Collection of algorithms concerning dynamic mean-field theory for spins at infinite temperature

Description:

This repository contains implementations of the numerical methods spinDMFT, CspinDMFT and nl-spinDMFT. The codes are written entirely in C++.

Some formulas do not show up properly in the markdown.
create readme pdf with pandoc README.md -o README.pdf --mathjax

1. Installation of required programms and libraries

The installation is yet only explained for Linux operating systems.

programm/ library	description
GCC 13.2.0	C++ compiler collection, C++17 has to be available
MPI 4.1.6	for parallelized computations
Blaze 3.8	efficient header-only C++ library for matrices and vectors, see https://bitbucket.org/blaze-lib/blaze/src/master/
HDF5 1.10.10	C++ library for handling hdf5 files in C++, hdf5: efficient file format allowing to store meta data
Boost 1.83	C++ library, required here for parameter handling at runtime via with boost programm options

programm/ library	description
BLAS & LAPACK 3.12.0	C++ libraries, required here for blaze (e.g., for diagonalization)

If BLAS & LAPACK are not available, diagonalizations can alternatively be carried out with eigen (another C++ library, see https://eigen.tuxfamily. org/index.php?title=Main_Page). This may affect the performance, see https://bitbucket.org/blaze-lib/blaze/src/master/ for more information. How to employ eigen will be addressed below under "Altering CMakeLists.txt".

2. Compiling the codes regularly

Once all requirements have been installed, the codes can be compiled using cmake. The procedure for this is explained on the example of spinDMFT in the following, but it works exactly the same for CspinDMFT and nl-spinDMFT. Open a terminal and navigate to the directory "spinDMFT/Algorithm". Execute the following commands:

```
cp CMakeLists.txt_ CMakeLists.txt
mkdir build
cd build
cmake ..
make
```

The executable file "spinDMFT/executable_DOUBLE.out" should have been generated, if the compilation was successful.

FFAST MATH?!

3. Altering CMakeLists.txt

The compilation can be adjusted by altering the file "CMakeLists.txt". The options under the bullet point "# things to be adapted" may be changed if desired: One can switch from the data type DOUBLE to FLOAT under "set(USE_TYPE DOUBLE)" which

reduces the computation time and numerical accuracy. If BLAS & LAPACK are not available, Blaze cannot be installed properly. However, Blaze itself is a header-only library so that most of its functionality can still be used if the repository is just downloaded. Depending on the location of the Blaze directory, it may be automatically found in the compilation process. If not, one can provide the path to the blaze directory manually under "include_directories(/path/to/blaze)".Specifically, diagonalizations cannot be performed by blaze, if LAPACK is not available. In this case, Eigen may be installed as an alternative. To force the use of Eigen for the necessary diagonalizations, one has to enable "add_definitions(-DEIGEN)".

4. Testing the codes

The main directory of each code contains a simple testing script named "Tests.sh", which may be carried out via

```
./Tests.sh
```

once the executable file "executable_DOUBLE.out" exists. The script runs several quick simulations and plots their output using python and matplotlib (which obviously need to be installed for this to work). If the code is changed or extended, the testing script may serve as a quick check. The plots show a comparison between fixed reference data and newly generated test data. The deviation between reference and test has to be small (but not exactly zero because the Monte-Carlo simulation implies a statistical deviation). Note that the executed tests do not cover all options and parameters of the code and, therefore, do not guarantee the absense of any bugs. Note also that the Test data are automatically deleted at the end of the script.

5. Usage of the code spinDMFT

The executables "executable_XXX.out" with XXX=FLOAT or DOUBLE can be carried out using mpirun. At first one may test the help command via

```
mpirun executable_XXX.out --help
```

A help menu should pop up in the terminal. It shows the available options and parameters to run the code. The syntax for running an actual simulation is

```
mpirun -n Y executable_XXX.out --parameter1=value1 --parameter2=value2 --option1 --option2
```

Here, "Y" should be replaced by the desired number of cores to be used, and "parameter", "value" and "option" accordingly. Several parameters and options can be added in this way. They may be added in arbitrary order to the mpirun command. As a specific example, one may run

```
mpirun -n 4 executable_DOUBLE.out --numSamplesPerCore=2500 --numTimeSteps=100 --dt=0.02 --
```

This command starts a simulation of spinDMFT on 4 cores with 2500 samples per core, 100 time steps, a time step width of 0.02 considering an isotropic Heisenberg model. A small description of the simulation should appear followed by some progress bars. Once the simulation is finished, one should find an hdf5-file containing the corresponding simulation data in the Data directory. This file contains the bare simulation results as well as a lot of meta data including numerical details, time measurements of the simulation and the initially set parameters.

5.1 Reading the data

The data of the hdf5-file can be read out using

```
h5dump filename.hdf5
```

if h5dump is installed. For a better overview, one can alternatively use python. To this end, navigate to the main directory of the repository and execute

```
python quickreader.py <filename.hdf5>
```

replacing <filename.hdf5> by the actual name of the hdf5-file including the correct relative path. Note that the python library h5py is required for the script to work. The script prints all attributes and datasets to the terminal. For the datasets, only the shape is shown. One can execute

```
python quickreader.py <filename.hdf5> showdsets
```

if one desires to see the content of the datasets.

6. Usage of the codes CspinDMFT and nl-spinDMFT

In contrast to spinDMFT, the simulations of CspinDMFT and nl-spinDMFT are more complex and cannot be started immediately. Only the help command is easily accessible via

```
mpirun executable_XXX.out --help
```

replacing "XXX" by FLOAT or DOUBLE. For an actual simulation, one requires specific configuration files that carry information about the spin cluster and the spin-mean-field couplings to be considered. These files are stored in the subfolder "Configuration_Data" of the corresponding directory of each code. More information on these files and how to generate them can be found under 6.1 Creating configuration files. Once a configuration file exists, the simulations for CspinDMFT or nl-spinDMFT can be started the same way as for spinDMFT via the syntax

```
mpirun -n Y executable_XXX.out --config=<config_file> --parameter1=value1 --parameter2=val
```

replacing <config_file> by the corresponding file without ".hdf5"-ending and without the path (if the configuration file is located in a subdirectory of "Configuration_Data", this is specified via the "--project" option). The simulation will then be carried out utilizing the data in the configuration file. Note that providing the flag "--config=<config_file>" is mandatory.

To see how the data from CspinDMFT and nl-spinDMFT are stored, one may consider the Reference Data required for the test scripts "Tests.sh" stored in "Data/Tests". To this end, one can use h5dump or python as described in 5.1 Reading the data.

6.1 Creating configuration files

7. Functionality of the code spinDMFT

Before trying to understand the code in detail, one should understand how the method itself works. Therefore, we refer the reader to the open access article DOI and/or dissertation DOI.

The algorithm includes five different directories: "Functions", "Parameter_Space", "Run_Time_Data", "Storage_Concept" and "Time_Measure", each containing up to three files: a header file (.h) a cpp file (.cpp) and an error handling file (.h). The main code "main.cpp" is located in the directory "spinDMFT/Algorithm" and contains the essential steps done during a simulation. The functions and classes defined in the header files of the different directories are included in the main through "main_header.h". Further required header files can be found in the directory "cpp_libs", which is documented under "...".

7.1 Parameter_Space

- contains: Parameter_Space.h, Parameter_Space.cpp, PS_Error_Handling.h

This directory contains the class `ParameterSpace`, which is responsible for the handling of parameters. The class is initialized with the help of boost program options allowing to access all parameters at runtime. The parameters are set in the constructor of the class, which also contains their default values and explanation within the bpo::options_description.

Important physical parameters include the employed "spinmodel", the quadratic coupling constant "JQ", and a potential external field "Bname" or noise "Cname". Important numeric parameters include the symmetry type "cstype" (this aspect will be explained under "..."), the sample size "numSamplesPerCore", the step width "dt" and number of time steps "numTimeSteps" and the initial correlations inserted to the self consistency "initdcorr" and "initdcorr". Via the option "loadinit" and the parameters "initcorrsrc" and "initcorrfile", the initial correlations can also be loaded from a file. This can be helpful, for example, to efficiently redo a simulation with higher precision. Importing the converged lower precision correlations leads to a

more rapid convergence. The importing procedure is performed in the method `read_initial_correlations_from_file`. The method `create_essentials_string` allows to print some crucial parameters to the terminal.

Aside from the class ParameterSpace, the namespace also contains the struct `Noise` which allows to add a static noise source to the self consistency.

spinDMFT help command not perfect yet...
extend readme by spin length which is now a parameter

7.2 Run_Time_Data

- contains: Run_Time_Data.h, Run_Time_Data.cpp, RTD_Error_Handling.h

This directory contains the class `RunTimeData`, which is responsible for several numerical data generated in the process of the simulation. It analyzes and processes negative eigenvalues as well as the statistical error and the iteration error. The constructor initializes an instance of RunTimeData by importing some crucial numerical parameters from the ParameterSpace.

The constructor also initializes the truncation scheme `truncate_if_negative` for negative eigenvalues that potentially arise during the diagonalization of the mean-field covariance matrix. Generated eigenvalues can be handed over to the method `process_and_check_eigenvalues`, which searches for negative eigenvalues and truncates them. This method also computes the ratio of the sum of all negative eigenvalues with respect to the sum of all positive eigenvalues and terminates the simulation if the threshold value `critical_eigenvalue_ratio` is exceeded.

The constructor also initializes the single-sample square sum `sample_sqsum` $\sum_i x_i^2$ and the single-sample standard deviations `sample_stds`. The method `compute_sample_stds` obtains the Monte-Carlo sample sum $\sum_i x_i$ and employs it together with the `sample_sqsum` to compute the single-sample standard deviations $\frac{1}{M}\sum_i (x_i - \mu)^2$, where M is the sample size and μ is the average.

The method `compute_iteration_error` obtains the results of the current and previous iteration step and computes the iteration error as the maximum (over different directions) of the time average of the deviation between the inputs.

The method `terminate` returns true, if the simulation has converged, i.e., if the iteration error is below the value of `absolute_iteration_error_threshold`. Convergence is the regular scenario. The method also returns true, if the `Iteration_Limit` has been reached, or if not self-consistent iteration should be performed (bpo-option "noselfcons").

7.3 Time_Measure

- contains: Time_Measure.h, Time_Measure.cpp

This directory contains the class `DerivedTimeMeasure`, which is responsible for measuring the duration of different steps of the simulation. Measures are performed in the main programm by calling the method `measure` providing the name of the measurement and a bool value to decide whether the result of the measurement should be directly printed to the terminal. The method `measure` is implemented in the base class `TimeMeasure`, which can be found in the directory "cpp_libs", and essentially calls the method `internal_measure`, which is implemented locally in "Time_Measure.cpp".

An important aspect complicating the time measurement is the occurrence of nested loops. To ensure that `DerivedTimeMeasure` works properly, an instance of this class needs to recognize, when it enters and when it leaves a loop (at least, if time measurements are performed within the loop). This is done using the methods `enter_loop` and `leave_loop` right before and right after the loop.

7.4 Functions

- contains: Functions.h, Functions.cpp, Error_Handling.h

This directory contains a collection of functions used in "main.cpp". Amongst others, it includes generate_initial_spin_correlations, which generates the initial spin correlations by some predefined functions (bpo-options "initdcorr" and "initdcorrsrc") or by imported data (bpo-option "loadinit").

The method `self_consistent_equations` performs the self consistency, i.e., it transforms the spin correlations into mean-field correlations. This involves a transformation depending on the underlying spin model, a prefactor JQ^2 (quadratic coupling constant) and an optional external noise. The spin-model transformation is

performed to capture the correct spin-mean-field coupling. E.g., in case of a Heisenberg coupling, nothing is changed, i.e., each spin correlation $\langle S^\alpha(t)S^\beta(0)\rangle$ transforms directly into the corresponding mean-field correlation $\overline{V^\alpha(t)V^\beta(0)}$. But, for example, in case of an Ising coupling, $\langle S^z(t)S^z(0)\rangle$ transforms to $\overline{V^z(t)V^z(0)}$, while the other mean-field correlations strictly vanish. The spin model is implemented as a struct in "cpp_libs" under "Physics/Physics.h" and essentially contains the corresponding transformation matrix.

Besides the self consistency, the "Functions" directory also contains the essential steps of the time evolution such as the computation of propagators, the computation of sigma(t) as well as the subsequent computation of the spin correlations. The computation of the propagators via CFETS (commutator-free exponential time propagators) is briefly explained in ... . The results of the Monte-Carlo simulation are shared among the cores via `MPI_share_results`. More details on the parallelization can be found under 7.6 Parallelization. The method `normalize` normalizes the spin correlations with respect to the number of samples.

7.5 Storage_Concept

- contains: Storage_Concept.h, Storage_Concept.cpp, STOC_Error_Handling.h

This directory contains the class `StorageConcept`, which is responsible for storing the simulation results and any attached meta data in hdf5-format. The constructor creates any required directories via `create_folder_branch` and opens a new hdf5-file via `create_file`. The parameter `m_storing_permission` ensures that only a single core (my_rank=0) operates on the file.

The method `store_main` stores the "parameters", "runtimedata" and "results" each in a separate group. Many of the employed storing routines are implemented in "cpp_libs" in the header "HDF5/HDF5_Routines.h". The method `store_time` stores the time measurement performed by an instance of the class `DerivedTimeMeasure`.

7.6 main.cpp and main_header.h

The main code "main.cpp" includes the previously mentioned functions and classes through the header file "main_header.h", which imports several header files and defines shorthands for certain namespaces. The main code is carried out on several

cores simultaneously using MPI (message passing interface). It begins with initializing MPI and defining the integers `my_rank`, which numbers the cores, and `world_size`, which corresponds to the total number of cores. Subsequently, the time measurement, the parameter space, the initial spin correlations, the run time data and the duration estimator are initialized. The latter estimates the duration of the Monte-Carlo simulation based on the duration of processing the first sample. The class `DurationEstimator` is implemented in "cpp_libs" and will be discussed under "...". The tensor of spin correlations is stored using the class `CorrelationTensor<Correlation>`, which is explained in 10.2 Observables/Tensors.h.

The self-consistent iteration starts right after the initialization. Each iteration step starts with initializing the new spin correlations to be calculated (`my_new_spin_correlations`) and generating the mean-field distribution. The latter is done by calculating the second mean-field moments self-consistently from the previously computed spin autocorrelations (`my_spin_correlations`) employing the function `self_consistent_equations`. The latter returns the mean-field covariance matrix, which is then diagonalized to obtain the eigenvalues and the orthogonal transformation matrix. The eigenvalues are truncated and checked by the runtimedata class method `process_and_check_eigenvalues` and then used to create the mean-field normal distributions in the diagonal (non-correlated) basis. Subsequently, an instance of the class `std::mt19937` (Mersenne Twister random number generator) is initialized by providing an iteration and rank-dependent seed.

This is where the Monte-Carlo simulation starts. The operations in this part of the code are dependent on the core, since the random number generator is initialized with a unique rank-dependent seed. The Monte-Carlo samples are drawn in sets of the size `num_SamplesPerSet` (bpo-option "numSamplesPerSet"). This is more efficient because performing the transformation to the non-diagonal basis for several samples at once is more cache coherent. However, for memory reasons, the size of a set of samples can be limited. The noise samples are created as an instance of the class `GaussianNoiseVectorsBlocks`, which is implemented in "cpp_libs" and will be discussed under "...". The sampling in the diagonal basis as well as the transformation to the non-diagonal basis both happens in the initialization of the class instance. The class `NoiseTensorReaderScheme` allows for efficiently accessing the created noise.

Next, the results are computed for each drawn sample. The time propagators are

computed using CFETs as explained in ... . Subsequently, the time-evolved Pauli matrices are computed, which are then used to calculate the desired spin correlations. The single-sample results are simply added up in `my_new_spin_correlations`, which is later normalized. Note that also the squared sum of the single-sample results is computed. This is required for the computation of the standard deviation.

After the Monte-Carlo simulation is finished, the results are shared among the cores via `MPI_share_results`. The finalization of the iteration step includes computing the Monte-Carlo standard deviation, normalizing the spin correlations with respect to the sample size, computing the iteration error and printing some details. The iteration is terminated if the method `terminate` of the instance `my_rtdata` returns true as explained in 7.2 Run_Time_Data.

Once the iteration is finished, the data are stored using the class `HDF5_Storage`. At last, MPI is finalized.

8. Functionality of the code CspinDMFT

Before trying to understand the code in detail, one should understand how the method itself works. Therefore, we refer the reader to the open access article DOI and dissertation DOI.

A full documentation of this code is still pending. The code is structured similar to spinDMFT.

in CspinDMFT 4D tensor JCRA, ij, kl
why? -> k and l from 1 to numSpins have a direct physical meaning; in nl-spinDMFT environment and cluster are not necessarily related

9. Functionality of the code nl-spinDMFT

Before trying to understand the code in detail, one should understand how the method itself works. Therefore, we refer the reader to the open access article DOI and dissertation DOI.

A full documentation of this code is still pending. The code is structured similar to

spinDMFT.

10. Functionality of cpp_libs

The directory "cpp_libs" contains a collection of header files that are used in the different codes.

A full documentation of this code is still pending.

Error Handling contained in all directories

RealType
symmetry_type

10.1 Observables/Correlations.h

This header contains the class `CorrelationVector`, which is essentially a wrapper around `std::vector`. Despite standard element access and iterators, this class also allows some standard vector operations such as multiplication by scalar or adding or subtracting another vector. Morevover, a correlation can be constructed providing a function and an equidistant discretization (fixed δt).

10.2 Observables/Tensors.h

This header contains the template class `CorrelationTensor`, which serves for straight-forwardly storing and accessing quantities in three-dimensional real space. The class is used in all three codes. It is a wrapper around `std::vector` and contains `Correlation` elements. Each element is assigned to two direction indices $\alpha, \beta \in \{0(x), 1(y), 2(z)\}$, which are stored in the index-pair list `m_direction_pairs`. A key ingredient of the class is the `symmetry_type` containing the information about which elements of the tensor are equivalent or zero. It is set in the constructors and can be accessed by the method `get_symmetry`. The currently availabe symmetry types are:

symmetry type	size	correlations
A	1	$G_1 = G^{xx} = G^{yy} = G^{zz}, \quad$ rest is zero
B	2	$G_1 = G^{xx} = G^{yy}, \quad G_2 = G^{zz}, \quad$ rest is zero
C	4	$G_1 = G^{xx} = G^{yy}, \quad G_2 = G^{xy}, \quad G_3 = G^{yx}, \quad G_4 = G^{zz}, \quad$ rest is zero
D	9	all nine correlations are stored

Further symmetry types may be added to this header file if required.

Except for the default constructor, the constructors all require the `symmetry_type` as an argument (in some constructors indirectly). The last constructor is rather special and allows to create the `CorrelationTensor` from another `CorrelationTensor` and a so-called transformation scheme, which determines each `Correlation` element of the new `CorrelationTensor` from a certain transformation of the inserted `CorrelationTensor`. This functionality is used in the implementation of the self-consistency conditions.

The contained `Correlation` elements can be accessed by the get functions, e.g., `get_xx`, by iterators or by the operator `[]`, which obtains the linearized index. Trying to explicitly access a correlation that is zero within the current symmetry type leads to either a run-time error, or, if the method `zero_according_to` is available in the template class `Correlation`, returns a corresponding implemented zero value.

The iterator functions `iterate` and `const_iterate` allow for iterating over all non-zero tensor elements appying a lambda function that is handed over. The lambda function takes not only the `Correlation` element but also the corresponding direction index pair as arguments. With the extended iterator functions `iterate2` and `const_iterate2`, one can also iterate over two `CorrelationTensors` of the same symmetry type simultaneously. In this case, the lambda function obtains three arguments, namely the two `Correlation` elements and the corresponding direction index pair.

10.3 Observables/Clusters.h

10. Multivariate_Gaussian/ Multivariate_Gaussian.h

This header contains the classes `CovarianceMatrix`, `DiagonalBasisNormalDistributions` and `GaussianNoiseVectors`.

The class `CovarianceMatrix` is a wrapper around `SymmetricMatrix` therefore enforcing the contained matrix to be symmetric. However, positivity, which is another neccessary condition for a covariance matrix, is not enforced and has to be checked manually. The matrix can be filled within construction or after default construction in several different ways. Note that a non-diagonal matrix element M_{ij} only needs to be filled in once at (i, j) and is implicitly filled into (j, i) to due to the usage of `blaze::SymmetricMatrix`. The covariance matrix may be filled by providing a matrix via `fill_correlationmatrix` or a vector via `fill_correlationvector`. The latter is typically used in spinDMFT and its extensions, where the covariance matrix is built from spin correlations, which are stored as vectors in time. The vectors (v1, v2, v3, ...) are inserted according to the following scheme:

$$\begin{pmatrix} v1 & v2 & v3 & \dots \\ v2 & v1 & v2 & \dots \\ v3 & v2 & v1 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

Matrices and vectors may also be inserted into a subblock of the matrix providing the start row and column via the methods `fill_correlationvector_to_subtriangle`, `fill_correlationmatrix_to_subsquare` or `fill_symmetriccorrelationmatrix_to_diagonal_subblock`. The method `fill_correlationvector_to_subtriangle` fills the triangle of a subblock as well as the mirrored triangle in the covariance matrix. The method `fill_correlationmatrix_to_subsquare` fills a subblock of the covariance matrix as well as the mirrored subblock in the covariance matrix. If the subblock is around the diagonal, one may use the more efficient method `fill_symmetriccorrelationmatrix_to_diagonal_subblock` instead.

The method `diagonalize` computes the eigenvalues and eigenvectors, which are

provided by reference as function arguments.

The class `DiagonalBasisNormalDistributions` is a wrapper around an `std::vector` of `std::normal_distribution`. The latter are initialized with zero mean and variances provided in the constructor or in the `fill` method in case of default construction. The normal distributions can be accessed by standard vector iterators.

The class `GaussianNoiseVectors` is a wrapper around an `blaze::DynamicMatrix` and essentially stores a set of Gaussian-distributed noise vectors in matrix format. The noise is constructed providing an instance of `DiagonalBasisNormalDistributions`, a pseudo-random generator `std::mt19937` and, if unequal to 1, the number of noise samples in the set. The noise can be transformed to another basis via `basis_transform` providing an orthogonal transformation matrix (orthogonality is not checked). The noise can be accessed by `()` operators or standard matrix iterators.

10. Multivariate_Gaussian/ Multivariate_Gaussian_Blocks.h

This header extends "Multivariate_Gaussian.h" by providing the classes `CovarianceMatrixBlocks`, `DiagonalBasisNormalDistributionsBlocks` and `GaussianNoiseVectorsBlocks`, which are each wrappers around `std::vector` of the corresponding classes in "Multivariate_Gaussian.h" and contain essentially the same methods. A special new method in `CovarianceMatrixBlocks` is `fill_from_scheme`, which allows a template class `FillingScheme` to fill the covariance matrix. This is used in the implementation of the filling schemes in "Multivariate_Gaussian/ Symmetry_Schemes.h".

10. Multivariate_Gaussian/Symmetry_Schemes.h

The other header files in this subdirectory do not directly contain or require any information about the `symmetry_type`. The latter enters through filling and reader schemes implemented in this header file. It contains the template classes `CorrelationVectorTensorFillingScheme`, `CorrelationVectorTensorClusterFillingScheme`, `CorrelationMatrixTensorFillingScheme`, `NoiseTensorReaderScheme` and `NoiseTensorClusterReaderScheme`, which essentially do what their names suggest.

The filling schemes are used to fill the covariance matrix with correlations according to a specific `symmetry_type` and, in case of `CorrelationVectorTensorClusterFillingScheme`, also according to the number of spins in the cluster (this is required for CspinDMFT and nl-spinDMFT).
Generally, the covariance matrix is filled according to

$$\begin{pmatrix} v^{xx} & v^{xy} & v^{xz} \\ v^{yx} & v^{yy} & v^{yz} \\ v^{zx} & v^{zy} & v^{zz} \end{pmatrix} \quad \text{with} \quad v^{\alpha\beta} = \\ \begin{pmatrix} v^{\alpha\beta}(0,0) & v^{\alpha\beta}(0,\delta t) & v^{\alpha\beta}(0,2\delta t) & \dots \\ v^{\alpha\beta}(\delta t,0) & v^{\alpha\beta}(\delta t,\delta t) & v^{\alpha\beta}(\delta t,2\delta t) & \dots \\ v^{\alpha\beta}(2\delta t,0) & v^{\alpha\beta}(2\delta t,\delta t) & v^{\alpha\beta}(2\delta t,2\delta t) & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} = \\ \begin{pmatrix} v^{\alpha\beta}(0,0) & v^{\alpha\beta}(0,\delta t) & v^{\alpha\beta}(0,2\delta t) & \dots \\ v^{\alpha\beta}(\delta t,0) & v^{\alpha\beta}(0,0) & v^{\alpha\beta}(0,\delta t) & \dots \\ v^{\alpha\beta}(2\delta t,0) & v^{\alpha\beta}(\delta t,0) & v^{\alpha\beta}(0,0) & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} \quad \text{and} \quad v^{\alpha\beta}(t_1,t_2) \coloneqq \overline{V^{\alpha}(t_1)V^{\beta}(t_2)},$$

where time-translation invariance is assumed in the last step. In case of several mean-fields (CspinDMFT and nl-spinDMFT), the submatrices are built according to

$$V^{\alpha\beta} = \\ \begin{pmatrix} v_{11}^{\alpha\beta}(0,0) & v_{11}^{\alpha\beta}(0,\delta t) & \dots & v_{12}^{\alpha\beta}(0,0) & v_{12}^{\alpha\beta}(0,\delta t) & \dots \\ v_{11}^{\alpha\beta}(\delta t,0) & v_{11}^{\alpha\beta}(0,0) & \dots & v_{12}^{\alpha\beta}(\delta t,0) & v_{12}^{\alpha\beta}(0,0) & \dots \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots \\ v_{21}^{\alpha\beta}(0,0) & v_{21}^{\alpha\beta}(0,\delta t) & \dots & v_{22}^{\alpha\beta}(0,0) & v_{22}^{\alpha\beta}(0,\delta t) & \dots \\ v_{21}^{\alpha\beta}(\delta t,0) & v_{21}^{\alpha\beta}(0,0) & \dots & v_{22}^{\alpha\beta}(\delta t,0) & v_{22}^{\alpha\beta}(0,0) & \dots \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots \end{pmatrix} \quad \text{with} \quad v_{ij}^{\alpha\beta}(t_1,t_2) \coloneqq \\ \overline{V_i^{\alpha}(t_1)V_j^{\beta}(t_2)}.$$

Symmetries often allow to store the `CovarianceMatrixBlocks` more efficiently. Depending on the `symmetry_type`, the `CovarianceMatrixBlocks` are stored according to the following table

symmetry type	block sizes	subblocks
A	1	$v_1 = v^{xx} = v^{yy} = v^{zz}, \quad$ rest is zero
B	2	$v_1 = v^{xx} = v^{yy}, \quad v_2 = v^{zz}, \quad$ rest is zero
C	2	$v_1 = \begin{pmatrix} v^{xx} & v^{xy} \\ v^{yx} & v^{yy} \end{pmatrix}, \quad v_2 = v^{zz}, \quad$ rest is zero
D	1	the whole covariance matrix is stored in the single block

The class `CorrelationVectorTensorFillingScheme` is essentially build to fill an instance of `CorrelationTensor<CorrelationVector>` into the `CovarianceMatrixBlocks` following the above scheme. The `CorrelationVectorTensorClusterFillingScheme` simply extends this to filling an instance of `CorrelationCluster<CorrelationTensor<CorrelationVector>>`. The `CorrelationMatrixTensorFillingScheme` provides an alternative way of filling the covariance matrix if time-translation invariance is not valid. This is not further discussed because it is not available in the current implementation of the code.

Generating `GaussianNoiseVectorsBlocks` is not very complex and therefore not performed by a filling scheme but simply by using the class constructors in combination with the function `return_num_noises_per_block` implemented in this header file. Depending on the `symmetry_type`, the `GaussianNoiseVectorsBlocks` are stored according to the following table

symmetry type	block sizes	subblocks
A	1	$\mathcal{V}_1 = \begin{pmatrix} \mathcal{V}^x & \mathcal{V}^y & \mathcal{V}^z \end{pmatrix}$
B	2	$\mathcal{V}_1 = \begin{pmatrix} \mathcal{V}^x & \mathcal{V}^y \end{pmatrix}, \quad \mathcal{V}_2 = \mathcal{V}^z$
C	2	$\mathcal{V}_1 = \begin{pmatrix} \mathcal{V}^x \\ \mathcal{V}^y \end{pmatrix}, \quad \mathcal{V}_2 = \mathcal{V}^z$
D	1	$\mathcal{V}_1 = \begin{pmatrix} \mathcal{V}^x \\ \mathcal{V}^y \\ \mathcal{V}^z \end{pmatrix}$

where

$$\mathcal{V}^{\alpha} = \begin{pmatrix} V^{\alpha}_{(1)}(0) & \dots & V^{\alpha}_{(M)}(0) \\ V^{\alpha}_{(1)}(\delta t) & \dots & V^{\alpha}_{(M)}(\delta t) \\ \vdots & \dots & \vdots \end{pmatrix}$$

is the noise set with M samples for direction α and all time steps. In case of several mean-fields (CspinDMFT and nl-spinDMFT), the noise is stored according to

$$\mathcal{V}^{\alpha} = \begin{pmatrix} V^{\alpha}_{1,(1)}(0) & \dots & V^{\alpha}_{1,(M)}(0) \\ V^{\alpha}_{1,(1)}(\delta t) & \dots & V^{\alpha}_{1,(M)}(\delta t) \\ \vdots & \dots & \vdots \\ V^{\alpha}_{2,(1)}(0) & \dots & V^{\alpha}_{2,(M)}(0) \\ \vdots & \dots & \vdots \end{pmatrix}.$$

Note that the covariance matrix, or to be precise, its eigenvalues and orthogonal transformation are required to generate `GaussianNoiseVectorsBlocks` explaining why the latter must have the same block structure.

Last but not least, this header contains the noise reader schemes `NoiseTensorReaderScheme` and `NoiseTensorClusterReaderScheme`. These each carry a `std::shared_ptr` to a `GaussianNoiseVectorsBlocks` and allow to straightforwardly and efficiently read elements from it using pointers. The read-out is done time step by time step using the method `read` which, in case of `NoiseTensorReaderScheme` (for spinDMFT), returns the mean-field at the next time step as `Vec` (3D `blaze::StaticVector`). In case of `NoiseTensorClusterReaderScheme` (for CspinDMFT or nl-spinDMFT), the method returns all mean-fields at the next time step as `VecVec` (`std::vector` of `Vec`).

10. Physics/Physics.h

This header contains several practical classes used in the different codes. Amongst others, it contains the classes `DiagonalSpinCorrelation` and `NonDiagonalSpinCorrelation`, which generate `std::function` objects that are used to define the initial spin correlations used in the algorithms. Providing an step width δt, the method `create_discretization` evaluates the function at a discrete equidistant

set of time steps.

The class `SpinModel` is used to define the model by which the spins couple to one another. The coupling is always bilinear and of the form $\vec{S}_i^\top \underline{\underline{D}} \vec{S}_j$ with some transformation matrix $\underline{\underline{D}}$, which is not necessarily a true rotation matrix.

The currently available spin models are Ising, Heisenberg and XY. The class `SpinModel` is used in the implementation of the self-consistency conditions in "Functions.cpp" and is not directly used in the main code.

10. Physics/CFET.h

10. Physics/Clusterization.h

Roadmap

Adding a full documentation of the code CspinDMFT and nl-spinDMFT is planned.
python_libs?

potential efficiency boost covmatrix diagonalization not parallelized

Project status

The project is updated frequently.

Contributing

Support

Feel free to contact the author by sending an e-mail to "timo.graesser@tu-dortmund.de" for any questions or suggestions. Note that the code has been written by a physicist not software engineer. Any suggestions for more clarity or higher efficiency are appreciated.

License

Extending the Codes

Generally

how to include new parameters:

1. add it in the header file Parameter_Space.h
2. add it in Parameter_Space.cpp, so that it can be adjusted with boost program options
   3a) perhaps add it to the essential parameters print function
   3b) add it to the filename string if its a physical parameter
3. add it to void HDF5_Storage::store_main in Storage_Concept.cpp
4. include the parameter in the code

----- Options -----

efficiency test with: perf record, perf report

Guideline for Naming:
classes, structs: 'UpperCamelCase'

functions, methods: 'lower_case'

variable instances: varying
-> int, uint: 'lowerCamelCase' (often 'num_UpperCamelCase')
-> double: 'lower_case'
-> string, char: 'lower_case'
-> own class/struct inst: 'my_lower_case'

static const instances: 'UPPER_CASE'

error functions: 'UPPER_CASE'

name spaces: 'Upper_case'

nl-spinDMFT

1. Importing mean-fields
   Tha algorithm imports spin correlations, which can be transformed into mean-field correlations...

explanation:
in case of loaded correlations: correlation tensor cluster should in the first step be independent
of numSpins (because it relates only to the environment)

1. import desired set of correlations (e.g. "11", "12", ...)
2. give each correlation an index k (k could be just the position of the correlation)
3. compute mean-field ViVj from transformation that maps from k to ij (new routine for this, not the standard self-consistency)