```
/*****************************************************************************************

        Diffusion Along a Circle
        (Canonical Ensemble)
        Eigensystem Approach

        Written by:  Christian Bracher
        Version   :  0.28
        Date      :  November 26, 2003

*****************************************************************************************/
```

DESCRIPTION OF THE FUNCTIONS INVOKED BY DIFF

double Average(string Sequence, double *ProbabilityVector)

Returns the average probability of patterns that match the indefinite pattern encoded by the string object Sequence (a string of length p consisting of the symbols "o", "." and "x"). *ProbabilityVector is the pointer to an array which contains the probability of each atomic arrangement, which must be indexed as follows:

```
Probability[#Primitive | nu] = ProbabilityVector[nu * PrimNum + #Primitive],
```

i.e., the first PrimNum entries encode the probabilities for all primitive patterns, the next set of PrimNum entries contain the probabilities of the same pattern set, rotated clockwise by one site, etc.

string CoefficientString(char CoeffA, char CoeffB, char CoeffC, char CoeffD)

Format an element of the coefficient rate matrix for output as a symbolic string. The integers CoeffA, CoeffB, CoeffC, and CoeffD are the coefficients for jumps of type A, B, C and D, respectively:

```
Rate = CoeffA * RateA + CoeffB * RateB + CoeffC * RateC + CoeffD * RateD
```

This function returns this expression in symbolic form as a formatted string of length 18. Example:

```
CoefficientString(5, 2, 0, 1) = "      5A+2B+D     ";
```

string ConvertIntegerToString(int Number)

Convert an integer into a string containing the numerical value. This auxiliary function is used in the output of the coefficient rate matrix.

void EigenSort(double *Array, long *Pointer)

Sort a numerical array of type double in ascending order. This function is used in DIFF to sort the eigenvalues in each momentum subspace. *Array is a pointer to the start of the numerical array, assumed to be of length PrimNum, and *Pointer is an array of the same length of type long. Instead of physically sorting the numerical array, EigenSort rather delivers pointers to these elements, i.e., Pointer[k] will contain the index of the k.th smallest entry in Array[].

inline long Element(long Row, long Column)

Calculate a pointer to the matrix element (Row, Column) according to the rule:

```
Pointer = Row * MatrixDim + Column;
```

In the current version of DIFF, MatrixDim and PrimNum are synonyms. This auxiliary function is called by various functions in this set.

double EqAverage(string Sequence)

Returns the equilibrium average probability of patterns that match the indefinite pattern encoded by the string object Sequence (a string of length p consisting of the symbols "o", "." and "x").

void ErrorMessage(string Message)

Sends the sequence "ERROR: ", followed by the argument Message, to the screen output and subsequently terminates program execution.

string ExtractString(string Key)

Extracts a string argument from a command line option by removing the switch part. Example:

ExtractString("/l=c:\test\5in11") = "c:\test\5in11"

double ExtractValue(string Key)

Extracts a numerical argument of type double from a command line option. Example:

ExtractValue("/A=0.997") = .997

void HouseholderHerm(complex<double> *Matrix, double *Diag, complex<double> *SupDiag)

Perform Householder tridiagonalization on the Hermitian matrix A pointed to by *Matrix. Matrix must be an array of type complex<double> of size MatrixDim * MatrixDim (MatrixDim, a global variable, indicates the size of the matrix). The indices of an element and its position in the array are related via the Element(Row, Column) function (see above). On output, Diag, an array of type double with size MatrixDim, contains the diagonal elements of the tridiagonal matrix T, and SupDiag, an array of type complex<double> and the same size, the superdiagonal elements, where the index indicates the column of the element. If the global boolean variable EigenvaluesOnly is set to false, the initial matrix Matrix will be replaced by the right-hand side unitary transformation matrix Q that connects T with the original matrix: A = Q^+TQ. Otherwise, the contents of Matrix are undefined.

void HouseholderSymm(double *Matrix, double *Diag, double *ExDiag)

Perform Householder tridiagonalization on the symmetric matrix A pointed to by *Matrix. Matrix must be an array of type double of size MatrixDim * MatrixDim (MatrixDim, a global variable, indicates the size of the matrix). The indices of an element and its position in the array are related via the Element(Row, Column) function (see above). On output, Diag, an array of type double with size MatrixDim, contains the diagonal elements of the tridiagonal matrix T, and ExDiag, an array of same type and size, the offdiagonal elements, where the index indicates the greater value of the row or column of the element - e.g., T[0,1] = T[1,0] is stored in ExDiag[1]. If the global boolean variable EigenvaluesOnly is set to false, the initial matrix Matrix will be replaced by the right-hand side orthogonal transformation matrix Q that connects T with the original matrix: A = Q^T T Q. Otherwise, the contents of Matrix are undefined.

bool IsPrime(long Test)

A simple primality test. The function returns true if Test is a prime number.

long ModuloDiv(long Numerator, long Denominator)

Perform divison modulo p, where the global variable p (number of sites) is prime. Denominator must not be zero (modulo p). The function returns the unique solution r (modulo p) to the problem:

(r * Denominator) mod p = Numerator.

string NumCoefficient(int Value)

An auxiliary function used by the function CoefficientString(CoeffA, CoeffB, CoeffC, CoeffD) (see there). Value must be between -99 and +99, and the function returns a string representing this number, unless Value is zero, where an empty string will be returned.

long PatternPosition(Pattern Primitive)

Locate a given primitive pattern in the global array *Primitive. Primitive patterns are stored in this array in ascending order of their binary code, and this function returns the index of the argument pattern.

double ProjectAnti(double *Vector, double* Proj)

Project an eigenvector onto the antisymmetric subspace. *Vector is a pointer to a field of type double with size PrimNum (a global variable indicating the number of primitive patterns) representing a real eigenvector of the zero momentum subspace. On output, the array *Proj contains its projection onto the subspace of vectors antisymmetric under mirror reflections. The function returns the square norm of this vector.

double ProjectSymm(double *Vector, double* Proj)

Project an eigenvector onto the symmetric subspace. *Vector is a pointer to a field of type double with size PrimNum (a global variable indicating the number of primitive patterns) representing a real eigenvector of the zero momentum subspace. On output, the array *Proj contains its projection onto the subspace of vectors invariant under mirror reflections. The function returns the square norm of this vector.

void QRHerm(complex<double> *Matrix, double *Diag, complex<double> *SupDiag)

Diagonalize a hermitian tridiagonal matrix using a QR algorithm with explicit shifts. The tridiagonal matrix T (of dimension MatrixDim, a global variable) is supplied by the arrays Diag (the real diagonal elements), and SupDiag (the superdiagonal elements of type complex<double>, starting with SupDiag[1]). *Matrix is a complex array of size MatrixDim * MatrixDim used to perform unitary transformations. On output, Diag contains the eigenvalues of the tridiagonal matrix (the diagonalized matrix D). No useful information is contained in SupDiag. If the global variable EigenvaluesOnly is set to true, the field Matrix is not used. Otherwise, the unitary transformation is applied to the matrix U stored in *Matrix. Formally, if Q^+DQ = T, then U is transformed to QU. Applications:

(i) If the identity matrix is used as input for Matrix, then the unitary transform Q will be returned.

(ii) Using the procedure HouseholderHerm(), an arbitrary Hermitian matrix A may be diagonalized by the calling sequence:

```
[prepare Matrix]
HouseholderHerm(&Matrix, &Diag, &SupDiag)  // Step 1
QRHerm(&Matrix, &Diag, &SupDiag)           // Step 2
```

After Step 1, Diag and SupDiag contain the tridiagonal matrix T, and Matrix contains the unitary transform P connecting T and A: P^+TP = A. In Step 2, T is diagonalized using the QR procedure: Q^+DQ = T, and Matrix is replaced by QP. Since (QP)^+D(QP) = A, D will consist of the eigenvalues of A, and Matrix will contain the unitary transformation matrix connecting D and A, i.e., the rows of Matrix are the complex conjugates of the eigenvectors of A, normalized to unit length.

void QRSymm(double *Matrix, double *Diag, double *ExDiag)

Diagonalize a symmetric tridiagonal matrix using a QR algorithm with explicit shifts. The tridiagonal matrix T (of dimension MatrixDim, a global variable) is supplied by the arrays Diag (the real diagonal elements), and SupDiag (the extradiagonal elements, starting with SupDiag[1]). *Matrix is an array of size MatrixDim * MatrixDim used to perform orthogonal transformations. On output, Diag contains the eigenvalues of the tridiagonal matrix (the diagonalized matrix D). No useful information is contained in ExDiag. If the global variable EigenvaluesOnly is set to true, the field Matrix is not used. Otherwise, the orthogonal transformation is applied to the matrix U stored in *Matrix. Formally, if Q^T D Q = T, then U is transformed to QU. Applications:

(i) If the identity matrix is used as input for Matrix, then the orthogonal transform Q will be returned.

(ii) Using the procedure HouseholderSymm(), an arbitrary symmetric matrix A may be diagonalized by the calling sequence:

```
[prepare Matrix]
HouseholderSymm(&Matrix, &Diag, &ExDiag)  // Step 1
QRSymm(&Matrix, &Diag, &ExDiag)           // Step 2
```

After Step 1, Diag and ExDiag contain the tridiagonal matrix T, and Matrix contains the orthogonal transform P connecting T and A: P^T T P = A. In Step 2, T is diagonalized using the QR procedure: Q^T D Q = T, and Matrix is replaced by QP. Since (QP)^T D (QP) = A, D will consist of the eigenvalues of A, and Matrix will contain the orthogonal transformation matrix connecting D and A, i.e., an orthonormal set of eigenvectors of A forms the rows of Matrix.

`void RemoveCommonPhase(complex<double> *Vector)`

Prepare a complex eigenvector (stored in the field *Vector of size PrimNum, a global variable indicating the number of primitive patterns) for compressed storage. The elements of the eigenvector corresponding to mirror symmetric pairs of primitive patterns can be chosen conjugate complex, and those representing palindromic patterns real. This function removes the random common phase factor usually present after QR diagonalization.

`void RemoveCommonSign(double *Vector)`

Select the sign of the real vector *Vector of size PrimNum so that the sum of the elements becomes positive. This function is useful in generating an equilibrium eigenvector with positive elements.

`void TestPattern(string AtomPattern, bool IsIndefinite)`

Test the validity of a string representing an atom pattern. Allowed characters are "o" (for an atom) and "." (for an empty site). If the switch parameter IsIndefinite is set to true, then also the token "x" (site with unspecified content) is permitted. If the function encounters other symbols, an error message is generated and program execution is aborted.

`void UnrecognizedKey(string Key)`

An auxiliary function generating an error message for unknown switches in the command line. Key is a string object containing the illegal option.

`inline void usage(int exit_value=0)`

Shows a help screen giving a short overview of options, and aborts program execution.

`bool VectorIsAntisymmetric(double *Vector)`

Check whether a real eigenvector in the zero momentum subspace is antisymmetric to float accuracy (square norm of symmetric part smaller than 1e-15). Except for the degenerate case occurring when all jump rates are equal, all zero momentum eigenvectors are either symmetric or antisymmetric under mirror reflections. However, numerically evaluated eigenvectors always contain admixtures of the opposite symmetry class, in particular for closely spaced eigenvalues.

`bool VectorIsSymmetric(double *Vector)`

Check whether a real eigenvector in the zero momentum subspace is symmetric to float accuracy (square norm of symmetric part smaller than 1e-15). Except for the degenerate case occurring when all jump rates are equal, all zero momentum eigenvectors are either symmetric or antisymmetric under mirror reflections. However, numerically evaluated eigenvectors always contain admixtures of the opposite symmetry class, in particular for closely spaced eigenvalues.