## 1 Source code availability and license

The source code can be downloaded at this link and it is covered by the ISC license:

## 2 Memory management

To handle all the matrix operations a stack-oriented approach has been taken. The dimension of the matrices is kept fixed and chosen at the start of any simulation.

The function initializing the matrix stack is InitMemory( matrix_dim, stack_dim) and it must be called once at the beginning of the program. Example:

```
int main()
{
        InitMemory( 4, 15);

        ... ALL OTHER CALLS ...

        return 0;
}
```

## 3 Notation

Comments follow the notation

```
INPUT_N ... INPUT_1 -- OUTPUT_M ... OUTPUT_1
```

adapted from FORTH. The matrix INPUT_1 is the first on top of the stack.

Complex numbers have been renamed through the following typedef command

```
typedef lapack_complex_double double_complex;
```

The type lapack_complex_double may be almost equal to double complex type understood by GCC. The complex type is not handled in a really natural way by GCC. Complex numbers in C have been introduced, I think, in the C99 standard. At the present time most of the C compilers, to my knowledge. are not completely compliant with such a reference standard. For these reasons hypothetical future versions of the library should avoid the inner complex data type.

## 4 Legend of common symbols

| | | |
|---|---|---|
| double temperature | $T$ | Temperature of the bath |
| double tau | $\tau$ | Hamiltonian control parameter |
| double tau_i | $\tau_i$ | initial Hamiltonian control parameter |
| double tau_f | $\tau_f$ | final Hamiltonian control parameter |
| double dt | $dt$ | Integration time step |
| double (∗omega_int)(int) | $[\omega_k]$ | Characteristic $\omega_k$ frequencies. Address to a function sending an integer $k$ to a real $\omega_k$ |
| double time_duration | $\Delta t$ | Duration of an evolution |
| double (∗spectral_function )(double) | $[\mathcal{J}(\omega)]$ | Spectral function. Address to a function sending a double $\omega$ to the value of the spectral function $\mathcal{J}(\omega)$ |
| double (∗hm)( double tau, int i , int j ) | $[H(\tau)]$ | Hamiltonian matrix. Address to a function generating the entries of the Hamiltonian matrix with a certain $\tau$ |
| double ∗work | $W$ | Work done on the system during a transformation. Address at which the value will be added. |
| double ∗heat | $Q$ | Heat transferred to the system during a transformation. Address at which the value will be added. |
| G or AAcross | $G$ | Correlation matrix on its natural basis |
| BBcross | $F$ | Correlation matrix on the Bogoliubov-Valatin diagonalizing basis |

## 5 Basic stack operations

| | | |
|---|---|---|
| void HLOAD( double_complex (*hm)( double tau, int i, int j ), double tau); | $( \to H(\tau) )$ | Calculates the Hamiltonian matrix with $\tau$ (tau) parameter and pushes it on top of the stack |
| void FLOAD( double_complex (*mm)(int i,int j)); | $( \to A )$ | Calculates the matrix related to the function pointed by mm and loads it on top of the stack |
| void ARG_FLOAD( double_complex (*mm)( int i, int j , void *args), void *args); | $( \to A )$ | Like FLOAD but passing the address args to the function pointed by mm |
| void RANDOM_HERMITIAN(); | $( \to A )$ | Loads a random Hermitian matrix on top of the stack |
| void LOAD(void *mat); | $( \to A )$ | Loads the matrix of address mat on top of the stack |
| void TLOAD(void *mat); | $( \to A^T )$ | Loads the matrix of address mat transposed on top of the stack |
| void DUP(); | $( A \to A\ A )$ | Duplicates the matrix on top of the stack |
| void OVER(); | $( A\ B \to A\ B\ A )$ | Duplicates the matrix on the second position on top of the stack |
| void PRINT_MATRIX(); | $( A \to A )$ | Prints the matrix on top of the stack ( leaving the stack unchanged) |
| void DROP(); | $( A \to )$ | Drops the first matrix on the stack |
| void ROT(); | $( A\ B\ C \to B\ C\ A )$ | Left rotation of the first three matrices on the stack |
| void MROT(); | $( A\ B\ C \to C\ A\ B )$ | Right rotation of the first three matrices on the stack |
| void SWAP(); | $( A\ B \to B\ A )$ | Swaps the first two matrices on the stack |
| void NIP(); | $( A\ B \to B )$ | Drops the second matrix on the stack |

## 6 Numerical stack operations

| | | |
|---|---|---|
| void MINUS_PART(); | $( A \to A_{\ominus} )$ | Substitutes A with its minus component |
| void PLUS_PART(); | $( A \to A_{\oplus} )$ | Substitutes A with its plus component |
| double NORM(); | $( A \to A )$ | Calculates the norm of A leaving stack unchanged. It returns the result on the call stack. |
| void PRINT_NORM(); | $( A \to A )$ | Calculates the norm of A leaving stack unchanged. |
| void CONJ(); | $( A \to A^{*} )$ | Substitutes A with its complex conjugate |
| void SUM(); | $( A\ B \to A + B )$ | Matrix summation |
| void DMUL(); | $( A\ B \to A \odot B) )$ | Term by term multiplication |
| void SUB(); | $( A\ B \to A - B )$ | Matrix subtraction |
| void MUL(); | $( A\ B \to AB )$ | Matrix inner multiplication |
| void SCALE( double_complex alpha); | $( A \to \alpha A )$ | Matrix multiplication by a complex factor alpha |
| void TRANS(); | $( A \to A^{T} )$ | Substitutes A with its transposed |
| void CROSS(); | $( A \to A^{\dagger} )$ | Substitutes A with its transposed complex conjugate |
| void COMMUTATOR(double_complex alpha); | $( A\ B \to \alpha\, [A, B] )$ | Calculates the commutator multiplied by a complex factor alpha |
| void ANTI_COMMUTATOR(double_complex alpha); | $( A\ B \to \alpha\, \{A, B\} )$ | Calculates the anticommutator multiplied by a complex factor alpha |
| void UcrossAU(); | $( A\ U \to U^{\dagger} A U )$ | Multiplication of U transposed complex conjugate with A followed by multiplication with U. |
| void UAUcross(); | $( U\ A \to U A U^{\dagger} )$ | Multiplication of U with A followed by multiplication with U transposed complex conjugate. |
| void DIAG(double ∗w); | $( A \to U )$ | Diagonalizes A with gheev, returns eigenvectors as a matrix on top of the stack, writes the corresponding eigenvalues in the array of address w. The eigenvalues array must have a MatrixDim capacity. |
| void EIGVECTS(); | $( A \to U )$ | Diagonalizes A with gheev, returns eigenvectors as a matrix on top of the stack, ignores eigenvalues. |
| void EIGVALS(double ∗w); | $( A \to )$ | Diagonalizes A with gheev, writes the eigenvalues in the array of address w, ignores eigenvectors. The eigenvalues array |

## 7 Bogoliubov-Valatin related operations

| | | |
|---|---|---|
| void BOG_DIAG(double *w); | $( H \to U )$ | Performs fermionic Bogoliubov-Valatin diagonalization, returns eigenvectors as a matrix on top of the stack, writes the corresponding eigenvalues in the array of address w. The eigenvalues array must have a MatrixDim capacity. |
| void LOAD_BOG_U( void *hm, double tau, double *w); | $( \to U )$ | It loads the Hamiltonian matrix with tau control parameter and calls BOG_DIAG The result is stored in a cache memory for later use. If the tau parameter is changed the diagonalization is performed again and the cache is updated |
| void make_ground_state(<br>double tau,<br>double_complex (*hm)(double tau,int i,int j )); | $( \to G_{\text{ground}} )$ | Produces the ground state correlation matrix through the Bogoliubov-Valatin transformation and loads it on top of the stack. |
| void make_thermal_state(<br>double tau,<br>double temperature,<br>double_complex (*hm)(double tau,int i,int j )); | $( \to G_{\text{thermal}}(T) )$ | Produces the ground state correlation matrix through the Bogoliubov-Valatin transformation and loads it on top of the stack. |
| void make_diag_thermal_state(<br>double tau,<br>double temp,<br>double_complex (*hm)(double tau,int i,int j )); | $( \to F_{\text{thermal}}(T) )$ | Produces a thermal correlation matrix through the Bogoliubov-Valatin transformation and loads it on top of the stack. |

## 8 Low level evolution routines

```
double evolve_matrix(
            double_complex (*hm)(double tau,int i,int j ),
            void (*f)(double tau,double dt),
            double tau_i ,
            double tau_f,
            double time_duration,
            double dt);
```

( $G(0) \to G(\Delta t)$ )

Makes the matrix G(0), on top of stack, evolve. The integration method is a 4th order Runge-Kutta. f is the function generating the increment ( $G(t) \to dG(t)$).
It can be freely chosen in such a way to make the integrator to simulate free evolution, Markovian thermalizations, etc.
hm is the address for the Hamiltonian generator function and it not used for the evolution itself, but for the calculation of the work received by the working substance.

```
void dissipative_F (
                     void *hm,
                     double tau,
                     double dt,
                     double (*omega_int)(int p),
                     double (*bath_sf)(double omega),
                     double beta,
                     double mu);
```

( $G(t) \to dG(t)$ )

Calculates the part of the increment of G related to the system being in contact with an heat-bath. To be used in the evolve_matrix function some variables must be kept fixed. As an example, this function is called by in_contact_evolve.

## 9 Middle level evolution routines

```
double free_evolve (
                     double_complex (*hm)(double tau,int i,int j ),
                     double tau_i ,
                     double tau_f,
                     double time_duration,
                     double dt);
```

( $G(0) \to G(\Delta T)$ )

Evolution without dissipation, returns work done on the system on call stack, final matrix on top of the stack.

```
double in_contact_evolve (
                     double_complex (*hm)(double tau,int i,int j ),
                     double tau_i ,
                     double tau_f,
                     double time_duration,
                     double dt,
                     double temperature,
                     double (*omega_int)(int k),
                     double (*spectral_function )(double omega));
```

( $G(0) \to G(\Delta T)$ )

Evolution with dissipation, returns the work done on the system on call stack, final matrix on top of the stack.

## 10 High level evolution operations

```
void thermalize(
                    double temperature,
                    double tau_fixed, double time_duration, double dt,
                    double (*omega_int)(int k),
                    double (*spectral_function)(double),
                    double_complex (*hm)(double tau,int i,int j),
                    double *work, double *heat);
```

( $G(0) \to G(\Delta t)$ ) Non-ideal thermalization

```
void isothermal(
                    double temperature,
                    double tau_i,
                    double tau_f,
                    double iso_time,
                    double dt,
                    double (*omega_int)(int k),
                    double (*spectral_function)(double),
                    double_complex (*hm)(double tau,int i,int j),
                    double *work, double *heat);
```

( $G(0) \to G(\Delta T)$ ) Imperfect isothermal transformation

```
void adiabatic(
                   double tau_i,
                   double tau_f,
                   double time_duration,
                   double dt,
                   double_complex (*hm)(double tau,int i,int j),
                   double *work, double *heat);
```

( $G(0) \to G(\Delta T)$ ) Adiabatic transformation

```
void double_isothermal(
                          double temperature_1,
                          double temperature_2,
                          double tau_i,
                          double tau_f,
                          double time_duration,
                          double dt,
                          double (*omega_int_1)(int k),
                          double (*spectral_function_1)(double),
                          double (*omega_int_2)(int k),
                          double (*spectral_function_2)(double),
                          double_complex (*hm)(double tau,int i,int j),
                          double *work, double *heat);
```

( $G(0) \to G(\Delta T)$ ) Imperfect isothermal transformation while in contact with two different heat-baths at the same time