TRABAJO DE INVESTIGACIÓN

TRANSICIONES DE FASE TOPOLÓGICAS EN MATERIA CONDENSADA

DEL MODELO DE ISING A LA CADENA DE KITAEV

SISTEMAS COMPLEJOS

Daniel Michel Pino González

Máster en Física Teórica

Curso académico 2021-22

ÍNDICE GENERAL

DEL MODELO DE ISING CLÁSICO

AL MODELO DE ISING TRANSVERSO

Transverse field Ising model (TFIM)

Considérese el hamiltoniano cuántico de la cadena unidimensional de Ising transverso,

$$\hat{H}_{\mathrm{TFIM}} = -Jg\sum_i \hat{\sigma}_i^x - J\sum_{\langle ij\rangle}\hat{\sigma}_i^z\hat{\sigma}_j^z$$

donde g es un parámetro de acoplo con un campo transverso a la cadena y J es un prefactor de escala con dimensiones de energía.

Los operadores son matrices de Pauli que cumplen el álgebra *su*(2),

$$[\hat{\sigma}_i^\alpha, \hat{\sigma}_j^\beta] = 2i\delta_{ijk}\epsilon_{\alpha\beta\gamma}\hat{\sigma}_k^\gamma \quad , \quad \alpha = x, y, z$$

Correspondencia con el modelo de Ising clásico

Suele existir un mapa uno-a-uno entre las **transiciones cuánticas a temperatura cero** en D dimensiones y las **transiciones térmicas** que presenta su análogo clásico (D + z)-dimensional, por lo que ambos sistemas pertenecen a la misma **clase de universalidad**.

En modelos con transiciones de fase de orden-desorden, debido al **ajuste lineal** de la energía con la longitud de correlación, el exponente dinámico **es siempre la unidad** en sistemas puros.

El TFIM 1D emerge como el **límite de la matriz de transferencia** efectiva de del modelo de Ising clásico e isotrópico sobre una malla cuadrada.

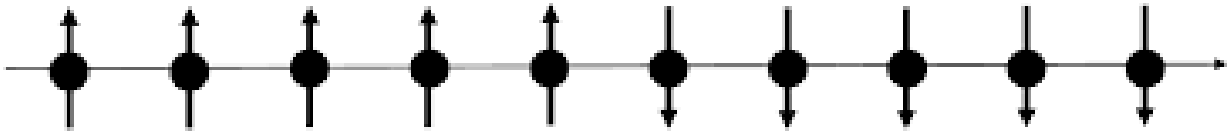

Figura 7: Representación de una cadena de espines.
[5] H. Moriya, IOP Pub. 2019 6 (2019).

ALGORITMO DE SIMULACIÓN

DEL MODELO DE ISING TRANSVERSO

ALGORITMO DE SIMULACIÓN

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <math.h>
#include <lapacke.h>
#include "include/def.h"
#include <time.h>

int main(int argc, char **argv) {

    time_t t_0, t_1;
    t_0 = time(NULL);

    int length = 10;
    float g = 2;

    tsbasis * sbasis = init_sbasis(length);
    tvbasis * vbasis = init_vbasis(length);
    basis_states(sbasis, vbasis);

    tsmatrix * Hmatrix = H(sbasis, g);
    tspectrum * spectrum = eigenvalues(Hmatrix);
    float M = magnetization_expectation_value(vbasis,
                      spectrum->eigvectors->data[0],
                      length);

    t_1 = time(NULL);
    printf("Timelapse: %ld seconds\n", (t_1 - t_0));

    return 0;
}
```

i. **Base vectorial de estados**
ii. **Hamiltoniano cuántico**
iii. **Espectro de energía**
iv. **Valor esperado de la magnetización**

ALGORITMO DE SIMULACIÓN

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <math.h>
#include <lapacke.h>
#include "include/def.h"
#include <time.h>

int main(int argc, char **argv) {

    time_t t_0, t_1;
    t_0 = time(NULL);

    int length = 10;
    float g = 2;

    tsbasis * sbasis = init_sbasis(length);
    tvbasis * vbasis = init_vbasis(length);
    basis_states(sbasis, vbasis);

    tsmatrix * Hmatrix = H(sbasis, g);
    tspectrum * spectrum = eigenvalues(Hmatrix);
    float M = magnetization_expectation_value(vbasis,
                      spectrum->eigvectors->data[0],
                      length);

    t_1 = time(NULL);
    printf("Timelapse: %ld seconds\n", (t_1 - t_0));

    return 0;
}
```

i. **Base vectorial de estados**

ii. Hamiltoniano cuántico

iii. Espectro de energía

iv. Valor esperado de la magnetización

BASE VECTORIAL DE ESTADOS

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <math.h>
#include <lapacke.h>
#include "include/def.h"
#include <time.h>

int main(int argc, char **argv) {

    time_t t_0, t_1;
    t_0 = time(NULL);

    int length = 10;
    float g = 2;

    tsbasis * sbasis = init_sbasis(length);
    tvbasis * vbasis = init_vbasis(length);
    basis_states(sbasis, vbasis);

    tsmatrix * Hmatrix = H(sbasis, g);
    tspectrum * spectrum = eigenvalues(Hmatrix);
    float M = magnetization_expectation_value(vbasis,
                        spectrum->eigvectors->data[0],
                        length);

    t_1 = time(NULL);
    printf("Timelapse: %ld seconds\n", (t_1 - t_0));

    return 0;
}
```

```c
// SBASIS
typedef struct {
    tsmatrix * state_matrix;        // (2D-array) basis states
    int nstates;                    // number of states
    int length;                     // chain length
} tsbasis;


// VBASIS
typedef struct {
    tsmatrix * vector_matrix;       // (2D-array) basis vectors
    int nvectors;                   // number of vectors
    int * nup;                      // number of spin-up
} tvbasis;
```

STATE BASIS	
length	2
nstates	$2^L = 4$
state matrix	
0	0
0	1
1	0
1	1

VECTOR BASIS				
nvectors				4
vector matrix				**nup**
0	0	0	1	0
0	0	1	0	1
0	1	0	0	1
1	0	0	0	2

BASE VECTORIAL DE ESTADOS

```c
void basis_states(tsbasis *sbasis, tvbasis *vbasis) {
    int n;

    for (int i = 0; i < sbasis->nstates; i++) {
        n = i;
        for (int j = 0; j < sbasis->length; j++) {
            sbasis->data[i]->data[j] = n%2;
            n /= 2;
        }
    }

    for (int i = 0; i < sbasis->nstates; i++) {
        tvector * aux_vector = init_vector(sbasis->nstates);

        int nup = 0;

        if (sbasis->data[i]->data[0] == 1) {
            aux_vector->data[0] = 1;
            nup = 1;
        } else {
            aux_vector->data[1] = 1;
        }
        aux_vector->length = 2;

        for (int j = 1; j < sbasis->length; j++) {
            tensor_vector_product(aux_vector, sbasis->data[i]->data[j]);
            nup += sbasis->data[i]->data[j];
        }

        int pos1;
        for (int j = 0; j < vbasis->nvectors; j++) {
            if (aux_vector->data[j] == 1) {
                pos1 = j;
                vbasis->nup[j] = nup;
            }
        }

        for (int j = 0; j < vbasis->nvectors; j++) {
            vbasis->vector_matrix->data[pos1][j] = aux_vector->data[j];
        }
        free(aux_vector);
    }
}
```

STATE BASIS	
length	2
nstates	$2^L = 4$
state matrix	
0 0	0 mod(2)
0 1	1 mod(2)
1 0	2 mod(2)
1 1	3 mod(2)

VECTOR BASIS				
nvectors				4
vector matrix				**nup**
0	0	0	1	0
0	0	1	0	1
0	1	0	0	1
1	0	0	0	2

[0] = (0, 1)
[1] = (1, 0)

[0, 0] = (0, 1) ⊗ (0, 1) = (0, 0, 0, 1)

ALGORITMO DE SIMULACIÓN

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <math.h>
#include <lapacke.h>
#include "include/def.h"
#include <time.h>

int main(int argc, char **argv) {

    time_t t_0, t_1;
    t_0 = time(NULL);

    int length = 10;
    float g = 2;

    tsbasis * sbasis = init_sbasis(length);
    tvbasis * vbasis = init_vbasis(length);
    basis_states(sbasis, vbasis);

    tsmatrix * Hmatrix = H(sbasis, g);
    tspectrum * spectrum = eigenvalues(Hmatrix);
    float M = magnetization_expectation_value(vbasis,
                        spectrum->eigvectors->data[0],
                        length);

    t_1 = time(NULL);
    printf("Timelapse: %ld seconds\n", (t_1 - t_0));

    return 0;
}
```

i. Base vectorial de estados

ii. Hamiltoniano cuántico

iii. Espectro de energía

iv. Valor esperado de la magnetización

HAMILTONIANO CUÁNTICO

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <math.h>
#include <lapacke.h>
#include "include/def.h"
#include <time.h>

int main(int argc, char **argv) {

    time_t t_0, t_1;
    t_0 = time(NULL);

    int length = 10;
    float g = 2;

    tsbasis * sbasis = init_sbasis(length);
    tvbasis * vbasis = init_vbasis(length);
    basis_states(sbasis, vbasis);

    tsmatrix * Hmatrix = H(sbasis, g);
    tspectrum * spectrum = eigenvalues(Hmatrix);
    float M = magnetization_expectation_value(vbasis,
                      spectrum->eigvectors->data[0],
                      length);

    t_1 = time(NULL);
    printf("Timelapse: %ld seconds\n", (t_1 - t_0));

    return 0;
}
```

```c
tsmatrix * hamiltonian(int length, float g) {
    int a;
    int dim = (int)(pow(2, length)+1e-9);
    tsmatrix * Hmatrix = init_square_matrix(dim, dim);

    tsmatrix * sigma_zz = init_square_matrix(4,4);
    sigma_zz->data[0][0] = -1;
    sigma_zz->data[1][1] = 1;
    sigma_zz->data[2][2] = 1;
    sigma_zz->data[3][3] = -1;

    sum_matrix(Hmatrix, tensor_matrix_product(sigma_zz, identity_matrix((int)(pow(2, length-2)+1e-9))), 1);
    for (int i = 1; i < (length-2); i++) {
        sum_matrix(Hmatrix, tensor_matrix_product(
                            tensor_matrix_product(identity_matrix((int)(pow(2, i)+1e-9)), sigma_zz),
                            identity_matrix((int)(pow(2, length-i-2)+1e-9))),
                            1);
    }
    sum_matrix(Hmatrix, tensor_matrix_product(identity_matrix((int)(pow(2, length-2)+1e-9)), sigma_zz), 1);

    tsmatrix * sigma_x = init_square_matrix(2,2);
    sigma_x->data[1][0] = -1;
    sigma_x->data[0][1] = -1;

    sum_matrix(Hmatrix, tensor_matrix_product(sigma_x, identity_matrix((int)(pow(2, length-1)+1e-9))), g);
    for (int i = 1; i < (length-1); i++) {
        sum_matrix(Hmatrix, tensor_matrix_product(
                            tensor_matrix_product(identity_matrix((int)(pow(2, i)+1e-9)), sigma_x),
                            identity_matrix((int)(pow(2, length-i-1)+1e-9))),
                            g);
    }
    sum_matrix(Hmatrix, tensor_matrix_product(identity_matrix((int)(pow(2, length-1)+1e-9)), sigma_x), g);

    return Hmatrix;
}
```

$$\hat{H}_{\mathrm{TFIM}} = -Jg\sum_{i}\hat{\sigma}_i^x - J\sum_{\langle ij\rangle}\hat{\sigma}_i^z\hat{\sigma}_j^z$$

HAMILTONIANO CUÁNTICO

$$\hat{H}_{\mathrm{TFIM}} = -Jg\sum_i \hat{\sigma}_i^x - J\sum_{\langle ij\rangle}\hat{\sigma}_i^z\hat{\sigma}_j^z$$

$$\hat{\sigma}_i^x = \mathbb{I}_{i-1}\otimes\hat{\sigma}^x\otimes\mathbb{I}_{N-i}$$

$$\hat{\sigma}_i^z\hat{\sigma}_{i-1}^z = \mathbb{I}_{i-1}\otimes(\hat{\sigma}^z\otimes\hat{\sigma}^z)\otimes\mathbb{I}_{N-i-1}$$

```
tsmatrix * hamiltonian(int length, float g) {
    int a;
    int dim = (int)(pow(2, length)+1e-9);
    tsmatrix * Hmatrix = init_square_matrix(dim, dim);

    tsmatrix * sigma_zz = init_square_matrix(4,4);
    sigma_zz->data[0][0] = -1;
    sigma_zz->data[1][1] = 1;
    sigma_zz->data[2][2] = 1;
    sigma_zz->data[3][3] = -1;

    sum_matrix(Hmatrix, tensor_matrix_product(sigma_zz, identity_matrix((int)(pow(2, length-2)+1e-9))), 1);
    for (int i = 1; i < (length-2); i++) {
        sum_matrix(Hmatrix, tensor_matrix_product(
                            tensor_matrix_product(identity_matrix((int)(pow(2, i)+1e-9)), sigma_zz),
                            identity_matrix((int)(pow(2, length-i-2)+1e-9))),
                            1);
    }
    sum_matrix(Hmatrix, tensor_matrix_product(identity_matrix((int)(pow(2, length-2)+1e-9)), sigma_zz), 1);

    tsmatrix * sigma_x = init_square_matrix(2,2);
    sigma_x->data[1][0] = -1;
    sigma_x->data[0][1] = -1;

    sum_matrix(Hmatrix, tensor_matrix_product(sigma_x, identity_matrix((int)(pow(2, length-1)+1e-9))), g);
    for (int i = 1; i < (length-1); i++) {
        sum_matrix(Hmatrix, tensor_matrix_product(
                            tensor_matrix_product(identity_matrix((int)(pow(2, i)+1e-9)), sigma_x),
                            identity_matrix((int)(pow(2, length-i-1)+1e-9))),
                            g);
    }
    sum_matrix(Hmatrix, tensor_matrix_product(identity_matrix((int)(pow(2, length-1)+1e-9)), sigma_x), g);

    return Hmatrix;
}
```

ALGORITMO DE SIMULACIÓN

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <math.h>
#include <lapacke.h>
#include "include/def.h"
#include <time.h>

int main(int argc, char **argv) {

    time_t t_0, t_1;
    t_0 = time(NULL);

    int length = 10;
    float g = 2;

    tsbasis * sbasis = init_sbasis(length);
    tvbasis * vbasis = init_vbasis(length);
    basis_states(sbasis, vbasis);

    tsmatrix * Hmatrix = H(sbasis, g);
    tspectrum * spectrum = eigenvalues(Hmatrix);
    float M = magnetization_expectation_value(vbasis,
                      spectrum->eigvectors->data[0],
                      length);

    t_1 = time(NULL);
    printf("Timelapse: %ld seconds\n", (t_1 - t_0));

    return 0;
}
```

i. Base vectorial de estados

ii. Hamiltoniano cuántico

iii. Espectro de energía

iv. Valor esperado de la magnetización

ESPECTRO DE ENERGÍA

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <math.h>
#include <lapacke.h>
#include "include/def.h"
#include <time.h>

int main(int argc, char **argv) {

    time_t t_0, t_1;
    t_0 = time(NULL);

    int length = 10;
    float g = 2;

    tsbasis * sbasis = init_sbasis(length);
    tvbasis * vbasis = init_vbasis(length);
    basis_states(sbasis, vbasis);

    tsmatrix * Hmatrix = H(sbasis, g);
    tspectrum * spectrum = eigenvalues(Hmatrix);
    float M = magnetization_expectation_value(vbasis,
                        spectrum->eigvectors->data[0],
                        length);

    t_1 = time(NULL);
    printf("Timelapse: %ld seconds\n", (t_1 - t_0));

    return 0;
}
```

```c
tspectrum * eigenvalues(tsmatrix *Hmatrix) {
    int dim = Hmatrix->nrow;
    tspectrum * spectrum = init_spectrum(dim, dim);     // (tspectrum) initialize spectrum
    int INFO = 1;       // (int) out-parameter of LAPACKE_ssyev function
                            // = 0: successful exit
                            // < 0: if INFO = -i, the i-th argument had an illegal value
                            // > 0: if INFO = i, the algorithm failed to converge; i off-diagonal elements of an
                              // intermediate tridiagonal form did not converge to zero
    tvector * matrix_array = init_vector(Hmatrix->ncol*Hmatrix->nrow);// (ncol*nrow_Hmatrix vector) conversion
                                                                      // from tsmatrix to float-array

    for (int i = 0; i < Hmatrix->nrow; i++) {                         // for each row
        for (int j = 0; j < Hmatrix->ncol; j++) {                     // and for each column
            matrix_array->data[i*Hmatrix->nrow+j] = Hmatrix->data[i][j];    // insert matrix elements into the
            //printf("%f    ", matrix_array->data[i*Hmatrix->nrow+j]);          // float-array matrix_array->data
        }
    }

    //LAPACKE_ssyev(int matrix_layout, char jobz, char uplo, lapack_int n, float* a, lapack_int lda, float* w);
    INFO = LAPACKE_ssyev(LAPACK_COL_MAJOR, 'V', 'L', dim, matrix_array->data, dim, spectrum->eigvals);
    if (INFO > 0) {
        printf("the algorithm failed to compute eigenvalues.\n");
        exit(1);
    }

    // copy eigenvectors from matrix_array->data[] to eigvector_matrix->data[][]
    for (int i = 0; i < Hmatrix->nrow; i++) {
        for (int j = 0; j < Hmatrix->ncol; j++) {
            spectrum->eigvectors->data[i][j] = matrix_array->data[i*Hmatrix->nrow+j];
        }
    }

    return spectrum;
}
```

L A P A C K
L -A P -A C -K
L A P A -C -K
L -A P -A -C K
L A -P -A C K
L -A -P A C -K

ALGORITMO DE SIMULACIÓN

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <math.h>
#include <lapacke.h>
#include "include/def.h"
#include <time.h>

int main(int argc, char **argv) {

    time_t t_0, t_1;
    t_0 = time(NULL);

    int length = 10;
    float g = 2;

    tsbasis * sbasis = init_sbasis(length);
    tvbasis * vbasis = init_vbasis(length);
    basis_states(sbasis, vbasis);

    tsmatrix * Hmatrix = H(sbasis, g);
    tspectrum * spectrum = eigenvalues(Hmatrix);
    float M = magnetization_expectation_value(vbasis,
                        spectrum->eigvectors->data[0],
                        length);

    t_1 = time(NULL);
    printf("Timelapse: %ld seconds\n", (t_1 - t_0));

    return 0;
}
```

i. Base vectorial de estados

ii. Hamiltoniano cuántico

iii. Espectro de energía

iv. Valor esperado de la magnetización

ALGORITMO DE SIMULACIÓN

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <math.h>
#include <lapacke.h>
#include "include/def.h"
#include <time.h>

int main(int argc, char **argv) {

    time_t t_0, t_1;
    t_0 = time(NULL);

    int length = 10;
    float g = 2;

    tsbasis * sbasis = init_sbasis(length);
    tvbasis * vbasis = init_vbasis(length);
    basis_states(sbasis, vbasis);

    tsmatrix * Hmatrix = H(sbasis, g);
    tspectrum * spectrum = eigenvalues(Hmatrix);
    float M = magnetization_expectation_value(vbasis,
                        spectrum->eigvectors->data[0],
                        length);

    t_1 = time(NULL);
    printf("Timelapse: %ld seconds\n", (t_1 - t_0));

    return 0;
}
```

```c
float magnetization_expectation_value(tvbasis *vbasis, float *state, int length) {
    float M = 0;

    for (int i = 0; i < vbasis->nvectors; i++) {
        M += pow(state[i], 2)*abs(2*vbasis->nup[i]- length);
    }

    M /= (float) length;

    return M;
}
```

$$\boldsymbol{v} = (v_1, v_2, \ldots, v_N) = v_1\,\boldsymbol{e}_1 + v_2\,\boldsymbol{e}_2 + \cdots + v_N\,\boldsymbol{e}_N$$

$$\langle n_\uparrow, n_\downarrow |\, M \,| n_\uparrow, n_\downarrow \rangle = \frac{1}{2}(n_\uparrow - n_\downarrow)\,\langle n_\uparrow, n_\downarrow | n_\uparrow, n_\downarrow \rangle \quad , \quad n_\downarrow = N - n_\uparrow$$

FASES CUÁNTICAS EN EL MODELO DE ISING TRANSVERSO

Simetría Z_2 del modelo de Ising transverso

El hamiltoniano del TFIM es invariante bajo las transformaciones unitarias de volteo de espín,

$$\hat{\zeta} = \prod_i \hat{\sigma}_i^x,$$

$$\hat{\zeta}|\uparrow\uparrow\downarrow\cdots\rangle = |\downarrow\downarrow\uparrow\cdots\rangle \quad , \quad \hat{\zeta}^2|\uparrow\uparrow\downarrow\cdots\rangle = |\uparrow\uparrow\downarrow\cdots\rangle$$

que corresponde a una simetría Z_2 del sistema.

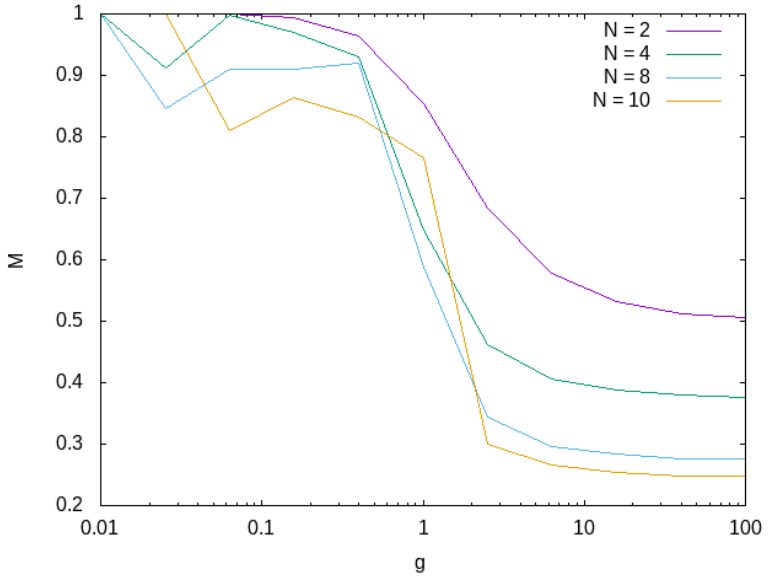

Figura 8: Transición de fase en el TFIM modulada por la magnetización.
Simulación por ordenador propia.

Fase desordenada (paramagnética)

Cuando g >> 1, sobrevive el **término de campo** y el sistema posee un único estado fundamental,

$$|\psi_0\rangle = |\rightarrow\rightarrow\rightarrow\cdots\rangle,$$

que **preserva la simetría Z_2** del sistema. El término perturbativo mueve una excitación de cuasipartícula a sus primeros vecinos,

$$\langle\psi_i|\,\hat{V}\,|\psi_j\rangle = -J(\delta_{j,i-1} + \delta_{j,i+1})$$

Fase ordenada (ferromagnética)

Cuando g << 1, predomina la **correlación de espines** y el sistema posee un estado fundamental con degeneración,

$$|\psi_\uparrow\rangle = |\uparrow\uparrow\uparrow\cdots\rangle \quad , \quad |\psi_\downarrow\rangle = |\downarrow\downarrow\downarrow\cdots\rangle$$

que **rompe la simetría Z_2** del sistema, y desaparece con teoría perturbativa. El término de campo mueve fronteras de dominio a próximos vecinos,

$$\phi_{i+\frac{1}{2}} = |\cdots\uparrow \underbrace{\uparrow}_{i} \vdots \underbrace{\downarrow}_{i+1} \downarrow\cdots\rangle$$

DEL MODELO DE ISING

A LA CADENA DE KITAEV

TRANSFORMACIONES DE JORDAN-WIGNER

De espines a fermiones

Aplicamos una transformación de Jordan-Wigner, que define unos operadores no locales que recuperan el álgebra de Grassmann,

$$\hat{c}_i = \prod_{j<i}(\hat{\sigma}_j^z)\hat{\sigma}_i^+ \quad , \quad \hat{c}_i^\dagger = \hat{\sigma}_i^- \prod_{j<i}(\hat{\sigma}_j^z)$$

$$\hat{c}_i\hat{c}_j = -\hat{c}_j\hat{c}_i \quad , \quad \hat{c}_i^2 = (\hat{c}_i^\dagger)^2 = 0 \quad , \quad \hat{c}_i\hat{c}_i^\dagger + \hat{c}_i^\dagger\hat{c}_i = 1$$

Hallando su transformación inversa,

$$\hat{\sigma}_i^x = 1 - 2\hat{c}_i^\dagger\hat{c}_i \quad , \quad \hat{\sigma}_i^z = -\prod_{j<i}(1-2\hat{c}_j^\dagger\hat{c}_j)(\hat{c}_i + \hat{c}_i^\dagger)$$

podemos reescribir el hamiltoniano del TFIM como

$$\hat{H}_{\text{TFIM}} = -J\sum_i \left(g - 2g\hat{c}_i^\dagger\hat{c}_i + \hat{c}_{i+1}\hat{c}_i + \hat{c}_i^\dagger\hat{c}_{i+1}^\dagger + \hat{c}_i^\dagger\hat{c}_{i+1} + \hat{c}_{i+1}^\dagger\hat{c}_i\right)$$

Condiciones de contorno

Atendiendo a las condiciones de contorno del sistema de Ising, si éste posee **condiciones periódicas**, la cadena de fermiones tendrá condiciones periódicas (antiperiódicas) si el número de fermiones es par (impar).

Una **cadena abierta** de espines se mapea sobre una cadena abierta de espines. Sólo bajo estas condiciones aparecerán *Majorana zero-modes* (MZM) relacionados con estados de borde con **degeneración topológica**.

Este sistema se corresponde con el **hamiltoniano de Kitaev** de superconductores *p-wave*,

$$\hat{H}_{\text{Kitaev}} = \sum_{j=1}^{N}\left[\underbrace{-\frac{t}{2}(\hat{c}_{j+1}^\dagger\hat{c}_j + \hat{c}_j^\dagger\hat{c}_{j+1})}_{\text{enlace fuerte}} - \underbrace{\mu\hat{c}_j^\dagger\hat{c}_j}_{\text{potencial químico}} + \underbrace{\frac{\Delta}{2}(\hat{c}_{j+1}^\dagger\hat{c}_j^\dagger + \hat{c}_j\hat{c}_{j+1})}_{\text{superconducción } p\text{-wave en campo medio}}\right]$$

con potencial químico igual a 2g, y un término de *hopping* y un gap superconductor iguales a 2.

Transformación de Fourier

Podemos representar los operadores fermiónicos en la base de momentos, tal que

$$\hat{c}_k = \frac{1}{\sqrt{N}} \sum_j \hat{c}_j e^{ikx_j}$$

Así, podemos definir un operador de paridad partícula-hueco que da valor de la simetría del sistema,

$$\hat{\mathcal{P}} = \prod_k \hat{\tau}_k^x \hat{\kappa} \quad , \quad \hat{\tau}^x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Diagonalización

Ahora, podemos aplicar una transformación de Bogoliubov, definiendo nuevos operadores fermiónicos de cuasipartículas

$$\hat{\gamma}_k = u_k \hat{c}_k - i v_k \hat{c}_{-k}^\dagger \quad , \quad \hat{\gamma}_k^\dagger = u_k^* \hat{c}_k^\dagger + i v_k^* \hat{c}_{-k}$$

de tal forma que el hamiltoniano de Ising queda diagonalizado,

$$\hat{H}_{\text{diag}} = \sum_k \epsilon_k \left(\hat{\gamma}_k^\dagger \hat{\gamma}_k - \frac{1}{2} \right) \quad , \quad \epsilon_k = 2J\sqrt{1 + g^2 - 2g\cos(k)}$$

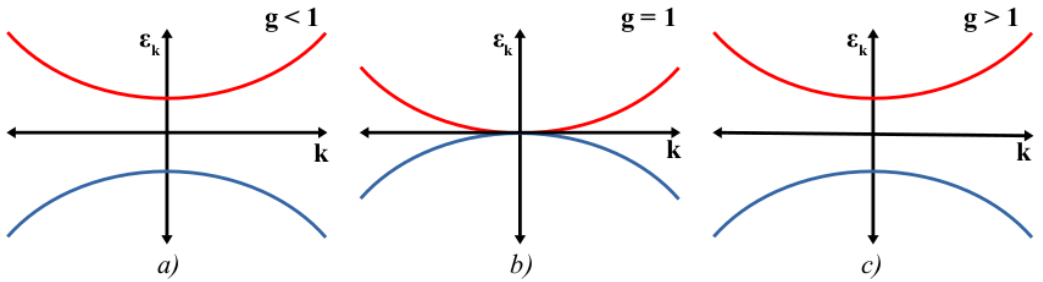

Figura 9: Relación de dispersión con simetría partícula-hueco.
[6] K. Chhajed, Res. 26 (2021).

Fermiones de Majorana

Se definen formalmente los operadores de Majorana como

$$\hat{a}_j = \hat{c}_j^\dagger + \hat{c}_j \quad , \quad \hat{b}_j = i(\hat{c}_j - \hat{c}_j) \qquad \hat{a}_j^\dagger = \hat{a}_j, \hat{b}_j^\dagger = \hat{b}_j$$

tales que, en términos de estos operadores, el hamiltoniano de Ising toma la forma

$$\hat{H}_{\text{Majorana}} = iJ \sum_j \left(\hat{a}_j \hat{b}_{j+1} + g \hat{a}_j \hat{b}_j \right)$$

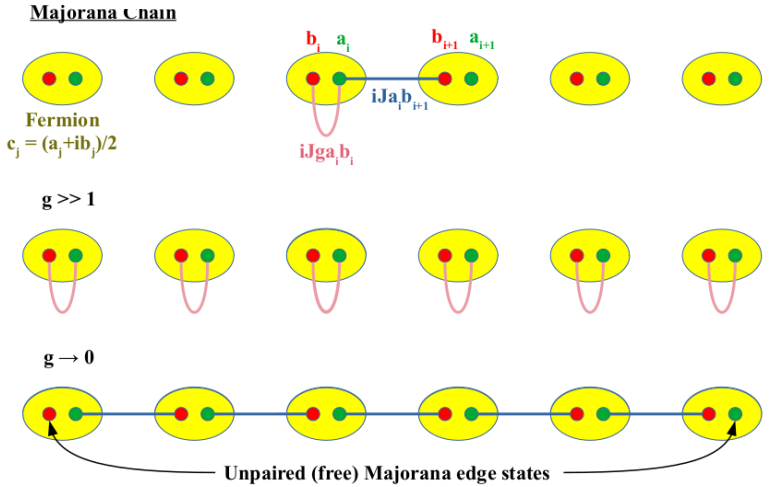

Figura 10: Interpretación de una cadena de fermiones mediante operadores de Majorana y sus diferentes fases topológicas.
[4] K. Chhajed, Resonance, vol. 26 (2021).

Estados de borde no triviales

En el límite cuando g tiende a cero, aparecen modos de Majorana desapareados de energía cero que construyen conjuntamente un fermión no local,

$$\hat{d}^\dagger = \frac{\hat{b}_0 + i\hat{a}_N}{2}$$

tal que se definen los dos estados fundamentales degenerados del sistema a partir de éste como *qubits*,

$$|\uparrow\rangle = \hat{d}^\dagger |0\rangle \quad , \quad |\downarrow\rangle = |0\rangle$$

Y AHORA...

FERMIONES