About

Code for paper: C. Zhang, M. Gruebele, D. E. Logan, and P. G. Wolynes, Surface Crossing and Energy Flow in Many-Dimensional Quantum Systems, Proc. Natl. Acad. Sci. U.S.A. 120, e2221690120 (2023) https://doi.org/10.1073/pnas.2221690120

Note about code

How to configure and run the code:

1. Make sure folder name of project is the same as that appear in CMAKEList.txt.

   For example if in CMAKEList.txt you write:

   (1) project(my_project_LW_model)

   Then the folder name must be : **my_project_LW_model**

2. in util.h

in my machine, I have #include<mpi/mpi.h>. This may be problematic on cluster or your computer. May change it to #include <mpi.h>

3. ```
   cd SOURCE_DIRECTORY/Release
   cmake ..
   make
   mpirun -np 10 ./executable (10 is number of process you want to use
   change it to different number if you want to use more or less)
   ```

4. How to compile the project

   check : c++ - Debug vs Release in CMake - Stack Overflow
   ```

```
# create file for debugging
mkdir Debug
cd ./Debug
cmake -DCMAKE_BUILD_TYPE=Debug ..
make

# create file for computing (production usage)
mkdir Release
cd Release
cmake -DCMAKE_BUILD_TYPE=Release ..
make
```

Structure of Code

We follow the order that program execute the functions :

Full_system(string path1)

```
full_system::full_system(string path1 , vector<vector<int>> &
initial_state_quantum_number) {
    path = path1;
    d.path = path;
    // read hyper parameter and time step from input.txt
    read_input_with_MPI();

    s.read_MPI(input, output, log);
    d.read_MPI(input, output, s.exciton_state_num, path);
    d.construct_initial_state_MPI( initial_state_quantum_number);

    compute_monomer_vib_state_basis_set_size_MPI();

    d.construct_monomer_Hamiltonian_MPI(input, output, log,
monomer1_vib_state_energy_all_pc, monomer2_vib_state_energy_all_pc,
                                        monomer_qn_list0,
                                        monomer_qn_list1);

    d.monomer1_vib_state_energy_all_pc = monomer1_vib_state_energy_all_pc;
    d.monomer2_vib_state_energy_all_pc = monomer2_vib_state_energy_all_pc;

    construct_dimer_Hamiltonian_matrix_with_energy_window_MPI();

    if(my_id ==0){
        cout<<"Finish constructing Matrix"<<endl;
        output_calculation_size_info(); // check if all matrix's dimension is
right.
    }

}
```

This function construct Hamiltonian H for our systems.

1 . s.read_MPI() , d.read_MPI()

s.read_MPI() , d.read_MPI() : read parameter from input file

(only process with rank 0 read the parameters and then broadcast parameter to all other process.

- **You can't let multiple process access one file at the same time, this may cause error.**
- **Make sure parameters read from input.txt are broadcasted to all other process, otherwise the simulation in other process will raise error because they don't know**

the value of the parameter !!!

)

2. construct_initial_state_MPI()

compute vibrational state energy of initial vibrational state we prepared in calculation.

Notice there is also self-anharmonicity we induced in Hamiltonian to avoid resonance.

3. compute_monomer_vib_state_basis_set_size_MPI( )

```
void full_system:: compute_monomer_vib_state_basis_set_size_MPI( ){
    if(my_id==0){
        int i, j, k;
        int i1, i2;
        double  detector0_energy, detector1_energy;
        // monomer0_qn and monomer1_qn indicate current monomer mode index we
are calculating energy.
        vector<int> monomer0_qn(d.nmodes[0]);
        vector <int> monomer1_qn(d.nmodes[1]);

        // record size of total matrix
        int location;
        bool exist=false;

        int state_space_distance;
......
```

Construct basis set for Hamiltonian:

```
 monomer0_qn[0] = -1; // this is for:  when we go into code: monomer0_qn[i]=
monomer0_qn[i]+1, our first state is |000000>
 while (1) {
            label2:;  // label2 is for detector0 to jump to beginning of
while(1) loop (this is inner layer of while(1))

             for (i1 = 0; i1 < d.nmodes[0]; i1++) {  // loop through detector0
                // define the way we loop through detector0:
                monomer0_qn[i1] = monomer0_qn[i1] + 1;
                if (monomer0_qn[i1] <= d.nmax[0][i1]) break;
                if (monomer0_qn[d.nmodes[0] - 1] > d.nmax[0][d.nmodes[0] - 1])
{
                    monomer0_qn[d.nmodes[0] - 1] = 0;
                    goto label1;  // use goto to jump out of nested loop
                }
                monomer0_qn[i1] = 0;
              }

 }
```

This code goes through all vibrational state at electronic state 0:

$|n_1, n_2, \cdots\rangle$, covering all state whose quantum number < n_{max} . We first increment index 1 , then index 2 , then index 3.

$$|0,0,\cdots\rangle \rightarrow |1,0,\cdots\rangle \rightarrow \cdots \rightarrow |n_{max},0,\cdots\rangle \rightarrow |0,1,\cdots\rangle \rightarrow \cdots \rightarrow |n_{max},1,\cdots\rangle$$

```
     // calculate monomer 0 energy
     for (i = 0; i < d.nmodes[0]; i++) {
        if (self_anharmonicity_bool){
            // add self-anharmonicity
             monomer0_energy = monomer0_energy + d.mfreq[0][i] *
(monomer0_qn[i] - pow(monomer0_qn[i], 2) * d.mfreq[0][i] / (4 *
self_anharmonicity_D) );
            }
        else{
             monomer0_energy = monomer0_energy + d.mfreq[0][i] * monomer0_qn[i]
;
        }
     }
```

compute monomer 0 vibrational state energy.

```
            //-------------------------------------------------------------------
---------------------------
           // criteria below make sure monomer 0 's energy is reasonable.
            if (monomer0_energy > d.initial_state_energy[0] +
d.vibrational_energy_window_size) {
                // monomer 0's energy can not be larger than its initial energy
+ photon energy
                // jump to next monomer state.
                k = 0;
                while (monomer0_qn[k] == 0) {
                    monomer0_qn[k] = d.nmax[0][k];
                    k++;
                    if (k >= d.nmodes[0]) {
                        break;
                    }
                }
                if (k < d.nmodes[0]) {
                    monomer0_qn[k] = d.nmax[0][k];
                }
                goto label2;
            }
```

This will speed up process of going through vibrational state space.

Because of the sequence we choose to go through state in state space, once energy is higher , then we can set all nonzero index to 0 and next 0 index to 1, for example :

$$|1,2,3,0,1,\cdots\rangle \rightarrow |0,0,0,1,1,\cdots\rangle$$

```
            // criteria below make sure monomer 0 can not be too far away from
bright state and lower bright state.
       state_space_distance =
                    compute_state_space_distance(monomer0_qn,
d.initial_vibrational_state[0], d.nmodes[0]);

            // we use distance constraint for state whose energy is between two
            if (state_space_distance > Rmax) {
                goto label2;
            }
```

We require quantum state we incorporate satisfy $|n - n_{init}| < R_{max}$

To facilitate future reference of state, we also order state according to their quantum number.

find_position_for_insert_binary is used to order the state according to q.n.

4. construct_monomer__Hamiltonian_matrix()

```
void monomer:: construct_monomer_Hamiltonian_MPI(ifstream & input, ofstream &
output, ofstream & log, vector<double> & dmat_diagonal_global0, vector<double>
& dmat_diagonal_global1, vector<vector<int>> & vmode0, vector<vector<int>> &
vmode1) {

    // previously information for state space is in process 0. Now we broadcast
this information to all procerss.
    construct_dv_dirow_dicol_dmatrix_MPI(log, dmat0, dmat1, monomer_qn_list0,
monomer_qn_list1);

    // compute index for initial state (here state at crossing region) in
Hamiltonian.
    compute_initial_vibrational_state_index();


compute_monomer_offdiag_part_MPI(log,dmat0,dmat1,monomer_qn_list0,monomer_qn_li
st1);
 ....
```

4.1 compute_initial_vibrational_state_index()

We know in some cases, the initial state we choose in program may be important for evaluating some quantity, for example, here we need to compute the survival probability from the initial vibrational states. Therefore, we compute state index in state list according to q.n. of initial state.

4.2 compute off diagonal matrix of monomer__Hamiltonian

```
void monomer::compute_monomer_offdiag_part_MPI(ofstream & log,vector<double> &
dmat0,  vector<double> & dmat1,  vector<vector<int>> & monomer_qn_list0,
vector<vector<int>> & monomer_qn_list1)
```

The Hamiltonian we compute is the local random matrix Hamiltonian:

$$\hat{H}_{\mathrm{a}} = \sum_{m} \prod_{\alpha} V_m \left(b_\alpha^\dagger\right)^{m_\alpha^+} b_\alpha^{m_\alpha^-}.$$
$$V_m = V_3 a^{m-3}$$

Here V_m is the strength of anharmonic coupling of m^{th} order. a is the scaling factor.

In the code, aij[m][k] is the scaling factor for monomer m , mode k.

if (ntot % 2 == 0) { value = V_intra; // anharmonic coupling V0. V3 = V0 * a^3. (a is anharmonic scaling factor) } else { value = -V_intra; } for (k = 0; k < nmodes[m]; k++) { value = value * pow(aij[m][k]* nbar[k], deln[k]); }

Here ntot = $\sum_i \Delta n_i$. 1-norm distance between two vibrational states.

nbar[i] : = $(n_{a,i}n_{b,i})^{1/4}$, here this is square of geometric mean of quantum number in mode i for state a and state b (we are computing anharmonic coupling between state a and state b)

deln[i]: Δn_i . quantum number difference between two states along mode i.

$V_m = V_0 \prod_i (a_i \sqrt{(n_{a,i}n_{b,i})^{1/2}})^{\Delta n_i}$

Here $V_0 = 300$, $a_i = \sqrt{f_i}/270$

For the choice of this value, see Bigwood et al PNAS paper : The vibrational energy flow transition in organic molecules: Theory meets experiment | PNAS

5. construct dimer Hamiltonian matrix

```
void full_system::construct_dimer_Hamiltonian_matrix_with_energy_window_MPI() {
    int i;

    // compute diagonal part of dimer Hamiltonian.
    compute_sstate_dstate_diagpart_dirow_dicol_MPI();

    // sort (exciton_state, vib_mode1, vib_mode2) into groups representing
states with the same (exciton_state, vib_mode1) or (exciton_state, vib_mode2).
    construct_quotient_state_all_MPI();

    compute_full_Hamiltonian_offdiagonal_part_MPI();
}
```

This function construct the Hamiltonian for dimer (here we refer as full system).

5.1 construct dimer states

```
// compute diagonal part of dimer Hamiltonian.
    compute_sstate_dstate_diagpart_dirow_dicol_MPI();
```

- compute exciton state energy as diagonal part of dimer Hamiltonian matrix.

Notice here by our definition, irow, icol should be global matrix index across different process. Therefore, you will see the code in this function add offset to irow, icol value later.

```
                energy = s.exciton_state_energy[i] +
monomer1_vib_state_energy_all_pc[j] + monomer2_vib_state_energy_all_pc[k]; //
energy of electronic state (s.tlmat[i]) + energy in two
monomer(d.monomer1_vib_state_energy_all_pc[j] +
d.monomer2_vib_state_energy_all_pc[k]).

                mat.push_back(energy);  // diagonal part is energy of
vibrational state
                irow.push_back(mat_index); //mat_index is local, we have to re-
compute it after all process compute its own irow, icol. See code below
                icol.push_back(mat_index);
```

- record exciton state for given dimer state in exciton_state_index_list
- record vibrational state index for dimer 0 and dimer 1

```
                exciton_state_index_list.push_back(i);
                vibrational_state_index_list[0].push_back(j); //
vibrational_state_index_list record monomer index across process (global index
, not index in each process)
                vibrational_state_index_list[1].push_back(k);
```

5.2 construct quotient state

See quotient.h for how we define quotient state in this program:

```
struct quotient_state {   // monomer quotient_space_state.

    // each quotient_state is defined for pair (exciton_state_index_list,
vmode), here vmode is vibrational mode in another monomer
    // exciton_state_index_list is exciton state ( 0 or 1).
    // states (exciton_state_index_list, vmode1, vmode2) is grouped into
different group according to (exciton_state_index_list, vmode2) for
monomer1_quotient_state_list and (exciton_state_index_list, vmode1) for
monomer2_quotient_state_list
    // monomer1_quotient_state_list is used when we construct anharmonic
coupling in monomer1, states are grouped according to
(exciton_state_index_list, vmode2)
    // monomer2_quotient_state_list is used when we construct anharmonic
coupling in monomer2, states are grouped according to
(exciton_state_index_list, vmode1)

    // Take monomer1_quotient_state_list for example:
    int exciton_state;  // exciton state. denote different potential energy
surface
    vector<int> vmode;  // vibrational mode of monomer2


    vector <int> full_hamiltonian_state_index_list;  // list of state index in
exciton_state_index_list + monomer vib state wave function that is defined with
(exciton_state_index_list, vmode1, vmode2),
                                                     // which is grouped according
to (exciton_state_index_list, vmode2)

    vector <int> monomer_state_index_list; // sorted list. records index in
monomer1 reduced density matrix basis. (monomer states with vibrational quantum
number vmode1)


    // anharmonic_coupling_info_index records anharmonic coupling for vib
states in monomer 1.
    // list of tuple (i,j,k,l,m):
    // i : vib state in monomer1, j : vib state in monomer1.  i,j monoer state
coupled with each other anharmonically.
    // k: state index in full_matrix, l: state index in full matrix.
    // m : index in monomer Hamiltonian for local anharmonic coupling
    vector<vector<int>> anharmonic_coupling_info_index_list;

    vector<double> anharmonic_coupling_value_list;

    // initialize quotient state. defined with vibrational states in another
monomer and exciton states
    quotient_state(vector<int>  & vmode1, int exciton_state1){
```

```
            exciton_state = exciton_state1;
            vmode = vmode1;
        }
    };
```

In summary, for dimer, when constructing anharmonic coupling within monomer 1, the coupling is between states that have same vibrational states in monomer 2 and the same exciton state.

Therefore, we group all dimer states with the same vibrational states of monomer 2 and the exciton state into monomer1_quotient_state_list.

We group all dimer states with the same vibrational states of monomer 1 and the exciton state into monomer2_quotient_state_list.

We create a strcuture :

```
struct quotient_state
```

which is defined by exciton state and vibrational state of another monomer (monomer2).

Then each quotient state will have list:

- full_hamiltonian_state_index_list
- monomer_state_index_list

which will record index of vibrational states of this monomer (monomer 1) and index of dimer states in full_hamiltonian basis set.

We also have the list attached to each quotient state represent anharmonic coupling in this monomer (monomer 1):

- anharmonic_coupling_info_index_list
- anharmonic_coupling_value_list

anharmonic_coupling_info_index_list : (i,j,k,l,m) records index of vibrational state in monomer 1 (i,j) and index in full matrix (k,l). m is index for anharmonic coupling in monomer hamiltonian.

The function below constructs anharmonic_coupling_info_index_list for each monomer.

```
    // construct MPI version of q_index is easy to do. just let every process
search the index in their local dlist.
    // anharmonic_coupling_info_index is monomer Hamiltonian's element relation
to location in full matrix.
    construct_anharmonic_coupling_info_index_list_MPI();
```

5.3 construct full Hamiltonian off diagonal part:

construct off-diagonal coupling for dimer states in full Hamiltonian basis set.

```
void full_system::compute_full_Hamiltonian_offdiagonal_part_MPI(){
    int i;
    int anharmonic_coupling_num, anharmonic_coupling_num_sum;
    int nonadiabatic_off_num, nonadiabatic_off_num_sum;

    // off-diagonal elements in Hamiltonian, due to anharmonic coupling between
states in the same monomer
    vector<double> anharmonic_coupling_mat;
    vector<int> anharmonic_coupling_irow;
    vector<int> anharmonic_coupling_icol;

compute_monomer_anharmonic_coupling_in_full_matrix_MPI(anharmonic_coupling_mat,
anharmonic_coupling_irow,

anharmonic_coupling_icol);

    // off-diagonal elements in Hamiltonian, due to nonadiabatic coupling
between states in different exciton states (potential energy surface)
    vector<double> nonadiabatic_off_mat;
    vector<int> nonadiabatic_off_irow;
    vector<int> nonadiabatic_off_icol;
    compute_nonadiabatic_offdiagonal_matrix_full_system(nonadiabatic_off_mat,
nonadiabatic_off_irow, nonadiabatic_off_icol);

    //we have to rearrange off-diagonal_matrix in full_system to make sure irow
is in  corresponding process.
    //Also we have to compute offnum, matnum
    combine_offdiagonal_term(anharmonic_coupling_mat, anharmonic_coupling_irow,
anharmonic_coupling_icol,
                             nonadiabatic_off_mat, nonadiabatic_off_irow,
nonadiabatic_off_icol);



}
```

There are two type of coupling between vibrational states in such system:

- anharmonic coupling in the monomer
- nonadiabatic coupling between states in different exciton state

They are computed by different function:

- compute_monomer_anharmonic_coupling_in_full_matrix_MPI
- compute_nonadiabatic_offdiagonal_matrix_full_system

Then anharmonic terms are combined together: combine_offdiagonal_term

FInal results recorded in mat, irow, icol matrix.

Details of computing anharmonic coupling in each monomer and nonadiabatic coupling between states in different exciton states are given below:

5.3.1 anharmonic coupling in full matrix

The anharmonic coupling between dimer states is due to anharmonic coupling within each monomer. It is computed using the quotient_state structure we have defined in section 5.2.

The (anharmonic_coupling_info_index_list) and (anharmonic_coupling_value_list) record coupling strength and info of dimer vibratonal states to facilitate constructing anharmonic couplings in dimer basis set.

5.3.2 compute nonadiabatic coupling between two surfaces

We first compute the table of franck condon factor for vibrational modes in each monomer. Because in our model, the monomer is symmetric, therefore, the results are the same for two monomers in our model.

The Franck Condon factor $\langle m|\alpha;n\rangle$ is decided by Huang-Rhys factor S_i involved in exciton transfer. The displacement factor along each mode α_i is given by $\alpha_i = \sqrt{S_i}$. See Appendix A of PNAS paper: C. Zhang, M. Gruebele, D. E. Logan, and P. G. Wolynes, Surface Crossing and Energy Flow in Many-Dimensional Quantum Systems, Proc. Natl. Acad. Sci. U.S.A. 120, e2221690120 (2023)

for detailed information.

Coupling between state in different vibrational state

ground state for shifted harmonic oscillator :

$$|\alpha,0\rangle = D(\alpha)|0\rangle = e^{-\frac{1}{2}\alpha^2} e^{\alpha b^\dagger} e^{-\alpha b}|0\rangle$$
$$= e^{-\frac{1}{2}\alpha^2} e^{\alpha b^\dagger}|0\rangle$$

Here to compute coupling between vibrational state in different electronic state we have to consider overlap of two vibrational state :

$$t_{m,\alpha n} = t\langle m \mid \alpha n\rangle =$$

$$\langle m \left|\alpha; n\right\rangle = \exp\left(-\alpha^2/2\right) \langle m \left|\exp\left(\alpha \hat{b}^\dagger\right) \exp(-\alpha \hat{b})\right| n\rangle\Big)$$

Above derivation use Baker–Campbell–Hausdorff formula :

Baker–Campbell–Hausdorff formula - Wikiwand

More specifically : $e^{\alpha_\downarrow b} e^{\alpha_\uparrow b^\dagger} = e^{\alpha_\downarrow b + \alpha_\uparrow b^\dagger + \frac{1}{2}\alpha_\downarrow \alpha_\uparrow}$

$e^{\alpha_\uparrow b^\dagger} e^{\alpha_\downarrow b} = e^{\alpha_\downarrow b + \alpha_\uparrow b^\dagger - \frac{1}{2}\alpha_\downarrow \alpha_\uparrow}$

The franck condon factor can be computed as following:

$$\left\langle m \left|\exp\left(\alpha \hat{b}^\dagger\right) \exp(-\alpha \hat{b})\right| n\right\rangle =$$
$$\left\langle m \left|\left(1 + \alpha \hat{b}^\dagger + \frac{(\alpha)^2 \left(\hat{b}^\dagger\right)}{2!} + \cdots\right)(1 - \alpha \hat{b} + \cdots)\right| n\right\rangle$$
$$= \sum_k \frac{(\alpha)^{m-k}}{(m-k)!} \left[\langle m| \left(\hat{b}^\dagger\right)^{m-k}\right] \times \frac{(-\alpha)^{n-k}}{(n-k)!} \left(\hat{b}^{n-k} |n\rangle\right)$$
$$= \sum_k \frac{(\alpha)^{m-k}}{(m-k)!} \left[\langle k| \frac{\sqrt{m!}}{\sqrt{k!}}\right] \times \frac{(-\alpha)^{n-k}}{(n-k)!} \left(\sqrt{\frac{n!}{k!}} |k\rangle\right) =$$
$$\sum_{k=0}^{\min(m,n)} (\alpha)^{m+n-2k} \frac{(-1)^{n-k}}{(m-k)!(n-k)!} \frac{\sqrt{m!n!}}{k!}$$

In the code, this is programmed as :

```
double franck_condon_factor = 0;
    // prefactor = e^{-alpha^2/2} sqrt(n! m!) * alpha^{n+m}
    // std::tgamma(n+1) = n!
    double prefactor = exp(- pow(alpha,2)/2) * pow(alpha, n + m) *
sqrt(std::tgamma(n+1)  * std::tgamma(m+1) );

    if (alpha!=0){
        for (l=0;l<=nm_min;l++){
            // sum: 1/(l! * (n-l)! * (m-l)!) * (-1)^{n-l} * alpha^{-2l}
            franck_condon_factor = franck_condon_factor + 1/( std::tgamma(l+1)
* std::tgamma(n-l+1) * std::tgamma(m-l+1) ) * pow(-1, n-l) * pow(alpha, -2*l);
        }

        franck_condon_factor = franck_condon_factor * prefactor;
    }
```

Evolve_full_sysem_MPI()

Evolve Schrodinger equation

$$i\hbar\frac{d\psi}{dt} = H\psi$$

1. d.prepare_evolution():

Matrix Vector multiple multiplication with process

① N

N×1 = N × N×1

Vector

② Now you have N process

N/n
N/n
N/n

= N/n
N/n
N/n

× N×1

③ 1

N/n

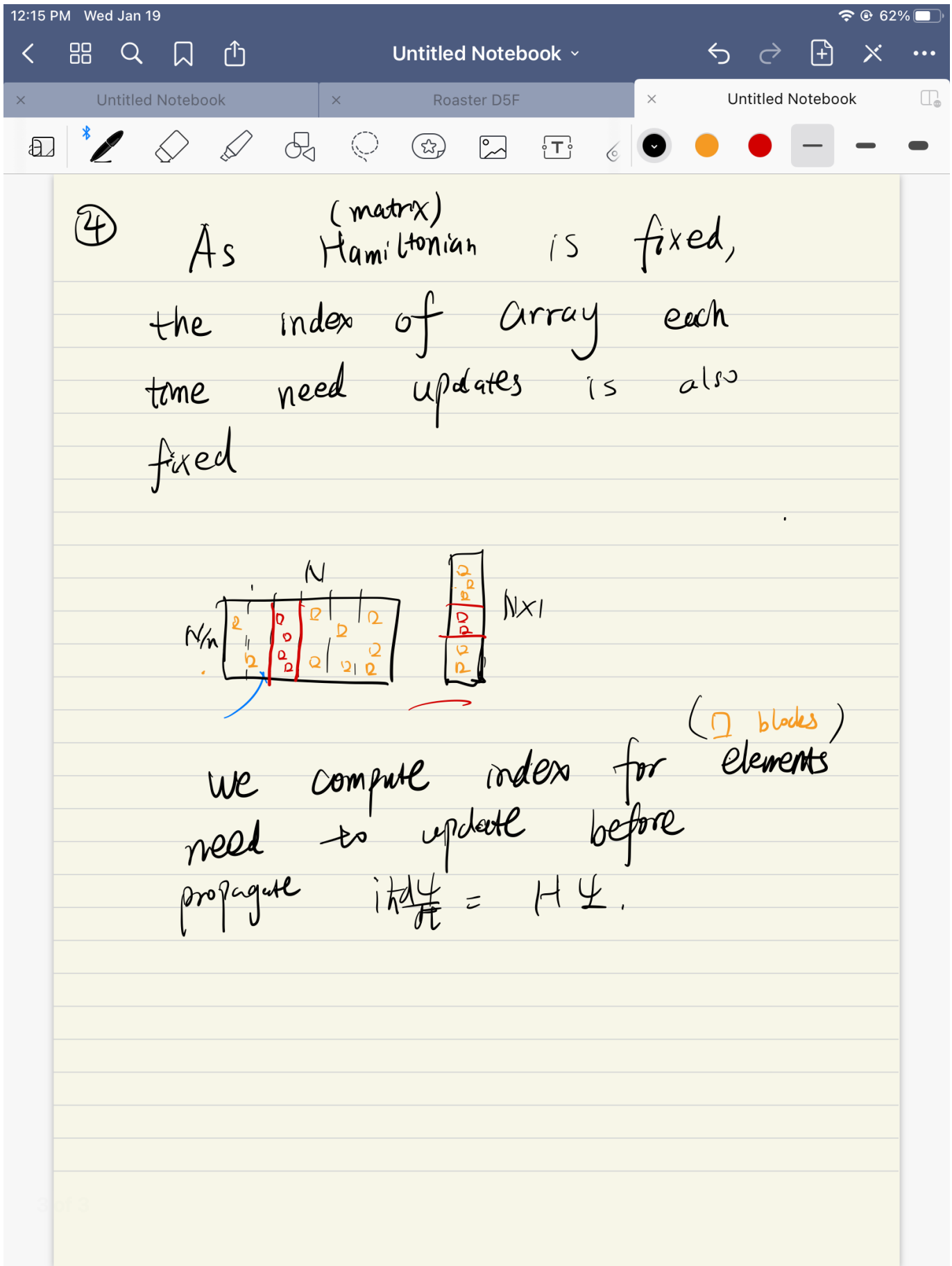
= N/n × N

N×1

Sparse matrix

These orange array elements need updates.

You have to update this array each time you have changed it



We use MPI to speed up matrix array multiplication.

We divide wave function ψ into different part and store in different processes.

$$d\psi = \psi(t+dt) - \psi = dt \times (H\psi)$$

We know Hamiltonian H as a matrix may have component : $H(m,n) \neq 0$, and element m, n belong to different process

Hamiltonian matrix H is shared in different processes.

For process p_n, it has array of ψ from index $n_1 \sim n_N$, if we have $H(n_j, m)$ where $n_j \in [n_1, n_N]$ but $m \notin [n_1, n_N]$.

Say we have in total M element that $H(n_j, m) \neq 0$, then we prolong the array that store ψ from size $n_N - n_1$ to $n_N - n_1 + M$. with local array of extra size M.

Then each time before we compute $H\psi$, we send data $\psi[m]$ from other processes and store in extra M elements in local array. Then when compute $H(n_j, m)$ we multiply H by element in local array.

```
    to_recv_buffer_len[0] =
construct_receive_buffer_index(remoteVecCount[0],remoteVecPtr[0],
            remoteVecIndex[0],0);  // construct buffer to receive.
    to_send_buffer_len[0]=
construct_send_buffer_index(remoteVecCount[0],remoteVecPtr[0],remoteVecIndex[0]
,

tosendVecCount[0],tosendVecPtr[0], tosendVecIndex[0]);
```

This code analyze Hamiltonian H and find all the index for m of vector ψ that should be trasferred between different process before matrix -vector multiplication.

1.1 construct_receive_buffer_index()

```
int full_system::construct_recvbuffer_index(){
    int i,j,k;
    int total_recv_count=0;
    int local_begin= matsize_offset_each_process[my_id];
    int local_end;
    if(my_id!=num_proc-1){
        local_end = matsize_offset_each_process[my_id+1];
    }
    else{
        local_end= total_matsize;
    }

    // --------- Allocate space for vector to receive -----------------------
    remoteVecCount = new int [num_proc];
    remoteVecPtr= new int [num_proc];
    remoteVecIndex = new int [offnum];
    for(i=0;i<num_proc;i++){
        remoteVecCount[i] = 0;
    }
    // ------------------------------------------------------------------


    vector<int> icol_copy = icol;
    sort(icol_copy.begin(),icol_copy.end());
    j=0;
    int prev_col=-1;
    int col_index;
    int col_pc_id;
    for(i=0;i<matnum;i++){
        col_index= icol_copy[i];
        if(col_index>prev_col and (col_index<local_begin or col_index >=
local_end) ){
            // find process id this icol belong to
            for(k=1;k<num_proc;k++){
                if(col_index < matsize_offset_each_process[k]){
                    break;
                }
            }
            col_pc_id= k-1;
            remoteVecCount[col_pc_id]++;
            remoteVecIndex[j]= col_index;
            j++;
        }
        prev_col= col_index;
    }

    remoteVecPtr[0]=0;
```

```
    for(i=1;i<num_proc;i++){
        remoteVecPtr[i]= remoteVecPtr[i-1] + remoteVecCount[i-1];
    }
    total_recv_count=0;
    for(i=0;i<num_proc;i++){
        total_recv_count = total_recv_count + remoteVecCount[i];
    }
    return total_recv_count;
}
```

Here icol_copy is column index for nonzero element in Hamiltonian reside in given process.

1 . We record index in remoteVecIndex_element

2. We record number of array index need to transfer in remoteVecCount_element

3. remoteVecPtr_element : used in MPI. Tell MPI the displacement of vectors.

   We use MPI_all_to_allv

1.2 constuct_send_buffer_index()

Prepare index information to send buffer to other process.

use MPI_Alltoall and MPI_Alltoallv

1.3 Allocate space for element received from other process.

```
    // we add extra space at the end of wave function array x,y.
    // Which will store the wave function elements received from other process
through MPI.
    x.resize(matsize + to_recv_buffer_len);
    y.resize(matsize+ to_recv_buffer_len);

    recv_x = new double [to_recv_buffer_len];
    recv_y= new double [to_recv_buffer_len];
    send_x = new double[to_send_buffer_len];
    send_y= new double [to_send_buffer_len];
```

This code extend x (real part of wave function) , y (imaginary part of wave function) array to reside extra M elements.

1.4 Decide row, index array for matrix multiplication

```
    // construct local_dirow, local_dicol
    local_dirow[0].reserve(monomer_matnum[0]);
    local_dicol[0].reserve(monomer_matnum[0]);
    for(i=0;i<monomer_matnum[0];i++){
        local_dirow[0].push_back(monomer_irow[0][i] - my_id * vsize);  // set
local index for row index
        col_index_to_search= monomer_icol[0][i];
        search_Ind=(int *)
bsearch(&col_index_to_search,remoteVecIndex[0],to_recv_buffer_len[0],sizeof(int
),compar);
        if(search_Ind!=NULL){
            // this column index is not in local matrix, and we should get it
from other process (remoteVec)
            local_dicol[0].push_back(monomer_matsize[0] + (search_Ind-
remoteVecIndex[0]) );
        }
        else{ // this column index is in local matrix.
            local_dicol[0].push_back (monomer_icol[0][i] - my_id * vsize );
        }
    }
```

As in Matrix Multiplication, we may use array element received from other process, we have to construct separate irow, icol for matrix vector multiplication.

Here remoteVecIndex is index in rang $[n_1, n_N]$ for process p_n .

local_dirow , local_dicol is used in matrix vector multiplication as row and column index for Hamiltonian H.

$$d\psi[row] = H[row, col] \times \psi[col]$$

local_dirow is always element in range $[n_1, n_N]$ for process p_n ,

local_dicol is in $[n_1, n_N]$ if $H[n, m]$ has $m \in [n_1, n_N]$, otherwise local_dicol is in range $[n_N, n_N + M]$.

2 . Evolve wave function ψ SUR_one_step():

```
void monomer::SUR_onestep_MPI(){
    int m, i;
    int irow,icol;
    int nearby_state_list_size = sampling_state_index.size();
    // do simulation starting from different state

    // update imaginary part of wave function calculated in different process.
    update_dy(nearby_state_list_size);
    // SUR algorithm
    for(i=0;i<monomer_matnum[0];i++){
        // make sure when we compute off-diagonal matrix, we record both
symmetric and asymmetric part
        irow = local_dirow[0][i];
        icol = local_dicol[0][i]; // compute to point to colindex in
        for(m=0;m<nearby_state_list_size;m++) {
            xd[m][irow] = xd[m][irow] + monomer_mat[0][i] * yd[m][icol] * cf;
        }
    }

    // update real part of wave function calculated in different process.
    update_dx(nearby_state_list_size);
    // SUR algorithm.
    for(i=0;i<monomer_matnum[0];i++){
        irow= local_dirow[0][i];
        icol= local_dicol[0][i];
        for(m=0;m<nearby_state_list_size;m++){
            yd[m][irow] = yd[m][irow] - monomer_mat[0][i] * xd[m][icol] * cf;
        }
    }

}
```

See before we do $H \times \psi$ we have : **update_dx()** , **update_dy()** .

These functions are for upodating elements of $\psi(m)$ for $H(n,m) \neq 0$ if m is not in local array of wave function ψ .