EasyQR

1.0

Generated by Doxygen 1.8.8

Wed Jan 25 2017 15:19:27

Contents

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Class Documentation

3.1 CSC Struct Reference

Public Attributes

- int cols
- int * col_start
- int * indices
- double * elements

3.1.1 Detailed Description

Matrix type in Compressesed Sparse Column (CSC) format

see `https://en.wikipedia.org/wiki/Sparse_matrix` for details of the data format

3.1.2 Member Data Documentation

3.1.2.1 int* CSC::col_start

indices that point to the beginning of each row in indices

3.1.2.2 int CSC::cols

number of columns

3.1.2.3 double* CSC::elements

list of all values

3.1.2.4 int* CSC::indices

list of all indices

The documentation for this struct was generated from the following file:

- easyQR.c

3.2 CSR Struct Reference

Public Attributes

- int rows
- int * row_start
- int * indices
- double * elements

3.2.1 Detailed Description

Matrix type in Compressesed Sparse Row (CSR) format

see https://en.wikipedia.org/wiki/Sparse_matrix for details of the data format

3.2.2 Member Data Documentation

3.2.2.1 double* CSR::elements

list of all values

3.2.2.2 int* CSR::indices

list of all indices

3.2.2.3 int* CSR::row_start

indices that point to the beginning of each row in indices

3.2.2.4 int CSR::rows

number of rows

The documentation for this struct was generated from the following file:

- easyQR.c

3.3 EIGEN Struct Reference

Public Attributes

- int dim
- double value
- double * vector

3.3.1 Detailed Description

Element of a real eigensystem consisting of an eigenvalue, its corrsponding eigenvector, and its dimension

3.3.2 Member Data Documentation

3.3.2.1 int EIGEN::dim

dimension of the eigenvector

3.3.2.2 double EIGEN::value

(real) eigenvalue

3.3.2.3 double* EIGEN::vector

(real) eigenvector as a list of <dim> doubles

The documentation for this struct was generated from the following file:

- easyQR.c

3.4 MATRIX_ELEMENT Struct Reference

Public Attributes

- int row_index
- int col_index
- double value

3.4.1 Detailed Description

Matrix-element used for storage of the "Matrix-Market"-format The element A_{ij} is represented by a row(i) and column(j) index and the corresponding value (A_{ij})

see `http://math.nist.gov/MatrixMarket/formats.html` for more info

3.4.2 Member Data Documentation

3.4.2.1 int MATRIX_ELEMENT::col_index

column index

3.4.2.2 int MATRIX_ELEMENT::row_index

row index

3.4.2.3 double MATRIX_ELEMENT::value

real value

The documentation for this struct was generated from the following file:

- easyQR.c

3.5 RBNODE Struct Reference

Public Attributes

- struct RBNODE * parent
- struct RBNODE * left
- struct RBNODE * right
- RBcolor color
- void * data

3.5.1 Detailed Description

Red-Black-Node type

see https://en.wikipedia.org/wiki/Red%E2%80%93black_tree

3.5.2 Member Data Documentation

3.5.2.1 RBcolor RBNODE::color

color of the node (either red or black)

3.5.2.2 void* RBNODE::data

pointer to the data represented by the node

3.5.2.3 struct RBNODE* RBNODE::left

pointer to left child

3.5.2.4 struct RBNODE* RBNODE::parent

pointer to parent node, if node is root then it contains NULL

3.5.2.5 struct RBNODE* RBNODE::right

pointer to right child

The documentation for this struct was generated from the following file:

- easyQR.c

Chapter 4

File Documentation

4.1 easyQR.c File Reference

```
#include "easyQR.h"
```

Classes

- struct CSR
- struct CSC
- struct MATRIX_ELEMENT
- struct EIGEN
- struct RBNODE

Typedefs

- typedef struct CSR CSR_matrix
- typedef struct CSC CSC_matrix
- typedef struct MATRIX_ELEMENT matrix_element
- typedef struct EIGEN eigen
- typedef enum RBCOLOR RBcolor
- typedef enum RELATION relation
- typedef struct RBNODE rbNode

Enumerations

- enum RBCOLOR { red, black }
- enum RELATION { LEFTLEFT, LEFTRIGHT, RIGHTLEFT, RIGHTRIGHT }

Functions

- static void RBTree_set_compare (int(∗Compare)(void ∗X1, void ∗X2))

  Sets user-defined compare functione for red-black ordering.
- static void RBTree_set_free (void(∗Free_data)(void ∗X))

  Sets user-defined free function for the red-black tree.
- static void RBTree_set_data_size (size_t data_size)

  Sets the size of a user-defined data element in bytes.

- static rbNode ∗ RBTcreateNode (rbNode ∗parent, void ∗data)

    Creates on node of a red-black tree.
- void RBTfree (rbNode ∗root)

    Frees a whole red-black tree.
- static rbNode ∗ getUncle (rbNode ∗node)

    Gets the uncle of a given red-black node.
- static relation getRelationCase (rbNode ∗child, rbNode ∗parent, rbNode ∗grand)

    Helper function for "RBfixTRee".
- static void RBTrotation (rbNode ∗child, rbNode ∗parent)

    Rotates child and parent node in a red-black tree.
- static void RBfixtree (rbNode ∗node)

    Helper function for "RBTinsert" that removes inconsistencies of red-black rules.
- static void RBTfindroot (rbNode ∗∗root)

    Finds the root of some given node in a red-black tree.
- static rbNode ∗ RBTinsert (rbNode ∗∗root, rbNode ∗injector, void ∗data)

    Helper function for "RBTinsertElement".
- static rbNode ∗ RBTinsertElement (rbNode ∗∗root, void ∗data)

    If no node with the same data exists in a red-black tree, a new node is created an inserted in the tree.
- rbNode ∗ RBTminNode (rbNode ∗root)

    Gets the node with minimal data with respect to the defiend compare function.
- rbNode ∗ RBTmaxNode (rbNode ∗root)

    Gets the node with maximal data with respect to the defiend compare function.
- static rbNode ∗ RBTupUntilRightChild (rbNode ∗node)

    Helper function for "RBTpredecessor".
- static rbNode ∗ RBTupUntilLeftChild (rbNode ∗node)

    Helper function for "RBTsuccessor".
- rbNode ∗ RBTpredecessor (rbNode ∗node)

    Gets the previous node in order of the compare function.
- rbNode ∗ RBTsuccessor (rbNode ∗node)

    Gets the next node in order of the compare function.
- int RBTnodeCount (rbNode ∗root)

    Gets the number of nodes in a red-black tree.
- static void inc_ptr (void ∗∗Iterator, const size_t element_memsize)

    Helper function for "RBTtoIncArray".
- void RBTtoIncArray (rbNode ∗root, void ∗∗Array, int ∗size)

    Extracts all the data of a red-black tree to an ascending-ordered array.
- static void print_vector (double ∗V, int n) __attribute__((unused))
- static double CSRmatrixNorm (CSR_matrix ∗A)

    Computes the row-sum norm of a CSR matrix.
- static double scalar (double ∗x1, double ∗x2, int dim)

    Scalar product.
- static int cmp_matr_ele (void ∗X1, void ∗X2)

    Compare function for index ordering.
- static void free_matr_ele (void ∗X)

    Frees memory of a matrix element.
- static int cmp_eigen (void ∗X1, void ∗X2)

    Compares eigenvalues of an eigensystem (for ordering)
- static void free_eigen (void ∗X)

    Frees an element of an eigensystem.
- static eigen ∗ create_eigen (double val, double ∗vec, int dim)

    Create function for a real eigenystem element.

- static int ∗ zero_int_list (int n)

  Allocates a list of integers initialized by zero.
- static double ∗ zero_vector (int n)

  Allocates a list of doubles initialized by zero.
- static void scale (double ∗x, double a, int n)

  Scales a vector (or dense matrix in linear meory) by some factor.
- static double denseMatrixNorm (double ∗A, int rows, int cols)

  Computes a row-sum norm of a dense matrix.
- static void normalize (double ∗x, int dim)

  Normalizes a vector.
- static void addMultVec (double ∗x1, double ∗x2, double a, int dim)

  Adds a multiple of a vector to another vector.
- static double ∗ denseIDmatrix (int dim)

  Dense identity matrix.
- static void denseMatrixAdd (double ∗A, double ∗B, int rows, int cols, double factor)

  Adds a multiple of a dense matrix to another dense matrix.
- static void free_CSR_matrix (CSR_matrix ∗∗const A)

  Frees all memory of a CSR-matrix struct and sets to NULL.
- static void free_CSC_matrix (CSC_matrix ∗∗const A)

  Frees all memory of a CSC-matrix struct and sets to NULL.
- static CSR_matrix ∗ CSRinit (int rows, int init_size)

  Initiates a CSR matrix.
- static CSC_matrix ∗ CSCinit (int cols, int init_size)

  Initiates a CSC matrix.
- static void CSRresizeBuffer (CSR_matrix ∗A, int new_size)

  Resizes the element and index buffers of a CSR matrix.
- static void CSCresizeBuffer (CSC_matrix ∗A, int new_size)

  Resizes the element and index buffers of a CSC matrix.
- static CSR_matrix ∗ CSRid (int dim)

  Creates an identety matrix in CSR format.
- static void CSRchop (CSR_matrix ∗A, double thres)

  Sets near-to-zero elements of a matrix to zero.
- static void print_CSR_diagnoals (CSR_matrix ∗A)

  Prints diagonal elements of a CSR matrix to the output stream.
- static int CSRisDiagonal (CSR_matrix ∗A, double tol)

  Checks if a CSR matrix is diagonal.
- static double ∗ CSRgetDiagonal (CSR_matrix ∗A)

  Gets all the diagonal elements of a CSR matrix.
- static double CSRgetElement (CSR_matrix ∗A, int row, int col)

  Gets the matrix element with spefic index of a CSR matrix.
- static double ∗ getSubMatrix2D (CSR_matrix ∗A, int i, int j)

  Creates a 2x2 submatrix of a CSR matrix restricted to the given indices.
- static double ∗ CSRtoDense (CSR_matrix ∗A, int cols)

  Converts a matrix from CSR to dense row-ordered format.
- static double ∗ denseTranspose (double ∗M_row, int rows, int cols)

  Computes the transpose of a real dense matrix.
- static void DenseTranspose (double ∗∗M_row, int n)

  Replace a dense square matrix by its transpose.
- static double ∗ CSRtoDenseCol (CSR_matrix ∗A, int cols)

  Converts a CSR matrix to dense column-ordered format.
- static CSR_matrix ∗ denseToCSR (double ∗A, int cols, int rows, double tol)

Converts a matrix in dense row-ordered format to CSR format.
- static CSC_matrix ∗ denseToCSC (double ∗A, int cols, int rows, double tol)

  Converts a matrix in dense row-ordered format to CSC format.
- static CSR_matrix ∗ denseColToCSC (double ∗A_col, int cols, int rows, double tol)

  Converts a matrix in dense column-ordered format to CSC format.
- static double ∗ CSRextractCol (CSR_matrix ∗A, int colIndex)

  Extracts a column vector from a CSR matrix.
- static CSR_matrix ∗ CSRgetSubBlock (CSR_matrix ∗S, int start, int end)

  Gets a sub block matrix from a CSR matrix.
- void CSRinsertSubBlock (CSR_matrix ∗∗S, CSR_matrix ∗A, int start)

  Inserts a square block into a square CSR matrix.
- static double ∗ denseMatrixMult (double ∗A, int rowsA, int colsA, double ∗B, int rowsB, int colsB)

  Dense matrix product.
- static void CSRmatrixAdd (CSR_matrix ∗A, CSR_matrix ∗B, int cols, double factor)

  Adds a CSR matrix to another CSR matrix.
- static void CSRgetZeroCouplings (CSR_matrix ∗A, int ∗∗list, int ∗len, double tol)

  Gets a list of indices that are decoupled.
- static double CSRxCSC_row_times_col (CSR_matrix ∗A, CSC_matrix ∗B, int i, int j)

  Helper function for "matrix_product_to_CSR".
- static CSR_matrix ∗ parallel_sqr_matrix_product_to_CSR (CSR_matrix ∗A, CSC_matrix ∗B)

  Computes the matrix product.
- int CSRtoDensePadded (CSR_matrix ∗A, int cols, double ∗∗denseA, int ∗pitch)

  Converts a CSR matrix to a dense matrix that is memory aligned.
- int CSCtoDenseColPadded (CSC_matrix ∗A, int rows, double ∗∗denseA, int ∗pitch)

  Converts a CSC matrix to a dense matrix that is memory aligned.
- static CSR_matrix ∗ dense_matrix_product_to_CSR (CSR_matrix ∗A, int colsA, CSC_matrix ∗B, int rowsB, double tol)

  Computes the dense matrix product.
- static CSR_matrix ∗ CSRxCSC_SqrMatrixProduct (CSR_matrix ∗A, CSC_matrix ∗B, double tol)

  Computes the product of two square matrices.
- static void CSRsqrMatrixRightMult (CSR_matrix ∗∗A, CSC_matrix ∗B, double tol)

  Right-multiplies a CSR matrix with a CSC matrix.
- static CSR_matrix ∗ CSRtranspose (CSR_matrix ∗A, int cols)

  Computes the transpose of matrix.
- static CSC_matrix ∗ CSRtoCSC (CSR_matrix ∗A, int cols)

  Converts a CSR matrix to a CSC matrix.
- static int getPermMap (CSR_matrix ∗H, double tol, int ∗∗map, int ∗∗inv)

  Creates a permutation map that moves decoupled indices to the end.
- static void CSRindexPermutation (CSR_matrix ∗∗A, int ∗P)

  Applies a permutation map of a square matrix.
- static void denseQuadMatrixMult (double ∗A, double ∗B, int dim)

  Right-multiplies a dense square matrix to another.
- static double getWilkinsonShift (CSR_matrix ∗H, int index)

  Computes the Wilkinson shift.
- static double ∗ getHouseholderVector (double ∗M_row, int dim, int col)

  Computes a vector to generate a Householder transformation.
- static void rightMultHouseholder (double ∗M_row, double ∗v, int dim)

  Right-multiplies with a Householder matrix.
- static void leftMultHouseholder (double ∗M_row, double ∗v, int dim)

  Left-multiplies with a Householder matrix.
- static void denseHessenbergDecomposition (double ∗M_row, int dim, double ∗∗H, double ∗∗Q)

Hessenberg decomposition.

- static void denseGivensRotation (double ∗M_row, int dim, int i, int j, double c, double s)

    Givens rotation.
- static void denseHessenbergQR (CSR_matrix ∗H, int dim, CSC_matrix ∗∗Q, CSR_matrix ∗∗R, CSR_matrix ∗∗T, double tol)

    QR-decomposition of an upper-Hessenberg matrix.
- static int HessenbergHasBlockStructure (CSR_matrix ∗H, double tol, int last_sep, int ∗∗separators, int ∗num)

    Checks if an upper-Hessenberg Matrix has approximately block-diagonal structure and returns the separating column indices.
- static CSR_matrix ∗ subQRiterations (CSR_matrix ∗∗H, int offset, int ∗eigen_ind, int ∗eig_num, double tol, double chop_thres, int max_iter, shiftStrategy st)

    Helper function for "QRiterations".
- static int getExeDir (char ∗buffer, int size)

    Get the directory where the current process is executed.
- static int createGnuplotFile (char ∗cmdname, const char ∗dataname, int ∗colsToPlot, int colNum, char ∗∗labels)

    Creates a gnuplot script file to visualize data.
- static int programInPath (char ∗progname)

    Checks is the program named <progname> is in PATH via the shell command "which".
- void QRsetVerboseLevel (int level)

    Sets the verbose level of the computation.
- void getRealEigenVectors2D (double ∗denseH, double ∗eig, double ∗denseQ, double tol)

    Computes the eigenvectors of a symmetric 2x2 matrix.
- void getEigen2D (double ∗A, double ∗eig1_real, double ∗eig2_real, double ∗eig_im)

    Computes the eigenvalues of a real 2x2 matrix.
- double denseRayleighQuotient (double ∗A, double ∗x, int dim)

    Computes the Rayleigh quotient.
- void QRiterations (void ∗matrix, double ∗eigenValues, double ∗∗eigenVectors, int eig_num, double tol, int max_iter, shiftStrategy st)

    QR algorithm to compute all eigenvales and vectors of a matrix.
- static double pot (double x)

    Exemplary potential of the Schroedinger equation.
- void testQR (int n, int threads)

    Test routine for the QR algorithm.

Variables

- static int verbose_level = 0
- static int(∗ compare)(void ∗X1, void ∗X2) = NULL
- static void(∗ free_data)(void ∗X) = NULL
- static size_t data_size_in_bytes = 0

4.1.1 Typedef Documentation

4.1.1.1 typedef struct CSC CSC_matrix

Matrix type in Compressesed Sparse Column (CSC) format

see `https://en.wikipedia.org/wiki/Sparse_matrix` for details of the data format

4.1.1.2 typedef struct CSR CSR_matrix

Matrix type in Compressesed Sparse Row (CSR) format

see https://en.wikipedia.org/wiki/Sparse_matrix for details of the data format

4.1.1.3 typedef struct EIGEN eigen

Element of a real eigensystem consisting of an eigenvalue, its corrsponding eigenvector, and its dimension

4.1.1.4 typedef struct MATRIX_ELEMENT matrix_element

Matrix-element used for storage of the "Matrix-Market"-format The element A_{ij} is represented by a row(i) and column(j) index and the corresponding value (A_{ij})

see http://math.nist.gov/MatrixMarket/formats.html for more info

4.1.1.5 typedef enum RBCOLOR RBcolor

Color type for the red-black nodes

4.1.1.6 typedef struct RBNODE rbNode

Red-Black-Node type

see https://en.wikipedia.org/wiki/Red%E2%80%93black_tree

4.1.1.7 typedef enum RELATION relation

Relational-indicator type for red-black trees

4.1.2 Enumeration Type Documentation

4.1.2.1 enum RBCOLOR

Color type for the red-black nodes

Enumerator

- ***red***
- ***black***

4.1.2.2 enum RELATION

Relational-indicator type for red-black trees

Enumerator

- ***LEFTLEFT***
- ***LEFTRIGHT***
- ***RIGHTLEFT***
- ***RIGHTRIGHT***

4.1.3 Function Documentation

4.1.3.1 static void addMultVec (double * *x1*, double * *x2*, double *a*, int *dim*) [static]

Adds a multiple of a vector to another vector.

Parameters

double∗	x1: pointer to the vector that is modified
double∗	x2: pointer to the vector that is added
double	a: scaling factor
int	dim: dimenison of the vectors

The function computes x1 -> x1+a∗x2

4.1.3.2 static int cmp_eigen (void ∗ *X1*, void ∗ *X2*) [static]

Compares eigenvalues of an eigensystem (for ordering)

Parameters

void∗	X1: pointer to the first eigenvalue (cast from type eigen∗)
void∗	X2: pointer to the second eigenvalue (cast from type eigen∗)

Returns

int: compare result

This function compares two elements of an eigensystem for ordering. It returns +1 if eigenval(X1)>eigenval(X2) and -1 in the opposite case. If they equal within some tolerance it returns zero if the corresponding eigenvectors are linear dependent and else +1.

4.1.3.3 static int cmp_matr_ele (void ∗ *X1*, void ∗ *X2*) [static]

Compare function for index ordering.

Parameters

void∗	x1: pointer to the first matrix_element
void∗	x2: pointer to the second matrix_element

Returns

int: compare result

This function compares two matrix elements A_{ij} and A_{kl} according to their indices. The ersult is +1 if (i>k) or (i==k & j>l) and -1 if (i<k) or (i==k & j<l). When (i==k & j==l) the function returns zero

4.1.3.4 static eigen∗ create_eigen (double *val*, double ∗ *vec*, int *dim*) [static]

Create function for a real eigenystem element.

Parameters

double	val: real eigenvalue
double∗	vec: pointer to the eigenvector
int	dim: dimension of the eigenvector

This function creates an eigensystem element

4.1.3.5 static int createGnuplotFile (char ∗ *cmdname*, const char ∗ *dataname*, int ∗ *colsToPlot*, int *colNum*, char ∗∗ *labels*) [static]

Creates a gnuplot script file to visualize data.

Parameters

char∗	cmdname: absolute filename of the output script file
const	char∗ dataname: filename relative to the gnuplot script where the data is stored
int∗	colsToPlot: pointer to an array of length <colNum> where the indices of the columns to be plotted are listed
int	colNum: number of columns to be plotted
char∗∗	labels: pointer to array of strings (of length <colNum>) whith plot labels

Returns

int: returns 1 when successful, else 0

This method creates a gnuplot script file that plots the data in the textfile <dataname> that is stored in columns. The column indices to be plotted must be given in the array <colsToPlot>. The scipt is output under filename <cmdname>. Optionally one can give labels to each plotted column. If <labels>==NULL automatic labels or diplayed.

4.1.3.6 static CSC_matrix∗ CSCinit (int *cols*, int *init_size*) [static]

Initiates a CSC matrix.

Parameters

int	col: number of columns
int	init_size: initial number of elements that are allocated

Returns

CSC_matrix∗ : pointer to the allocated CSC matrix

The function allocates memory for storing <init_size> elements in a CSC matrix with <cols> columns. All column pointers are set to zero.

4.1.3.7 static void CSCresizeBuffer (CSC_matrix ∗ *A*, int *new_size*) [static]

Resizes the element and index buffers of a CSC matrix.

Parameters

CSC_matrix∗	A: pointer to CSC matrix to be resized
int	new_size: new buffer size

4.1.3.8 int CSCtoDenseColPadded (CSC_matrix ∗ *A*, int *rows*, double ∗∗ *denseA*, int ∗ *pitch*)

Converts a CSC matrix to a dense matrix that is memory aligned.

Parameters

CSC_matrix∗	A: pointer to matrix in CSR format
int	rows: number of columns of the matrix
double∗∗	denseA: pointer to address where the result of the conversion is stored
int∗	pitch: number of elements of each column in memory

Returns

int: 0 if allocation succeeded, else: one of the error codes EINVAL and ENOMEM

This function converts a CSC matrix to dense column-ordered format. The allocated memory is chache-optimized for 64bit-line read and memory aligned. Therefore, the actual number of doubles in each stored column, called pitch, may be larger than the number of rows. The result is computed and stored under adress pitch.

4.1.3.9 static void CSRchop (**CSR_matrix** ∗ *A*, double *thres*) [static]

Sets near-to-zero elements of a matrix to zero.

Parameters

CSR_matrix∗	A: pointer to a matrix in CSR format
double	thres: threshold under which matrix elements are approsimated a zeros

Deletes all elaments with |A_{ij}|<thres frome the CSR matrix

4.1.3.10 static double∗ CSRextractCol (**CSR_matrix** ∗ *A*, int *colIndex*) [static]

Extracts a column vector from a CSR matrix.

Parameters

CSR_matrix∗	A: pointer to CSR matrix
int	colIndex: index of the column to be extracted

Returns

double∗ : pointer to the created column vector

The function returns the column vector of index <colIndex> from the CSR matrix .

4.1.3.11 static double∗ CSRgetDiagonal (**CSR_matrix** ∗ *A*) [static]

Gets all the diagonal elements of a CSR matrix.

Parameters

CSR_matrix∗	A: pointer to CSR matrix

Returns

double∗ : pointer to vector where the diagonal elements are stored

The function returns a vector of length <A->rows> where all the diagonal elements are stored.

4.1.3.12 static double CSRgetElement (**CSR_matrix** ∗ *A*, int *row*, int *col*) [static]

Gets the matrix element with spefic index of a CSR matrix.

Parameters

CSR_matrix∗	A: pointer to a CSR matrix

int	row: row index
int	col: column index

Returns

double: value of matrix element

This function simply returns the matrix element A_{row,col} of a CSR matrix.

4.1.3.13 static CSR_matrix∗ CSRgetSubBlock (CSR_matrix ∗ *S*, int *start*, int *end*) [static]

Gets a sub block matrix from a CSR matrix.

Parameters

CSR_matrix∗	S: pointer to CSR matrix the block is extracted from
int	start: smallest index of the block
int	end: largest index of the block

Returns

CSR_matrix∗ : pointer to the created sub block

The function extracts a block with row indices i=start,start+1,...,end-1 and column index i=start,start+1,...,end-1 from the input and creates a new CSR matrix from the block.

4.1.3.14 static void CSRgetZeroCouplings (CSR_matrix ∗ *A*, int ∗∗ *list*, int ∗ *len*, double *tol*) [static]

Gets a list of indices that are decoupled.

Parameters

CSR_matrix∗	A: pointer to matrix in CSR format
int∗∗	list: pointer ot the address where the ersulting index list is stored
int∗	len: address where the length of the created list is stored
double	tol: tolerance for considering elements as zeros

The function generates a list of indices that are decoupled from the rest of the matrix . An index i is considered decoupled if {j}|A_{ij}|<tol and {j}|A_{ji}|<tol.

4.1.3.15 static CSR_matrix∗ CSRid (int *dim*) [static]

Creates an identety matrix in CSR format.

Parameters

int	dim: dimension of vector space of the matrix

Returns

CSR_matrix∗ : pointer to alloced CSR matrix

This function creates a dim x dim identity matrix in CSR format.

4.1.3.16 static void CSRindexPermutation (CSR_matrix ∗∗ *A*, int ∗ *P*) [static]

Applies a permutation map of a square matrix.

Parameters

CSR_matrix∗∗	A: pointer to address of the matrix to be permuted
int∗	P: poitner to index-permutation map as list of length <A->rows>

The function applies an index permutation to the input square CSR matrix.

4.1.3.17 static CSR_matrix∗ CSRinit (int *rows*, int *init_size*) [static]

Initiates a CSR matrix.

Parameters

int	row: number of rows
int	init_size: initial number of elements that are allocated

Returns

CSR_matrix∗ : pointer to the allocated CSR matrix

The function allocates memory for storing <init_size> elements in a CSR matrix with <rows> rows. All row pointers are set to zero.

4.1.3.18 void CSRinsertSubBlock (CSR_matrix ∗∗ *S*, CSR_matrix ∗ *A*, int *start*)

Inserts a square block into a square CSR matrix.

Parameters

CSR_matrix∗∗	S: pointer to address of the CSR matrix, in which the matrix is inserted
CSR_matrix∗	A: pointer to the CSR matrix that is inserted into <S>
int	start: insertion index

The function expands the (nxn)-matrix <S> by inserting the (mxm)-matrix A. The result is a (n+m x n+m)-matrix. For the new matrix it is Snew_{i,j}=Sold_{i,j} for 0<=i<start,0<=j<start Snew_{start+i,start+j} = A_{i,j} for 0<=i<m,0<=j<m. Snew_{m+i,m+j}=Sold_{i,j} for start<=i<n,start<=j<n Snew_{i,j}=0 else.

4.1.3.19 static int CSRisDiagonal (CSR_matrix ∗ *A*, double *tol*) [static]

Checks if a CSR matrix is diagonal.

Parameters

CSR_matrix∗	A: pointer to a matrix in CSR format
double	tol: tolerance for considering elements as zeros

Returns

int: test result (0 or 1)

The function checks if the CSR matrix is diagonal within tolerance <tol> and return 1 if so, and 0 else.

4.1.3.20 static void CSRmatrixAdd (CSR_matrix ∗ *A*, CSR_matrix ∗ *B*, int *cols*, double *factor*) [static]

Adds a CSR matrix to another CSR matrix.

Parameters

CSR_matrix∗	A: pointer to the CSR matrix that is modified
CSR_matrix∗	B: pointer to the CSR matrix that is added
int	cols: the column number of both matrices (that must be equal)
double	factor: scaling factor

Thos function computes the sum A->A+factor∗B of CSR matrices and stores it back in A.

4.1.3.21 static double CSRmatrixNorm (CSR_matrix ∗ *A*) [static]

Computes the row-sum norm of a CSR matrix.

Parameters

CSR_matrix∗	A: pointer a matrix in CSR format

Returns

double: row-sum norm

The function computes all the row-sums s_i={j}|A_{ij}| and returns the largest.

4.1.3.22 static void CSRresizeBuffer (CSR_matrix ∗ *A*, int *new_size*) [static]

Resizes the element and index buffers of a CSR matrix.

Parameters

CSR_matrix∗	A: pointer to CSR matrix to be resized
int	new_size: new buffer size

4.1.3.23 static void CSRsqrMatrixRightMult (CSR_matrix ∗∗ *A*, CSC_matrix ∗ *B*, double *tol*) [static]

Right-multiplies a CSR matrix with a CSC matrix.

Parameters

CSR_matrix∗∗	A: pointer to address of the matrix product in CSR format
CSC_matrix∗	B: pointer to a CSC matrix
double	tol: tolerance to consider matrix elements of the product to be zero

This function replaces the original matrix by the product .**where is given in CSR and in CSC format.**

4.1.3.24 static CSC_matrix∗ CSRtoCSC (CSR_matrix ∗ *A*, int *cols*) [static]

Converts a CSR matrix to a CSC matrix.

Parameters

CSR_matrix∗	A: pointer to a matrix in CSR format
int	cols: number of columns of the matrix

Returns

CSC_matrix∗ : the converted in CSC format

The function creates a copy of the input CSR matrix in CSC format.

4.1.3.25 static double* CSRtoDense (CSR_matrix * A, int cols) [static]

Converts a matrix from CSR to dense row-ordered format.

Parameters

CSR_matrix∗	A: pointer to CSR matrix to be converted
int	cols: number of columns of the matrix .

Returns

double∗ : pointer to converted dense matrix

The function creates a copy of a CSR matrix in dense row-ordered format.

4.1.3.26 static double∗ CSRtoDenseCol (CSR_matrix ∗ *A*, int *cols*) [static]

Converts a CSR matrix to dense column-ordered format.

Parameters

CSR_matrix∗	A: pointer to a CSR matrix
int	cols: number of columns of matrix

Returns

double∗ : pointer to the converted matrix

This function creates a copy of the input CSR matrix converted to column-ordered dense format.

4.1.3.27 int CSRtoDensePadded (CSR_matrix ∗ *A*, int *cols*, double ∗∗ *denseA*, int ∗ *pitch*)

Converts a CSR matrix to a dense matrix that is memory aligned.

Parameters

CSR_matrix∗	A: pointer to matrix in CSR format
int	cols: number of columns of the matrix
double∗∗	denseA: pointer to address where the result of the conversion is stored
int∗	pitch: number of elements of each row in memory

This function converts a CSR matrix to dense row-ordered format. The allocated memory is chache-optimized for 64bit-line read and memory aligned. Therefore, the actual number of doubles in each stored row, called pitch, may be larger than the number of columns. The result is computed and stored under adress pitch.

4.1.3.28 static CSR_matrix∗ CSRtranspose (CSR_matrix ∗ *A*, int *cols*) [static]

Computes the transpose of matrix.

Parameters

CSR_matrix∗	A: pointer to input matrix in CSR format
int	cols: number of columns of the matrix

Returns

CSR_matrix∗ : transpoed of the input in CSR format

This function returns the transpose of a real square matrix in CSR format

4.1.3.29 static double CSRxCSC_row_times_col (CSR_matrix ∗ *A*, CSC_matrix ∗ *B*, int *i*, int *j*) [static]

Helper function for "matrix_product_to_CSR".

Parameters

CSR_matrix∗	A: first factor of matrix product
CSC_matrix∗	B: second factor of matrix product
int	i: row index of matrix A
int	j: column index of matrix B

This helper function computes the row x column product, in order to compute a matrix product.

4.1.3.30 static CSR_matrix∗ CSRxCSC_SqrMatrixProduct (CSR_matrix ∗ *A*, CSC_matrix ∗ *B*, double *tol*) [static]

Computes the product of two square matrices.

Parameters

CSR_matrix∗	A: first factor in CSR format
CSC_matrix∗	B: second factor in CSC format
double	tol: tolerance to consider matrix elements of the product to be zero

Returns

CSR_matrix: product of the matrices in CSR format

This function choses a suitable method for a square matrix product computation depending on the size of the matrix.

4.1.3.31 static CSR_matrix∗ dense_matrix_product_to_CSR (CSR_matrix ∗ *A*, int *colsA*, CSC_matrix ∗ *B*, int *rowsB*, double *tol*) [static]

Computes the dense matrix product.

Parameters

CSR_matrix∗	A: first factor in CSR format
int	colsA: number of columns of
CSC_matrix∗	B: second factor in CSC format
int	rowsB: number of rows of
double	**tol: tolerance to consider matrix elements of the product to be zero**

Returns

CSR_matrix: product of the matrices in CSR format

The function computes the matrix product and is suited for small and highly filled matrices. The input matrices are converted to dense matrices to compute the matrx-matrix product for faster computation. The result is converted to CSR format afterwards.

4.1.3.32 static CSR_matrix∗ denseColToCSC (double ∗ *A_col*, int *cols*, int *rows*, double *tol*) [static]

Converts a matrix in dense column-ordered format to CSC format.

Parameters

double∗	A: pointer to the column-ordered dense matrix

int	col: number of columns of matrix
int	rows: number of rows of matrix
double	tol: tolerance for considering matrix elements as zeros

Returns

CSC_matrix∗ : pointer to converted matrix

The function creates a copy of a dense column-ordered matrix in CSC format. Elements with absolute value smaller than <tol> are considered as zeros.

4.1.3.33 static void denseGivensRotation (double ∗ *M_row*, int *dim*, int *i*, int *j*, double *c*, double *s*) `[static]`

Givens rotation.

Parameters

double∗	M_row: input square matrix in dense row-ordered format
int	dim: dimension of the vector space <M_row> is acting on.
int	i: subspace index one
int	j: subspace index two
int	c: cosine of rotation angle
inc	s: sine of rotation angle (s^2+c^2=1)

Applies a Givens rotation to the 2D-subspace spanned by indices *and <j> of the the row-ordered dense input matrix <M_row>. -> multiplies by {{c,-s},{s,c}} in subspace span(e_i,e_j)*

4.1.3.34 static void denseHessenbergDecomposition (double ∗ *M_row*, int *dim*, double ∗∗ *H*, double ∗∗ *Q*) `[static]`

Hessenberg decomposition.

Parameters

double∗	M_row: dense square matrix in row-orderd format that shall be decomposed
int	dim: dimension of the vector space <M_row> acts on
double∗∗	H: pointer to address under which the resulting upper-Hessenberg matrix is stored
double∗∗	Q: pointer to address under which the orthogonal transformation matrix is stored

The function generates a Hessenberg decomposition of the input matrix <M_row> that is M_row = Q.H.Q^T, where H is an upper-Hessenber matrix and Q is orthogonal (Q^T.Q=1).

4.1.3.35 static void denseHessenbergQR (CSR_matrix ∗ *H*, int *dim*, CSC_matrix ∗∗ *Q*, CSR_matrix ∗∗ *R*, CSR_matrix ∗∗ *T*, double *tol*) `[static]`

QR-decomposition of an upper-Hessenberg matrix.

Parameters

CSR_matrix∗	H: pointer to an upper-Hessenberg matrix in CSR format
int	dim: dimension of the vector space <H> acts on
CSC_matrix∗∗	Q: pointer to address where the orthogonal factor of the QR-decomposition is stored in CSC format
CSR_format∗∗	R: pointer to address where the triangular factor of the QR-decomposition is stored in CSR format

CSR_format∗∗	T: this is optional. If a NULL pointer is stored under the given address nothing is done. If there is a pointer to a CSR matrix, it is left-multiplied by <Q>
double	tol: tolerance for considering matrix elements as zeros.

This function creates a QR-decomposition of the given upper-Hessenberg matrix <H>: H = Q.R, where Q is orthogonal and R triangular. One may optinally provide a pointer <∗T> to a transformation matrix that is left multiplied by Q if one wishes to accumulate several transformations in one matrix.

4.1.3.36 static double∗ denseIDmatrix (int *dim*) [static]

Dense identity matrix.

Parameters

int	dim: dimension of vector space for the identety matrix

Returns

double: pointer to the linear row-indexed mememory

This function creates a dense version of a dim x dim identity matrix and returns a pointer to the newly created memory.

4.1.3.37 static void denseMatrixAdd (double ∗ *A,* double ∗ *B,* int *rows,* int *cols,* double *factor*) [static]

Adds a multiple of a dense matrix to another dense matrix.

Parameters

double∗	A: pointer to the dense matrix (row-ordered) that is modified
double∗	B: pointer to the dense matrix (row-ordered) that is added
int	rows: row number of the matrices
int	cols: column number of the matrices
double	factor: scaling factor

The function computes A -> A+factor∗B. It does the same thing a the function "addMultVec" but is defined separately for clearity

4.1.3.38 static double∗ denseMatrixMult (double ∗ *A,* int *rowsA,* int *colsA,* double ∗ *B,* int *rowsB,* int *colsB*) [static]

Dense matrix product.

Parameters

double∗	A: first factor in dense row-ordered format
int	rowsA: row number of matrix
int	colsA: column number of matrix
double∗	B: second factor in dense row-ordered format
int	rowsB: row number of matrix
int	**colsB: column number of matrix**

Returns

double∗ : pointer to product .that is dense and row-ordered

The function computes the dense matrix product C=A.B.

4.1.3.39 static double denseMatrixNorm (double ∗ *A,* int *rows,* int *cols*) [static]

Computes a row-sum norm of a dense matrix.

Parameters

double∗	A: pointer to an array of matrix elements stored in linear row format
int	rows: number of rows
int	cols: number of cols

Returns

double: row-sum norm

The norm takes a row-ordered array (ptr(i,j) = i∗cols+j) representing the matrix as first parameter. It computes all the row-sums s_i={j}|A_{ij}| and returns the largest.

4.1.3.40 static void denseQuadMatrixMult (double ∗ *A*, double ∗ *B*, int *dim*) [static]

Right-multiplies a dense square matrix to another.

Parameters

double∗∗	A: pointer to address of the dense matrix product in row-ordered format
double∗	B: pointer to dense matrix in column-ordered format
int	dim: dimension of the vector spaces the matrices act on

This function replaces the original dense square matrix by the product .**where is given in row-order and in columns order. The product is row-ordered again.**

4.1.3.41 double denseRayleighQuotient (double ∗ *A*, double ∗ *x*, int *dim*)

Computes the Rayleigh quotient.

Parameters

double∗	A: pointer to <dim>x<dim> square matrix in dense-row format.
double∗	x: pointer to a vector of length <dim>
int	dim: dimension of space

Returns

double: the Rayleigh quotient

This method computes the Rayleigh quotient R of a vector <x> for a given matrix according to R = <x,A.x>/<x,x>, where <∗,∗> is a scalar product.

4.1.3.42 static CSC_matrix∗ denseToCSC (double ∗ *A*, int *cols*, int *rows*, double *tol*) [static]

Converts a matrix in dense row-ordered format to CSC format.

Parameters

double∗	A: pointer to the row-ordered dense matrix
int	col: number of columns of matrix
int	rows: number of rows of matrix
double	tol: tolerance for considering matrix elements as zeros

Returns

CSC_matrix∗ : pointer to converted matrix

The function creates a copy of a dense row-ordered matrix in CSC format. Elements with absolute value smaller than <tol> are considered as zeros.

4.1.3.43 static CSR_matrix* denseToCSR (double * A, int cols, int rows, double tol) [static]

Converts a matrix in dense row-ordered format to CSR format.

Parameters

double∗	A: pointer to the row-ordered dense matrix
int	col: number of columns of matrix
int	rows: number of rows of matrix
double	tol: tolerance for considering matrix elements as zeros

Returns

CSR_matrix∗ : pointer to converted matrix

The function creates a copy of a dense row-ordered matrix in CSR format. Elements with absolute value smaller than <tol> are considered as zeros.

4.1.3.44 static double∗ denseTranspose (double ∗ *M_row,* int *rows,* int *cols*) [static]

Computes the transpose of a real dense matrix.

Parameters

double∗	M_row: pointer to a row-ordered dense matrix
int	rows: row number
int	cols: column number

Returns

double∗ : pointer to the created dense transpose

The function return the transpose of the matrix <M_row> also in row-ordered dense format.

4.1.3.45 static void DenseTranspose (double ∗∗ *M_row,* int *n*) [static]

Replace a dense square matrix by its transpose.

Parameters

double∗∗	M_row: pointer to the adress of a row-ordered dense matrix
int	n: row number of the nxn matrix

4.1.3.46 static void free_CSC_matrix (CSC_matrix ∗∗const *A*) [static]

Frees all memory of a CSC-matrix struct and sets to NULL.

Parameters

CSC_matrix∗∗	A: pointer to the pointer where the matrix is stored

4.1.3.47 static void free_CSR_matrix (CSR_matrix ∗∗const *A*) [static]

Frees all memory of a CSR-matrix struct and sets to NULL.

Parameters

CSR_matrix∗∗	A: pointer to the pointer where the matrix is stored

4.1.3.48 static void free_eigen (void ∗ *X*) [static]

Frees an element of an eigensystem.

Parameters

void∗	X: pointer to an eigenvalue (cast from type eigen∗)

4.1.3.49 static void free_matr_ele (void ∗ *X*) [static]

Frees memory of a matrix element.

Parameters

void∗	X: pointer to a matrix element

4.1.3.50 void getEigen2D (double ∗ *A*, double ∗ *eig1_real*, double ∗ *eig2_real*, double ∗ *eig_im*)

Computes the eigenvalues of a real 2x2 matrix.

Parameters

double∗	A: matrix in dense-row format
double∗	eig1_real: pointer to a double, where the real part of the smaller eigenvalues is stored
double∗	eig2_real: pointer to a double, where the real part of the larger eigenvalues is stored
double∗	eig_im: if not NULL the absolute imaginary part of both eigenvalues is stored at this address

This method computes the eigenvalue of a real (not necessary symmetric) matrix. Its eigenvalues are stored in the following manner: eigevalue#1 = <eig1_real>-i∗<eig_im> and eigevalue#2 = <eig2_real>+i∗<eig_im>

4.1.3.51 static int getExeDir (char ∗ *buffer*, int *size*) [static]

Get the directory where the current process is executed.

Parameters

char∗	buffer: string buffer where the result is stored
int	size: size of the buffer

Returns

int: 1 when succeeded, else 0

4.1.3.52 static double∗ getHouseholderVector (double ∗ *M_row*, int *dim*, int *col*) [static]

Computes a vector to generate a Householder transformation.

Parameters

double∗	M_row: square matrix to apply the transformation on in row-ordered dense format

int	dim: dimension of the vector space the matrix acts on.
int	col: column index for which the Housholder reflection is computed

Returns

double∗ : pointer where the result of length <dim> is stored

In order to create a Householder transformation T = 1-2∗v.v^T for input matrix <M_row> with corresponding column index <col>, the vector v is computed.

4.1.3.53 static int getPermMap (CSR_matrix ∗ *H,* double *tol,* int ∗∗ *map,* int ∗∗ *inv*) [static]

Creates a permutation map that moves decoupled indices to the end.

Parameters

CSR_matrix∗	H: pointer to input matrix in CSR format
double	tol: tolerance to consider matrix elaments a zeros
int∗∗	map: pointer to address where the index permutation map is stored
int∗∗	inv: poitner to address where the inverse map is stored

Returns

int: index where the decoupled block of the permuted matrix begins

This function looks for decoupled indices and creates a permutation map that moves the corresponding elemnts to the last part of the matrix. The inverse map is created, too. The index where the decoupled block starts is returned

4.1.3.54 void getRealEigenVectors2D (double ∗ *denseH,* double ∗ *eig,* double ∗ *denseQ,* double *tol*)

Computes the eigenvectors of a symmetric 2x2 matrix.

Parameters

double∗	denseH: The input matrix in dense-row format
double∗	eig: if eigenvalues are known, input a pointer to an array of them. Else set to NULL.
double∗	denseQ: The resulting eigenvectors are stored in an allocated matrix in dense-column format
double	tol: tolerance for off-diagonal elements to be treated as zeros

This method computes the (real) eigenvectors of a symmetric 2x2 matrix. The result is stored in <denseQ> that mus be a pointer to an allocated array of size=4∗sizeof(double). When v1 and v2 are the eigenvectors, then v1=(Q[0],Q[1])^T and v2 = (Q[2],Q[3])^T.

4.1.3.55 static relation getRelationCase (rbNode ∗ *child,* rbNode ∗ *parent,* rbNode ∗ *grand*) [static]

Helper function for "RBfixTRee".

Parameters

rbNode∗	child: child node
rbNode∗	parent: parent node of <child>
rbNode∗	grand: parent node of <parent>

Returns

relation: relational case

4.1.3.56 static double* getSubMatrix2D (CSR_matrix * A, int i, int j) [static]

Creates a 2x2 submatrix of a CSR matrix restricted to the given indices.

Parameters

CSR_matrix∗	A: pointer to a matrix in CSR format
int	i: index of first base vector
int	j: index of second base vector

Returns

double∗ : pointer to dense row-ordered 2x2 submatrix

This function creates a submatrix acting on the subspace span{e_i,e_j} from the given CSR matrix A (e_i,e_j base vectors).

4.1.3.57 static rbNode∗ getUncle (rbNode ∗ *node*) [static]

Gets the uncle of a given red-black node.

Parameters

rbNode∗	node: pointer to red-black node.

Returns

rbNode∗ : the uncle node, if <node> is root then NULL

4.1.3.58 static double getWilkinsonShift (CSR_matrix ∗ *H*, int *index*) [static]

Computes the Wilkinson shift.

Parameters

CSR_matrix∗	H: pointer to input matrix in CSR format
int	index: diagonal index of the 2x2 block for which the Wilkinson shift is computed

Returns

double: Wilkinson shift

The function computes the Wilkinson shift of a 2x2 block at diagonal index <index>.

4.1.3.59 static int HessenbergHasBlockStructure (CSR_matrix ∗ *H*, double *tol*, int *last_sep*, int ∗∗ *separators*, int ∗ *num*) [static]

Checks if an upper-Hessenberg Matrix has approximately block-diagonal structure and returns the separating column indices.

Parameters

CSR_matrix∗	H: pointer to an upper-Hessenberg matrix in CSR format
double	tol: threshold beneath wich matrix elemnts are consiedered as zero
int	last_sep: column index that confines the check to submatrix {0,...,<last_sep>}
int∗∗	separators: pointer to an address where the address of the newly created array of separator indices is stored.

int∗	num: address where the length of the created separator list is stored

4.1.3.60 static void inc_ptr (void ∗∗ *Iterator,* const size_t *element_memsize*) [inline], [static]

Helper function for "RBTtoIncArray".

Parameters

void∗∗	Iterator: pointer to address of a data array
const	size_t element_memsize: size of a data element in bytes

4.1.3.61 static void leftMultHouseholder (double ∗ *M_row,* double ∗ *v,* int *dim*) [static]

Left-multiplies with a Householder matrix.

Parameters

double∗	M_row: row-ordered dense square matrix that is left-multiplied
double∗	v: vector from which the Householder transformation T(v)=1-2∗v.v^T is build.
int	dim: dimension of the vector space <M_row> acts on

The function left-multiplies the dense square matrix <M_row> by a Householder transformation T(v)=1-2∗v.v^T created from vector <v>. The result is stored under the same address.

4.1.3.62 static void normalize (double ∗ *x,* int *dim*) [static]

Normalizes a vector.

Parameters

double∗	x: pointer to vector
int	dim: dimension of vector

This function normalizes a vector with respect to the Euklidean norm: x-> x/sqrt(<x,x>)

4.1.3.63 static CSR_matrix∗ parallel_sqr_matrix_product_to_CSR (CSR_matrix ∗ *A,* CSC_matrix ∗ *B*) [static]

Computes the matrix product.

Parameters

CSR_matrix∗	A: first factor of matrix product
CSC_matrix∗	B: second factor of matrix product

Returns

CSR_matrix∗: pointer to the newly allocated matrix product P=A.B

This method computes the matrix product P of the (n x k)-matrix A and the (k x m)-matrix B. The result is the (n x m)-matrix P=A.B. For faster computation the matrix A is given in row format and B in column format.

4.1.3.64 static double pot (double *x*) [static]

Exemplary potential of the Schroedinger equation.

Parameters

double	x: position in space

Returns

double: potential value

4.1.3.65 static void print_CSR_diagnoals (CSR_matrix ∗ *A*) [static]

Prints diagonal elements of a CSR matrix to the output stream.

Parameters

CSR_matrix∗	A: pointer to matrix in CSR format

The function prints all the diagonal elements (comma separated) to the output stream.

4.1.3.66 static void print_vector (double ∗ *V*, int *n*) [static]

4.1.3.67 static int programInPath (char ∗ *progname*) [static]

Checks is the program named <progname> is in PATH via the shell command "which".

Parameters

char∗	progname: Name of the executable to check

Returns

int: Returns 1 if the program is registered in PATH. If the executalbe does not exist or is not in PATH then 0 is returned

4.1.3.68 void QRiterations (void ∗ *matrix*, double ∗ *eigenValues*, double ∗∗ *eigenVectors*, int *eig_num*, double *tol*, int *max_iter*, shiftStrategy *st*)

QR algorithm to compute all eigenvales and vectors of a matrix.

Parameters

void∗	matrix: The matrix whose eigenvalues we are interested in. It must be stored in CSR_matrix format and its pointer cast to void∗
double∗	eigenValues: A pointer to an allocated array (of length <eig_num>) where the computed eigenvalues are stored
double∗∗	eigenVectors: A pointer to an allocated array of pointers (of length <eig_num>) where the computed eigenvectors are stored
int	eig_num: The number of eigenvalues/vectors to be computed
double	tol: floating point number tolarance. Beneath this threshold, matrix elements are approximated as zero entries
int	max_iter: maximum number of QR iterations
shiftStrategy	st: shift strategy for faster convergence

This method computes <eig_num> eigenvalues of a symmetric indefinite matrix <matrix> (where <eig_num> <= dimension(<matrix>)). For this, the QR algorithm with shift and deflation is used. First the matrix is transformed to an upper-Hessenberg matrix with Householder transformations. Subsequently a number of QR iterations is applied until some subdiagonal elements approach zero. The matrix is split if it approaches a block-diagonal form. For these blocks the QR algorithm is applied recursively (deflation). To improve convergence the spectrum can be improved by using a shift the the matrix. Three options are available for <st>: (1) ZERO: zero shift (is useful if one is only interested in the smallest absolute eigenvalues) (2) LASTDIAG: the shift is set to the last diagonal entry of the current matrix (3) WILKINSON: the Wilkinson shift is obtained by the eigenvalues of the last 2x2 block of the current Hessenber matrix. The eigenvalue that is closest to the last diagonal entry is chosen to be the shift.

4.1.3.69 void QRsetVerboseLevel (int *level*)

Sets the verbose level of the computation.

Parameters

int	level: detail level for terminam output

Depending on the verbose level more and more detailed output is written to the terminal. If <level>=0 the QR algorithm runs silently if no errors occur. For <level>=1,2 more details are written.

4.1.3.70 static void RBfixtree (rbNode * *node*) [static]

Helper function for "RBTinsert" that removes inconsistencies of red-black rules.

Parameters

*rbNode**	node: node of a subtree that may be inconsistent

4.1.3.71 static rbNode* RBTcreateNode (rbNode * *parent,* void * *data*) [static]

Creates on node of a red-black tree.

Parameters

*rbNode**	parent: parent node of the new node
*void**	data: pointer to the data that the node contains (note: that data is not copied, just the pointer is stored)

Returns

rbNode* : pointer to the newly created node

4.1.3.72 static void RBTfindroot (rbNode ** *root*) [static]

Finds the root of some given node in a red-black tree.

Parameters

*rNode***	root: pointer to address of some red-black node. The result is stored by the same pointer.

4.1.3.73 void RBTfree (rbNode * *root*)

Frees a whole red-black tree.

Parameters

rbNode	root: pointer to the root element of the red-black tree

4.1.3.74 static rbNode* RBTinsert (rbNode ** *root,* rbNode * *injector,* void * *data*) [static]

Helper function for "RBTinsertElement".

Parameters

rbNode∗∗	root: pointer to address of the root of the red-black tree
rbNode∗	injector: node where the newly created node is inserted
void∗	data: pointer to a data structure that the new node contains

Returns

rbNode∗: If a node with the same data is found, no new node is created and the pointer to this node is returned, else NULL is returned

4.1.3.75 static rbNode∗ RBTinsertElement (rbNode ∗∗ *root,* void ∗ *data*) [static]

If no node with the same data exists in a red-black tree, a new node is created an inserted in the tree.

Parameters

rbNode∗∗	root: pointer to address of the root node of the tree
void∗	data: pointer to a data structure that the new node contains

Returns

rbNode∗: If a node with the same data is found, no new node is created and the pointer to this node is returned, else NULL is returned

4.1.3.76 rbNode∗ RBTmaxNode (rbNode ∗ *root*)

Gets the node with maximal data with respect to the defiend compare function.

Parameters

rbNode∗	root: pointer to root node of the red-black tree
rbNode∗	pointer to maximal node

4.1.3.77 rbNode∗ RBTminNode (rbNode ∗ *root*)

Gets the node with minimal data with respect to the defiend compare function.

Parameters

rbNode∗	root: pointer to root node of the red-black tree
rbNode∗	pointer to minimal node

4.1.3.78 int RBTnodeCount (rbNode ∗ *root*)

Gets the number of nodes in a red-black tree.

Parameters

rbNode∗	root: pointer to root node of the red-black tree

4.1.3.79 rbNode∗ RBTpredecessor (rbNode ∗ *node*)

Gets the previous node in order of the compare function.

Parameters

rbNode∗	root: pointer to root node of the red-black tree
rbNode∗	: ponter to the predecessor node, if none exists then NULL

4.1.3.80 static void RBTree_set_compare (int(∗)(void ∗X1, void ∗X2) *Compare*) [static]

Sets user-defined compare functione for red-black ordering.

Parameters

int	(*Compare)(void* X1,void∗ X2): compare function

4.1.3.81 static void RBTree_set_data_size (size_t *data_size*) [static]

Sets the size of a user-defined data element in bytes.

Parameters

size_t	data_size: size of a data element in bytes

4.1.3.82 static void RBTree_set_free (void(∗)(void ∗X) *Free_data*) [static]

Sets user-defined free function for the red-black tree.

Parameters

void	(*Free_data)(void* X): free function

4.1.3.83 static void RBTrotation (rbNode ∗ *child*, rbNode ∗ *parent*) [static]

Rotates child and parent node in a red-black tree.

Parameters

rbNode∗	child: child node
rbNode∗	parent: parent node of <child>

4.1.3.84 rbNode∗ RBTsuccessor (rbNode ∗ *node*)

Gets the next node in order of the compare function.

Parameters

rbNode∗	root: pointer to root node of the red-black tree
rbNode∗	: ponter to the successor node, if none exists then NULL

4.1.3.85 void RBTtoIncArray (rbNode ∗ *root*, void ∗∗ *Array*, int ∗ *size*)

Extracts all the data of a red-black tree to an ascending-ordered array.

Parameters

rbNode∗	root: pointer to root node of the red-black tree
void∗∗	Array: pointer to address where the data array is stored
int∗	size: the size of the resulting array is stored under this pointer

4.1.3.86 static rbNode∗ RBTupUntilLeftChild (rbNode ∗ *node*) [static]

Helper function for "RBTsuccessor".

Parameters

rbNode∗	root: pointer to root node of the red-black tree

Returns

rbNode∗ : pointer to first upper node that has a left child, else NULL

4.1.3.87 static rbNode∗ RBTupUntilRightChild (rbNode ∗ *node*) [static]

Helper function for "RBTpredecessor".

Parameters

rbNode∗	root: pointer to root node of the red-black tree

Returns

rbNode∗ : pointer to first upper node that has a right child, else NULL

4.1.3.88 static void rightMultHouseholder (double ∗ *M_row,* double ∗ *v,* int *dim*) [static]

Right-multiplies with a Householder matrix.

Parameters

double∗	M_row: row-ordered dense square matrix that is right-multiplied
double∗	v: vector from which the Householder transformation T(v)=1-2∗v.v^T is build.
int	dim: dimension of the vector space <M_row> acts on

The function right-multiplies the dense square matrix <M_row> by a Householder transformation T(v)=1-2∗v.v^T created from vector <v>. The result is stored under the same address.

4.1.3.89 static double scalar (double ∗ *x1,* double ∗ *x2,* int *dim*) [static]

Scalar product.

Parameters

double∗	x1: pointer to first vector
double∗	x2: pointer to second vector
int	dim: dimension of the two vectors

Returns

double: the scalar product <x1,x2>

This function computes the euklidean scalar product p=<x1,x2> of two vectors x1 and x2.

4.1.3.90 static void scale (double * *x*, double *a*, int *n*) [static]

Scales a vector (or dense matrix in linear meory) by some factor.

Parameters

double∗	x: pointer to a vector(/dense matrix)
double	a: scaling factor
int	n: dimension of vector (or total storage size of dense matrix)

4.1.3.91 static CSR_matrix∗ subQRiterations (CSR_matrix ∗∗ *H*, int *offset*, int ∗ *eigen_ind*, int ∗ *eig_num*, double *tol*, double *chop_thres*, int *max_iter*, shiftStrategy *st*) [static]

Helper function for "QRiterations".

Parameters

CSR_matrix∗∗	H: pointer to the address of the upper-Hessenberg matrix whose eigensystem is requested
int	offset: column-index offset with respect to the larger supermatrix
int∗	eigen_ind: list of columns whose eigenvalues are found
int∗	eig_num: pointer to the length of the list <eigen_ind>
double	tol: tolerance beneath wich matrix elements are considered as zeros
double	chop_thres: cutoff threshold beneath which matrix element are deleted after matrix multiplication
int	max_iter: maximum number of QR iterations within some subdiagonal should approach zero (< <tol>)
shiftStrategy	st: shift strategy to apply

For detaila see function description of "QRiterations".

4.1.3.92 void testQR (int *n*, int *threads*)

Test routine for the QR algorithm.

Parameters

int	n: number of discretization nodes of the system
threads	number of threads to use

This method is a test routine for the QR algorithm. As an example it computes the energy levels of the one-dimensional stationary Schroedinger equation: (-d^2/dx^2+Vpot(x))psi = E∗psi on interval [0,1] with a potential well in the center. A finite difference scheme with n discretization nodes is used. When finished it outputs the all eigenvalues and eigenvectors to the terminal. If the system size is n>20 then the eigenvectors corresponding to the five lowest energy levels are written to the file "eigendata.txt" in the directory where the program is executed. If "gnuplot" is installed and in the PATH variable it is started and the eigenfunctions are shown (gnu script stored under "cmg.gnu").

4.1.3.93 static int∗ zero_int_list (int *n*) [static]

Allocates a list of integers initialized by zero.

Parameters

int	n: number of entries of the list

Returns

int∗ : pointer to allocated list

4.1.3.94 static double∗ zero_vector (int *n*) [static]

Allocates a list of doubles initialized by zero.

Parameters

int	n: dimension of vector

Returns

double* : pointer to allocated list

4.1.4 Variable Documentation

4.1.4.1 int(* compare)(void *X1, void *X2) = NULL [static]

Global variable where a pointer to the compare function for red-black trees is stored

4.1.4.2 size_t data_size_in_bytes = 0 [static]

Global variable that holds the size of a data element of a red-black node in bytes

4.1.4.3 void(* free_data)(void *X) = NULL [static]

Global variable where a pointer to the free function for red-black trees is stored

4.1.4.4 int verbose_level = 0 [static]

Verbose level of plotting infromation during computation in range 0-3, 0 is quiet

4.2 easyQR.h File Reference

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <omp.h>
#include <string.h>
#include <time.h>
```

Macros

- #define BUFF_SIZE 512

Typedefs

- typedef enum SHIFTSTRATEGY shiftStrategy

Enumerations

- enum SHIFTSTRATEGY { ZERO, LASTDIAG, WILKINSON }

Functions

- void testQR (int n, int threads)

  Test routine for the QR algorithm.
- void QRsetVerboseLevel (int level)

  Sets the verbose level of the computation.
- void QRiterations (void *A, double *eigenValues, double **eigenVectors, int eig_num, double tol, int max_↩ iter, shiftStrategy st)

  QR algorithm to compute all eigenvales and vectors of a matrix.
- void getEigen2D (double *A, double *eig1_real, double *eig2_real, double *eig_im)

  Computes the eigenvalues of a real 2x2 matrix.
- void getRealEigenVectors2D (double *denseH, double *eig, double *denseQ, double tol)

  Computes the eigenvectors of a symmetric 2x2 matrix.
- double denseRayleighQuotient (double *A, double *x, int dim)

  Computes the Rayleigh quotient.

4.2.1 Macro Definition Documentation

4.2.1.1 #define BUFF_SIZE 512

4.2.2 Typedef Documentation

4.2.2.1 typedef enum SHIFTSTRATEGY shiftStrategy

Shift strategy to apply for the QR algorithm. See https://de.wikipedia.org/wiki/QR-Algorithmus for details.

4.2.3 Enumeration Type Documentation

4.2.3.1 enum SHIFTSTRATEGY

Enumerator

ZERO

LASTDIAG

WILKINSON

4.2.4 Function Documentation

4.2.4.1 double denseRayleighQuotient (double * *A*, double * *x*, int *dim*)

Computes the Rayleigh quotient.

Parameters

*double**	A: pointer to <dim>x<dim> square matrix in dense-row format.
*double**	x: pointer to a vector of length <dim>
int	dim: dimension of space

Returns

double: the Rayleigh quotient

This method computes the Rayleigh quotient R of a vector <x> for a given matrix according to R = <x,A.x>/<x,x>, where <*,*> is a scalar product.

4.2.4.2 void getEigen2D (double ∗ *A,* double ∗ *eig1_real,* double ∗ *eig2_real,* double ∗ *eig_im*)

Computes the eigenvalues of a real 2x2 matrix.

Parameters

double∗	A: matrix in dense-row format
double∗	eig1_real: pointer to a double, where the real part of the smaller eigenvalues is stored
double∗	eig2_real: pointer to a double, where the real part of the larger eigenvalues is stored
double∗	eig_im: if not NULL the absolute imaginary part of both eigenvalues is stored at this address

This method computes the eigenvalue of a real (not necessary symmetric) matrix. Its eigenvalues are stored in the following manner: eigevalue#1 = <eig1_real>-i∗<eig_im> and eigevalue#2 = <eig2_real>+i∗<eig_im>

4.2.4.3 void getRealEigenVectors2D (double ∗ *denseH,* double ∗ *eig,* double ∗ *denseQ,* double *tol*)

Computes the eigenvectors of a symmetric 2x2 matrix.

Parameters

double∗	denseH: The input matrix in dense-row format
double∗	eig: if eigenvalues are known, input a pointer to an array of them. Else set to NULL.
double∗	denseQ: The resulting eigenvectors are stored in an allocated matrix in dense-column format
double	tol: tolerance for off-diagonal elements to be treated as zeros

This method computes the (real) eigenvectors of a symmetric 2x2 matrix. The result is stored in <denseQ> that mus be a pointer to an allocated array of size=4∗sizeof(double). When v1 and v2 are the eigenvectors, then v1=(Q[0],Q[1])^T and v2 = (Q[2],Q[3])^T.

4.2.4.4 void QRiterations (void ∗ *matrix,* double ∗ *eigenValues,* double ∗∗ *eigenVectors,* int *eig_num,* double *tol,* int *max_iter,* shiftStrategy *st*)

QR algorithm to compute all eigenvales and vectors of a matrix.

Parameters

void∗	matrix: The matrix whose eigenvalues we are interested in. It must be stored in CSR_matrix format and its pointer cast to void∗
double∗	eigenValues: A pointer to an allocated array (of length <eig_num>) where the computed eigenvalues are stored
double∗∗	eigenVectors: A pointer to an allocated array of pointers (of length <eig_num>) where the computed eigenvectors are stored
int	eig_num: The number of eigenvalues/vectors to be computed
double	tol: floating point number tolarance. Beneath this threshold, matrix elements are approximated as zero entries
int	max_iter: maximum number of QR iterations
shiftStrategy	st: shift strategy for faster convergence

This method computes <eig_num> eigenvalues of a symmetric indefinite matrix <matrix> (where <eig_num> <= dimension(<matrix>)). For this, the QR algorithm with shift and deflation is used. First the matrix is transformed to an upper-Hessenberg matrix with Householder transformations. Subsequently a number of QR iterations is applied until some subdiagonal elements approach zero. The matrix is split if it approaches a block-diagonal form. For these blocks the QR algorithm is applied recursively (deflation). To improve convergence the spectrum can be improved by using a shift the the matrix. Three options are available for <st>: (1) ZERO: zero shift (is useful if one is only interested in the smallest absolute eigenvalues) (2) LASTDIAG: the shift is set to the last diagonal entry of the current matrix (3) WILKINSON: the Wilkinson shift is obtained by the eigenvalues of the last 2x2 block of the current Hessenber matrix. The eigenvalue that is closest to the last diagonal entry is chosen to be the shift.

4.2.4.5 void QRsetVerboseLevel (int *level*)

Sets the verbose level of the computation.

Parameters

int	level: detail level for terminam output

Depending on the verbose level more and more detailed output is written to the terminal. If <level>=0 the QR algorithm runs silently if no errors occur. For <level>=1,2 more details are written.

4.2.4.6 void testQR (int *n,* int *threads*)

Test routine for the QR algorithm.

Parameters

int	n: number of discretization nodes of the system
threads	number of threads to use

This method is a test routine for the QR algorithm. As an example it computes the energy levels of the one-dimensional stationary Schroedinger equation: (-d^2/dx^2+Vpot(x))psi = E*psi on interval [0,1] with a potential well in the center. A finite difference scheme with n discretization nodes is used. When finished it outputs the all eigenvalues and eigenvectors to the terminal. If the system size is n>20 then the eigenvectors corresponding to the five lowest energy levels are written to the file "eigendata.txt" in the directory where the program is executed. If "gnuplot" is installed and in the PATH variable it is started and the eigenfunctions are shown (gnu script stored under "cmg.gnu").

4.3 testQR.c File Reference

```
#include "easyQR.h"
```

Functions

- int main (int argc, char *argv[])

    Test program to test the QR algorithm.

4.3.1 Function Documentation

4.3.1.1 int main (int *argc,* char * *argv[]*)

Test program to test the QR algorithm.

Parameters

arg	#1: the number of nodes to use (default value is n=200)
arg	#2: the number of threads to start

This little test program calls the test routine "testQR" from EasyQR.c. It computes the eigensystem of some matrix and plots it (see testQR)

Index