QcMatrix

Version 0.1.0

Bin Gao

March 14, 2015

Centre for Theoretical and Computational Chemistry (CTCC)
Department of Chemistry
University of Tromsø — The Arctic University of Norway
N–9037, Tromsø, Norway

Contents

Chapter 1

What is QcMatrix?

QcMatrix is an "abstract" matrix library written in C language (with C++ and Fortran interface), and provides a "speical" square block complex matrix (as shown in Fig. 1.1 and see the discussion below) and corresponding functions.

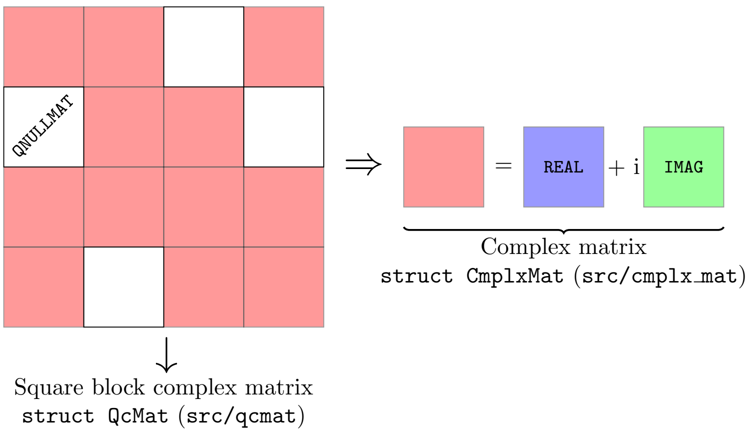

Figure 1.1: Illustration of the square block complex matrix implemented in QcMatrix — 4×4 blocks, with red blocks as the non-zero complex matrices. Each block is square matrix with the same dimension.

In general, a square block matrix $\mathbf{A}$ satisfies[1]:

1. $\mathbf{A}$ is a square matrix,
2. the blocks form a square matrix,
3. the diagonal blocks are also square matrices;

for instance, a $N \times N$ square block matrix $\mathbf{A}$ takes the following structure

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \cdots & \mathbf{A}_{1N} \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \cdots & \mathbf{A}_{2N} \\ \cdots & \cdots & \cdots & \cdots \\ \mathbf{A}_{N1} & \mathbf{A}_{N2} & \cdots & \mathbf{A}_{NN} \end{bmatrix}, \tag{1.1}$$

where $\mathbf{A}_{II}$ $(1 \leq I \leq N)$ is also square matrix ($\mathbf{A}_{I \neq J}$ may not be square matrix).

The square block complex matrix implemented in QcMatrix is a special square block matrix that we also require **all the blocks have the same dimension**[1]. As shown in Fig. 1.1, in the QcMatrix

[1]So all the blocks are square matrix with the same dimension.

library, the square block complex matrix and its operations at the block level are taken care by the codes in `src/qcmat` directory, while the complex matrix of each non-zero block (zero blocks denoted as `QNULLMAT`, will not participate in the matrix operations) and its operations are carried out by the codes in `src/cmplx_mat` directory.

As an "abstract" matrix library, QcMatrix should be in general built on top of external matrix library, which is written either in C, C++ or Fortran language. As illustrated in Fig. 1.2, QcMatrix can be viewed as an adapter between external matrix libraries and application libraries (which depends on the matrix and matrix operations) written in different languages — C, C++ or Fortran. The conversion between different languages is taken care by QcMatrix, so that the application libraries can in principle, without any modification, use different external matrix libraries through the QcMatrix.

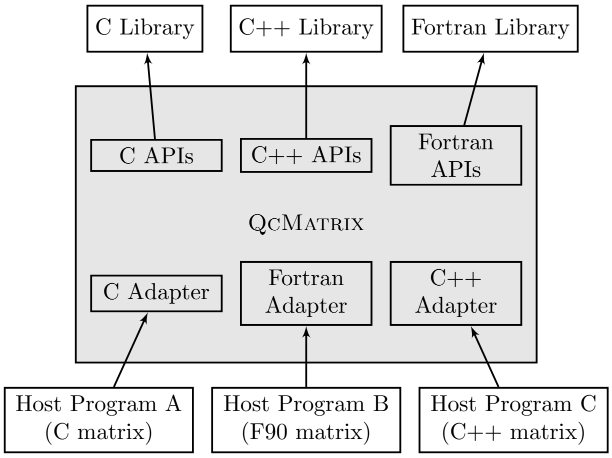

Figure 1.2: QcMatrix as an adapter between different languages.

QcMatrix can also work as an adapter between different matrices. As shown in Fig. 1.3, whatever implemented in the external library — real matrix, complex matrix, square block real matrix or square block complex matrix — can all be "translated" by QcMatrix, which then provides the square block complex matrix to different application libraries.

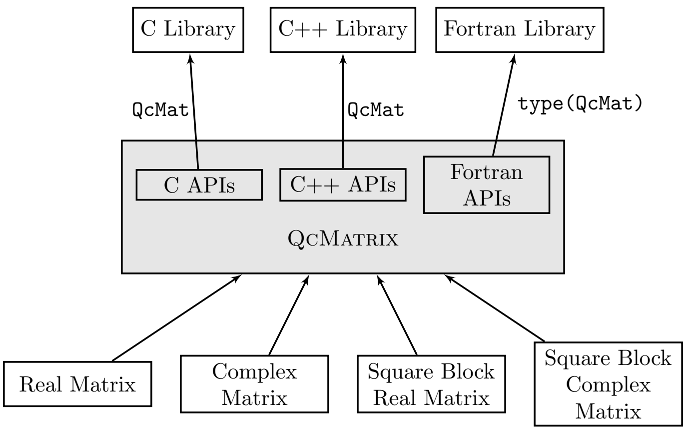

Figure 1.3: QcMatrix as an adapter between different matrices.

The application programming interface (API) of QcMatrix for different languages is slight different. For instance, including QcMatrix in a code and declaration of a matrix are a bit different in different languages, as illustrated in Table 1.1. Details of the QcMatrix APIs are given in Chapter 3.

Table 1.1: Using QcMatrix in different languages.

Language	Including QcMatrix	Declaration of a matrix
C	`#include "qcmatrix.h"`	`QcMat A;`
C++		
Fortran	`use qcmatrix_f`	`type(QcMat) A`

To use QcMatrix, the external library should also implement the matrix type and functions/subroutines required by QcMatrix. Detailed information can be found in Chapter 4, we give in Table 1.2 the required header files/modules and implemented matrix type in the external library.

Table 1.2: Required header files/modules and implemented matrix in the external library.

Language	Header file/Module	Implemented matrix
C	`LANG_C_HEADER`	`struct LANG_C_MATRIX`
C++		
Fortran	`LANG_F_MODULE`	`type LANG_F_MATRIX`

The left chapters in this manual are: Chapter 2 describes the installation and test suite of QcMatrix, Chapters 5–8 are the applications built on top of the QcMatrix library, respectively, the self-consistent field solvers (Chapter 5), linear response solvers (Chapter 6), simulations of X-ray spectroscopies (Chapter 7) and molecular electronics (Chapter 8). Chapter 9 contains some advanced topics of QcMatrix, which may be only relevant for QcMatrix developers. Finally, Chapters 10 and 11 respectively describe the files and directories of QcMatrix, and some limitations of QcMatrix.

If there is any question regarding the use of QcMatrix, please contact the authors in the file `AUTHORS`. If you have used QcMatrix and found it is useful, please consider to cite QcMatrix as

```
@misc{QcMatrix,
  author = {Bin Gao},
  title = {{QcMatrix Version 0.1.0}},
  year = {2015},
  note = {https://gitlab.com/bingao/qcmatrix}
}
```

Chapter 2

Installation

Before installing QcMatrix, you need to make sure the following programs are installed on your computer:

1. Git,
2. CMake (≥ 2.8),
3. HDF 5 (≥ 1.8) if matrix I/O is enabled (if HDF 5 is not available, ordinary binary format file will be used which is not portable),
4. C, C++ (if C++ adapter and APIs required) and/or Fortran (if Fortran adapter and APIs needed) compilers,
5. BLAS and LAPACK libraries for test suite and/or QcMatrix internal real matrix library.

QcMatrix can be got as:

```
git clone git@gitlab.com:bingao/qcmatrix.git
```

Afterwards, you could start to compile QcMatrix.

2.1 CMake

For the time being, only CMake could be used to compile QcMatrix. In general, QcMatrix should be compiled together with the external libraries or host programs. See for example the Dalton program[1].

If there is no external library, you could still compile QcMatrix by (i) using its own internal real matrix library (in `src/real_mat`, may not be efficient), or (ii) using the simple C++ or Fortran matrix libraries in `tests/cxx/adapter` and `tests/f90/adapter`[2]. Let us assume that you want to compile the library in directory `build`, you could invoke the following commands:

```
mkdir build
cd build
ccmake ..
make
```

During the step `ccmake`, you need to set some parameters appropriately for the compilation. For instance, if you enable `QCMATRIX_TEST_EXECUTABLE`, some executables for the test suite will be built and can run after compilation. So that you are able to check if QcMatrix has been successfully compiled. A detailed list of the parameters controlling the compilation is given in Table 2.1.

[1] Currently, QcMatrix is implemented in the `qcmatrix` branch of Dalton program. Interested users could check these directories in Dalton: `DALTON/qcmatrix` and `LSDALTON/qcmatrix`.

[2] In that case, you first need to compile the C++ or Fortran matrix libraries. Please follow the `README` file therein.

Table 2.1: CMake parameters controlling the compilation of QcMatrix.

Parameter	Description	Default
QCMATRIX_3M_METHOD	Enable 3M method for complex matrix-matrix multiplication.	ON
QCMATRIX_64BIT_INTEGER	Use 64 bit integer.	OFF
QCMATRIX_AUTO_ERROR_EXIT[3]	Enable automatic exit on error.	OFF
QCMATRIX_BLAS_64BIT	Use 64 bit BLAS and LAPACK libraries.	OFF
QCMATRIX_ENABLE_VIEW	Enable matrix I/O.	OFF
QCMATRIX_ENABLE_HDF5	Enable the use of HDF5 library for matrix I/O.	ON
QCMATRIX_SINGLE_PRECISION	Use single precision for real numbers.	OFF
QCMATRIX_STORAGE_MODE	Enable different matrix storage modes.	OFF
QCMATRIX_STRASSEN_METHOD	Strassen's method for the square block complex matrix-matrix multiplication.	ON
LIB_QCMATRIX_NAME	Sets the name of the QcMatrix library.	qcmatrix
QCMATRIX_ROW_MAJOR[4]	Row major order for matrix elements.	OFF
QCMATRIX_ZERO_BASED[5]	Zero-based numbering.	ON
Adapter		
QCMATRIX_BUILD_ADAPTER	Build the adapter for external matrix library.	OFF
QCMATRIX_ADAPTER_TYPE	Choose the type of the external library, valid entries are C;CXX;F90;F03.	C
EXTERNAL_BLOCK_MATRIX	Square block matrix implemented in the external library.	OFF
EXTERNAL_COMPLEX_MATRIX	Complex matrix implemented in the external library.	OFF
QCMATRIX_EXTERNAL_LIBRARIES	Sets the external libraries for QcMatrix (like -lxxxx).	None
QCMATRIX_EXTERNAL_PATH	Sets the path of external libraries for QcMatrix.	None
Adapter (C)		
LANG_C_HEADER	Name of header file of the external C library.	lib_matrix.h
LANG_C_MATRIX	Name of external C matrix struct.	matrix_t
Adapter (F90)		
LANG_F_MATRIX	Name of external Fortran 90 matrix type.	matrix_t
LANG_F_MODULE	Name of external Fortran 90 matrix module.	lib_matrix
SIZEOF_F_TYPE_P	Size (in bytes) of Fortran 90 derived types with a single pointer member.	12
Adapter (F03)		

Continued on next page

[3] If QCMATRIX_AUTO_ERROR_EXIT=ON, QcMatrix will automatically exit on error, and users can not check the return error information.

[4] The column major order is recommended if QcMatrix internal real matrix library is used (i.e., if BUILD_ADAPTER=OFF).

[5] QCMATRIX_SINGLE_PRECISION, QCMATRIX_ROW_MAJOR and QCMATRIX_ZERO_BASED should be consistent with external matrix library if BUILD_ADAPTER=ON.

Table 2.1 – continued from previous page

Parameter	Description	Default
LANG_F_MATRIX	Name of external Fortran 2003 matrix type.	matrix_t
LANG_F_MODULE	Name of external Fortran 2003 matrix module.	lib_matrix
API		
QCMATRIX_CXX_API	Build C++ API.	OFF
QCMATRIX_Fortran_API	Build Fortran API, options are: None;F90;F03.	None
Matrix I/O		
HDF5_DIR	The directory containing a CMake configuration file for HDF5.	HDF5_DIR-NOTFOUND
HDF5_ROOT	Provide a hind about where to find the HDF5 installation.	None
HDF5_USE_STATIC_LIBRARIES	Enable the static link for HDF5.	ON
Test suite		
QCMATRIX_TEST_3M_METHOD	Build the test of efficiency of 3M method.	OFF
QCMATRIX_TEST_EXECUTABLE	Build the test suite excutables.	ON

If no error happened, you will have a library named `lib${LIB_QCMATRIX_NAME}.a`, where `LIB_QCMATRIX_NAME` is set during CMake procedure.

2.2 Test Suite

All the tests are in the directory `tests`, and will also be compiled. As aforementioned, these tests will be built as executables if `QCMATRIX_TEST_EXECUTABLE` is enabled (more explicitly, `${LIB_QCMATRIX_NAME}_c_test` will always be built, `${LIB_QCMATRIX_NAME}_cxx_test` and `${LIB_QCMATRIX_NAME}_f_test` will be built only if C++ and Fortran APIs are built). Otherwise, these tests will be compiled and into the library `lib${LIB_QCMATRIX_NAME}.a` so that they could be invoked from the host program as, `ierr = test_c_QcMatrix()` (C and C++) or `call test_f_QcMatrix(io_log)` (Fortran).

In most tests, we compare the results from QcMatrix and those from the BLAS and/or LAPACK routines by using the whole matrix as an array.

Chapter 3

QcMatrix API Reference

As mentioned in Chapter 1, the API of QcMatrix is only slight different for different languages. Including QcMatrix in a code and declaration of a matrix has been given in Table 1.1. In this chapter, we will describe the QcMatrix API references in detail.

First, there are some parameters defined in QcMatrix that can be used by users, as given in Table 3.1.

Table 3.1: Public parameters provided by QcMatrix.

Parameter	Type	Description
QSUCCESS	QErrorCode	Function returns no error.
QFAILURE	QErrorCode	Function returns with error.
QTRUE	QBool	Boolean type, true.
QFALSE	QBool	Boolean type, false.
QZEROTHRSH	QReal	Threshold for nearly negligible number.
QSYMMAT	QcSymType	Symmetric (Hermitian) matrix.
QANTISYMMAT	QcSymType	Anti-symmetric (anti-Hermitian) matrix.
QNONSYMMAT	QcSymType	Non-symmetric (non-Hermitian) matrix.
QNULLMAT	QcDataType	Matrix that neither real nor imaginary part is assembled.
QREALMAT	QcDataType	Real matrix.
QIMAGMAT	QcDataType	Imaginary matrix.
QCMPLXMAT	QcDataType	Complex matrix.
UNKNOWN_STORAGE_MODE[1]	QcStorageMode	Unknown storage mode (mostly for error handling), while specific storage modes should be defined and implemented in the external library.
COPY_PATTERN_ONLY	QcDuplicateOption	Duplicate option, copies the pattern of a matrix only (previous numerical values may be removed depending on the external library).
COPY_PATTERN_AND_VALUE	QcDuplicateOption	Duplicate option, copies an entire matrix including its numerical values.
MAT_NO_OPERATION	QcMatOperation	No matrix operation performed.
MAT_TRANSPOSE	QcMatOperation	Transpose.
MAT_HERM_TRANSPOSE	QcMatOperation	Hermitian transpose.
MAT_COMPLEX_CONJUGATE	QcMatOperation	Complex conjugate.
BINARY_VIEW[2]	QcViewOption	Reads/writes a matrix in file using binary format.
ASCII_VIEW	QcViewOption	Reads/writes a matrix in file using ASCII format.

[1] Available if `QCMATRIX_STORAGE_MODE` is set in CMake.

[2] Both `BINARY_VIEW` and `ASCII_VIEW` are available if `QCMATRIX_ENABLE_VIEW` is set in CMake.

For Fortran users, the types in Table 3.1 are different. Please refer to Table 3.2 for the convention of types in Fortran.

Table 3.2: Fortran type conventions.

Type in QcMatrix	**Fortran**
`struct QcMat`	`type(QcMat)`
`QErrorCode`	`integer`
`QChar`	`character*(*)`
`QInt`	`integer`
`QBool`	`logical`
`QReal`	`real(QREAL)`[3]
`QcDataType`	`integer`
`QcDuplicateOption`	`integer`
`QcMatOperation`	`integer`
`QcStorageMode`	`integer`
`QcSymType`	`integer`
`QcViewOption`	`integer`

The functions provided by QcMatrix are listed in Table 3.3, in which

1. All functions are implemented in the directory `src/qcmat`.
2. All functions should be used as `ierr = QcMat...(...)`, where `QErrorCode ierr` contains the error information. It should be `QSUCCESS` if no error happened.
3. All matrices must first be created by calling `QcMatCreate`, and finally destroyed by calling `QcMatDestroy`. Futher requirements could be needed for some functions (for instance, the matrix should be assembled by calling `QcMatAssemble`), please check the description of the functions in Table 3.3.
4. All functions can be used in the same way in Fortran code, but the types of some arguments are different from those in C/C++ code, please refer to Table 3.2 for the convention of types in Fortran.

Table 3.3: Public functions provided by QcMatrix.

Function/Arguments	**Description**
`QcMatCreate`	Creates the context of a matrix.
`QcMat *A`	*Input & output*, the matrix.
`QcMatBlockCreate`	Sets the dimension of blocks and creates the blocks.
`QcMat *A`	*Input & output*, the matrix, should be created by `QcMatCreate`.
`const QInt dim_block`	*Input*, the dimension of blocks.
`QcMatSetSymType`	Sets the symmetry type of a matrix.
`QcMat *A`	*Input & output*, the matrix, should be created by `QcMatCreate` and `QcMatBlockCreate`.
`const QcSymType sym_type`	*Input*, given symmetry type, see file `include/types/mat_symmetry.h`.
`QcMatSetDataType`	Sets the data types of matrix elements of some blocks.
`QcMat *A`	*Input & output*, the matrix, should be created by `QcMatCreate` and `QcMatBlockCreate`.

Continued on next page

[3]`QReal` and `QREAL` are the type of real numbers, determined by `QCMATRIX_SINGLE_PRECISION` during setting up CMake.

Table 3.3 – continued from previous page

Function/Arguments	Description
const QInt num_blocks	*Input*, number of blocks to set the data types.
const QInt idx_block_row[]	*Input*, row indices of the blocks.
const QInt idx_block_col[]	*Input*, column indices of the blocks.
const QcDataType block_data_types[]	*Input*, given data types of the blocks, see file `include/types/mat_data.h`.
QcMatSetDimMat	Sets the dimension of each block of a matrix.
QcMat *A	*Input & output*, the matrix, should be created by `QcMatCreate` and `QcMatBlockCreate`.
const QInt num_row	*Input*, number of rows of each block.
const QInt num_col	*Input*, number of columns of each block.
QcMatSetStorageMode	Sets the matrix storage mode of a matrix, enabled by setting `QCMATRIX_STORAGE_MODE` as `ON` in CMake.
QcMat *A	*Input & output*, the matrix, should be created by `QcMatCreate` and `QcMatBlockCreate`.
const QcStorageMode storage_mode	*Input*, given matrix storage mode, should be defined and implemented in external library.
QcMatAssemble	Assembles a matrix (e.g. allocating memory) so that it could be used in further matrix calculations, this function should be invoked after `QcMatCreate`, `QcMatBlockCreate` and `QcMatSet...`.
QcMat *A	*Input & output*, the matrix to be assembled.
QcMatGetDimBlock	Gets the dimension of blocks.
QcMat *A	*Input*, the matrix, should be at least created by `QcMatCreate` and `QcMatBlockCreate`.
QInt *dim_block	*Output*, the dimension of blocks.
QcMatGetSymType	Gets the symmetry type of a matrix.
QcMat *A	*Input*, the matrix, should be at least created by `QcMatCreate`.
QcSymType *sym_type	*Output*, symmetry type of the matrix, see file `include/types/mat_symmetry.h`.
QcMatGetDataType	Gets the data types of matrix elements of some blocks.
QcMat *A	*Input*, the matrix, should be at least created by `QcMatCreate` and `QcMatBlockCreate`.
const QInt num_blocks	*Input*, number of blocks to get the data types.
const QInt idx_block_row[]	*Input*, row indices of the blocks.
const QInt idx_block_col[]	*Input*, column indices of the blocks.
QcDataType *data_type	*Output*, data types of the blocks, see file `include/types/mat_data.h`.
QcMatGetDimMat	Gets the dimension of each block of a matrix.
QcMat *A	*Input*, the matrix, should be at least created by `QcMatCreate` and `QcMatBlockCreate`.
QInt *num_row	*Output*, number of rows of each block.
QInt *num_col	*Output*, number of columns of each block.
QcMatGetStorageMode	Gets the matrix storage mode of a matrix, enabled by setting `QCMATRIX_STORAGE_MODE` as `ON` in CMake.
QcMat *A	*Input*, the matrix, should be at least created by `QcMatCreate` and `QcMatBlockCreate`.
QcStorageMode *storage_mode	*Output*, return matrix storage mode, should be defined and implemented in external library.
QcMatIsAssembled	Checks if a matrix is assembled or not.

Continued on next page

Table 3.3 – continued from previous page

Function/Arguments	Description
QcMat *A	*Input*, the matrix, should be at least created by QcMatCreate and QcMatBlockCreate.
QBool *assembled	*Output*, indicates if the matrix is assembled or not.
QcMatSetValues	Sets the values of a matrix.
QcMat *A	*Input & output*, the matrix, should be created by QcMatCreate and QcMatBlockCreate.
const QInt idx_block_row	*Input*, index of the block row.
const QInt idx_block_col	*Input*, index of the block column.
const QInt idx_first_row	*Input*, index of the first row to set values.
const QInt num_row_set	*Input*, number of rows to set.
const QInt idx_first_col	*Input*, index of the first column to set values.
const QInt num_col_set	*Input*, number of columns to set.
const QReal *values_real	*Input*, values of the real part.
const QReal *values_imag	*Input*, values of the imaginary part.
QcMatAddValues	Adds the values to a matrix.
QcMat *A	*Input & output*, the matrix, should be created by QcMatCreate and QcMatBlockCreate.
const QInt idx_block_row	*Input*, index of the block row.
const QInt idx_block_col	*Input*, index of the block column.
const QInt idx_first_row	*Input*, index of the first row to add values.
const QInt num_row_add	*Input*, number of rows to add.
const QInt idx_first_col	*Input*, index of the first column to add values.
const QInt num_col_add	*Input*, number of columns to add.
const QReal *values_real	*Input*, values of the real part.
const QReal *values_imag	*Input*, values of the imaginary part.
QcMatGetValues	Gets the values of a matrix.
QcMat *A	*Input & output*, the matrix, should be at least assembled by QcMatAssemble, otherwise returns zero.
const QInt idx_block_row	*Input*, index of the block row.
const QInt idx_block_col	*Input*, index of the block column.
const QInt idx_first_row	*Input*, index of the first row to get values.
const QInt num_row_get	*Input*, number of rows to get.
const QInt idx_first_col	*Input*, index of the first column to get values.
const QInt num_col_get	*Input*, number of columns to get.
QReal *values_real	*Output*, values of the real part.
QReal *values_imag	*Output*, values of the imaginary part.
QcMatDuplicate	Duplicates a matrix.
QcMat *A	*Input*, the matrix, should be at least created by QcMatCreate and QcMatBlockCreate.
const QcDuplicateOption duplicate_option	*Input*, duplicate option, see file include/types/mat_duplicate.h.
QcMat *B	*Input & output*, the new matrix, should be at least created by QcMatCreate, and all its previous information will be destroyed.
QcMatZeroEntries	Zeros all entries of a matrix.
QcMat *A	*Input & output*, the matrix, should be at least created by QcMatCreate and QcMatBlockCreate.
QcMatGetTrace	Gets the traces of the first few diagonal blocks of a matrix.
QcMat *A	*Input*, the matrix, should be at least created by QcMatCreate and QcMatBlockCreate.

Continued on next page

Table 3.3 – continued from previous page

Function/Arguments	Description
`const QInt num_blocks`	*Input*, the number of diagonal blocks.
`QReal *trace`	*Output*, the traces, size is `2*\var{num_blocks}`.
`QcMatGetMatProdTrace`	Gets the traces of the first few diagonal blocks of a matrix-matrix product `A*op(B)`.
`QcMat *A`	*Input*, the left matrix, should be at least created by `QcMatCreate` and `QcMatBlockCreate`.
`QcMat *B`	*Input*, the right matrix, should be at least created by `QcMatCreate` and `QcMatBlockCreate`.
`const QcMatOperation op_B`	*Input*, the operation on the matrix `B`, see file `include/types/mat_operations.h`.
`const QInt num_blocks`	*Input*, the number of diagonal blocks.
`QReal *trace`	*Output*, the traces, size is `2*\var{num_blocks}`.
`QcMatDestroy`	Frees space taken by a matrix.
`QcMat *A`	*Input & output*, the matrix, should be at least created by `QcMatCreate`.
`QcMatWrite`	Writes a matrix to file, enabled by setting `QCMATRIX_ENABLE_VIEW` as `ON` in CMake.
`QcMat *A`	*Input*, the matrix, should be at least assembled by `QcMatAssemble`.
`const QChar *mat_label`	*Input*, label of the matrix, should be unique.
`const QcViewOption view_option`	*Input*, option of writing, see file `include/types/mat_view.h`.
`QcMatRead`	Reads a matrix from file, enabled by setting `QCMATRIX_ENABLE_VIEW` as `ON` in CMake.
`QcMat *A`	*Input & output*, the matrix, should be created by `QcMatCreate`.
`const QChar *mat_label`	*Input*, label of the matrix, should be unique.
`const QcViewOption view_option`	*Input*, option of reading, see file `include/types/mat_view.h`.
`QcMatScale`	Scales all elements of a matrix by a given (complex) number.
`const QReal scal_number[]`	*Input*, the scaling number with `scal_number[0]` being the real part and `scal_number[1]` the imaginary part.
`QcMat *A`	*Input & output*, the matrix to be scaled, should be at least assembled by `QcMatAssemble`.
`QcMatAXPY`	Computes `Y = a*X+Y`.
`const QReal multiplier[]`	*Input*, the complex multiplier `a` with `multiplier[0]` being the real part and `multiplier[1]` the imaginary part.
`QcMat *X`	*Input*, the first matrix, should be at least assembled by `QcMatAssemble`.
`QcMat *Y`	*Input & output*, the second matrix, should be at least created by `QcMatCreate`.
`QcMatTranspose`	Performs an in-place or out-of-place matrix operation `B = op(A)`.
`const QcMatOperation op_A`	*Input*, the operation on the matrix `A`, see file `include/types/mat_operations.h`.
`QcMat *A`	*Input*, the matrix to perform matrix operation, should be at least assembled by `QcMatAssemble`.
`QcMat *B`	*Input & output*, the result matrix; it could be `A`, or it should be at least created by `QcMatCreate`.

Continued on next page

Table 3.3 – continued from previous page

Function/Arguments	**Description**
`QcMatGEMM`	Performs matrix-matrix multiplication `C = alpha*op(A)*op(B)+beta*C`, where valid operations `op(...)` can be found in file `include/types/mat_operations.h`.
`const QcMatOperation op_A`	*Input*, the operation on the matrix `A`, see file `include/types/mat_operations.h`.
`const QcMatOperation op_B`	*Input*, the operation on the matrix `B`, see file `include/types/mat_operations.h`.
`const QReal alpha[]`	*Input*, the scalar number.
`QcMat *A`	*Input*, the left matrix, should be at least assembled by `QcMatAssemble`.
`QcMat *B`	*Input*, the right matrix, should be at least assembled by `QcMatAssemble`.
`const QReal beta[]`	*Input*, the scalar number.
`QcMat *C`	*Input & output*, the product matrix, should be at least created by `QcMatCreate`, so that we require function `CmplxMatGEMM`[4] could assemble the matrix `C` if it is not.
`QcMatMatCommutator`	Calculates the commutator `C = [A,B] = A*B-B*A`.
`QcMat *A`	*Input*, the left matrix, should be at least assembled by `QcMatAssemble`
`QcMat *B`	*Input*, the right matrix, should be at least assembled by `QcMatAssemble`.
`QcMat *C`	*Input & output*, the result matrix, should be at least created by `QcMatCreate`.
`QcMatMatSCommutator`	Calculates the commutator `C = [A,B]_{S} = A*B*S-S*B*A`.
`QcMat *A`	*Input*, the left matrix, should be at least assembled by `QcMatAssemble`
`QcMat *B`	*Input*, the right matrix, should be at least assembled by `QcMatAssemble`.
`QcMat *S`	*Input*, the `S` matrix, should be at least assembled by `QcMatAssemble`.
`QcMat *C`	*Input & output*, the result matrix, should be at least created by `QcMatCreate`.
`QcMatMatHermCommutator`	Calculates the commutator `C = A*B-B*A^{\dagger}`.
`QcMat *A`	*Input*, the left matrix, should be at least assembled by `QcMatAssemble`
`QcMat *B`	*Input*, the right matrix, should be at least assembled by `QcMatAssemble`.
`QcMat *C`	*Input & output*, the result matrix, should be at least created by `QcMatCreate`.
`QcMatMatSHermCommutator`	Calculates the commutator `C = A*B*S-S*B*A^{\dagger}`.
`QcMat *A`	*Input*, the left matrix, should be at least assembled by `QcMatAssemble`
`QcMat *B`	*Input*, the right matrix, should be at least assembled by `QcMatAssemble`.
`QcMat *S`	*Input*, the `S` matrix, should be at least assembled by `QcMatAssemble`.

Continued on next page

[4] Function for the complex matrix-matrix multiplication.

Table 3.3 – continued from previous page

Function/Arguments	Description
`QcMat *C`	*Input & output*, the result matrix, should be at least created by `QcMatCreate`.

To facilitate the communication between the application libraries and host programs, QcMatrix has further provided two functions[5]

1. `QcMatSetExternalMat`: sets an external matrix as (part of) the matrix of QcMatrix[6].
2. `QcMatGetExternalMat`: gets an external matrix from (part of) the matrix of QcMatrix.

The former `QcMatSetExternalMat` can be used, for instance, in the interface of calling the application library from the host program[7]:

```
subroutine host_interface(A_host, ...)
    implicit none
    type(LANG_F_MATRIX), intent(inout) :: A_host
    type(QcMat) A
    ! Creates the matrix A and sets its information
    ... ...
    ! Sets the real part of block(1,1) of the matrix A
    ierr = QcMatSetExternalMat(A=A,              &
                               idx_block_row=1,    &
                               idx_block_col=1,    &
                               data_type=QREALMAT, &
                               A_ext=A_host)
    if (ierr/=QSUCCESS) then
        ... ...
    end if
    ! Calls application library using the matrix A
    ... ...
    ! Destroys the matrix A
    ierr = QcMatDestroy(A)
    if (ierr/=QSUCCESS) then
        ... ...
    end if
    return
end subroutine host_tdrsp_interface
```

The later `QcMatGetExternalMat` can be used in the “callback” function for the application library:

```
subroutine host_callback(A, ...)
    implicit none
    type(QcMat), intent(inout) :: A
    type(LANG_F_MATRIX), pointer :: A_host
    ! Gets the real part of block(1,1) of the matrix A
```

[5]It seems no reason that someone uses QcMatrix together with external C square block matrix library. So that these two functions are not available for the external C square block matrix library.

[6]The context of the external matrix will not be destroyed by `QcMatDestroy`.

[7]In this example and following “callback” function example, we assume that the host program has implemented the real matrix.

```
    ierr = QcMatGetExternalMat(A=A,              &
                               idx_block_row=1,  &
                               idx_block_col=1,  &
                               data_type=QREALMAT, &
                               A_ext=A_host)
    if (ierr/=QSUCCESS) then
        ... ...
    end if
    ! Calls the subroutines in the host program using A_host
    ... ...
    return
  end subroutine host_callback
```

It should be noted that users should **not** manipulate the external matrix `A_host` (from the function `QcMatGetExternalMat()`) if it is `QNULLMAT` (or in other words, if it is not assembled)[8].

Details regarding how to implement these two functions can be found, for instance in Sections 9.3 and 9.4. In Table 3.4, we give the descriptions of the arguments of these two functions with respect to different implemented external C matrices. In Table 3.5, we only list the arguments of these two functions with respect to different implemented external Fortran matrices.

Table 3.4: Functions `QcMatSetExternalMat` and `QcMatGetExternalMat` (C version).

External matrix	**Arguments**	**Description**
Square block real	`QcMatSetExternalMat`	
	`QcMat *A`	*Input & output*, the matrix, should be at least created by `QcMatCreate` and `QcMatBlockCreate`.
	`const QcDataType data_type`	*Input*, which part to set, see file `include/types/mat_data.h`.
	`LANG_C_MATRIX A_ext`	*Input*, the square block real matrix implemented in the external library.
	`QcMatGetExternalMat`	
	`QcMat *A`	*Input*, the matrix, should be at least created by `QcMatCreate` and `QcMatBlockCreate`.
	`const QcDataType data_type`	*Input*, which part to get, see file `include/types/mat_data.h`.
	`LANG_C_MATRIX **A_ext`	*Output*, the square block real matrix implemented in the external library.
Complex	`QcMatSetExternalMat`	
	`QcMat *A`	*Input & output*, the matrix, should be at least created by `QcMatCreate` and `QcMatBlockCreate`.
	`const QInt idx_block_row`	*Input*, index of the block row.
	`const QInt idx_block_col`	*Input*, index of the block column.
	`LANG_C_MATRIX A_ext`	*Input*, the complex matrix implemented in the external library.
	`QcMatGetExternalMat`	
	`QcMat *A`	*Input*, the matrix, should be at least created by `QcMatCreate` and `QcMatBlockCreate`.

Continued on next page

[8] If you have such requirement, please write to the authors.

Table 3.4 – continued from previous page

External matrix	Arguments	Description
	const QInt idx_block_row	*Input*, index of the block row.
	const QInt idx_block_col	*Input*, index of the block column.
	LANG_C_MATRIX **A_ext	*Output*, the complex matrix implemented in the external library.
Real	QcMatSetExternalMat	
	QcMat *A	*Input & output*, the matrix, should be at least created by `QcMatCreate` and `QcMatBlockCreate`.
	const QInt idx_block_row	*Input*, index of the block row.
	const QInt idx_block_col	*Input*, index of the block column.
	const QcDataType data_type	*Input*, which part to set, see file `include/types/mat_data.h`.
	LANG_C_MATRIX A_ext	*Input*, the real matrix implemented in the external library.
	QcMatGetExternalMat	
	QcMat *A	*Input*, the matrix, should be at least created by `QcMatCreate` and `QcMatBlockCreate`.
	const QInt idx_block_row	*Input*, index of the block row.
	const QInt idx_block_col	*Input*, index of the block column.
	const QcDataType data_type	*Input*, which part to get, see file `include/types/mat_data.h`.
	LANG_C_MATRIX **A_ext	*Output*, the real matrix implemented in the external library.

Table 3.5: Functions `QcMatSetExternalMat` and `QcMatGetExternalMat` (Fortran version).

External matrix	Arguments
Square block complex	QcMatSetExternalMat
	type(QcMat), intent(inout) :: A
	type(LANG_F_MATRIX), intent(in) :: A_ext
	QcMatGetExternalMat
	type(QcMat), intent(in) :: A
	type(LANG_F_MATRIX), pointer, intent(inout) :: A_ext
Square block real	QcMatSetExternalMat
	type(QcMat), intent(inout) :: A
	integer, intent(in) :: data_type
	type(LANG_F_MATRIX), intent(in) :: A_ext
	QcMatGetExternalMat
	type(QcMat), intent(in) :: A
	integer, intent(in) :: data_type
	type(LANG_F_MATRIX), pointer, intent(inout) :: A_ext
Complex	QcMatSetExternalMat
	type(QcMat), intent(inout) :: A
	integer, intent(in) :: idx_block_row
	integer, intent(in) :: idx_block_col
	type(LANG_F_MATRIX), intent(in) :: A_ext
	QcMatGetExternalMat
	type(QcMat), intent(in) :: A
	integer, intent(in) :: idx_block_row

Continued on next page

Table 3.5 – continued from previous page

External matrix	Arguments
	`integer, intent(in) :: idx_block_col` `type(LANG_F_MATRIX), pointer, intent(inout) :: A_ext`

Real	`QcMatSetExternalMat`
	`type(QcMat), intent(inout) :: A`
	`integer, intent(in) :: idx_block_row`
	`integer, intent(in) :: idx_block_col`
	`integer, intent(in) :: data_type`
	`type(LANG_F_MATRIX), intent(in) :: A_ext`

	`QcMatGetExternalMat`
	`type(QcMat), intent(in) :: A`
	`integer, intent(in) :: idx_block_row`
	`integer, intent(in) :: idx_block_col`
	`integer, intent(in) :: data_type`
	`type(LANG_F_MATRIX), pointer, intent(inout) :: A_ext`

Moreover, we have implemented several functions mainly for the purpose of tests, which can be found in the directory `src/qcmat/tests`, and are given in Table 3.6.

Table 3.6: Private functions in QcMatrix.

Function/Arguments	Description
`QcMatSetRandMat`	Sets the data types and values of a matrix randomly according to its symmetry and data types, may be only for test suite.
`QcMat *A`	*Input & output*, the matrix, should be created by `QcMatCreate`.
`const QcSymType sym_type`	*Input*, given symmetry type, see file `include/types/mat_symmetry.h`.
`const QcDataType data_type`	*Input*, given data type of the matrix, see file `include/types/mat_data.h`.
`const QInt dim_block`	*Input*, the dimension of blocks.
`const QInt num_row`	*Input*, number of rows of each block.
`const QInt num_col`	*Input*, number of columns of each block.
`QcMatIsEqual`	Compares if two matrices are equal, may be only used for test suite.
`QcMat *A`	*Input*, the matrix, should be at least created by `QcMatCreate`.
`QcMat *B`	*Input*, the matrix, should be at least created by `QcMatCreate`.
`const QBool cf_values`	*Input*, indicates if comparing values.
`QBool *is_equal`	*Output*, indicates if two matrices are equal (pattern and/or values).
`QcMatCfArray`	Compares if the values of a matrix and two arrays (real and imaginary parts) are equal, may be only used for test suite.
`QcMat *A`	*Input*, the matrix, should be at least created by `QcMatCreate`.
`const QBool row_major`	*Input*, if given values in row major order.
`const QInt size_values`	*Input*, the size of values of the real and imaginary parts.
`const QReal *values_real`	*Input*, the values of real part.
`const QReal *values_imag`	*Input*, the values of imaginary part.
`QBool *is_equal`	*Output*, indicates if the values of the matrix and the arrays are equal.
`QcMatGetAllValues`	Gets all values of a matrix, may be only used for test suite.
`QcMat *A`	*Input*, the matrix, should be at least created by `QcMatCreate`.
`const QBool row_major`	*Input*, if returning values in row major order.
`const QInt size_values`	*Input*, the size of values of the real and imaginary parts.
`QReal *values_real`	*Output*, values of the real part.
`QReal *values_imag`	*Output*, values of the imaginary part.

Chapter 4

Requisite for External Library

This chapter describes the requisite for external libraries to be able to use the QcMatrix library. For the time being, QcMatrix only support external libraries written in C, C++ or Fortran languages.

Except for the required header files/modules and implemented matrix in Table 1.2, external libraries also need to implement different functions/subroutines required by QcMatrix. In the following sections, we describe these functions in detail for C, C++ and Fortran external libraries.

4.1 C External Library

The required functions from the C external library are given in Table 4.1 (as well as in the file `include/types/mat_adapter.h`).

Table 4.1: External functions required by QcMatrix.

Function/Arguments	**Description**
`Matrix_Create`	Creates the context of a matrix.
`LANG_C_MATRIX *A`	*Input & output*, the matrix.
`Matrix_BlockCreate`	Sets the dimension of blocks and creates the blocks (**Only required for external square block complex or square block real matrix library**).
`LANG_C_MATRIX *A`	*Input & output*, the matrix, should be created by `Matrix_Create`.
`const QInt dim_block`	*Input*, the dimension of blocks.
`Matrix_SetSymType`	Sets the symmetry type of a matrix.
`LANG_C_MATRIX *A`	*Input & output*, the matrix, should be created by `Matrix_Create` and `Matrix_BlockCreate`.
`const QcSymType sym_type`	*Input*, given symmetry type, see file `include/types/mat_symmetry.h`.
`Matrix_SetDataType`	Sets the data types of matrix elements of some blocks (**Only required for external square block complex matrix library**).
`LANG_C_MATRIX *A`	*Input & output*, the matrix, should be created by `Matrix_Create` and `Matrix_BlockCreate`.
`const QInt num_blocks`	*Input*, number of blocks to set the data types.
`const QInt idx_block_row[]`	*Input*, row indices of the blocks.
`const QInt idx_block_col[]`	*Input*, column indices of the blocks.

Continued on next page

Table 4.1 – continued from previous page

Function/Arguments	Description
`const QcDataType block_data_types[]`	*Input*, given data types of the blocks, see file `include/types/mat_data.h`.
`Matrix_SetNonZeroBlocks`	Sets the non-zero blocks of a square block real matrix (**Only required for external square block real matrix library**).
`LANG_C_MATRIX *A`	*Input & output*, the matrix, should be created by `Matrix_Create` and `Matrix_BlockCreate`.
`const QInt num_blocks`	*Input*, number of non-zero blocks to set.
`const QInt idx_block_row[]`	*Input*, row indices of the non-zero blocks.
`const QInt idx_block_col[]`	*Input*, column indices of the non-zero blocks.
`Matrix_SetDataType`	Sets the data type of a complex matrix (**Only required for external complex matrix library**).
`LANG_C_MATRIX *A`	*Input & output*, the matrix, should be created by `Matrix_Create`.
`const QcDataType data_type`	*Input*, given data type of the matrix, see file `include/types/mat_data.h`.
`Matrix_SetDimMat`	Sets the dimension of each block of a matrix.
`LANG_C_MATRIX *A`	*Input & output*, the matrix, should be created by `Matrix_Create` and `Matrix_BlockCreate`.
`const QInt num_row`	*Input*, number of rows of each block.
`const QInt num_col`	*Input*, number of columns of each block.
`Matrix_SetStorageMode`	Sets the matrix storage mode of a matrix, enabled by setting `QCMATRIX_STORAGE_MODE` as `ON` in CMake.
`LANG_C_MATRIX *A`	*Input & output*, the matrix, should be created by `Matrix_Create` and `Matrix_BlockCreate`.
`const QcStorageMode storage_mode`	*Input*, given matrix storage mode, should be defined and implemented in external library.
`Matrix_Assemble`	Assembles a matrix (e.g. allocating memory) so that it could be used in further matrix calculations, this function should be invoked after `Matrix_Create`, `Matrix_BlockCreate` and `Matrix_Set...`.
`LANG_C_MATRIX *A`	*Input & output*, the matrix to be assembled.
`Matrix_GetDimBlock`	Gets the dimension of blocks (**Only required for external square block complex or square block real matrix library**).
`LANG_C_MATRIX *A`	*Input*, the matrix, should be at least created by `Matrix_Create` and `Matrix_BlockCreate`.
`QInt *dim_block`	*Output*, the dimension of blocks.
`Matrix_GetSymType`	Gets the symmetry type of a matrix.
`LANG_C_MATRIX *A`	*Input*, the matrix, should be at least created by `Matrix_Create`.
`QcSymType *sym_type`	*Output*, symmetry type of the matrix, see file `include/types/mat_symmetry.h`.
`Matrix_GetDataType`	Gets the data types of matrix elements of some blocks (**Only required for external square block complex matrix library**).
`LANG_C_MATRIX *A`	*Input*, the matrix, should be at least created by `Matrix_Create` and `Matrix_BlockCreate`.
`const QInt num_blocks`	*Input*, number of blocks to get the data types.
`const QInt idx_block_row[]`	*Input*, row indices of the blocks.

Continued on next page

Table 4.1 – continued from previous page

Function/Arguments	**Description**
const QInt idx_block_col[]	*Input*, column indices of the blocks.
QcDataType *data_type	*Output*, data types of the blocks, see file include/types/mat_data.h.
Matrix_GetNonZeroBlocks	Checks if some given blocks of a square block real matrix are non-zero blocks, (**Only required for external square block real matrix library**).
LANG_C_MATRIX *A	*Input*, the matrix, should be created by Matrix_Create and Matrix_BlockCreate.
const QInt num_blocks	*Input*, number of blocks to check.
const QInt idx_block_row[]	*Input*, row indices of the blocks.
const QInt idx_block_col[]	*Input*, column indices of the blocks.
QBool *nz_blocks	*Output*, QTRUE if the block is non-zero block, otherwise QFALSE.
Matrix_GetDataType	Gets the data type of a complex matrix (**Only required for external complex matrix library**).
LANG_C_MATRIX *A	*Input*, the matrix, should be created by Matrix_Create.
QcDataType *data_type	*Output*, the data type of the matrix, see file include/types/mat_data.h.
Matrix_GetDimMat	Gets the dimension of each block of a matrix.
LANG_C_MATRIX *A	*Input*, the matrix, should be at least created by Matrix_Create and Matrix_BlockCreate.
QInt *num_row	*Output*, number of rows of each block.
QInt *num_col	*Output*, number of columns of each block.
Matrix_GetStorageMode	Gets the matrix storage mode of a matrix, enabled by setting QCMATRIX_STORAGE_MODE as ON in CMake.
LANG_C_MATRIX *A	*Input*, the matrix, should be at least created by Matrix_Create and Matrix_BlockCreate.
QcStorageMode *storage_mode	*Output*, return matrix storage mode, should be defined and implemented in external library.
Matrix_IsAssembled	Checks if a matrix is assembled or not.
LANG_C_MATRIX *A	*Input*, the matrix, should be at least created by Matrix_Create and Matrix_BlockCreate.
QBool *assembled	*Output*, indicates if the matrix is assembled or not.
Matrix_SetValues	Sets the values of a matrix.
LANG_C_MATRIX *A	*Input & output*, the matrix, should be created by Matrix_Create and Matrix_BlockCreate.
const QInt idx_block_row	*Input*, index of the block row (**Only required for external square block complex or square block real matrix library**).
const QInt idx_block_col	*Input*, index of the block column (**Only required for external square block complex or square block real matrix library**).
const QInt idx_first_row	*Input*, index of the first row to set values.
const QInt num_row_set	*Input*, number of rows to set.
const QInt idx_first_col	*Input*, index of the first column to set values.
const QInt num_col_set	*Input*, number of columns to set.
const QReal *values_real	*Input*, values of the real part.
const QReal *values_imag	*Input*, values of the imaginary part (**Only required for external (square block) complex matrix library**).
Matrix_AddValues	Adds the values too a matrix.

Continued on next page

Table 4.1 – continued from previous page

Function/Arguments	Description
`LANG_C_MATRIX *A`	*Input & output*, the matrix, should be created by `Matrix_Create` and `Matrix_BlockCreate`.
`const QInt idx_block_row`	*Input*, index of the block row (**Only required for external square block complex or square block real matrix library**).
`const QInt idx_block_col`	*Input*, index of the block column (**Only required for external square block complex or square block real matrix library**).
`const QInt idx_first_row`	*Input*, index of the first row to add values.
`const QInt num_row_add`	*Input*, number of rows to add.
`const QInt idx_first_col`	*Input*, index of the first column to add values.
`const QInt num_col_add`	*Input*, number of columns to add.
`const QReal *values_real`	*Input*, values of the real part.
`const QReal *values_imag`	*Input*, values of the imaginary part (**Only required for external (square block) complex matrix library**).
`Matrix_GetValues`	Gets the values of a matrix.
`LANG_C_MATRIX *A`	*Input & output*, the matrix, should be at least assembled by `Matrix_Assemble`, otherwise returns zero.
`const QInt idx_block_row`	*Input*, index of the block row (**Only required for external square block complex or square block real matrix library**).
`const QInt idx_block_col`	*Input*, index of the block column(**Only required for external square block complex or square block real matrix library**).
`const QInt idx_first_row`	*Input*, index of the first row to get values.
`const QInt num_row_get`	*Input*, number of rows to get.
`const QInt idx_first_col`	*Input*, index of the first column to get values.
`const QInt num_col_get`	*Input*, number of columns to get.
`QReal *values_real`	*Output*, values of the real part.
`QReal *values_imag`	*Output*, values of the imaginary part (**Only required for external (square block) complex matrix library**).
`Matrix_Duplicate`	Duplicates a matrix.
`LANG_C_MATRIX *A`	*Input*, the matrix, should be at least created by `Matrix_Create` and `Matrix_BlockCreate`.
`const QcDuplicateOption duplicate_option`	*Input*, duplicate option, see file `include/types/mat_duplicate.h`.
`LANG_C_MATRIX *B`	*Input & output*, the new matrix, should be at least created by `Matrix_Create`, and all its previous information will be destroyed.
`Matrix_ZeroEntries`	Zeros all entries of a matrix.
`LANG_C_MATRIX *A`	*Input & output*, the matrix, should be at least created by `Matrix_Create` and `Matrix_BlockCreate`.
`Matrix_GetTrace`	Gets the traces of the first few diagonal blocks of a matrix.
`LANG_C_MATRIX *A`	*Input*, the matrix, should be at least created by `Matrix_Create` and `Matrix_BlockCreate`.
`const QInt num_blocks`	*Input*, the number of diagonal blocks (**Only required for external square block complex or square block real matrix library**).

Continued on next page

Table 4.1 – continued from previous page

Function/Arguments	Description
QReal *trace	*Output*, the traces, size is 2*\var{num_blocks} (**external square block complex matrix library**), \var{num_blocks} (**external square block real matrix library**), 2 (**external complex matrix library**), or 1 (**external real matrix library**).
Matrix_GetMatProdTrace	Gets the traces of the first few diagonal blocks of a matrix-matrix product A*op(B).
LANG_C_MATRIX *A	*Input*, the left matrix, should be at least created by Matrix_Create and Matrix_BlockCreate.
LANG_C_MATRIX *B	*Input*, the right matrix, should be at least created by Matrix_Create and Matrix_BlockCreate.
const QcMatOperation op_B	*Input*, the operation on the matrix B, see file include/types/mat_operations.h (**the Hermitian transpose MAT_HERM_TRANSPOSE and complex conjugate MAT_COMPLEX_CONJUGATE are not needed for external (square block) real matrix library**).
const QInt num_blocks	*Input*, the number of diagonal blocks (**Only required for external square block complex or square block real matrix library**).
QReal *trace	*Output*, the traces, size is 2*\var{num_blocks} (**external square block complex matrix library**), \var{num_blocks} (**external square block real matrix library**), 2 (**external complex matrix library**), or 1 (**external real matrix library**).
Matrix_Destroy	Frees space taken by a matrix.
LANG_C_MATRIX *A	*Input & output*, the matrix, should be at least created by Matrix_Create.
Matrix_Write	Writes a matrix to file, enabled by setting QCMATRIX_ENABLE_VIEW as ON in CMake.
LANG_C_MATRIX *A	*Input*, the matrix, should be at least assembled by Matrix_Assemble.
const QChar *mat_label	*Input*, label of the matrix, should be unique.
const QcViewOption view_option	*Input*, option of writing, see file include/types/mat_view.h.
Matrix_Read	Reads a matrix from file, enabled by setting QCMATRIX_ENABLE_VIEW as ON in CMake.
LANG_C_MATRIX *A	*Input & output*, the matrix, should be created by Matrix_Create.
const QChar *mat_label	*Input*, label of the matrix, should be unique.
const QcViewOption view_option	*Input*, option of reading, see file include/types/mat_view.h.
Matrix_Scale	Scales all elements of a matrix by a given (complex) number.
const QReal scal_number[]	*Input*, the scaling number with scal_number[0] being the real part and scal_number[1] the imaginary part (**Only required for external (square block) complex matrix library**).
const QReal scal_number	*Input*, the scaling number (**Only required for external (square block) real matrix library**).

Continued on next page

Table 4.1 – continued from previous page

Function/Arguments	Description
LANG_C_MATRIX *A	*Input & output*, the matrix to be scaled, should be at least assembled by Matrix_Assemble.
Matrix_AXPY	Computes Y = a*X+Y.
const QReal multiplier[]	*Input*, the complex multiplier a with multiplier[0] being the real part and multiplier[1] the imaginary part (**Only required for external (square block) complex matrix library**).
const QReal multiplier	*Input*, the multiplier a (**Only required for external (square block) real matrix library**).
LANG_C_MATRIX *X	*Input*, the first matrix, should be at least assembled by Matrix_Assemble.
LANG_C_MATRIX *Y	*Input & output*, the second matrix, should be at least created by Matrix_Create.
Matrix_Transpose	Performs an in-place or out-of-place matrix operation B = op(A).
const QcMatOperation op_A	*Input*, the operation on the matrix A, see file include/types/mat_operations.h (**the Hermitian transpose MAT_HERM_TRANSPOSE and complex conjugate MAT_COMPLEX_CONJUGATE are not needed for external (square block) real matrix library**).
LANG_C_MATRIX *A	*Input*, the matrix to perform matrix operation, should be at least assembled by Matrix_Assemble.
LANG_C_MATRIX *B	*Input & output*, the result matrix; it could be A, or it should be at least created by Matrix_Create.
Matrix_GEMM	Performs matrix-matrix multiplication C = alpha*op(A)*op(B)+beta*C, where valid operations op(...) can be found in file include/types/mat_operations.h.
const QcMatOperation op_A	*Input*, the operation on the matrix A, see file include/types/mat_operations.h (**the Hermitian transpose MAT_HERM_TRANSPOSE and complex conjugate MAT_COMPLEX_CONJUGATE are not needed for external (square block) real matrix library**).
const QcMatOperation op_B	*Input*, the operation on the matrix B, see file include/types/mat_operations.h (**the Hermitian transpose MAT_HERM_TRANSPOSE and complex conjugate MAT_COMPLEX_CONJUGATE are not needed for external (square block) real matrix library**).
const QReal alpha[]	*Input*, the scalar number (**Only required for external (square block) complex matrix library**).
const QReal alpha	*Input*, the scalar number (**Only required for external (square block) real matrix library**).
LANG_C_MATRIX *A	*Input*, the left matrix, should be at least assembled by Matrix_Assemble.
LANG_C_MATRIX *B	*Input*, the right matrix, should be at least assembled by Matrix_Assemble.
const QReal beta[]	*Input*, the scalar number (**Only required for external (square block) complex matrix library**).
const QReal beta	*Input*, the scalar number (**Only required for external (square block) real matrix library**).

Continued on next page

Table 4.1 – continued from previous page

Function/Arguments	Description
`LANG_C_MATRIX *C`	*Input & output*, the product matrix, should be at least created by `Matrix_Create`, so that we require function `Matrix_GEMM` could assemble the matrix `C` if it is not.

4.2 C++ External Library

4.3 Fortran External Library

The Fortran external library should implement subroutines (instead of functions) which can carry out the same functionalities and arguments as thoes in Table 4.1. These subroutines should also be named as `Matrix_...`, and used as `call Matrix_...(...)`. Instead of `struct LANG_C_MATRIX`, these subroutines will take `type(LANG_F_MATRIX)` as input/output. Other type conventions can be found in Table 3.2.

Chapter 5

Self-consistent Field Solvers

molecular and periodic systems?

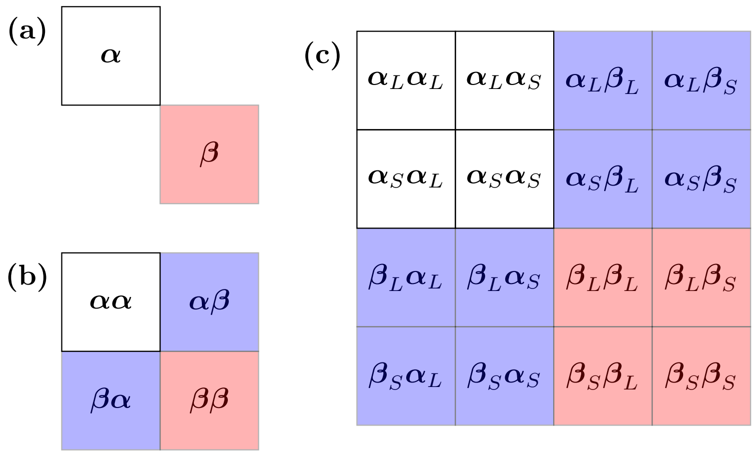

Figure 5.1: (a) One- (b) two- and (c) four-component open-shell matrices (red part allocated only if open-shell, blue part allocated only with spin-orbit coupling, only upper and diagonal parts might be allocated for Hermitian or anti-Hermitian matrix).

Chapter 6

Linear Response Solvers

molecular and periodic systems?

Chapter 7

X-ray Spectroscopies

In this chapter, we will describe the procedure to simulate different X-ray spectroscopies from the molecular orbital coefficients $\mathbf{C}$, which might be ground state, $Z+1$ approximation, or others depending on the approximation you have chosen.

7.1 Transition Dipole Moment

Using molecular orbital (MO) coefficients $\mathbf{C}$, the transition dipole moment between two molecular orbitals I and J is simply

$$\boldsymbol{r}_{IJ} = \left[\mathbf{C}^{\dagger}\hat{\mathbf{r}}\mathbf{C}\right]_{IJ}, \tag{7.1}$$

where $\hat{\mathbf{r}} = (\mathbf{x}, \mathbf{y}, \mathbf{z})$ is the dipole length integrals in atomic orbitals (AOs).

The oscillator strength of the one-photon absorption (OPA) could be simply calculated from $\boldsymbol{r}_{IJ}$ based on dipole approximation, sudden approximation and final state rule, as (in atomic unit)

$$\mathsf{f}_{IJ} = 2\varepsilon_{IJ}|\boldsymbol{\epsilon}\cdot\boldsymbol{r}_{IJ}|^2, \tag{7.2}$$

where ε_{IJ} is the energy difference between MOs I and J, $\boldsymbol{\epsilon}$ is the polarization vector of incoming light.

For molecule system with random orientation, we may use the averaged absorption oscillator strength over the x, y and z components, which gives

$$\mathsf{f}_{IJ} = \frac{2}{3}\varepsilon_{IJ}|\boldsymbol{r}_{IJ}|^2. \tag{7.3}$$

The transition rate of non-resonant (or normal) x-ray emission spectroscopy (XES) could also be evaluated from $\boldsymbol{r}_{IJ}$ by noticing it is proportional to the Einstein A coefficient (using final state rule)

$$\mathrm{I}^{\mathrm{XES}}_{IJ} \propto \varepsilon^3_{IJ}|\boldsymbol{r}_{IJ}|^2. \tag{7.4}$$

Moreover, the final state of XES process is a valence ionized state, which in practical calculation is usually approximated as the ground state (ignoring the effect of the valence hole).

7.2 Ultraviolet Photoelectron Spectra

In Mulliken population analysis (MPA) the gross atomic population of atom $\boldsymbol{A}$ is

$$\mathrm{GAP}_{\boldsymbol{A}} = \sum_J n_J \sum_{\mu\in\boldsymbol{A}}\sum_{\nu}\mathbf{C}^*_{\mu J}\mathbf{C}_{\nu J}\mathbf{S}_{\mu\nu} = \sum_J n_J \sum_{\mu\in\boldsymbol{A}}\mathbf{C}^*_{\mu J}(\mathbf{SC})_{\mu J} = \sum_J n_J \sum_{\mu\in\boldsymbol{A}}\mathbf{P}^{\boldsymbol{A}}_{\mu J}, \tag{7.5}$$

where J runs over all MOs, n_J is the occupation number of Jth MO. μ represents AOs of atom $\boldsymbol{A}$, and ν runs over all AOs. $\mathbf{S}_{\mu\nu}$ is the overlap integrals between AOs. $\mathbf{P}^{\boldsymbol{A}}_{\mu J}$ is the gross atomic population on atom $\boldsymbol{A}$ from the AO μ in the Jth MO

$$\mathbf{P}^{\boldsymbol{A}}_{\mu J} = \mathbf{C}^*_{\mu J}(\mathbf{SC})_{\mu J}. \tag{7.6}$$

Based on (a) the Born-Oppenheimer approximation, (b) the sudden approximation, (c) the plane wave approximation for the free electron and (d) the photoionization cross section of the Jth molecular orbital should be independent of the interatomic shape of the molecular orbital and should be expressed as a sum of atomic terms (sum over all atoms $\boldsymbol{A}$), i.e.

$$\sigma_J = \sum_{\boldsymbol{A}} \sigma^{\boldsymbol{A}}_J, \tag{7.7}$$

Gelius[2, 3] proved that the intensity of the Jth molecular orbital $\mathrm{I}^{\mathrm{PES}}_J$ for photoelectron spectroscopy (PES) could be written as

$$\mathrm{I}^{\mathrm{PES}}_J \propto \sum_{\boldsymbol{A}} \sum_{\mu \in \boldsymbol{A}} \frac{\sigma^{\boldsymbol{A}}_\mu}{\sigma^{\boldsymbol{A}}_{\mu 0}} \mathbf{P}^{\boldsymbol{A}}_{\mu J} = \sum_{\boldsymbol{A}} \sum_{\mu \in \boldsymbol{A}} \frac{\sigma^{\boldsymbol{A}}_\mu}{\sigma^{\boldsymbol{A}}_{\mu 0}} \mathbf{C}^*_{\mu J}(\mathbf{SC})_{\mu J} = \left[\mathbf{C}^\dagger_\sigma \mathbf{SC}\right]_{JJ}, \tag{7.8}$$

where $\sigma^{\boldsymbol{A}}_\mu$ is the atomic subshell μ photoionization cross section that could be obtained from Ref. [4] and $\sigma^{\boldsymbol{A}}_{\mu 0}$ is the photoionization cross section of a particular atomic subshell μ_0 (which might be neglected during calculations). $\mathbf{C}_\sigma$ is MO coefficients $\mathbf{C}$ scaled by $\frac{\sigma^{\boldsymbol{A}}_\mu}{\sigma^{\boldsymbol{A}}_{\mu 0}}$ on each row (AO μ).

7.3 Resonant Inelastic X-ray Scattering

give the final formulas using matrix operations for code ...

The resonant inelastic x-ray scattering (RIXS) process is more properly viewed as a quasi-simultaneous two-photon absorption-emission process, whose cross section is expressed by the Kramers-Heisenberg scattering amplitude[5, 6]:

$$F_{\nu n}(\omega, \omega') = \sum_k \alpha \omega_{\nu k} \omega_{nk}(\nu) \left[\frac{(\mathbf{d}_{\nu k} \cdot \mathbf{e}_1)(\mathbf{e}_2 \cdot \mathbf{d}_{kn}(\nu))}{\omega - \omega_{\nu k} + i\Gamma_{\nu k}} + \frac{(\mathbf{e}_2 \cdot \mathbf{d}_{\nu k})(\mathbf{d}_{kn}(\nu) \cdot \mathbf{e}_1)}{\omega' + \omega_{\nu k}} \right], \tag{7.9}$$

where we have used atomic units ($h = m_e = e = 1$, $\alpha = \frac{1}{137}$). The indices k represent a core level, n a valence occupied level, and ν an unoccupied level. $\mathbf{d}_{\nu k}$ is the probability for the absorption ($k \to \nu$) and $\mathbf{d}_{kn}(\nu)$ the probability for the emission ($n \to k$) transitions. The remaining terms ω and ω' and $\mathbf{e}_1$ and $\mathbf{e}_2$ are the frequencies and the polarization vectors of the incoming and emitted photons. $\Gamma_{\nu k}$ is the lifetime of the intermediate state. The first term of this expression is also denoted as the resonant anomalous scattering term, and is responsible for a resonance in case ω equals $\omega_{\nu k}$. The second term, the non resonant scattering term, is important only far from resonance and can therefore be neglected at resonance. The differential cross section of RIXS for scattering in a solid angle is[7–9]:

$$\frac{d^2\sigma}{d\omega' d\Omega} = \sum_\nu \sum_n \frac{\omega'}{\omega} |F_{\nu n}(\omega)|^2 \Delta(\omega - \omega' - \omega_{\nu n}, \Gamma_{\nu n}), \tag{7.10}$$

where $d\Omega$ is the solid angle of photon scattering, $\omega_{\nu n}$ is the resonant frequency of the optical transition $n \to \nu$ and $\Gamma_{\nu n}$ is the final-state life time broadening. The Δ function can be written as

$$\Delta(\omega, \Gamma) = \frac{\Gamma}{\pi(\omega^2 + \Gamma^2)}. \tag{7.11}$$

Eq. (7.10) describes the x-ray fluorescence spectra excited by monochromatic x-ray beams. To describe a realistic experimental situation, we must consider the convolution[7–9]

$$\frac{d\sigma(\omega',\omega_0)}{d\Omega} = \int d\omega \frac{d^2\sigma}{d\omega' d\Omega}\Phi(\omega-\omega_0,\gamma) \tag{7.12}$$

of the differential cross section with the incoming photon distribution function $\Phi(\omega-\omega_0,\gamma)$ centered at frequency ω_0. At the present time theoretical evaluations of the incoming photon distribution function $\Phi(\Omega,\gamma)$ seem to be lacking, but most often in numerical simulations a simple Gaussian form is used[9]

$$\Phi(\Omega,\gamma) = \frac{1}{\gamma}\sqrt{\frac{\ln 2}{\pi}} \exp\left[-\left(\frac{\Omega}{\gamma}\right)^2 \ln 2\right] \tag{7.13}$$

with γ as the half width at half maximum (HWHM).

Notice that the lifetime broadening, $\Gamma_{\nu n}$, of the optical transition $n \to \nu$ is negligibly small in comparison with the width of x-ray transitions $\Gamma_{\nu k}$. This allows us to use $\Gamma_{\nu n} = 0$ and to replace the Δ function in Eq. (7.10) by the Dirac δ-function $\delta(\omega-\omega'-\omega_{\nu n})$ and the convolution of Eq. (7.12) becomes[7, 8]

$$\frac{d\sigma(\omega',\omega_0)}{d\Omega} = \sum_{\nu n} \frac{\omega'}{\omega}|F_{\nu n}(\omega)|^2 \Phi(\omega'+\omega_{\nu n}-\omega_0,\gamma), \tag{7.14}$$

which is restricted only by the width γ of the spectral function Φ of incoming x-rays and, of course, by the instrumental resolution. The frequency ω' of the emitted x-ray photons has a Raman related shift (Stokes shift) into the long-wave region relative to the frequency ω of the absorbed photon

$$\omega' = \omega + \omega_{\nu n} \tag{7.15}$$

in accordance with the energy conservation law reflected by the $\delta(\omega-\omega'-\omega_{\nu n})$ function.

For samples in gas or solution phases it is necessary to average the cross section (7.14) over all molecular orientations. This is equivalent to averaging the quantity $|F_{\nu n}(\omega)|^2$ which we now consider. Luo *et al.*[7, 8] have developed a general average procedure:

$$\langle |F_{\nu n}|^2\rangle = \lambda_{\nu n} = F\lambda^F_{\nu n} + G\lambda^G_{\nu n} + H\lambda^H_{\nu n}, \tag{7.16}$$

with

$$\lambda^F_{\nu n} = \sum_\beta F^{\beta\beta}_{\nu n} \sum_\gamma F^{\gamma\gamma *}_{\nu n}, \tag{7.17}$$

$$\lambda^G_{\nu n} = \sum_{\beta\gamma} F^{\beta\gamma}_{\nu n} F^{\beta\gamma *}_{\nu n}, \tag{7.18}$$

$$\lambda^H_{\nu n} = \sum_{\beta\gamma} F^{\beta\gamma}_{\nu n} F^{\gamma\beta *}_{\nu n}, \tag{7.19}$$

and

$$F^{\beta\gamma}_{\nu n} = \alpha \sum_k \omega_{\nu k}\omega_{nk}(\nu) \frac{d^\beta_{\nu k} d^\gamma_{kn}(\nu)}{\omega' - \omega_{nk} + i\Gamma_{\nu k}}. \tag{7.20}$$

The F, G and H factors are

$$F = -|\mathbf{e}_1 \cdot \mathbf{e}_2^*|^2 + 4|\mathbf{e}_1 \cdot \mathbf{e}_2|^2 - 1, \tag{7.21}$$

$$G = -|\mathbf{e}_1 \cdot \mathbf{e}_2^*|^2 - |\mathbf{e}_1 \cdot \mathbf{e}_2|^2 + 4, \tag{7.22}$$

$$H = 4|\mathbf{e}_1 \cdot \mathbf{e}_2^*|^2 - |\mathbf{e}_1 \cdot \mathbf{e}_2|^2 - 1. \tag{7.23}$$

And finally, the averaged cross section is given by[7, 8]

$$\begin{aligned}\langle\sigma(\omega',\omega_0)\rangle &= \sum_{\nu n}\frac{\omega'}{\omega}\lambda_{\nu n}\Phi(\omega'+\omega_{\nu n}-\omega_0,\gamma) \\ &= \sum_{\nu n}\frac{\omega'}{\omega}(F\lambda^F_{\nu n}+G\lambda^G_{\nu n}+H\lambda^H_{\nu n})\Phi(\omega'+\omega_{\nu n}-\omega_0,\gamma).\end{aligned} \tag{7.24}$$

From Eq. (7.24), the expressions of molecular parameters $\lambda^F_{\nu n}$, $\lambda^G_{\nu n}$, $\lambda^H_{\nu n}$ and the F, G, H factors, we can see that the cross section of RIXS has a strong dependence on the polarization of the absorbed and emitted photons, and on the symmetries of the electronic levels involved. The general symmetry selection rules for RIXS have been expressed by means of group theory[7, 8].

7.4 XPS Shake-up Process

In the equivalent core hole time-dependent density functional theory (ECH-TDDFT) method[10, 11], the core hole in XPS (x-ray photoelectron spectroscopy) has been approximated by the equivalent core. The valence excitations in the presence of the core hole are computed by using TDDFT calculations within the ECH approximation. Within ECH-TDDFT, a two-step model[11] has been adopted. In the first step, a core electron is emitted by the x-ray photon and left with a core hole, which is approximated by the equivalent core. In the second step, the electron excitations between a valence and a virtual orbital occur in the presence of the core hole which is approximated as the equivalent core. Therefore, for the N-electron system, the n'-th final state after the second step can be written as,

$$\Psi_{fn'}(N) = \sum_n a_{n'n}\psi_f^{(n)}(N), \tag{7.25}$$

where $a_{n'n}$ is the CI expansion coefficients which can be obtained from TDDFT calculations. $\psi_f^{(n)}(N)$ is the so-called excited "configuration state functions" (CSF)[12], and $\psi_f^{(0)}(N)$ is the determinant of the positive charged equivalent core system in the ECH approximation.

The intensity ratio of the n'-th XPS shake-up peak to the main peak is thus[12],

$$\frac{I(n')}{I(0)} \simeq \frac{|\sum_n a_{n'n}S_n|^2}{|\sum_n a_{0n}S_n|^2}. \tag{7.26}$$

Here S_n is the overlap between the ground state $\psi_g(N)$ and the CSF $\psi_f^{(n)}(N)$. Assuming a valence orbital I has been excited into a virtual orbital J in $\psi_f^{(n)}(N, I\to J)$, then

$$\begin{aligned} S_n &= \langle\psi_f^{(n)}(N, I\to J)|\psi_g(N)\rangle, \\ &= \begin{vmatrix} \langle 1'|1\rangle & \dots & \langle 1'|I\rangle & \dots & \langle 1'|N\rangle \\ \langle 2'|1\rangle & \dots & \langle 2'|I\rangle & \dots & \langle 2'|N\rangle \\ \dots & \dots & \dots & \dots & \dots \\ \langle J'|1\rangle & \dots & \langle J'|I\rangle & \dots & \langle J'|N\rangle \\ \dots & \dots & \dots & \dots & \dots \\ \langle N'|1\rangle & \dots & \langle N'|I\rangle & \dots & \langle N'|N\rangle \end{vmatrix}, \end{aligned} \tag{7.27}$$

where the abbreviated notation $\langle J'|I\rangle$ is the one-electron orbital overlap integral

$$\langle\phi'_J|\phi_I\rangle = \sum_{\mu\nu}\mathbf{C}^*_{f,\mu J}\mathbf{C}_{g,\nu I}\langle\chi_{f,\mu J}|\chi_{g,\nu I}\rangle = \sum_{\mu\nu}\mathbf{C}^*_{f,\mu J}\mathbf{C}_{g,\nu I}\mathbf{S}_{fg,\mu\nu} = \left[\mathbf{C}_f^\dagger\mathbf{S}_{fg}\mathbf{C}_g\right]_{JI}. \tag{7.28}$$

Therefore we have

$$S_n = \det\left[\mathbf{C}_f^\dagger \mathbf{S}_{fg} \mathbf{C}_g\right], \tag{7.29}$$

in which $\mathbf{S}_{fg}$ could be approximated as $\mathbf{S}_f$ or $\mathbf{S}_g$.

Chapter 8

Molecular Electronics

8.1 Scattering Theory Approach

The Hamiltonian of a typical molecular junction with a molecule (M) sandwiched between two electron reservoirs, the source (S) and the drain (D) could be written as[13]

$$\hat{H} = \begin{pmatrix} \hat{H}^{\mathrm{S}} & \hat{U}^{\mathrm{SM}} & \hat{U}^{\mathrm{SD}} \\ \hat{U}^{\mathrm{MS}} & \hat{H}^{\mathrm{M}} & \hat{U}^{\mathrm{MD}} \\ \hat{U}^{\mathrm{DS}} & \hat{U}^{\mathrm{DM}} & \hat{H}^{\mathrm{D}} \end{pmatrix}, \tag{8.1}$$

where $\hat{H}^{\mathrm{S,D,M}}$ are respectively the Hamiltonian of subsystems S, D and M, and different $\hat{U}$'s represent the interactions between or among subsystems. Similarly, the eigenstate (molecular orbital) $|\phi_I\rangle$ of the Hamiltonian $\hat{H}$ at energy level ε_I can be partitioned into

$$|\phi_I\rangle = \left|\phi_I^{\mathrm{S}}\right\rangle + \left|\phi_I^{\mathrm{M}}\right\rangle + \left|\phi_I^{\mathrm{D}}\right\rangle, \tag{8.2}$$

where $\phi_I^{\mathrm{S,D,M}}$ are respectively the molecular orbitals of subsystems S, D and M, and can be expressed as linear combinations of atomic orbitals (AOs) $\chi_\mu^{\mathrm{S,D,M}}$

$$\left|\phi_I^{\mathrm{S}}\right\rangle = \sum_\mu \mathbf{C}_{\mu I}^{\mathrm{S}} \left|\chi_\mu^{\mathrm{S}}\right\rangle, \tag{8.3}$$

$$\left|\phi_I^{\mathrm{M}}\right\rangle = \sum_\mu \mathbf{C}_{\mu I}^{\mathrm{M}} \left|\chi_\mu^{\mathrm{M}}\right\rangle, \tag{8.4}$$

$$\left|\phi_I^{\mathrm{D}}\right\rangle = \sum_\mu \mathbf{C}_{\mu I}^{\mathrm{D}} \left|\chi_\mu^{\mathrm{D}}\right\rangle, \tag{8.5}$$

with $\mathbf{C}$'s being the molecular orbital (MO) coefficients.

Normally, we require that $\left|\phi_I^{\mathrm{S}}\right\rangle$, $\left|\phi_I^{\mathrm{M}}\right\rangle$ and $\left|\phi_I^{\mathrm{D}}\right\rangle$ are orthonormal[1],

$$\left\langle \phi_I^{\mathrm{A}} \middle| \phi_J^{\mathrm{B}} \right\rangle = \delta_{\mathrm{AB}}\delta_{IJ}, \tag{8.6}$$

which will be used for Eq. (8.11).

[1] In the actual calculation, we mostly use the extended molecular model, which contains only one atom layer for the source electrode, one atom layer for the drain electrode, and the sandwiched molecule. In this case, we acutally only have the wave function of ϕ^M. We further approximate the electrode-molecule couplings to be the couplings between the molcule and one atom layer of the electrodes.

Therefore, the interaction between the subsystems is[13, 14]

$$\hat{U} = \sum_I \hat{U}_I = \sum_I \left(V^{\mathrm{SM}} \left|\phi_I^{\mathrm{S}}\right\rangle \left\langle \phi_I^{\mathrm{M}}\right| + V^{\mathrm{MD}} \left|\phi_I^{\mathrm{M}}\right\rangle \left\langle \phi_I^{\mathrm{D}}\right| + V^{\mathrm{SD}} \left|\phi_I^{\mathrm{S}}\right\rangle \left\langle \phi_I^{\mathrm{D}}\right| + \text{c.c.} \right), \tag{8.7}$$

where V^{AB} represents the coupling energy between the subsystems A and B

$$V^{\mathrm{AB}} = \sum_I^{\mathrm{occ.}} \left\langle \phi_I^{\mathrm{A}} \left| \hat{H} \right| \phi_I^{\mathrm{B}} \right\rangle = \sum_I^{\mathrm{occ.}} \sum_{\mu\nu} \left(\mathbf{C}_{\mu I}^{\mathrm{A}}\right)^\dagger \mathbf{C}_{\nu I}^{\mathrm{B}} \left\langle \chi_\mu^{\mathrm{A}} \left| \hat{H} \right| \chi_\nu^{\mathrm{B}} \right\rangle = \sum_I^{\mathrm{occ.}} \left(\mathbf{C}_I^{\mathrm{A}}\right)^\dagger \mathbf{H}^{\mathrm{AB}} \mathbf{C}_I^{\mathrm{B}}. \tag{8.8}$$

Based on the elastic-scattering Green's function theory, the transition operator is defined as

$$\hat{T} = \hat{U} + \hat{U}\hat{G}\hat{U}, \tag{8.9}$$

where $\hat{G}$ is the Green's function

$$\hat{G}(z) = \left(z - \hat{H}\right)^{-1}. \tag{8.10}$$

By considering Eq. (8.7), for an electron scattering from the initial state $\left|\phi_I^{\mathrm{S}}\right\rangle$ of reservoir S to the final state $\left|\phi_J^{\mathrm{D}}\right\rangle$ of reservoir D, the transition matrix element will be[13]

$$\begin{aligned} \mathbf{T}_{JI} &= \left\langle \phi_J^{\mathrm{D}} \left| \hat{U} \right| \phi_I^{\mathrm{S}} \right\rangle + \left\langle \phi_J^{\mathrm{D}} \left| \hat{U}\hat{G}\hat{U} \right| \phi_I^{\mathrm{S}} \right\rangle \\ &= V^{\mathrm{DM}} V^{\mathrm{MS}} \left\langle \phi_J^{\mathrm{M}} \left| \hat{G} \right| \phi_I^{\mathrm{M}} \right\rangle + V^{\mathrm{DS}} V^{\mathrm{MS}} \left\langle \phi_J^{\mathrm{S}} \left| \hat{G} \right| \phi_I^{\mathrm{M}} \right\rangle + V^{\mathrm{DM}} V^{\mathrm{DS}} \left\langle \phi_J^{\mathrm{M}} \left| \hat{G} \right| \phi_I^{\mathrm{D}} \right\rangle, \end{aligned} \tag{8.11}$$

where we have used the fact that there is no direct coupling between two reservoirs S and D. $\left\langle \phi_J^{\mathrm{A}} \left| \hat{G} \right| \phi_I^{\mathrm{B}} \right\rangle$ is the carrier-conduction contribution from the scattering channel ε_I, which can be expressed as

$$\begin{aligned} \left\langle \phi_J^{\mathrm{A}} \left| \hat{G} \right| \phi_I^{\mathrm{B}} \right\rangle &= \left\langle \phi_J^{\mathrm{A}} \left| \frac{1}{z - \hat{H}} \right| \phi_I^{\mathrm{B}} \right\rangle \\ &= \sum_K \left\langle \phi_J^{\mathrm{A}} \left| \frac{1}{z - \hat{H}} \right| \phi_K \right\rangle \left\langle \phi_K \middle| \phi_I^{\mathrm{B}} \right\rangle \\ &= \sum_K \frac{\left\langle \phi_J^{\mathrm{A}} \middle| \phi_K \right\rangle \left\langle \phi_K \middle| \phi_I^{\mathrm{B}} \right\rangle}{z - \varepsilon_K} \\ &= \sum_K \frac{\left(\sum_{\mu\nu} \left(\mathbf{C}_{\mu J}^{\mathrm{A}}\right)^\dagger \mathbf{C}_{\nu K} \left\langle \chi_\mu^{\mathrm{A}} \middle| \chi_\nu \right\rangle \right) \left(\sum_{\mu\nu} \mathbf{C}_{\mu K}^\dagger \mathbf{C}_{\nu I}^{\mathrm{B}} \left\langle \chi_\mu \middle| \chi_\nu^{\mathrm{B}} \right\rangle \right)}{z - \varepsilon_K} \\ &= \sum_K \frac{\left[\left(\mathbf{C}_J^{\mathrm{A}}\right)^\dagger \mathbf{S}^{\mathrm{A}} \mathbf{C}_K \right] \left[\mathbf{C}_K^\dagger \left(\mathbf{S}^{\mathrm{B}}\right)^\dagger \mathbf{C}_I^{\mathrm{B}} \right]}{z - \varepsilon_K}, \end{aligned} \tag{8.12}$$

where parameter z in the Green's function is a complex variable, $z = E + \mathrm{i}\Gamma$, and E is the energy at which the scattering process is observed. $1/\Gamma$ is the escape rate, which is determined by the Fermi Golden rule[13] *(I need to check if it is correct.)*

$$\begin{aligned} \Gamma_{K,IJ}^{\mathrm{AB,CD}} &= \pi n^{\mathrm{A}}(E_f) \left(V^{\mathrm{AB}}\right)^2 \left| \left\langle \phi_I^{\mathrm{B}} \middle| \phi_K \right\rangle \right|^2 + \pi n^{\mathrm{D}}(E_f) \left(V^{\mathrm{CD}}\right)^2 \left| \left\langle \phi_K \middle| \phi_J^{\mathrm{D}} \right\rangle \right|^2 \\ &= \pi n^{\mathrm{A}}(E_f) \left(V^{\mathrm{AB}}\right)^2 \left| \left(\mathbf{C}_I^{\mathrm{B}}\right)^\dagger \mathbf{S}^{\mathrm{B}} \mathbf{C}_K \right|^2 + \pi n^{\mathrm{D}}(E_f) \left(V^{\mathrm{CD}}\right)^2 \left| \mathbf{C}_K^\dagger \left(\mathbf{S}^{\mathrm{D}}\right)^\dagger \mathbf{C}_J^{\mathrm{D}} \right|^2, \end{aligned} \tag{8.13}$$

where $n^{\mathrm{A}}(E_f)$ and $n^{\mathrm{D}}(E_f)$ are the density of states (DOS) of the subsystems A and D at the Fermi level E_f, respectively.

Electric current through a molecular wire can be computed by integrating the transition probability over all energy states in the reservoir. We assume that the molecule is aligned along the z direction, which is also the direction of current transport. In the effective mass approximation, energy states in the conduction band of the reservoir can be expressed as the summation, $E = E_{x,y} + E_z + E_c$, where E_c is the condition band edge and is used as energy reference. It is assumed that the parabolic dispersion relation for the energy states in metal holds. The electrons in the reservoir are assumed to be all in equilibrium at a temperature T and Fermi level E_f. When an applied voltage V is introduced, the tunneling current density from source (S) to drain (D) is[13–15] ***(this is just copied from JJ's previous JCP paper, needs to rewrite ...)***

$$i_{\mathrm{SD}} = \frac{2\pi e}{\hbar} \sum_{E_{x,y}} \sum_{E_z^I, E_z^J} \left[f\left(E_{x,y} + E_z^I - eV\right) - f\left(E_{x,y} + E_z^J\right)\right] T_{JI} \delta\left(E_z^J - E_z^I\right), \tag{8.14}$$

where $f(E)$ is the Fermi distribution function

$$f(E) = \frac{1}{\exp\left(\frac{E-E_f}{k_B T}\right) + 1}, \tag{8.15}$$

and k_B the Boltzmann constant. $T_{JI} = |\mathbf{T}_{JI}|^2$ is the transition probability of the scattering process from the initial state $\left|\phi_I^{\mathrm{S}}\right\rangle$ to the final state $\left|\phi_J^{\mathrm{D}}\right\rangle$, as a function of the quantized injection energies along the z axis, E_z^I and E_z^J.

Last but not least, once we get the molcule-electrode couplings V^{AB}, we only need a few molecular orbitals close to the Fermi level to calculate the electron or hole transportation according to Eq. (8.12). Normally, we could choose the orbitals that are within 10 eV energy difference to the Fermi level.

8.2 Spin-orbit Coupling

As regards the unperturbed systems, we have in a matrix form

$$\begin{pmatrix} \mathbf{H}^\alpha & \mathbf{0} \\ \mathbf{0} & \mathbf{H}^\beta \end{pmatrix} \begin{pmatrix} \mathbf{C}^\alpha & \mathbf{0} \\ \mathbf{0} & \mathbf{C}^\beta \end{pmatrix} = \boldsymbol{\varepsilon} \begin{pmatrix} \mathbf{S}^\alpha & \mathbf{0} \\ \mathbf{0} & \mathbf{S}^\beta \end{pmatrix} \begin{pmatrix} \mathbf{C}^\alpha & \mathbf{0} \\ \mathbf{0} & \mathbf{C}^\beta \end{pmatrix}, \tag{8.16}$$

or

$$\mathbf{HC} = \boldsymbol{\varepsilon}\mathbf{SC}, \tag{8.17}$$

where the diagonal matrix $\boldsymbol{\varepsilon}$ contains the energy levels of all spin orbitals, and $\mathbf{S}^\alpha = \mathbf{S}^\beta$.

The spin-orbit coupling (SOC) term acting on the conductance electron is[16]

$$\hat{H}_{\mathrm{SO}} = \frac{\alpha^2}{4} \nabla V \cdot (\hat{\sigma} \times \hat{p}), \tag{8.18}$$

where we have used the atomic unit. α is the fine structure constant, the Pauli matrices $\hat{\sigma} = (\boldsymbol{\sigma}_x, \boldsymbol{\sigma}_y, \boldsymbol{\sigma}_z)$ and the momentum operator $\hat{p} = -\mathrm{i}\nabla$.

The electrostatic potential is[17]

$$V(\boldsymbol{r}) = V_{\mathrm{nuc}}(\boldsymbol{r}) + V_{\mathrm{elec}}(\boldsymbol{r}) = \sum_A \frac{Z_A}{|\boldsymbol{r} - \boldsymbol{R}_A|} - \int \frac{\rho(\boldsymbol{r}')}{|\boldsymbol{r} - \boldsymbol{r}'|} \mathrm{d}^3\boldsymbol{r}', \tag{8.19}$$

where the sum of A runs over all atoms at the position $\boldsymbol{R}_A$ with charge Z_A. $\rho(\boldsymbol{r}')$ is the electron density at the space point $\boldsymbol{r}'$. The second term in Eq. (8.19) involves the two-electron contributions, which is time-consuming. If we could approximate $V(\boldsymbol{r})$ as a screening one-electron potential

$$V(\boldsymbol{r}) \approx \sum_A \frac{Z_A^{\text{eff}}}{|\boldsymbol{r}-\boldsymbol{R}_A|}, \tag{8.20}$$

where Z_A^{eff} denotes the effective charge, and might be able to get from Ref. [18]. We therefore have

$$\begin{aligned}\left\langle \chi_\mu(\boldsymbol{r}) \left| \hat{H}_{\text{SO}} \right| \chi_\nu(\boldsymbol{r}) \right\rangle &\approx -\frac{\alpha^2}{4}\sum_A \left\langle \chi_\mu(\boldsymbol{r}) \left| \frac{Z_A^{\text{eff}}\boldsymbol{r}_A}{|\boldsymbol{r}-\boldsymbol{R}_A|^3}\cdot(\hat{\sigma}\times\hat{p}) \right| \chi_\nu(\boldsymbol{r}) \right\rangle \\ &= -\frac{\mathrm{i}\alpha^2}{4}\sum_A \left\langle \chi_\mu(\boldsymbol{r}) \left| \frac{Z_A^{\text{eff}}}{|\boldsymbol{r}-\boldsymbol{R}_A|^3}\hat{\sigma}\cdot(\boldsymbol{r}_A\times\nabla) \right| \chi_\nu(\boldsymbol{r}) \right\rangle,\end{aligned} \tag{8.21}$$

and the first-order energy correction to a spin orbital I could be obtained from the perturbation theory as

$$\varepsilon_I^{(1)} = \frac{\mathbf{C}_I^\dagger \mathbf{H}_{\text{SO}} \mathbf{C}_I}{\mathbf{C}_I^\dagger \mathbf{S} \mathbf{C}_I} = \mathbf{C}_I^\dagger \mathbf{H}_{\text{SO}} \mathbf{C}_I = 0, \tag{8.22}$$

by using that

$$\mathbf{H}\mathbf{C}_I = \varepsilon_I \mathbf{S}\mathbf{C}_I, \tag{8.23}$$

$$\mathbf{C}_I^\dagger \mathbf{H} = \varepsilon_I \mathbf{C}_I^\dagger \mathbf{S}, \tag{8.24}$$

$$\mathbf{C}_I^\dagger \mathbf{S}\mathbf{C}_I = \langle \phi_I | \phi_I \rangle = 1. \tag{8.25}$$

The MO coefficients with the first-order correction satisfies

$$\mathbf{H}\mathbf{C}_I^{(1)} + \mathbf{H}_{\text{SO}}\mathbf{C}_I = \varepsilon_I \mathbf{S}\mathbf{C}_I^{(1)} + \varepsilon_I^{(1)}\mathbf{S}\mathbf{C}_I = \varepsilon_I \mathbf{S}\mathbf{C}_I^{(1)}, \tag{8.26}$$

$$(\varepsilon_I \mathbf{S} - \mathbf{H})\,\mathbf{C}_I^{(1)} = \mathbf{H}_{\text{SO}}\mathbf{C}_I, \tag{8.27}$$

or in the orthonormal basis sets (for instance using the Cholesky decomposition of overlap matrix)

$$\left(\varepsilon_I \mathbf{I} - \mathbf{H}^{\text{orth}}\right)\left(\mathbf{C}^{(1)}\right)_I^{\text{orth}} = \mathbf{H}_{\text{SO}}^{\text{orth}}\mathbf{C}_I^{\text{orth}}, \tag{8.28}$$

where $\mathbf{I}$ is the identity matrix.

For restricted calculations, $\mathbf{H}^\alpha = \mathbf{H}^\beta$ and $\mathbf{C}^\alpha = \mathbf{C}^\beta$, and notice that

$$\mathbf{H}_{\text{SO}} = \begin{pmatrix} \mathrm{i}\mathbf{H}_1 & \mathbf{H}_2 + \mathrm{i}\mathbf{H}_3 \\ -\mathbf{H}_2 + \mathrm{i}\mathbf{H}_3 & -\mathrm{i}\mathbf{H}_1 \end{pmatrix}, \tag{8.29}$$

where $\mathbf{H}_{1,2,3}$ are anti-symmetric matrices, we therefore have

$$\mathbf{C}_I' = \begin{pmatrix} \mathbf{C}_I^\alpha \\ \mathbf{0} \end{pmatrix} + \mathbf{C}_I'^{(1)} = \begin{pmatrix} \mathbf{C}_I^\alpha + \mathrm{i}\mathbf{x}_I \\ \mathbf{y}_I + \mathrm{i}\mathbf{z}_I \end{pmatrix}, \tag{8.30}$$

$$\mathbf{C}_I'' = \begin{pmatrix} \mathbf{0} \\ \mathbf{C}_I^\alpha \end{pmatrix} + \mathbf{C}_I''^{(1)} = \begin{pmatrix} -\mathbf{y}_I + \mathrm{i}\mathbf{z}_I \\ \mathbf{C}_I^\alpha - \mathrm{i}\mathbf{x}_I \end{pmatrix}. \tag{8.31}$$

The projection $\langle \phi_I | \phi_K \rangle$ of α spin of ϕ_I includes

$$\begin{aligned}
\mathbf{P}^\alpha \mathbf{C}_I'^\dagger \mathbf{S} \mathbf{C}_K' &= \left(\mathbf{C}_I^{\alpha,T} - \mathrm{i}\mathbf{x}_I^T, \quad \mathbf{0} \right) \begin{pmatrix} \mathbf{S}^\alpha & \mathbf{0} \\ \mathbf{0} & \mathbf{S}^\alpha \end{pmatrix} \begin{pmatrix} \mathbf{C}_K^\alpha + \mathrm{i}\mathbf{x}_K \\ \mathbf{y}_K + \mathrm{i}\mathbf{z}_K \end{pmatrix} \\
&= \mathbf{C}_I^{\alpha,T} \mathbf{S}^\alpha \mathbf{C}_K^\alpha + \mathbf{x}_I^T \mathbf{S}^\alpha \mathbf{x}_K + \mathrm{i} \left(\mathbf{C}_I^{\alpha,T} \mathbf{S}^\alpha \mathbf{x}_K - \mathbf{x}_I^T \mathbf{S}^\alpha \mathbf{C}_K^\alpha \right),
\end{aligned} \tag{8.32}$$

$$\begin{aligned}
\mathbf{P}^\alpha \mathbf{C}_I''^\dagger \mathbf{S} \mathbf{C}_K' &= \left(-\mathbf{y}_I^T - \mathrm{i}\mathbf{z}_I^T, \quad \mathbf{0} \right) \begin{pmatrix} \mathbf{S}^\alpha & \mathbf{0} \\ \mathbf{0} & \mathbf{S}^\alpha \end{pmatrix} \begin{pmatrix} \mathbf{C}_K^\alpha + \mathrm{i}\mathbf{x}_K \\ \mathbf{y}_K + \mathrm{i}\mathbf{z}_K \end{pmatrix} \\
&= -\mathbf{y}_I^T \mathbf{S}^\alpha \mathbf{C}_K^\alpha + \mathbf{z}_I^T \mathbf{S}^\alpha \mathbf{x}_K + \mathrm{i} \left(-\mathbf{y}_I^T \mathbf{S}^\alpha \mathbf{x}_K - \mathbf{z}_I^T \mathbf{S}^\alpha \mathbf{C}_K^\alpha \right),
\end{aligned} \tag{8.33}$$

$$\begin{aligned}
\mathbf{P}^\alpha \mathbf{C}_I'^\dagger \mathbf{S} \mathbf{C}_K'' &= \left(\mathbf{C}_I^{\alpha,T} - \mathrm{i}\mathbf{x}_I^T, \quad \mathbf{0} \right) \begin{pmatrix} \mathbf{S}^\alpha & \mathbf{0} \\ \mathbf{0} & \mathbf{S}^\alpha \end{pmatrix} \begin{pmatrix} -\mathbf{y}_K + \mathrm{i}\mathbf{z}_K \\ \mathbf{C}_K^\alpha - \mathrm{i}\mathbf{x}_K \end{pmatrix} \\
&= -\mathbf{C}_I^{\alpha,T} \mathbf{S}^\alpha \mathbf{y}_K + \mathbf{x}_I^T \mathbf{S}^\alpha \mathbf{z}_K + \mathrm{i} \left(\mathbf{C}_I^{\alpha,T} \mathbf{S}^\alpha \mathbf{z}_K + \mathbf{x}_I^T \mathbf{S}^\alpha \mathbf{y}_K \right),
\end{aligned} \tag{8.34}$$

$$\begin{aligned}
\mathbf{P}^\alpha \mathbf{C}_I''^\dagger \mathbf{S} \mathbf{C}_K'' &= \left(-\mathbf{y}_I^T - \mathrm{i}\mathbf{z}_I^T, \quad \mathbf{0} \right) \begin{pmatrix} \mathbf{S}^\alpha & \mathbf{0} \\ \mathbf{0} & \mathbf{S}^\alpha \end{pmatrix} \begin{pmatrix} -\mathbf{y}_K + \mathrm{i}\mathbf{z}_K \\ \mathbf{C}_K^\alpha - \mathrm{i}\mathbf{x}_K \end{pmatrix} \\
&= \mathbf{y}_I^T \mathbf{S}^\alpha \mathbf{y}_K + \mathbf{z}_I^T \mathbf{S}^\alpha \mathbf{z}_K + \mathrm{i} \left(-\mathbf{y}_I^T \mathbf{S}^\alpha \mathbf{z}_K + \mathbf{z}_I^T \mathbf{S}^\alpha \mathbf{y}_K \right),
\end{aligned} \tag{8.35}$$

where

$$\mathbf{P}^\alpha = \begin{pmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{pmatrix}. \tag{8.36}$$

Similarly, the projection $\langle \phi_I | \phi_K \rangle$ of β spin of ϕ_I includes

$$\mathbf{P}^\beta \mathbf{C}_I'^\dagger \mathbf{S} \mathbf{C}_K' = \left(\mathbf{0}, \quad \mathbf{y}_I^T - \mathrm{i}\mathbf{z}_I^T \right) \begin{pmatrix} \mathbf{S}^\alpha & \mathbf{0} \\ \mathbf{0} & \mathbf{S}^\alpha \end{pmatrix} \begin{pmatrix} \mathbf{C}_K^\alpha + \mathrm{i}\mathbf{x}_K \\ \mathbf{y}_K + \mathrm{i}\mathbf{z}_K \end{pmatrix} = \left(\mathbf{P}^\alpha \mathbf{C}_I''^\dagger \mathbf{S} \mathbf{C}_K'' \right)^\dagger, \tag{8.37}$$

$$\mathbf{P}^\beta \mathbf{C}_I''^\dagger \mathbf{S} \mathbf{C}_K' = \left(\mathbf{0}, \quad \mathbf{C}_I^{\alpha,T} + \mathrm{i}\mathbf{x}_I^T \right) \begin{pmatrix} \mathbf{S}^\alpha & \mathbf{0} \\ \mathbf{0} & \mathbf{S}^\alpha \end{pmatrix} \begin{pmatrix} \mathbf{C}_K^\alpha + \mathrm{i}\mathbf{x}_K \\ \mathbf{y}_K + \mathrm{i}\mathbf{z}_K \end{pmatrix} = -\left(\mathbf{P}^\alpha \mathbf{C}_I'^\dagger \mathbf{S} \mathbf{C}_K'' \right)^\dagger, \tag{8.38}$$

$$\mathbf{P}^\beta \mathbf{C}_I'^\dagger \mathbf{S} \mathbf{C}_K'' = \left(\mathbf{0}, \quad \mathbf{y}_I^T - \mathrm{i}\mathbf{z}_I^T \right) \begin{pmatrix} \mathbf{S}^\alpha & \mathbf{0} \\ \mathbf{0} & \mathbf{S}^\alpha \end{pmatrix} \begin{pmatrix} -\mathbf{y}_K + \mathrm{i}\mathbf{z}_K \\ \mathbf{C}_K^\alpha - \mathrm{i}\mathbf{x}_K \end{pmatrix} = -\left(\mathbf{P}^\alpha \mathbf{C}_I''^\dagger \mathbf{S} \mathbf{C}_K' \right)^\dagger, \tag{8.39}$$

$$\mathbf{P}^\beta \mathbf{C}_I''^\dagger \mathbf{S} \mathbf{C}_K'' = \left(\mathbf{0}, \quad \mathbf{C}_I^{\alpha,T} + \mathrm{i}\mathbf{x}_I^T \right) \begin{pmatrix} \mathbf{S}^\alpha & \mathbf{0} \\ \mathbf{0} & \mathbf{S}^\alpha \end{pmatrix} \begin{pmatrix} -\mathbf{y}_K + \mathrm{i}\mathbf{z}_K \\ \mathbf{C}_K^\alpha - \mathrm{i}\mathbf{x}_K \end{pmatrix} = \left(\mathbf{P}^\alpha \mathbf{C}_I'^\dagger \mathbf{S} \mathbf{C}_K' \right)^\dagger, \tag{8.40}$$

where

$$\mathbf{P}^\beta = \begin{pmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{pmatrix}. \tag{8.41}$$

Chapter 9

Advanced Topics of QcMatrix

9.1 Square Block Complex Matrix

For external square block complex matrix library, QcMatrix does not support the external C library. As regarding the Fortran library, please refer to Sections 9.3 and 9.4 for the implementation of Fortran adapters.

For external square block real matrix library, the `struct QcMat` is implemented as

```
typedef struct {
    QcSymType sym_type;    /* Hermitian, anti-Hermitian or non-Hermitian */
    QcDataType data_type;  /* real, imaginary or complex */
    QInt real_part;        /* pointer to the real part if allocated, default 0 */
    QInt imag_part;        /* pointer to the imaginary part if allocated, default 1 */
    RealMat *cmplx_mat;    /* array with size of 2, for the real and imaginary parts */
} QcMat;
```

which is exactly the form of a complex matrix. So that all the functions implemented for the complex matrix (see Section 9.2 and the codes in directory `src/cmplx_mat`) will be used for the `struct QcMat`.

If external library has implemented complex or real matrix, as shown in Fig. 1.1 the `struct QcMat` is implemented as

```
typedef struct {
    QcSymType sym_type;  /* Hermitian, anti-Hermitian or non-Hermitian */
    QInt dim_block;      /* dimension of blocks */
    QBool **assembled;   /* \var(dim_block)x\var(dim_block) array indicating
                            if the blocks are assembled */
    CmplxMat **blocks;   /* \var(dim_block)x\var(dim_block) array for the blocks */
} QcMat;
```

and the codes can be found in directory `src/qcmat`. For the external real matrix, the `struct CmplxMat` and its corresponding functions are implemented and can be found in Section 9.2 and the codes in directory `src/cmplx_mat`.

In this section, we will discuss the codes in the directory `src/qcmat`.

9.1.1 Matrix-Matrix Multiplication

The Strassen method for the square block matrix-matrix multiplication:

$$\mathbf{A}=\begin{bmatrix}\mathbf{A}_{1,1} & \mathbf{A}_{1,2}\\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2}\end{bmatrix},\ \mathbf{B}=\begin{bmatrix}\mathbf{B}_{1,1} & \mathbf{B}_{1,2}\\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2}\end{bmatrix},\ \mathbf{C}=\begin{bmatrix}\mathbf{C}_{1,1} & \mathbf{C}_{1,2}\\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2}\end{bmatrix}, \tag{9.1}$$

$$\begin{aligned}
\mathbf{T}_1 &= (\mathbf{A}_{1,1}+\mathbf{A}_{2,2})(\mathbf{B}_{1,1}+\mathbf{B}_{2,2}),\\
\mathbf{T}_2 &= (\mathbf{A}_{2,1}+\mathbf{A}_{2,2})\mathbf{B}_{1,1},\\
\mathbf{T}_3 &= \mathbf{A}_{1,1}(\mathbf{B}_{1,2}-\mathbf{B}_{2,2}),\\
\mathbf{T}_4 &= \mathbf{A}_{2,2}(\mathbf{B}_{2,1}-\mathbf{B}_{1,1}),\\
\mathbf{T}_5 &= (\mathbf{A}_{1,1}+\mathbf{A}_{1,2})\mathbf{B}_{2,2},\\
\mathbf{T}_6 &= (\mathbf{A}_{2,1}-\mathbf{A}_{1,1})(\mathbf{B}_{1,1}+\mathbf{B}_{1,2}),\\
\mathbf{T}_7 &= (\mathbf{A}_{1,2}-\mathbf{A}_{2,2})(\mathbf{B}_{2,1}+\mathbf{B}_{2,2}),
\end{aligned} \tag{9.2}$$

$$\begin{aligned}
\mathbf{C}_{1,1} &= \mathbf{T}_1+\mathbf{T}_4-\mathbf{T}_5+\mathbf{T}_7,\\
\mathbf{C}_{1,2} &= \mathbf{T}_3+\mathbf{T}_5,\\
\mathbf{C}_{2,1} &= \mathbf{T}_2+\mathbf{T}_4,\\
\mathbf{C}_{2,2} &= \mathbf{T}_1-\mathbf{T}_2+\mathbf{T}_3+\mathbf{T}_6.
\end{aligned} \tag{9.3}$$

9.1.2 Cholesky Decomposition

9.1.3 Eigenvalue Solver

9.1.4 Linear Response Solver

9.1.5 Determinant

9.2 Complex Matrix

The complex matrix `struct CmplxMat` as shown in Fig. 1.1 is implemented as

```
typedef struct {
    QcSymType sym_type;    /* Hermitian, anti-Hermitian or non-Hermitian */
    QcDataType data_type;  /* real, imaginary or complex */
    QInt real_part;        /* pointer to the real part if allocated, default 0 */
    QInt imag_part;        /* pointer to the imaginary part if allocated, default 1 */
    RealMat *cmplx_mat;    /* array with size of 2, for the real and imaginary parts */
} CmplxMat;
```

in which `cmplx_mat[real_part]` and `cmplx_mat[imag_part]` stores respectively the real and imaginary part of the matrix.

The reason of introducing `real_part` and `imag_part` can be illustrated from the operation `CmplxMatScale` with a pure imaginary number, for instance,

$$\mathrm{i}a(\mathbf{A}_1+\mathrm{i}\mathbf{A}_2) = -a\mathbf{A}_2+\mathrm{i}a\mathbf{A}_1, \tag{9.4}$$

which could be easily done by

```
err_code = RealMatScale(scal_number[1], &A->cmplx_mat[A->real_part]);
err_code = RealMatScale(-scal_number[1], &A->cmplx_mat[A->imag_part]);
/* swaps the real and imaginary parts of the matrix */
A->real_part = A->imag_part;
A->imag_part = 1-A->imag_part;
/* changes the symmetry and data types of the matrix */
A->sym_type = -A->sym_type;
A->data_type = -A->data_type;
```

9.2.1 Matrix-Matrix Multiplication

The complex matrix-matrix multiplication

$$\begin{aligned}\mathbf{C} = \mathbf{C}_{\mathrm{R}} + \mathrm{i}\mathbf{C}_{\mathrm{I}} &= \mathbf{A}\mathbf{B} = (\mathbf{A}_{\mathrm{R}} + \mathrm{i}\mathbf{A}_{\mathrm{I}})(\mathbf{B}_{\mathrm{R}} + \mathrm{i}\mathbf{B}_{\mathrm{I}})\\ \mathbf{C}_{\mathrm{R}} &= \mathbf{A}_{\mathrm{R}}\mathbf{B}_{\mathrm{R}} - \mathbf{A}_{\mathrm{I}}\mathbf{B}_{\mathrm{I}}\\ \mathbf{C}_{\mathrm{I}} &= \mathbf{A}_{\mathrm{R}}\mathbf{B}_{\mathrm{I}} + \mathbf{A}_{\mathrm{I}}\mathbf{B}_{\mathrm{R}},\end{aligned} \tag{9.5}$$

could be calculated as

$$\begin{aligned}\mathbf{T}_1 &= \mathbf{A}_{\mathrm{R}}\mathbf{B}_{\mathrm{R}},\\ \mathbf{T}_2 &= \mathbf{A}_{\mathrm{I}}\mathbf{B}_{\mathrm{I}},\\ \mathbf{C}_{\mathrm{R}} &= \mathbf{T}_1 - \mathbf{T}_2,\\ \mathbf{C}_{\mathrm{I}} &= (\mathbf{A}_{\mathrm{R}} + \mathbf{A}_{\mathrm{I}})(\mathbf{B}_{\mathrm{R}} + \mathbf{B}_{\mathrm{I}}) - \mathbf{T}_1 - \mathbf{T}_2,\end{aligned} \tag{9.6}$$

in the 3M method[19], which uses only three multiplications (rather than four) and five additions or subtractions (rather than two). There is a gain in speed since matrix-matrix multiplication is more expensive than additions or subtractions, in the cost of two temporary real matrices $\mathbf{T}_1$ and $\mathbf{T}_2$ being created during multiplication. In Fig. 9.1, we have shown the speed-up of benchmark calculations using the 3M Method.

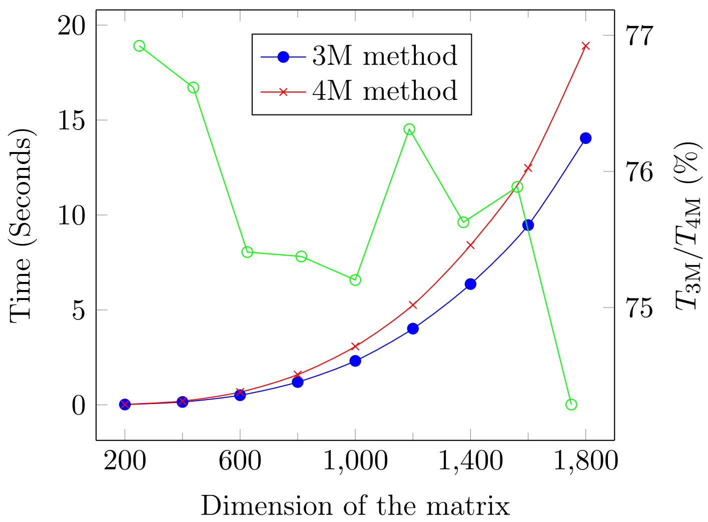

Figure 9.1: Benchmark calculations of the 3M Method performed on Stallo (https://www.notur.no/hardware/stallo), using Hermitian matrix with elements as random numbers.

However, the numerical stability of the 3M Method needs to be taken care, especially, for instance, if the real part is much larger than the imaginary part[20]

$$z = x + \mathrm{i}y = \left(\theta + \frac{\mathrm{i}}{\theta}\right)^2 = \theta^2 - \frac{1}{\theta^2} + 2\mathrm{i}, \tag{9.7}$$

$$y = \left(\theta + \frac{1}{\theta}\right)\left(\theta + \frac{1}{\theta}\right) - \theta^2 - \frac{1}{\theta^2}. \tag{9.8}$$

9.2.2 Cholesky Decomposition

9.2.3 Eigenvalue Solver

$$\left(\mathbf{A}_{\mathrm{R}} + \mathrm{i}\mathbf{A}_{\mathrm{I}}\right)\left(\mathbf{X}_{\mathrm{R}} + \mathrm{i}\mathbf{X}_{\mathrm{I}}\right) = \left(\mathbf{\Lambda}_{\mathrm{R}} + \mathrm{i}\mathbf{\Lambda}_{\mathrm{I}}\right)\left(\mathbf{X}_{\mathrm{R}} + \mathrm{i}\mathbf{X}_{\mathrm{I}}\right), \tag{9.9}$$

is equivalent to

$$\begin{bmatrix}\mathbf{A}_{\mathrm{R}} & -\mathbf{A}_{\mathrm{I}}\\ \mathbf{A}_{\mathrm{I}} & \mathbf{A}_{\mathrm{R}}\end{bmatrix}\begin{bmatrix}\mathbf{X}_{\mathrm{R}}\\ \mathbf{X}_{\mathrm{I}}\end{bmatrix} = \begin{bmatrix}\mathbf{\Lambda}_{\mathrm{R}} & -\mathbf{\Lambda}_{\mathrm{I}}\\ \mathbf{\Lambda}_{\mathrm{I}} & \mathbf{\Lambda}_{\mathrm{R}}\end{bmatrix}\begin{bmatrix}\mathbf{X}_{\mathrm{R}}\\ \mathbf{X}_{\mathrm{I}}\end{bmatrix}, \tag{9.10}$$

Suppose $\mathbf{\Lambda}$ is real, we have

$$\begin{bmatrix}\mathbf{A}_{\mathrm{R}} & -\mathbf{A}_{\mathrm{I}}\\ \mathbf{A}_{\mathrm{I}} & \mathbf{A}_{\mathrm{R}}\end{bmatrix}\begin{bmatrix}\mathbf{X}_{\mathrm{R}}\\ \mathbf{X}_{\mathrm{I}}\end{bmatrix} = \begin{bmatrix}\mathbf{\Lambda}_{\mathrm{R}} & \mathbf{0}\\ \mathbf{0} & \mathbf{\Lambda}_{\mathrm{R}}\end{bmatrix}\begin{bmatrix}\mathbf{X}_{\mathrm{R}}\\ \mathbf{X}_{\mathrm{I}}\end{bmatrix}. \tag{9.11}$$

9.2.4 Linear Response Solver

$$\left(\mathbf{A}_{\mathrm{R}} + \mathrm{i}\mathbf{A}_{\mathrm{I}}\right)\left(\mathbf{X}_{\mathrm{R}} + \mathrm{i}\mathbf{X}_{\mathrm{I}}\right) = \left(\mathbf{B}_{\mathrm{R}} + \mathrm{i}\mathbf{B}_{\mathrm{I}}\right) \tag{9.12}$$

is equivalent to

$$\begin{bmatrix}\mathbf{A}_{\mathrm{R}} & -\mathbf{A}_{\mathrm{I}}\\ \mathbf{A}_{\mathrm{I}} & \mathbf{A}_{\mathrm{R}}\end{bmatrix}\begin{bmatrix}\mathbf{X}_{\mathrm{R}}\\ \mathbf{X}_{\mathrm{I}}\end{bmatrix} = \begin{bmatrix}\mathbf{B}_{\mathrm{R}}\\ \mathbf{B}_{\mathrm{I}}\end{bmatrix}, \tag{9.13}$$

9.2.5 Determinant

Let

$$\mathbf{B} = \begin{bmatrix}\mathbf{A}_{\mathrm{R}} & \mathrm{i}\mathbf{A}_{\mathrm{I}}\\ \mathrm{i}\mathbf{A}_{\mathrm{I}} & \mathbf{A}_{\mathrm{R}}\end{bmatrix}, \tag{9.14}$$

we have

$$\det\mathbf{B} = \det\begin{bmatrix}\mathbf{A}_{\mathrm{R}} + \mathrm{i}\mathbf{A}_{\mathrm{I}} & \mathrm{i}\mathbf{A}_{\mathrm{I}}\\ \mathbf{A}_{\mathrm{R}} + \mathrm{i}\mathbf{A}_{\mathrm{I}} & \mathbf{A}_{\mathrm{R}}\end{bmatrix} = \det\begin{bmatrix}\mathbf{I} & \mathrm{i}\mathbf{A}_{\mathrm{I}}\\ \mathbf{I} & \mathbf{A}_{\mathrm{R}}\end{bmatrix}\det\begin{bmatrix}\mathbf{A}_{\mathrm{R}} + \mathrm{i}\mathbf{A}_{\mathrm{I}} & \mathbf{0}\\ \mathbf{0} & \mathbf{I}\end{bmatrix}, \tag{9.15}$$

and

$$\det\begin{bmatrix}\mathbf{I} & \mathrm{i}\mathbf{A}_{\mathrm{I}}\\ \mathbf{I} & \mathbf{A}_{\mathrm{R}}\end{bmatrix} = \det\begin{bmatrix}\mathbf{I} & \mathrm{i}\mathbf{A}_{\mathrm{I}}\\ \mathbf{0} & \mathbf{A}_{\mathrm{R}} - \mathrm{i}\mathbf{A}_{\mathrm{I}}\end{bmatrix}, \tag{9.16}$$

hence

$$\det\mathbf{B} = \det(\mathbf{A}_{\mathrm{R}} - \mathrm{i}\mathbf{A}_{\mathrm{I}})\det(\mathbf{A}_{\mathrm{R}} + \mathrm{i}\mathbf{A}_{\mathrm{I}}) = \det\mathbf{A}\det\mathbf{A}^{\dagger} = |\det\mathbf{A}|^2. \tag{9.17}$$

Moreover, if we let

$$\mathbf{D} = \begin{bmatrix}\mathbf{I} & \mathbf{0}\\ \mathbf{0} & \mathrm{i}\mathbf{I}\end{bmatrix}, \tag{9.18}$$

we have

$$\mathbf{D}^{-1}\mathbf{B}\mathbf{D} = \begin{bmatrix}\mathbf{A}_{\mathrm{R}} & -\mathbf{A}_{\mathrm{I}}\\ \mathbf{A}_{\mathrm{I}} & \mathbf{A}_{\mathrm{R}}\end{bmatrix}, \tag{9.19}$$

and

$$\det(\mathbf{D}^{-1}\mathbf{B}\mathbf{D}) = \det\mathbf{B} = |\det\mathbf{A}|^2. \tag{9.20}$$

9.3 The Fortran 90 Adapter

To access the Fortran type in the C library QcMatrix is not trivial. We take the strategy in Ref. [21], by defining a Fortran type with only one pointer member poiting to the Fortran matrix type `LANG_F_MATRIX`. As illustrated in Fig. 9.2, the address of this defined Fortran type `matrix_ptr_t` can be saved in an integer array `f90_imat[SIZEOF_F_TYPE_P]` and accessed by the C function `RealMatWrite`. The convention between the integer array `f90_imat[SIZEOF_F_TYPE_P]` and the Fortran type `matrix_ptr_t` can be done by the Fortran function `transfer`.

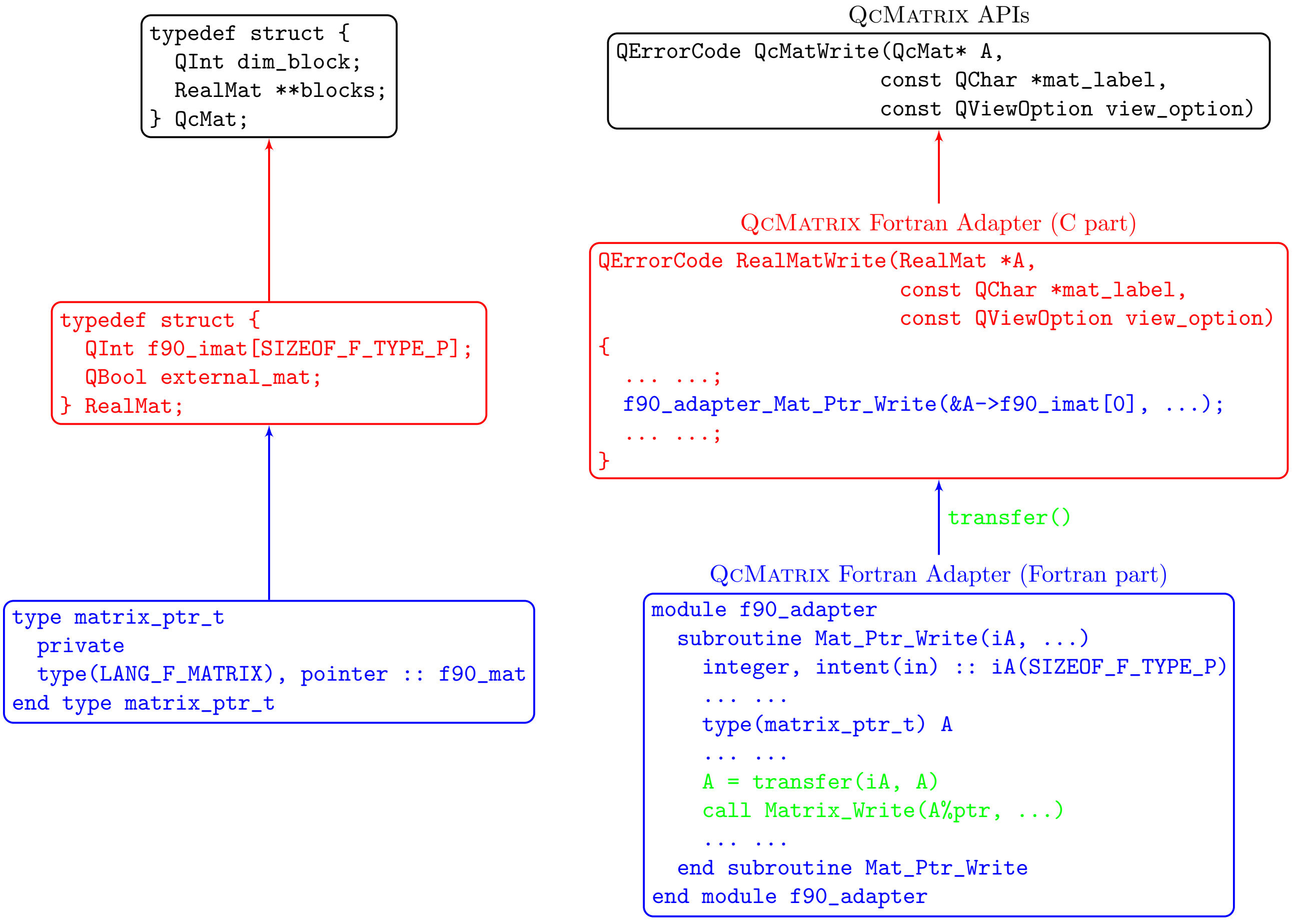

Figure 9.2: Fortran 90 adapter for the external real matrix.

Notice that the Fortran adapter subroutine is implemented in a module `f90_adapter`, the name mangling is taken care by CMake (see files `CMakeLists.txt` and `cmake/f90_adapter.cmake`):

```
CMakeLists.txt:
IF(QCMATRIX_FC_MANGLING)
    INCLUDE(FortranCInterface)
    FortranCInterface_VERIFY()
    #FortranCInterface_VERIFY(CXX)
    FortranCInterface_HEADER(f90_mangling.h
                             MACRO_NAMESPACE "FC_"
                             SYMBOLS ${FC_MANGLING_SUB})
```

```
ENDIF()

cmake/f90_adapter.cmake:
IF(QCMATRIX_ENABLE_VIEW)
    SET(FC_MANGLING_SUB ${FC_MANGLING_SUB} f90_adapter:Mat_Ptr_Write)
ENDIF()
```

In the C code of the Fortran adapter, we have:

```
/* uses CMake generated header file with auto-detected mangling */
#include "f90_mangling.h"

/* declaration of Fortran 90 subroutines */
... ...
#if defined(QCMATRIX_ENABLE_VIEW)
extern QVoid f90_adapter_Mat_Ptr_Write(QInt *iA,
                                       const QInt *len_mat_label,
                                       const QChar *mat_label,
                                       const QInt *view_option);
#endif
```

9.4 The Fortran 2003 Adapter

In Fortran 2003, we can use the `iso_c_binding` to facilitate the access of Fortran type from the C code. As shown in Fig. 9.3, the Fortran matrix type `type(LANG_F_MATRIX)` pointer `f_A` can be converted from the C pointer `type(C_PTR) A` using the function `c_f_pointer`. So that QcMatrix can access the Fortran matrix type. Moreover, the name mangling is also taken care by `iso_c_binding`.

9.5 The Fortran 90 API

In contrast to the Fortran 90 adapter, the problem of API is how to access the `struct QcMat` in Fortran 90 code. In QcMatrix, we resort to the similar strategy of the PETSc library (http://www.mcs.anl.gov/petsc/). As shown in Fig. 9.4, inside the `type QcMat`, there is only an integer (`kind=SIZEOF_VOID_P` for offset) for saving the address of C `struct`.

To be consistent with the `type QcMat` defined in the Fortran API, the `struct QcMat_ptr` only has one member as the `QcMat` pointer, so that `type QcMat` can be directly converted to `struct QcMat_ptr`.

Again, the name mangling is taken care by CMake (file `cmake/f90_api.cmake`):

```
IF(QCMATRIX_ENABLE_VIEW)
    SET(FC_MANGLING_SUB ${FC_MANGLING_SUB} f90_api_QcMatWrite)
ENDIF()
```

9.6 The Fortran 2003 API

The use of `iso_c_binding` also facilitates the Fortran 2003 API of QcMatrix. As shown in Fig. 9.5, the C pointer `type(C_PTR)` can be directly used in the Fortran code and passed to the C part, and converted to `struct QcMat`.

The name mangling can be solved by the declaration of the C function in the `interface` of the Fortran part and using the `iso_c_binding`:

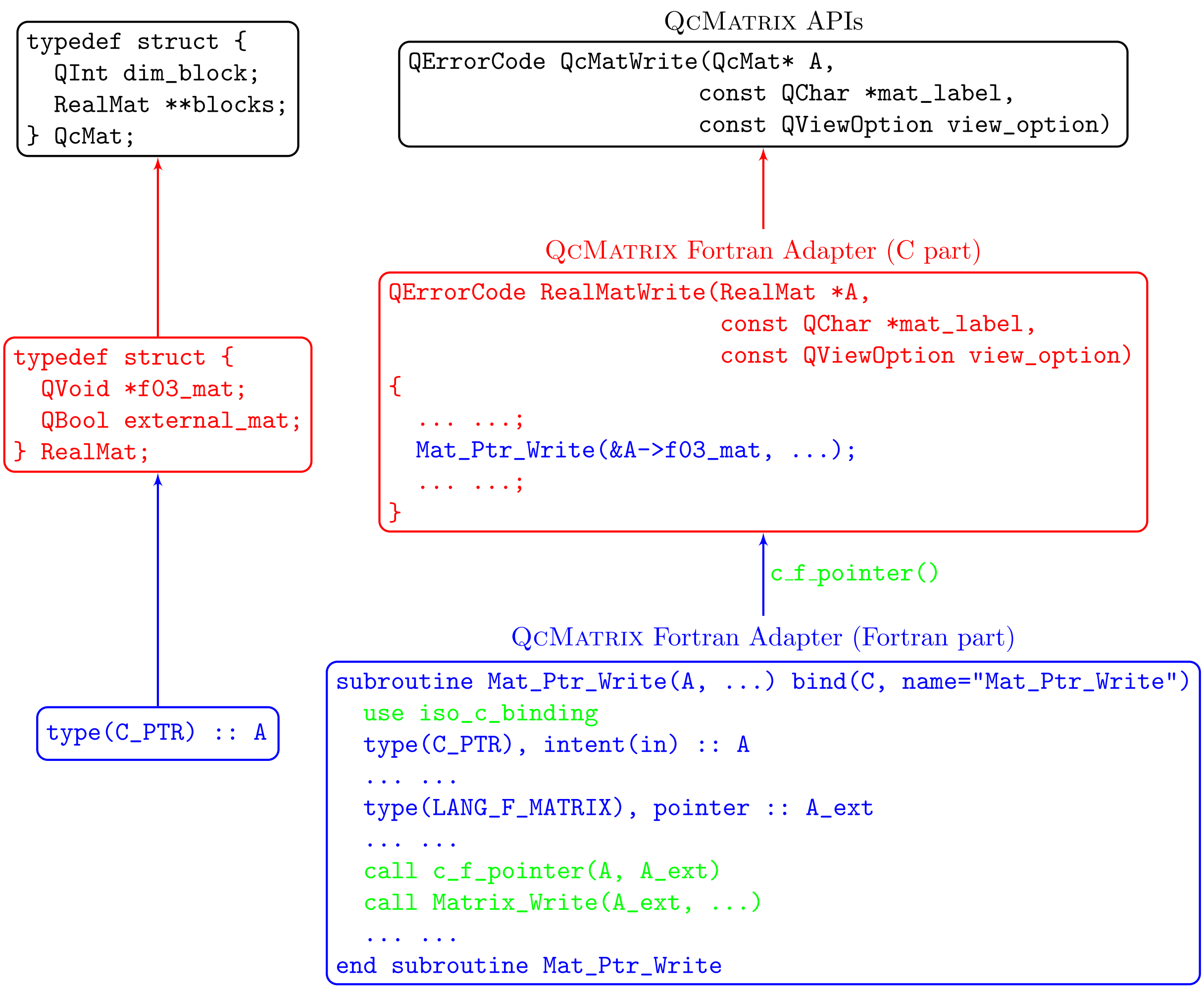

Figure 9.3: Fortran 2003 adapter for the external real matrix.

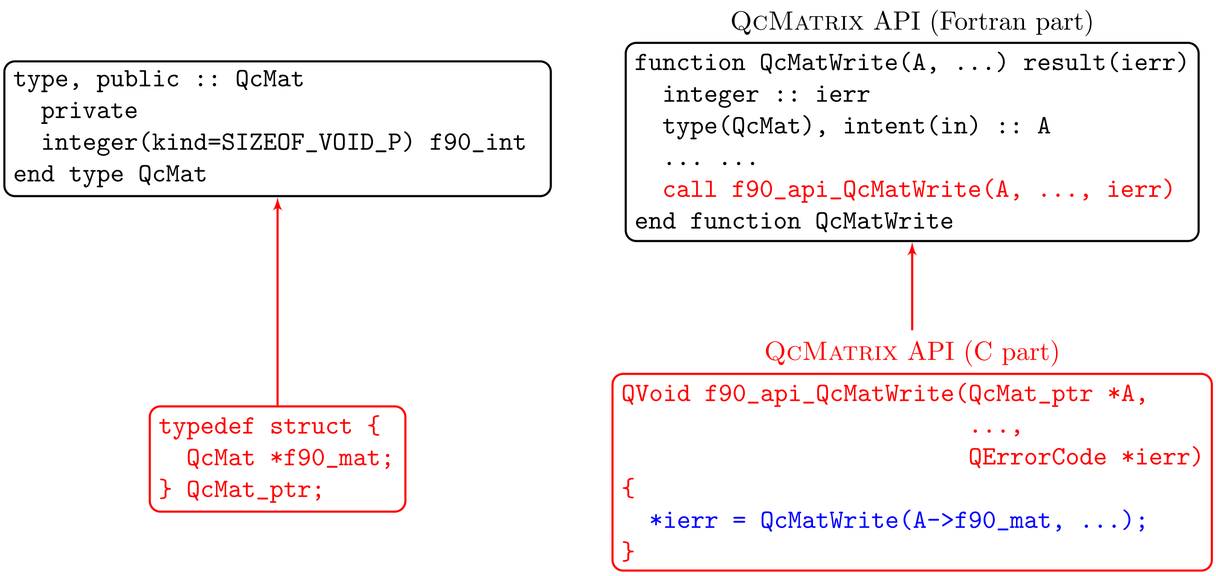

Figure 9.4: Type convention in the QcMatrixFortran 90 API.

```
type, public :: QcMat
  private
  type(C_PTR) :: c_mat = C_NULL_PTR
end type QcMat
```

QcMatrix API (Fortran part)

```
function QcMatWrite(A, ...) result(ierr)
  integer :: ierr
  type(QcMat), intent(in) :: A
  ... ...
  ierr = f03_api_QcMatWrite(A%c_mat, ...)
end function QcMatWrite
```

QcMatrix API (C part)

```
QErrorCode f03_api_QcMatWrite(QVoid **A,
                              ...)
{
  QErrorCode ierr;
  QcMat *c_A;
  c_A = (QcMat *)(*A);
  ierr = QcMatWrite(c_A, ...);
  return ierr;
}
```

Figure 9.5: Type convention in the QcMatrixFortran 2003 API.

```
    interface
#if defined(QCMATRIX_ENABLE_VIEW)
        integer(C_INT) function f03_api_QcMatWrite(A, mat_label, view_option) &
            bind(C, name="f03_api_QcMatWrite")
            use iso_c_binding
            type(C_PTR), intent(in) :: A
            character(C_CHAR), intent(in) :: mat_label(*)
            integer(C_INT), value, intent(in) :: view_option
        end function f03_api_QcMatWrite
#endif
    end interface
```

9.7 Procedure of `QcMatSetExternalMat`

The function `QcMatSetExternalMat` involves conventions of different matrix types in the API and adapter parts, which are illustrated in Fig. 9.6 and Fig. 9.7 respectively for the Fortran 90 and 2003.

It should be noticed that the context of the external matrix `A_ext` will not be destroyed by `QcMatDestroy`. So that the `QBool external_mat` has been introduced to indicate if the `QInt f90_imag[SIZEOF_F_TYPE_P]` or `QVoid *f03_mat` points to an external matrix.

9.8 Procedure of `QcMatGetExternalMat`

The function `QcMatGetExternalMat` also involves the conventions of different matrix types in the API and adapter parts, which are illustrated in Fig. 9.8 and Fig. 9.9 respectively for the Fortran 90 and 2003.

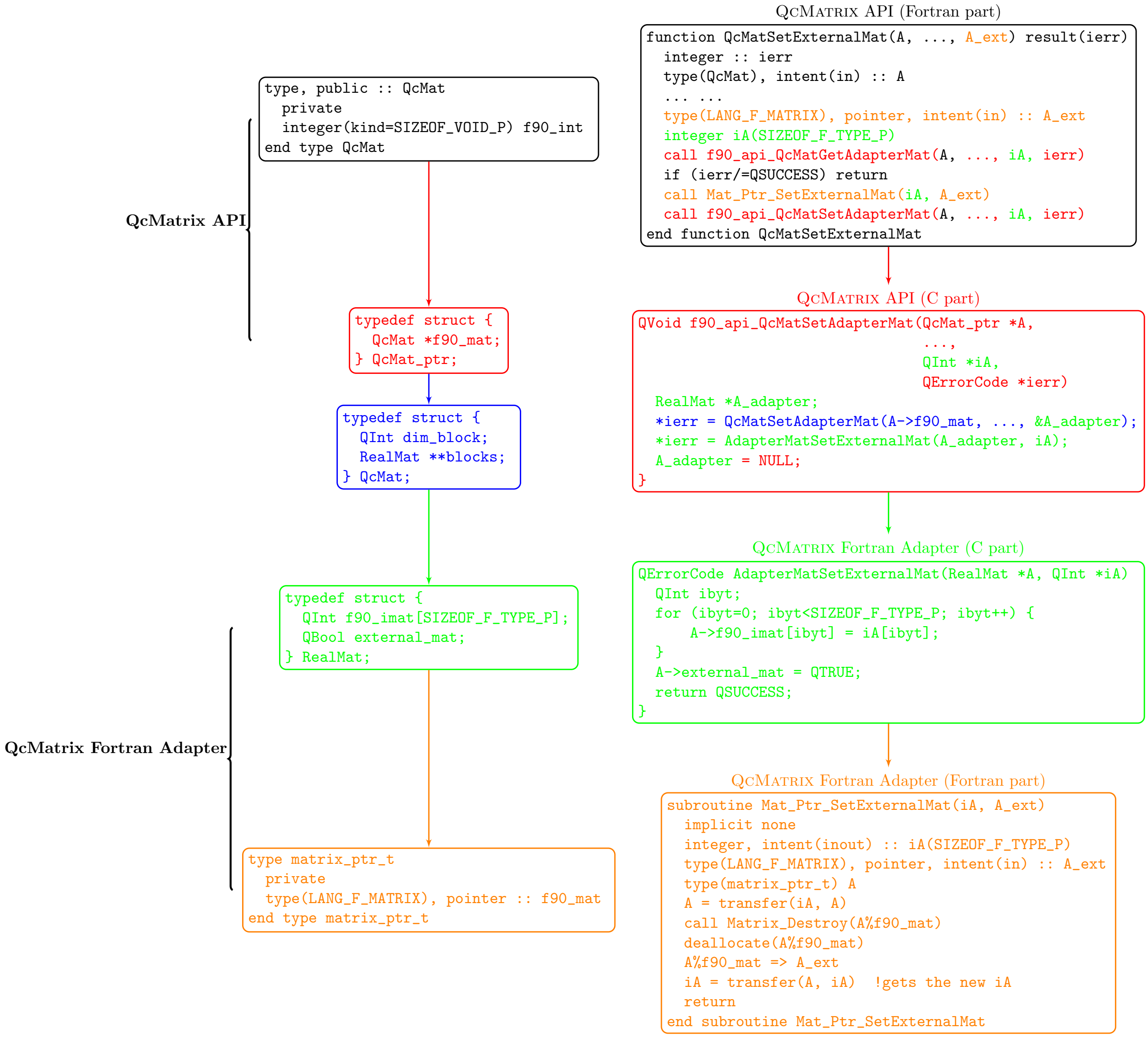

Figure 9.6: Procedure of `QcMatSetExternalMat` (Fortran 90).

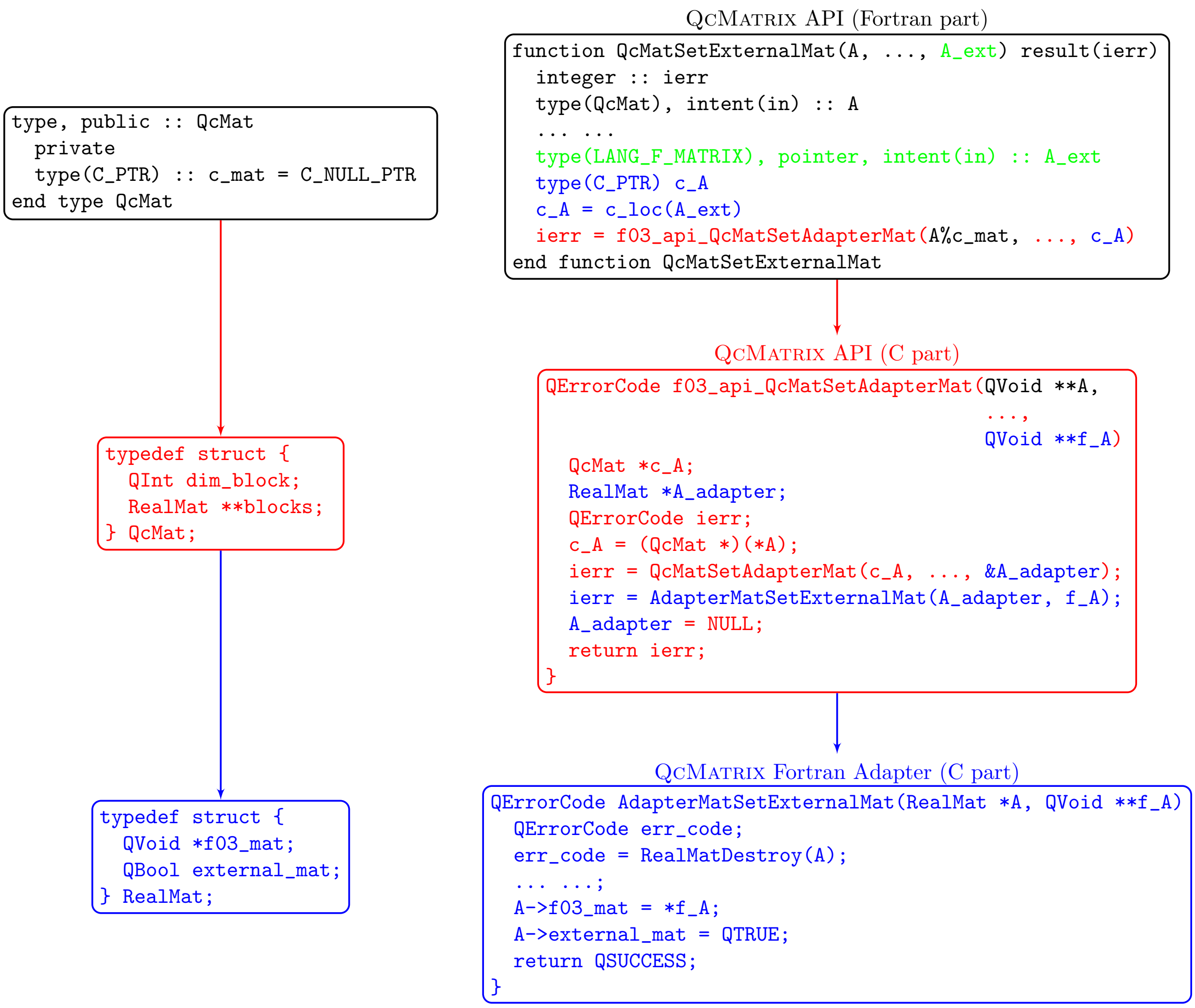

Figure 9.7: Procedure of `QcMatSetExternalMat` (Fortran 2003).

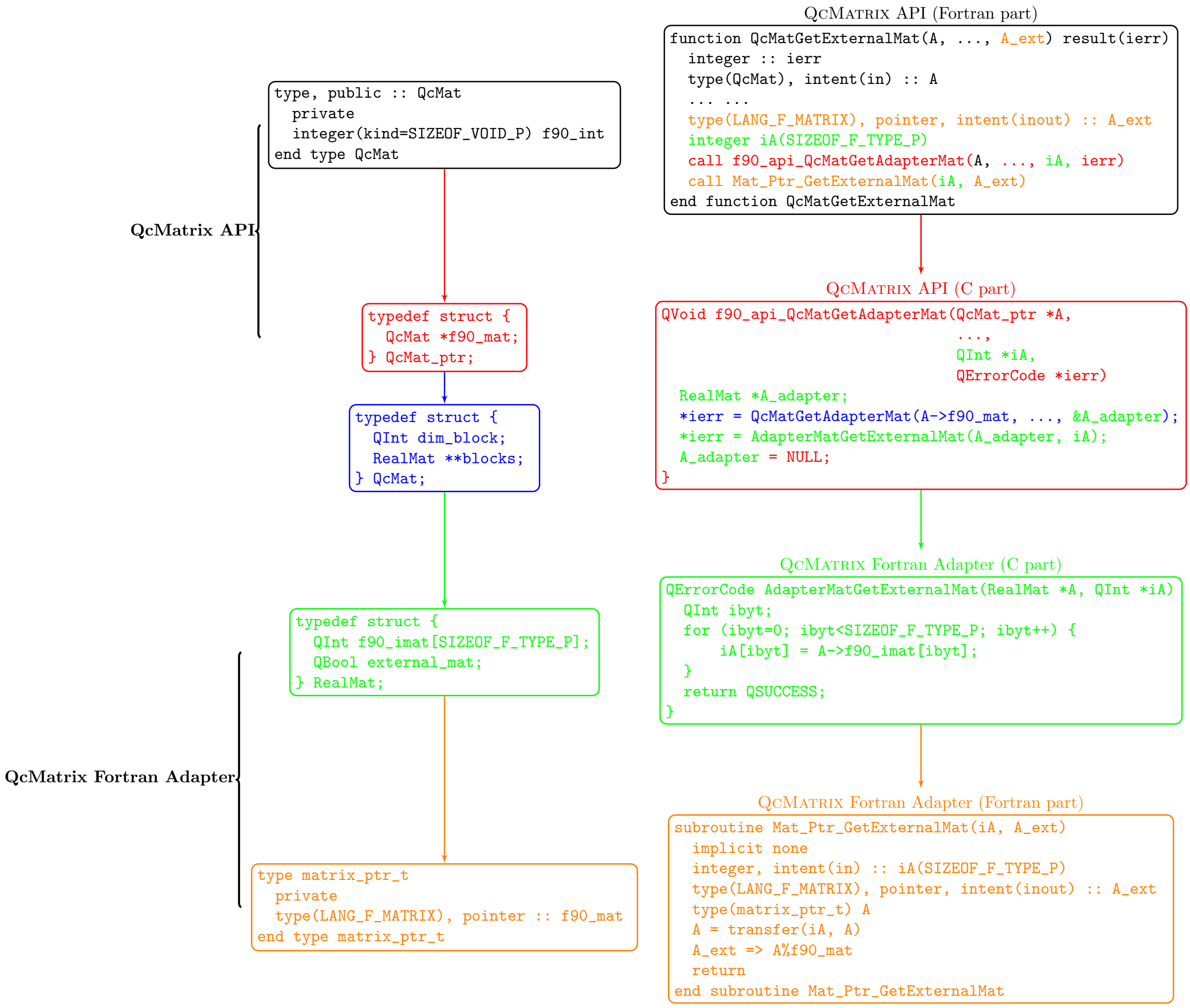

Figure 9.8: Procedure of `QcMatGetExternalMat` (Fortran 90).

QcMatrix API (Fortran part)

```
function QcMatGetExternalMat(A, ..., A_ext) result(ierr)
  integer :: ierr
  type(QcMat), intent(in) :: A
  ... ...
  type(LANG_F_MATRIX), pointer, intent(inout) :: A_ext
  type(C_PTR) c_A
  ierr = f03_api_QcMatGetAdapterMat(A%c_mat, ..., c_A)
  call c_f_pointer(c_A, A_ext)
end function QcMatGetExternalMat
```

```
type, public :: QcMat
  private
  type(C_PTR) :: c_mat = C_NULL_PTR
end type QcMat
```

QcMatrix API (C part)

```
QErrorCode f03_api_QcMatGetAdapterMat(QVoid **A,
                                      ...,
                                      QVoid **f_A)
  QcMat *c_A;
  RealMat *A_adapter;
  QErrorCode ierr;
  c_A = (QcMat *)(*A);
  ierr = QcMatGetAdapterMat(c_A, ..., &A_adapter);
  ierr = AdapterMatGetExternalMat(A_adapter, f_A);
  A_adapter = NULL;
  return ierr;
}
```

```
typedef struct {
  QInt dim_block;
  RealMat **blocks;
} QcMat;
```

QcMatrix Fortran Adapter (C part)

```
QErrorCode AdapterMatGetExternalMat(RealMat *A, QVoid **f_A)
  *f_A = (QVoid *)A->f03_mat;
  return QSUCCESS;
}
```

```
typedef struct {
  QVoid *f03_mat;
  QBool external_mat;
} RealMat;
```

Figure 9.9: Procedure of `QcMatGetExternalMat` (Fortran 2003).

Chapter 10

Files and Directories of QcMatrix

1. AUTHORS: Author information.
2. ChangeLog: Changes made.
3. cmake and CMakeLists.txt: CMake files.
4. COPYING and COPYING.LESSER: The license.
5. doc
   (a) figures: Figures.
   (b) logo: Logos.
   (c) manual.pdf, manual.tex and refs.bib: Manual files.
   (d) tutorial.pdf and tutorial.tex: Tutorial files.
6. examples: Examples for tutorial.
7. include: Header files.
   (a) adapter: Header files for the adapters of external libraries.
   (b) api: Header files of APIs.
   (c) impls: Header files of implemented matrices in QcMatrix.
   (d) lapack: Header files for calling BLAS and LAPACK libraries, used by internal real matrix library and test suite.
   (e) qcmatrix.h: Header file of QcMatrix C APIs.
   (f) README: Some rules for these header files.
   (g) tests: Header files of test suite.
   (h) types: Header files of basic and matrix types used in QcMatrix.
   (i) utilities: Header files of utilities.
8. INSTALL: Installation instruction.
9. maintenance: Maintenance files.
10. MANIFEST.in: Python manifest template for source distribution.
11. NEWS: List of user-visible changes.
12. PKG-INFO: PKG-INFO metadata file.
13. QcMatrix: Python files.
14. qcmatrix.bib: QcMatrix citation.
15. README: A very important file ;-).
16. RULES: Rules for contribution.
17. setup.cfg: Python setup configuration file.
18. setup.py: Python setup script.
19. src
   (a) adapter: Source codes of adapters.

   (b) cmplx_mat: Source codes of complex matrix.
   (c) qcmat: Source codes of square block complex matrix.
      i. c: Source codes of APIs QcMatGetExternalMat and QcMatSetExternalMat.
      ii. f03: Source codes of Fortran APIs (using Fortran 2003 functionalities).
      iii. f90: Source codes of Fortran APIs (using Fortran 90 functionalities).
      iv. tests: Source codes of APIs for tests only.
   (d) real_mat: Source codes of real matrix.
   (e) xray: Source codes of X-ray spectroscopies.
20. tests
   (a) c: Source codes of C test suite.
      i. adapter: A simple external C matrix library (indeed it just QcMatrix source codes themselves, but to be compiled here to mimic the external C matrix library).
         A. clean_c_adapter.sh: Script for cleaning the files created by init_c_adapter.sh.
         B. cmake and CMakeLists.txt: CMake files.
         C. include/impls: Header files.
         D. init_c_adapter.sh: Script for copying files from QcMatrix for this external C matrix library.
         E. README: Please follow this file to compile this external C matrix library.
      ii. api: Source codes of testing QcMatrix C APIs.
   (b) f90: Source codes of Fortran test suite.
      i. adapter: A simple external Fortran matrix library.
         A. clean_f_adapter.sh: Script for cleaning the files created by init_f_adapter.sh.
         B. CMakeLists.txt: CMake file.
         C. init_f_adapter.sh: Script for copying files from QcMatrix for this external Fortran matrix library.
         D. README: Please follow this file to compile this external Fortran matrix library.
         E. src: Source codes of this external Fortran matrix library.
      ii. api: Source codes of testing QcMatrix Fortran APIs.
      iii. test_3M_method.F90: Source code for testing the efficiency of 3M method for complex matrix-matrix multiplication.
      iv. timer.F90: Source code recording the CPU time, needed by test_3M_method.F90.
21. TODO: Todo list.
22. tools: Tools for QcMatrix.

Chapter 11

Limitations of QcMatrix

1. Chapters 5, 6, 7 and 8, Sections 9.1 and 9.2 to be finished.
2. Only the real matrix adapter has been tested.
3. Reading ASCII format file is not tested.
4. C++ and Python adapters and APIs to be implemented.
5. The parallelization is not implemented in QcMatrix, but relies on the external matrix libraries.
6. Internal C real matrix may not be efficient, and is not parallelized.
7. Only one type of adapter can be compiled (C, C++ or Fortran) at once.
8. We do not guarantee that QcMatrix will always check the validity of all input arguments, users should be more careful on sending correct arguments to QcMatrix.
9. There will be some compiler warnings if HDF5 library is used.
10. There is also a bug regarding the use of HDF5 library: if QcMatrix does not use HDF5 library, but the external matrix library does, then it may have error when linking the QcMatrix test suite as executables.

Bibliography

[1] Seymour Lipschutz and Marc Lipson. *Schaum's Easy Outline of Linear Algebra*. McGraw-Hill, 2002.

[2] U. Gelius. *Proceedings of the International Conference on Electron Spectroscopy (edited D.A. Shirley)*. North-Holland, Amsterdam, 1972.

[3] U. Gelius. Recent progress in ESCA studies of gases. *J. Electron Spectrosc. Relat. Phenom.*, 5(1):985–1057, 1974.

[4] J. J. Yeh and I. Lindau. Atomic subshell photoionization cross sections and asymmetry parameters: $1 \leq Z \leq 103$. *Atom. Data Nucl. Data Tables*, 32(1):1–155, 1985.

[5] H. A. Kramers and W. Heisenberg. The scattering of radiation by atoms. *Z. Phys.*, 31:681–708, 1925.

[6] P. A. M. Dirac. The Quantum Theory of Dispersion. *Proc. Roy. Soc. London, Ser. A*, 114:710–728, May 1927.

[7] Yi Luo, H Ågren, and F Gel'mukhanov. Symmetry assignments of occupied and unoccupied molecular orbitals through spectra of polarized resonance inelastic X-ray scattering. *J. Phys. B*, 27(18):4169–4180, 1994.

[8] Yi Luo, Hans Ågren, Faris Gel'mukhanov, Jinghua Guo, Per Skytt, Nial Wassdahl, and Joseph Nordgren. Symmetry-selective resonant inelastic x-ray scattering of C_{60}. *Phys. Rev. B*, 52(20):14479–14496, Nov 1995.

[9] F. Gel'mukhanov and H. Ågren. Resonant X-ray Raman scattering. *Phys. Rep.*, 312:87–330, 1999.

[10] Barbara Brena, Yi Luo, Mats Nyberg, Stéphane Carniato, Katharina Nilson, Ylvi Alfredsson, John Åhlund, Nils Mårtensson, Hans Siegbahn, and Carla Puglia. Equivalent core-hole time-dependent density functional theory calculations of carbon 1*s* shake-up states of phthalocyanine. *Phys. Rev. B*, 70(19):195214, Nov 2004.

[11] Barbara Brena, Stéphane Carniato, and Yi Luo. Functional and basis set dependence of K-edge shake-up spectra of molecules. *J. Chem. Phys.*, 122(18):184316, 2005.

[12] R. L. Martin and D. A. Shirley. Theory of core-level photoemission correlation state spectra. *J. Chem. Phys.*, 64(9):3685–3689, 1976.

[13] Jun Jiang, Mathias Kula, and Yi Luo. A generalized quantum chemical approach for elastic and inelastic electron transports in molecular electronics devices. *J. Chem. Phys.*, 124(3):034708, 2006.

[14] Chuan-Kui Wang, Ying Fu, and Yi Luo. A quantum chemistry approach for current-voltage characterization of molecular junctions. *Phys. Chem. Chem. Phys.*, 3:5017–5023, 2001.

[15] V. Mujica, M. Kemp, and M. A. Ratner. Electron conduction in molecular wires. I. A scattering formalism. *J. Chem. Phys.*, 101(8):6849–6855, 1994.

[16] Ai-Min Guo and Qing-Feng Sun. Spin-Selective Transport of Electrons in DNA Double Helix. *Phys. Rev. Lett.*, 108:218102, May 2012.

[17] Anatoliy Volkov, Harry F. King, Philip Coppens, and Louis J. Farrugia. On the calculation of the electrostatic potential, electric field and electric field gradient from the aspherical pseudoatom model. *Acta Cryst. A*, 62(5):400–408, Sep 2006.

[18] Shiro Koseki, Michael W. Schmidt, and Mark S. Gordon. MCSCF/6-31G(d,p) Calculations of One-Electron Spin-Orbit Coupllng Constants in Dlatomic Molecules. *J. Phys. Chem.*, 96(26):10768–10772, 1992.

[19] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms.* Society for Industrial and Applied Mathematics, 2nd edition, 2002.

[20] N. Higham. Stability of a Method for Multiplying Complex Matrices with Three Real Matrix Multiplications. *SIAM J. Matrix Anal. Appl.*, 13(3):681–687, 1992.

[21] Alexander Pletzer, Douglas McCune, Stefan Muszala, Srinath Vadlamani, and Scott Kruger. Exposing Fortran Derived Types to C and Other Languages. *Comput. Sci. Eng.*, 10:86–92, 2008.

Index