3 Numerical calculation for integro-differential equation

The integro-differential equation central to the analysis of our model was solved numerically using C++ code. This section outlines the key computational techniques employed within the code.

3.1 Structure of C++ code

The structure of the C++ code designed to solve the integro-differential equation is outlined below. The C++ code for solving the integro-differential equation is structured as follows: the functions performing the calculations are declared beforehand within a header file (.hpp) using a class, and then defined within a source file (.cpp). Appropriate numerical methods were adopted for each computational step.

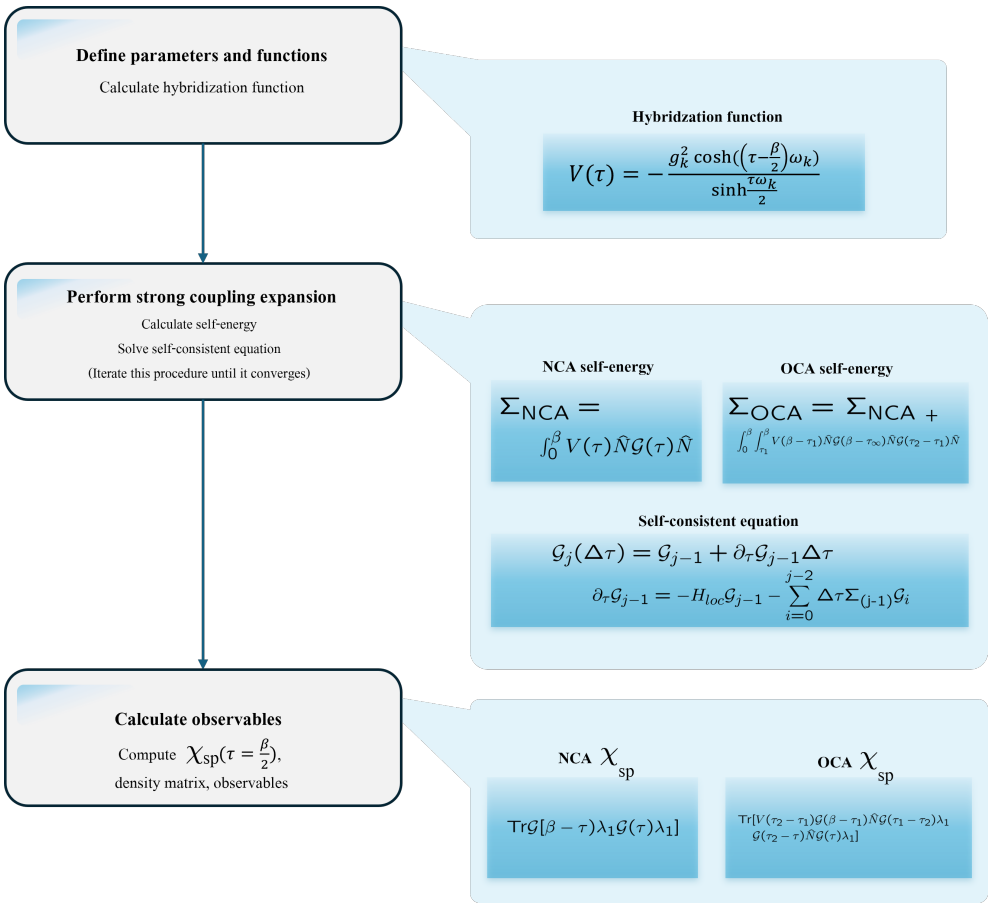

Figure 9: The flowchart of Program structure

3.2 Hybridzation function - Simpson's Rule

The hybridization function, calculated as a scalar rather than a matrix, was evaluated by considering the form of the retarded Matsubara Green's function. To compute the value of g_k, a parameter that determines the coupling between the RC circuit and the Josephson junction, Simpson's rule of integration was employed. The formula for

calculating g_k for the application of this integration method is given below:

$$g_k = \sqrt{\frac{2W}{\alpha}\left(\frac{W}{1+(\nu x)^2}\right)} \quad , \quad (x = \frac{k}{W}) \tag{3.1}$$

The integration method using Simpson's method follows:

$$\int f(x)dx = \frac{b-a}{3n}\left[f(x_0) + \sum_{i=odd} 4f(x_i) + 2\sum_{i=\text{even}} f(x_i) + f(x_0) + f(x_f)\right] \tag{3.2}$$

Here, we calculate the initial value using the case of $V(\tau, k = 0)$. The part where the Simpson's rule is implemented in the code is as follows. After generating a vector array suitable for the length of the k-index to calculate V(), the integration is performed only for the k-value for g_k and ω_k in the V() expression. Using the resulting expression, the value for each interval is stored in an array of length .

3.3 Trapezoidal method

The trapezoidal method was employed for the calculation of the self-energy and the propagator. This section provides a brief overview of the implementation for calculating the propagator. The numerical integration formula using the trapezoidal method is as follows:

$$\int_a^b f(x)\,dx \approx \frac{\Delta x}{2}\left[f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{n-1}) + f(x_n)\right] \tag{3.3}$$

To correspond to the actual numerical integration program code, the formula can be rewritten in the form of an infinite series as follows:

$$\int_a^b f(x)\,dx \approx \Delta x\left[\frac{1}{2}f(x_0) + \sum_{j=1}^{n-1} f(x_j) + \frac{1}{2}f(x_n)\right] \tag{3.4}$$

And the self-consistent equation we aim to solve is as follows:

$$\mathcal{G}_j = \mathcal{G}_{j-1} + \partial_\tau \mathcal{G}_{j-1}\Delta\tau \tag{3.5}$$

When implementing the trapezoidal method, it's crucial to follow a specific sequence of calculations:

1. First, update the value of $\partial_\tau \mathcal{G}_i$ using $\mathcal{G}_i$, which corresponds to each $f(x_i)$.
2. Next, calculate the temporary value of $\mathcal{G}'_{i+1}$ using the updated $\partial_\tau \mathcal{G}_i$.
3. Then, calculate $\partial_\tau \mathcal{G}_{i+1}$using the temporary value of $\mathcal{G}'_{i+1}$ calculated in the previous step.
4. Finally, perform the final trapezoidal integration calculation using $\frac{1}{2}(\partial_\tau \mathcal{G}_i + \partial_\tau \mathcal{G}_{i+1})$.

To implement this in the program, we introduced two arrays: P-array to store the final calculation result $\mathcal{G}_i$, and S-array to store the temporary value calculated in step 2. We also devised a method to use the value of S-array in step 3. This process is visually represented in the diagram below, where:

This is implemented in the code as follows:

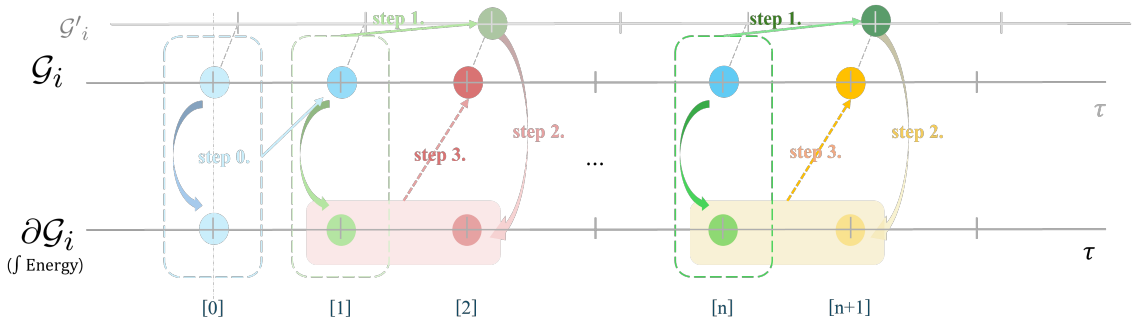

Figure 10: Image depicting the procedure of Trapezoidal process

```cpp
vector<MatrixXd> MD_OC::Propagator(const vector<MatrixXd>& sig, const MatrixXd& loc)
{
    vector<MatrixXd> P_arr(t, MatrixXd::Zero(siz,siz)); // Declaration of an array to store the final calculation result
    vector<MatrixXd> S_arr(t, MatrixXd::Zero(siz,siz)); // Declaration of an array to store temporary results during calculation

    P_arr[0] = MatrixXd::Identity(siz,siz);
    S_arr[0] = MatrixXd::Identity(siz,siz);

    MatrixXd sig_form = MatrixXd::Zero(siz,siz); // A matrix to temporarily store the result of process 1.
    MatrixXd sig_late = MatrixXd::Zero(siz,siz); // A matrix to temporarily store the result of process 3.


    for (int i = 1; i < t; i++)
    {
        P_arr[1] = P_arr[0];
        sig_late = 0.5 * Delta_t * (0.5 * Delta_t * (sig[1] * P_arr[0] + sig[0] * (P_arr[0] + Delta_t * P_arr[0])));
        P_arr[1] = P_arr[0] - 0.5 * Delta_t * loc * (2 * P_arr[0] + Delta_t * P_arr[0]) + sig_late;
        S_arr[1] = P_arr[1];

        if (i > 1)
        {
            sig_form = round_propagator_ite(loc, sig, P_arr, i - 1, 0);
            S_arr[i] = P_arr[i - 1] + Delta_t * sig_form; // Calculation process of 2. is performed in this part.

            sig_late = 0.5 * Delta_t * (round_propagator_ite(loc, sig, P_arr, i - 1, 1) + round_propagator_ite(loc, sig, S_arr, i, 1)); // This part is the result of performing trapezoidal calculation for the integral calculation in the round propagator.
            P_arr[i] = P_arr[i - 1] - 0.5 * Delta_t * loc * (2 * P_arr[i - 1] + Delta_t * sig_form) + sig_late; // This part is the result of performing trapezoidal calculation for
```

```
     the integral calculation in the round propagator.

        }
    }

    return P_arr;
}
```

Listing 1: Trapezoidal calculation code

3.4 Iteration truncation - Relative entropy

Our targeting integro-differential equation has a recursive structure, which means that a single calculation is not enough to get a complete result. The obtained array of $G(\tau)$ in the previous iteration is to be used as an initial condition for multiple iterative calculations until the result converges sufficiently. To truncate the iteration of calculations in flexibility, the concept of relative entropy was introduced into the code. The formula for calculating relative entropy is as follows:

$$D(p||q) = \sum_{x \in \mathcal{X}} p(x) \ln \frac{p(x)}{q(x)}$$

The notations $p(x)$ and $q(x)$ represent two different probability distributions. In the case of $\mathcal{G}(\tau = \beta)$ in the interval $[0, \beta]$, The Green's function is represented by the same formula as the partition function of a thermal equilibrium state in the grand canonical ensemble. In this case, the partition function plays the same role as the density operator, which represents the mixed state of the system being investigated from a quantum mechanical perspective. This allows us to calculate the probability distribution in the final stage. The code was designed to terminate the calculation of G() when the distance between the probability distributions obtained at each calculation step becomes sufficiently small, indicating that the result has converged.

```
// Function that returns the value of \mathcal{G}(\tau=\beta)
vector<double> MD_OC::temp_itemin(vector<MatrixXd> &arrr, double minpo, int size)
{
    vector<double> dist_return(size,0);
    for (int i = 0 ; i < size; i++){
        dist_return[i] = arrr[t-1](i,i); // Value of \mathcal{G} at \tau = \beta
    }
    return dist_return;
}

// Function that processes recursive calculation.
vector<MatrixXd> MD_OC::Iteration(const int& n)
{
    // ~ Calculation part
    ////////////////////////////////////////////////////////////
    double temp_minpoin;
    vector<vector<double> > temp_itemi(2,vector<double>(siz,0)); // Arrays to store the
    results of the previous and current iteration. siz represents the dimension of the
    calculated square matrix.
```

```
    double RELA_ENTROPY;
    ////////////////////////////////////////////////////////////////
    for (int i = 0; i <= n; i++){
        // ~ Calculation part
            ////////////////////////////////////////////////////////////////////////////////

            temp_minpoin = //beta value;

            ////////////////////////////////////////////////////////////////////////////////
       // ~ Calculation part
           ////////////////////////////////////////////////////////////////////////////////

            temp_itemi[(i-1)%2] = temp_itemin(Prop,temp_minpoin,siz); // temporary store previous iteration data
            RELA_ENTROPY = 0;

            ////////////////////////////////////////////////////////////////////////////////
      // ~ Calculation part
            ////////////////////////////////////////////////////////////////////////////////

            temp_itemi[i%2] = temp_itemin(Prop,temp_minpoin,siz); // temporary store present iteration data

            // Relative entropy calculation

            for (int j = 0; j < siz; j++){
                RELA_ENTROPY += temp_itemi[i%2][j] * log(temp_itemi[i%2][j]/temp_itemi[(i-1)%2][j]);
            }


            if (i > 1){
                cout << "\t""\t" << i << " th Iteration stop value : " << fabs(RELA_ENTROPY) << endl;
                if (fabs(RELA_ENTROPY) < 0.00001){
                    break; // If the distance between the probability distributions of the previous and current steps is less than 0.00001, the calculation is terminated.
                }
            }
            ////////////////////////////////////////////////////////////////////////////////
        }
    return //(Calculation result);
}
```

Listing 2: Iteration truncation code

3.5 Implementation of Correlation function

The T-matrix was first introduced in the process of representing the OCA self-energy formula in diagrammatic form. Implementing this in a computational setting leads to a reduction in the calculation time of the approximation method. Based upon this idea, we construct the matrix structure $\hat{T}_{chi}$ to pre-calculate the product of the propagator and the N matrix, store it in matrix form, and then use the values from the matrix elements as needed during the self-energy calculation.

Correlation function calculation

In the case of OCA, a total of 8 matrix multiplications are required, where total calculation trial is upon $\mathcal{O}(n^n)$ This can be simply represented as follows:

$$\chi_{sp} = V_{mn}\mathrm{Tr}(\hat{\mathcal{G}}_{k-m-1} * \hat{N} * \hat{\mathcal{G}}_{m-i} * \hat{\lambda}_1 * \hat{\mathcal{G}}_{i-n} * \hat{N} * \hat{\mathcal{G}}_n * \hat{\lambda}_1) \tag{3.6}$$

If the part corresponding to $\hat{N} * \hat{\mathcal{G}}_i * \hat{N} * \hat{\mathcal{G}}_j * \hat{\lambda}_1$ is pre-computed and stored in matrix form for later use in the calculation, the eight matrix multiplications are reduced to a single matrix multiplication.

$$\hat{T}_{chi} = \hat{\mathcal{G}}_{i'} * \hat{N} * \hat{\mathcal{G}}_{j'} * \hat{\lambda}_1 \tag{3.7}$$

$$\chi_{sp} = \mathrm{Tr}(\hat{T}_{chi} * \hat{T}'_{chi}) \tag{3.8}$$

χ_{sp} T-matrix $\hat{T}_{chi}$: k=3 case

The code for the general formula to calculate χ_{sp} before introducing the T-matrix is as follows:

```
for (int i=0; i<k; i++)
{
    MatrixXd Stmp = MatrixXd::Zero(3,3);
      for (int n=0; n<=i; n++) for (int m=i; m<k; m++)
      {
         Stmp += V[m-n] * Prop[k-m-1] * N * Prop[m-i] * GELL
                   * Prop[i-n] * N * Prop[n] * GELL;
      }
      OCA_chi_array0[i] = pow(Delta_t,2)*Stmp.trace();
}
```

Listing 3: Full One-crossing Approximation implementation code

In the above equation, for the case of k=3, the number of calculation cases according to n, m, and i is as follows: Here, we can see that the total number of calculation cases can be represented by selecting and arranging two numbers from the three numbers 2, 1, and 0. To achieve this, we can introduce the following matrix: For example, if we want to represent all the possible cases as shown above, using the matrix above and calculating as below, we obtain all possible calculation cases.

$$\hat{G}_{k-m-1} * \hat{N} * \hat{G}_{m-i} * \hat{\lambda}_1 * \hat{G}_{i-n} * \hat{N} * \hat{G}_n * \hat{\lambda}_1$$

(K=3)

$_3P_2$

i=0 → n=0 → m=0	**[2][0][0][0]**
m=1	**[1][1][0][0]**
m=2	**[0][2][0][0]**
i=1 → n=0 → m=1	**[1][0][1][0]**
→ m=2	**[0][1][1][0]**
→ n=1 → m=1	**[1][0][1][0]**
i=2 → n=1 → m=2	**[0][1][0][1]**
→ n=0 → m=2	**[0][0][2][0]**
→ n=1 → m=2	**[0][0][1][1]**
→ n=2 → m=2	**[0][0][0][2]**

Figure 11: Considering the case of K=3, where the total indices to be calculated are $[0, 1, 2]$ -in the code, K matches the number of grids in τ grid, the total possible calculation order is as shown on the right. For example, in the first line, the calculation proceeds in the order of $\mathcal{G}[2]\hat{N}\mathcal{G}[0]\lambda_1\mathcal{G}[0]\hat{N}\mathcal{G}[0]\lambda_1$. When examining the listed order on the right, we can see that the order of repeated numbers is listed, which is the same as the number of cases where two numbers are selected and arranged from $[0, 1, 2]$.

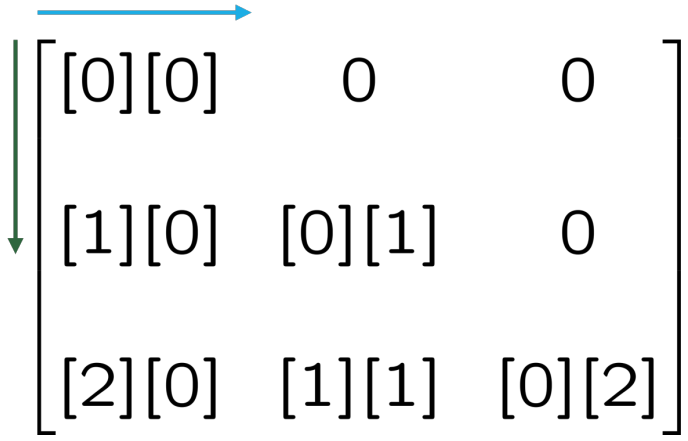

Figure 12: Therefore, a matrix to store the calculation cases corresponding to the number of ways to choose two numbers from [0, 1, 2] is created as follows.

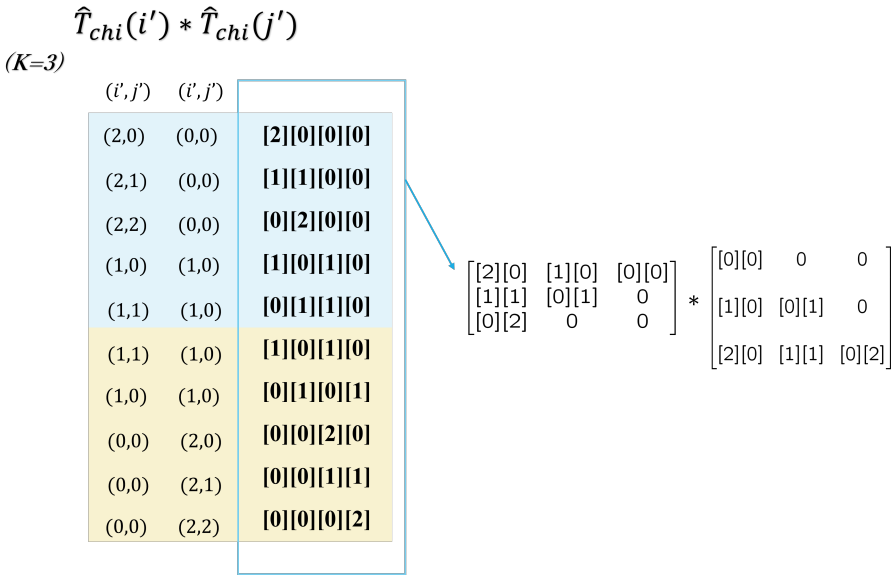

Figure 13: With the introduction of the above matrix, the overall calculation cases can be expressed in all forms through matrix multiplication.