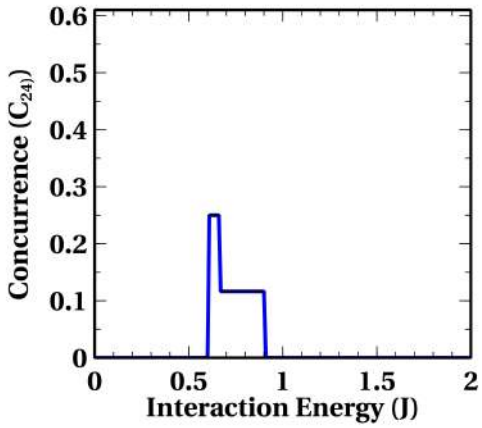

Concurrence (C_{34})
Concurrence (C_{35})
Concurrence (C_{36})
Concurrence (C_{45})
Concurrence (C_{46})
Concurrence (C_{56})
0
0.1
0.2
0.3
0.4
0.5
0.6
0
0.5
1
1.5
2
Interaction Energy (J)

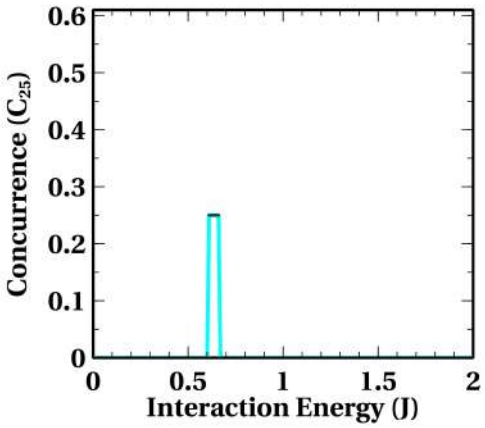
4.2.3 Concurrence vs Interaction Energy (J) for a 8 qubit AKLT Model

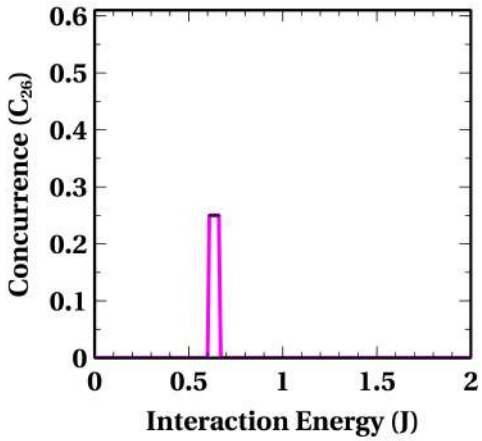


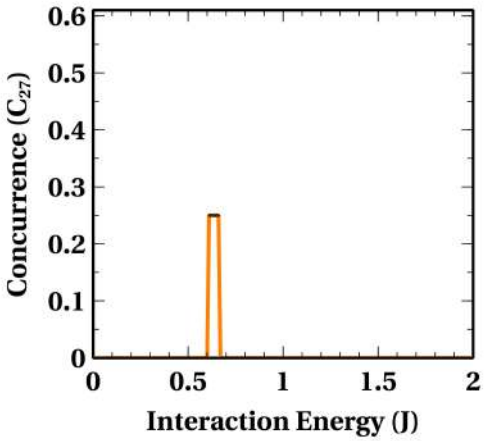

Concurrence (C_{18})
Interaction Energy (J)

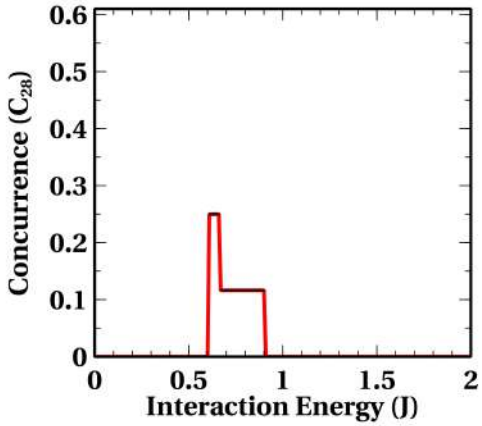

Concurrence (C_{23})
Interaction Energy (J)

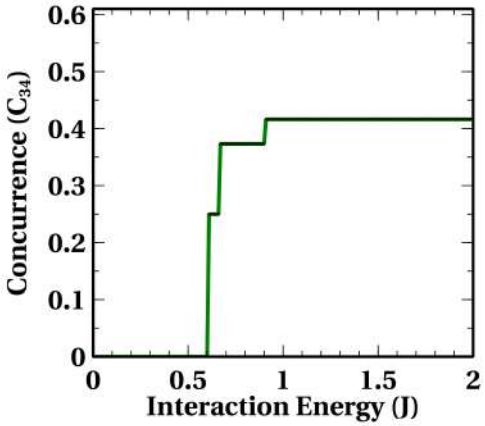

Concurrence (C_{24})
Interaction Energy (J)

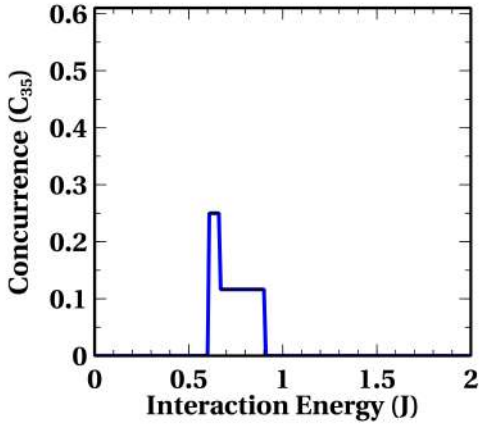

Concurrence (C_{25})
Interaction Energy (J)

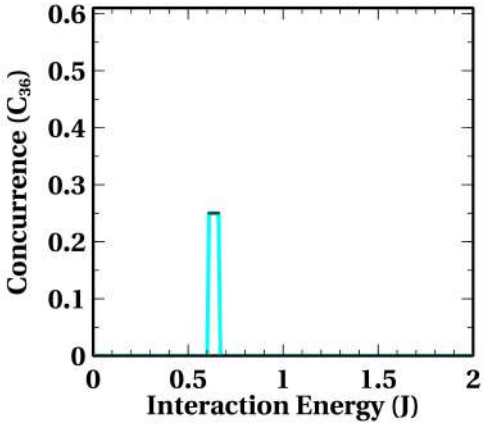

Concurrence (C_{26})
Interaction Energy (J)

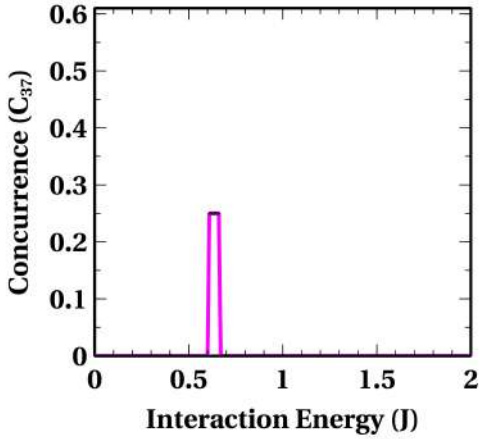

Concurrence (C_{27})
Interaction Energy (J)

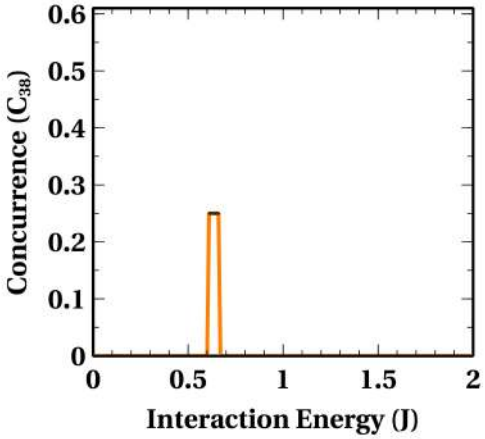

Concurrence (C_{28})
Interaction Energy (J)

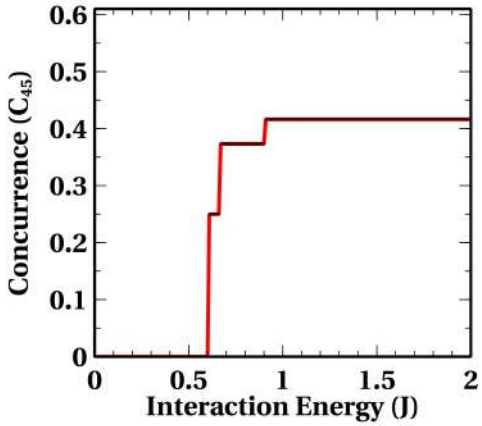

Concurrence (C_{34})
Interaction Energy (J)

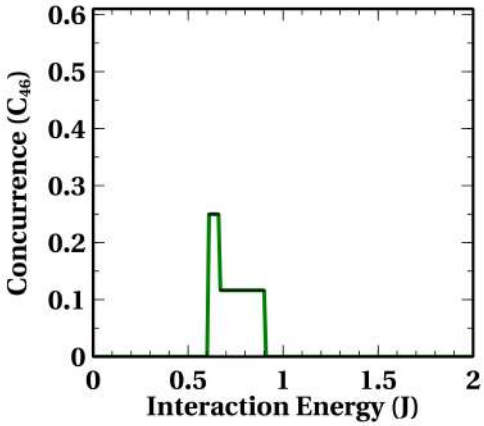

Concurrence (C_{35})
Interaction Energy (J)

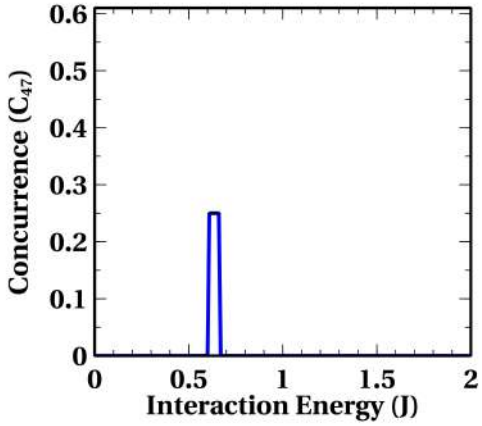

Concurrence (C_{36})
Interaction Energy (J)

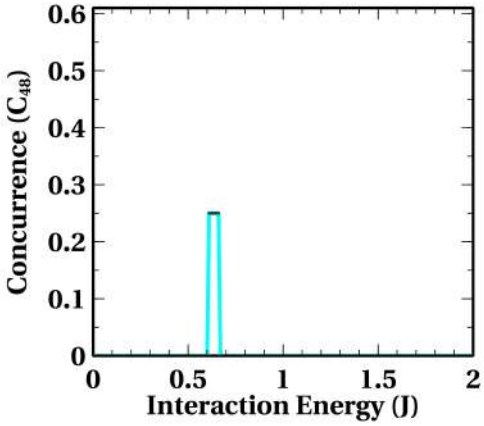

Concurrence (C_{37})
Interaction Energy (J)

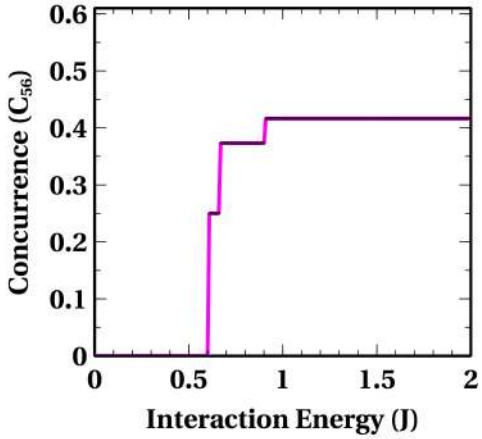

Concurrence (C_{38})
Interaction Energy (J)

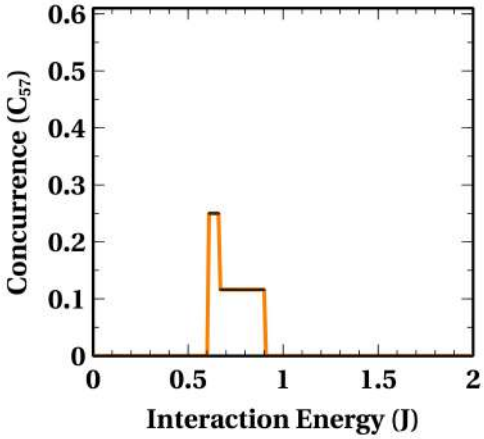

Concurrence (C_{45})
Interaction Energy (J)

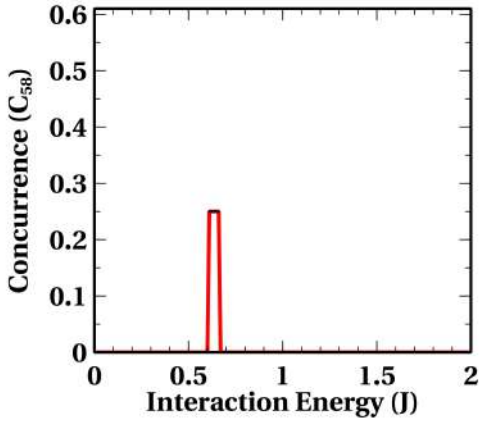

Concurrence (C_{46})
Interaction Energy (J)

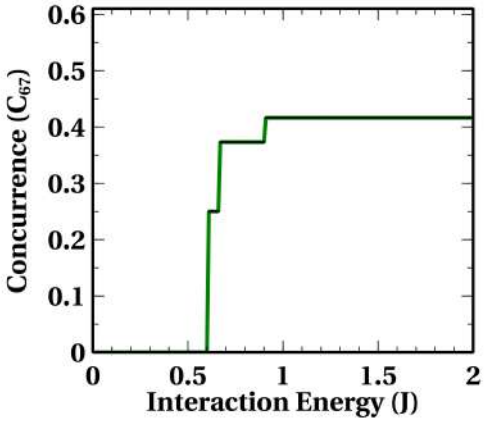

Concurrence (C_{47})
Interaction Energy (J)

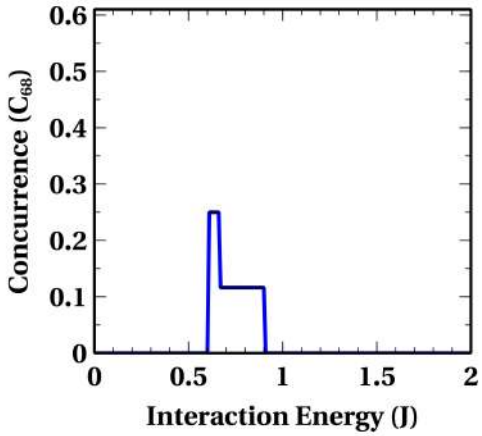

Concurrence (C_{48})
Interaction Energy (J)

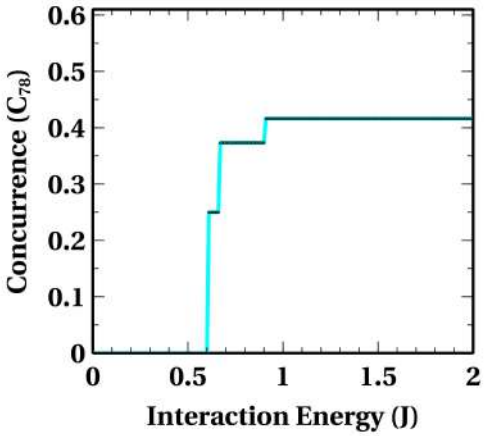

Concurrence (C_{56})
Interaction Energy (J)


Concurrence (C_{57})
Interaction Energy (J)

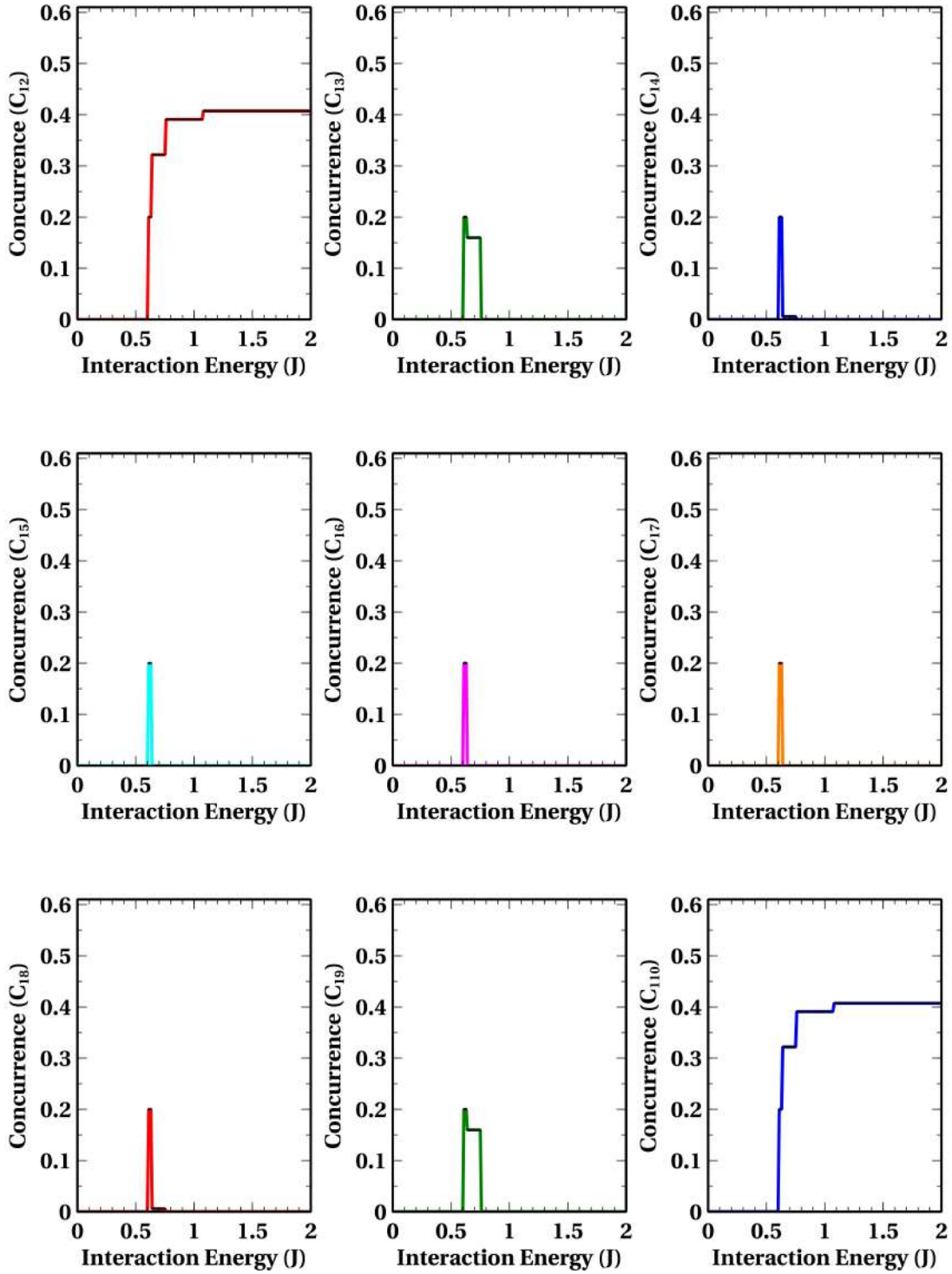

Concurrence (C_{58})
Interaction Energy (J)
0
0.1
0.2
0.3
0.4
0.5
0.6
0
0.5
1
1.5
2

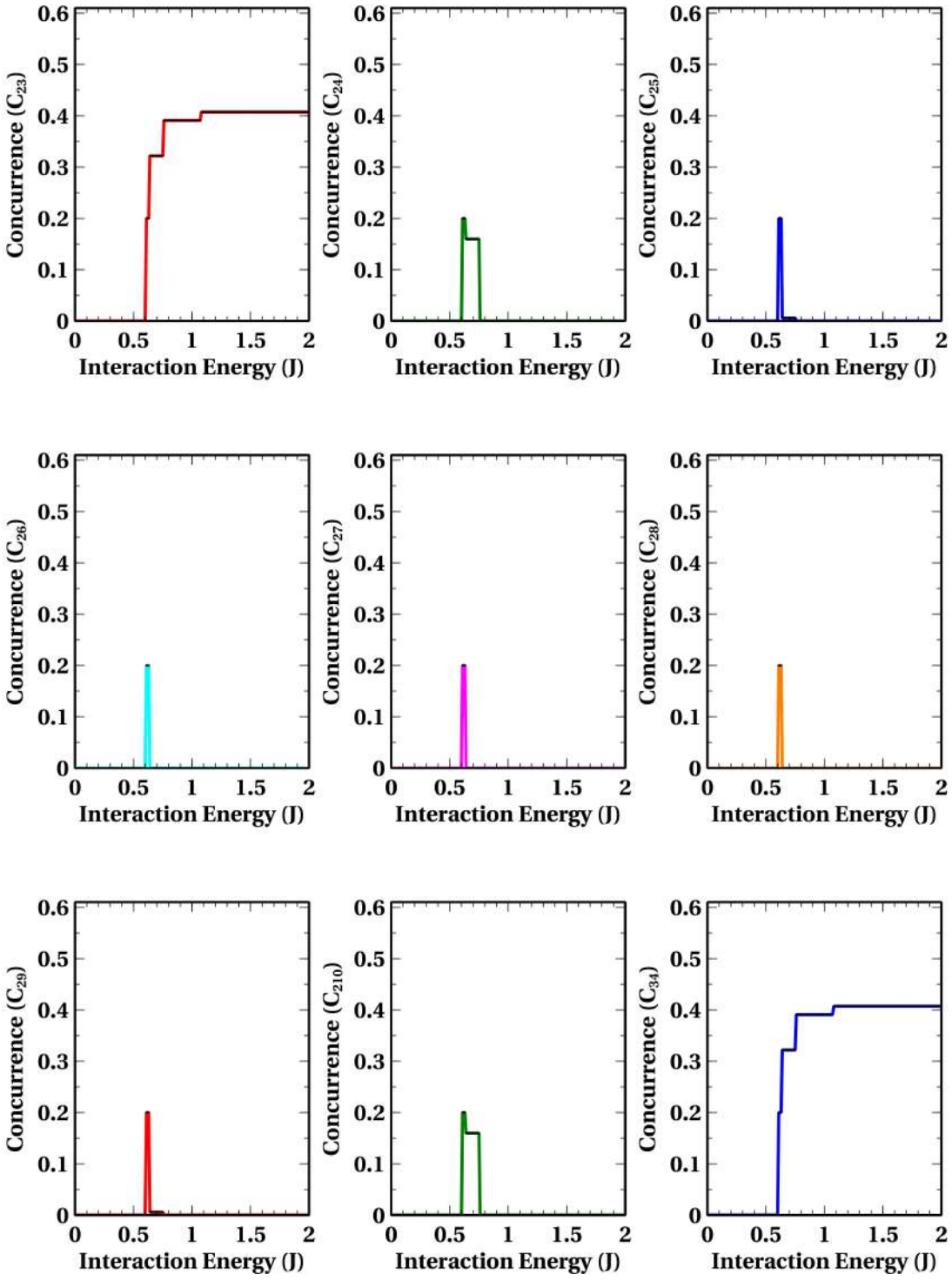

Concurrence (C_{67})
Interaction Energy (J)
0
0.1
0.2
0.3
0.4
0.5
0.6
0
0.5
1
1.5
2

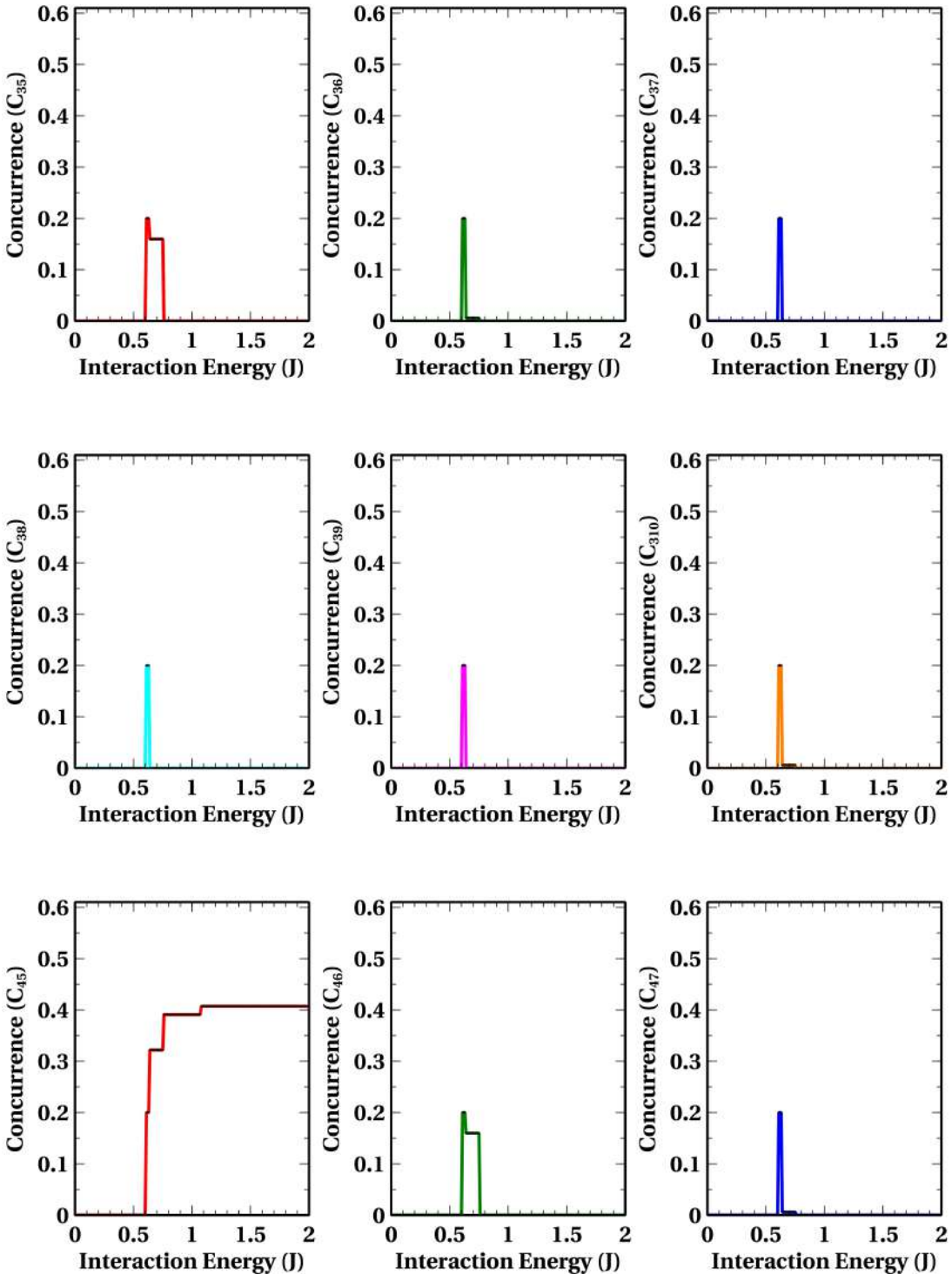

Concurrence (C_{68})
Interaction Energy (J)
0
0.1
0.2
0.3
0.4
0.5
0.6
0
0.5
1
1.5
2

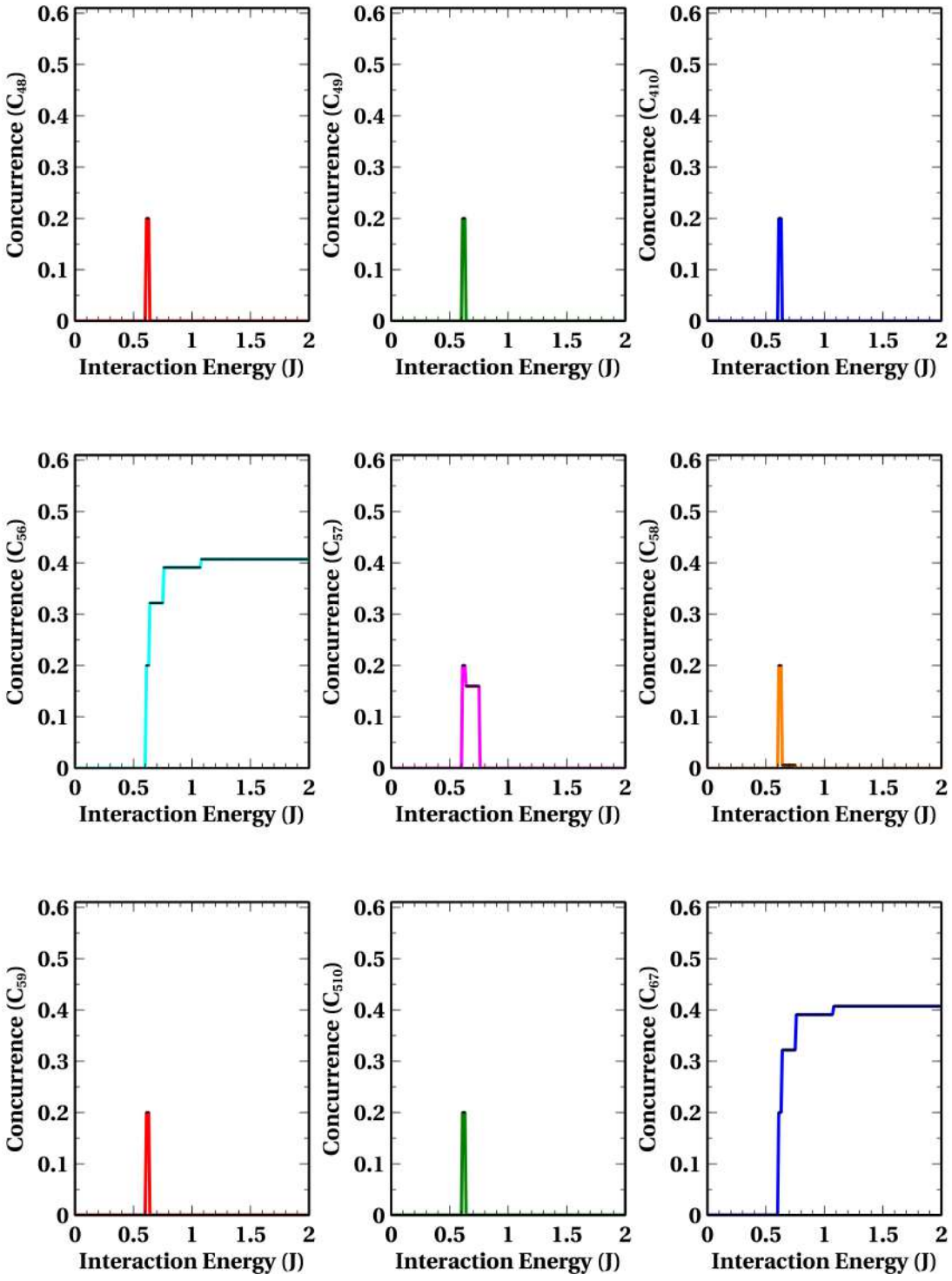

Concurrence (C_{78})
Interaction Energy (J)
0
0.1
0.2
0.3
0.4
0.5
0.6
0
0.5
1
1.5
2

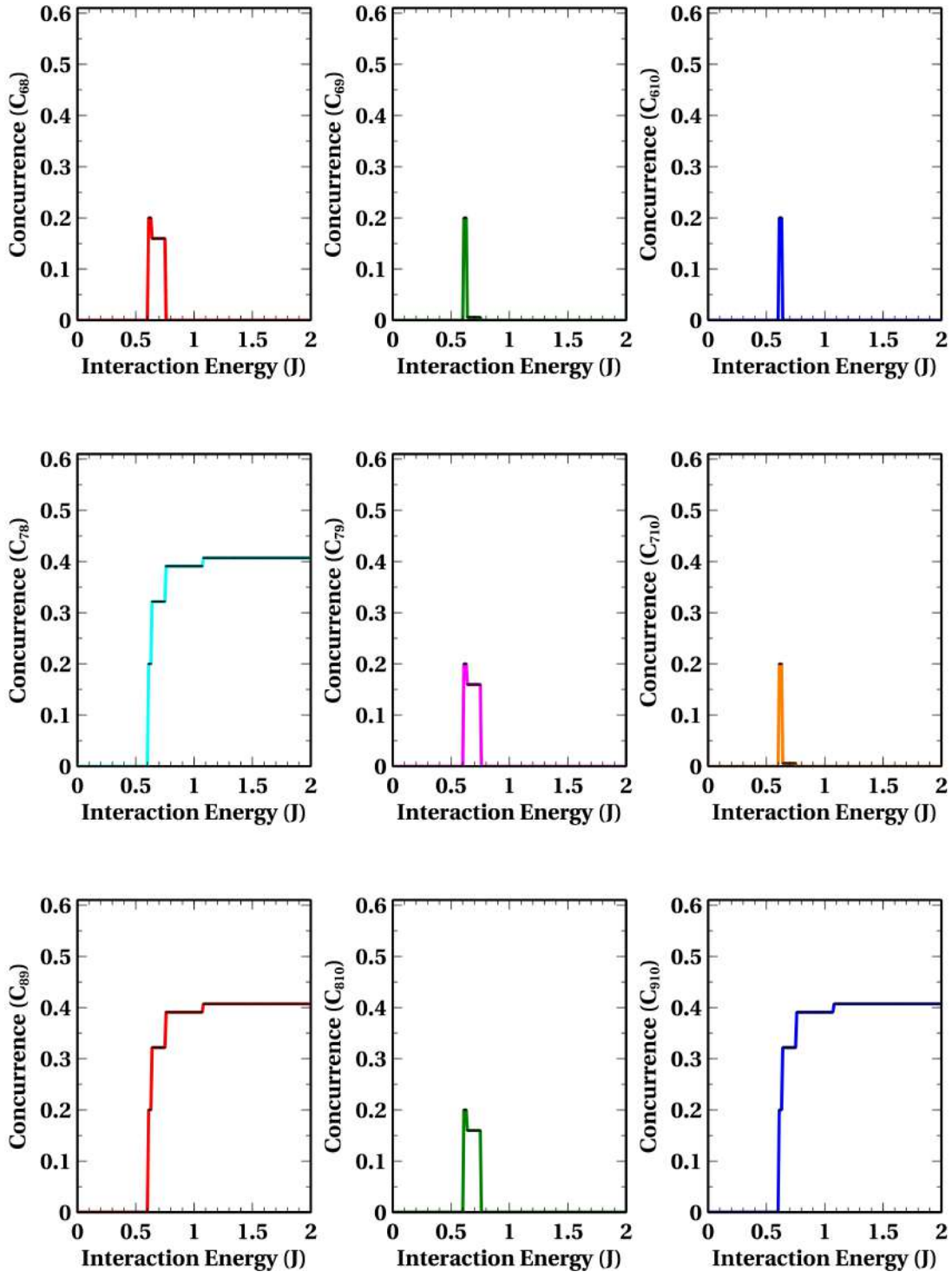
4.2.4 Concurrence vs Interaction Energy (J) for a 10 qubit AKLT Model




Concurrence (C_{23})
Concurrence (C_{24})
Concurrence (C_{25})
Concurrence (C_{26})
Concurrence (C_{27})
Concurrence (C_{28})
Concurrence (C_{29})
Concurrence (C_{210})
Concurrence (C_{34})
Interaction Energy (J)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.5
1
1.5
2


Concurrence (C_{35})
Concurrence (C_{36})
Concurrence (C_{37})
Concurrence (C_{38})
Concurrence (C_{39})
Concurrence (C_{310})
Concurrence (C_{45})
Concurrence (C_{46})
Concurrence (C_{47})
Interaction Energy (J)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.5
1
1.5
2


Concurrence (C_{48})
Concurrence (C_{49})
Concurrence (C_{410})
Concurrence (C_{56})
Concurrence (C_{57})
Concurrence (C_{58})
Concurrence (C_{59})
Concurrence (C_{510})
Concurrence (C_{67})
Interaction Energy (J)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.5
1
1.5
2



From the plots, it is visible that, as the number of qubits increase, the number of transitions (the jumps) also increases. The maximum value of concurrence is 0.5, and as the number of qubits increase, the concurrence decreases, which obeys

the monogamy of entanglement [28], which says that if two parties are maximally entangled, they cannot be entangled with a third party. Therefore, as the number of parties sharing quantum correlations increases, the entanglement between any two individual parties generally decreases. This will be clear later from the comparison between 2-qubit and multiqubit measures of entanglement.

Now, to analyse the plots in more detail, let us see the plots of C_{12} vs interaction energy for different numbers of qubits.

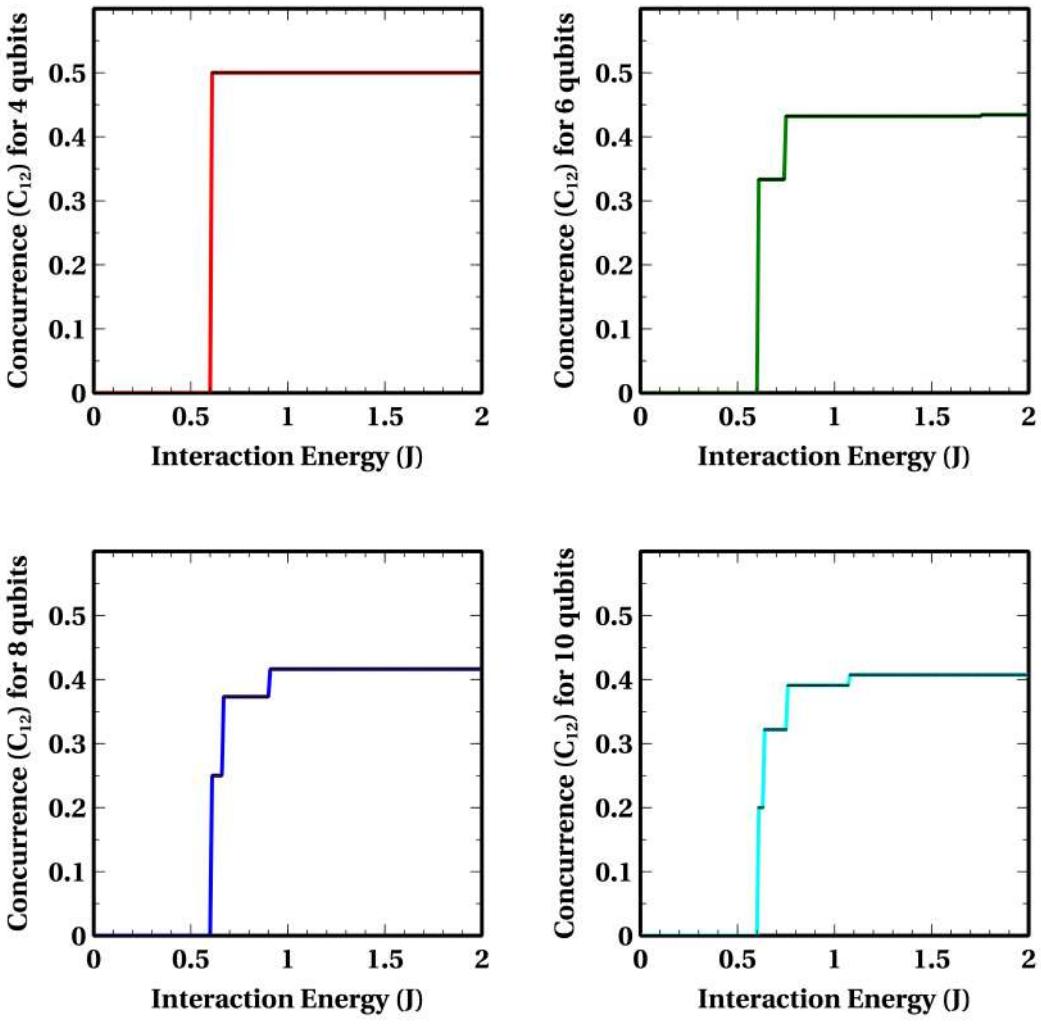


Figure 4.2: Concurrence (C_{12}) vs interaction energy (J)

For further discussion in this section, the concurrence means C_{12} (concurrence between 1 and 2 qubit). So, for 4 qubits, the concurrence is 0.5, and as we increase the number of qubits, it starts to decrease slightly; also, the number of transitions

increases with the increase in the number of qubits, but the stepsize of the transition decreases and from observation we can say that in thermodynamic limit (i.e. $n \to \infty$) it will never cross 0.5, which also obeys the monogamy of entanglement. From plots, one may think that the first transition always happens at J = 0.6. Let us analyse that in detail.

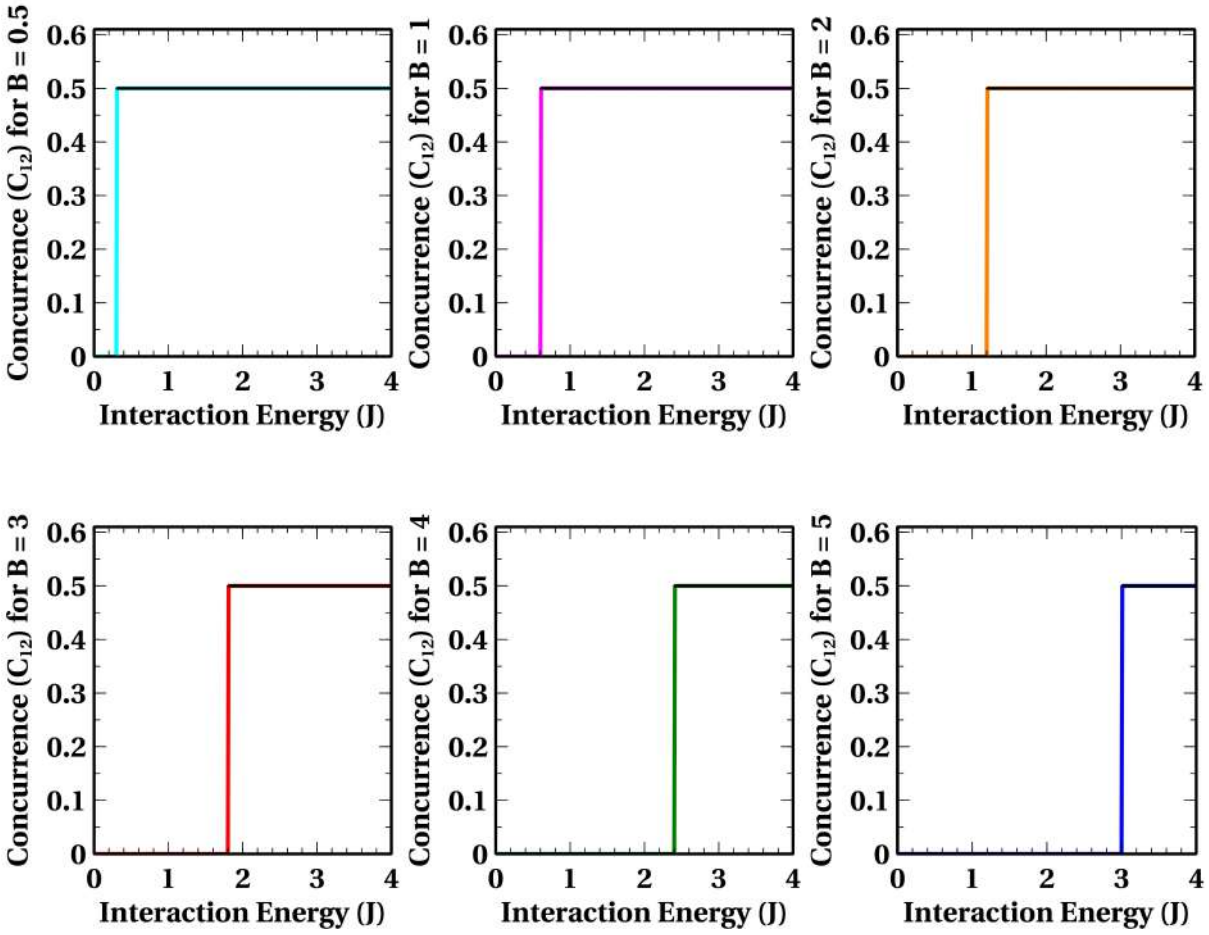


Figure 4.3: Concurrence (C_{12}) vs interaction energy (J) for different values of magnetic field (B).

Now it is clear from the plots in Fig.[4.3], as the value of B changes, the first transition point also changes, which is because the ground state energy changes and thus the ground state wave function. Before doing further analysis, let us verify the concurrence using Mathematica symbolically to reinforce our numerics.

Verification of Concurrence for 4 - Qubit AKLT Model

Case 1: 0 <= J <= 0.6

In[ ]:= psi = $\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$

```
Out[ ]= {{0}, {0}, {0}, {0}, {0}, {0}, {0}, {0}, {0}, {0}, {0}, {0}, {0}, {0}, {0}, {1}}
```

```
In[ ]:= rho = psi.Transpose[psi]
```

```
Out[ ]= {{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, 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, 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, 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, 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, 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, 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, 1}}
```

```
In[ ]:= MatrixForm[rho]
Out[ ]//MatrixForm=
```

$$\begin{pmatrix} 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1 \end{pmatrix}$$

```
In[ ]:= rho12 = ResourceFunction["MatrixPartialTrace"][rho, {3, 4}, 2]
Out[ ]= {{0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 1}}

In[ ]:= MatrixForm[rho12]
Out[ ]//MatrixForm=
```

$$\begin{pmatrix} 0&0&0&0\\ 0&0&0&0\\ 0&0&0&0\\ 0&0&0&1 \end{pmatrix}$$

```
In[ ]:= SF =
```

$$\begin{pmatrix} 0&0&0&-1\\ 0&0&1&0\\ 0&1&0&0\\ -1&0&0&0 \end{pmatrix}$$

```
Out[ ]= {{0, 0, 0, -1}, {0, 0, 1, 0}, {0, 1, 0, 0}, {-1, 0, 0, 0}}

In[ ]:= rho =
```

$$\begin{pmatrix} 0.249999&-0.16079&-0.12862&0.082729\\ -0.16079&0.249999&0.082729&-0.12862\\ -0.12862&0.082729&0.249999&-0.16079\\ 0.082729&-0.12862&-0.16079&0.249999 \end{pmatrix}$$

```
Out[ ]= {{0.249999, -0.16079, -0.12862, 0.082729}, {-0.16079, 0.249999, 0.082729, -0.12862},
  {-0.12862, 0.082729, 0.249999, -0.16079}, {0.082729, -0.12862, -0.16079, 0.249999}}

In[ ]:= RRT = rho.SF.rho.SF
```

$$\text{Out[ ]= } \{\{0.0269471,\ 1.80385\times10^{-18},\ -2.32537\times10^{-18},\ 2.71494\times10^{-6}\},$$
$$\{-1.80385\times10^{-18},\ 0.0269471,\ 2.71494\times10^{-6},\ -8.46285\times10^{-19}\},$$
$$\{2.32537\times10^{-18},\ 2.71494\times10^{-6},\ 0.0269471,\ 2.62411\times10^{-18}\},$$
$$\{2.71494\times10^{-6},\ 8.46285\times10^{-19},\ -2.62411\times10^{-18},\ 0.0269471\}\}$$

In[•]:= Eigenvalues[RRT]

Out[•]= {0.0269498, 0.0269498, 0.0269443, 0.0269443}

In[•]:= ev = Sqrt[Eigenvalues[RRT]]

Out[•]= {0.164164, 0.164164, 0.164147, 0.164147}

In[•]:= con = 0

Out[•]= 0

Case 2: 0.61 <= J <= 1.20

In[•]:= psi = $\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -0.49999 \\ 0 \\ 0 \\ 0 \\ 0.49999 \\ 0 \\ -0.49999 \\ 0.5 \\ 0 \end{pmatrix}$

Out[•]= {{0}, {0}, {0}, {0}, {0}, {0}, {0}, {-0.49999}, {0}, {0}, {0}, {0.49999}, {0}, {-0.49999}, {0.5}, {0}}

```
In[•]:= rho = psi.Transpose[psi]

Out[•]= {{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., 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., 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., 0.},
  {0., 0., 0., 0., 0., 0., 0., 0.24999, 0., 0., 0., -0.24999, 0., 0.24999, -0.249995, 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., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.},
  {0., 0., 0., 0., 0., 0., 0., -0.24999, 0., 0., 0., 0.24999, 0., -0.24999, 0.249995, 0.},
  {0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.},
  {0., 0., 0., 0., 0., 0., 0., 0.24999, 0., 0., 0., -0.24999, 0., 0.24999, -0.249995, 0.},
  {0., 0., 0., 0., 0., 0., 0., -0.249995, 0., 0., 0., 0.249995, 0., -0.249995, 0.25, 0.},
  {0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.}}

In[•]:= MatrixForm[rho]

Out[•]//MatrixForm=
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. 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. 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.       0.
0. 0. 0. 0. 0. 0. 0.  0.24999   0. 0. 0. -0.24999   0.  0.24999  -0.249995   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. 0. 0. 0.    0.      0. 0. 0.    0.      0.    0.        0.       0.
0. 0. 0. 0. 0. 0. 0. -0.24999   0. 0. 0.  0.24999   0. -0.24999   0.249995   0.
0. 0. 0. 0. 0. 0. 0.    0.      0. 0. 0.    0.      0.    0.        0.       0.
0. 0. 0. 0. 0. 0. 0.  0.24999   0. 0. 0. -0.24999   0.  0.24999  -0.249995   0.
0. 0. 0. 0. 0. 0. 0. -0.249995 0. 0. 0.  0.249995  0. -0.249995    0.25     0.
0. 0. 0. 0. 0. 0. 0.    0.      0. 0. 0.    0.      0.    0.        0.       0.

In[•]:= rho12 = ResourceFunction["MatrixPartialTrace"][rho, {3, 4}, 2]

Out[•]= {{0., 0., 0., 0.}, {0., 0.24999, -0.24999, 0.},
  {0., -0.24999, 0.24999, 0.}, {0., 0., 0., 0.49999}}

In[•]:= MatrixForm[rho12]

Out[•]//MatrixForm=
0.     0.        0.       0.
0.  0.24999  -0.24999     0.
0. -0.24999   0.24999     0.
0.     0.        0.    0.49999
```

```
In[•]:= RSF = SF.rho12.SF
Out[•]= {{0.49999, 0., 0., 0.}, {0., 0.24999, -0.24999, 0.},
         {0., -0.24999, 0.24999, 0.}, {0., 0., 0., 0.}}

In[•]:= RRT = rho12.RSF
Out[•]= {{0., 0., 0., 0.}, {0., 0.12499, -0.12499, 0.}, {0., -0.12499, 0.12499, 0.}, {0., 0., 0., 0.}}

In[•]:= MatrixForm[RRT]
Out[•]//MatrixForm=
        ⎛ 0.     0.        0.     0. ⎞
        ⎜ 0.  0.12499  -0.12499  0. ⎟
        ⎜ 0. -0.12499   0.12499  0. ⎟
        ⎝ 0.     0.        0.     0. ⎠

In[•]:= Eigenvalues[RRT]
Out[•]= {0.24998, 0., 0., 0.}

In[•]:= ev = Sqrt[Eigenvalues[RRT]]
Out[•]= {0.49998, 0., 0., 0.}

In[•]:= con = 0.4999800002
Out[•]= 0.49998
```

Case 3: 1.21 <= J <= 4

```
In[•]:= psi = ⎛     0     ⎞
              ⎜     0     ⎟
              ⎜     0     ⎟
              ⎜ -0.288675 ⎟
              ⎜     0     ⎟
              ⎜  0.57735  ⎟
              ⎜ -0.288675 ⎟
              ⎜     0     ⎟
              ⎜     0     ⎟
              ⎜ -0.288675 ⎟
              ⎜  0.57735  ⎟
              ⎜     0     ⎟
              ⎜ -0.288675 ⎟
              ⎜     0     ⎟
              ⎜     0     ⎟
              ⎝     0     ⎠

Out[•]= {{0}, {0}, {0}, {-0.288675}, {0}, {0.57735}, {-0.288675},
         {0}, {0}, {-0.288675}, {0.57735}, {0}, {-0.288675}, {0}, {0}, {0}}
```

In[•]:= **rho = psi.Transpose[psi]**

Out[•]= {{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., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.},
{0., 0., 0., 0.0833333, 0., -0.166667, 0.0833333, 0., 0., 0.0833333, -0.166667, 0.,
0.0833333, 0., 0., 0.}, {0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.},
{0., 0., 0., -0.166667, 0., 0.333333, -0.166667, 0., 0., -0.166667, 0.333333,
0., -0.166667, 0., 0., 0.}, {0., 0., 0., 0.0833333, 0., -0.166667,
0.0833333, 0., 0., 0.0833333, -0.166667, 0., 0.0833333, 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., 0., 0.0833333, 0., -0.166667, 0.0833333, 0., 0., 0.0833333,
-0.166667, 0., 0.0833333, 0., 0., 0.}, {0., 0., 0., -0.166667, 0., 0.333333,
-0.166667, 0., 0., -0.166667, 0.333333, 0., -0.166667, 0., 0., 0.},
{0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.},
{0., 0., 0., 0.0833333, 0., -0.166667, 0.0833333, 0., 0., 0.0833333, -0.166667, 0.,
0.0833333, 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., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.}}

In[•]:= **rho12 = ResourceFunction["MatrixPartialTrace"][rho, {3, 4}, 2]**

Out[•]= {{0.0833333, 0., 0., 0.}, {0., 0.416666, -0.333333, 0.},
{0., -0.333333, 0.416666, 0.}, {0., 0., 0., 0.0833333}}

In[•]:= **RRT = rho12.SF.rho12.SF**

Out[•]= {{0.00694443, 0., 0., 0.}, {0., 0.284722, -0.277777, 0.},
{0., -0.277777, 0.284722, 0.}, {0., 0., 0., 0.00694443}}

In[•]:= **Eigenvalues[RRT]**

Out[•]= {0.562499, 0.00694443, 0.00694443, 0.00694443}

In[•]:= **ev = Sqrt[Eigenvalues[RRT]]**

Out[•]= {0.749999, 0.0833333, 0.0833333, 0.0833333}

In[•]:= **con = 0.7499993006250001 - 0.083333255625 - 0.083333255625 - 0.08333325562499982**

Out[•]= 0.5

This is the analytical verification of concurrence using Mathematica for the 4-qubit AKLT Model, which exactly matches our results. Now, let us see the analytical study for the ground state of our AKLT Model, how it changes with interaction energy (J) and due to this, how the concurrence is changing, and also the transitions are because of the change in ground state wave function.

Ground State for 4-qubit AKLT Model:

- **For** $0 \leqslant J \leqslant 0.6$

$$|\psi\rangle_{gs} = |1111\rangle$$

- **For** $0.61 \leqslant J \leqslant 1.20$

$$\begin{aligned}
|\psi\rangle_{gs} &= \frac{|1110\rangle - |1101\rangle + |1011\rangle - |0111\rangle}{2} \\
&= \frac{|11\rangle_{12}}{\sqrt{2}} \otimes \frac{(|10\rangle - |01\rangle)_{34}}{\sqrt{2}} + \frac{(|10\rangle - |01\rangle)_{12}}{\sqrt{2}} \otimes \frac{|11\rangle_{34}}{\sqrt{2}} \\
&= -\frac{|11\rangle_{12}}{\sqrt{2}} \otimes S_{34} - S_{12} \otimes \frac{|11\rangle_{34}}{\sqrt{2}}
\end{aligned}$$

- **For** $1.21 \leqslant J \leqslant 4$

$$\begin{aligned}
|\psi\rangle_{gs} &= \frac{|1010\rangle + |0101\rangle}{\sqrt{3}} - \frac{|0011\rangle + |0110\rangle + |1001\rangle + |1100\rangle}{2\sqrt{3}} \\
&= \frac{|1010\rangle + |0101\rangle - |0011\rangle - |0110\rangle - |1001\rangle - |1100\rangle + |1010\rangle + |0101\rangle}{2\sqrt{3}} \\
&= \frac{(|01\rangle - |10\rangle)_{12} \otimes (|01\rangle - |10\rangle)_{34} + (I + \sigma_x^{\otimes 4})|1\rangle_1 \otimes (|01\rangle - |10\rangle)_{23} \otimes |0\rangle_4}{\sqrt{2}\sqrt{2}\sqrt{3}} \\
&= \frac{S_{12} \otimes S_{34}}{\sqrt{3}} + \frac{(I + \sigma_x^{\otimes 4})|1\rangle_1 \otimes S_{23} \otimes |0\rangle_4}{\sqrt{6}}
\end{aligned}$$

Similarly, we can group the terms nicely as above, and one can observe the same pattern in the ground state wave function for a larger number of qubits.

Ground State for 6-qubit AKLT Model:

- **For** $0 \leqslant J \leqslant 0.6$

$$|\psi\rangle_{gs} = |111111\rangle$$

- **For** $0.61 \leqslant J \leqslant 0.74$

$$|\psi\rangle_{gs} = \frac{|111110\rangle - |111101\rangle + |111011\rangle - |110111\rangle + |101111\rangle - |011111\rangle}{\sqrt{6}}$$

- **For** $0.75 \leqslant J \leqslant 1.75$

$$\begin{aligned}|\psi\rangle_{gs} = -0.112837(&|001111\rangle + |011110\rangle + |100111\rangle + |110011\rangle + \\ &|111001\rangle + |111100\rangle) \\ +0.295411(&|010111\rangle + |011101\rangle + |101011\rangle + |101110\rangle + \\ &|110101\rangle + |111010\rangle) \\ -0.365148(&|011011\rangle + |101101\rangle + |110110\rangle)\end{aligned}$$

- **For** $1.76 \leqslant J \leqslant 4$

$$\begin{aligned}|\psi\rangle_{gs} =&0.2078(|001011\rangle - |001101\rangle - |010011\rangle - |010110\rangle - \\ &|011001\rangle + |011010\rangle - |100101\rangle + |100110\rangle + \\ &|101001\rangle + |101110\rangle + |110010\rangle - |110100\rangle) \\ &+ 0.06292(|001110\rangle - |000111\rangle + |100011\rangle - |011100\rangle + \\ &|111000\rangle - |110001\rangle) + 0.478546(|010101\rangle - |101010\rangle)\end{aligned}$$

We can also write the ground state wave function for the higher qubit AKLT Model, but that is not of too much use because the nature of the state is the same, just the normalisation constants are different.

Now let us see the variation of concurrence with magnetic field (B) for a constant interaction energy ($J = 1$) in the AKLT model.

```
program spin1qubit
    implicit none
    integer, parameter :: d =2, s = 4, s1 = 2
    integer, parameter :: num = d**s, N = num, di = d**s1
```

```
    integer :: i, j, k, l, t, flagx, flagy, pos, u
    integer, allocatable :: digits1(:), digits2(:), decimal(:), site
   (:)
    character(len=:), allocatable :: state(:)
    double precision, allocatable :: E(:,:), C(:,:), F(:,:), G(:,:),
    H(:,:), Z(:,:)
    character*1 :: JOBZ, UPLO, RANGE
    integer ::  LDA = num, LWORK = 8*num, INFO, IL, IU, M, LDZ,
   IWORK(5*N), IFAIL(N)
    double precision :: W(num), WORK(8*num), VL, VU, ABSTOL
    real*8 :: sum, trace, con, Jo, B
    real*8, allocatable :: psi(:,:), rho(:,:), SF(:,:), RT(:,:), RRT
   (:,:)
    real*8, allocatable :: rdm(:,:), RW(:), RWORK(:) !,red_rho(:,:)
    integer :: RLDA = di, RINFO, RLWORK = 3*di -1

    ! Allocate arrays after determining the value of num
    allocate(character(len=s) :: state(num))
    allocate(integer :: decimal(num))
    allocate(integer :: digits1(s), digits2(s))
    allocate(real(8) :: C(num, num))
    allocate(real(8) :: E(num, num))
    allocate(real(8) :: F(num, num))
    allocate(real(8) :: G(num, num))
    allocate(real(8) :: H(0:num-1 , 0:num-1))
    allocate(real(8) :: Z(0:num-1 , 0:num-1))
    allocate(psi(0:num-1,0:0))
    allocate(rho(0:num-1, 0:num-1))
    allocate(rdm(di, di))
    ! allocate(red_rho(0:di-1,0:di-1))
    allocate(site(0:s1-1))
    allocate(SF(di,di))
    allocate(RT(di,di))
    allocate(RRT(di,di))
    allocate(RW(di))
    allocate(RWORK(RLWORK))

    do i = 0, num - 1
```

```
39          call integer_binary(i, state(i+1), s)  ! Adjusted for 1-
    based indexing
40          !print *, state(i)
41       end do
42       do i = 1, num
43          decimal(i) = btod(state(i))
44          !print*, ' Decimal equivalent of ', state(i) ,' = ', decimal
    (i)
45       end do
46
47       G = 0.0d0
48       do k =1 , s
49          E = 0.0d0
50          do i = 1, num
51              do j = 1, num
52                  ! Computing Sigma_i^x * Sigma_i+1^x for each site
53                  flagx = 0
54                  if (decimal(i) .ne. decimal(j)) then
55                      call sd(state(i), digits1)
56                      call sd(state(j), digits2)
57                      if (k < s) then
58                          if (digits1(k) .ne. digits2(k) .and. digits1
    (k+1) .ne. digits2(k+1)) then
59                              do l = 1, k - 1
60                                  if (digits1(l) .ne. digits2(l)) then
61                                      flagx  = flagx + 1
62                                  end if
63                              end do
64                              do l = k + 2, s
65                                  if (digits1(l) .ne. digits2(l)) then
66                                      flagx  = flagx + 1
67                                  end if
68                              end do
69                              if (flagx == 0) then
70                                  E(i, j) = E(i, j) + 1.0
71                              else
72                                  E(i, j) = E(i, j) + 0.0
73                              end if
```

```
                        else
                            E(i, j) = E(i, j) + 0.0
                        end if
                    else
                        if (digits1(1) .ne. digits2(1) .and. digits1
    (s) .ne. digits2(s)) then
                            do l = 2, s - 1
                                if (digits1(l) .ne. digits2(l)) then
                                    flagx  = flagx + 1
                                end if
                            end do
                            if (flagx == 0) then
                                E(i, j) = E(i, j) + 1.0
                            else
                                E(i, j) = E(i, j) + 0.0
                            end if
                        else
                            E(i, j) = E(i, j) + 0.0
                        end if
                    end if
                else
                    E(i, j) = E(i, j) + 0.0
                end if

                ! Computing Sigma_i^y * Sigma_i+1^y for each site
                flagy = 0
                if (decimal(i) .ne. decimal(j)) then
                    call sd(state(i), digits1)
                    call sd(state(j), digits2)
                    if (k < s) then
                        if (digits1(k) .ne. digits2(k) .and. digits1
    (k+1) .ne. digits2(k+1)) then
                            do l = 1, k - 1
                                if (digits1(l) .ne. digits2(l)) then
                                    flagy  = flagy + 1
                                end if
                            end do
                            do l = k + 2, s
```

```
110                                 if (digits1(l) .ne. digits2(l)) then
111                                     flagy  = flagy + 1
112                                 end if
113                             end do
114                             if (flagy == 0) then
115                                 E(i, j) = E(i, j) + (-1.0 * (-1.0)
    **(digits2(k) + digits2(k+1)))
116                             else
117                                 E(i, j) = E(i, j) + 0.0
118                             end if
119                         else
120                             E(i, j) = E(i, j) + 0.0
121                         end if
122                     else
123                         if (digits1(1) .ne. digits2(1) .and. digits1
    (s) .ne. digits2(s)) then
124                             do l = 2, s - 1
125                                 if (digits1(l) .ne. digits2(l)) then
126                                     flagy  = flagy + 1
127                                 end if
128                             end do
129                             if (flagy == 0) then
130                                 E(i, j) = E(i, j)  + (-1.0 * (-1.0)
    **(digits2(s) + digits2(1)))
131                             else
132                                 E(i, j) = E(i, j) + 0.0
133                             end if
134                         else
135                             E(i, j) = E(i, j) + 0.0
136                         end if
137                     end if
138                 else
139                     E(i, j) = E(i, j) + 0.0
140                 end if
141
142                 ! Computing Sigma_i^z * Sigma_i+1^z for each site
143                 call sd(state(i), digits1)
144                 if (decimal(i) .ne. decimal(j)) then
```

```
145                     E(i, j) =  E(i,j) + 0.0
146                 else
147                     if (k < s) then
148                         E(i, j) = E(i, j) + (1 - 2 * digits1(k)) *
    (1 - 2 * digits1(k+1))
149                     else
150                         E(i, j) = E(i, j) + (1 - 2 * digits1(1)) *
    (1 - 2 * digits1(s))
151                     end if
152                 end if
153
154                 ! Computing Sigma_i^z for each site
155                 call sd(state(i), digits1)
156                 if (decimal(i) .ne. decimal(j)) then
157                     G(i, j) =  G(i,j) + 0.0
158                 else
159                     G(i, j) = G(i, j) + (1 - 2 * digits1(k))
160                 end if
161             end do
162         end do
163     C = C + (1.0d0/4.0d0)*E
164     F = F + matmul((1.0d0/4.0d0)*E,(1.0d0/4.0d0)*E)
165     end do
166     ! write(58,*) C
167
168     do k = 0, s-2
169         do l = k+1, s-1
170             site = (/k, l/)
171                 do t = 0, 200
172                     B =  0.01 * t
173                     Jo = 1.0d0
174                     H = 0.0d0
175                     do i = 1, num
176                         do j = 1, num
177                             H(i-1,j-1) = Jo * (C(i,j) + (1.0d0/3.0d0
    )* F(i,j)) + B * (1.0d0/2.0d0)* G(i,j)
178                             !if (i .le. j) then
179                                 ! write(57,*) i, j, H(i-1,j-1)
```

```
180                             !end if
181                         end do
182                     end do
183 
184                     ! Define parameters for DSYEVX
185                     JOBZ = 'V'  ! Compute eigenvalues and
    eigenvectors
186                     RANGE = 'I' ! Compute selective eigenvalues and
    eigenvectors
187                     UPLO = 'U'  ! Upper triangular part of A is
    stored
188                     ABSTOL = 0.0d0
189                     IL = 1
190                     IU = 1
191                     M = IU - IL + 1
192                     LDZ = N
193                     ! Call DSYEVX to compute ground state eigenvalue
     and eigen vector
194                     call DSYEVX (JOBZ, RANGE, UPLO, N, H, LDA, VL,
    VU, IL, IU, ABSTOL &
195                     , M, W, Z, LDZ, WORK, LWORK, IWORK, IFAIL, INFO)
196                     ! Check for successful execution
197                     ! sum = 0.0d0
198                     ! if (INFO == 0) then
199                     !     do i = 1, M
200                     !         write(3,*) B, W(i)
201                     !     end do
202                     ! else
203                     !     print*, 'Error in DSYEVX, info =', INFO
204                     ! end if
205 
206                     do i = 0, M-1
207                         do j = 0, num - 1
208                             if (abs(Z(j,i)) .le. 0.000000000001 .and
    . abs(Z(j,i)) .ge. 0) then
209                                 psi(j,i) = 0
210                             else
211                                 psi(j,i) = Z(j,i)
```

```
212                             end if
213                             ! write(7,*) j, i, psi(j,i)
214                         end do
215                     end do
216
217                     rho = matmul(psi, transpose(psi))
218                     ! do i = 0, num-1
219                     !     do j = 0, num-1
220                     !         if(i .eq. j) then
221                     !             trace = trace + rho(i,j)
222                     !         end if
223                     !         write(7,*) i, j, rho(i, j)
224                     !     end do
225                     ! end do
226                     ! print*, "Trace = ", trace
227
228                     call ptrace(rho, d, s, s1, site, rdm)
229                     ! trace = 0.0d0
230                     ! do i = 0, di-1
231                     !     do j = 0, di-1
232                     !         ! rdm(i+1,j+1) = red_rho(i,j)
233                     !         if (i .eq. j) then
234                     !             trace = trace + rdm(i+1,j+1)
235                     !         end if
236                     !         ! write(7,*) i, j, rdm(i+1, j+1)
237                     !     end do
238                     ! end do
239                     ! print*, "Trace = ", trace
240
241                     call spin_flip_matrix(d,SF)              !
    Generating Spin Flip Matrix
242                     RT = matmul(SF, matmul(rdm, SF))
243                     RRT = matmul(rdm, RT)
244
245                     ! Define parameters for DSYEVX
246                     JOBZ = 'N'  ! Compute eigenvalues only
247                     ! Call DSYEV to compute eigenvalues and
    eigenvectors
```

```
248                     call DSYEV(JOBZ, UPLO, di, RRT, RLDA, RW, RWORK,
    RLWORK, RINFO)
249                     sum = 0.0d0
250                     ! Check for successful execution
251                     if (RINFO == 0) then
252                         call sort_dec(RW, di)
253                         do i = 2, di
254                             sum = sum + sqrt(abs(RW(i)))
255                         end do
256                     else
257                         print*, 'Error in DSYEV, info =', RINFO
258                     end if
259                     con = max(0.0d0, (sqrt(abs(RW(1)))- sum))
260                     pos = k*10 + l
261                     write(400+pos,*) B, con
262                 ! end do
263             end do
264         end do
265     end do
266
267 contains
268
269     subroutine integer_binary(num, binary_string, n)
270         implicit none
271         integer :: num
272         integer, intent(in) :: n
273         integer :: i, temp_num
274         character(len=n), intent(out) :: binary_string
275         temp_num = num
276         ! Initialize binary_string to all zeros
277         binary_string = repeat('0', n)
278         ! Convert the integer to binary
279         do i = 0, n - 1
280             if (mod(temp_num, 2) == 1) then
281                 binary_string(n-i:n-i) = '1'  ! Set the bit to '1'
282             else
283                 binary_string(n-i:n-i) = '0'  ! Set the bit to '0'
284             end if
```

```
285             temp_num = temp_num / 2  ! Divide num by 2 to shift
    right
286         end do
287     end subroutine integer_binary
288
289     subroutine sd(binaryString, digits)
290         character(len=*), intent(in) :: binaryString
291         integer, allocatable, intent(out) :: digits(:)
292         integer :: i, len
293         len = len_trim(binaryString)  ! Get the length of the binary
     string
294         allocate(digits(len))          ! Allocate array to hold
    digits
295         ! Convert each character to an integer
296         do i = 1, len
297             if (binaryString(i:i) == '1') then
298                 digits(i) = 1
299             else if (binaryString(i:i) == '0') then
300                 digits(i) = 0
301             else
302                 print *, "Invalid character in binary string."
303                 digits = 0  ! Set to zero if invalid character is
    found
304                 return
305             end if
306         end do
307     end subroutine sd
308
309     function btod(binaryString) result(decimalValue)
310         implicit none
311         character(len=*), intent(in) :: binaryString
312         integer :: decimalValue
313         integer :: i, length
314
315         decimalValue = 0
316         length = len_trim(binaryString)  ! Get the length of the
    binary string
317
```

```
318         ! Convert binary string to decimal
319         do i = 1, length
320             if (binaryString(i:i) == '1') then
321                 decimalValue = decimalValue + 2**(length - i)
322             else if (binaryString(i:i) /= '0') then
323                 print *, "Invalid character in binary string."
324                 decimalValue = -1  ! Indicate an error with -1
325                 return
326             end if
327         end do
328     end function btod
329 
330     subroutine ptrace(rho, d, s, s1, site, rdm)
331         implicit none
332         integer, intent(in) :: d, s, s1
333         integer, intent(in) :: site(0:s1-1)
334         real*8, intent(in) :: rho(0:d**s-1, 0:d**s-1)
335         real*8, intent(out) :: rdm(d**s1, d**s1)
336         integer, allocatable :: idits(:), jdits(:), ridits(:),
    rjdits(:), to(:)
337         integer :: i, j, k, p, flag, ri, rj
338         real*8 :: a
339 
340         allocate(idits(0:s-1), jdits(0:s-1))
341         allocate(ridits(0:s1-1), rjdits(0:s1-1))
342         allocate(to(0:s-s1-1))
343 
344         ! Compute the indices to trace out
345         call TOI(s, site, s1, to)
346 
347         rdm = 0.0d0
348         do i = 0, d**s - 1
349             do j = 0, d**s - 1
350                 if (rho(i,j) .ne. 0.0d0) then
351                     a = rho(i,j)
352                     call DECTOD(i, d, s, idits)
353                     call DECTOD(j, d, s, jdits)
354                     flag = 0
```

```
355                     do k = 0, s - s1 - 1
356                         if (idits(to(k)) .ne. jdits(to(k))) then
357                             flag = 1
358                             exit
359                         end if
360                     end do
361                     if (flag .eq. 0) then
362                         do k = 0, s1 - 1
363                             p = site(k)
364                             ridits(k) = idits(p)
365                             rjdits(k) = jdits(p)
366                         end do
367                         call DTODEC(d, s1, ridits, ri)
368                         call DTODEC(d, s1, rjdits, rj)
369                         rdm(ri+1, rj+1) = rdm(ri+1, rj+1) + a
370                     end if
371                 end if
372             end do
373         end do
374 
375         deallocate(idits, jdits, ridits, rjdits, to)
376     end subroutine ptrace
377 
378     subroutine DECTOD(dec, d, s, dits)
379         implicit none
380         integer, intent(in) :: dec, d, s
381         integer, allocatable, intent(out) :: dits(:)
382         integer :: i, temp_dec
383 
384         allocate(dits(0:s-1))
385 
386         temp_dec = dec
387         do i = s-1, 0, -1
388             dits(i) = mod(temp_dec, d)
389             temp_dec = temp_dec / d
390         end do
391     end subroutine DECTOD
392 
```

```
393     subroutine DTODEC(d, s, dits, dec)
394         implicit none
395         integer, intent(in) :: d, s
396         integer, dimension(0:s-1), intent(in) :: dits
397         integer, intent(out) :: dec
398         integer :: i
399 
400         dec = 0
401         do i = 0, s - 1
402             dec = dec * d + dits(i)
403         end do
404     end subroutine DTODEC
405 
406     subroutine TOI(s, site, s1, to)
407         implicit none
408         integer, intent(in) :: s, s1
409         integer, intent(in) :: site(0:s1-1)
410         integer, intent(out) :: to(0:s-s1-1)
411         integer :: i, j, k, found
412 
413         k = 0
414         do i = 0, s-1
415             found = 0
416             do j = 0, s1 - 1
417                 if (i == site(j)) then
418                     found = 1
419                     exit
420                 end if
421             end do
422             if (found == 0) then
423                 to(k) = i
424                 k = k + 1
425             end if
426         end do
427     end subroutine TOI
428 
429     subroutine spin_flip_matrix(n, SFM)
430         implicit none
```

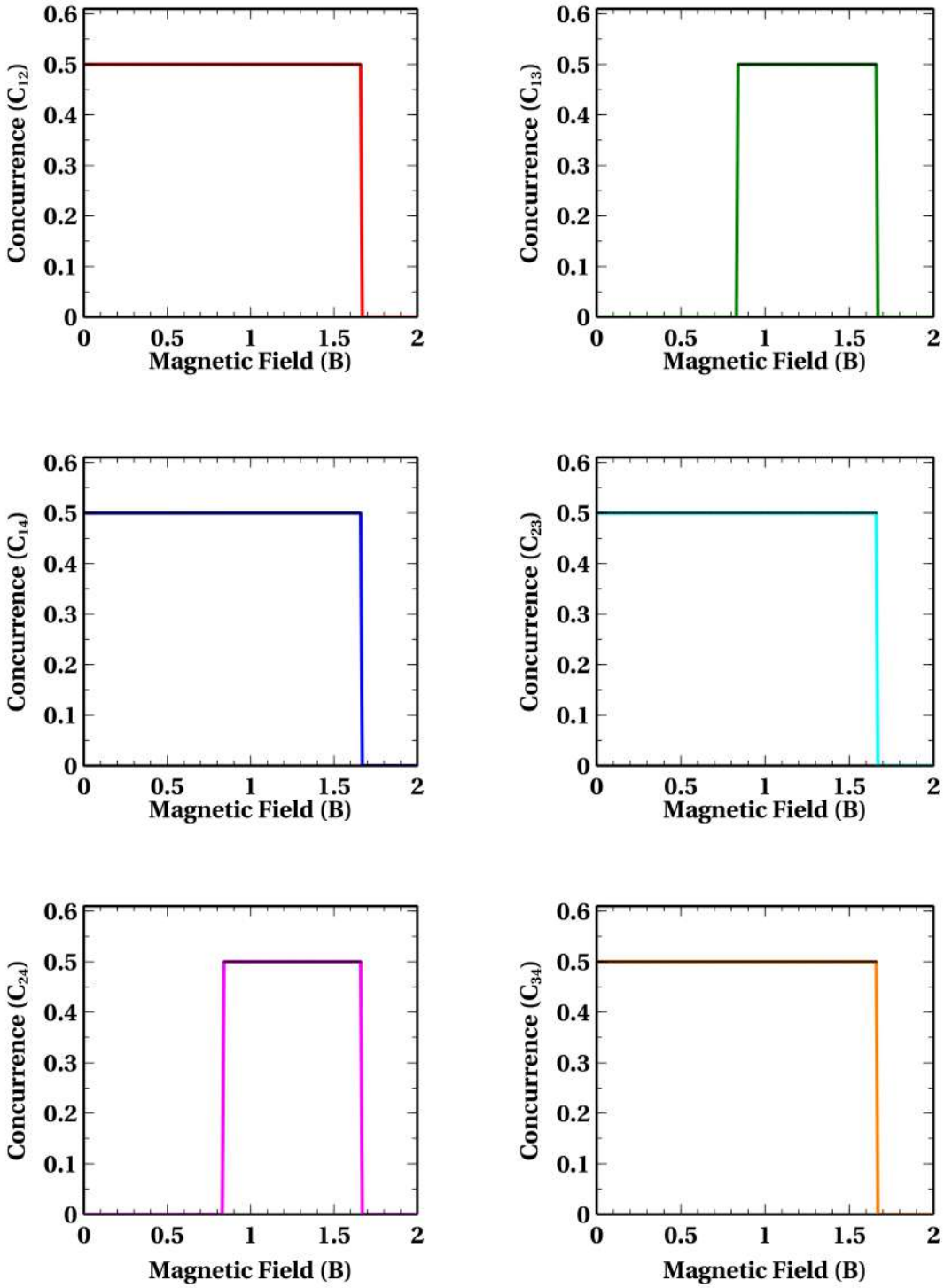
```
        integer, intent(in) :: n
        real*8, intent(out) :: SFM(n**2, n**2)  ! Output matrix
        integer :: size, i, j
        ! Calculate the size of the matrix (n^2)
        size = n**2
        ! Initialize the matrix with zeros
        SFM = 0.0d0
        ! Fill the matrix based on the alternating pattern
        do i = 1, size
            if (mod(i - 1, n + 1) == 0) then
                SFM(i, size - i + 1) = -1
            else
                SFM(i, size - i + 1) = 1
            end if
        end do
    end subroutine spin_flip_matrix

    subroutine sort_dec(arr, n)
        implicit none
        integer, intent(in) :: n
        real*8, intent(inout) :: arr(n)
        integer :: i, j
        real*8 :: temp
        ! Insertion sort (Descending Order)
        do i = 2, n
            temp = arr(i)
            j = i - 1
            do while (j > 0 .and. arr(j) < temp)
                arr(j + 1) = arr(j)
                j = j - 1
            end do
            arr(j + 1) = temp
        end do
    end subroutine sort_dec
end program
```

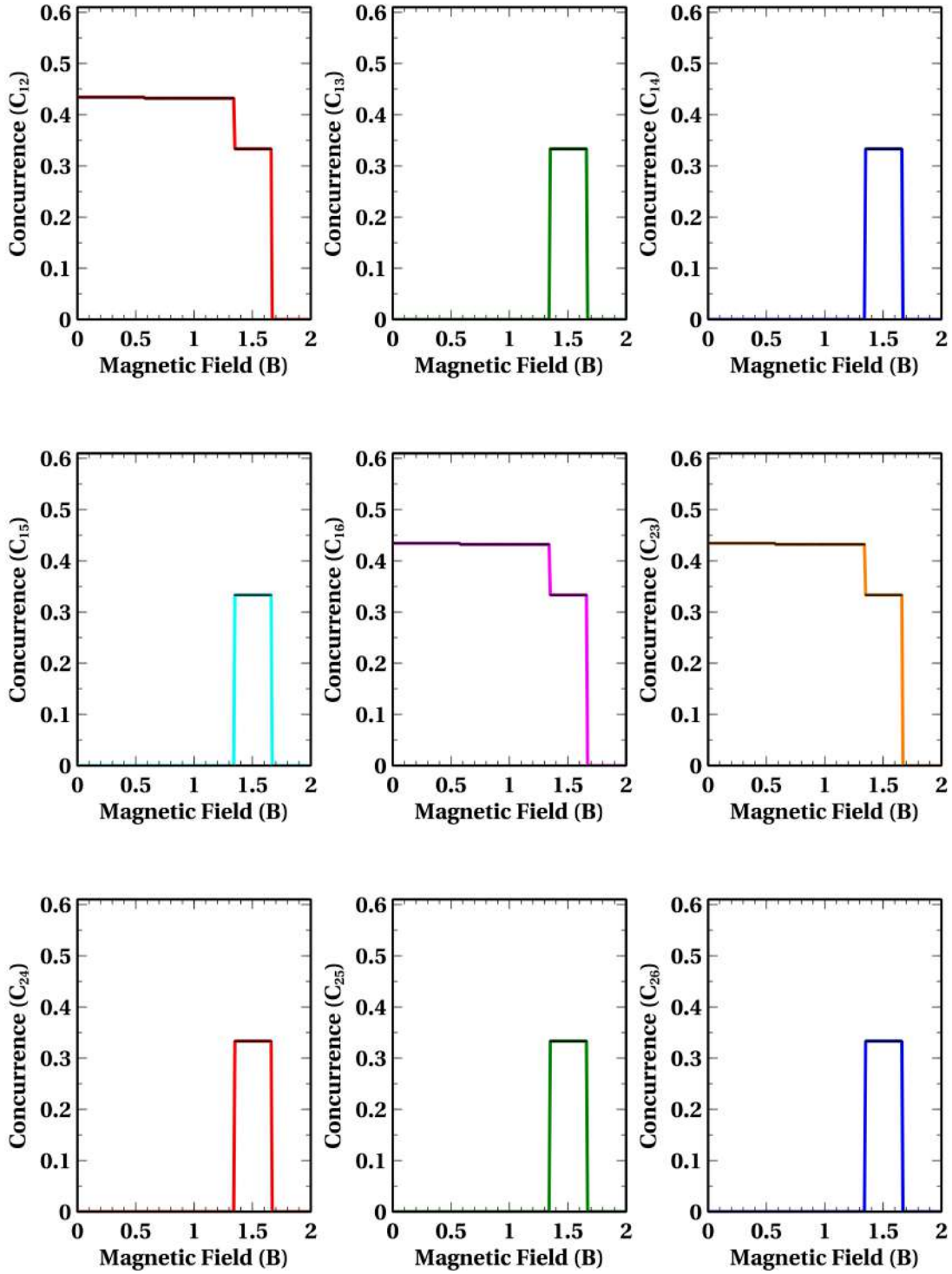
This is the general program to calculate the concurrence for all the combinations of

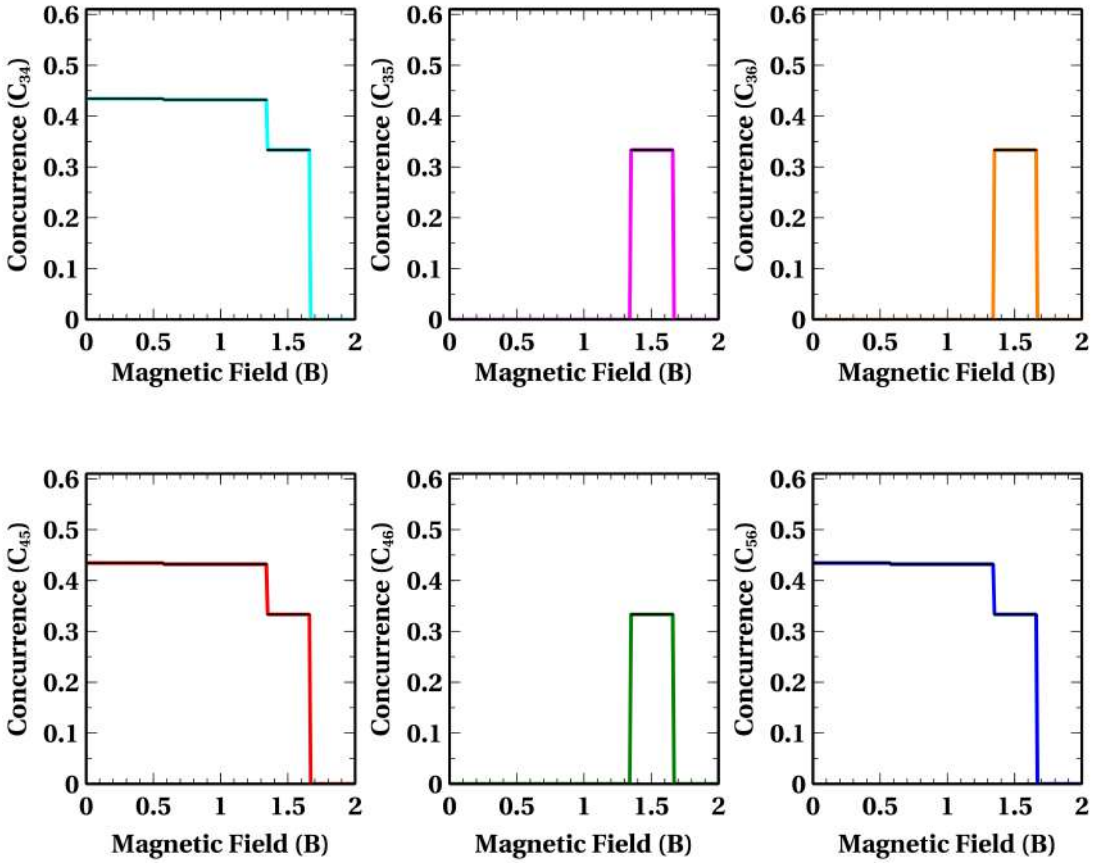
qubits by varying the magnetic field (B) and keeping the interaction energy constant as ($J = 1$). Let us see the concurrence plots for the different combinations of qubits.

4.2.5 Concurrence vs Magnetic Field (B) for a 4 qubit AKLT Model

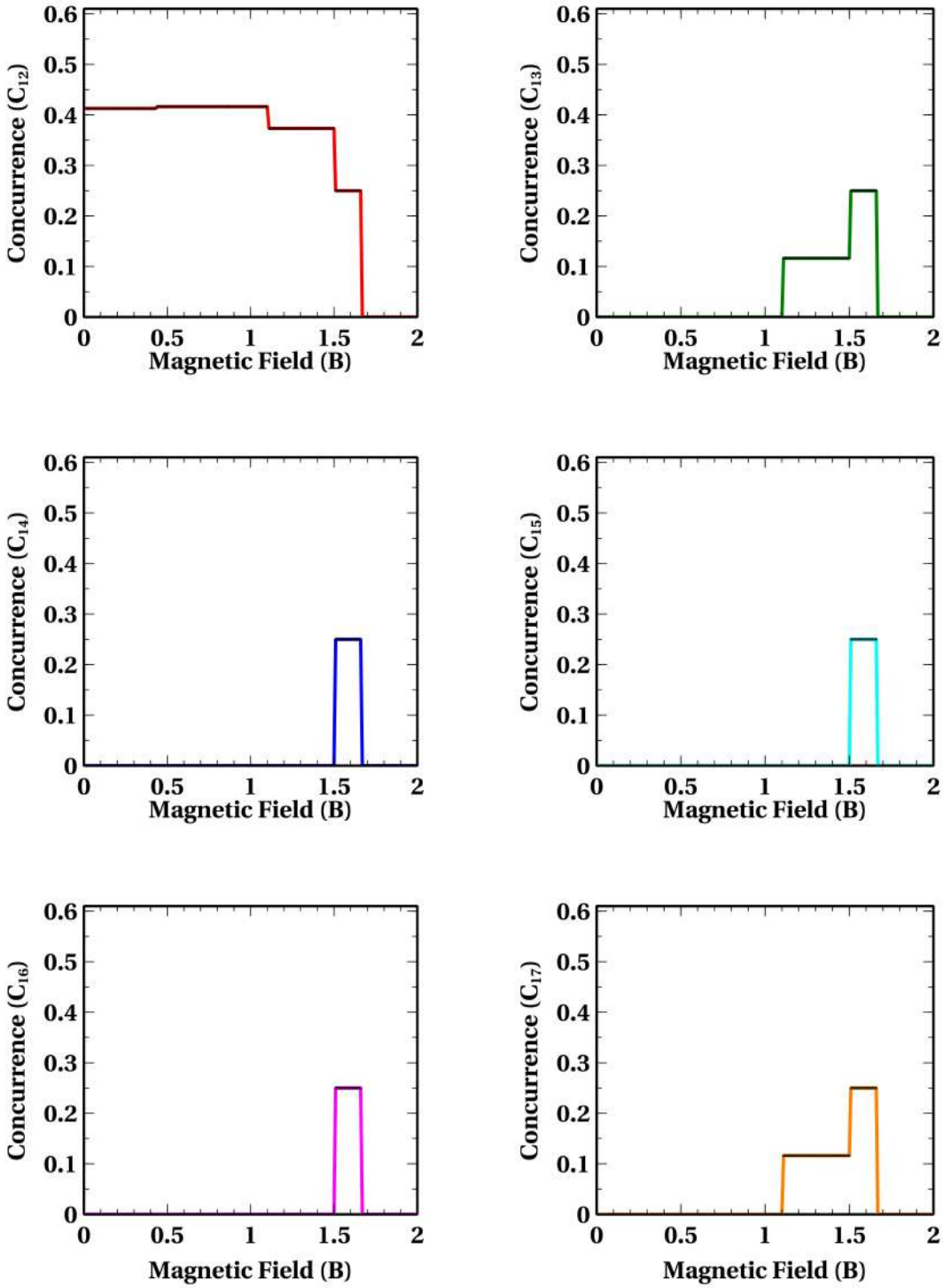


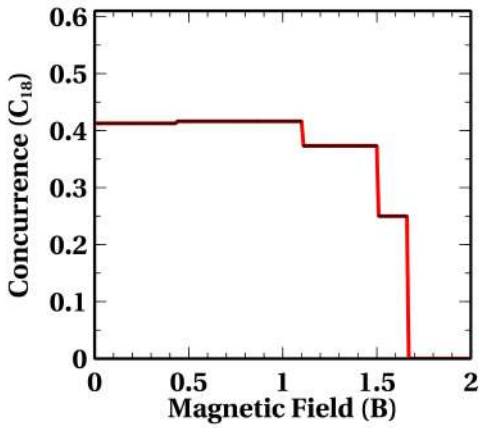
4.2.6 Concurrence vs Magnetic Field (B) for a 6 qubit AKLT Model

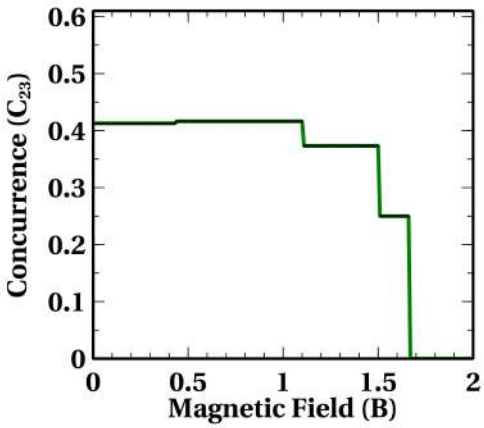


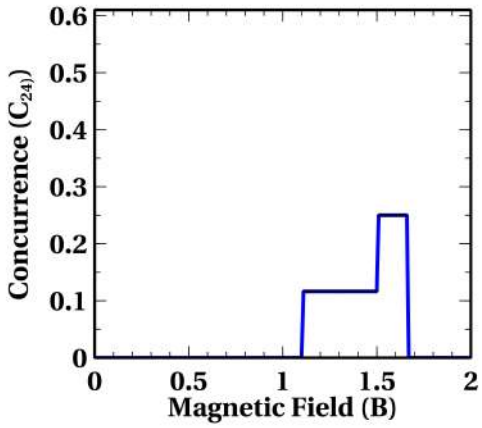

Concurrence (C_{34})
Concurrence (C_{35})
Concurrence (C_{36})
Concurrence (C_{45})
Concurrence (C_{46})
Concurrence (C_{56})
Magnetic Field (B)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.5
1
1.5
2

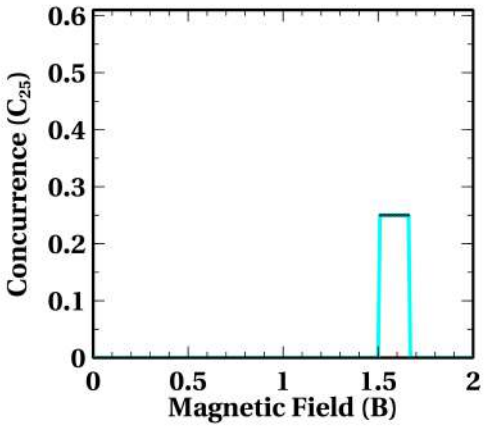
4.2.7 Concurrence vs Magnetic Field (B) for a 8 qubit AKLT Model

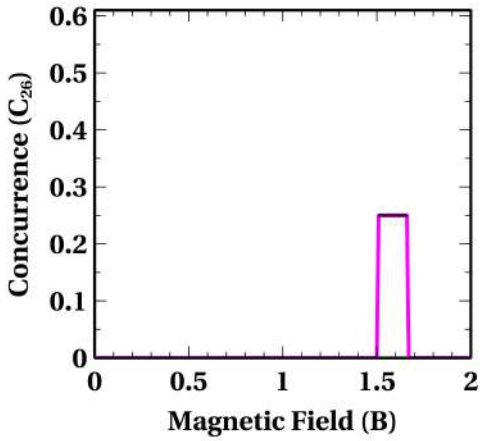


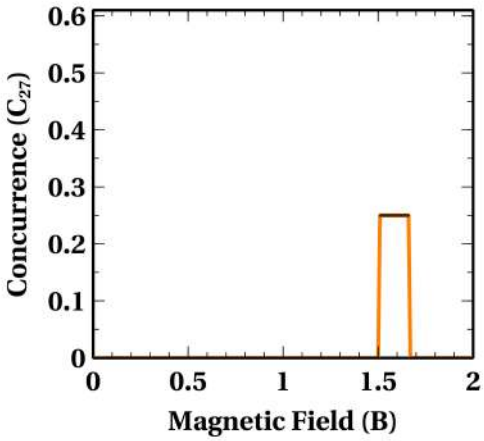

Concurrence (C_{18})
Magnetic Field (B)

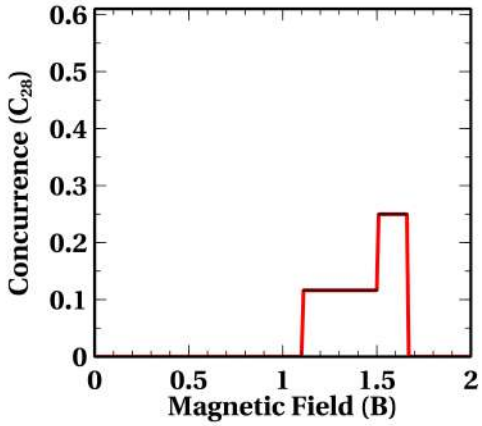

Concurrence (C_{23})
Magnetic Field (B)

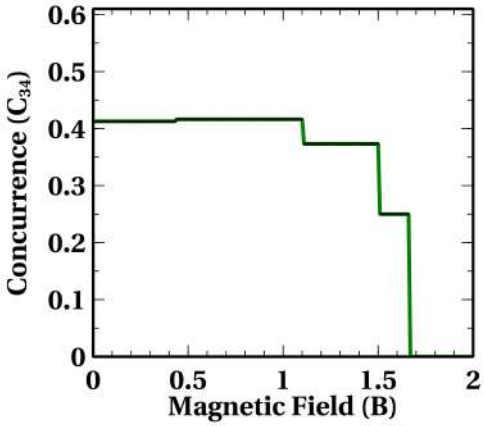

Concurrence (C_{24})
Magnetic Field (B)

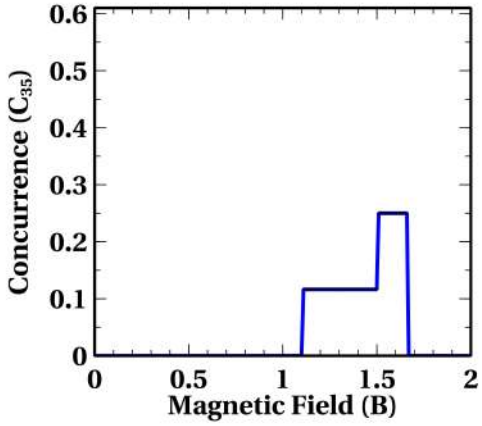

Concurrence (C_{25})
Magnetic Field (B)

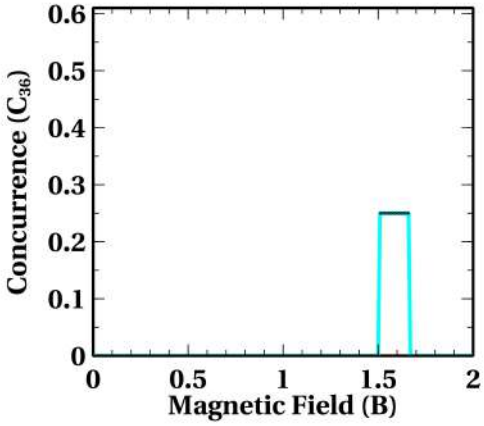

Concurrence (C_{26})
Magnetic Field (B)

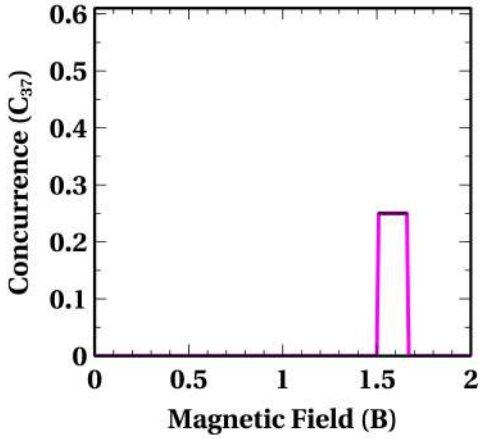

Concurrence (C_{27})
Magnetic Field (B)

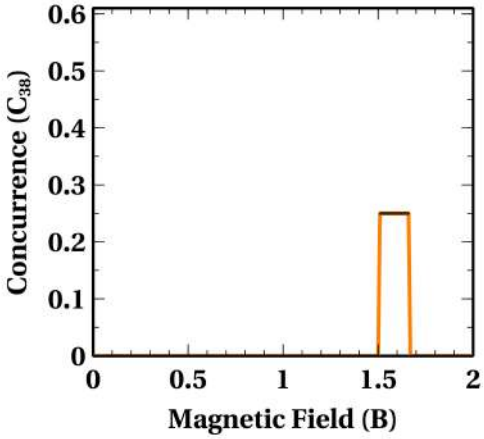

Concurrence (C_{28})
Magnetic Field (B)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.5
1
1.5
2

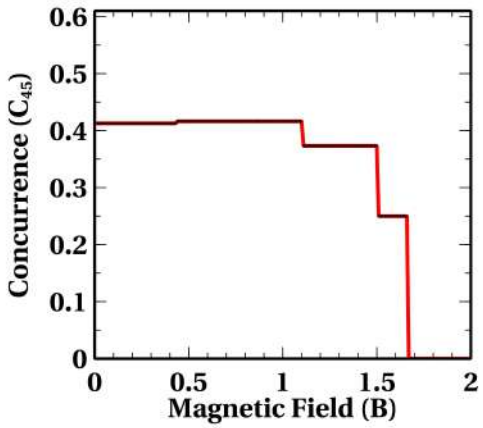

Concurrence (C_{34})
Magnetic Field (B)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.5
1
1.5
2

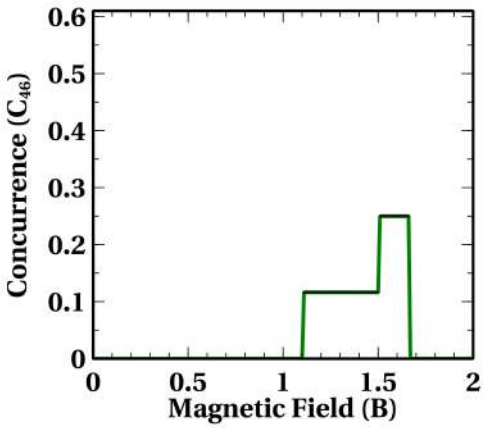

Concurrence (C_{35})
Magnetic Field (B)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.5
1
1.5
2

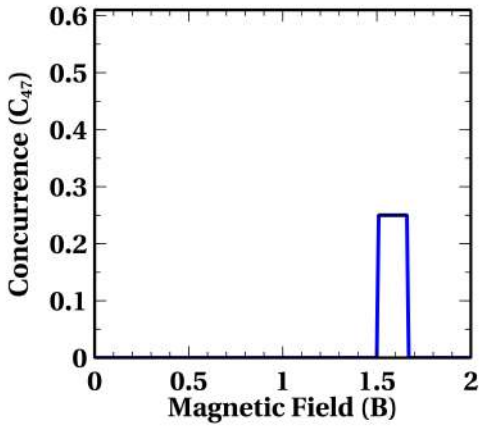

Concurrence (C_{36})
Magnetic Field (B)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.5
1
1.5
2

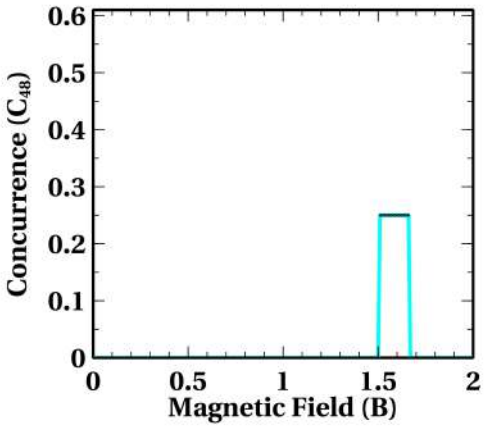

Concurrence (C_{37})
Magnetic Field (B)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.5
1
1.5
2

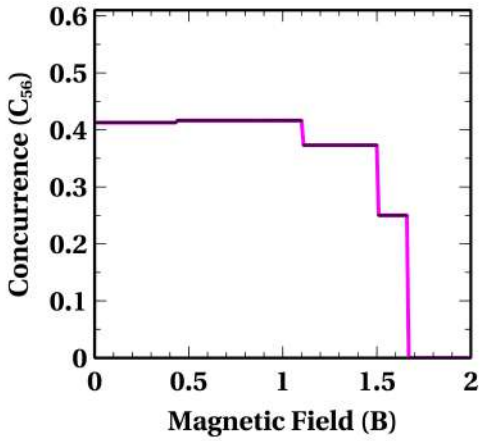

Concurrence (C_{38})
Magnetic Field (B)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.5
1
1.5
2

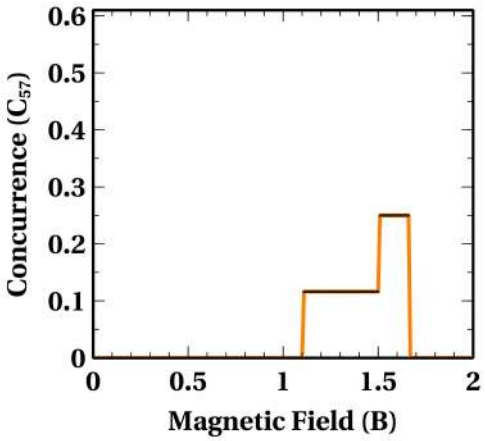

Concurrence (C_{45})
Magnetic Field (B)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.5
1
1.5
2

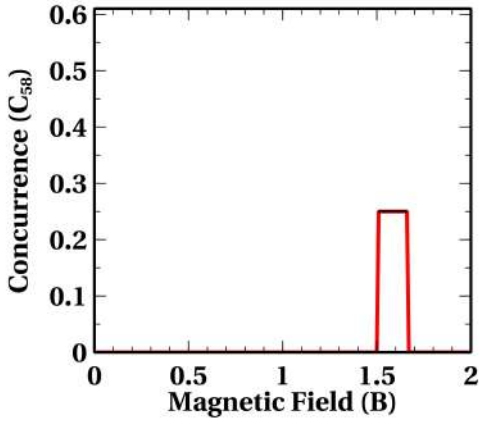

Concurrence (C_{46})
Magnetic Field (B)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.5
1
1.5
2

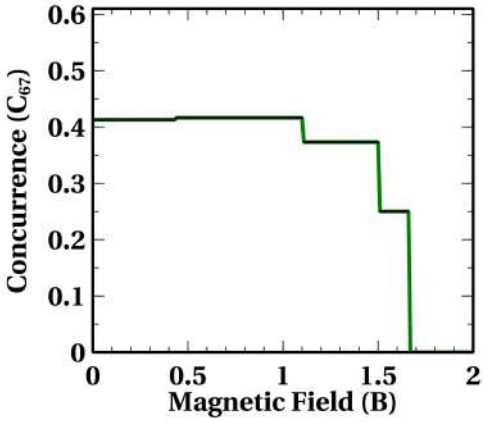

Concurrence (C_{47})
Magnetic Field (B)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.5
1
1.5
2

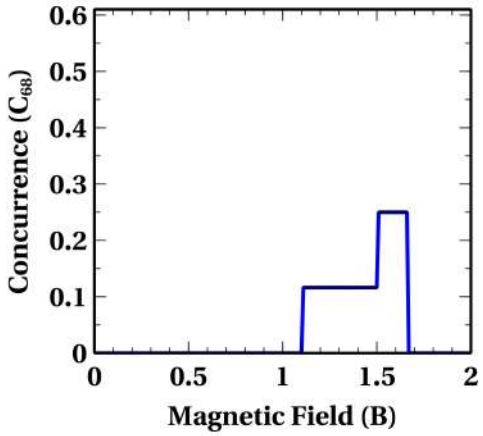

Concurrence (C_{48})
Magnetic Field (B)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.5
1
1.5
2

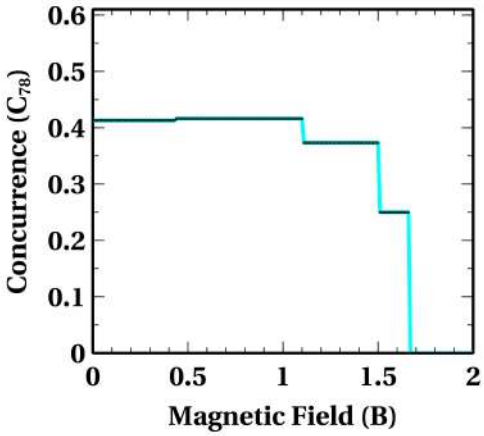

Concurrence (C_{56})
Magnetic Field (B)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.5
1
1.5
2


Concurrence (C_{57})
Magnetic Field (B)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.5
1
1.5
2

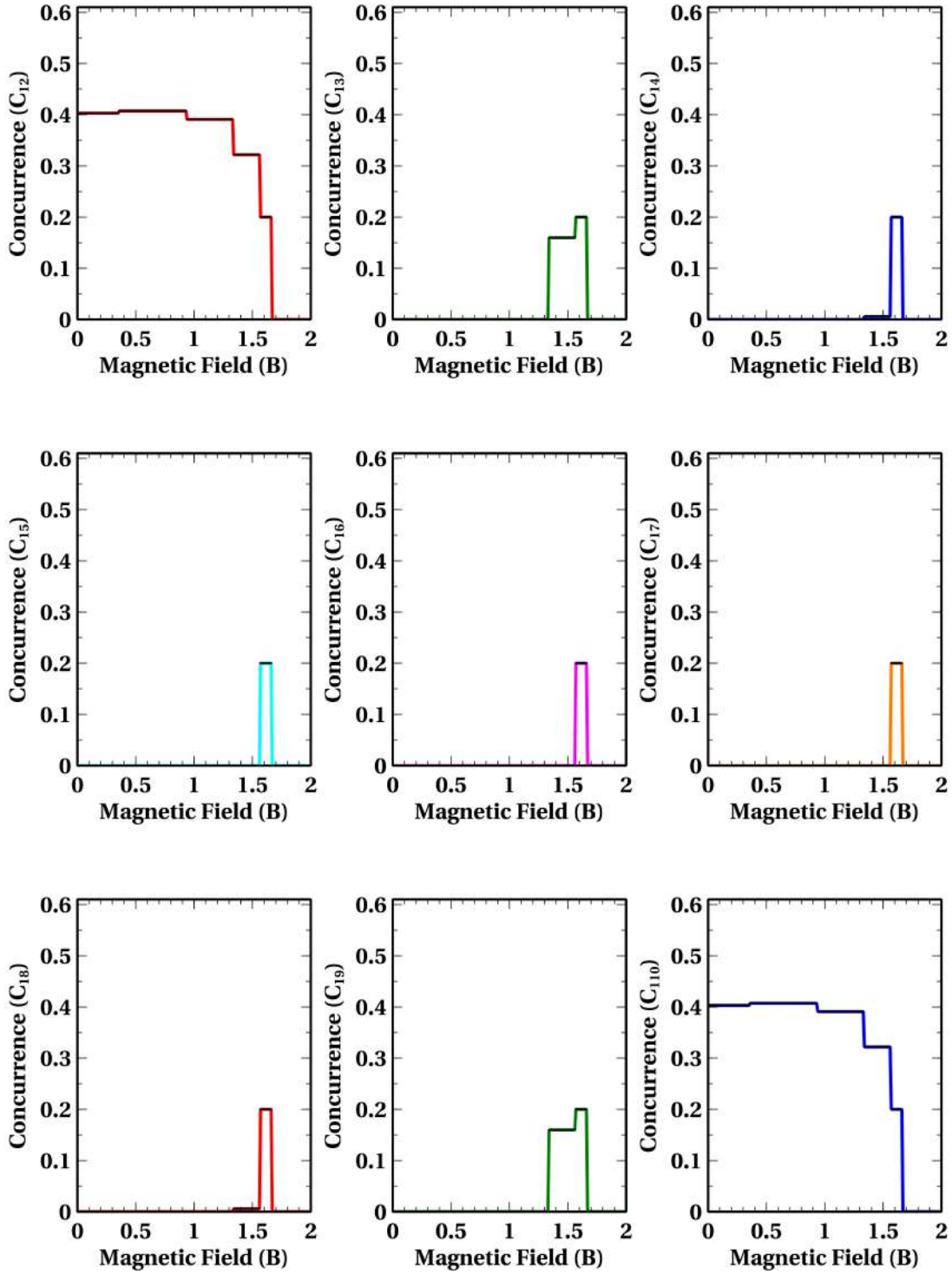

Concurrence (C_{58})
Magnetic Field (B)
0
0.1
0.2
0.3
0.4
0.5
0.6
0
0.5
1
1.5
2

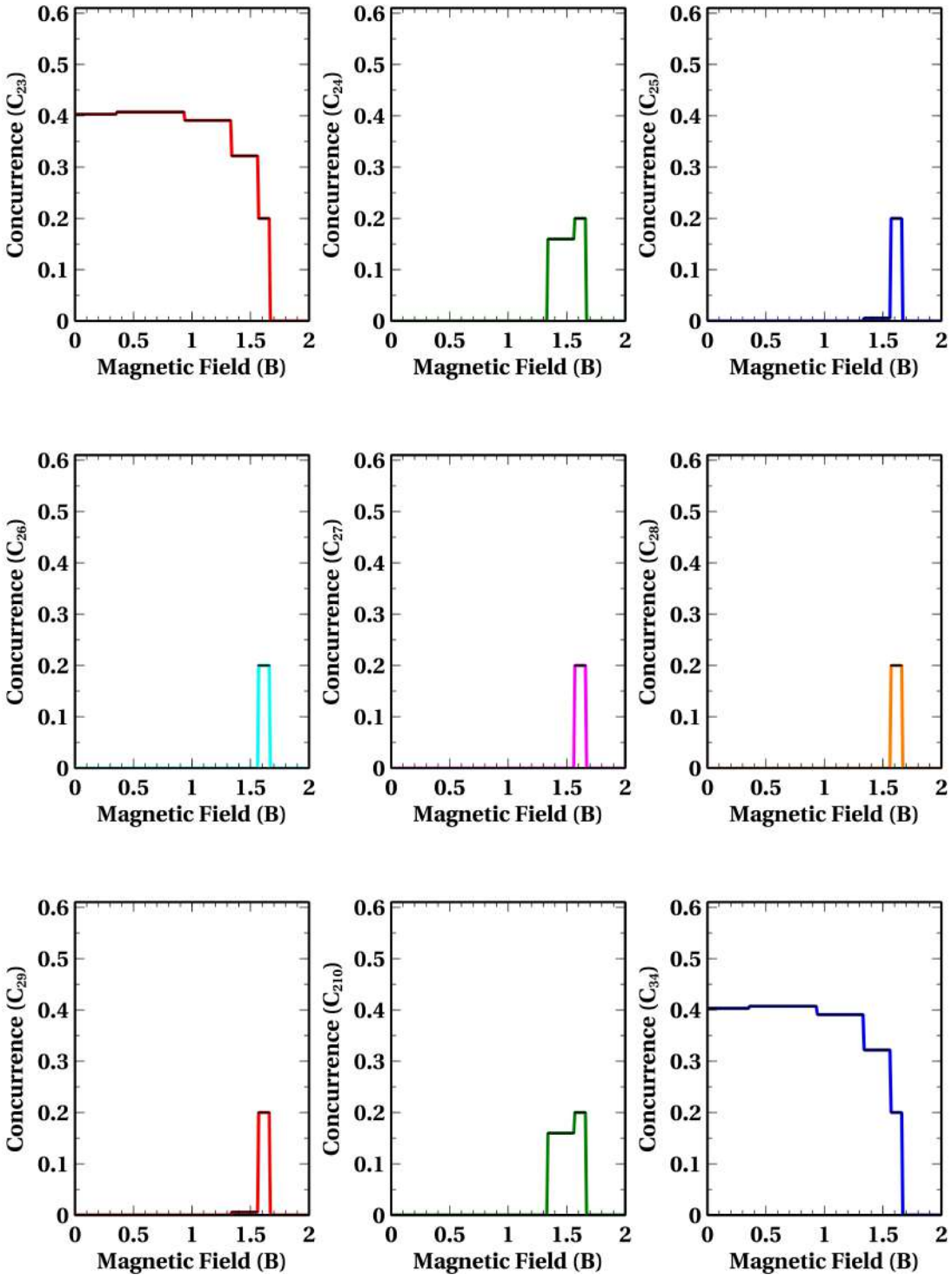

Concurrence (C_{67})
Magnetic Field (B)
0
0.1
0.2
0.3
0.4
0.5
0.6
0
0.5
1
1.5
2

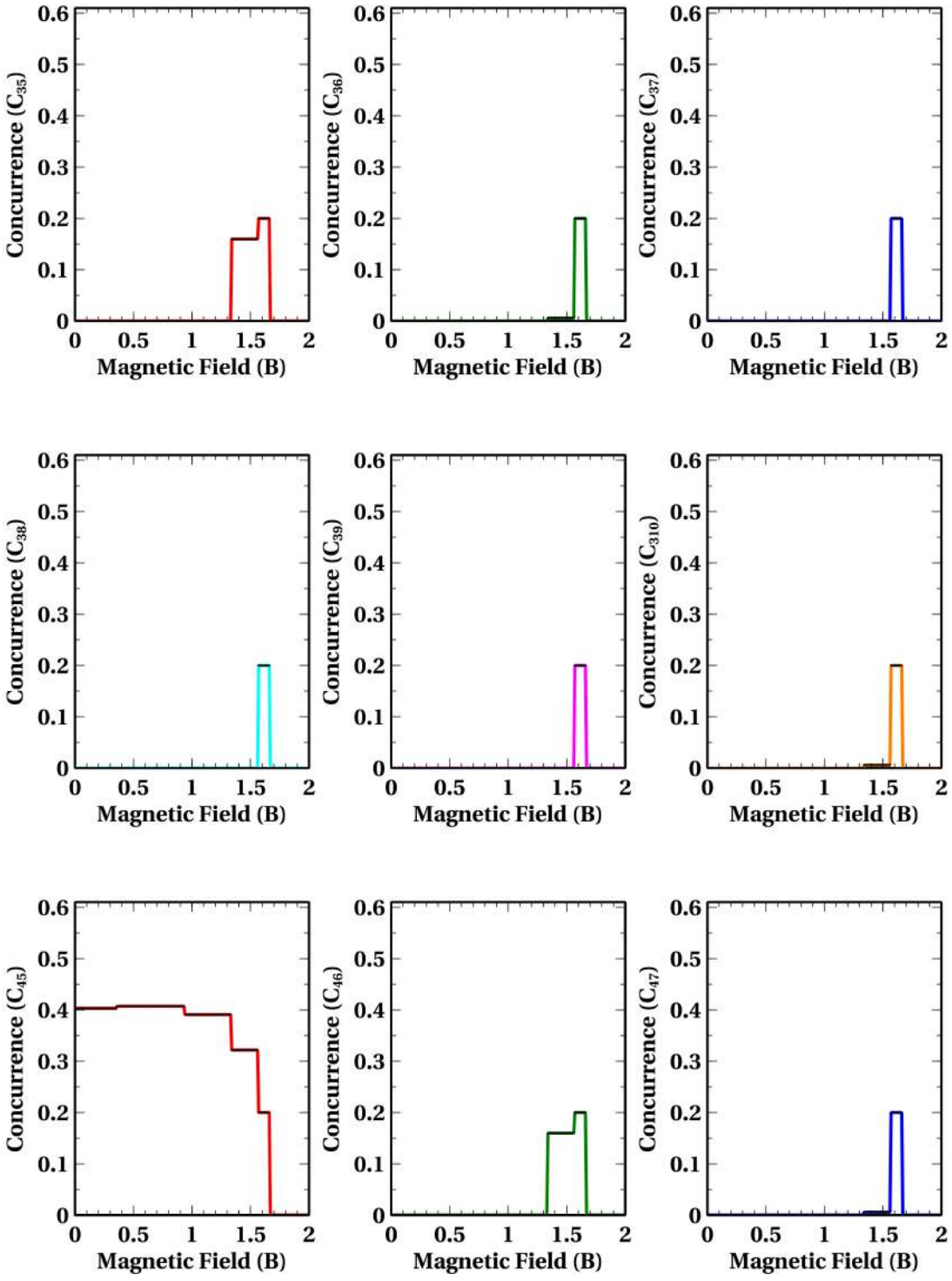

Concurrence (C_{68})
Magnetic Field (B)
0
0.1
0.2
0.3
0.4
0.5
0.6
0
0.5
1
1.5
2

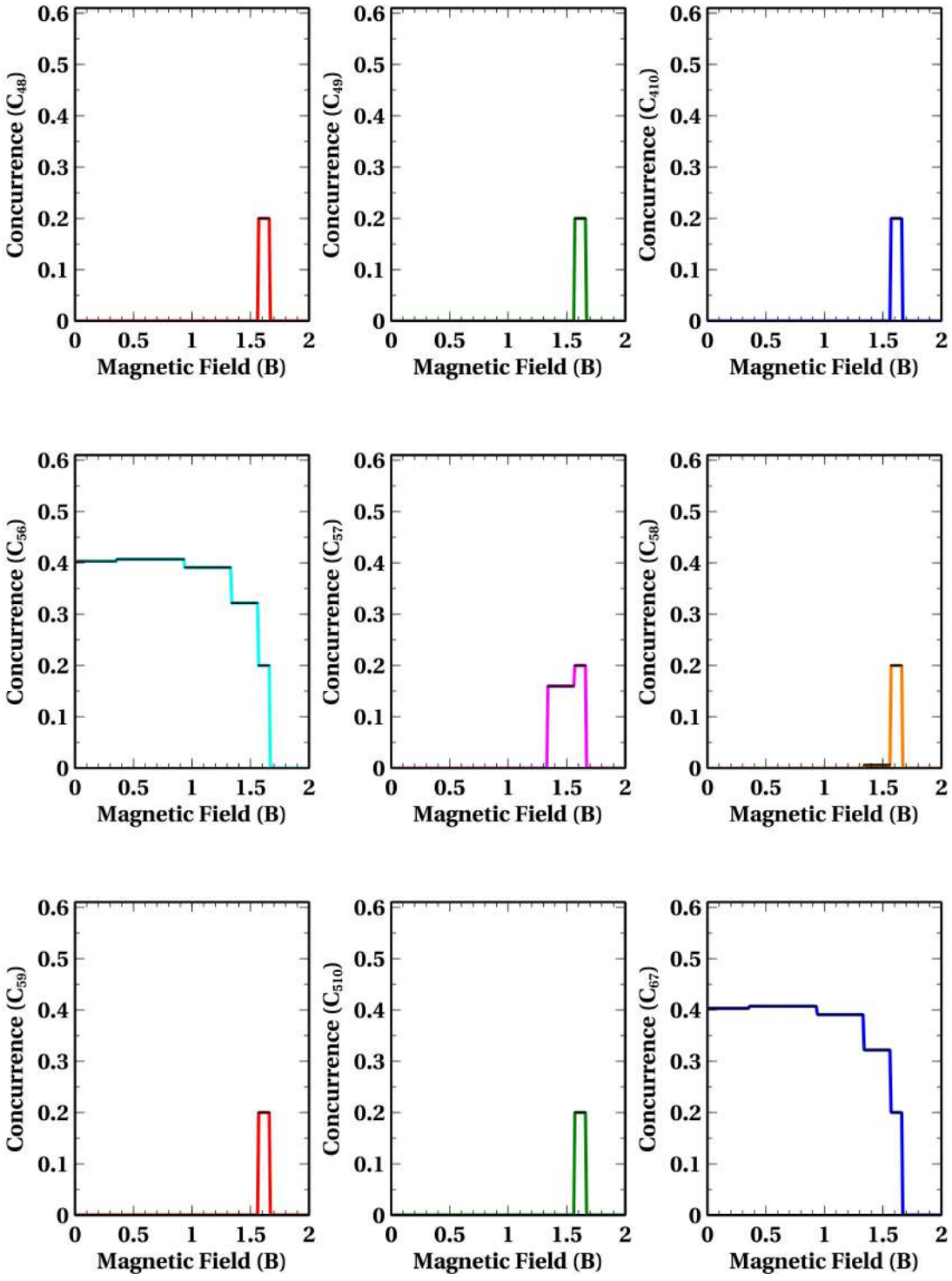

Concurrence (C_{78})
Magnetic Field (B)
0
0.1
0.2
0.3
0.4
0.5
0.6
0
0.5
1
1.5
2

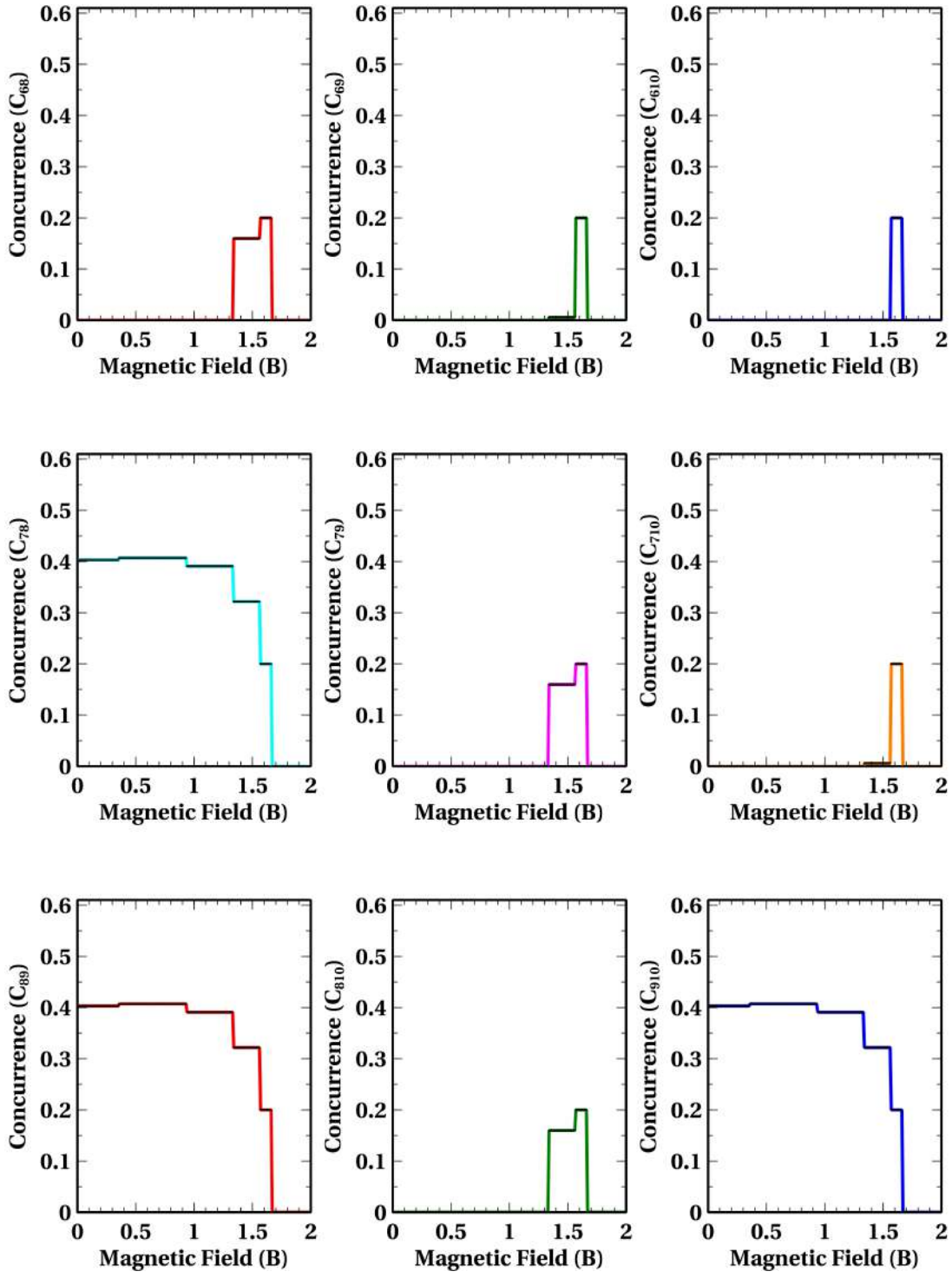
4.2.8 Concurrence vs Magnetic Field (B) for a 10 qubit AKLT Model




Concurrence (C_{23})
Concurrence (C_{24})
Concurrence (C_{25})
Concurrence (C_{26})
Concurrence (C_{27})
Concurrence (C_{28})
Concurrence (C_{29})
Concurrence (C_{210})
Concurrence (C_{34})
Magnetic Field (B)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.5
1
1.5
2


Concurrence (C_{35})
Concurrence (C_{36})
Concurrence (C_{37})
Concurrence (C_{38})
Concurrence (C_{39})
Concurrence (C_{310})
Concurrence (C_{45})
Concurrence (C_{46})
Concurrence (C_{47})
Magnetic Field (B)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.5
1
1.5
2


Concurrence (C_{48})
Concurrence (C_{49})
Concurrence (C_{410})
Concurrence (C_{56})
Concurrence (C_{57})
Concurrence (C_{58})
Concurrence (C_{59})
Concurrence (C_{510})
Concurrence (C_{67})
Magnetic Field (B)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.5
1
1.5
2



Again, from the plots, it is visible that, as the number of qubits increases, the number of transitions also increases. The maximum value of concurrence is 0.5, and as the number of qubits increases, the concurrence decreases, which obeys the monogamy of

entanglement. Also, the variation of concurrence for varying magnetic field is opposite to the trend shown by varying interaction energy. Also, similar to interaction energy vs concurrence, the transition point changes with changing interaction energy for concurrence vs magnetic field. However, we have also verified the claim of AKLT that the ground state of this model will be the valence-bond state (i.e. spin-1 state). Let us turn our attention to the study of the multipartite entanglement measure, which is the von Neumann entropy.

4.3 von Neumann Entropy and Block Entropy

For a two-qubit pure state $|\psi\rangle$, the most accepted measure of entanglement($E(|\psi\rangle)$) in a pure state is the von Neumann entropy [30], also known as entanglement entropy. So for a pure state $|\psi\rangle$, von Neumann entropy is defined as [31],

$$E(\psi) = -\mathrm{Tr}(\rho^A \log_2 \rho^A) = -\mathrm{Tr}(\rho^B \log_2 \rho^B) \tag{4.6}$$

$$= -\sum_{i=1}^{dim(\rho^A)} \lambda_i \log_2(\lambda_i) = -\sum_{j=1}^{dim(\rho^B)} \lambda_j \log_2(\lambda_j), \tag{4.7}$$

here ρ^A and ρ^B are the reduced density matrices of ρ^{AB}, and λ_i and λ_j are the non-zero eigenvalues of the reduced density matrix of ρ^A and ρ^B respectively. The total entropy of a maximally entangled composite system is zero, but the entropies for the two subsystems are unity.

For a multiqubit pure state, the concept of block entropy [32] comes into the picture, which is nothing but the extension of von Neumann entropy to a multiqubit bipartite system. Suppose we are considering a N qubit pure state, then we can split the system into two subsystems like L and $N-L$, where $L < N$. Now we can find the block entropy corresponding to these two blocks as follows,

$$S(\rho^L) = -\mathrm{Tr}(\rho^L \log_2 \rho^L) = -\sum_{i=1}^{\dim(\rho^L)} \lambda_i' \log_2 \lambda_i' \tag{4.8}$$

$$S(\rho^{N-L}) = -\mathrm{Tr}(\rho^{N-L} \log_2 \rho^{N-L}) = -\sum_{i=1}^{\dim(\rho^{N-L})} \lambda_i'' \log_2 \lambda_i''. \tag{4.9}$$

Here, $\lambda_i's$ and $\lambda_i^{"}$s are the non-zero eigenvalues of the matrices ρ^L or ρ^{N-L} respectively which are equal.

Now, let us study the entanglement properties of our system using the above-defined entropy. Here is the Fortran implementation of the entropy on our AKLT system. However, one can make a simple program for this, but we are giving an optimised program for parallel programming in Fortran.

```
program akltenmpi
    use mpi
    implicit none

    integer, parameter :: d = 2, s = 4, s1 = 2
    integer, parameter :: num = d**s, N = num, di = d**s1
    integer :: i, j, k, l, t, flagx, flagy, pos, u
    integer :: myrank, numprocs, ierr
    integer, allocatable :: digits1(:), digits2(:), decimal(:), site
   (:)
    character(len=:), allocatable :: state(:)
    double precision, allocatable :: E(:,:), C(:,:), F(:,:), G(:,:),
    H(:,:), Z(:,:)
    character*1 :: JOBZ, UPLO, RANGE
    integer ::  LDA = num, LWORK = 8*num, INFO, IL, IU, M, LDZ,
   IWORK(5*N), IFAIL(N)
    double precision :: W(num), WORK(8*num), VL, VU, ABSTOL
    real*8 :: sum, trace, Jo, B, be
    real*8, allocatable :: psi(:,:)
    real*8 :: startTime, endTime

    call MPI_Init(ierr)
    call MPI_Comm_rank(MPI_COMM_WORLD, myrank, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, numprocs, ierr)

    if (myrank < 3) then
        call acc_set_device_num(0, acc_device_nvidia)
    else
        call acc_set_device_num(1, acc_device_nvidia)
```

```
27      end if
28
29      call cpu_time(startTime)
30
31      ! Allocate arrays
32      allocate(character(len=s) :: state(num))
33      allocate(integer :: decimal(num))
34      allocate(integer :: digits1(s), digits2(s))
35      allocate(real(8) :: C(num, num))
36      allocate(real(8) :: E(num, num))
37      allocate(real(8) :: F(num, num))
38      allocate(real(8) :: G(num, num))
39      allocate(real(8) :: H(0:num-1 , 0:num-1))
40      allocate(real(8) :: Z(0:num-1 , 0:num-1))
41      allocate(psi(0:num-1,0:0))
42      allocate(site(0:s1-1))
43
44      C = 0.0d0
45      F = 0.0d0
46      G = 0.0d0
47
48      !$acc data copyin(C,F,G) copy(state,decimal)
49
50      do i = 0, num - 1
51          call integer_binary(i, state(i+1), s)
52      end do
53
54      do i = 1, num
55          decimal(i) = btod(state(i))
56      end do
57
58
59      ! -------------------- Build full Hamiltonian terms C, F, G
    --------------------
60      G = 0.0d0
61      C = 0.0d0
62      F = 0.0d0
63
```

```
64      do k = 1, s
65           E = 0.0d0
66           do i = 1, num
67                do j = 1, num
68                     ! Computing Sigma_i^x * Sigma_i+1^x for each site
69                     flagx = 0
70                     if (decimal(i) .ne. decimal(j)) then
71                          call sd(state(i), digits1)
72                          call sd(state(j), digits2)
73                          if (k < s) then
74                               if (digits1(k) .ne. digits2(k) .and. digits1
    (k+1) .ne. digits2(k+1)) then
75                                    do l = 1, k - 1
76                                         if (digits1(l) .ne. digits2(l)) then
77                                              flagx  = flagx + 1
78                                         end if
79                                    end do
80                                    do l = k + 2, s
81                                         if (digits1(l) .ne. digits2(l)) then
82                                              flagx  = flagx + 1
83                                         end if
84                                    end do
85                                    if (flagx == 0) then
86                                         E(i, j) = E(i, j) + 1.0
87                                    else
88                                         E(i, j) = E(i, j) + 0.0
89                                    end if
90                               else
91                                    E(i, j) = E(i, j) + 0.0
92                               end if
93                          else
94                               if (digits1(1) .ne. digits2(1) .and. digits1
    (s) .ne. digits2(s)) then
95                                    do l = 2, s - 1
96                                         if (digits1(l) .ne. digits2(l)) then
97                                              flagx  = flagx + 1
98                                         end if
99                                    end do
```

```
100                             if (flagx == 0) then
101                                 E(i, j) = E(i, j) + 1.0
102                             else
103                                 E(i, j) = E(i, j) + 0.0
104                             end if
105                         else
106                             E(i, j) = E(i, j) + 0.0
107                         end if
108                     end if
109                 else
110                     E(i, j) = E(i, j) + 0.0
111                 end if
112 
113                 ! Computing Sigma_i^y * Sigma_i+1^y for each site
114                 flagy = 0
115                 if (decimal(i) .ne. decimal(j)) then
116                     call sd(state(i), digits1)
117                     call sd(state(j), digits2)
118                     if (k < s) then
119                         if (digits1(k) .ne. digits2(k) .and. digits1
    (k+1) .ne. digits2(k+1)) then
120                             do l = 1, k - 1
121                                 if (digits1(l) .ne. digits2(l)) then
122                                     flagy  = flagy + 1
123                                 end if
124                             end do
125                             do l = k + 2, s
126                                 if (digits1(l) .ne. digits2(l)) then
127                                     flagy  = flagy + 1
128                                 end if
129                             end do
130                             if (flagy == 0) then
131                                 E(i, j) = E(i, j) + (-1.0 * (-1.0)
    **(digits2(k) + digits2(k+1)))
132                             else
133                                 E(i, j) = E(i, j) + 0.0
134                             end if
135                         else
```

```
136                             E(i, j) = E(i, j) + 0.0
137                         end if
138                     else
139                         if (digits1(1) .ne. digits2(1) .and. digits1
    (s) .ne. digits2(s)) then
140                             do l = 2, s - 1
141                                 if (digits1(l) .ne. digits2(l)) then
142                                     flagy  = flagy + 1
143                                 end if
144                             end do
145                             if (flagy == 0) then
146                                 E(i, j) = E(i, j)  + (-1.0 * (-1.0)
    **(digits2(s) + digits2(1)))
147                             else
148                                 E(i, j) = E(i, j) + 0.0
149                             end if
150                         else
151                             E(i, j) = E(i, j) + 0.0
152                         end if
153                     end if
154                 else
155                     E(i, j) = E(i, j) + 0.0
156                 end if
157
158                 ! Computing Sigma_i^z * Sigma_i+1^z for each site
159                 call sd(state(i), digits1)
160                 if (decimal(i) .ne. decimal(j)) then
161                     E(i, j) =  E(i,j) + 0.0
162                 else
163                     if (k < s) then
164                         E(i, j) = E(i, j) + (1 - 2 * digits1(k)) *
    (1 - 2 * digits1(k+1))
165                     else
166                         E(i, j) = E(i, j) + (1 - 2 * digits1(1)) *
    (1 - 2 * digits1(s))
167                     end if
168                 end if
169
```

```
                ! Computing Sigma_i^z for each site
                call sd(state(i), digits1)
                if (decimal(i) .ne. decimal(j)) then
                    G(i, j) =  G(i,j) + 0.0
                else
                    G(i, j) = G(i, j) + (1 - 2 * digits1(k))
                end if
            end do
        end do
        C = C + (1.0d0/4.0d0)*E
        F = F + matmul((1.0d0/4.0d0)*E, (1.0d0/4.0d0)*E)
    end do

!$acc data copyin(C, F, G) create(H, Z, W, psi)
    do k = 0, s-2
        do l = k+1, s-1
            site = (/k, l/)
            do t = 0, 300
                Jo =  0.01 * t
                B = 1.0d0

                !$acc parallel loop collapse(2) present(H, C, F, G)
                do i = 1, num
                    do j = 1, num
                        H(i-1,j-1) = Jo * (C(i,j) + (1.0d0/3.0d0) *
   F(i,j)) + B * (1.0d0/2.0d0) * G(i,j)
                    end do
                end do

                call DSYEVX(JOBZ, RANGE, UPLO, N, H, LDA, VL, VU, IL
   , IU, ABSTOL, M, W, Z, LDZ, WORK, LWORK, IWORK, IFAIL, INFO)

                !$acc parallel loop collapse(2) present(Z, psi)
                do i = 0, M-1
                    do j = 0, num - 1
                        if (abs(Z(j,i)) <= 1.0d-12) then
                            psi(j,i) = 0.0d0
                        else
```

```
                                psi(j,i) = Z(j,i)
                            end if
                        end do
                    end do

                    call BERPVR(s, s1, site, psi, be)
                    pos = k*10 + l
                    write(400+pos,*) Jo, be
                end do
            end do
        end do
!$acc end data

    call cpu_time(endTime)
    if (myrank == 0) then
        print *, "Execution Time (s): ", endTime - startTime
    end if

    call MPI_Finalize(ierr)

contains

    subroutine integer_binary(num, binary_string, n)
        implicit none
        integer :: num
        integer, intent(in) :: n
        integer :: i, temp_num
        character(len=n), intent(out) :: binary_string
        temp_num = num
        ! Initialize binary_string to all zeros
        binary_string = repeat('0', n)
        ! Convert the integer to binary
        do i = 0, n - 1
            if (mod(temp_num, 2) == 1) then
                binary_string(n-i:n-i) = '1'  ! Set the bit to '1'
            else
                binary_string(n-i:n-i) = '0'  ! Set the bit to '0'
            end if
```

```
244             temp_num = temp_num / 2  ! Divide num by 2 to shift
    right
245         end do
246     end subroutine integer_binary
247
248     subroutine sd(binaryString, digits)
249         character(len=*), intent(in) :: binaryString
250         integer, allocatable, intent(out) :: digits(:)
251         integer :: i, len
252         len = len_trim(binaryString)  ! Get the length of the binary
     string
253         allocate(digits(len))          ! Allocate array to hold
    digits
254         ! Convert each character to an integer
255         do i = 1, len
256             if (binaryString(i:i) == '1') then
257                 digits(i) = 1
258             else if (binaryString(i:i) == '0') then
259                 digits(i) = 0
260             else
261                 print *, "Invalid character in binary string."
262                 digits = 0  ! Set to zero if invalid character is
    found
263                 return
264             end if
265         end do
266     end subroutine sd
267
268     function btod(binaryString) result(decimalValue)
269         implicit none
270         character(len=*), intent(in) :: binaryString
271         integer :: decimalValue
272         integer :: i, length
273
274         decimalValue = 0
275         length = len_trim(binaryString)  ! Get the length of the
    binary string
276
```

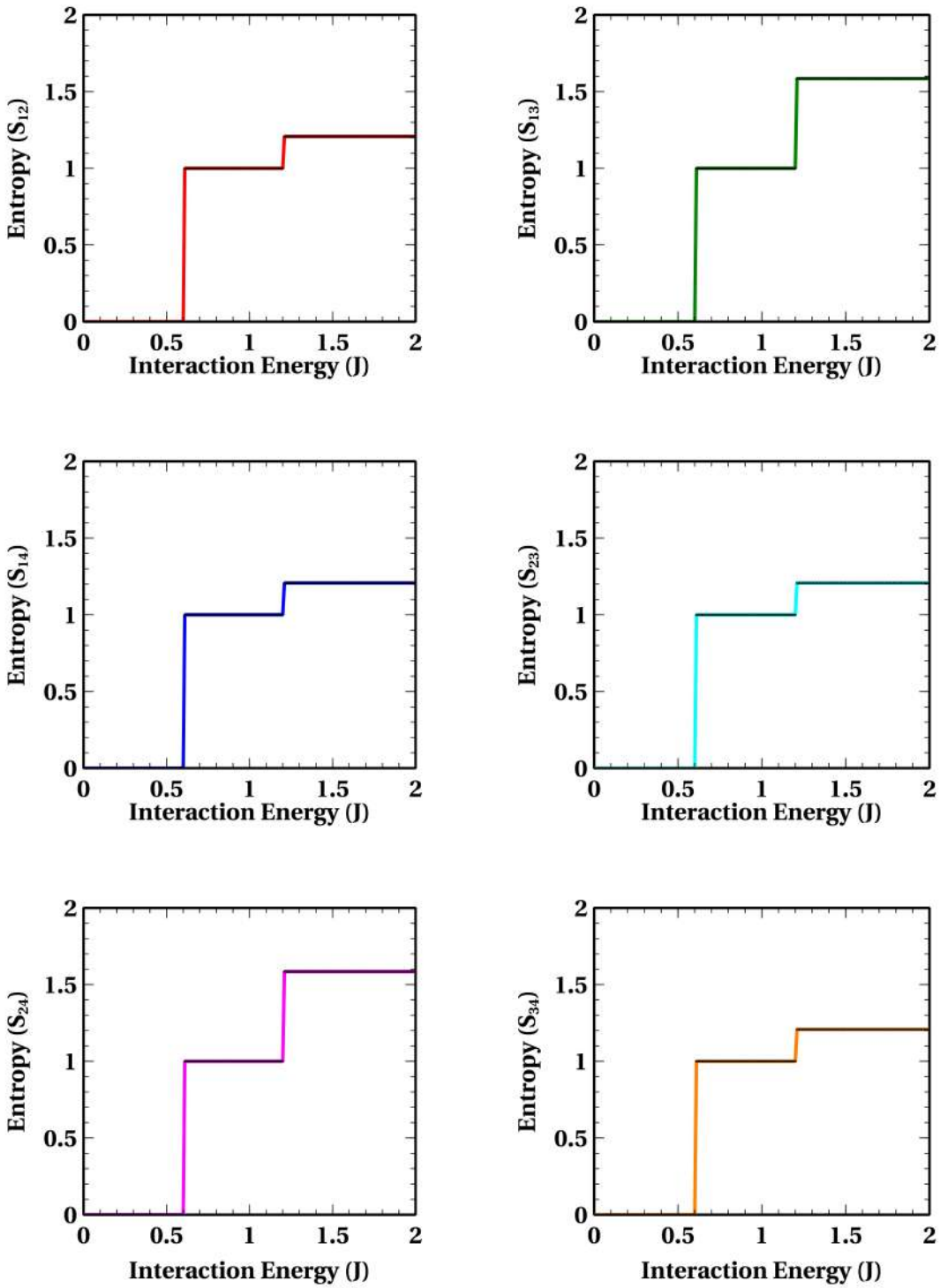
```
277         ! Convert binary string to decimal
278         do i = 1, length
279             if (binaryString(i:i) == '1') then
280                 decimalValue = decimalValue + 2**(length - i)
281             else if (binaryString(i:i) /= '0') then
282                 print *, "Invalid character in binary string."
283                 decimalValue = -1  ! Indicate an error with -1
284                 return
285             end if
286         end do
287     end function btod
288
289     subroutine BERPVR(s,s1,site,vin,be)
290         implicit none
291         integer::s, s1,site(0:s1-1)
292         real*8::be,betemp,vin(0:2**s-1),rdm(0:2**s1-1,0:2**s1-1)
293         integer::i,INFO
294         real*8::WORK(0:3*(2**s1)-1),W(0:2**s1-1)
295
296         call PTRVR(s,s1,site,vin,rdm)
297         call DSYEV('N','U', 2**s1, rdm, 2**s1, W, WORK,3*(2**s1)-1,
    INFO)
298         be=0
299         betemp=0
300         do i=0,2**s1-1,1
301             if (W(i)>0.0000000000001) then
302                 betemp=-(dlog(abs(W(i)))*abs(W(i)))/dlog(2.0d0)
303                 be=be+betemp
304             end if
305         end do
306     end subroutine
307
308     subroutine PTRVR(s,s1,site,vin,rdm)
309         implicit none
310         integer::s, s1,s2
311         real*8::trace
312         real*8,dimension(0:2**s-1)::vin
313         real*8,dimension(0:2**s1-1,0:2**s1-1)::rdm
```

```
314         integer::ii,i1,a,i0,ia,j1,oo,k1,x1,y1,j,t1,t2,z1,i
315         integer,dimension(0:s1-1)::site,site2
316         integer,dimension(0:s1-1)::bin
317         integer,dimension(0:(2**(s-s1))*(2**s1)-1)::ind
318         s2=s-s1
319         x1=0
320         y1=0
321         ind=0
322         do j1=2**s1-1,0,-1
323             do ii=0,2**s-1
324                 a=ii
325                 do i1=0,s1-1,1
326                     i0=site(i1)
327                     call DTOBONEBIT(a,ia,i0,s)
328                     site2(i1)=ia
329                 enddo
330                 call DTOB(j1,bin,s1)
331                 oo=1
332                 do k1=0,s1-1,1
333                     oo=oo*(bin(k1)-site2(k1))
334                 end do
335                 if(abs(oo)==1) then
336                     ind(x1)=ii
337                     x1=x1+1
338                 end if
339             end do
340         end do
341         rdm=0.0d0
342         do t1=0,2**s1-1,1
343             do t2=0,2**s1-1,1
344                 do z1=0,2**s2-1,1
345                     rdm(t1,t2)=rdm(t1,t2)+vin(ind((2**s2)*t1+z1)) &
346                     *vin(ind((2**s2)*t2+z1))
347                 enddo
348             end do
349         end do
350     end subroutine
351
```

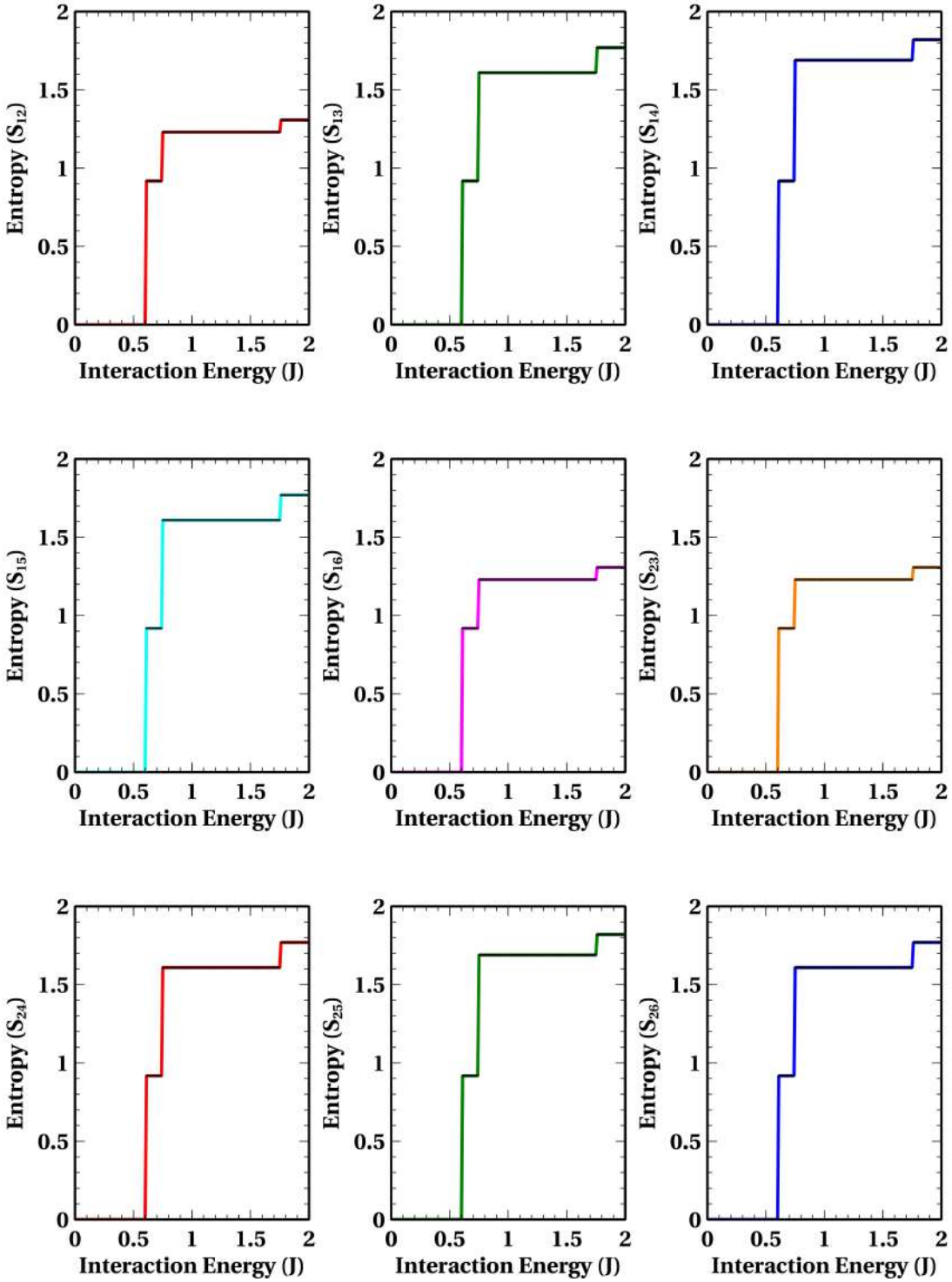
```
352     subroutine DTOB(m,tt,s)
353         implicit none
354         integer::s
355         integer,dimension(0:s-1)::tt
356         integer::m,k,a2
357         tt=0
358         a2=m
359         do k = 0,s-1,1
360             tt(s-k-1) = mod(a2,2)
361             a2 = a2/2
362             if (a2== 0) then
363                 exit
364             end if
365         end do
366     end subroutine
367 
368     subroutine DTOBONEBIT(m,ia,i0,s)
369         implicit none
370         integer*4::s
371         integer,dimension(0:s-1)::tt
372         integer::m,ia,i0
373         call DTOB(m,tt,s)
374         ia=tt(i0)
375     end subroutine
376 end program
```

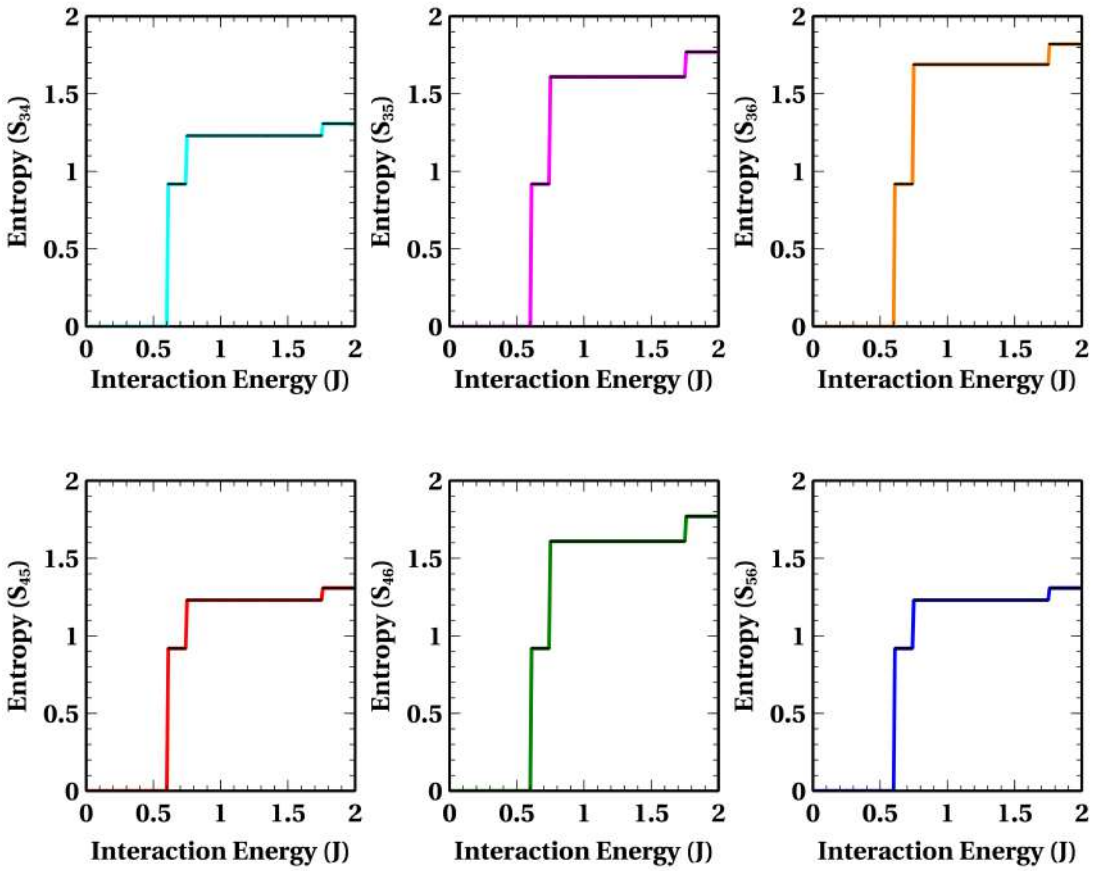
This is the general program to calculate the entropy for all bipartites by varying interaction energy (J) and keeping the magnetic field constant as ($B = 1$), where one can change the number of qubits by changing the value of s. Let us see the entropy plots for different bipartites.

4.3.1 Entropy vs Interaction Energy (J) for a 4 qubit AKLT Model

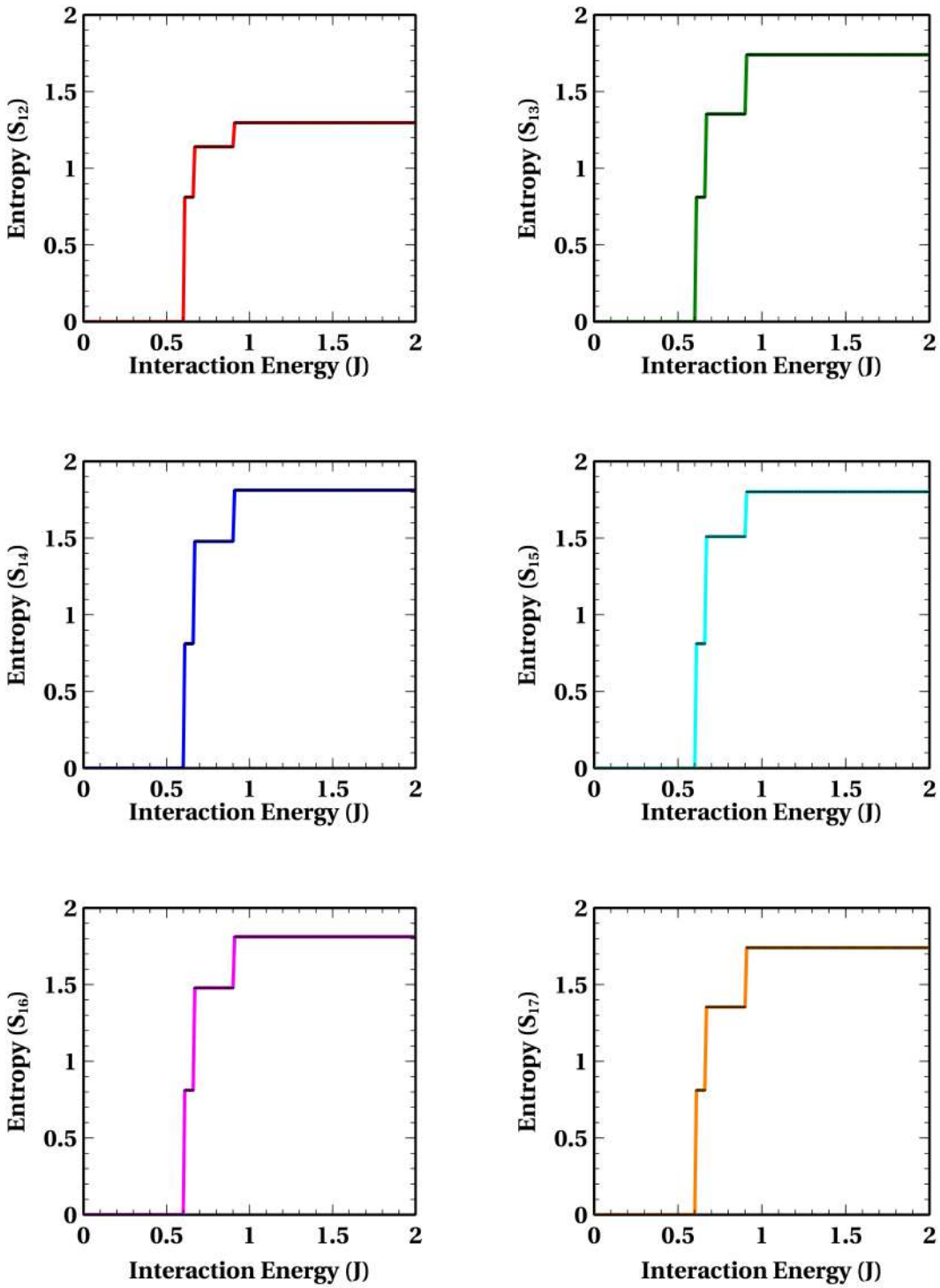


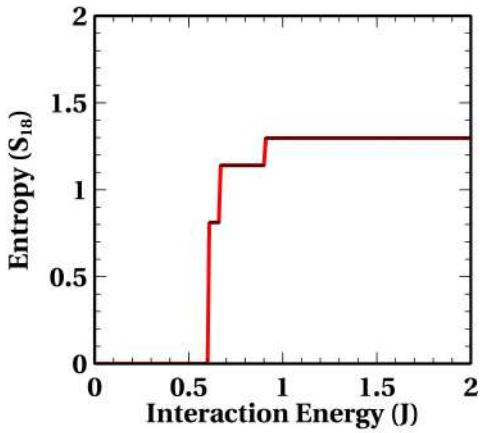
4.3.2 Entropy vs Interaction Energy (J) for a 6 qubit AKLT Model

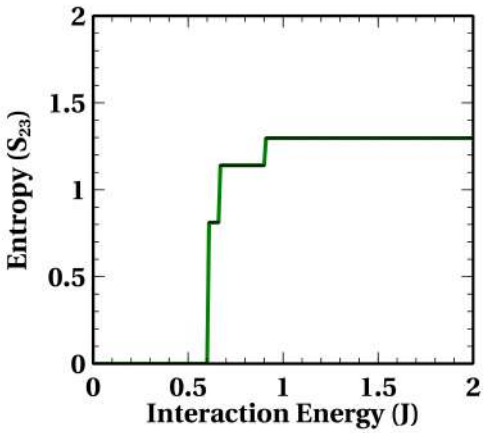


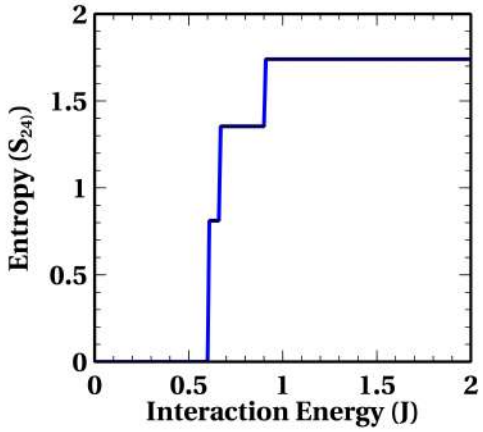

Entropy (S_{34})
Entropy (S_{35})
Entropy (S_{36})
Entropy (S_{45})
Entropy (S_{46})
Entropy (S_{56})
Interaction Energy (J)
0
0.5
1
1.5
2

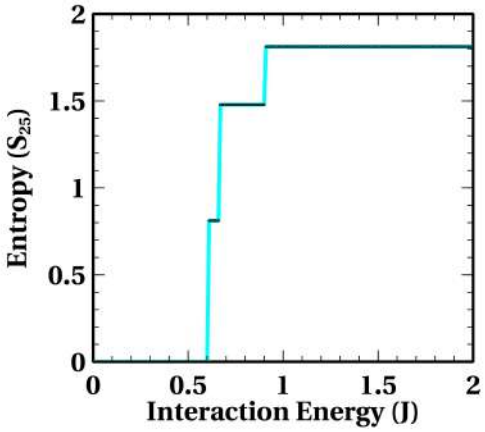
4.3.3 Entropy vs Interaction Energy (J) for a 8 qubit AKLT Model

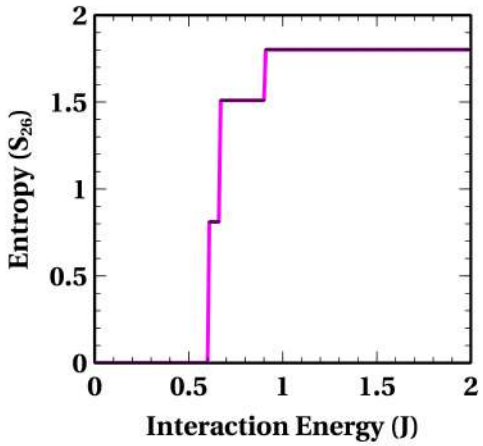


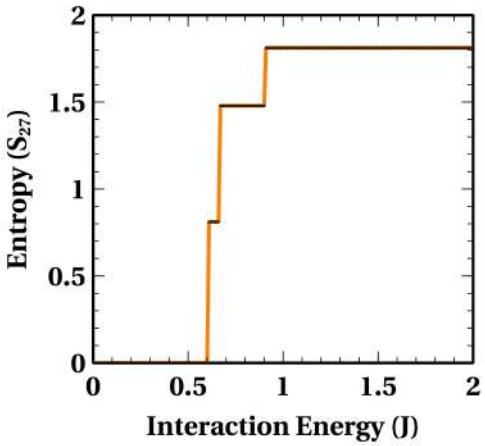

Entropy (S_{18})
Interaction Energy (J)
0
0.5
1
1.5
2

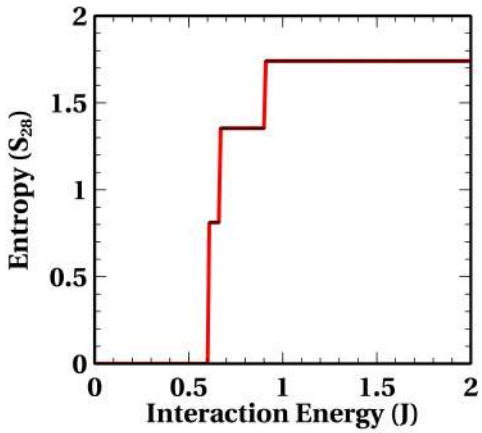

Entropy (S_{23})
Interaction Energy (J)
0
0.5
1
1.5
2

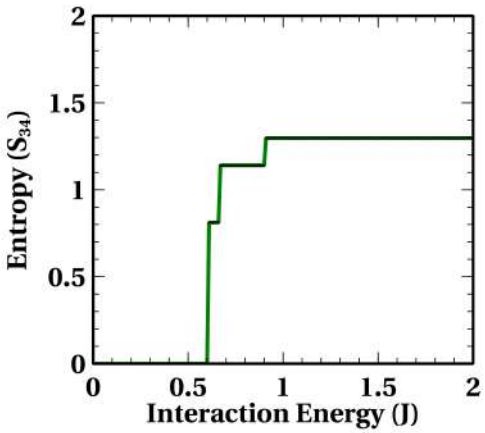

Entropy (S_{24})
Interaction Energy (J)
0
0.5
1
1.5
2

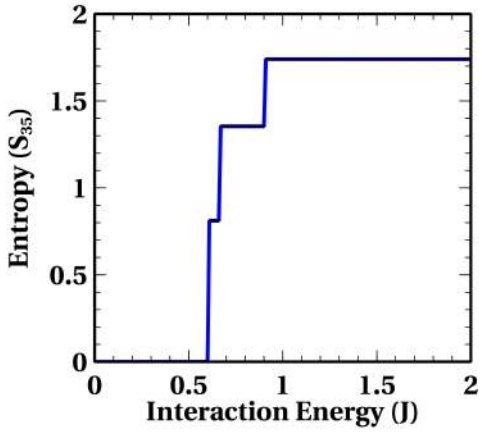

Entropy (S_{25})
Interaction Energy (J)
0
0.5
1
1.5
2

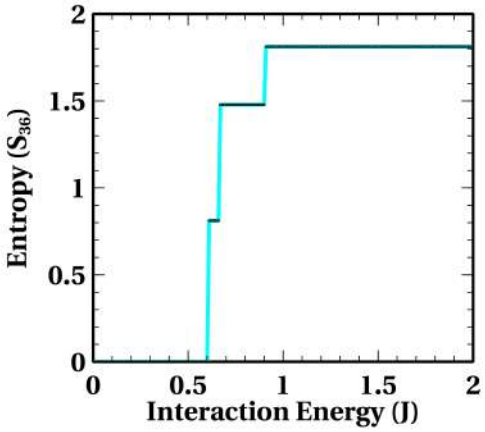

Entropy (S_{26})
Interaction Energy (J)
0
0.5
1
1.5
2

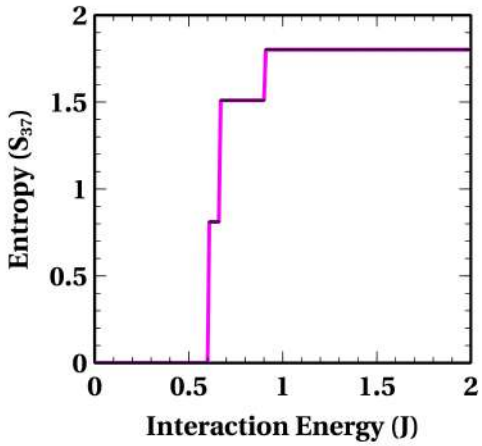

Entropy (S_{27})
Interaction Energy (J)
0
0.5
1
1.5
2

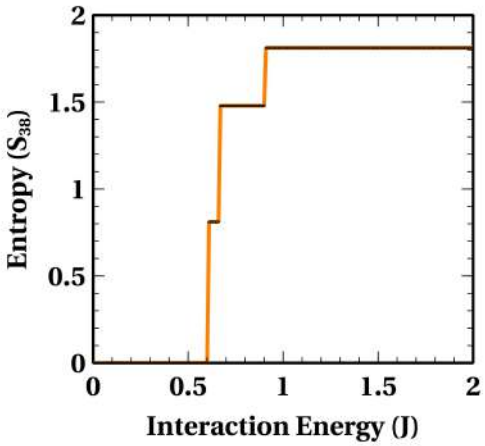

Entropy (S_{28})
Interaction Energy (J)
0
0.5
1
1.5
2

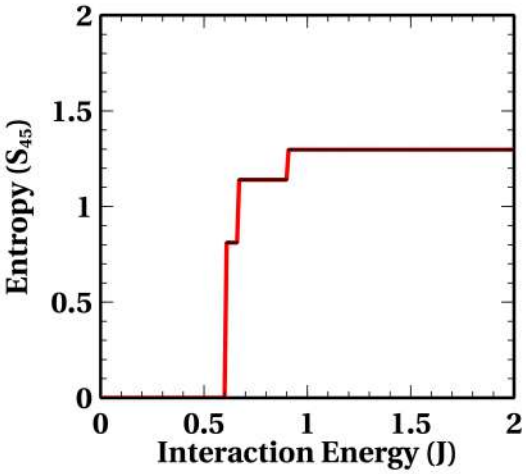

Entropy (S_{34})
Interaction Energy (J)
0
0.5
1
1.5
2

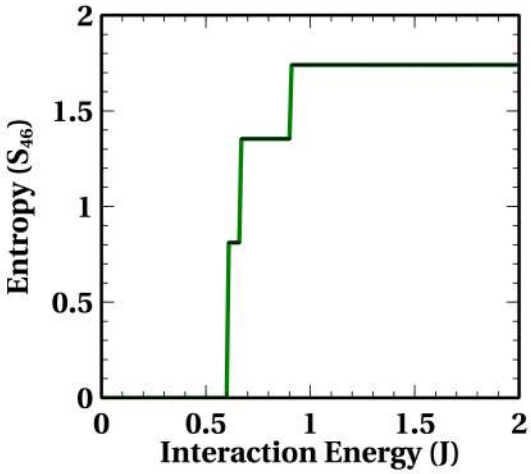

Entropy (S_{35})
Interaction Energy (J)
0
0.5
1
1.5
2

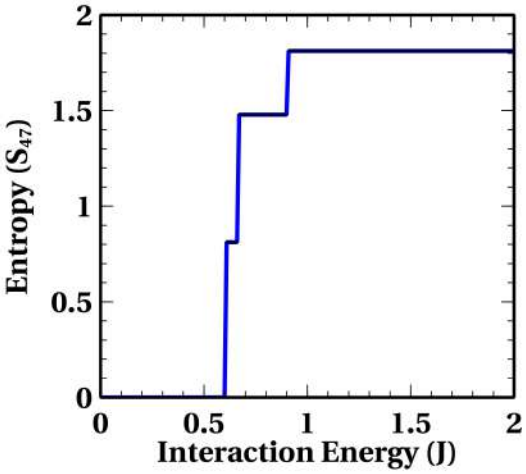

Entropy (S_{36})
Interaction Energy (J)
0
0.5
1
1.5
2

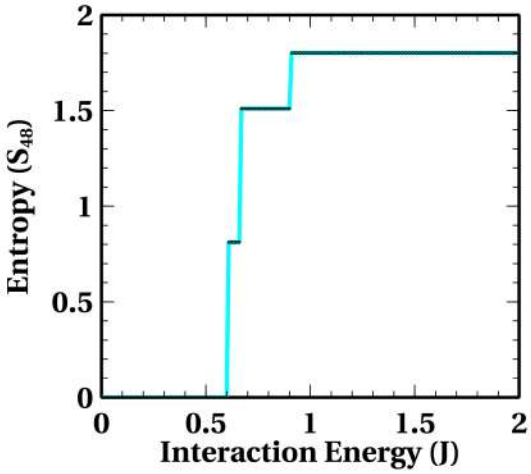

Entropy (S_{37})
Interaction Energy (J)
0
0.5
1
1.5
2

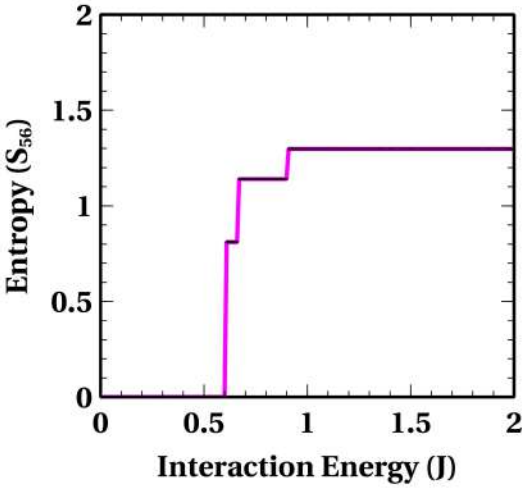

Entropy (S_{38})
Interaction Energy (J)
0
0.5
1
1.5
2

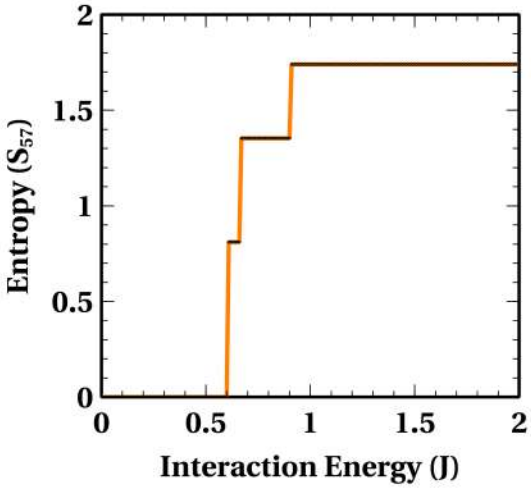

Entropy (S_{45})
Interaction Energy (J)
0
0.5
1
1.5
2

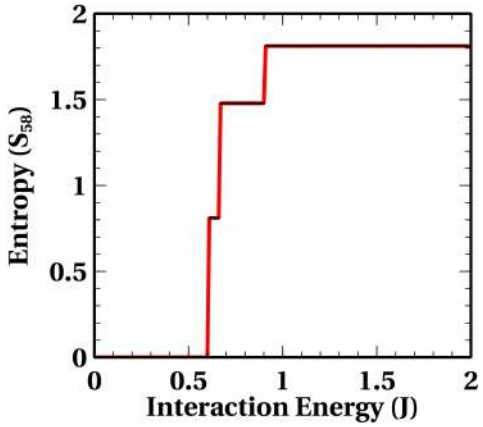

Entropy (S_{46})
Interaction Energy (J)
0
0.5
1
1.5
2

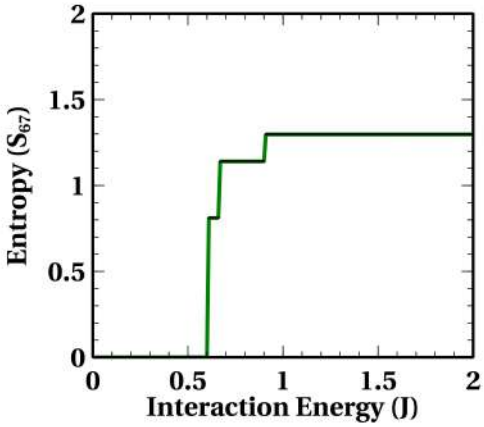

Entropy (S_{47})
Interaction Energy (J)
0
0.5
1
1.5
2

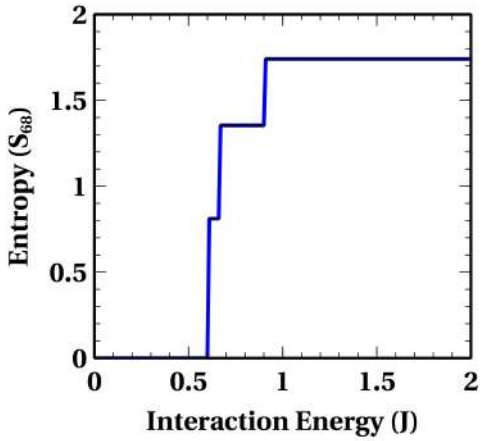

Entropy (S_{48})
Interaction Energy (J)
0
0.5
1
1.5
2

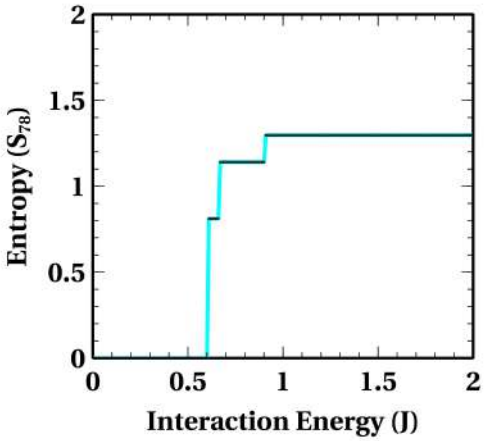

Entropy (S_{56})
Interaction Energy (J)
0
0.5
1
1.5
2


Entropy (S_{57})
Interaction Energy (J)
0
0.5
1
1.5
2

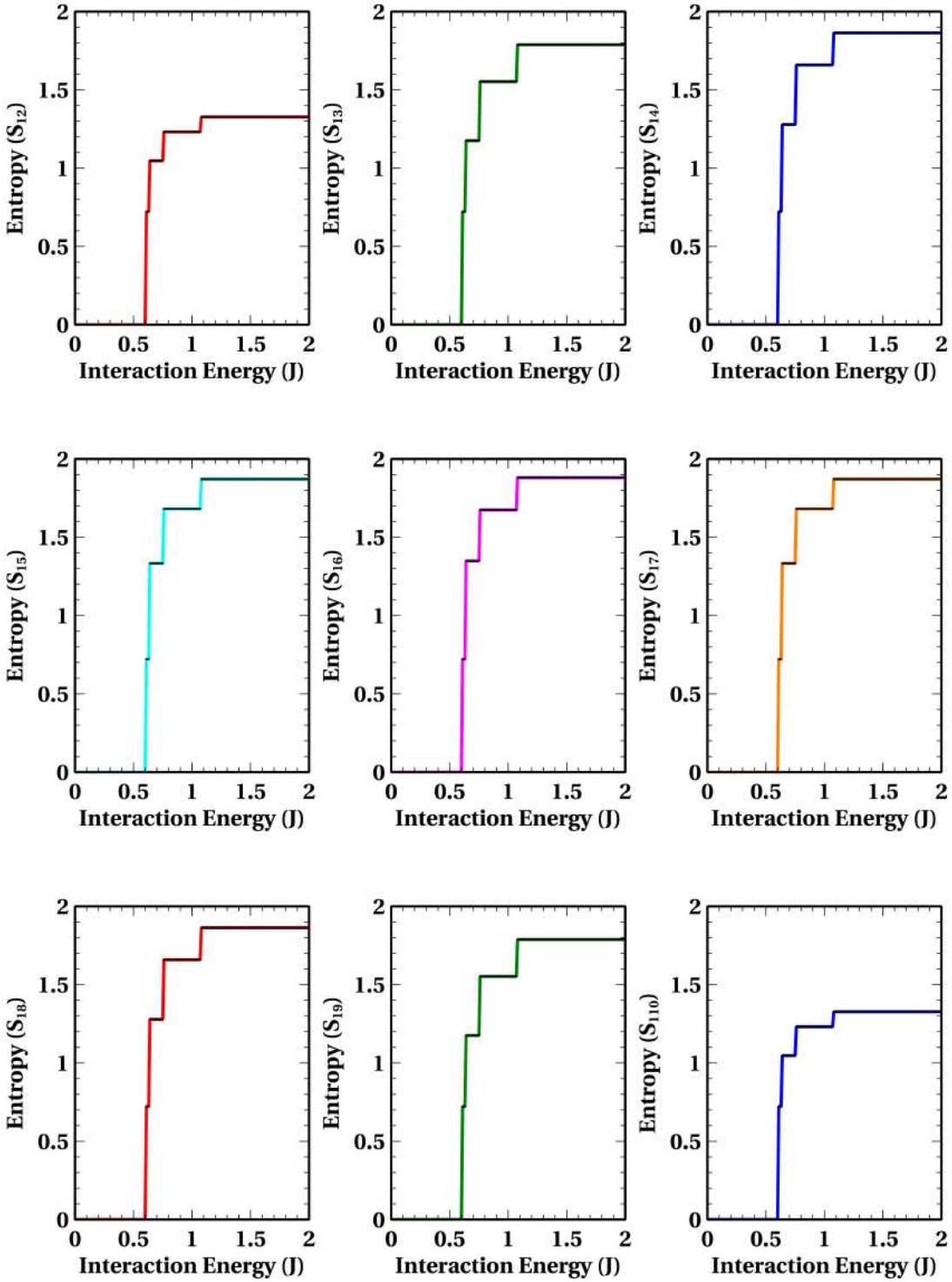

Entropy (S_{58})
Interaction Energy (J)
0
0.5
1
1.5
2

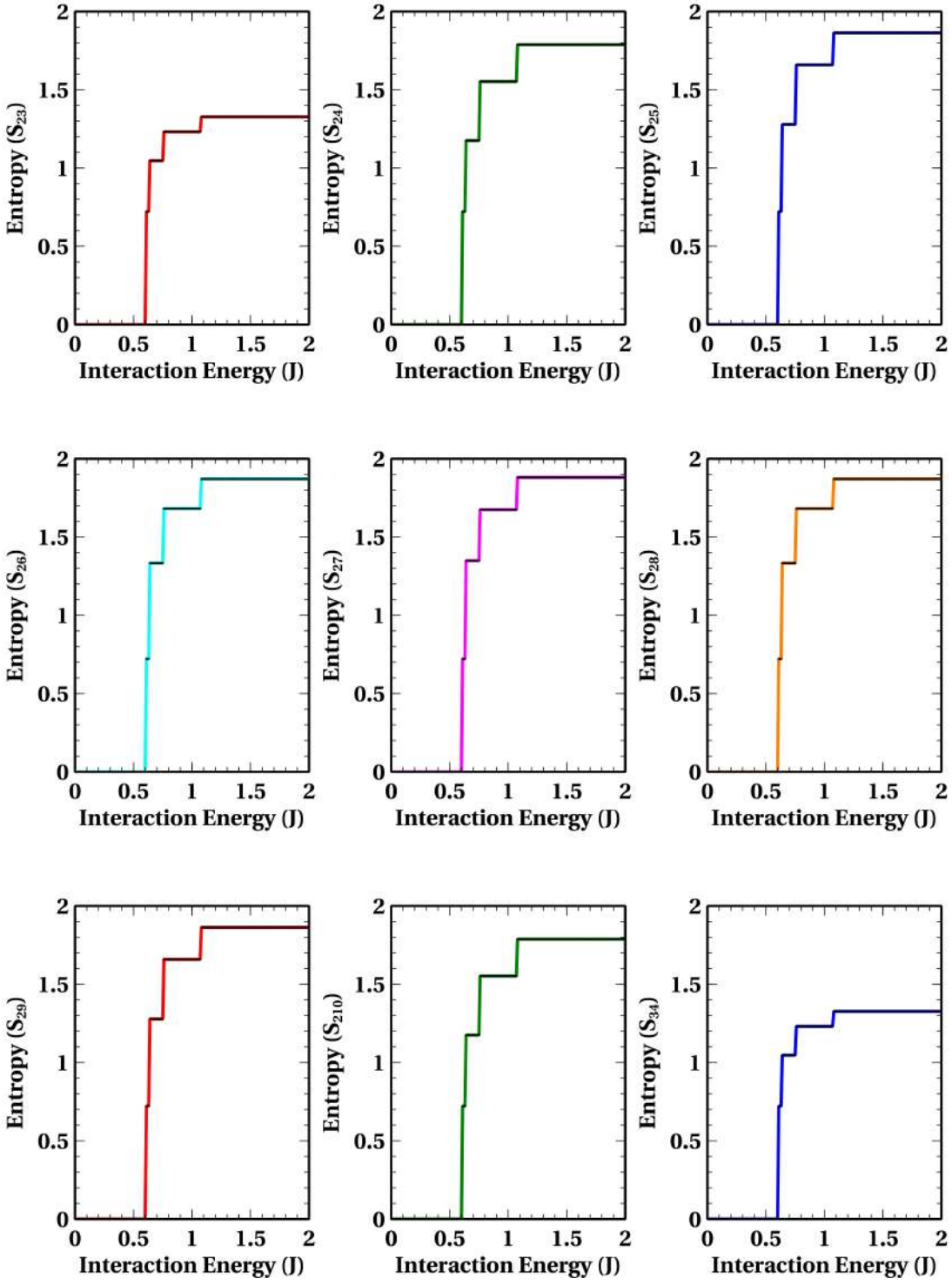

Entropy (S_{67})
Interaction Energy (J)
0
0.5
1
1.5
2

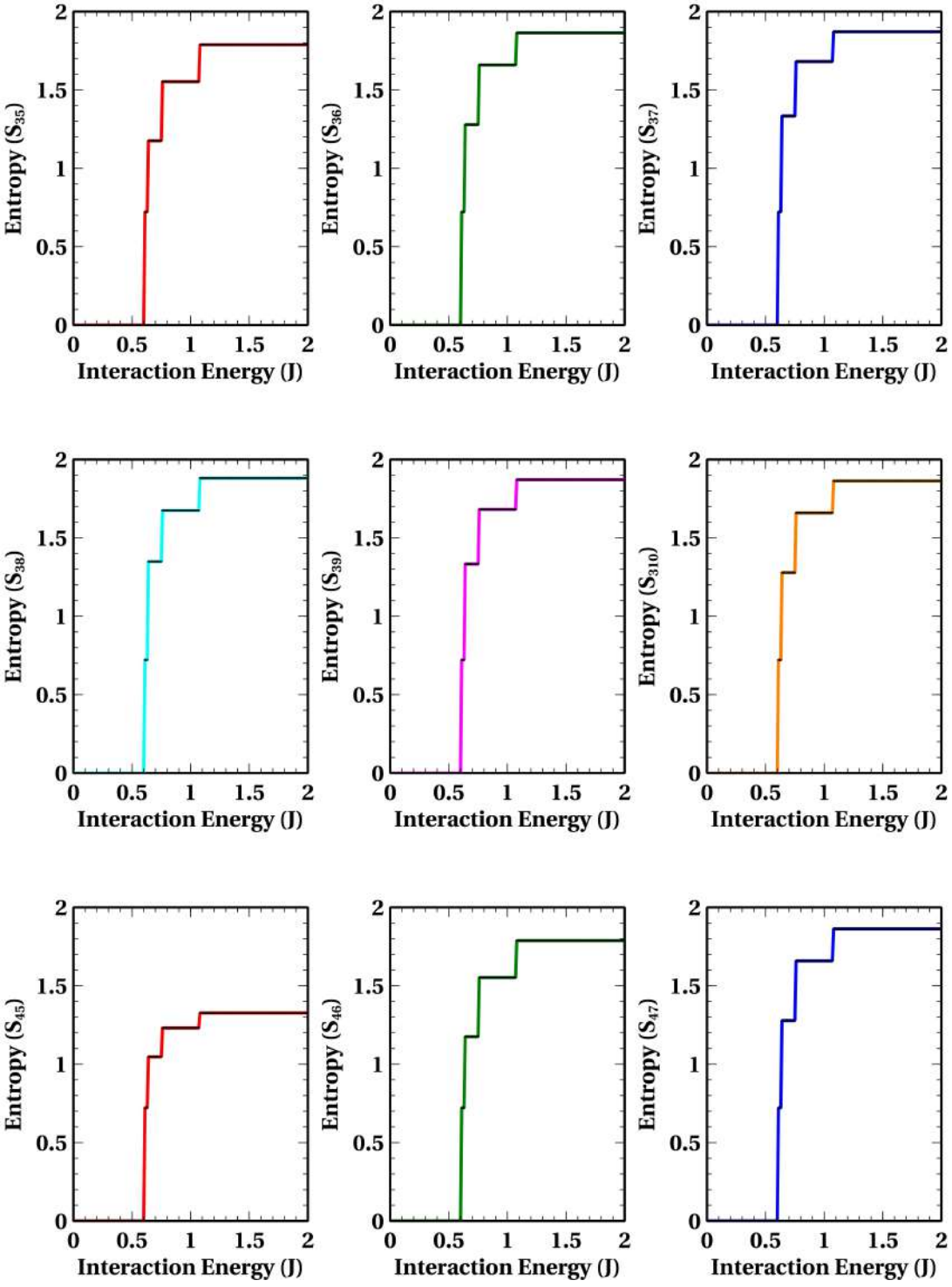

Entropy (S_{68})
Interaction Energy (J)
0
0.5
1
1.5
2

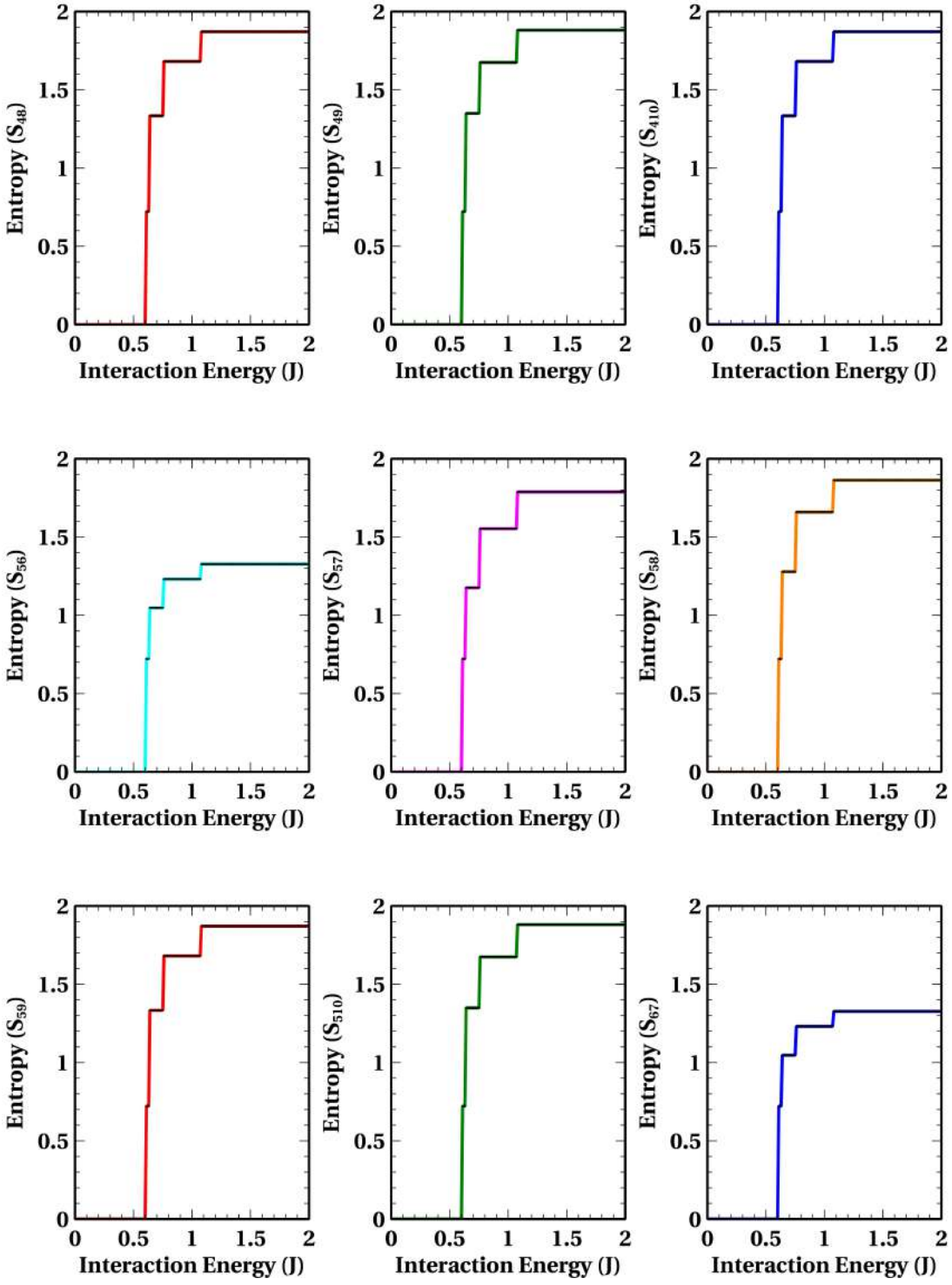

Entropy (S_{78})
Interaction Energy (J)
0
0.5
1
1.5
2

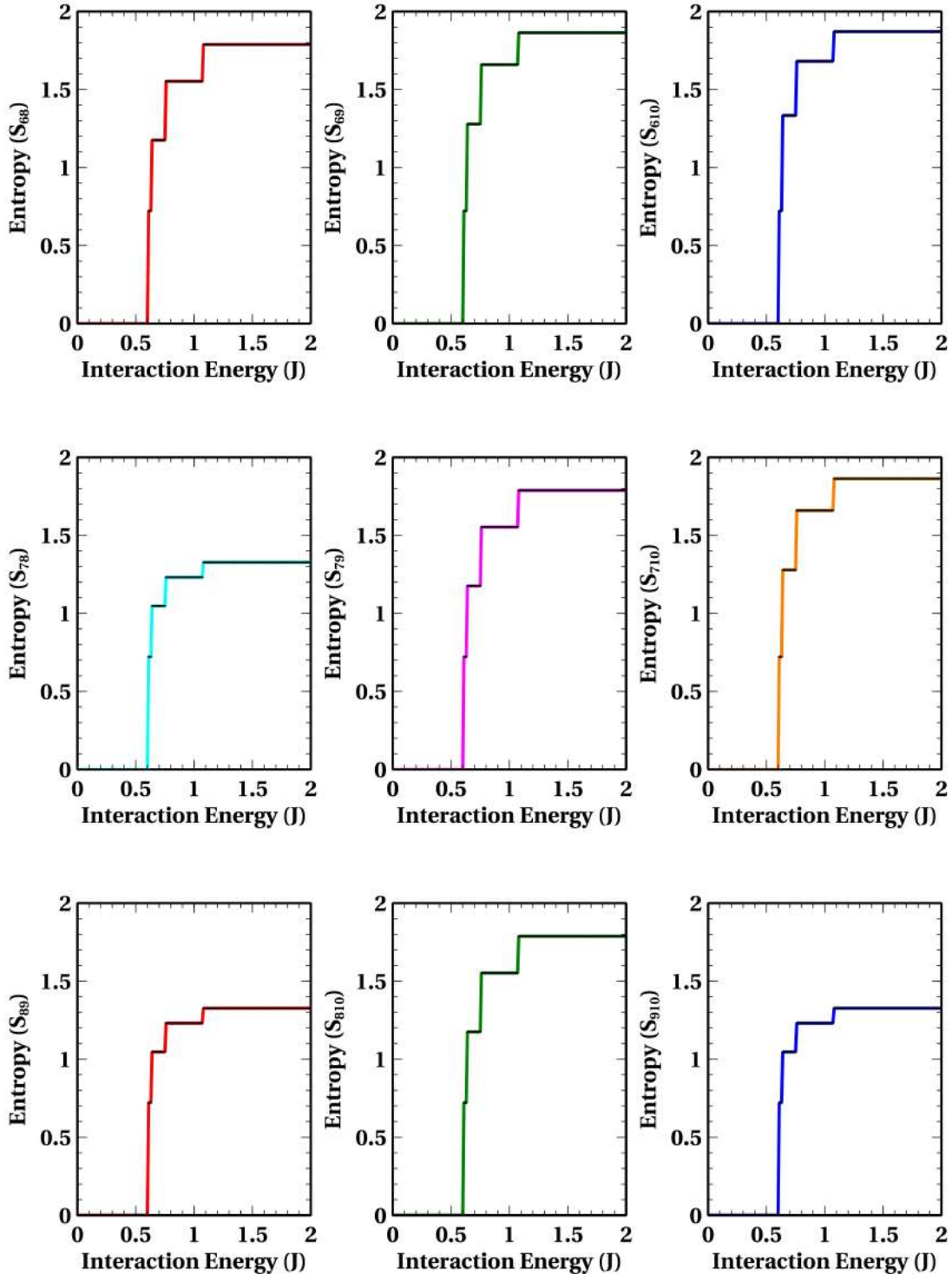
4.3.4 Entropy vs Interaction Energy (J) for a 10 qubit AKLT Model




Entropy (S23)
Entropy (S24)
Entropy (S25)
Entropy (S26)
Entropy (S27)
Entropy (S28)
Entropy (S29)
Entropy (S210)
Entropy (S34)
Interaction Energy (J)
0
0.5
1
1.5
2


Entropy (S_{35})
Entropy (S_{36})
Entropy (S_{37})
Entropy (S_{38})
Entropy (S_{39})
Entropy (S_{310})
Entropy (S_{45})
Entropy (S_{46})
Entropy (S_{47})
Interaction Energy (J)
0
0.5
1
1.5
2


Entropy (S_{48})
Entropy (S_{49})
Entropy (S_{410})
Entropy (S_{56})
Entropy (S_{57})
Entropy (S_{58})
Entropy (S_{59})
Entropy (S_{510})
Entropy (S_{67})
Interaction Energy (J)
0
0.5
1
1.5
2



The plots show that, as the number of qubits increase, the number of transitions also increase. The value of entropy decreases as the number of qubits increases, which obeys the monogamy of entanglement. As the number of transitions increase, the

width of each transition decreases.

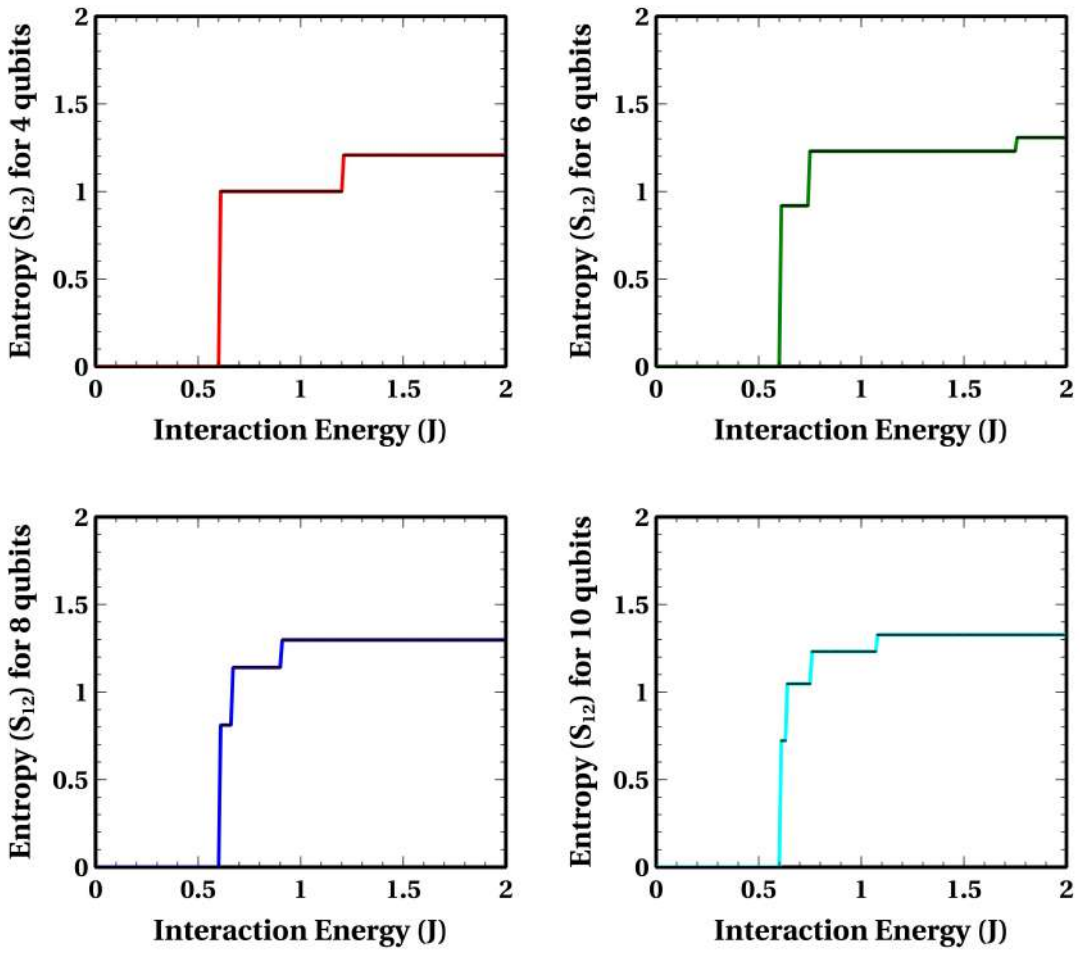


Figure 4.4: Entropy (S_{12}) vs interaction energy (J).

From the above graph, one can think that the first transition point is J = 0.6 only, but it is not the case; remember the variation of concurrence with interaction energy in Fig. [4.3]. Similarly, if we change the value of the magnetic field in this case, the first transition point changes in the same way as in Fig. [4.3]. Now, let us see the variation of entropy with varying magnetic field using a Fortran program in our AKLT model.

```
program akltenmpi
    use mpi
    implicit none
    integer, parameter :: d = 2, s = 4, s1 = 2
```

```
    integer, parameter :: num = d**s, N = num, di = d**s1
    integer :: i, j, k, l, t, flagx, flagy, pos, u
    integer :: myrank, numprocs, ierr
    integer, allocatable :: digits1(:), digits2(:), decimal(:), site
   (:)
    character(len=:), allocatable :: state(:)
    double precision, allocatable :: E(:,:), C(:,:), F(:,:), G(:,:),
    H(:,:), Z(:,:)
    character*1 :: JOBZ, UPLO, RANGE
    integer ::  LDA = num, LWORK = 8*num, INFO, IL, IU, M, LDZ,
   IWORK(5*N), IFAIL(N)
    double precision :: W(num), WORK(8*num), VL, VU, ABSTOL
    real*8 :: sum, trace, Jo, B, be
    real*8, allocatable :: psi(:,:)
    real*8 :: startTime, endTime
    call MPI_Init(ierr)
    call MPI_Comm_rank(MPI_COMM_WORLD, myrank, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, numprocs, ierr)

    if (myrank < 3) then
        call acc_set_device_num(0, acc_device_nvidia)
    else
        call acc_set_device_num(1, acc_device_nvidia)
    end if
    call cpu_time(startTime)
    ! Allocate arrays
    allocate(character(len=s) :: state(num))
    allocate(integer :: decimal(num))
    allocate(integer :: digits1(s), digits2(s))
    allocate(real(8) :: C(num, num))
    allocate(real(8) :: E(num, num))
    allocate(real(8) :: F(num, num))
    allocate(real(8) :: G(num, num))
    allocate(real(8) :: H(0:num-1 , 0:num-1))
    allocate(real(8) :: Z(0:num-1 , 0:num-1))
    allocate(psi(0:num-1,0:0))
    allocate(site(0:s1-1))

```

```
40      C = 0.0d0
41      F = 0.0d0
42      G = 0.0d0
43      !$acc data copyin(C,F,G) copy(state,decimal)
44      do i = 0, num - 1
45          call integer_binary(i, state(i+1), s)
46      end do
47
48      do i = 1, num
49          decimal(i) = btod(state(i))
50      end do
51      ! -------------------- Build full Hamiltonian terms C, F, G
    --------------------
52      G = 0.0d0
53      C = 0.0d0
54      F = 0.0d0
55      do k = 1, s
56          E = 0.0d0
57          do i = 1, num
58              do j = 1, num
59                  ! Computing Sigma_i^x * Sigma_i+1^x for each site
60                  flagx = 0
61                  if (decimal(i) .ne. decimal(j)) then
62                      call sd(state(i), digits1)
63                      call sd(state(j), digits2)
64                      if (k < s) then
65                          if (digits1(k) .ne. digits2(k) .and. digits1
    (k+1) .ne. digits2(k+1)) then
66                              do l = 1, k - 1
67                                  if (digits1(l) .ne. digits2(l)) then
68                                      flagx  = flagx + 1
69                                  end if
70                              end do
71                              do l = k + 2, s
72                                  if (digits1(l) .ne. digits2(l)) then
73                                      flagx  = flagx + 1
74                                  end if
75                              end do
```

```
                            if (flagx == 0) then
                                E(i, j) = E(i, j) + 1.0
                            else
                                E(i, j) = E(i, j) + 0.0
                            end if
                        else
                            E(i, j) = E(i, j) + 0.0
                        end if
                    else
                        if (digits1(1) .ne. digits2(1) .and. digits1
    (s) .ne. digits2(s)) then
                            do l = 2, s - 1
                                if (digits1(l) .ne. digits2(l)) then
                                    flagx  = flagx + 1
                                end if
                            end do
                            if (flagx == 0) then
                                E(i, j) = E(i, j) + 1.0
                            else
                                E(i, j) = E(i, j) + 0.0
                            end if
                        else
                            E(i, j) = E(i, j) + 0.0
                        end if
                    end if
                else
                    E(i, j) = E(i, j) + 0.0
                end if

                ! Computing Sigma_i^y * Sigma_i+1^y for each site
                flagy = 0
                if (decimal(i) .ne. decimal(j)) then
                    call sd(state(i), digits1)
                    call sd(state(j), digits2)
                    if (k < s) then
                        if (digits1(k) .ne. digits2(k) .and. digits1
    (k+1) .ne. digits2(k+1)) then
                            do l = 1, k - 1
```

```
                                if (digits1(l) .ne. digits2(l)) then
                                    flagy  = flagy + 1
                                end if
                            end do
                            do l = k + 2, s
                                if (digits1(l) .ne. digits2(l)) then
                                    flagy  = flagy + 1
                                end if
                            end do
                            if (flagy == 0) then
                                E(i, j) = E(i, j) + (-1.0 * (-1.0)
**(digits2(k) + digits2(k+1)))
                            else
                                E(i, j) = E(i, j) + 0.0
                            end if
                        else
                            E(i, j) = E(i, j) + 0.0
                        end if
                    else
                        if (digits1(1) .ne. digits2(1) .and. digits1
(s) .ne. digits2(s)) then
                            do l = 2, s - 1
                                if (digits1(l) .ne. digits2(l)) then
                                    flagy  = flagy + 1
                                end if
                            end do
                            if (flagy == 0) then
                                E(i, j) = E(i, j)  + (-1.0 * (-1.0)
**(digits2(s) + digits2(1)))
                            else
                                E(i, j) = E(i, j) + 0.0
                            end if
                        else
                            E(i, j) = E(i, j) + 0.0
                        end if
                    end if
                else
                    E(i, j) = E(i, j) + 0.0
```

```
147                 end if
148
149                 ! Computing Sigma_i^z * Sigma_i+1^z for each site
150                 call sd(state(i), digits1)
151                 if (decimal(i) .ne. decimal(j)) then
152                     E(i, j) =  E(i,j) + 0.0
153                 else
154                     if (k < s) then
155                         E(i, j) = E(i, j) + (1 - 2 * digits1(k)) *
    (1 - 2 * digits1(k+1))
156                     else
157                         E(i, j) = E(i, j) + (1 - 2 * digits1(1)) *
    (1 - 2 * digits1(s))
158                     end if
159                 end if
160
161                 ! Computing Sigma_i^z for each site
162                 call sd(state(i), digits1)
163                 if (decimal(i) .ne. decimal(j)) then
164                     G(i, j) =  G(i,j) + 0.0
165                 else
166                     G(i, j) = G(i, j) + (1 - 2 * digits1(k))
167                 end if
168             end do
169         end do
170         C = C + (1.0d0/4.0d0)*E
171         F = F + matmul((1.0d0/4.0d0)*E, (1.0d0/4.0d0)*E)
172     end do
173 !$acc data copyin(C, F, G) create(H, Z, W, psi)
174     do k = 0, s-2
175         do l = k+1, s-1
176             site = (/k, l/)
177             do t = 0, 300
178                 B =  0.01 * t
179                 Jo = 1.0d0
180                 !$acc parallel loop collapse(2) present(H, C, F, G)
181                 do i = 1, num
182                     do j = 1, num
```

```
                        H(i-1,j-1) = Jo * (C(i,j) + (1.0d0/3.0d0) *
    F(i,j)) + B * (1.0d0/2.0d0) * G(i,j)
                    end do
                end do
                call DSYEVX(JOBZ, RANGE, UPLO, N, H, LDA, VL, VU, IL
    , IU, ABSTOL, M, W, Z, LDZ, WORK, LWORK, IWORK, IFAIL, INFO)
                !$acc parallel loop collapse(2) present(Z, psi)
                do i = 0, M-1
                    do j = 0, num - 1
                        if (abs(Z(j,i)) <= 1.0d-12) then
                            psi(j,i) = 0.0d0
                        else
                            psi(j,i) = Z(j,i)
                        end if
                    end do
                end do
                call BERPVR(s, s1, site, psi, be)
                pos = k*10 + l
                write(400+pos,*) B, be
            end do
        end do
    end do
!$acc end data
call cpu_time(endTime)
if (myrank == 0) then
    print *, "Execution Time (s): ", endTime - startTime
end if
call MPI_Finalize(ierr)
contains
    subroutine integer_binary(num, binary_string, n)
        implicit none
        integer :: num
        integer, intent(in) :: n
        integer :: i, temp_num
        character(len=n), intent(out) :: binary_string
        temp_num = num
        ! Initialize binary_string to all zeros
        binary_string = repeat('0', n)
```

```
219         ! Convert the integer to binary
220         do i = 0, n - 1
221             if (mod(temp_num, 2) == 1) then
222                 binary_string(n-i:n-i) = '1'  ! Set the bit to '1'
223             else
224                 binary_string(n-i:n-i) = '0'  ! Set the bit to '0'
225             end if
226             temp_num = temp_num / 2  ! Divide num by 2 to shift
    right
227         end do
228     end subroutine integer_binary
229     subroutine sd(binaryString, digits)
230         character(len=*), intent(in) :: binaryString
231         integer, allocatable, intent(out) :: digits(:)
232         integer :: i, len
233         len = len_trim(binaryString)  ! Get the length of the binary
     string
234         allocate(digits(len))          ! Allocate array to hold
    digits
235         ! Convert each character to an integer
236         do i = 1, len
237             if (binaryString(i:i) == '1') then
238                 digits(i) = 1
239             else if (binaryString(i:i) == '0') then
240                 digits(i) = 0
241             else
242                 print *, "Invalid character in binary string."
243                 digits = 0  ! Set to zero if invalid character is
    found
244                 return
245             end if
246         end do
247     end subroutine sd
248     function btod(binaryString) result(decimalValue)
249         implicit none
250         character(len=*), intent(in) :: binaryString
251         integer :: decimalValue
252         integer :: i, length
```

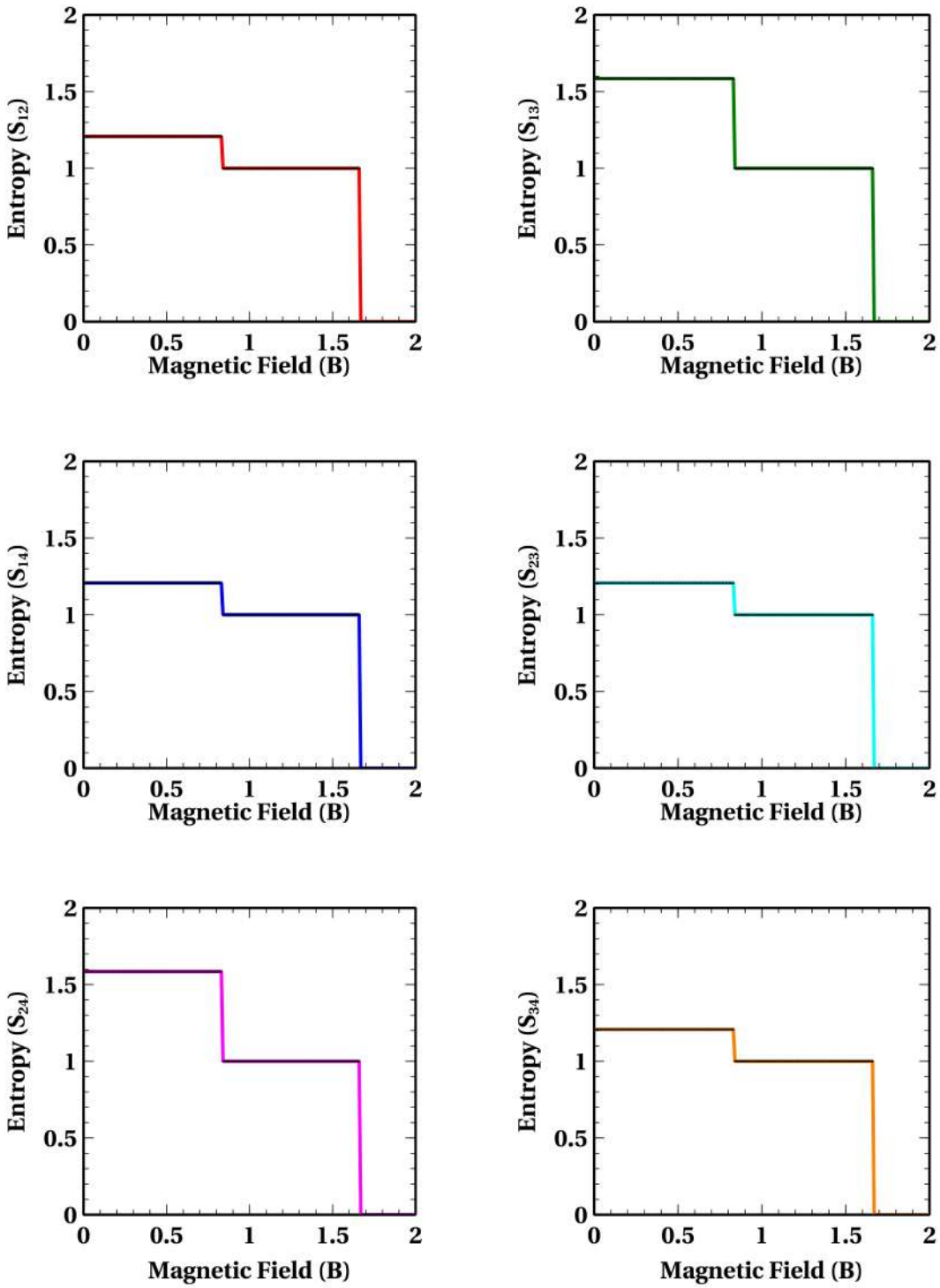
```
253         decimalValue = 0
254         length = len_trim(binaryString)  ! Get the length of the
    binary string
255         ! Convert binary string to decimal
256         do i = 1, length
257             if (binaryString(i:i) == '1') then
258                 decimalValue = decimalValue + 2**(length - i)
259             else if (binaryString(i:i) /= '0') then
260                 print *, "Invalid character in binary string."
261                 decimalValue = -1  ! Indicate an error with -1
262                 return
263             end if
264         end do
265     end function btod
266     subroutine BERPVR(s,s1,site,vin,be)
267         implicit none
268         integer::s, s1,site(0:s1-1)
269         real*8::be,betemp,vin(0:2**s-1),rdm(0:2**s1-1,0:2**s1-1)
270         integer::i,INFO
271         real*8::WORK(0:3*(2**s1)-1),W(0:2**s1-1)
272
273         call PTRVR(s,s1,site,vin,rdm)
274         call DSYEV('N','U', 2**s1, rdm, 2**s1, W, WORK,3*(2**s1)-1,
    INFO)
275         be=0
276         betemp=0
277         do i=0,2**s1-1,1
278             if (W(i)>0.0000000000001) then
279                 betemp=-(dlog(abs(W(i)))*abs(W(i)))/dlog(2.0d0)
280                 be=be+betemp
281             end if
282         end do
283     end subroutine
284     subroutine PTRVR(s,s1,site,vin,rdm)
285         implicit none
286         integer::s, s1,s2
287         real*8::trace
288         real*8,dimension(0:2**s-1)::vin
```

```
289         real*8,dimension(0:2**s1-1,0:2**s1-1)::rdm
290         integer::ii,i1,a,i0,ia,j1,oo,k1,x1,y1,j,t1,t2,z1,i
291         integer,dimension(0:s1-1)::site,site2
292         integer,dimension(0:s1-1)::bin
293         integer,dimension(0:(2**(s-s1))*(2**s1)-1)::ind
294         s2=s-s1
295         x1=0
296         y1=0
297         ind=0
298         do j1=2**s1-1,0,-1
299             do ii=0,2**s-1
300                 a=ii
301                 do i1=0,s1-1,1
302                     i0=site(i1)
303                     call DTOBONEBIT(a,ia,i0,s)
304                     site2(i1)=ia
305                 enddo
306                 call DTOB(j1,bin,s1)
307                 oo=1
308                 do k1=0,s1-1,1
309                     oo=oo*(bin(k1)-site2(k1))
310                 end do
311                 if(abs(oo)==1) then
312                     ind(x1)=ii
313                     x1=x1+1
314                 end if
315             end do
316         end do
317         rdm=0.0d0
318         do t1=0,2**s1-1,1
319             do t2=0,2**s1-1,1
320                 do z1=0,2**s2-1,1
321                     rdm(t1,t2)=rdm(t1,t2)+vin(ind((2**s2)*t1+z1)) &
322                     *vin(ind((2**s2)*t2+z1))
323                 enddo
324             end do
325         end do
326     end subroutine
```

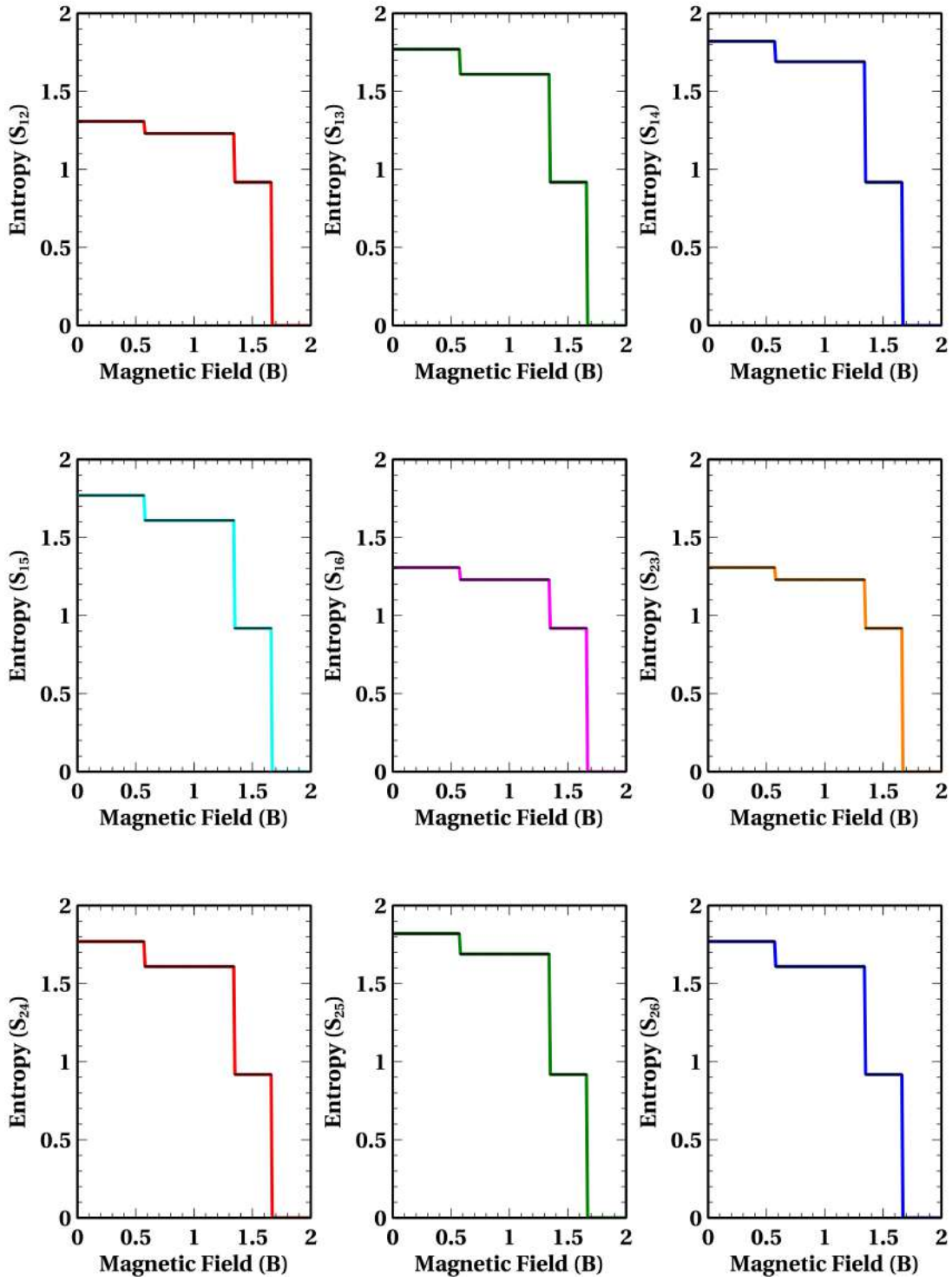
```
327     subroutine DTOB(m,tt,s)
328         implicit none
329         integer::s
330         integer,dimension(0:s-1)::tt
331         integer::m,k,a2
332         tt=0
333         a2=m
334         do k = 0,s-1,1
335             tt(s-k-1) = mod(a2,2)
336             a2 = a2/2
337             if (a2== 0) then
338                 exit
339             end if
340         end do
341     end subroutine
342     subroutine DTOBONEBIT(m,ia,i0,s)
343         implicit none
344         integer*4::s
345         integer,dimension(0:s-1)::tt
346         integer::m,ia,i0
347         call DTOB(m,tt,s)
348         ia=tt(i0)
349     end subroutine
350 end program
```

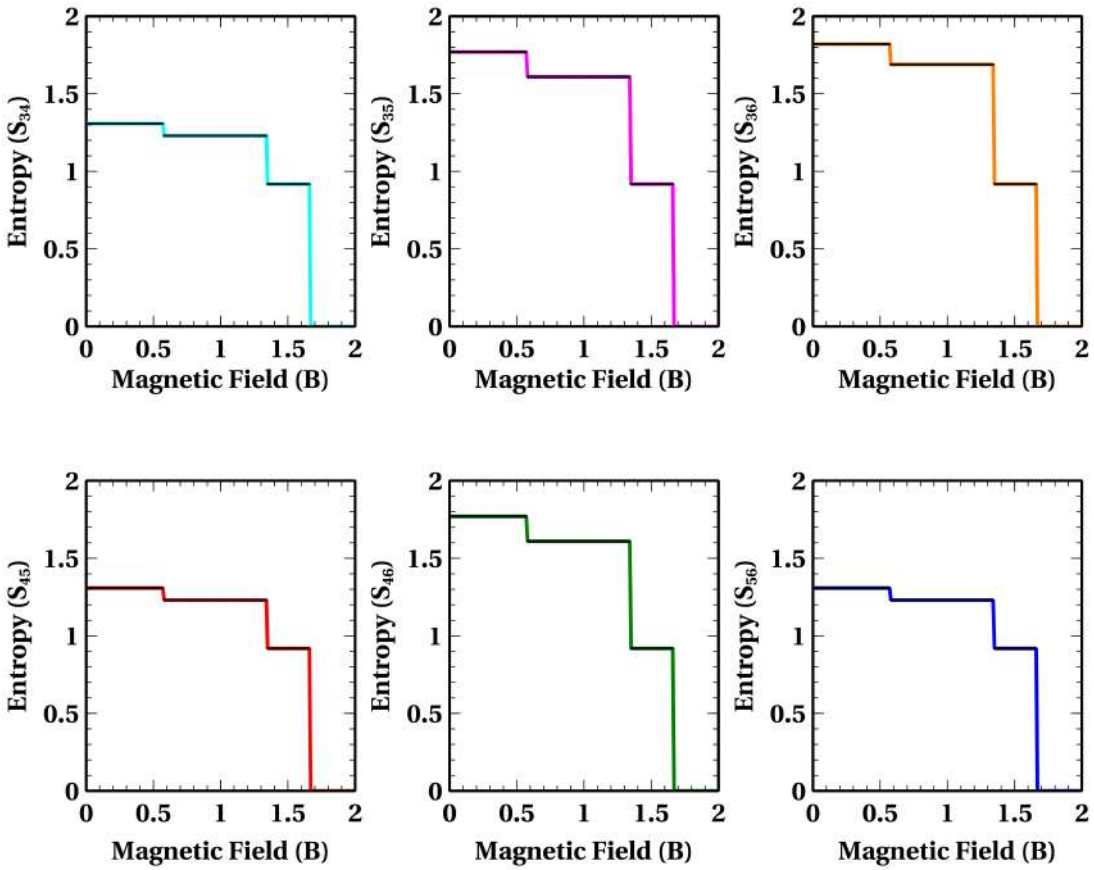
For better understanding and studying the variation analytically, let us plot the graphs.

4.3.5 Entropy vs Magnetic Field (B) for a 4 qubit AKLT Model

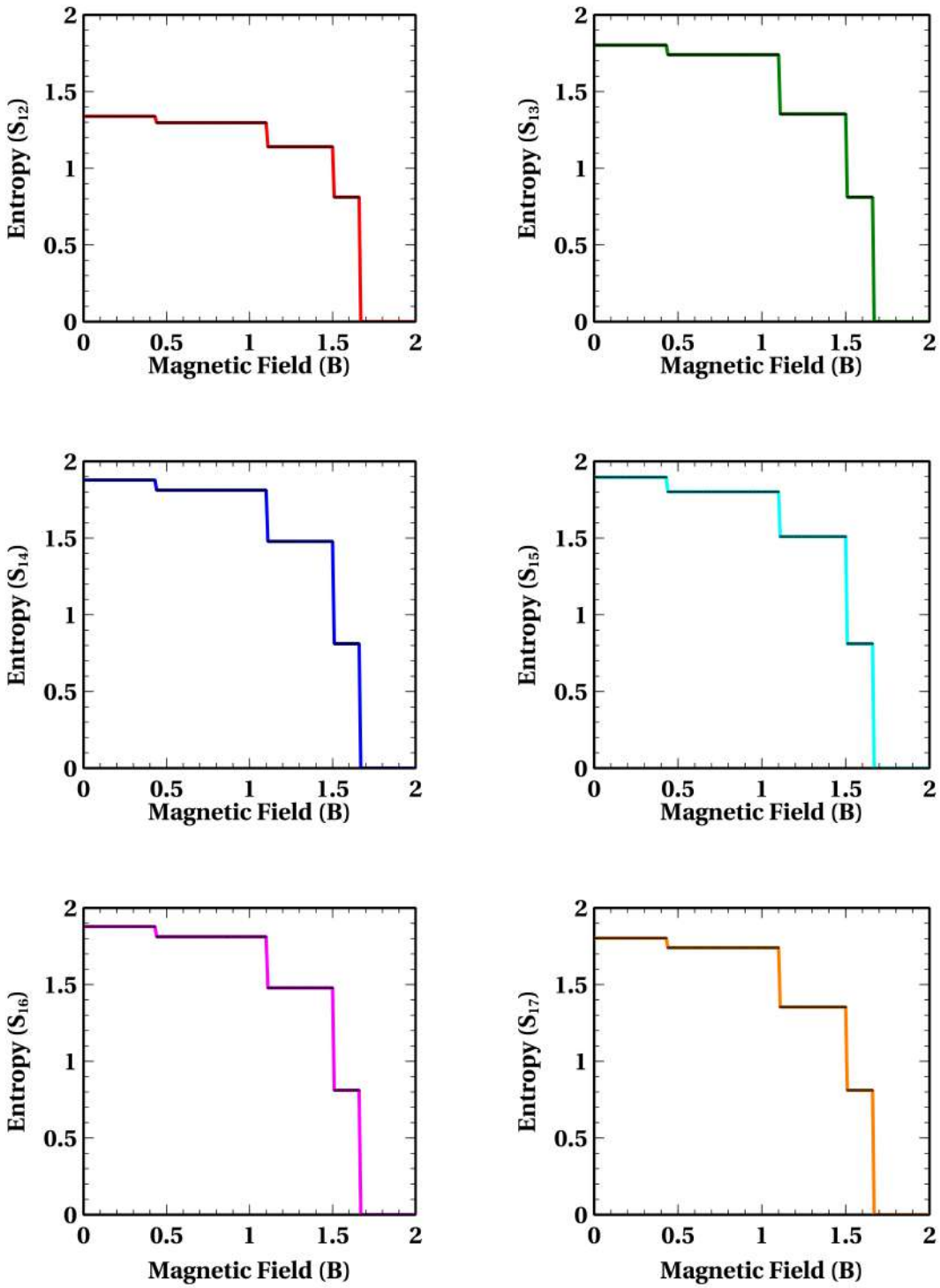


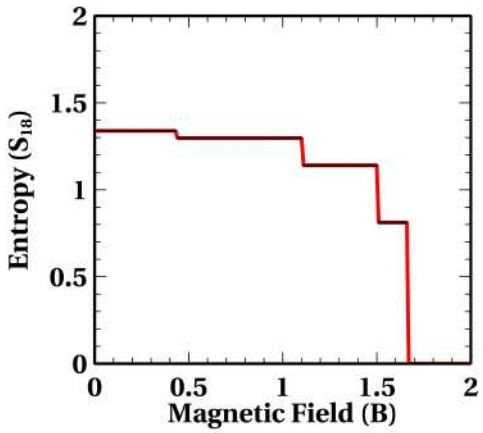
4.3.6 Entropy vs Magnetic Field (B) for a 6 qubit AKLT Model

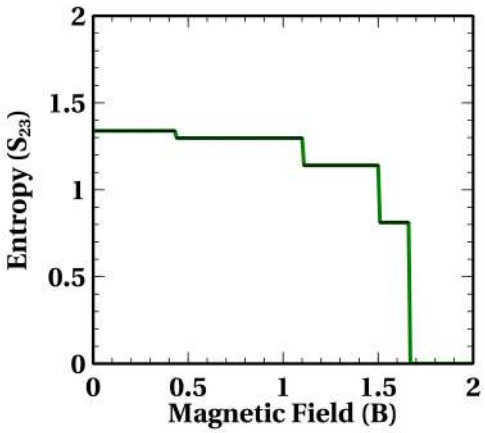


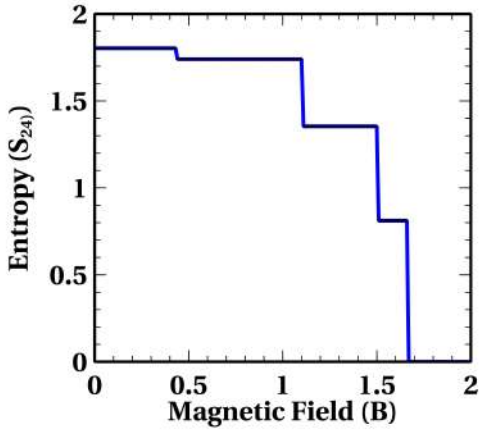

Entropy (S_{34})
Entropy (S_{35})
Entropy (S_{36})
Entropy (S_{45})
Entropy (S_{46})
Entropy (S_{56})
Magnetic Field (B)
0
0.5
1
1.5
2

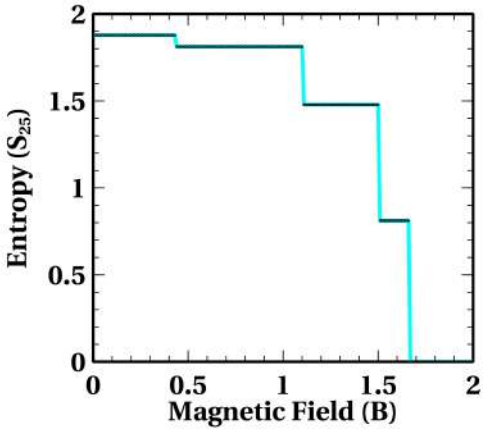
4.3.7 Entropy vs Magnetic Field (B) for a 8 qubit AKLT Model

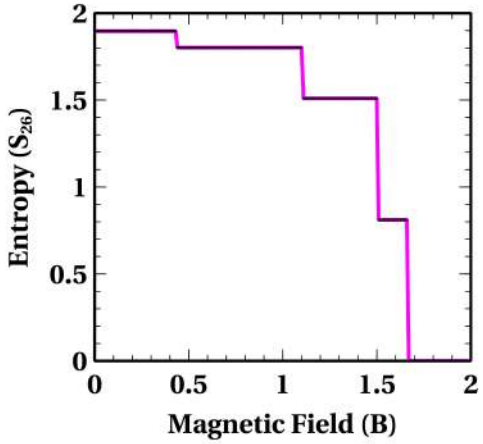


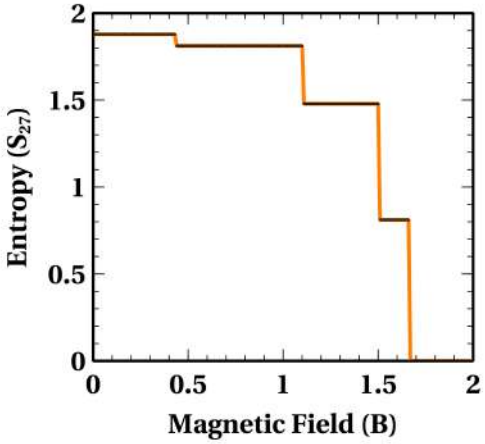

Entropy (S_{18})
Magnetic Field (B)
0
0.5
1
1.5
2

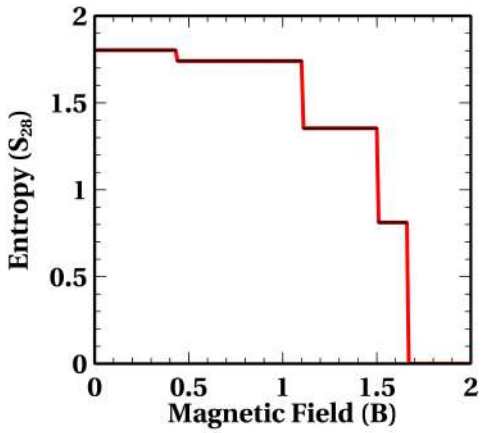

Entropy (S_{23})
Magnetic Field (B)
0
0.5
1
1.5
2

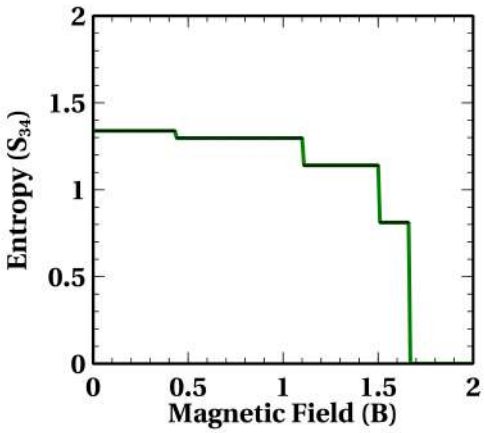

Entropy (S_{24})
Magnetic Field (B)
0
0.5
1
1.5
2

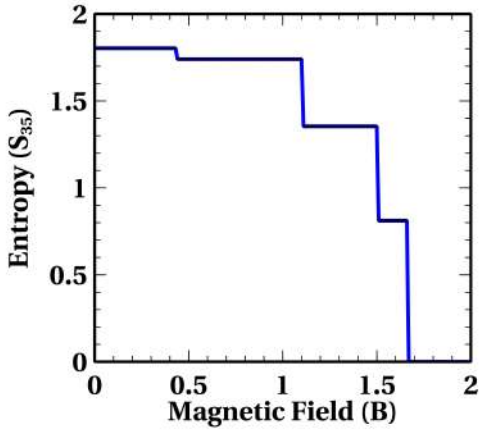

Entropy (S_{25})
Magnetic Field (B)
0
0.5
1
1.5
2

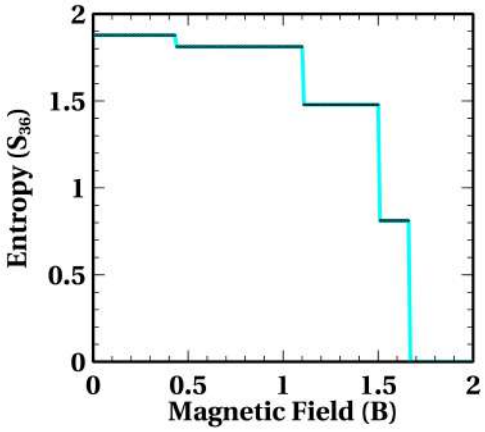

Entropy (S_{26})
Magnetic Field (B)
0
0.5
1
1.5
2

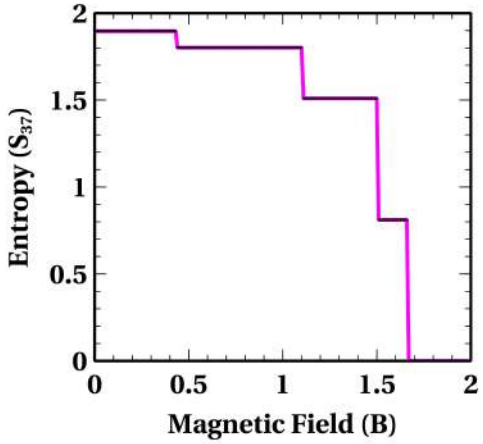

Entropy (S_{27})
Magnetic Field (B)
0
0.5
1
1.5
2

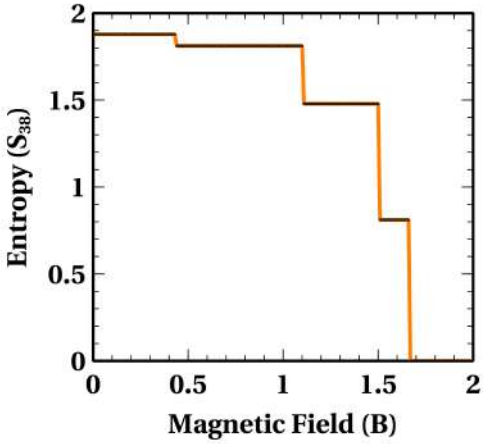

Entropy (S_{28})
Magnetic Field (B)
0
0.5
1
1.5
2

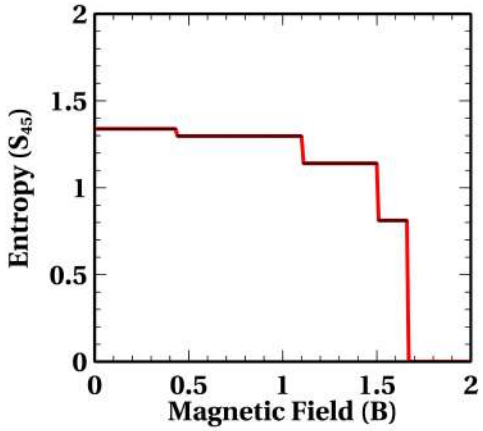

Entropy (S_{34})
Magnetic Field (B)
0
0.5
1
1.5
2

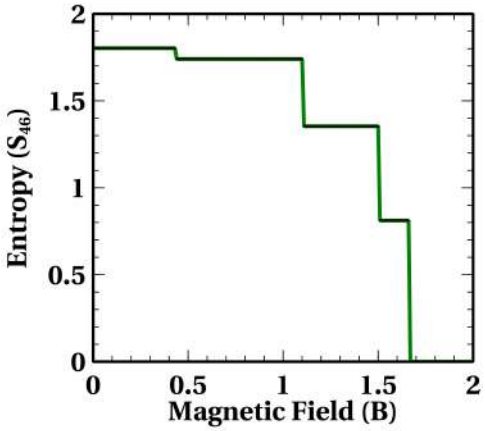

Entropy (S_{35})
Magnetic Field (B)
0
0.5
1
1.5
2

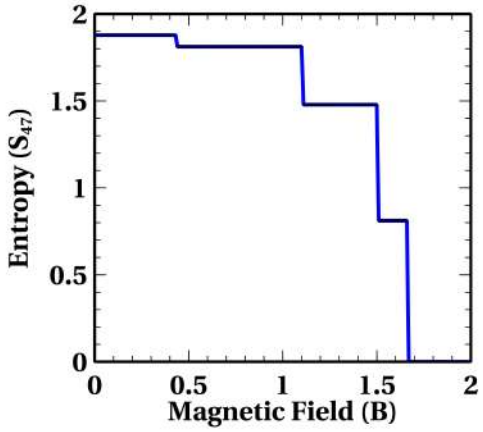

Entropy (S_{36})
Magnetic Field (B)
0
0.5
1
1.5
2

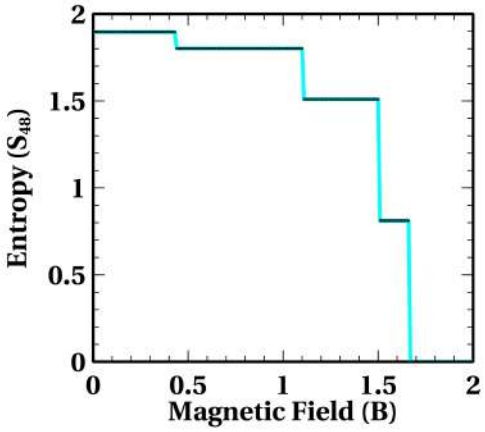

Entropy (S_{37})
Magnetic Field (B)
0
0.5
1
1.5
2

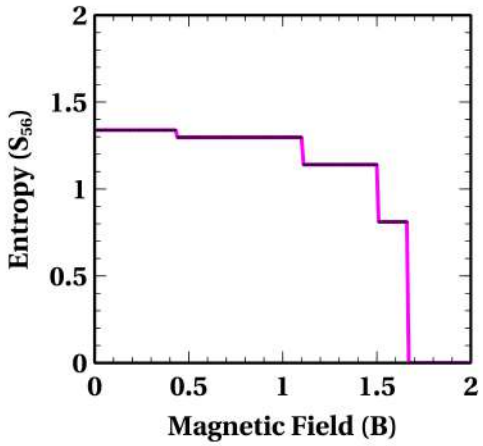

Entropy (S_{38})
Magnetic Field (B)
0
0.5
1
1.5
2

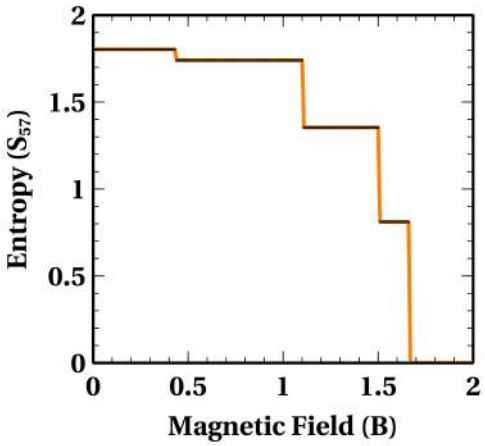

Entropy (S_{45})
Magnetic Field (B)
0
0.5
1
1.5
2

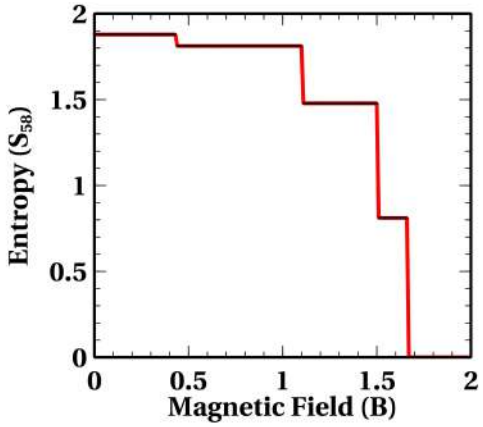

Entropy (S_{46})
Magnetic Field (B)
0
0.5
1
1.5
2

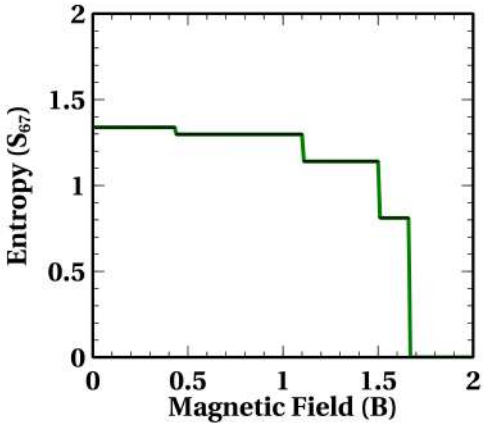

Entropy (S_{47})
Magnetic Field (B)
0
0.5
1
1.5
2

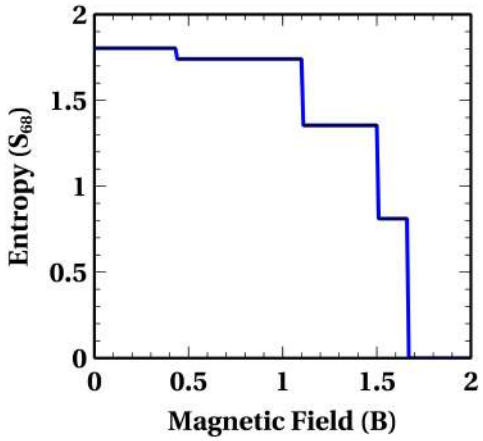

Entropy (S_{48})
Magnetic Field (B)
0
0.5
1
1.5
2

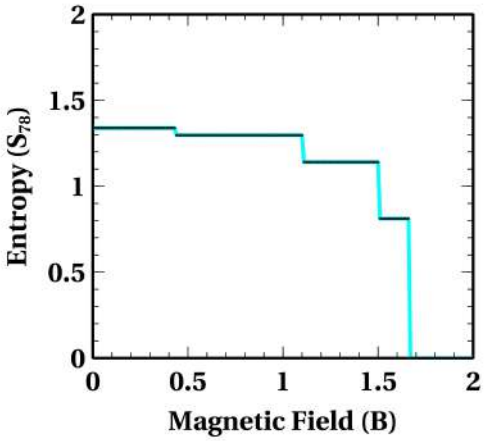

Entropy (S_{56})
Magnetic Field (B)
0
0.5
1
1.5
2


Entropy (S_{57})
Magnetic Field (B)
0
0.5
1
1.5
2

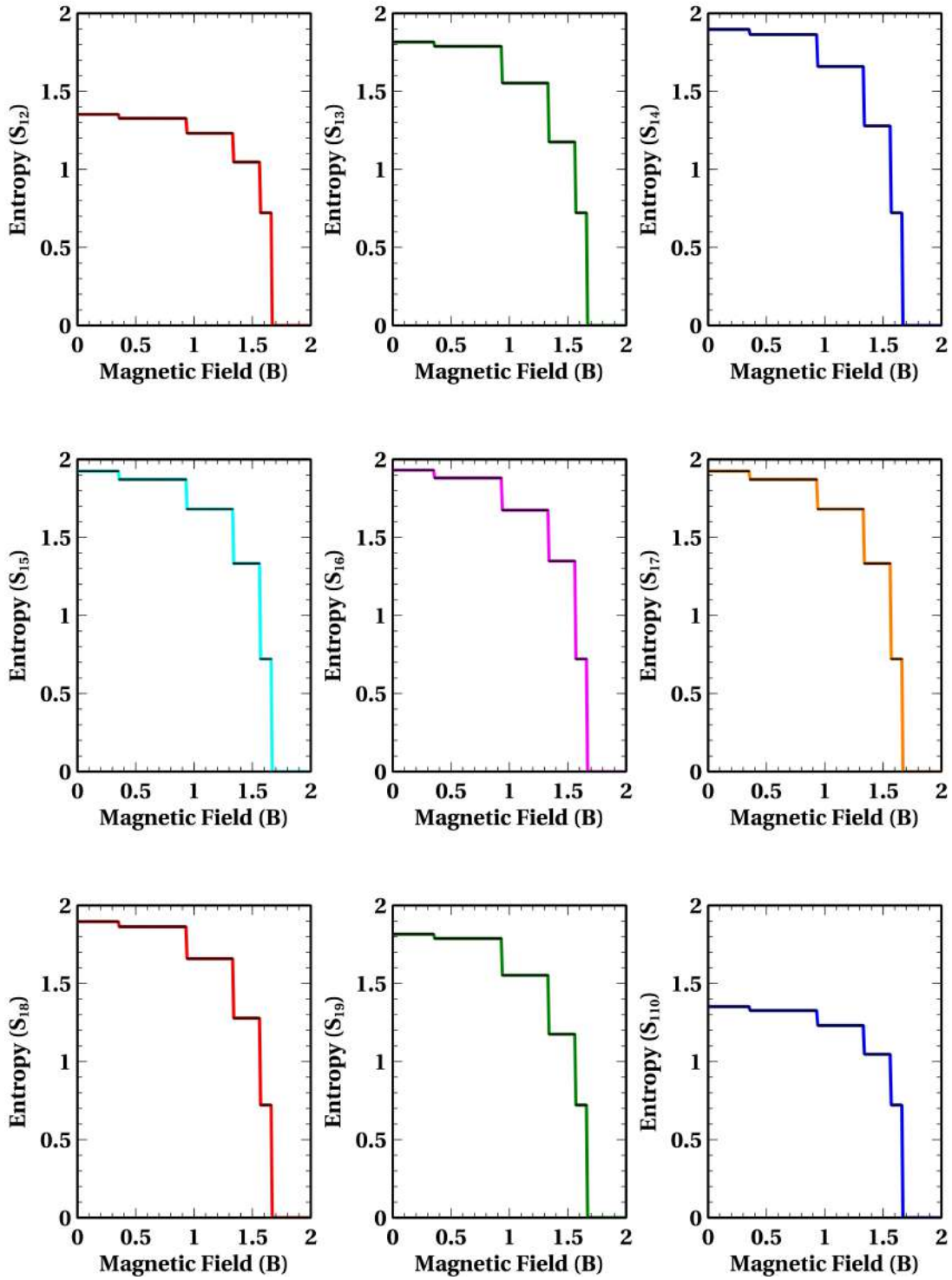

Entropy (S_{58})
Magnetic Field (B)
0
0.5
1
1.5
2

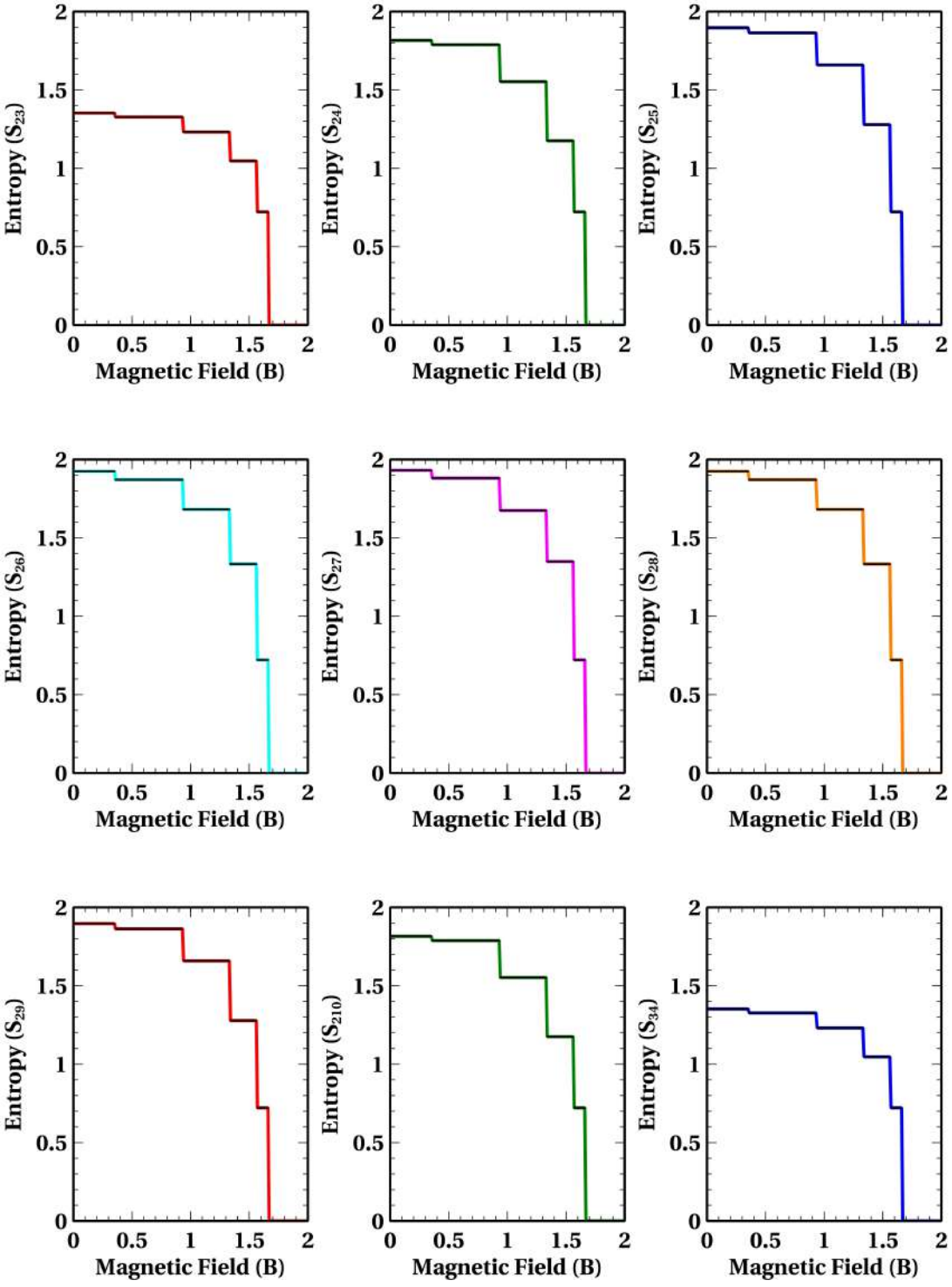

Entropy (S_{67})
Magnetic Field (B)
0
0.5
1
1.5
2

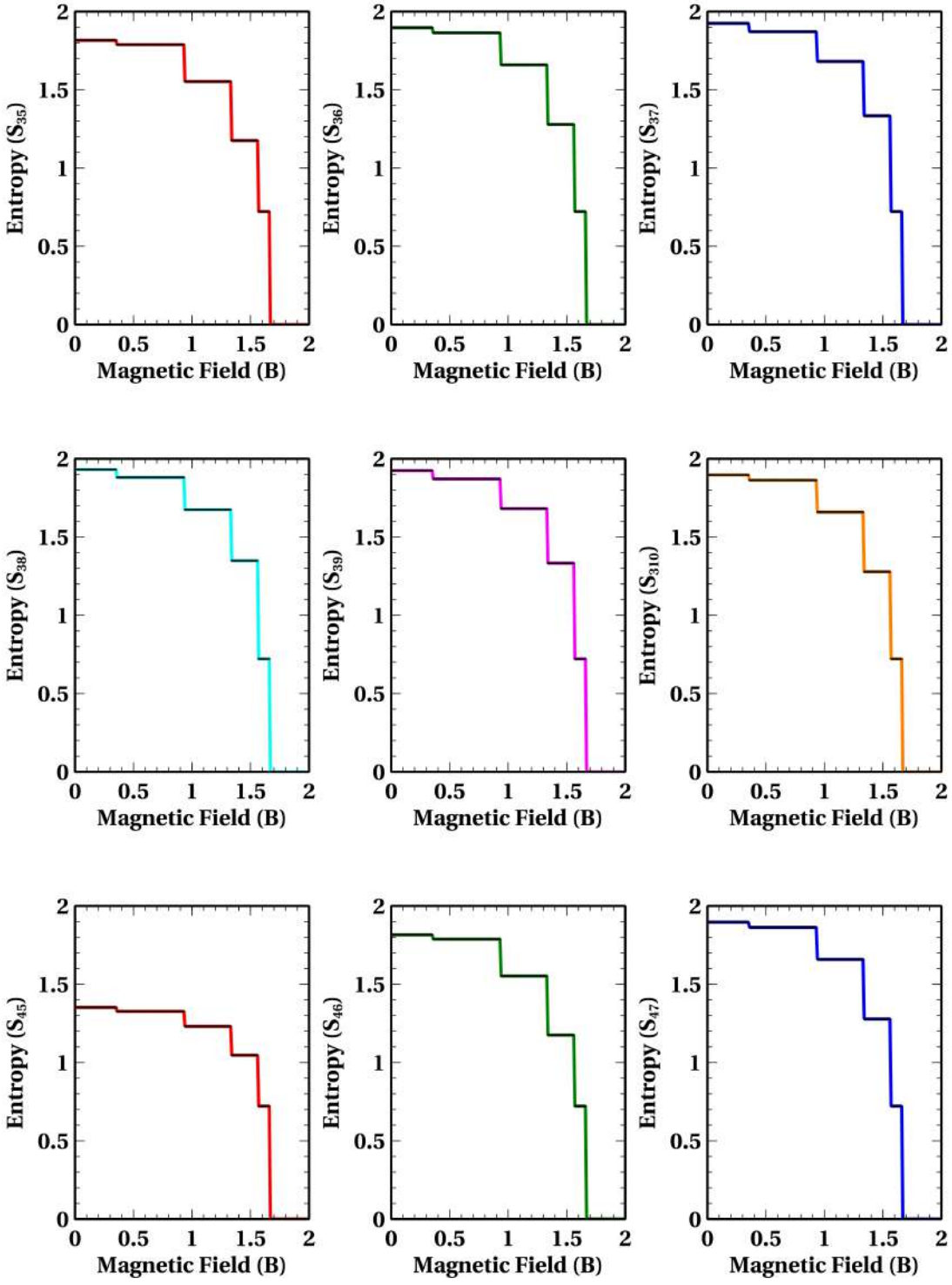

Entropy (S_{68})
Magnetic Field (B)
0
0.5
1
1.5
2

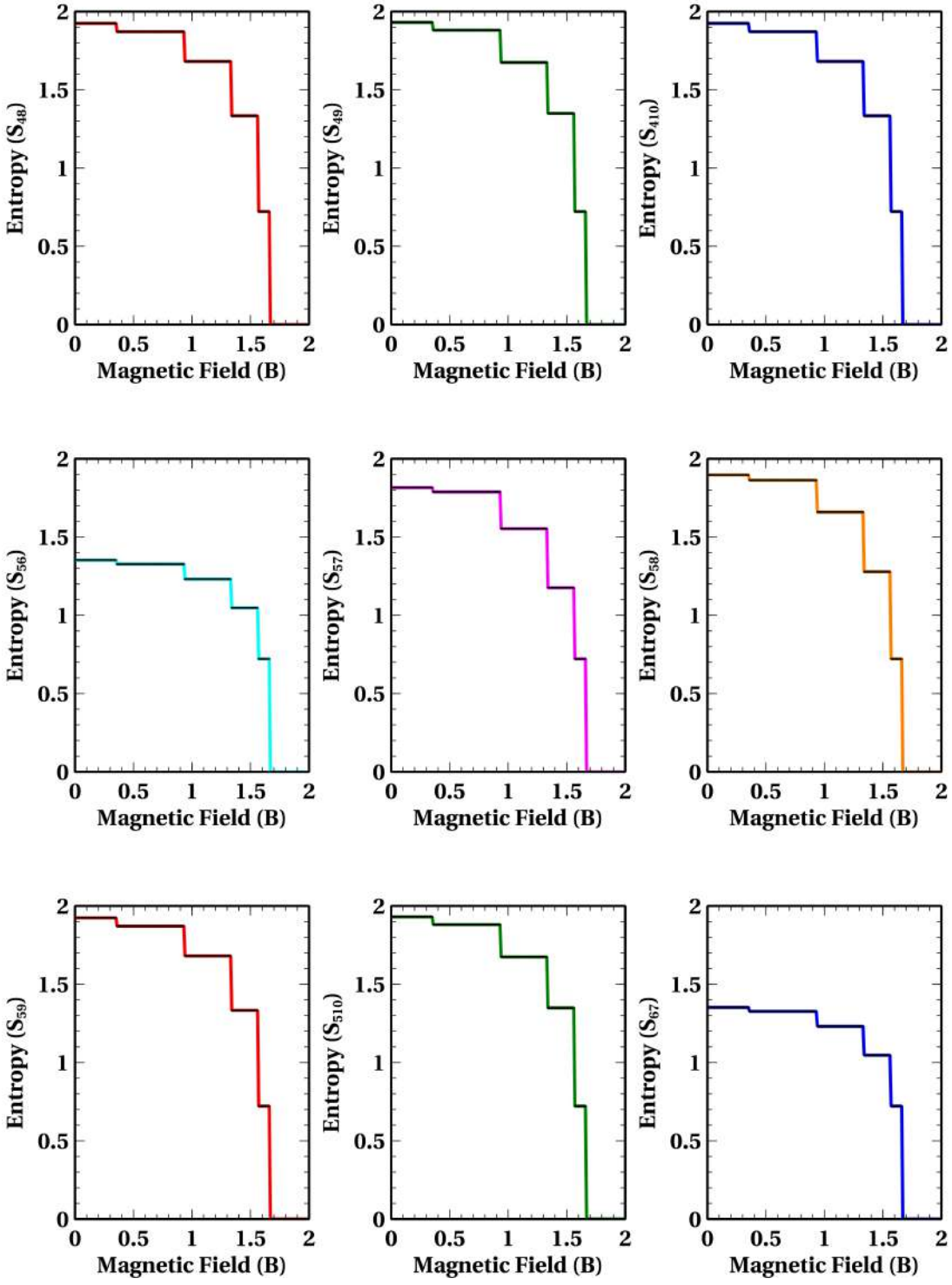

Entropy (S_{78})
Magnetic Field (B)
0
0.5
1
1.5
2

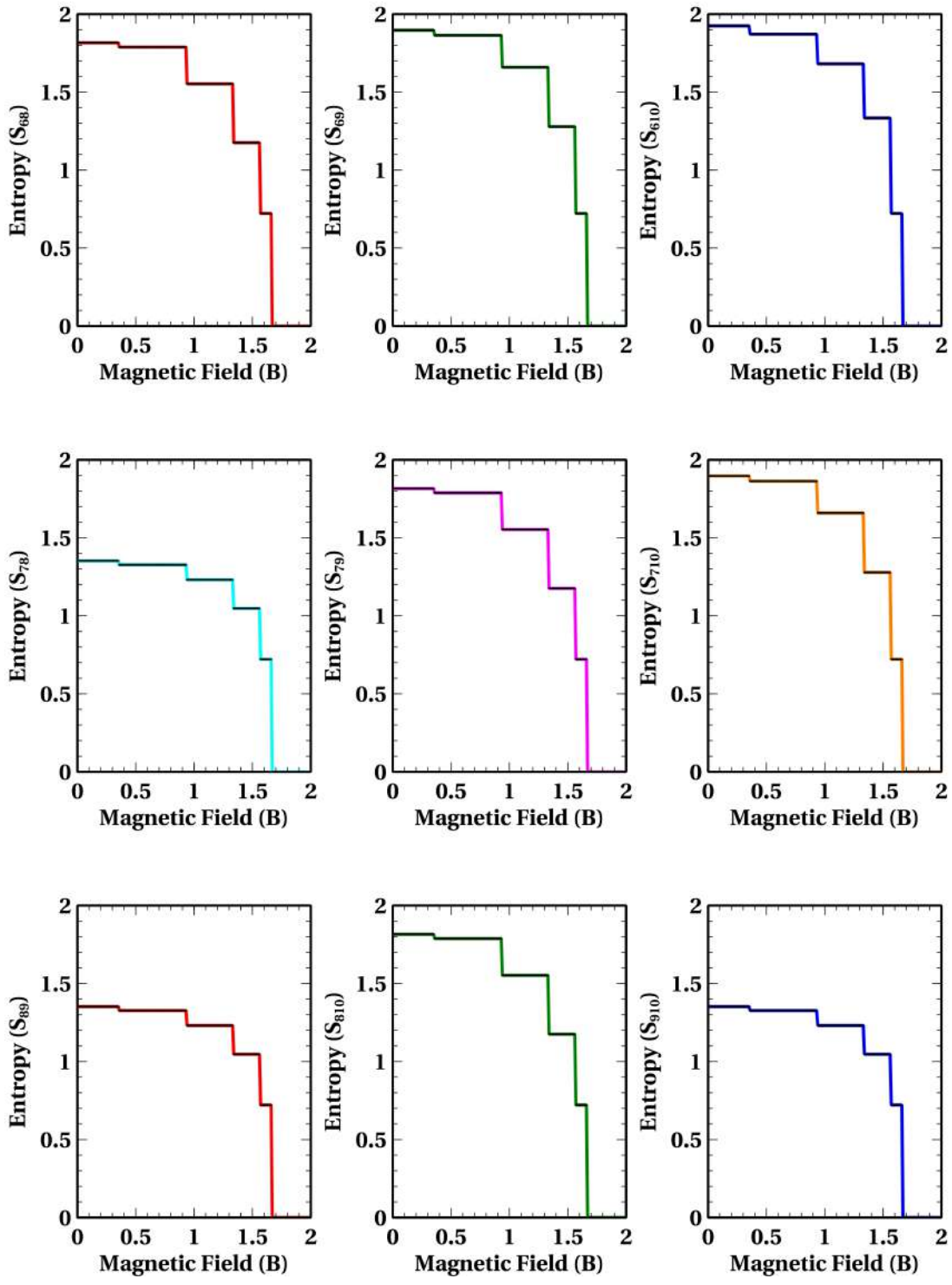
4.3.8 Entropy vs Magnetic Field (B) for a 10 qubit AKLT Model




Entropy (S_{23})
Entropy (S_{24})
Entropy (S_{25})
Entropy (S_{26})
Entropy (S_{27})
Entropy (S_{28})
Entropy (S_{29})
Entropy (S_{210})
Entropy (S_{34})
Magnetic Field (B)
0
0.5
1
1.5
2


Entropy (S_{35})
Entropy (S_{36})
Entropy (S_{37})
Entropy (S_{38})
Entropy (S_{39})
Entropy (S_{310})
Entropy (S_{45})
Entropy (S_{46})
Entropy (S_{47})
Magnetic Field (B)
0
0.5
1
1.5
2


Entropy (S_{48})
Entropy (S_{49})
Entropy (S_{410})
Entropy (S_{56})
Entropy (S_{57})
Entropy (S_{58})
Entropy (S_{59})
Entropy (S_{510})
Entropy (S_{67})
Magnetic Field (B)
0
0.5
1
1.5
2



So it is clear that the entropy vs magnetic field (B) is the opposite of that for interaction energy (J). Also, as the number of qubits increase, the width of the transition decreases. Also, we can see that the entropy is greater than one, which is

why our maximum value of concurrence is 0.5. The transitions are because of the change in ground-state eigenfunction. Also, the transition points can be shifted by changing the value of the interaction energy (J), similar to the previous cases.

Chapter 5

Chapter Summary and Future Directions

5.1 Chapter 1

In this chapter, we studied the one-dimensional Ising Model in open and periodic chain conditions for transverse and longitudinal fields. Initially, we defined the Hamiltonian using the tensor product approach, but that made computation difficult, so we defined a new method called the bit operation approach, which considers the operation of Pauli matrices on qubits or specific lattice sites. After building the Hamiltonian, we studied the eigenvalue spectrum for varying interaction energy (J) and magnetic field (B), which we observed is linear for the magnetic field along the longitudinal direction and nonlinear for the magnetic field along the transverse direction, which is because of the commutation relation of σ_z and σ_x matrices that are the constituents of the Ising Model.

We also observed the variation of ground state energy for different numbers of qubits, in which we found that the ground state energy for variation in interaction energy increases in steps, except for the three-qubit system, where it first increases and then decreases. For the variation in the magnetic field, we observed that all the energies are almost constant while the odd number of qubits has more energy than consecutive even numbers of qubits (i.e. a 3-qubit system has higher ground state energy than a 4-qubit system, a 5-qubit system has higher ground state energy than a 6-qubit

system, and so on).

5.2 Chapter 2

In this chapter, we studied the one-dimensional AKLT Model (spin-1 model) in periodic chain conditions for the varying interaction energy (J) and the magnetic field (B). First, we studied the one-dimensional Heisenberg model and used that concept to understand the Majumdar-Ghosh model. Then we learned the concept of valence bond state in the Majumdar-Ghosh Model, and then used that concept to derive our desired AKLT Hamiltonian. Then we computed the Hamiltonian using the tensor product approach, but, due to the presence of linear as well as quadratic matrix operations, it became an extremely difficult task, so we again used our bit operation approach to compute the Hamiltonian.

After building the Hamiltonian, we studied the eigenvalue spectrum for varying interaction energy and magnetic field and observed that the variation is linear, as σ^2 and σ_z commute with each other. Then we also observed the variation of the ground state eigenvalue for different numbers of qubits, which first remains constant and then it decreases for all numbers of qubits, also the transition points (points where energy starts to decrease) are different.

5.3 Chapter 3

Earlier, we had formulated the Hamiltonian, but the computational time was more for higher number of qubits due to the linear and quadratic matrix operations. So, in order to reduce the computational time, we worked on optimization methods such as using DSYEVX instead of DSYEV (which reduced the computational time by almost 2-3 times), Sparsification (which reduced the computational time by almost 5-6 times), CPU and GPU parallel programming (which we used to reduce the computational time for entanglement studies like concurrence and entropy).

We also did a comparison of computation time for building the Hamiltonian and computing the eigenvalues and eigenvectors using different smart approaches, and

reduced the time significantly by a factor of 5-6 times, which is very significant when we are dealing with the higher number of qubits.

5.4 Chapter 4

After optimizing our program, we did entanglement studies in this chapter using concurrence and entropy for varying interaction energy (J) and magnetic field (B) on the AKLT Model. Firstly, we observed that the state can be written as a valence bond state (singlet), which was the claim by AKLT.

As we increased the number of qubits, the transitions also increased for concurrence vs interaction energy (J) and magnetic field (B). The width of transitions decreases with an increase in the number of qubits. Also, the value of concurrence decreases with an increase in the number of qubits that satisfy the monogamy of entanglement. We also observed that the maximum value of concurrence is 0.5, which is for a four-qubit state. We also observed that the transition states obey some patterns, which we are working on currently, as it is a deep problem. Also, the concurrence vs magnetic field (B) shows the opposite trend to that of concurrence vs interaction energy (J).

As we increased the number of qubits, the transitions also increased for entropy vs interaction energy (J) and magnetic field (B). Also, the width of transitions decreases with an increase in the number of qubits. Also, the value of entropy for a particular transition decreases with an increase in the number of qubits that satisfy the monogamy of entanglement. We also observed that the value of entropy for all the states is greater than 1, which gives us the reason for the maximum value of concurrence to be 0.5 only.

5.5 Future Directions

We have done numerical calculations for the AKLT Model and done the preliminary analytical calculations to obtain ground states of the AKLT model as valence bond states (singlets). However, the analytical behavior of concurrence and von Neumann entropy for the ground state is an interesting topic for further study.

While working with concurrence, we got an idea for entanglement in qudits, for which we studied the mathematical aspects of Wootters concurrence in detail and tried to extend the spin-flip for d-dimensional qudits. We found an interesting technique that allows us to extend the spin-flip matrix for qudits. Following that pattern, we worked on computing the concurrence of the current problem under consideration. For general qudits and up to 8-dimensional qudits, we have verified the results (i.e. for separable states, the concurrence being zero, and for maximally entangled states, the concurrence being 1 and concurrence for all the rest states is somewhere between 0 and 1). Also, the properties of the spin flip matrix remain the same as for two-qubits (i.e. eigenvalues and eigenvectors are the same). Currently we are working on the generalization of this to create an analytical framework for N particle, d dimensional qudit systems.

Appendix A

Schmidt Decomposition

A.1 Schmidt Decomposition

Schmidt Decomposition is a very Important tool in quantum information and quantum computing [23]. The Schmidt decomposition (named after its originator, Erhard Schmidt) refers to a particular way of expressing a vector in the tensor product of two inner product spaces. It has numerous applications in quantum information theory, for example, in entanglement characterisation and state purification.

Let us consider a composite state $|\psi\rangle_{AB}$ with two subsystems A and B, with basis $|i\rangle$ and $|j\rangle$ in the Hilbert spaces $\mathcal{H}_A$ (dimension N) and $\mathcal{H}_B$ (dimension M), respectively. Then the state $|\psi\rangle_{AB}$ of the composite system AB can be written as:

$$|\psi_{AB}\rangle = \sum_{i=1}^{N}\sum_{\alpha=1}^{M} a_{ij}\,|i\rangle\,|j\rangle\,, \tag{A.1}$$

where $a_{i\alpha}$ is an $N \times M$ matrix, called the coefficient matrix.

$$\rho_{AB} = |\psi_{AB}\rangle\langle\psi_{AB}| = (\sum_{i=1}^{N}\sum_{j=1}^{M} a_{ij}|i\rangle \otimes |j\rangle)(\sum_{k=1}^{N}\sum_{l=1}^{M} a_{kl}^{*}\langle k| \otimes \langle l|)$$
$$= \sum_{i,j,k,l} a_{ij}a_{kl}^{*}|i\rangle\langle k| \otimes |j\rangle\langle l| \tag{A.2}$$

Now, let us take the partial trace with respect to the B subsystem

$$\rho_A = tr_B(|\psi_{AB}\rangle\langle\psi_{AB}|)$$
$$= \sum_{p}\sum_{i,j,k,l} a_{ij}a_{kl}^{*}|i\rangle\langle k| \otimes \langle p|j\rangle\langle l|p\rangle$$
$$= \sum_{p}\sum_{i,k} a_{ip}a_{kp}^{*}|i\rangle\langle k|$$
$$\rho_A = \sum_{i,k}\sum_{p} a_{ip}a_{kp}^{*}|i\rangle\langle k| \tag{A.3}$$
$$\rho_A = AA^{\dagger} \tag{A.4}$$

Similarly, one can easily calculate $\rho_B = A^{\dagger}A$.

Now, from Singular Value Decomposition, we can write for any matrix A, with U and V being unitary matrices of appropriate dimensions,

$$A = UDV^{\dagger}.$$

So,

$$AA^{\dagger} = UDV^{\dagger}VDU^{\dagger}$$
$$AA^{\dagger} = UD^2U^{\dagger} \tag{A.5}$$

So, $\sqrt{\lambda_i}$ are the eigenvalues of $AA^{\dagger}$.
Similarly, $A^{\dagger}A = VD^2V^{\dagger}$.
So, $\sqrt{\lambda_i}$ are the eigenvalues of $A^{\dagger}A$.

The Schmidt decomposition can be understood using a pictorial representation as,

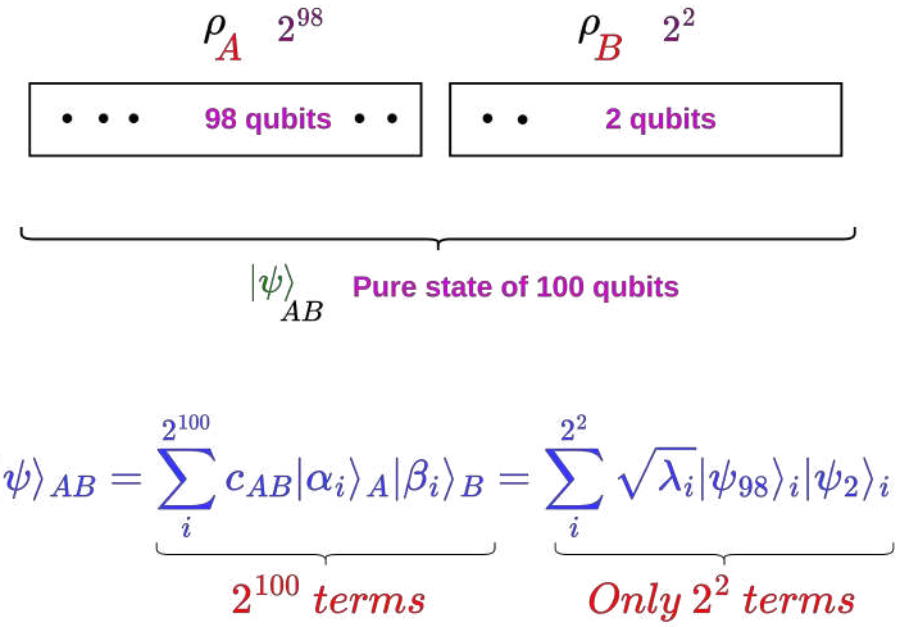


Figure A.1: Schmidt Decompostion

So if we have a 100 qubits state and we divide it into two subsystems one consisting 98 qubits and another consisting 2 qubits, then we do not need to calculate all the 2^{98} eigen values of the first subsystem because the eigenvalues are common to both the subsystems, so we can calculate them simply using 2 qubit state.

Because of this property, we can write any pure state as [23],

$$|\psi\rangle_{AB} = \sum_i \lambda_i \, |i\rangle_A \, |i\rangle_B \, , \tag{A.6}$$

where λ_i are non-negative real numbers, satisfying $\sum_i \lambda_i^2 = 1$ known as Schmidt co-effecients. The bases $|i\rangle_A$ and $|i\rangle_B$ are called Schmidt bases for A and B, respectively, and the number of non-zero values of λ_i is called the Schmidt number of the state $|\psi\rangle_{AB}$. Now we aim to find the Schmidt coefficients and the orthonormal basis for A and B subsystems of any general state $|\psi\rangle_{AB}$. The Fortran program for this is as below,

```
1 program schmidtdec
2     implicit none
```

```
    integer:: i, j , N, k, s, num, nt, s1, M, d
    integer:: r, q, bool, flag
    real*8, allocatable ::  b(:),res1(:,:), res2(:,:)
    character(len=:), allocatable :: state
    integer, allocatable :: state_matrix(:,:), site1(:), site2(:)
    double precision, allocatable ::psi(:,:), vin(:), rdm1(:,:),
   rdm2(:,:), W1(:), WORK1(:)
    double precision, allocatable ::U(:,:), V(:,:), E(:), F(:) , W2
   (:), WORK2(:)
    real*8, allocatable ::Ia(:,:), Ib(:,:), result
    double precision, allocatable :: phi(:,:), phi1(:,:)
    character*1 :: JOBZ, UPLO
    integer ::  LDA , LWORK , INFO

    ! Define the vectors for '0' and '1'
    integer, parameter :: zero_vector(2) = [1, 0]
    integer, parameter :: one_vector(2) = [0, 1]

    ! User Input total number of bits
    print*, "Enter total no of qubits: "
    read*, s
    ! Enter total number of bits:
    !s = 4
    num = 2**s

    ! User Input number of linearly independent terms
    print*, "Enter the total number of terms: "
    read*, nt
    ! Enter total number of terms:
    !nt = 2

    ! Allocate arrays after determining the value of num
    allocate(b(nt))
    allocate(double precision :: psi(num, 1))
    allocate(character(len=s) :: state)
    allocate(state_matrix(2**s, 1))

    ! Cacluating the state matrix from the lineraly independent
```

```
    terms
     psi = 0.0d0
     do i = 1, nt
          print *, "Enter the coefficient of", i ," term : "
          read *, b(i)
          print *, "Enter the ", i ," term : "
          read *, state

          ! Initialize the state matrix with the first vector
          if (state(1:1) == '1') then
               state_matrix(1, 1) = one_vector(1)
               state_matrix(2, 1) = one_vector(2)
          else
               state_matrix(1, 1) = zero_vector(1)
               state_matrix(2, 1) = zero_vector(2)
          end if

          ! Compute the tensor product iteratively
          do j = 2, s
               call genstate(state_matrix, state(j:j))
          end do
          psi = psi + b(i) * state_matrix
     end do


     print*, "Enter the number of qubits not to be traced out : "
     read*, s1
     !s1 = 2

     ! Allocating matrices for PTRVR & DSYEV
     N = 2**s1
     LDA = N
     LWORK = 3*N - 1
     d = s - s1
     allocate(site1(0:s1-1))
     allocate(site2(0:d-1))
     allocate(vin(0:2**s-1))
     allocate(rdm1(0:2**s1-1,0:2**s1-1))
```

```
76      allocate(rdm2(0:2**d-1,0:2**d-1))
77      allocate(W1(0:2**s1-1))
78      allocate(WORK1(0:LWORK))
79      allocate(E(0:2**s1-1))
80      allocate(F(0:2**d-1))
81      allocate(V(0:2**d-1,0:2**d-1))
82      allocate(U(0:2**s1-1,0:2**s1-1))
83      allocate(res1(0:2**s1-1,1))
84      allocate(res2(0:2**d-1,1))
85      allocate(Ia(0:2**s1-1,1))
86      allocate(Ib(0:2**d-1,1))
87
88      do i = 0, s1-1
89          print*, "Enter the site indeces of qubits not to be traced
    out :"
90          read*, k
91          site1(i) = k - 1
92      end do
93
94      do i = 0, num - 1
95          vin(i) = psi(i+1,1)
96          write(90,*) i, vin(i)
97      end do
98
99      call PTRVR(s, s1, site1, vin, rdm1)
100     write(90,*) rdm1
101     U = rdm1
102
103     ! Define parameters for DSYEV
104     JOBZ = 'V'  ! Compute eigenvalues and eigenvectors
105     UPLO = 'L'  ! Upper triangular part of A is stored
106
107     ! Call DSYEV to compute eigenvalues and eigenvectors
108     call DSYEV(JOBZ, UPLO, N, U, LDA, W1, WORK1, LWORK, INFO)
109     do i = 0,2**s1-1
110         E(i) = sqrt(W1(i))
111         write(90,*) i, E(i)
112         do j = 0,2**s1-1
```

```
113             write(90,*) j,i, U(j,i)
114         end do
115     end do
116
117     do i = 0, d -1
118         print*, "Enter the site indeces of qubits not to be traced
    out :"
119         read*, k
120         site2(i) = k - 1
121     end do
122
123     call PTRVR(s, d, site2, vin, rdm2)
124     write(90,*) rdm2
125     V = rdm2
126
127     M = 2**d
128     allocate(W2(0:2**d-1))
129     allocate(WORK2(0:3*M -1))
130     ! Define parameters for DSYEV
131     JOBZ = 'V'  ! Compute eigenvalues and eigenvectors
132     UPLO = 'L'  ! Upper triangular part of A is stored
133
134     ! Call DSYEV to compute eigenvalues and eigenvectors
135     call DSYEV(JOBZ, UPLO, M, V, M, W2, WORK2, 3*M -1, INFO)
136     do i = 0,2**d -1
137         F(i) = sqrt(W2(i))
138         write(90,*) i, F(i)
139         do j = 0,2**d -1
140             write(90,*) j,i,V(j,i)
141         end do
142     end do
143
144     allocate(phi(0:2**s-1, 1))
145     allocate(phi1(0:2**s-1,1))
146
147     phi = 0.0d0
148     do i = 0, 2**s1 - 1
149         do j = 0, 2**d -1
```

```
                if (E(i) == F(j) .AND. E(i) /= 0 ) then
                    Ia(:,1) = U(:,i)
                    Ib(:,1) = V(:,j)
                    ! print*, Ia
                    ! print*, Ib
                    flag = 0.0d0
                    bool = 0.0d0
                    call MMULMR(rdm1, Ia, 2**s1, 2**s1, 1, res1)
                    call MMULMR(rdm2, Ib, 2**d, 2**d, 1, res2)
                    ! print*, res1
                    ! print*, res2
                    do r = 0, 2**s1 - 1
                        if (res1(r,1) - E(i)**2 *U(r,i) /= 0) then
                            flag = flag + 1
                        end if
                    end do
                    do q = 0, 2**d - 1
                        if (res2(q,1) - F(j)**2 *V(q,j) /= 0) then
                        bool = bool + 1
                        end if
                    end do
                    if (flag == 0 .AND. bool == 0 ) then
                        call tensor_product(U(:,i), V(:,j), phi1
    (:,1), s1, s)
                        !print*, phi1
                        phi = phi + E(i) * phi1
                    end if
                end if
            end do
        end do

        ! do i = 0, 2**s - 1
        !     write(90,*) phi(i,1)
        ! end do

        do i = 0, 2**s -1
            result = result + psi(i+1,1) * phi(i,1)
        end do
```

```
187     print*, "<phi|psi> = ", result
188
189     deallocate(psi)
190     deallocate(state)
191     deallocate(state_matrix)
192 contains
193
194     ! Subroutine to compute the tensor product
195     subroutine genstate(matrix, bit)
196         integer, intent(inout) :: matrix(:,:)
197         character(len=1), intent(in) :: bit
198         integer :: current_rows, new_rows, l
199         integer, allocatable :: temp_matrix(:,:)
200
201         ! Determine current number of rows in the matrix
202         current_rows = size(matrix, 1)
203         new_rows = current_rows * 2
204
205         ! Allocate a new matrix for the tensor product
206         allocate(temp_matrix(new_rows, 1))
207
208         ! Fill the new tensor product matrix
209         if (bit == '1') then
210             do l = 1, current_rows
211                 temp_matrix(2*l-1, 1) = matrix(l, 1) * 0  ! Corresponds to [0]
212                 temp_matrix(2*l, 1) = matrix(l, 1) * 1      ! Corresponds to [1]
213             end do
214         else
215             do l = 1, current_rows
216                 temp_matrix(2*l-1, 1) = matrix(l, 1) * 1  ! Corresponds to [1]
217                 temp_matrix(2*l, 1) = matrix(l, 1) * 0      ! Corresponds to [0]
218             end do
219         end if
220
```

```
221         ! Update the original matrix with the new tensor product
222         matrix = temp_matrix
223         deallocate(temp_matrix)
224     end subroutine genstate
225 
226     subroutine PTRVR(s,s1,site,vin,rdm)
227         implicit none
228         integer::s, s1,s2
229         real*8::trace
230         real*8,dimension(0:2**s-1)::vin
231         real*8,dimension(0:2**s1-1,0:2**s1-1)::rdm
232         integer::ii,i1,a,i0,ia,j1,oo,k1,x1,y1,j,t1,t2,z1,i
233         integer,dimension(0:s1-1)::site,site2
234         integer,dimension(0:s1-1)::bin
235         integer,dimension(0:(2**(s-s1))*(2**s1)-1)::ind
236         s2=s-s1
237         x1=0
238         y1=0
239         ind=0
240         do j1=2**s1-1,0,-1
241             do ii=0,2**s-1
242                 a=ii
243                 do i1=0,s1-1,1
244                     i0=site(i1)
245                     call DTOBONEBIT(a,ia,i0,s)
246                     site2(i1)=ia
247                 enddo
248                 call DTOB(j1,bin,s1)
249                 oo=1
250                 do k1=0,s1-1,1
251                     oo=oo*(bin(k1)-site2(k1))
252                 end do
253                 if(abs(oo)==1) then
254                     ind(x1)=ii
255                     x1=x1+1
256                 end if
257             end do
258         end do
```

```
259         rdm=0.0d0
260         do t1=0,2**s1-1,1
261             do t2=0,2**s1-1,1
262                 do z1=0,2**s2-1,1
263                     rdm(t1,t2)=rdm(t1,t2)+vin(ind((2**s2)*t1+z1)) &
264                     *vin(ind((2**s2)*t2+z1))
265                 enddo
266             end do
267         end do
268     end subroutine
269
270     subroutine tensor_product(Ia, Ib, phi1, s1,s)
271         implicit none
272         integer :: i, j, k, l, s1, s
273         double precision, dimension(0:2**s1-1,1) :: Ia
274         double precision, dimension(0:2**(s-s1)-1,1) :: Ib
275         double precision, dimension(0:2**s-1,1) :: phi1
276
277         ! Initialize the output matrix C to zero
278         phi1 = 0.0d0
279
280         !print*, Ia, Ib
281
282         ! Compute the tensor product
283         do i = 0, 2**s1-1
284             do k = 0, 2**(s-s1)-1
285                 phi1((i)*2**(s-s1) + k, 1) = Ia(i, 1) * Ib(k, 1)
286             end do
287         end do
288     end subroutine tensor_product
289
290     subroutine DTOB(m,tt,s)
291         implicit none
292         integer::s
293         integer,dimension(0:s-1)::tt
294         integer::m,k,a2
295         tt=0
296         a2=m
```

```
297         do k = 0,s-1,1
298             tt(s-k-1) = mod(a2,2)
299             a2 = a2/2
300             if (a2== 0) then
301                 exit
302             end if
303         end do
304     end subroutine
305
306     subroutine DTOBONEBIT(m,ia,i0,s)
307         implicit none
308         integer*4::s
309         integer,dimension(0:s-1)::tt
310         integer::m,ia,i0
311         call DTOB(m,tt,s)
312         ia=tt(i0)
313     end subroutine
314
315     subroutine MMULMR(A,B,m,n,p,C)
316         implicit none
317         integer::n,m,l,i,j,k,p
318         real*8::A(0:m-1,0:n-1),B(0:n-1,0:p-1),C(0:m-1,0:p-1),temp
319         ! print*, A
320         ! print*, B
321         do i=0,m-1,1
322             do j=0,p-1,1
323                 temp = 0.0d0
324                 do k=0,n-1,1
325                     temp=temp+A(i,k)*B(k,j)
326                 end do
327             C(i,j) = temp
328             end do
329         end do
330     end subroutine
331 end program schmidtdec
```

This is the general program for calculating the Schmidt coefficients and the orthonor-

mal basis for any pure state of any general qubit. One can understand this code using the flow chart given below,

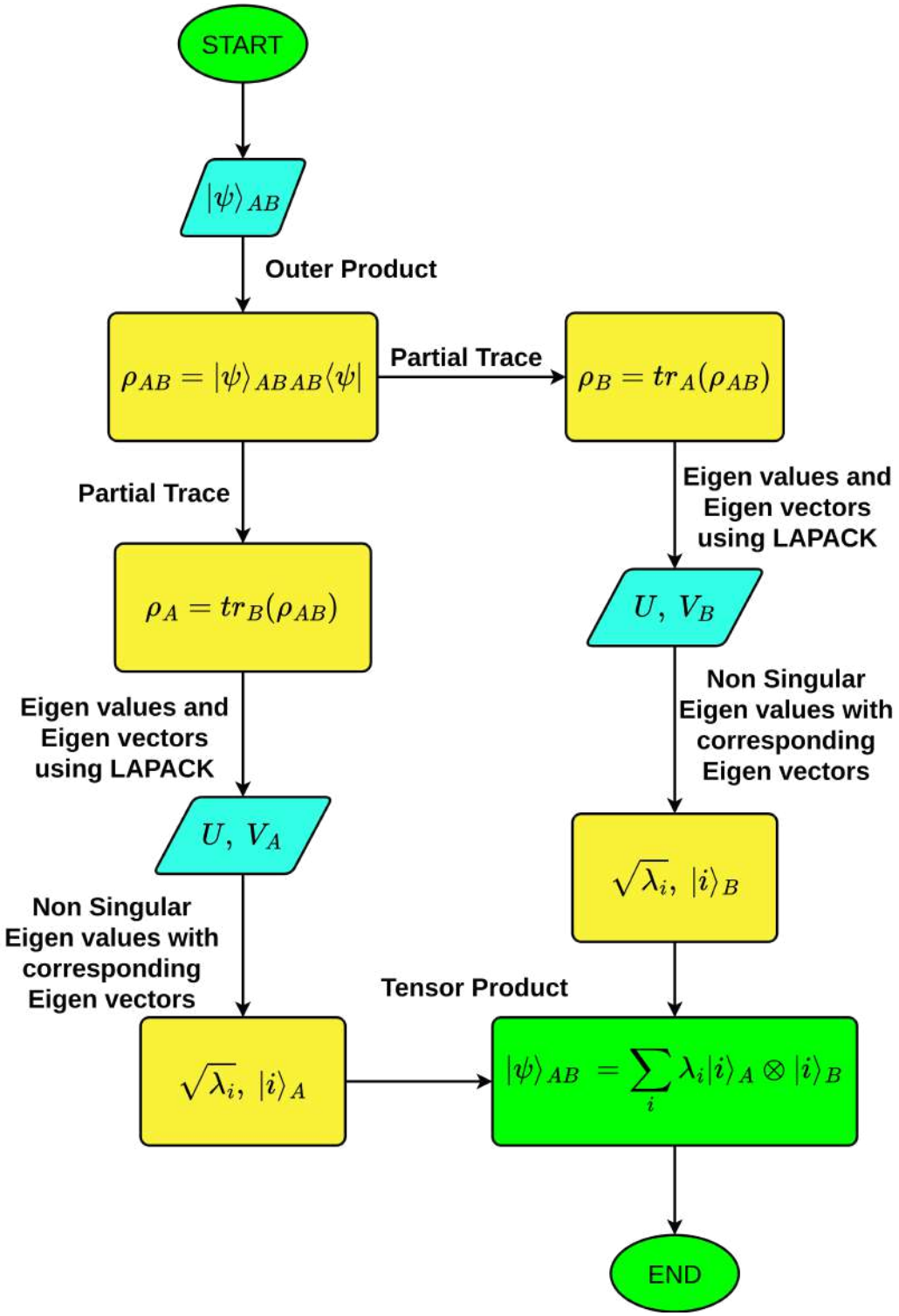


Figure A.2: Flowchart for the Schmidt decomposition procedure

Appendix B

Trit Operation Approach

B.1 Solving any given Hamiltonian using Trit Operation Approach

Let us define a similar (just taken for implementing the trit operation approach) Hamiltonian ($\mathcal{H}$) as,

$$\mathcal{H} = \sum_i \vec{S}_i \vec{S}_{i+1} + \frac{1}{3}\left(\sum_i \vec{S}_i \vec{S}_{i+1}\right)^2, \tag{B.1}$$

where $\vec{S}_i$ is a spin one operator acting on i^{th} spin, where $\vec{S}_i$ is defined as,

$$\vec{S}_i = S_i^x \hat{x} + S_i^y \hat{y} + S_i^z \hat{z}, \tag{B.2}$$

where $S_i^x,\ S_i^y$ and S_i^z are Gell-Mann Matrices [33].

$$S_i^x = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, S_i^y = \begin{bmatrix} 0 & -i & 0 \\ i & 0 & -i \\ 0 & i & 0 \end{bmatrix}, S_i^z = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix}, \tag{B.3}$$

and these Gell-Mann matrices satisfy the commutation relation,

$$[S_i^a, S_j^b] = i\epsilon_{abc} S_j^c \delta_{ij}. \tag{B.4}$$

where, ϵ_{abc} is the Levi Civita symbol

$$\epsilon_{abc} = \begin{cases} +1 & \text{if (a,b,c)} \equiv \text{(1,2,3),(2,3,1),(3,1,2)} \\ -1 & \text{if (a,b,c)} \equiv \text{(3,2,1),(1,3,2),(2,1,3)} \\ 0 & \text{if a = b, b = c, c = a} \end{cases}, \tag{B.5}$$

and, i and j represent the position of spin, and a,b, and c are the Gell-Mann matrices (i.e. x, y, z).

Since we know the spin for each pair, s = 1, we can calculate the number of basis states [34] as,

$$N = 2s + 1, \tag{B.6}$$

Substitute the value of s in Eq. [B.6], the total number of basis states will be 3. The basis states for this system are $|-1\rangle$, $|0\rangle$, and $|1\rangle$, which are the **eigenvectors** of S_z with eigenvalues 1, 0, and -1, respectively, which form the computational basis for qutrit systems. For simplicity we will represent these basis states as $|0\rangle$, $|1\rangle$ and $|2\rangle$ respectively, where

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, |1\rangle = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, |2\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}. \tag{B.7}$$

B.1.1 Building the Hamiltonian and Computing Eigenvalues

We have discussed this in detail for qubit (spin-$\frac{1}{2}$) system used in the Ising model and the AKLT model, where we understood the tensor product approach and the difficulties in that method, and thus we defined a new method to build the Hamiltonian called the bit operation approach. Now, if we take our sample Hamiltonian that uses qutrit (spin-1) system. For a qubit, the dimension of an operator is 2×2, but for a qutrit, the dimension of an operator is 3×3, and as we increase the number of qutrits, it becomes very difficult for us to build the Hamiltonian using tensor product approach, thus we define a new method called trit operation approach.

B.1.1.1 Trit Operation Approach

Before learning the method, let us look into the operation of Gell-Mann matrices on the qutrits. One can simply calculate these operations using matrix multiplication, we implement this using a Python exercise that one can do to compute these operations.

```
#Importing Libraries
import numpy as np
from scipy import linalg
import math
from IPython.display import display, Math
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore")

#Defining Spin 1-Operators I=S^0, X=S^x, Y=S^y, Z=S^z
I = np.eye(3)
X = np.array([[0, 1, 0], [1, 0, 1], [0, 1, 0]])
Y = np.array([[0, -1j, 0], [1j, 0, -1j], [0, 1j, 0]])
Z = np.array([[1, 0, 0], [0, 0, 0], [0, 0, -1]])

#Defining State
zero = [[1],[0],[0]]
one = [[0],[1],[0]]
two = [[0],[0],[1]]

#Defining a Function for Printing Matrix into Latex Format
def print_matrix(array):
    matrix = ''
    for row in array:
        try:
            for number in row:
                matrix += f'{number}&'
        except TypeError:
            matrix += f'{row}&'
        matrix = matrix[:-1] + r'\\'
    display(Math(r'\begin{bmatrix}'+matrix+r'\end{bmatrix}'))
```

```
32
33 display(Math(r"S^x|0\rangle ="))
34 print_matrix(np.matmul(X,zero))
35 display(Math(r"S^x|1\rangle ="))
36 print_matrix(np.matmul(X,one))
37 display(Math(r"S^x|2\rangle ="))
38 print_matrix(np.matmul(X,two))
39 display(Math(r"S^y|0\rangle ="))
40 print_matrix(np.matmul(Y,zero))
41 display(Math(r"S^y|1\rangle ="))
42 print_matrix(np.matmul(Y,one))
43 display(Math(r"S^y|2\rangle ="))
44 print_matrix(np.matmul(Y,two))
45 display(Math(r"S^z|0\rangle ="))
46 print_matrix(np.matmul(Z,zero))
47 display(Math(r"S^z|1\rangle ="))
48 print_matrix(np.matmul(Z,one))
49 display(Math(r"S^z|2\rangle ="))
50 print_matrix(np.matmul(Z,two))
51
```

Once we are done with this exercise we will get the operations of Gell-Mann matrices on qutrits as,

$$S^x|0\rangle = |1\rangle, \qquad S^x|1\rangle = |0\rangle + |2\rangle, \qquad S^x|2\rangle = |1\rangle, \tag{B.8}$$

$$S^y|0\rangle = i|1\rangle, \qquad S^y|1\rangle = -i|0\rangle + i|2\rangle, \qquad S^y|2\rangle = -i|1\rangle, \tag{B.9}$$

$$S^z|0\rangle = |0\rangle, \qquad S^z|1\rangle = 0|1\rangle, \qquad S^z|2\rangle = -|2\rangle. \tag{B.10}$$

Similar to the qubit case if we want to find a general pattern for Gell Mann matrices operating on i^{th} qutrit, It is difficult to find such a pattern for S^x and S^y but for S^z it can be generalized as,

$$S^z|a\rangle = (1-a)|a\rangle, \tag{B.11}$$

where a represents the qutrit in which the Gell-Mann matrices are operating. Now let us try to find an alternative for S^x and S^y.

Since the Hamiltonian matrix can be defined as,

$$\mathcal{H} = \langle\psi|\hat{H}|\psi\rangle, \tag{B.12}$$

where $|\psi\rangle$ is a wave function that corresponds to the Hilbert space of the system and $\hat{H}$ is the Hamiltonian operator. In our case, wave functions are a combination of the computational basis. Now let us do another Python exercise for computing the expectation values of spin-1 operators,

```
#Importing Libraries
import numpy as np
from scipy import linalg
import math
from IPython.display import display, Math
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore")

#Defining Spin 1-Operators I=S^0, X=S^x, Y=S^y, Z=S^z
I = np.eye(3)
X = np.array([[0, 1, 0], [1, 0, 1], [0, 1, 0]])
Y = np.array([[0, -1j, 0], [1j, 0, -1j], [0, 1j, 0]])
Z = np.array([[1, 0, 0], [0, 0, 0], [0, 0, -1]])

#Defining State
zero = [[1],[0],[0]]
one = [[0],[1],[0]]
two = [[0],[0],[1]]

#Defining a Function for Printing Matrix into Latex Format
def print_matrix(array):
    matrix = ''
    for row in array:
        try:
            for number in row:
                matrix += f'{number}&'
```

```
        except TypeError:
            matrix += f'{row}&'
        matrix = matrix[:-1] + r'\\'
    display(Math(r'\begin{bmatrix}'+matrix+r'\end{bmatrix}'))

display(Math(r"\langle0|S^x|0\rangle ="))
print_matrix(np.matmul(np.transpose(zero),np.matmul(X,zero)))
display(Math(r"\langle0|S^x|1\rangle ="))
print_matrix(np.matmul(np.transpose(zero),np.matmul(X,one)))
display(Math(r"\langle0|S^x|2\rangle ="))
print_matrix(np.matmul(np.transpose(zero),np.matmul(X,two)))
display(Math(r"\langle1|S^x|0\rangle ="))
print_matrix(np.matmul(np.transpose(one),np.matmul(X,zero)))
display(Math(r"\langle1|S^x|1\rangle ="))
print_matrix(np.matmul(np.transpose(one),np.matmul(X,one)))
display(Math(r"\langle1|S^x|2\rangle ="))
print_matrix(np.matmul(np.transpose(one),np.matmul(X,two)))
display(Math(r"\langle2|S^x|0\rangle ="))
print_matrix(np.matmul(np.transpose(two),np.matmul(X,zero)))
display(Math(r"\langle2|S^x|1\rangle ="))
print_matrix(np.matmul(np.transpose(two),np.matmul(X,one)))
display(Math(r"\langle2|S^x|2\rangle ="))
print_matrix(np.matmul(np.transpose(two),np.matmul(X,two)))

```

In this program we have computed the expectation for S^x only, one can compute the expectation for S^y in the same manner. Now let us try to analyze the results and define a method for computing Hamiltonian.

For S^x, the results are,

$$\langle 0|S^x|0\rangle = 0, \qquad \langle 1|S^x|0\rangle = 1, \qquad \langle 2|S^x|0\rangle = 0, \tag{B.13}$$

$$\langle 0|S^x|1\rangle = 1, \qquad \langle 1|S^x|1\rangle = 0, \qquad \langle 2|S^x|1\rangle = 1, \tag{B.14}$$

$$\langle 0|S^x|2\rangle = 0, \qquad \langle 1|S^x|2\rangle = 1, \qquad \langle 2|S^x|2\rangle = 0. \tag{B.15}$$

So this can be generalized as,

$$\langle a|S^x|b\rangle = \begin{cases} a-b & \text{if } (a-b)=1, \\ b-a & \text{if } (b-a)=1, \\ 0 & \text{if } |a-b|\neq 1. \end{cases} \tag{B.16}$$

where a and b are the computational basis for the qutrit system (i.e. $\{|0\rangle, |1\rangle, |2\rangle\}$). Now let us see the results for S^y,

$$\langle 0|S^y|0\rangle = 0, \qquad \langle 1|S^y|0\rangle = i, \qquad \langle 2|S^y|0\rangle = 0, \tag{B.17}$$

$$\langle 0|S^y|1\rangle = -i, \qquad \langle 1|S^y|1\rangle = 0, \qquad \langle 2|S^y|1\rangle = i, \tag{B.18}$$

$$\langle 0|S^y|2\rangle = 0, \qquad \langle 1|S^y|2\rangle = -i, \qquad \langle 2|S^y|2\rangle = 0. \tag{B.19}$$

So this can be generalised as,

$$\langle a|S^x|b\rangle = \begin{cases} (a-b)i & \text{if } (a-b)=1, \\ (a-b)i & \text{if } (b-a)=1, \\ 0 & \text{if } |a-b|\neq 1. \end{cases} \tag{B.20}$$

where a and b are the computational basis for the qutrit system (i.e. $\{|0\rangle, |1\rangle, |2\rangle\}$). Now we have found a general pattern for the operation of Gell-Mann matrices on qutrits, let us implement this in a Fortran program.

```
1 program spin1_trit_approach
2     implicit none
3     integer, parameter :: s = 4     !Total number of qutrits
4     integer, parameter :: num = 3**s        !Total number of states
5     integer :: i, j, k, l, p, flag
6     integer, allocatable :: decimal(:), decimal1(:), decimal2(:)
7     integer, allocatable :: digits1(:), digits2(:), digits3(:)
8     real*8, allocatable :: D(:,:), C(:,:), E(:,:), F(:,:), G(:,:), H
    (:,:), M(:,:)
9     character(len=:), allocatable :: state(:)
```

```
10      double precision :: W(num), WORK(3*num)
11      integer ::  LDA = num, LWORK = 3*num, INFO
12      character*1 :: JOBZ, UPLO
13      real*8 :: sum
14
15      ! Allocate arrays after determining the value of num
16      allocate(character(len=s) :: state(num))
17      allocate(integer :: decimal(num))
18      allocate(integer :: decimal1(num))
19      allocate(integer :: decimal2(num))
20      allocate(integer :: digits1(s))
21      allocate(integer :: digits2(s))
22      allocate(integer :: digits3(s))
23      allocate(real(8) :: D(num, num))
24      allocate(real(8) :: C(num, num))
25      allocate(real(8) :: E(num, num))
26      allocate(real(8) :: F(num, num))
27      allocate(real(8) :: G(num, num))
28      allocate(real(8) :: H(0:num-1, 0:num-1))
29      allocate(real(8) :: M(0:num-1, 0:num-1))
30
31      ! print *, "The states are:"
32      do i = 0, num -1
33          call integer_ternary(i, state(i+1), s)
34          write(1,*) state(i)
35          !print*, state(i)
36      end do
37
38      do i = 1, num
39          decimal(i) = ttod(state(i))
40          !print*, ' Decimal equivalent of ', state(i) ,' = ', decimal
    (i)
41      end do
42
43      C = 0.0d0
44          do i = 1, num
45              do j = 1, num
46                  call sd(state(i), digits1)
```

```
47                  call sd(state(j), digits2)
48                  !print*, digits1(:), digits2(:)
49                  do k = 1, s
50                      if (k < s) then
51                          flag = 0
52                          do l = k+2, s
53                              if (digits1(l) .ne. digits2(l)) then
54                                  flag = flag + 1
55                              else
56                                  continue
57                              end if
58                          end do
59                          if (flag == 0) then
60                              do p = 1, k-1
61                                  if (digits1(p) .ne. digits2(p)) then
62                                      flag = flag + 1
63                                  else
64                                      continue
65                                  end if
66                              end do
67                          else
68                              C(i,j) = C(i,j) + 0.0
69                          end if
70                          if (flag == 0) then
71                              if((digits1(k)- digits2(k)==1) .and. &
72                              (digits1(k+1)- digits2(k+1)==1)) &
73                              then C(i,j) = C(i,j) + 1.0
74                              else if ((digits1(k)- digits2(k)==1)
75                              .and. (digits2(k+1)- digits1(k+1)==1)) &
76                              then C(i,j) = C(i,j) + 1.0
77                              else if ((digits2(k)- digits1(k)==1) &
78                              .and. (digits1(k+1)- digits2(k+1)==1)) &
79                              then C(i,j) = C(i,j) + 1.0
80                              else if ((digits2(k)- digits1(k)==1) &
81                              .and. (digits2(k+1)- digits1(k+1)==1)) &
82                              then C(i,j) = C(i,j) + 1.0
83                              else
84                                  C(i,j) = C(i,j) + 0.0
```

```
85                              end if
86                          else
87                              C(i,j) = C(i,j) + 0.0
88                          end if
89                      else if (k == s) then
90                          flag = 0
91                          do l = 2, k - 1
92                              if (digits1(l) .ne. digits2(l)) then
93                                  flag = flag + 1
94                              else
95                                  continue
96                              end if
97                          end do
98                          if (flag == 0) then
99                              if((digits1(k)- digits2(k)==1) .and. &
100                             (digits1(1)- digits2(1)==1)) &
101                             then C(i,j) = C(i,j) + 1.0
102                             else if ((digits1(k)- digits2(k)==1)  &
103                             .and. (digits2(1)- digits1(1)==1)) &
104                             then C(i,j) = C(i,j) + 1.0
105                             else if ((digits2(k)- digits1(k)==1) &
106                             .and. (digits1(1)- digits2(1)==1)) &
107                             then C(i,j) = C(i,j) + 1.0
108                             else if ((digits2(k)- digits1(k)==1) &
109                             .and. (digits2(1)- digits1(1)==1)) &
110                             then C(i,j) = C(i,j) + 1.0
111                             else
112                                 C(i,j) = C(i,j) + 0.0
113                             end if
114                         else
115                             C(i,j) = C(i,j) + 0.0
116                         end if
117                     end if
118                 end do
119             end do
120         end do
121     ! write(3,*) "C :"
122     do k = 1, num
```

```
123         do l = 1, num
124             if( C(k,l) .le. 0.00000000001) then
125                 C(k,l) = 0.0
126             else
127                 continue
128             end if
129             ! write(3,*) k, l, C(k,l)
130         end do
131     end do
132 
133         D = 0.0d0  ! Initialize the matrix
134         do i = 1, num
135             do j = 1, num
136                 call sd(state(i), digits1)
137                 call sd(state(j), digits2)
138                 do k = 1, s
139                     flag = 0
140                     ! Check for differences from k+2 to s
141                     if (k < s) then
142                         do l = k + 2, s
143                             if (digits1(l) .ne. digits2(l)) then
144                                 flag = flag + 1
145                             end if
146                         end do
147                         if (flag == 0) then
148                             do p = 1, k - 1
149                                 if (digits1(p) .ne. digits2(p)) then
150                                     flag = flag + 1
151                                 end if
152                             end do
153                         end if
154                         !print*, i, j, k, flag
155                         if (flag == 0) then
156                             if ((digits1(k) - digits2(k)==1) &
157                             .and. (digits1(k+1)-digits2(k+1)==1)) &
158                             then D(i, j) = D(i, j) - 1.0
159                             else if ((digits1(k) - digits2(k)==1) &
160                             .and. (digits2(k+1)-digits1(k+1)==1)) &
```

```
                                then D(i, j) = D(i, j) + 1.0
                                else if ((digits2(k) - digits1(k)==1) &
                                .and. (digits1(k+1)-digits2(k+1)==1)) &
                                then D(i, j) = D(i, j) + 1.0
                                else if ((digits2(k) - digits1(k)==1) &
                                .and. (digits2(k+1)-digits1(k+1)==1)) &
                                then D(i, j) = D(i, j) - 1.0
                                end if
                            end if
                        else if (k == s) then
                            flag = 0
                            ! Check for differences from 2 to k-1
                            do l = 2, k - 1
                                if (digits1(l) .ne. digits2(l)) then
                                    flag = flag + 1
                                end if
                            end do
                            if (flag == 0) then
                                if ((digits1(k)-digits2(k)==1) &
                                .and. (digits1(1)-digits2(1)==1)) &
                                then D(i, j) = D(i, j) - 1.0
                                else if ((digits1(k)-digits2(k)==1) &
                                .and. (digits2(1)-digits1(1)==1)) &
                                then D(i, j) = D(i, j) + 1.0
                                else if ((digits2(k)-digits1(k)==1) &
                                .and. (digits1(1)-digits2(1)==1)) &
                                then D(i, j) = D(i, j) + 1.0
                                else if ((digits2(k)-digits1(k)==1) &
                                .and. (digits2(1)-digits1(1)==1)) &
                                then D(i, j) = D(i, j) - 1.0
                                end if
                            end if
                        end if
                    end do
                end do
            end do
        ! write(3,*) "D :"
        do k = 1, num
```

```
199         do l = 1, num
200             if( D(k,l) .ge. 0 .and. D(k,l) .le. 0.00000000001) then
201                 D(k,l) = 0.0
202             else
203                 continue
204             end if
205             ! write(3,*) k, l, D(k,l)
206         end do
207     end do
208 
209     E = 0.0
210     do i = 1, num
211         do j = 1, num
212             if (i /= j) then
213                 E(i, j) = 0.0
214             else
215                 E(i, j) = 0.0
216                 call sd(state(i), digits3)
217                 do l = 1, s - 1
218                     E(i, j) = E(i, j) + (1 - digits3(l)) * &
219                               (1 - digits3(l + 1))
220                 end do
221                 E(i, j) = E(i, j) + (1 - digits3(1)) * &
222                               (1 - digits3(s))
223             end if
224         end do
225     end do
226     ! write(3,*) "E :"
227     ! do k = 1, num
228     !     do l = 1, num
229     !         write(3,*) k, l, E(k, l)
230     !     end do
231     ! end do
232 
233     F = 0.0d0
234     G = 0.0d0
235     H = 0.0d0
236     F = C + D + E
```

```
237     G = matmul(F,F)
238     do i = 1, num
239         do j = 1, num
240             !if (i .le. j) then
241                 H(i-1,j-1) = F(i,j) + (1/3)* G(i,j)
242                 write(3,*) i, j, H(i-1,j-1)
243             !end if
244         end do
245     end do
246     M = H
247 
248     ! Define parameters for DSYEV
249     JOBZ = 'V'  ! Compute eigenvalues and eigenvectors
250     UPLO = 'U'  ! Upper triangular part of A is stored
251     ! Call DSYEV to compute eigenvalues and eigenvectors
252     call DSYEV(JOBZ, UPLO, num, H, LDA, W, WORK, LWORK, INFO)
253 
254     ! Check for successful execution
255     sum = 0.0d0
256     if (INFO == 0) then
257         write(3,*) 'Eigenvalues are:'
258         do l = 1, num
259             write(3,*) W(l)
260             ! sum = sum + W(l)
261         end do
262     else
263         print*, 'Error in DSYEV, info =', INFO
264     end if
265     !print*, sum
266         !write(2,*) 'Eigen Vectors are :'
267     do k = 1, num
268         !write(2,*) 'Eigen Vector ', k, ':'
269         do l = 1, num
270             write(7,*) k, l, H(l-1,k-1)
271         end do
272     end do
273 
274     deallocate(state)
```

```
275     deallocate(decimal)
276     deallocate(decimal1)
277     deallocate(decimal2)
278     deallocate(digits1)
279     deallocate(digits2)
280     deallocate(digits3)
281     deallocate(D)
282     deallocate(C)
283     deallocate(E)
284     deallocate(F)
285     deallocate(G)
286     deallocate(H)
287     deallocate(M)
288 contains
289 
290 function ttod(ternaryString) result(decimalValue)
291     implicit none
292     character(len=*), intent(in) :: ternaryString
293     integer :: decimalValue
294     integer :: i, length
295 
296     decimalValue = 0
297     length = len_trim(ternaryString)  ! Get the length of the
    ternary string
298 
299     ! Convert ternary string to decimal
300     do i = 1, length
301         select case (ternaryString(i:i))
302             case ('0')
303                 ! Do nothing for '0'
304             case ('1')
305                 decimalValue = decimalValue + 1*3**(length - i)
306             case ('2')
307                 decimalValue = decimalValue + 2*3**(length - i)
308             case default
309                 print *, "Invalid character in ternary string."
310                 decimalValue = -1  ! Indicate an error with -1
311                 return
```

```
312         end select
313     end do
314 end function ttod
315 
316 subroutine integer_ternary(num, ternary_string, n)
317     implicit none
318     integer, intent(in) :: num, n
319     character(len=n), intent(out) :: ternary_string
320     integer :: i, temp_num
321 
322     temp_num = num
323     ternary_string = repeat('0', n)
324 
325     ! Convert the integer to ternary
326     do i = n, 1, -1
327         ternary_string(i:i) = achar(mod(temp_num, 3)+ichar('0'))
328         temp_num = temp_num / 3
329     end do
330 end subroutine integer_ternary
331 
332 subroutine sd(ternaryString, digits)
333     character(len=*), intent(in) :: ternaryString
334     integer, allocatable, intent(out) :: digits(:)
335     integer :: i, len
336 
337     len = len_trim(ternaryString)
338     allocate(digits(len))          ! Allocate array to hold digits
339 
340     ! Convert each character to an integer
341     do i = 1, len
342         select case (ternaryString(i:i))
343             case ('0')
344                 digits(i) = 0
345             case ('1')
346                 digits(i) = 1
347             case ('2')
348                 digits(i) = 2
349             case default
```

```
                print *, "Invalid character in ternary string."
                digits = 0  ! Set to zero if invalid character
                return
            end select
        end do
end subroutine sd
end program

```

In this program, we have defined s as a parameter that gives us the freedom to change the number of qutrits and allows us to compute the Hamiltonian for a desired system. So by using this approach, we have reduced the complexity of defining several matrices required in the case of the tensor product approach, also we have reduced the computation time tangibly.

Appendix C

General Partial Trace for n-parties and d-dimensional Qudits

C.1 Partial Trace

The most important application of the density matrix is as a descriptive tool for the subsystems of the composite system, known as the reduced density matrix [23].

Suppose we have a physical system AB, whose state is given by ρ_{AB}, and if we want to study subsystem A, then we can get the reduced density matrix of A, using an operation known as partial trace taken over the subsystem B as,

$$\rho_A \equiv \mathrm{tr}_B(\rho_{AB}), \tag{C.1}$$

where tr_B is the partial trace operator. The partial trace is defined by,

$$\mathrm{tr}_B(|a_1\rangle\langle a_2| \otimes |b_1\rangle\langle b_2|) \equiv |a_1\rangle\langle a_2|\mathrm{tr}(|b_1\rangle\langle b_2|), \tag{C.2}$$

where $|a_1\rangle$ and $|a_2\rangle$ are any two vectors in the state space of A and $|b_1\rangle$ and $|b_2\rangle$ are any two vectors in the state space of B.

Now, one may think, why is the partial trace used to describe part of a larger system? This is because the partial trace is a unique operation that gives the correct

description of observable quantities for subsystems of a composite system. This can be generalised to multi-qudit systems as well. If we have an N-qudit state $\rho_{123\cdots N}$, if we are interested in the state of the even-qudit subsystem, then we can write:

$$\rho_{246\cdots} \equiv \mathrm{Tr}_{135\cdots}\left(\rho_{123\cdots N}\right). \tag{C.3}$$

This implies that the partial trace can be performed over any arbitrary subset of qudits. The partial trace operation serves as the quantum analogue of obtaining a marginal distribution from a multivariate probability distribution. We have made a Fortran program to compute the partial trace for n-parties and a d-dimensional system. The motivation behind this is that no subroutine in Fortran performs d-dimensional, n-parties partial trace; some subroutines perform partial trace for qubits [35], but when we are dealing with higher dimensions, there is no partial trace subroutine as such.

The importance of this subroutine is, it is the most general case of partial trace; it gives us a hand to trace out any number of parties, for any dimensional systems. Let us look into the details of this subroutine for real density matrices,

---

subroutine ptrace(rho, d, s, s1, site, rdm)

In/Out	Argument	Description
in	`rho`	rho is `real*8` array of dimension `(0:d**s-1, 0:d**s-1)` representing the input density matrix.
in	`d`	d (integer) is the dimension of each qudit.
in	`s`	s (integer) is the total number of qudits in the system.
in	`s1`	s1 (integer) is the number of qudits not to be traced out.

In/Out	Argument	Description
in	`site`	site is `real*8` array of dimension `(0:s-1)` containing site indices of qudits to keep.
out	`rdm`	rdm is `real*8` array of dimension `(0:d**s1-1, 0:d**s1-1)` containing reduced density matrix after partial trace.

Table C.1: Arguments for the subroutine computing the reduced density matrix (RDM).

Implementation

```
subroutine ptrace(rho, d, s, s1, site, rdm)
    implicit none
    integer, intent(in) :: d, s, s1
    integer, intent(in) :: site(0:s1-1)
    real*8, intent(in) :: rho(0:d**s-1, 0:d**s-1)
    real*8, intent(out) :: rdm(d**s1, d**s1)
    integer, allocatable :: idits(:), jdits(:), ridits(:)
    integer, allocatable :: rjdits(:), to(:)
    integer :: i, j, k, p, flag, ri, rj
    real*8 :: a
    allocate(idits(0:s-1), jdits(0:s-1))
    allocate(ridits(0:s1-1), rjdits(0:s1-1))
    allocate(to(0:s-s1-1))
    ! Compute the indices to trace out
    call TOI(s, site, s1, to
    rdm = 0.0d0
    do i = 0, d**s - 1
        do j = 0, d**s - 1
            if (rho(i,j) .ne. 0.0d0) then
                a = rho(i,j)
                call DECTOD(i, d, s, idits)
                call DECTOD(j, d, s, jdits)
                flag = 0
                do k = 0, s - s1 - 1
                    if (idits(to(k)) .ne. jdits(to(k))) then
                        flag = 1
                        exit
```

```
                    end if
                end do
                if (flag .eq. 0) then
                    do k = 0, s1 - 1
                        p = site(k)
                        ridits(k) = idits(p)
                        rjdits(k) = jdits(p)
                    end do
                    call DTODEC(d, s1, ridits, ri)
                    call DTODEC(d, s1, rjdits, rj)
                    rdm(ri+1, rj+1) = rdm(ri+1, rj+1) + a
                end if
            end if
        end do
    end do

    deallocate(idits, jdits, ridits, rjdits, to)
end subroutine ptrace

```

A few subroutines that are used in this are,

Subroutine	Function
`TOI`	This subroutine computes the indices to be traced out and returns them into the array `to`.
`DECTOD`	This subroutine computes the qudit equivalent of any decimal number and returns it into the array `dits`.
`DTODEC`	This subroutine computes the decimal equivalent of any qudit and returns it into the integer `dec`.

Implementation of these subroutines is,

```
subroutine TOI(s, site, s1, to)
    implicit none
    integer, intent(in) :: s, s1
    integer, intent(in) :: site(0:s1-1)
    integer, intent(out) :: to(0:s-s1-1)
```

```
    integer :: i, j, k, found
    k = 0
    do i = 0, s-1
        found = 0
        do j = 0, s1 - 1
            if (i == site(j)) then
                found = 1
                exit
            end if
        end do
        if (found == 0) then
            to(k) = i
            k = k + 1
        end if
    end do
end subroutine TOI

subroutine DECTOD(dec, d, s, dits)
    implicit none
    integer, intent(in) :: dec, d, s
    integer, allocatable, intent(out) :: dits(:)
    integer :: i, temp_dec
    allocate(dits(0:s-1))
    temp_dec = dec
    do i = s-1, 0, -1
        dits(i) = mod(temp_dec, d)
        temp_dec = temp_dec / d
    end do
end subroutine DECTOD

subroutine DTODEC(d, s, dits, dec)
    implicit none
    integer, intent(in) :: d, s
    integer, dimension(0:s-1), intent(in) :: dits
    integer, intent(out) :: dec
    integer :: i
    dec = 0
    do i = 0, s - 1
```

```
44          dec = dec * d + dits(i)
45      end do
46  end subroutine DTODEC
```

In the same way, one can get the subroutine for complex density matrices just by changing the data type of the arrays involved; the logic will be the same.

Bibliography

[1] M. Plischke and B. Bergersen, *Equilibrium Statistical Physics*, World Scientific, Singapore (2006).

[2] S. G. Brush, History of the Lenz-Ising model, *Reviews of Modern Physics*, **39**, 4, 883–893 (1967).

[3] P. Pfeuty, The one-dimensional Ising model with a transverse field, *Annals of Physics*, **57**, 1, 79–90 (1970).

[4] N. Zettili, *Quantum Mechanics: Concepts and Applications*, John Wiley & Sons, Chichester (2009).

[5] I. Affleck, T. Kennedy, E. H. Lieb, and H. Tasaki, Rigorous results on valence-bond ground states in antiferromagnets, *Phys. Rev. Lett.*, **59**, 7, 799–802 (1987).

[6] W. Heisenberg, Zur Theorie des Ferromagnetismus, *Zeitschrift für Physik*, **49**, 619–636 (1928).

[7] L. Hulthén, Über das Austauschproblem eines Kristalles, *Arkiv för Matematik, Astronomi och Fysik*, **26A**, 11, 1–106 (1938).

[8] E. Annala, T. Gorda, A. Kurkela, and A. Vuorinen, Evidence for quark-matter cores in massive neutron stars, *Nature Physics*, **16**, 907–910 (2020).

[9] L. McLerran, R. Pisarski, and C. Ratti, Quarkyonic matter and applications in neutron stars (2024).

[10] AAS Nova Editors, Could spinning soup start starquakes and explain a magnetar mystery?, *AAS Nova* (2023).

[11] C. K. Majumdar and D. Ghosh, On Next-Nearest-Neighbor Interaction in Linear Chain, *Journal of Mathematical Physics*, **10**, 1388–1398 (1969).

[12] R. Shankar, Chapter 14 – Spin, in *Principles of Quantum Mechanics*, 2nd ed., Springer (1994).

[13] D. J. Griffiths, *Introduction to Quantum Mechanics*, 2nd ed., Prentice Hall (2004).

[14] E. Anderson et al., *LAPACK Users' Guide*, 3rd ed., SIAM, Philadelphia, PA (1999).

[15] S. Pissanetzky, *Sparse Matrix Technology*, Academic Press (1984).

[16] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed., SIAM, Philadelphia (2003).

[17] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, 3rd ed., MIT Press (2014).

[18] J. Reinders and M. Sweeney, *High Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming with MPI, OpenMP, and CUDA*, Elsevier (2018).

[19] M. Cao and L. Zhang, Efficient Parallelization of Quantum Monte Carlo Simulations on GPUs Using OpenACC, *Journal of Computational Physics*, **318**, 271–281 (2016).

[20] A. Einstein, B. Podolsky, and N. Rosen, Can quantum-mechanical description of physical reality be considered complete?, *Physical Review*, **47**, 10, 777–780 (1935).

[21] E. Schrödinger, Discussion of Probability Relations between Separated Systems,

Math. Proc. Cambridge Philos. Soc., **31**, 555–563 (1935).

[22] C. H. Bennett et al., Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels, *Physical Review Letters*, **70**, 13, 1895–1899 (1993).

[23] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information: 10th Anniversary Edition*, Cambridge University Press (2010).

[24] R. Horodecki et al., Quantum entanglement, *Rev. Mod. Phys.*, **81**, 865–942 (2009).

[25] R. F. Werner, Quantum states with Einstein-Podolsky-Rosen correlations admitting a hidden-variable model, *Phys. Rev. A*, **40**, 4277–4281 (1989).

[26] W. K. Wootters, Entanglement of formation and concurrence, *Quantum Inf. Comput.*, **1**, 27–44 (2001).

[27] S. A. Hill and W. K. Wootters, Entanglement of a Pair of Quantum Bits, *Physical Review Letters*, **78**, 26, 5022–5025 (1997).

[28] X.-L. Zong et al., Monogamy of Quantum Entanglement, *Frontiers in Physics*, **10**, 880560 (2022).

[29] C. H. Bennett et al., Concentrating partial entanglement by local operations, *Phys. Rev. A*, **53**, 2046–2052 (1996).

[30] J. von Neumann, *Mathematical Foundations of Quantum Mechanics*, Princeton University Press (1955).

[31] A. Wehrl, General properties of entropy, *Reviews of Modern Physics*, **50**, 2, 221–260 (1978).

[32] G. Vidal et al., Entanglement in quantum critical phenomena, *Physical Review Letters*, **90**, 22, 227902 (2003).

[33] M. Gell-Mann, Symmetries of Baryons and Mesons, *Physical Review*, **125**, 3,

1067–1084 (1962).

[34] M. S. Ramkarthik and E. L. Pereira, Qudits and Generalized Bell States for Pedestrians: A Dive Into the World of Qudits and Quantum Teleportation, *Resonance: Journal of Science Education*, **28**, 12, 1845–1863 (2023).

[35] M. S. Ramkarthik and P. D. Solanki, *Numerical Recipes in Quantum Information Theory and Quantum Computing: An Adventure in FORTRAN 90*, CRC Press (2021).