MolDyPoP

Version 2.0

Generated by Doxygen 1.9.7

Chapter 1

MolDyPoP Main page

1.1 Introduction

Welcome to MolDyPoP, the simulation environment for Molecular Dynamics for Polar Particles. You will find a full documentation of the MolDyPoP package on these pages. Additionally, there is a quick tutorial on how to set yourself up for a simulation run.

If you truly wish to understand the code, you will have to be able to dig a little bit. The core MolDyPoP package is written in C++ and this documentation provides you with an exhaustive explanation of what the individual C++ files are meant to do, and how the classes and namespaces work.

However, calls to MolDyPoP typically happen via the console and you will need to submit MolDyPoP calculations to the SCC cluster, which in itself is not part of the C++ routine and requires knowledge of some shell coding. Some basic scripts that may be of value to you have been compiled here - to properly understand them requires an understanding of the bash-shell.

Finally, the results produced in MolDyPoP runs are MATLAB-executable files. The MATLAB-scripts I used in data analysis are therefore also provided here.

Note that neither the bash nor the MATLAB scripts are fully commented for use, the commenting effort has been focused on giving a proper explanation of the workings of MolDyPoP. I will give quick explanations of what the other scripts do, and with some digging in bash and experimenting with matlab syntax you should be able to understand those scripts fairly quickly.

1.2 structure

You will find four folders in this implementation.

1. A folder MolDyPoP/.

Chapter 2

Namespace Index

2.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 3

Class Index

3.1 Class List

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

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 routines Namespace Reference

Different calculation routines with xygroups and mxygroups. Carries out simulation tasks.

Functions

- int integration (parameters par)

  basic integration routine
- int sampling (parameters par)

  basic sampling routine (no integration performed)
- int equilibrate (group &G, const parameters &par, sampler &samp, const double Tmax, double &t, const std::string breakcond, std::ofstream &stdoutfile)

  equilibration routine
- void integrate_snapshots (group &G, const parameters &par, sampler &samp, std::ofstream &stdoutfile)

  integration routine (the one that does the work)
- void initprint (std::string routine_name, std::ofstream &outfile)

  Initial print of each routine. States the routine name.
- void terminateprint (std::string routine_name, std::ofstream &outfile)

  Terminal print of each routine. States the routine name.

5.1.1 Detailed Description

Different calculation routines with xygroups and mxygroups. Carries out simulation tasks.

Author

Thomas Bissinger

Date

Created: 2019-12-11

Last Updated: 2023-08-06

5.1.2 Function Documentation

5.1.2.1 equilibrate()

```
int routines::equilibrate (
            group & G,
            const parameters & par,
            sampler & samp,
            const double Tmax,
            double & t,
            const std::string breakcond,
            std::ofstream & stdoutfile )
```

equilibration routine

Takes in a group and integrates it until Tmax (or another break condition is met), then returns whether or not the group is equilibrated then.

Reads data from snapshot files provided in a list file and performs sampling on it. Proceeds as follows:

1. Preliminary stuff (opening files, initial print)
2. Performs time integration (depending on which integrator chosen)
3. After a time set in par, the integration checks for equlibration and decides whether or not to continue equilibrating
4. Cleanup, final prints

Parameters

in,out	*G*	The group that should be equilibrated
in	*par*	A set of simulation parameters
in,out	*samp*	The sampler in which equilibration data should be stored (careful, will be reset during run - TODO!)
in	*Tmax*	Maximum equilibration time
in	*t*	Current time
in	*breakcond*	Break condition. The following values can be taken: **Table 5.1 Values of fluctname** **breakcond value** / **meaning** ¨time¨ — Wait until Tmax. After that, another equilibration check is performed and the routine may be called again. ¨time_hard¨ — Wait until Tmax. No further equilibration performed. ¨temperature¨ — Breaks if the desired temperature is reached and maintained for a certain amount of time. ¨any¨ — Any of the above.
in	*stdoutfile*	File to which output is to be printed.

5.1.2.2 initprint()

```
void routines::initprint (
            std::string routine_name,
            std::ofstream & outfile )
```

Initial print of each routine. States the routine name.

5.1.2.3 integrate_snapshots()

```
void routines::integrate_snapshots (
            group & G,
            const parameters & par,
            sampler & samp,
            std::ofstream & stdoutfile )
```

integration routine (the one that does the work)

Takes in a group and integrates it until par.Tmax(). May store snapshots or perform sampling on the fly, depending on parameters. Many details depend on the parameters chosen and can be checked in the declaration of parameters.h

Proceeds as follows:

1. Preliminary stuff (opening files, initial print)
2. Initializes group (setting positions to lattice, initializing partition, drawing random velocities etc.)
3. Performs time integration (depending on which integrator chosen)
4. During integration, samples and stores data
5. Cleanup, final prints - no print of sampled data, that is done in the routine integration that typically calls for this function

Parameters

in,out	*G*	The group that should be equilibrated
in	*par*	A set of simulation parameters
in,out	*samp*	The sampler in which the run data will be stored
in	*stdoutfile*	File to which output is to be printed.

5.1.2.4 integration()

```
int routines::integration (
            parameters par )
```

basic integration routine

Proceeds as follows:

1. Preliminary stuff (opening files, initial print)

2. Initializes all relevant objects (group, integrator, sampler)
3. Performs an equilibration run (typically with check to temperature, depends on switch)
4. Performs an integration run (used for sampling)
5. Cleanup, final prints

Note

The routine equilibrate and the routine integrate_snapshots are used within this routine.

Equilibration is not really managed in an elegant way. It is recommended to check manually whether data has been equilbrated and to use a fixed equilibration time.

5.1.2.5 sampling()

```
int routines::sampling (
            parameters par )
```

basic sampling routine (no integration performed)

Reads data from snapshot files provided in a list file and performs sampling on it. Proceeds as follows:

1. Preliminary stuff (opening files, initial print)
2. Initializes all relevant objects (group and sampler)
3. For each time step in the list file, reads out the group and performs sampling on the group at that time instant.
4. Cleanup, final prints

Note

This routine can only be used when snapshots are stored during another integration/sampling run. Useful for explorative investigations, but large numbers of snapshots should not be stored for a large sample and it is recommended to perform on-fly sampling.

5.1.2.6 terminateprint()

```
void routines::terminateprint (
            std::string routine_name,
            std::ofstream & outfile )
```

Terminal print of each routine. States the routine name.

5.2 topology Namespace Reference

Contains the vector classes and vector functions and operations. So far, only the class Vector2d is used in further routines and fully implemented.

Classes

- class angle2d

  A (double-valued) angle in 2d space with the possibility of identifying it with its corresponding Vector2d unit vector. In the current state of the simulation, this is redundant.
- class Vector2d

  Mathematical 2d vectors. Can be added, multiplied by a scalar, norm computation is possible. There are print-to-file and print-to-command-line functions available.

Functions

- Vector2d operator+ (const Vector2d &v, const Vector2d &w)
- Vector2d operator- (const Vector2d &v, const Vector2d &w)
- Vector2d operator∗ (const Vector2d &v, const double a)

  scalar ∗ operator
- Vector2d operator∗ (const double a, const Vector2d &v)

  scalar ∗ operator
- Vector2d operator∗ (const Vector2d &v, const int a)

  scalar ∗ operator
- Vector2d operator∗ (const int a, const Vector2d &v)

  scalar ∗ operator
- Vector2d operator/ (const Vector2d &v, const double a)

  scalar / operator
- Vector2d operator/ (const Vector2d &v, const int a)

  scalar / operator
- double norm2 (Vector2d v)

  Returns the L2 norm of a vector.
- Vector2d normalized (Vector2d v)

  Normalizes a vector.
- double periodic_distance_squared (const Vector2d &v, const Vector2d &w, const double &L)

  Returns squared distance |w - v|^2 considering periodic boundaries (square box, length L). Squared function faster to calculate.
- double periodic_distance (const Vector2d &v, const Vector2d &w, const double &L)

  Returns distance |w - v| considering periodic boundaries (square box, length L). Taking sqrt takes more time than returning the squared quantity by dist_periodic_squared.
- Vector2d periodic_distance_vector (const Vector2d &v, const Vector2d &w, const double &L)

  Returns distance vector w - v considering periodic boundaries (square box, length L).
- double periodic_distance_squared (const Vector2d &v, const Vector2d &w, const Vector2d &L)

  Returns squared distance |w - v|^2 considering periodic boundaries (rectangular box, widths stored in L). Squared function faster to calculate.
- double periodic_distance (const Vector2d &v, const Vector2d &w, const Vector2d &L)

  Returns squared distance |w - v| considering periodic boundaries (rectangular box, widths stored in L). Taking sqrt takes more time than returning the squared quantity by dist_periodic_squared.
- Vector2d periodic_distance_vector (const Vector2d &v, const Vector2d &w, const Vector2d &L)

  Returns distance vector w - v considering periodic boundaries (rectangular box, widths stored in L).
- Vector2d rotate (Vector2d v, double theta)

  Rotates a vector.
- Vector2d rotate_orthogonal (Vector2d v)

  Rotates 90 degrees.
- double innerproduct (Vector2d v, Vector2d w)

  Inner product.
- double parallel_projection (Vector2d v, Vector2d w)

Parallel projection.

- double orthogonal_projection (Vector2d v, Vector2d w)

  Parallel projection.
- Vector2d random_vector (const Vector2d &minima, const Vector2d &maxima)

  Returns a random vector within a volume [minima, maxima].
- Vector2d random_vector (const Vector2d &maxima)

  Returns a random vector within a volume [0,maxima].
- Vector2d random_gaussian_vector (const double &sigma_squared)

  Returns a random vector with Gaussian distribution in each component.
- Vector2d nearest_gridvec (Vector2d v, double gridsep)

  Returns nearest neighbor to 2D-Vector v on grid of grid point separation gridsep.
- Vector2d nearest_gridvec (Vector2d v, Vector2d gridseps)

  Returns nearest neighbor to 2D-Vector v on grid of anisotropic grid point separation gridsep.
- std::vector< Vector2d > qvalues_within_radius (double qmin, double qmax, Vector2d gridseps, int qsamps↩ _per_bin)

  Creates a std::vector containing qsamps_per_bin 2D-vectors that lie on a grid with modulus between qmin and qmax. No vector appears double.
- Vector2d random_vector_first_quadrant (const double length)

  Returns a random vector in the first quadrant.
- Vector2d random_vector_sector (const double length, const double thetamin, const double thetamax)

  Returns a random vector within a sector given by thetamin, thetamax.
- Vector2d random_velocity (const double r)

  Returns a random vector on a sphere surface of radius r.
- Vector2d vector_from_angle (const double angle, const double r)

  Returns a vector of length r and orientation angle.
- Vector2d vector_from_angle (const double angle)

  Returns a vector of unit length and orientation angle.
- double angle_from_vector (const topology::Vector2d &v)

  Returns the orientation angle of a vector.
- Vector2d spin (double theta)

  Returns a spin vector, i.e. unit vector, with given angle to x-axis.
- Vector2d orthospin (double theta)

  Returns an orthogonal spin vector, i.e. a unit vector rotated 90^ °counterclockwise from the vector of spin(theta)
- Vector2d vector_on_squarelattice (int index, int Nx, int Ny, double spacing)

  Index-dependent position vector for spin at index, square lattice.
- Vector2d vector_on_trigonallattice (int index, int Nx, int Ny, double spacing)

  Index-dependent position vector for spin at index, trigonal lattice.
- void print_matlab (const std::vector< Vector2d > &v, std::string name, std::ostream &outfile)

  Prints a list of vectors to in matlab-readable form.
- angle2d operator+ (const angle2d &v, const angle2d &a)

  Addition operator.
- angle2d operator- (const angle2d &v, const angle2d &w)

  Subtraction operator.
- angle2d operator∗ (const angle2d &v, const double a)

  Multiplication operator (double, right)
- angle2d operator∗ (const double a, const angle2d &v)

  Multiplication operator (double, left)
- angle2d operator/ (const angle2d &v, const double a)

  Division operator (double)

5.2.1 Detailed Description

Contains the vector classes and vector functions and operations. So far, only the class Vector2d is used in further routines and fully implemented.

Generalizations are possible. The namespace was originallz intended to be extensible to 3d vectors and spherical coordinates. Right now, however, no such routines are implemented. The class Angle2d exists with basic routines and some conversion and other relations with Vector2d are available, but it is not fully developed.

Author

Thomas Bissinger

Date

Created: early 2017

Last Updated: 2023-08-01

5.2.2 Function Documentation

5.2.2.1 angle_from_vector()

```
double topology::angle_from_vector (
            const topology::Vector2d & v )  [inline]
```

Returns the orientation angle of a vector.

5.2.2.2 innerproduct()

```
double topology::innerproduct (
            Vector2d v,
            Vector2d w )  [inline]
```

Inner product.

5.2.2.3 nearest_gridvec() [1/2]

```
topology::Vector2d topology::nearest_gridvec (
            topology::Vector2d v,
            double gridsep )
```

Returns nearest neighbor to 2D-Vector v on grid of grid point separation gridsep.

5.2.2.4 nearest_gridvec() [2/2]

```
topology::Vector2d topology::nearest_gridvec (
            Vector2d v,
            Vector2d gridseps )
```

Returns nearest neighbor to 2D-Vector v on grid of anisotropic grid point separation gridsep.

5.2.2.5 norm2()

```
double topology::norm2 (
            Vector2d v )  [inline]
```

Returns the L2 norm of a vector.

5.2.2.6 normalized()

```
Vector2d topology::normalized (
            Vector2d v )  [inline]
```

Normalizes a vector.

5.2.2.7 operator∗() [1/6]

```
topology::angle2d topology::operator* (
            const angle2d & v,
            const double a )
```

Multiplication operator (double, right)

5.2.2.8 operator∗() [2/6]

```
topology::angle2d topology::operator* (
            const double a,
            const angle2d & v )
```

Multiplication operator (double, left)

5.2.2.9 operator∗() [3/6]

```
topology::Vector2d topology::operator* (
            const double a,
            const Vector2d & v )
```

scalar ∗ operator

5.2.2.10 operator∗() [4/6]

```
topology::Vector2d topology::operator* (
            const int a,
            const Vector2d & v )
```

scalar ∗ operator

5.2.2.11 operator*() [5/6]

```
topology::Vector2d topology::operator* (
            const Vector2d & v,
            const double a )
```

scalar * operator

5.2.2.12 operator*() [6/6]

```
topology::Vector2d topology::operator* (
            const Vector2d & v,
            const int a )
```

scalar * operator

5.2.2.13 operator+() [1/2]

```
topology::angle2d topology::operator+ (
            const angle2d & v,
            const angle2d & a )
```

Addition operator.

5.2.2.14 operator+() [2/2]

```
topology::Vector2d topology::operator+ (
            const Vector2d & v,
            const Vector2d & w )
```

- operator

5.2.2.15 operator-() [1/2]

```
topology::angle2d topology::operator- (
            const angle2d & v,
            const angle2d & w )
```

Subtraction operator.

5.2.2.16 operator-() [2/2]

```
topology::Vector2d topology::operator- (
            const Vector2d & v,
            const Vector2d & w )
```

- operator

5.2.2.17 operator/() [1/3]

```
topology::angle2d topology::operator/ (
            const angle2d & v,
            const double a )
```

Division operator (double)

5.2.2.18 operator/() [2/3]

```
topology::Vector2d topology::operator/ (
            const Vector2d & v,
            const double a )
```

scalar / operator

5.2.2.19 operator/() [3/3]

```
topology::Vector2d topology::operator/ (
            const Vector2d & v,
            const int a )
```

scalar / operator

5.2.2.20 orthogonal_projection()

```
double topology::orthogonal_projection (
            Vector2d v,
            Vector2d w )  [inline]
```

Parallel projection.

5.2.2.21 orthospin()

```
Vector2d topology::orthospin (
            double theta )  [inline]
```

Returns an orthogonal spin vector, i.e. a unit vector rotated 90^° counterclockwise from the vector of spin(theta)

5.2.2.22 parallel_projection()

```
double topology::parallel_projection (
            Vector2d v,
            Vector2d w )  [inline]
```

Parallel projection.

5.2.2.23 periodic_distance() [1/2]

```
double topology::periodic_distance (
            const Vector2d & v,
            const Vector2d & w,
            const double & L )  [inline]
```

Returns distance |w - v| considering periodic boundaries (square box, length L). Taking sqrt takes more time than returning the squared quantity by dist_periodic_squared.

5.2.2.24 periodic_distance() [2/2]

```
double topology::periodic_distance (
            const Vector2d & v,
            const Vector2d & w,
            const Vector2d & L )  [inline]
```

Returns squared distance |w - v| considering periodic boundaries (rectangular box, widths stored in L). Taking sqrt takes more time than returning the squared quantity by dist_periodic_squared.

5.2.2.25 periodic_distance_squared() [1/2]

```
double topology::periodic_distance_squared (
            const Vector2d & v,
            const Vector2d & w,
            const double & L )
```

Returns squared distance |w - v|^2 considering periodic boundaries (square box, length L). Squared function faster to calculate.

5.2.2.26 periodic_distance_squared() [2/2]

```
double topology::periodic_distance_squared (
            const Vector2d & v,
            const Vector2d & w,
            const Vector2d & L )
```

Returns squared distance |w - v|^2 considering periodic boundaries (rectangular box, widths stored in L). Squared function faster to calculate.

5.2.2.27 periodic_distance_vector() [1/2]

```
topology::Vector2d topology::periodic_distance_vector (
            const Vector2d & v,
            const Vector2d & w,
            const double & L )
```

Returns distance vector w - v considering periodic boundaries (square box, length L).

5.2.2.28 periodic_distance_vector() [2/2]

```
topology::Vector2d topology::periodic_distance_vector (
            const Vector2d & v,
            const Vector2d & w,
            const Vector2d & L )
```

Returns distance vector w - v considering periodic boundaries (rectangular box, widths stored in L).

5.2.2.29 print_matlab()

```
void topology::print_matlab (
            const std::vector< Vector2d > & v,
            std::string name,
            std::ostream & outfile )
```

Prints a list of vectors to in matlab-readable form.

5.2.2.30 qvalues_within_radius()

```
std::vector< topology::Vector2d > topology::qvalues_within_radius (
            double qmin,
            double qmax,
            topology::Vector2d gridseps,
            int qsamps_per_bin )
```

Creates a std::vector containing qsamps_per_bin 2D-vectors that lie on a grid with modulus between qmin and qmax. No vector appears double.

5.2.2.31 random_gaussian_vector()

```
topology::Vector2d topology::random_gaussian_vector (
            const double & sigma_squared )
```

Returns a random vector with Gaussian distribution in each component.

5.2.2.32 random_vector() [1/2]

```
topology::Vector2d topology::random_vector (
            const Vector2d & maxima )
```

Returns a random vector within a volume [0,maxima].

5.2.2.33 random_vector() [2/2]

```
topology::Vector2d topology::random_vector (
            const Vector2d & minima,
            const Vector2d & maxima )
```

Returns a random vector within a volume [minima, maxima].

5.2.2.34 random_vector_first_quadrant()

```
topology::Vector2d topology::random_vector_first_quadrant (
            const double length )
```

Returns a random vector in the first quadrant.

5.2.2.35 random_vector_sector()

```
topology::Vector2d topology::random_vector_sector (
            const double length,
            const double thetamin,
            const double thetamax )
```

Returns a random vector within a sector given by thetamin, thetamax.

5.2.2.36 random_velocity()

```
topology::Vector2d topology::random_velocity (
            const double r )
```

Returns a random vector on a sphere surface of radius r.

5.2.2.37 rotate()

```
Vector2d topology::rotate (
            Vector2d v,
            double theta )  [inline]
```

Rotates a vector.

5.2.2.38 rotate_orthogonal()

```
Vector2d topology::rotate_orthogonal (
            Vector2d v )  [inline]
```

Rotates 90 degrees.

5.2.2.39 spin()

```
Vector2d topology::spin (
            double theta )  [inline]
```

Returns a spin vector, i.e. unit vector, with given angle to x-axis.

5.2.2.40 vector_from_angle() [1/2]

```
Vector2d topology::vector_from_angle (
            const double angle )  [inline]
```

Returns a vector of unit length and orientation angle.

5.2.2.41 vector_from_angle() [2/2]

```
topology::Vector2d topology::vector_from_angle (
            const double angle,
            const double r )
```

Returns a vector of length r and orientation angle.

5.2.2.42 vector_on_squarelattice()

```
topology::Vector2d topology::vector_on_squarelattice (
            int index,
            int Nx,
            int Ny,
            double spacing )
```

Index-dependent position vector for spin at index, square lattice.

5.2.2.43 vector_on_trigonallattice()

```
topology::Vector2d topology::vector_on_trigonallattice (
            int index,
            int Nx,
            int Ny,
            double spacing )
```

Index-dependent position vector for spin at index, trigonal lattice.

Chapter 6

Class Documentation

6.1 topology::angle2d Class Reference

A (double-valued) angle in 2d space with the possibility of identifying it with its corresponding Vector2d unit vector. In the current state of the simulation, this is redundant.

`#include <topology.h>`

Public Member Functions

- angle2d ()
  Constructor without argument.
- angle2d (const double &x)
  Constructor from a double.
- angle2d (const angle2d &w)
  Copy constructor.
- operator double () const
- operator Vector2d () const
  Conversion to Vector2d - creates a unit vector with angle theta_ to x-axis.
- Vector2d spin ()
  Returns a spin vector, i.e. unit vector, with given angle to x-axis.
- Vector2d orthospin ()
  Returns an orthogonal spin vector, i.e. a unit vector rotated 90^ °counterclockwise from the vector of spin(theta)
- void boundary ()
  Resets the angle to be within (-pi, pi].
- angle2d & operator+= (const double &a)
  Addition by double.
- angle2d & operator-= (const double &a)
  Subtraction of double.
- angle2d & operator*= (const double a)
  Multiplication by double.
- angle2d & operator/= (const double a)
  Division by double.
- angle2d operator- () const
  Unary additive inversion.

Protected Attributes

- double theta_

  Angle theta, to be interpreted as an angle with respect to the x-axis in a 2d xy-plane.

6.1.1 Detailed Description

A (double-valued) angle in 2d space with the possibility of identifying it with its corresponding Vector2d unit vector. In the current state of the simulation, this is redundant.

Mostly incomplete and unnecessary. The idea was to have a specialized double-like class that can easily be converted to Vector2d and back for simplified calcultion. But the spin and orthospin functions defined for double -> Vector2d actually do the trick perfectly fine. Leaving this here for someone feeling a bit freaky.

Author

Thomas Bissinger

Date

Created: a somewhat uneventful weekend in late 2019

Last Updated: 2023-08-01

6.1.2 Constructor & Destructor Documentation

6.1.2.1 angle2d() [1/3]

```
topology::angle2d::angle2d ( )  [inline]
```

Constructor without argument.

6.1.2.2 angle2d() [2/3]

```
topology::angle2d::angle2d (
            const double & x )
```

Constructor from a double.

6.1.2.3 angle2d() [3/3]

```
topology::angle2d::angle2d (
            const angle2d & w )
```

Copy constructor.

Conversion to double.

6.1.3 Member Function Documentation

6.1.3.1 boundary()

```
void topology::angle2d::boundary ( )
```

Resets the angle to be within (-pi, pi].

6.1.3.2 operator double()

```
topology::angle2d::operator double ( ) const  [inline]
```

6.1.3.3 operator Vector2d()

```
topology::angle2d::operator Vector2d ( ) const  [inline]
```

Conversion to Vector2d - creates a unit vector with angle theta_ to x-axis.

6.1.3.4 operator*=()

```
topology::angle2d & topology::angle2d::operator*= (
            const double a )
```

Multiplication by double.

6.1.3.5 operator+=()

```
topology::angle2d & topology::angle2d::operator+= (
            const double & a )
```

Addition by double.

6.1.3.6 operator-()

```
topology::angle2d topology::angle2d::operator- ( ) const
```

Unary additive inversion.

6.1.3.7 operator-=()

```
topology::angle2d & topology::angle2d::operator-= (
            const double & a )
```

Subtraction of double.

6.1.3.8 operator/=()

```
topology::angle2d & topology::angle2d::operator/= (
            const double a )
```

Division by double.

6.1.3.9 orthospin()

```
Vector2d topology::angle2d::orthospin ( )  [inline]
```

Returns an orthogonal spin vector, i.e. a unit vector rotated 90^ ° counterclockwise from the vector of spin(theta)

6.1.3.10 spin()

```
Vector2d topology::angle2d::spin ( )  [inline]
```

Returns a spin vector, i.e. unit vector, with given angle to x-axis.

6.1.4 Member Data Documentation

6.1.4.1 theta_

```
double topology::angle2d::theta_  [protected]
```

Angle theta, to be interpreted as an angle with respect to the x-axis in a 2d xy-plane.

The documentation for this class was generated from the following files:

- topology.h
- topology.cpp

6.2 group Class Reference

A group of polar particles. Stores vectors with particle positions, velocities, spin orientations and spin rotation velocity, as well as further group properties.

```
#include <group.h>
```

Public Member Functions

- group ()
- group (const parameters &par)

  Constructor from values stored in parameters. Only sets simulation parameters, does not initialize particle data.
- group (const int N, const std::string group_type)

  Reduced constructor, useful for time derivative group.
- void clear ()

  Clears particles and partition.
- void initialize (const parameters &par)

  Initializes particle data for the group based on parameters given.
- void initialize_random (double kbT=0)

  Initializes the mobile group with random particle positions and fills the partition.
- void randomize_particles (double kbT=0)

  Sets particles to random values.
- void mom_to_zero ()

  Sets momenta to zero by shifts.
- void r_to_squarelattice ()

  Sets positions to square lattice.
- void r_to_trigonallattice ()

  Sets positions to trigonal lattice. CAREFUL! Trigonal lattice does not fit well into square box.
- void r_to_lattice ()

  Sets positions to lattice. Decides which lattice depending on lattice_type_ member variable.
- void initialize_zero ()

  Sets all particles to zero.
- void fill_partition ()

  Fills, i.e. computes the partition.
- void read_from_snapshot (std::string snapshotname)

  Reads coordinates and momenta from file snapshotname.
- void scale_from_subgroup (const group &G)

  Takes a subgroup (smaller group) and scales it up to the correct size of the group by copying.
- void scale_from_subgroup (std::string snapshotname)

  Reads a subgroup (smaller group) from a file and scales it up to the correct size of the group by copying.
- void print_group (std::ofstream &outputfile) const

  Prints the entire group to the outputfile.
- void print_r (std::ofstream &outputfile) const

  Prints only position coordinates of group to the outputfile.
- int get_N () const

  Returns number of particles.
- int size () const

  Same as get_N()
- int get_sqrtN () const

  Returns sqrt of number of particles.
- topology::Vector2d get_L () const

  Returns simulation box size.
- double get_boxsize () const

  Returns smallest box length.
- double get_volume () const

  Returns volume.
- double get_density () const

  Returns density.

- double get_I () const

  Returns member variable I_ (spin inertia)
- double get_J () const

  Returns member variable J_ (spin coupling strength)
- double get_m () const

  Returns member variable m_ (particle mass)
- double get_cutoff () const

  Returns member variable cutoff_ (interaction cutoff length)
- double get_vm_v () const

  Returns member variable vm_v_ (Vicsek model velocity)
- double get_vm_eta () const

  Returns member variable vm_eta_ (Vicsek model noise strength)
- std::string get_group_type () const

  Returns member variable group_type_ (type of group)
- std::vector< double > get_theta () const

  Returns member vector theta_ (spin angles). Length N.
- std::vector< double > get_w () const

  Returns member vector w_ (spin momenta). Length N.
- std::vector< topology::Vector2d > get_r () const

  Returns member vector r_ (positions). Length N.
- std::vector< topology::Vector2d > get_p () const

  Returns member vector p_ (linear momenta). Length N.
- std::vector< double > get_coord () const

  Returns vector of all coordinates (angles theta_ and poitions r_, length 3N)
- std::vector< double > get_mom () const

  Returns vector of all momenta (spin momenta w_ and linear momenta p_, length 3N)
- double get_theta (int i) const

  Returns spin angle theta_[i] of particle i.
- double get_w (int i) const

  Returns spin momentum w_[i] of particle i.
- topology::Vector2d get_r (int i) const

  Returns position r_[i] of particle i.
- topology::Vector2d get_p (int i) const

  Returns linear momentum p_[i] of particle i.
- double J_pot (double dist) const

  Returns spin interaction potential (distance-dependence)
- double U_pot (double dist) const

  Returns spatial interaction potential.
- double J_pot_prime (double dist) const

  Returns derivative of spin interaction potential (distance-dependence)
- double U_pot_prime (double dist) const

  Returns derivative of spatial interaction potential.
- double J_pot_primeprime (double dist) const

  Returns second derivative of spin interaction potential (distance-dependence)
- double U_pot_primeprime (double dist) const

  Returns second derivative of spatial interaction potential.
- std::vector< int > get_neighbors (int i, std::string cellselect, std::vector< double > &distances) const

  Returns indices of neighbors of the particle. Selection of cells possible.
- std::vector< int > get_neighbors (int i, std::vector< double > &distances) const

  Returns indices of neighbors of the particle. Uses the member variable nb_rule_ to determine which cells to select.
- void generate_neighbor_list ()

Fills the variables nb_index_, nb_first_, nb_dist_ according to the current neighborhood situation. Strongly recommended for fmxy model, recommended for xy and fvm model with small system sizes.

- double theta_diff (int i, int j) const

  Difference in angles of two different particles. θ_{ij} in Bore paper.
- double periodic_distance_squared (int i, int j) const

  Returns squared distance between particle i and j considering periodic boundaries (square box). Squared function faster to calculate.
- double periodic_distance (int i, int j) const

  Returns distance between particle i and j considering periodic boundaries (square box). Taking sqrt takes more time than returning the squared quantity by dist_periodic_squared.
- topology::Vector2d periodic_distance_vector (int i, int j) const

  Returns distance vector between particle i and j considering periodic boundaries (square box).
- void set_theta (double theta, int i)

  Gives theta_ of particle i a specified value.
- void set_w (double w, int i)

  Gives w_ of particle i a specified value.
- void set_r (topology::Vector2d r, int i)

  Gives r_ of particle i a specified value.
- void set_rx (double x, int i)

  Gives x-component of r_ of particle i a specified value.
- void set_ry (double y, int i)

  Gives y-component of r_ of particle i a specified value.
- void set_p (topology::Vector2d p, int i)

  Gives p_ of particle i a specified value.
- void set_px (double px, int i)

  Gives x-component of p_ of particle i a specified value.
- void set_py (double py, int i)

  Gives y-component of p_ of particle i a specified value.
- void set_particle (double theta, double w, topology::Vector2d r, topology::Vector2d p, int i)

  Sets all values theta_, w_, r_, p_ of particle i to the designated values.
- void set_all_w (double w)

  Sets all w to given value (useful for setting T = 0)
- void set_all_p (topology::Vector2d p)

  Sets all p to given value (useful for setting T = 0)
- void set_all_theta (double theta)

  Sets all theta to given value (useful for perfect spin alignment)
- void set_temperature (double kT, int i)

  Randomizes momenta to be in agreement with given kT of particle i.
- void set_temperature (double kT)

  Randomizes momenta to be in agreement with given kT of all particles.
- void set_temperature_p (double kT, int i)

  Randomizes linear momenta to be in agreement with given kT of particle i.
- void set_temperature_p (double kT)

  Randomizes linear momenta to be in agreement with given kT of all particles.
- void set_temperature_w (double kT, int i)

  Randomizes spin momenta to be in agreement with given kT of particle i.
- void set_temperature_w (double kT)

  Randomizes spin momenta to be in agreement with given kT of all particles.
- void scale_mom (double a)

  Scales all momenta (w_, p_) by a factor a.
- void add_to_theta (const std::vector< double > &theta, double factor=1)

Adds vector of theta to theta_, scales by factor. Vector must have length of at least N_. Vector of doubles.

- void add_to_r (const std::vector< topology::Vector2d > &r, double factor=1)

  *Adds vector of r to r_, scales by factor. Vector must have length of at least 2 * N_. Vector of doubles.*
- void add_to_r (const std::vector< double > &r, double factor=1)

  Adds vector of r to r_, scales by factor. Vector must have length of at least N_. Vecot of topology::Vector2d.
- void add_to_coord (const std::vector< double > &coord, double factor=1)

  Adds vector of coord to all coordinates (r and theta, if available). Vector must have length of at least N_. Vector of doubles. Does nothing for Vicsek type models.
- void add_to_coord_inertialscaling (const std::vector< double > &coord, double factor=1)

  Adds vector of coord to all coordinates (r and theta, if available). Scales by factor and inverse intertia (1/m or 1/I, respectively). Vector must have length of at least N_. Vector of doubles. Does nothing for Vicsek type models.
- void add_to_w (const std::vector< double > &w, double factor=1)

  Adds vector of w to w_, scales by factor. Vector must have length of at least N_. Vector of doubles.
- void add_to_p (const std::vector< topology::Vector2d > &p, double factor=1)

  Adds vector of p to p_, scales by factor. Vector must have length of at least N_. Vector of doubles.
- void add_to_p (const std::vector< double > &p, double factor=1)

  Adds vector of p to p_, scales by factor. Vector must have length of at least N_. Vector of topology::Vector2d.
- void add_to_mom (const std::vector< double > &mom, double factor=1)

  Adds vector of mom to momenta (w and p, if available), scales by factor. Vector must have length of at least N_. Vector of doubles. Does nothing for Vicsek type models.
- void add_random_angle (double angmax)

  Adds a uniformly distributed angle in (-angmax, angmax) to each particle's theta_.
- void add_random_displacement (double rmax)

  Adds a uniformly distributed displacement (-rmax, rmax)^2 to each particle's r_.
- void stream_along_spin (double v)

  *Streams along spin, r_new = r + v * spin(theta)*
- void set_theta_to_interval ()

  Sets theta_ values to interval (-pi, pi)
- void set_r_to_pbc ()

  Sets particle positions according to boundary conditions.
- double sum_w () const

  Returns sum over omega, basically N_<w>. Extensive.
- double sum_w_squared () const

  Omega squared, basically N_<w^2>. Proportional to kinetic energy. Extensive.
- double sum_w_4 () const

  Omega to the fourth power, basically N_<w^4>. Proportional to kinetic energy. Extensive.
- double sum_theta () const

  Returns sum over all theta, basically N_<theta>. Extensive. Probably pointless.
- topology::Vector2d sum_s () const

  Magnetization. Basically N_<s>. Extensive.
- double sum_s_squared () const

  Magnetization squared. Basically N_<s>^2. Extensive.
- double sum_s_4 () const

  Magnetization to the fourth power. Basically N_<s>^4. Extensive.
- topology::Vector2d sum_p () const

  Total momentum. Basically N_<p>. Extensive.
- double sum_p_squared () const

  Sum over momentum squared. Basically N_<p^2>. Extensive.
- double sum_p_4 () const

  Sum over momentum to the fourth power. Basically N_<p^4>. Extensive.
- double sum_e_squared () const

Total energy squared, basically N_<e_i^2>. Extensive.

- double sum_ekin_squared () const

  Kinetic energy squared, basically N_<e_{i,kin}^2>. Extensive.
- double sum_eint_squared () const

  Interaction energy squared, basically N_<e_{i,int}^2>. Extensive.
- double binder_cumulant () const

  Binder cumulant. 1 - <s^4> / (3 <s^2>). Intensive.
- double calc_interaction_energy () const

  System interaction energy. Extensive.
- double calc_interaction_energy (int i) const

  Interaction energy of particle i.
- double calc_kinetic_energy () const

  Returns system energy. Extensive.
- double calc_kinetic_energy (int i) const

  Returns kinetic energy of particle i.
- double calc_energy () const

  Returns system energy. Extensive.
- double calc_energy (int i) const

  Energy of particle i. Extensive.
- double calc_temperature () const

  Returns temperature. Careful, this function returns ((<p^2>-.
- double calc_temperature_w () const

  Returns spin angular momentum temperature.
- double calc_temperature_p () const

  Returns linear momentum temperature.
- std::vector< int > plaquette (int i) const

  Return the plaquette the particle i belongs to. i is in the lower left corner. Only works for lattice-based models.
- double calc_vorticity (int index) const

  Returns vorticity along the plaquette at index.
- double calc_vortexdensity_unsigned () const

  Returns the unsigned vortex density (i.e. number of vortices divided by box area).
- double calc_vortexdensity_signed () const

  Returns the signed vortex density (i.e. number of positive vortices minus number of negative vortices divided by box area).
- double calc_space_angular_mom () const

  Returns total spatial angular momentum of particles.
- double calc_space_angular_mom (int i) const

  Returns spatial angular momentum of the particle with index i.
- double calc_neighbor_mean (double te_pow, double r_pow, double cos_pow, double sin_pow, double J_pow, double Up_pow, double Upp_pow) const

  Calculates the mean over nearest neighbors.
- std::vector< double > calc_helicity (double beta) const

  Calculates the helicity modulus and auxiliary quantities. Output is a vector with entries (Upsilon,H_x,H_y,I_x,I_y)
- std::complex< double > calc_eiqr (const topology::Vector2d q, int i) const

  Calculates e^(i q r_i) for particle i.
- std::complex< double > calc_mxq (const topology::Vector2d q, double Mx_0) const

  Calculates $m_{x,q}$ (see Bissinger PhD thesis) and calc_fieldfluct .
- std::complex< double > calc_myq (const topology::Vector2d q, double My_0) const

  Calculates $m_{y,q}$ (see Bissinger PhD thesis) and calc_fieldfluct .
- std::complex< double > calc_wq (const topology::Vector2d q, double W_0) const

  Calculates w_{q} (see Bissinger PhD thesis) and calc_fieldfluct .

- std::complex< double > calc_eq (const topology::Vector2d q, double E_0) const

  Calculates e_{q} (see Bissinger PhD thesis) and calc_fieldfluct .
- std::complex< double > calc_teq (const topology::Vector2d q, double Te_0) const

  Calculates $theta_{q}$ (see Bissinger PhD thesis) and calc_fieldfluct .
- std::complex< double > calc_rq (const topology::Vector2d q) const

  Calculates rho_{q} (see Bissinger PhD thesis) and calc_fieldfluct .
- std::vector< std::complex< double > > calc_jq (const topology::Vector2d q, topology::Vector2d J_0) const

  Calculates j_{q} (see Bissinger PhD thesis) and calc_fieldfluct .
- std::complex< double > calc_jqpar (const topology::Vector2d q, topology::Vector2d J_0) const

  Calculates $j_{q,L}$ (see Bissinger PhD thesis) and calc_fieldfluct .
- std::complex< double > calc_jqperp (const topology::Vector2d q, topology::Vector2d J_0) const

  Calculates $j_{q,T}$ (see Bissinger PhD thesis) and calc_fieldfluct .
- std::complex< double > calc_lq (const topology::Vector2d q, double L_0) const

  Calculates l_{q} (see Bissinger PhD thesis) and calc_fieldfluct .
- double calc_fieldfluct_average (std::string fluctname, topology::Vector2d q=0) const

  Calculates the average of the field fluctuation fluctname. (e.g. for "wq" this returns sum omega_i.)
- double calc_one_particle_density (int index, std::string fluctname, topology::Vector2d q=0) const

  Calculates the one-particle density associated with the field fluctuation fluctname. (e.g. for "wq" this returns omega_index.)
- std::vector< std::complex< double > > calc_fieldfluct (const std::vector< topology::Vector2d > qvals, std::string fluctname) const

  Calculates the field fluctuation for the quantity specified in fluctname.
- std::vector< std::complex< double > > calc_fieldfluct_convolution (const std::vector< topology::Vector2d > qvals, std::string fluctname_1, std::string fluctname_2) const

  Calculates the field fluctuation for the quantity specified in fluctname.
- topology::Vector2d calc_tau () const

  Calculates τ, as defined for the xy model. NOT CORRECT FOR THE MOBILE CASE.
- topology::Vector2d calc_je () const

  Calculates j^e, as defined for the xy model. NOT CORRECT FOR THE MOBILE CASE.
- topology::Vector2d calc_current (std::string currentname) const

  Calculates the current for the quantity specified in currentname. currentname = {"tau","je"}. NOT CORRECT FOR THE MOBILE CASE.
- std::vector< double > calc_SCF_S_individual (const int index, const std::vector< double > rbin, std::vector< int > &counts) const

  Calculates the static spin correlation function for a specific particle at index.
- std::vector< double > calc_SCF_S_oriented_individual (const int index, const std::vector< double > rbin, const double &orientation_angle, std::vector< int > &counts) const

  Calculates the static oriented spin correlation function for a specific particle at index.
- std::vector< double > calc_SCF_g_individual (const int index, const std::vector< double > rbin) const

  Calculates the pair distribution function g(r) for a specific particle at index.
- std::vector< double > calc_SCF_g (const std::vector< double > rbin, int number_of_points) const

  Calculates the overall pair distribution function g(r)
- std::vector< double > calc_SCF_anglediff_individual (const int index, const std::vector< double > rbin, std::vector< int > &counts) const

  Calculates the static angle difference correlation function for a specific particle at index.
- std::vector< double > calc_SCF_E_individual (const int index, const std::vector< double > rbin, std::vector< int > &counts) const

  Calculates the static total energy correlation function for a specific particle at index.
- std::vector< double > calc_SCF_Ekin_individual (const int index, const std::vector< double > rbin, std::vector< int > &counts) const

  Calculates the static kinetic energy correlation function for a specific particle at index.

- std::vector< double > calc_SCF_Eint_individual (const int index, const std::vector< double > rbin, std::vector< int > &counts) const

  Calculates the static interaction energy correlation function for a specific particle at index.
- std::vector< double > calc_SCF_P_individual (const int index, const std::vector< double > rbin, std::vector< int > &counts) const

  Calculates the static momentum correlation function for a specific particle at index.
- std::vector< double > calc_SCF_W_individual (const int index, const std::vector< double > rbin, std::vector< int > &counts) const

  Calculates the static spin momentum correlation function for a specific particle at index.
- std::vector< double > calc_SCF_averaged (const std::vector< double > rbin, int number_of_points, const std::string name) const

  Calculates the static correlation function specified by name for number_of_points many random particles.
- double calc_ACF_S (const group &G_initial) const

  Calculates the spin autocorrelation-function averaged over all indices.
- double calc_ACF_anglediff (const group &G_initial) const

  Calculates the angle difference autocorrelation-function averaged over all indices.
- double calc_ACF_sp (const group &G_initial, const std::string name) const

  Calculates the single-particle autocorrelation-function for some quantity specified by name.
- double calc_ACF_q0 (const group &G_initial, const std::string name) const

  Calculates the (q=0)-autocorrelation-function for some quantity specified by name.
- std::vector< std::complex< double > > calc_TCF (const group &G_initial, std::vector< topology::Vector2d > qvals, std::string fluctname_initial, std::string fluctname_current) const

  Calculates time-correlation function between two different groups. Fluctuation names must be specified.
- std::vector< double > time_derivative_theta () const

  Returns theta (spin angle) time derivative (splitting useful for leapfrog)
- std::vector< double > time_derivative_w () const

  Returns omega (spin momentum) time derivative (splitting useful for leapfrog)
- std::vector< double > time_derivative_r () const

  Returns r (particle position) time derivative. First N_ entries are x direction, N_+1 to 2N_ is y direction (splitting useful for leapfrog)
- std::vector< double > time_derivative_p () const

  Returns p (linear momentum) time derivative. First N_ entries are x direction, N_+1 to 2N_ is y direction (splitting useful for leapfrog)
- std::vector< double > time_derivative_coord () const

  Returns coordinate time derivative (first N_ entries are theta, then r_x, then r_y).
- std::vector< double > time_derivative_mom () const

  Returns momenta time derivative (first N_ entries are omega, then p_x, then p_y).
- group time_derivative () const

  Returns time derivative of the entire group.
- std::vector< double > coord_diff (const group &G) const

  Returns coordinate difference between this group and another one, with proper care of boundaries. First N_ entries are theta, then r_x, then r_y.
- void accumulative_MSD (std::vector< double > &MSD, const group &last_G) const

  Same as coord_diff, but adding the difference to an MSD vector.
- group & operator+= (const group &G)

  Adds particle entries (used for adding time derivatives and such).
- group & operator*= (const double a)

  Multiplies particles by constant (used for adding time derivatives and such).

Protected Attributes

- std::string group_type_

  Type of the group. Can be ¨xy¨ for the XY model, ¨mxy¨ for the mobile XY model, ¨fmxy¨ for a mobile XY model frozen in place, ¨vm¨ for the Vicsek model and ¨fvm¨ for the frozen (static) Vicsek model.
- int N_

  Size of the group.
- int sqrtN_

  Square root of the group size (often useful).
- topology::Vector2d L_

  Size of the box (some functions only defined for square boxes yet).
- double I_ = 1

  Spin inertia.
- double m_ = 1

  Mass.
- double J_ = 1

  Nearest neighbor interaction strength.
- double U_ = 1

  Spatial repulsion interaction strength.
- double cutoff_ = 1

  Interaction cutoff radius.
- std::vector< topology::Vector2d > r_

  Particle positions in the group.
- std::vector< topology::Vector2d > p_

  Particle momenta/velocities in the group.
- std::vector< double > theta_

  Particle spin angles in the group.
- std::vector< double > w_

  Particle spin momenta in the group.
- partition partition_

  Partition (cell list) for neighborhood interaction.
- std::string nb_rule_

  Neighbor calculation rule. Possible values: ¨bruteforce¨, ¨all¨, ¨ur¨ for full, (partition with) all and (partition with) upper right neighbors.
- double nb_mult_factor_

  If neighbor rule leads to double counting, this factor has to be .5, otherwise 1.
- neighbor_list * nb_list_

  neighbor-list
- char lattice_type_

  lattice type. (type 's': square, type 't': trigonal, type 'n': none (mxy model etc))
- double vm_eta_

  Vicsek model parameter eta: Angle for random noise.
- double vm_v_

  Vicsek model parameter v: Streaming velocity.

6.2.1 Detailed Description

A group of polar particles. Stores vectors with particle positions, velocities, spin orientations and spin rotation velocity, as well as further group properties.

Contains the main data to be manipulated in a simulation of the MXY model and the other models.

Functionalities

- Constructors
- Functions for clearing and initialization as well as handling the partition member variable
- Operations for reading and copying from other groups
- Printing operations
- Simple information extraction
- Simple arithmetic operations on individual particles and their properties (differences, scaling, setting to new values etc.)
- Calculation of physical properties (kinetic temperature, energz, momentum, helicity etc.)
- Calculation of field fluctuations
- Calculation for spatial and temporal correlation functions as well as correlations in reciprocal space
- Calculation of time derivatives

Author

Thomas Bissinger

Date

Created: 2020-02-29 (full rewrite)

Last Updated: 2023-07-23

6.2.2 Constructor & Destructor Documentation

6.2.2.1 group() [1/3]

```
group::group ( )  [inline]
```

6.2.2.2 group() [2/3]

```
group::group (
            const parameters & par )
```

Constructor from values stored in parameters. Only sets simulation parameters, does not initialize particle data.

6.2.2.3 group() [3/3]

```
group::group (
            const int N,
            const std::string group_type )
```

Reduced constructor, useful for time derivative group.

6.2.3 Member Function Documentation

6.2.3.1 accumulative_MSD()

```
void group::accumulative_MSD (
            std::vector< double > & MSD,
            const group & last_G ) const
```

Same as coord_diff, but adding the difference to an MSD vector.

6.2.3.2 add_random_angle()

```
void group::add_random_angle (
            double angmax )
```

Adds a uniformly distributed angle in (-angmax, angmax) to each particle's theta_.

6.2.3.3 add_random_displacement()

```
void group::add_random_displacement (
            double rmax )
```

Adds a uniformly distributed displacement (-rmax, rmax)^2 to each particle's r_.

6.2.3.4 add_to_coord()

```
void group::add_to_coord (
            const std::vector< double > & coord,
            double factor = 1 )
```

Adds vector of coord to all coordinates (r and theta, if available). Vector must have length of at least N_. Vector of doubles. Does nothing for Vicsek type models.

6.2.3.5 add_to_coord_inertialscaling()

```
void group::add_to_coord_inertialscaling (
            const std::vector< double > & coord,
            double factor = 1 )
```

Adds vector of coord to all coordinates (r and theta, if available). Scales by factor and inverse intertia (1/m or 1/I, respectively). Vector must have length of at least N_. Vector of doubles. Does nothing for Vicsek type models.

6.2.3.6 add_to_mom()

```
void group::add_to_mom (
            const std::vector< double > & mom,
            double factor = 1 )
```

Adds vector of mom to momenta (w and p, if available), scales by factor. Vector must have length of at least N_. Vector of doubles. Does nothing for Vicsek type models.

6.2.3.7 add_to_p() [1/2]

```
void group::add_to_p (
            const std::vector< double > & p,
            double factor = 1 )
```

Adds vector of p to p_, scales by factor. Vector must have length of at least N_. Vector of topology::Vector2d.

6.2.3.8 add_to_p() [2/2]

```
void group::add_to_p (
            const std::vector< topology::Vector2d > & p,
            double factor = 1 )
```

Adds vector of p to p_, scales by factor. Vector must have length of at least N_. Vector of doubles.

6.2.3.9 add_to_r() [1/2]

```
void group::add_to_r (
            const std::vector< double > & r,
            double factor = 1 )
```

Adds vector of r to r_, scales by factor. Vector must have length of at least N_. Vecot of topology::Vector2d.

6.2.3.10 add_to_r() [2/2]

```
void group::add_to_r (
            const std::vector< topology::Vector2d > & r,
            double factor = 1 )
```

Adds vector of r to r_, scales by factor. Vector must have length of at least $2 * N\_$. Vector of doubles.

6.2.3.11 add_to_theta()

```
void group::add_to_theta (
            const std::vector< double > & theta,
            double factor = 1 )
```

Adds vector of theta to theta_, scales by factor. Vector must have length of at least N_. Vector of doubles.

6.2.3.12 add_to_w()

```
void group::add_to_w (
            const std::vector< double > & w,
            double factor = 1 )
```

Adds vector of w to w_, scales by factor. Vector must have length of at least N_. Vector of doubles.

6.2.3.13 binder_cumulant()

```
double group::binder_cumulant ( ) const  [inline]
```

Binder cumulant. 1 - <s^4> / (3 <s^2>). Intensive.

6.2.3.14 calc_ACF_anglediff()

```
double group::calc_ACF_anglediff (
            const group & G_initial ) const
```

Calculates the angle difference autocorrelation-function averaged over all indices.

Computes $\frac{1}{N}\sum_{i=1}^{N}(\theta_i^{\mathrm{G\_initial}} - \theta_i^{\mathrm{G}})^2$, where $\theta_i^{\mathrm{G\_initial}}$ is the i-th spin angle in group G_initial and θ_i^{G} is the i-th spin angle in the current instance of group for which calc_ACF_anglediff is called.

6.2.3.15 calc_ACF_q0()

```
double group::calc_ACF_q0 (
            const group & G_initial,
            const std::string name ) const
```

Calculates the (q=0)-autocorrelation-function for some quantity specified by name.

Computes $\frac{1}{N}\sum_{i=1}^{N}\left\langle a_i^{\mathrm{G\_initial}} \cdot a_i^{\mathrm{G}}\right\rangle$, where a_i is a quantity defined for each particle individually. $a_i^{\mathrm{G\_initial}}$ is then the quantity associated to the i-th particle in the group G_initial, while a_i^{G} the the quantity associated to the i-th particle in the current instance of group for which calc_ACF_sp is called.

Improvement possibilities

- *Case handling*. Case handling for different names follows syntactic simplicity. One could rewrite the code to drastically reduce calls to if-cases.

Parameters

in	*G_initial*	group with which the correlation is to be compared. In most cases, this is the simulated group at a previous time.

Parameters

in	*name*	Used to choose a specific correlation function to be calcualted. Options are

Table 6.1 Values of name

name value	Operation
¨S¨	$a_i = \mathbf{s}_i$, same as calc_ACF_S
¨Sx¨	$a_i = s_{i,x}$, x-component of spin
¨anglediff¨	$a_i = \theta_i$, averges over $(\theta_i^{\mathrm{G\_initial}} - \theta_i^{\mathrm{G}})^2$. See calc_ACF_anglediff
¨Spar¨	$a_i = s_{i,\parallel}$, that is spins oriented along the total magnetization angle
¨Sperp¨	$a_i = s_{i,\perp}$, that is spins oriented perpendicular to the total magnetization angle
¨P¨	$a_i = \mathbf{p}_i$, linear momentum
¨Px¨	$a_i = \mathbf{p}_{i,x}$, x-component of linear momentum
¨Py¨	$a_i = \mathbf{p}_{i,y}$, y-component of linear momentum
¨Ppar¨	$a_i = \mathbf{p}_{i,\parallel}$, component of linear momentum parallel to the total magnetization
¨Pperp¨	$a_i = \mathbf{p}_{i,\perp}$, component of linear momentum perpendicular to the total magnetization
¨W¨	$a_i = \omega_i$, spin momentum
¨E¨	$a_i = e_i$, energy per particle
¨Ekin¨	$a_i = e_{\mathrm{kin},i}$, kinetic energy per particle
¨Eint¨	$a_i = e_{\mathrm{int},i}$, interaction energy per particle
¨MSD¨	Averages over $\left(\mathbf{r}_i^{\mathrm{G}} - \mathbf{r}_i^{\mathrm{G\_initial}}\right)^2$. Careful, does not take periodic boundary into consideration
Other	For any other entry, the return value is set to 0. A warning is printed to std::cerr.

6.2.3.16 calc_ACF_S()

```
double group::calc_ACF_S (
            const group & G_initial ) const
```

Calculates the spin autocorrelation-function averaged over all indices.

Computes $\frac{1}{N}\sum_{i=1}^{N} \mathbf{s}_i^{\mathrm{G}} \cdot \mathbf{s}^{\mathrm{G\_initial}}$, where $\mathbf{s}_i^{\mathrm{G\_initial}}$ is the i-th spin in the group G_initial and $\mathbf{s}_i^{\mathrm{G}}$ is the i-th spin in the current instance of group for which calc_ACF_S is called.

6.2.3.17 calc_ACF_sp()

```
double group::calc_ACF_sp (
            const group & G_initial,
            const std::string name ) const
```

Calculates the single-particle autocorrelation-function for some quantity specified by name.

Computes $\frac{1}{N}\sum_{i=1}^{N}\left\langle a_i^{\mathtt{G\_initial}} \cdot a_i^{\mathrm{G}}\right\rangle$, where a_i is a quantity defined for each particle individually. $a_i^{\mathtt{G\_initial}}$ is then the quantity associated to the i-th particle in the group G_initial, while a_i^{G} the the quantity associated to the i-th particle in the current instance of group for which calc_ACF_sp is called.

Improvement possibilities

- *Case handling*. Case handling for different names follows syntactic simplicity. One could rewrite the code to drastically reduce calls to if-cases.

Parameters

in	*G_initial*	group with which the correlation is to be compared. In most cases, this is the simulated group at a previous time.
in	*name*	Used to choose a specific correlation function to be calcualted. Options are

Table 6.2 Values of name

name value	Operation
¨S¨	$a_i = \mathbf{s}_i$, same as calc_ACF_S
¨Sx¨	$a_i = s_{i,x}$, x-component of spin
¨anglediff¨	$a_i = \theta_i$, averges over $(\theta_i^{\mathtt{G\_initial}} - \theta_i^{\mathrm{G}})^2$. See calc_ACF_anglediff
¨Spar¨	$a_i = s_{i,\parallel}$, that is spins oriented along the total magnetization angle
¨Sperp¨	$a_i = s_{i,\perp}$, that is spins oriented perpendicular to the total magnetization angle
¨P¨	$a_i = \mathbf{p}_i$, linear momentum
¨Px¨	$a_i = \mathbf{p}_{i,x}$, x-component of linear momentum
¨Py¨	$a_i = \mathbf{p}_{i,y}$, y-component of linear momentum
¨Ppar¨	$a_i = \mathbf{p}_{i,\parallel}$, component of linear momentum parallel to the total magnetization
¨Pperp¨	$a_i = \mathbf{p}_{i,\perp}$, component of linear momentum perpendicular to the total magnetization
¨W¨	$a_i = \omega_i$, spin momentum
¨E¨	$a_i = e_i$, energy per particle
¨Ekin¨	$a_i = e_{\mathrm{kin},i}$, kinetic energy per particle
¨Eint¨	$a_i = e_{\mathrm{int},i}$, interaction energy per particle
¨MSD¨	Averages over $\left(\mathbf{r}_i^{\mathrm{G}} - \mathbf{r}_i^{\mathtt{G\_initial}}\right)^2$. Careful, does not take periodic boundary into consideration
Other	For any other entry, the return value is set to 0. A warning is printed to std::cerr.

6.2.3.18 calc_current()

```
topology::Vector2d group::calc_current (
            std::string currentname ) const
```

Calculates the current for the quantity specified in currentname. currentname = {¨tau¨,¨je¨}. NOT CORRECT FOR THE MOBILE CASE.

6.2.3.19 calc_eiqr()

```
std::complex< double > group::calc_eiqr (
            const topology::Vector2d q,
            int i ) const  [inline]
```

Calculates e^(i q r_i) for particle i.

6.2.3.20 calc_energy() [1/2]

```
double group::calc_energy ( ) const  [inline]
```

Returns system energy. Extensive.

6.2.3.21 calc_energy() [2/2]

```
double group::calc_energy (
            int i ) const  [inline]
```

Energy of particle i. Extensive.

6.2.3.22 calc_eq()

```
std::complex< double > group::calc_eq (
            const topology::Vector2d q,
            double E_0 ) const
```

Calculates e_{q} (see Bissinger PhD thesis) and calc_fieldfluct .

6.2.3.23 calc_fieldfluct()

```
group::calc_fieldfluct (
            const std::vector< topology::Vector2d > qvals,
            std::string fluctname ) const
```

Calculates the field fluctuation for the quantity specified in fluctname.

A field fluctuation is a quantity

$$a_{\mathbf{q}} = \frac{1}{\sqrt{N}} \sum_j a_j e^{-i\mathbf{q}\cdot\mathbf{r}_j}$$

for some property a_j carried by each particle.

This function obtains the value of the field fluctuation at all values of $\mathbf{q}$ stored in the vector qvals. The type of the field fluctuation is defined by the value of fluctname.

Parameters

in	*qvals*	Vector containing the wavevectors $\mathbf{q}$
in	*fluctname*	Name of the fluctuation, specifies which one to compute

Table 6.3 Values of fluctname

name value	$a_{\mathbf{q}}$	a_i	Meaning
¨mxq¨	$m_{x,\mathbf{q}}$	$s_{i,x}$	Magnetization in x-direction
¨myq¨	$m_{y,\mathbf{q}}$	$s_{i,y}$	Magnetization in y-direction
¨wq¨	$w_{\mathbf{q}}$	ω_i	Spin angular momentum
¨eq¨	$e_{\mathbf{q}}$	e_i	Energy density
¨teq¨	$\theta_{\mathbf{q}}$	θ_i	Spin angle (not recommended to use)
¨rq¨	$\rho_{\mathbf{q}}$	1	Density
¨lq¨	$l_{\mathbf{q}}$	l_i	Spatial angular momentum (not recommended to use)
¨jparq¨	$j_{\parallel,\mathbf{q}}$	$v_i^{\parallel}$	Longitudinal velocity fluctuation (along $\mathbf{q}$)
¨jparq¨	$j_{\perp,\mathbf{q}}$	$v_i^{\perp}$	Transversal velocity fluctuation (perpendicular to $\mathbf{q}$)
Other	–	–	For any other entry, the return value is set to 0. A warning is printed to std::cerr.

6.2.3.24 calc_fieldfluct_average()

```
double group::calc_fieldfluct_average (
            std::string fluctname,
            topology::Vector2d q = 0 ) const
```

Calculates the average of the field fluctuation fluctname. (e.g. for ¨wq¨ this returns sum omega_i.)

For further details on the variable fluctname, see calc_fieldfluct

6.2.3.25 calc_fieldfluct_convolution()

```
std::vector< std::complex< double > > group::calc_fieldfluct_convolution (
            const std::vector< topology::Vector2d > qvals,
            std::string fluctname_1,
            std::string fluctname_2 ) const
```

Calculates the field fluctuation for the quantity specified in fluctname.

6.2.3.26 calc_helicity()

```
std::vector< double > group::calc_helicity (
            double beta ) const
```

Calculates the helicity modulus and auxiliary quantities. Output is a vector with entries (Upsilon,H_x,H_y,I_x,I_y)

6.2.3.27 calc_interaction_energy() [1/2]

```
double group::calc_interaction_energy ( ) const
```

System interaction energy. Extensive.

6.2.3.28 calc_interaction_energy() [2/2]

```
double group::calc_interaction_energy (
            int i ) const
```

Interaction energy of particle i.

6.2.3.29 calc_je()

```
topology::Vector2d group::calc_je ( ) const
```

Calculates j^e, as defined for the xy model. NOT CORRECT FOR THE MOBILE CASE.

6.2.3.30 calc_jq()

```
std::vector< std::complex< double > > group::calc_jq (
            const topology::Vector2d q,
            topology::Vector2d J_0 ) const
```

Calculates j_{q} (see Bissinger PhD thesis) and calc_fieldfluct .

6.2.3.31 calc_jqpar()

```
std::complex< double > group::calc_jqpar (
            const topology::Vector2d q,
            topology::Vector2d J_0 ) const
```

Calculates $j_{q,L}$ (see Bissinger PhD thesis) and calc_fieldfluct .

6.2.3.32 calc_jqperp()

```
std::complex< double > group::calc_jqperp (
            const topology::Vector2d q,
            topology::Vector2d J_0 ) const
```

Calculates $j_{q,T}$ (see Bissinger PhD thesis) and calc_fieldfluct .

6.2.3.33 calc_kinetic_energy() [1/2]

```
double group::calc_kinetic_energy ( ) const
```

Returns system energy. Extensive.

6.2.3.34 calc_kinetic_energy() [2/2]

```
double group::calc_kinetic_energy (
            int i ) const
```

Returns kinetic energy of particle i.

6.2.3.35 calc_lq()

```
std::complex< double > group::calc_lq (
            const topology::Vector2d q,
            double L_0 ) const
```

Calculates l_{q} (see Bissinger PhD thesis) and calc_fieldfluct .

6.2.3.36 calc_mxq()

```
std::complex< double > group::calc_mxq (
            const topology::Vector2d q,
            double Mx_0 ) const
```

Calculates $m_{x,q}$ (see Bissinger PhD thesis) and calc_fieldfluct .

6.2.3.37 calc_myq()

```
std::complex< double > group::calc_myq (
            const topology::Vector2d q,
            double My_0 ) const
```

Calculates $m_{y,q}$ (see Bissinger PhD thesis) and calc_fieldfluct .

6.2.3.38 calc_neighbor_mean()

```
double group::calc_neighbor_mean (
            double te_pow,
            double r_pow,
            double cos_pow,
            double sin_pow,
            double J_pow,
            double Up_pow,
            double Upp_pow ) const
```

Calculates the mean over nearest neighbors.

For each pair of particles, the function determines ¨theta^te_pow ∗ r^r_pow ∗ cos(theta)^cos_pow ∗ sin(theta)^sin_pow ∗ J(r)^J_pow ∗ U'(r)^Up_pow ∗ U"(r)^Upp_pow¨ and adds all the values up. Here, theta is the spin angle difference between the two particles, r is the distance between the particles. Can be used to calculate transport coefficients or other diagnostics.

6.2.3.39 calc_one_particle_density()

```
double group::calc_one_particle_density (
            int index,
            std::string fluctname,
            topology::Vector2d q = 0 ) const
```

Calculates the one-particle density associated with the field fluctuation fluctname. (e.g. for ¨wq¨ this returns omega_index.)

For further details on the variable fluctname, see calc_fieldfluct

6.2.3.40 calc_rq()

```
std::complex< double > group::calc_rq (
            const topology::Vector2d q ) const
```

Calculates rho_{q} (see Bissinger PhD thesis) and calc_fieldfluct .

6.2.3.41 calc_SCF_anglediff_individual()

```
std::vector< double > group::calc_SCF_anglediff_individual (
            const int index,
            const std::vector< double > rbin,
            std::vector< int > & counts ) const
```

Calculates the static angle difference correlation function for a specific particle at index.

More precisely, obtains <theta_[index]-theta_[j]> for all j in the sample, sums the results and sorts them into bins.

Parameters

in	*index*	Particle index.
in	*rbin*	Bin prescription. Contains bin edges, e.g. rbin = (0,dr,2∗dr,...,rmax-dr,rmax) for an equidistant bin. Particles further from the focal particle than rmax are not considered.
in,out	*counts*	Counts how many particle are found within a bin. As the function is typically repeatedly to sample many particles, this parameter can be updated for each individual call to the function. Must be initialized to zero before the first call.

6.2.3.42 calc_SCF_averaged()

```
std::vector< double > group::calc_SCF_averaged (
            const std::vector< double > rbin,
            int number_of_points,
            const std::string name ) const
```

Calculates the static correlation function specified by name for number_of_points many random particles.

Uses one of the individual SCF calculation functions for number_of_points many randomly chosen particles and averages over the result.

Improvement possibilities

- *Case handling*. Case handling for different names follows syntactic simplicity. One could rewrite the code to drastically reduce calls to if-cases.
- *Information efficiency*. In this function, points within a specific bin are determined. If one calls for this function repeatedly, these points are always calculated anew. This is a great loss of efficiency and could be mended by more careful code.

Parameters

in	*rbin*	Bin prescription. Contains bin edges, e.g. rbin = (0,dr,2∗dr,...,rmax-dr,rmax) for an equidistant bin. Particles further from the focal particle than rmax are not considered.
in	*number_of_points*	Determines how often the function calls an SCF_individual function.
in	*name*	Used to choose a specific correlation function to be calcualted. Options are

Table 6.5 Values of name

name value	Operation
¨g¨	uses calc_SCF_g_individual, calculates g(r)
¨anglediff¨	uses calc_SCF_anglediff_individual, calculates mean angle difference
¨S¨	uses calc_SCF_S_individual, calculates mean spin alignment
¨S_par¨	uses calc_SCF_S_oriented_individual with the orientation along the total magnetization angle
¨S_perp¨	uses calc_SCF_S_oriented_individual with the orientation perpendicular to the total magnetization angle
¨P¨	uses calc_SCF_P_individual, calculates mean momentum alignment
¨W¨	uses calc_SCF_W_individual, calculates mean spin momentum correlation
¨E¨	uses calc_SCF_E_individual, calculates mean energy correlation
¨Ekin¨	uses calc_SCF_Ekin_individual, calculates mean kinetic energy correlation
¨Eint¨	uses calc_SCF_Eint_individual, calculates mean interaction energy correlation
Other	For any other entry, the return value is set to 0. A warning is printed to std::cerr.

6.2.3.43 calc_SCF_E_individual()

```
std::vector< double > group::calc_SCF_E_individual (
            const int index,
```

```
            const std::vector< double > rbin,
            std::vector< int > & counts ) const
```

Calculates the static total energy correlation function for a specific particle at index.

More precisely, obtains $<$e(index)$*$e(j)$>$ for all j in the sample, with e the total energy of a particle, sums the results and sorts them into bins.

Parameters

in	*index*	Particle index.
in	*rbin*	Bin prescription. Contains bin edges, e.g. rbin = (0,dr,2$*$dr,...,rmax-dr,rmax) for an equidistant bin. Particles further from the focal particle than rmax are not considered.
in,out	*counts*	Counts how many particle are found within a bin. As the function is typically repeatedly to sample many particles, this parameter can be updated for each individual call to the function. Must be initialized to zero before the first call.

6.2.3.44 calc_SCF_Eint_individual()

```
std::vector< double > group::calc_SCF_Eint_individual (
            const int index,
            const std::vector< double > rbin,
            std::vector< int > & counts ) const
```

Calculates the static interaction energy correlation function for a specific particle at index.

More precisely, obtains $<$e_int(index)$*$e_int(j)$>$ for all j in the sample, with e_int the interaction energy of a particle, sums the results and sorts them into bins.

Parameters

in	*index*	Particle index.
in	*rbin*	Bin prescription. Contains bin edges, e.g. rbin = (0,dr,2$*$dr,...,rmax-dr,rmax) for an equidistant bin. Particles further from the focal particle than rmax are not considered.
in,out	*counts*	Counts how many particle are found within a bin. As the function is typically repeatedly to sample many particles, this parameter can be updated for each individual call to the function. Must be initialized to zero before the first call.

6.2.3.45 calc_SCF_Ekin_individual()

```
std::vector< double > group::calc_SCF_Ekin_individual (
            const int index,
            const std::vector< double > rbin,
            std::vector< int > & counts ) const
```

Calculates the static kinetic energy correlation function for a specific particle at index.

More precisely, obtains $<$e_kin(index)$*$e_kin(j)$>$ for all j in the sample, with e_kin the kinetic energy of a particle, sums the results and sorts them into bins.

6.2.3.46 calc_SCF_g()

```
std::vector< double > group::calc_SCF_g (
            const std::vector< double > rbin,
            int number_of_points ) const
```

Calculates the overall pair distribution function g(r)

Uses calc_SCF_g_individual for number_of_points many randomly chosen particles and averages over the result

Parameters

in	*rbin*	Bin prescription. Contains bin edges, e.g. rbin = (0,dr,2∗dr,...,rmax-dr,rmax) for an equidistant bin. Particles further from the focal particle than rmax are not considered.
in	*number_of_points*	Determines how often the function calls calc_SCF_g_individual.

6.2.3.47 calc_SCF_g_individual()

```
std::vector< double > group::calc_SCF_g_individual (
            const int index,
            const std::vector< double > rbin ) const
```

Calculates the pair distribution function g(r) for a specific particle at index.

More precisely, counts particles j within the interval (rbin[k],rbin[k+1]]. The result is multiplied by 2 ∗ pi ∗ rbin[k] ∗ dr[k] ∗ rho, with the density rho and the bin width dr[k] = rbin[k+1] - rbin[k].

Parameters

in	*index*	Particle index.
in	*rbin*	Bin prescription. Contains bin edges, e.g. rbin = (0,dr,2∗dr,...,rmax-dr,rmax) for an equidistant bin. Particles further from the focal particle than rmax are not considered.

6.2.3.48 calc_SCF_P_individual()

```
std::vector< double > group::calc_SCF_P_individual (
            const int index,
            const std::vector< double > rbin,
            std::vector< int > & counts ) const
```

Calculates the static momentum correlation function for a specific particle at index.

More precisely, obtains <p_[index]∗p[j]> for all j in the sample (meaning the inner product in this case), sums the results and sorts them into bins.

6.2.3.49 calc_SCF_S_individual()

```
std::vector< double > group::calc_SCF_S_individual (
            const int index,
```

```
            const std::vector< double > rbin,
            std::vector< int > & counts ) const
```

Calculates the static spin correlation function for a specific particle at index.

More precisely, obtains <S_index ∗ S_j>=<cos(theta_[index]-theta_[j])> for all j in the sample, sums the results and sorts them into bins.

Parameters

in	*index*	Particle index.
in	*rbin*	Bin prescription. Contains bin edges, e.g. rbin = (0,dr,2∗dr,...,rmax-dr,rmax) for an equidistant bin. Particles further from the focal particle than rmax are not considered.
in,out	*counts*	Counts how many particle are found within a bin. As the function is typically repeatedly to sample many particles, this parameter can be updated for each individual call to the function. Must be initialized to zero before the first call.

6.2.3.50 calc_SCF_S_oriented_individual()

```
std::vector< double > group::calc_SCF_S_oriented_individual (
            const int index,
            const std::vector< double > rbin,
            const double & orientation_angle,
            std::vector< int > & counts ) const
```

Calculates the static oriented spin correlation function for a specific particle at index.

More precisely, obtains <cos(theta_[index])∗cos(theta_[j])> for all j in the sample, sums the results and sorts them into bins.

Parameters

in	*index*	Particle index.
in	*rbin*	Bin prescription. Contains bin edges, e.g. rbin = (0,dr,2∗dr,...,rmax-dr,rmax) for an equidistant bin. Particles further from the focal particle than rmax are not considered.
in,out	*counts*	Counts how many particle are found within a bin. As the function is typically repeatedly to sample many particles, this parameter can be updated for each individual call to the function. Must be initialized to zero before the first call.

6.2.3.51 calc_SCF_W_individual()

```
std::vector< double > group::calc_SCF_W_individual (
            const int index,
            const std::vector< double > rbin,
            std::vector< int > & counts ) const
```

Calculates the static spin momentum correlation function for a specific particle at index.

More precisely, obtains <w_[index]∗w[j]> for all j in the sample, sums the results and sorts them into bins.

Parameters

in	*index*	Particle index.
in	*rbin*	Bin prescription. Contains bin edges, e.g. rbin = (0,dr,2∗dr,...,rmax-dr,rmax) for an equidistant bin. Particles further from the focal particle than rmax are not considered.
in,out	*counts*	Counts how many particle are found within a bin. As the function is typically repeatedly to sample many particles, this parameter can be updated for each individual call to the function. Must be initialized to zero before the first call.

6.2.3.52 calc_space_angular_mom() [1/2]

```
double group::calc_space_angular_mom ( ) const
```

Returns total spatial angular momentum of particles.

6.2.3.53 calc_space_angular_mom() [2/2]

```
double group::calc_space_angular_mom (
            int i ) const  [inline]
```

Returns spatial angular momentum of the particle with index i.

6.2.3.54 calc_tau()

```
topology::Vector2d group::calc_tau ( ) const
```

Calculates τ, as defined for the xy model. NOT CORRECT FOR THE MOBILE CASE.

6.2.3.55 calc_TCF()

```
group::calc_TCF (
            const group & G_initial,
            std::vector< topology::Vector2d > qvals,
            std::string fluctname_initial,
            std::string fluctname_current ) const
```

Calculates time-correlation function between two different groups. Fluctuation names must be specified.

Calculates $a^*_{\mathbf{q}}b(t)$, or more accurately $(a^{\mathtt{G\_initial}}_{\mathbf{q}})^* b^{\mathtt{G}}$, where $\mathbf{q}$ is a wave vector and $a^{\mathtt{G\_initial}}_{\mathbf{q}}$ and $b^{\mathtt{G}}_{\mathbf{q}}$ are field fluctuations associated with the group `G_initial` and `G` (the current instance of group for which calc_TCF is called), respectively.

Returns a vector whose entries correspond to the wavevectors in qvals.

Parameters

in	*G_initial*	group with which the correlation is computed. In most cases, this is the simulated group at a previous time.
in	*qvals*	vector of $\mathbf{q}$-values for which the product of field fluctuations is calculated
in	*fluctname_initial*	specifies $a^{G_initial}_{\mathbf{q}}$, the field fluctuation of G_initial. For details, see calc_fieldfluct
in	*fluctname_current*	specifies $b^{G}_{\mathbf{q}}$, the field fluctuation in G. For details, see calc_fieldfluct

6.2.3.56 calc_temperature()

```
double group::calc_temperature ( ) const
```

Returns temperature. Careful, this function returns ((<p^2>-.

^2)/m + (<w^2>-<w>^2)/I)/3, not the kinetic energy. Intensive.

6.2.3.57 calc_temperature_p()

```
double group::calc_temperature_p ( ) const  [inline]
```

Returns linear momentum temperature.

6.2.3.58 calc_temperature_w()

```
double group::calc_temperature_w ( ) const  [inline]
```

Returns spin angular momentum temperature.

6.2.3.59 calc_teq()

```
std::complex< double > group::calc_teq (
            const topology::Vector2d q,
            double Te_0 ) const
```

Calculates $theta_{q}$ (see Bissinger PhD thesis) and calc_fieldfluct .

6.2.3.60 calc_vortexdensity_signed()

```
double group::calc_vortexdensity_signed ( ) const
```

Returns the signed vortex density (i.e. number of positive vortices minus number of negative vortices divided by box area).

6.2.3.61 calc_vortexdensity_unsigned()

```
double group::calc_vortexdensity_unsigned ( ) const
```

Returns the unsigned vortex density (i.e. number of vortices divided by box area).

6.2.3.62 calc_vorticity()

```
double group::calc_vorticity (
            int index ) const
```

Returns vorticity along the plaquette at index.

6.2.3.63 calc_wq()

```
std::complex< double > group::calc_wq (
            const topology::Vector2d q,
            double W_0 ) const
```

Calculates w_{q} (see Bissinger PhD thesis) and calc_fieldfluct .

6.2.3.64 clear()

```
void group::clear ( )
```

Clears particles and partition.

6.2.3.65 coord_diff()

```
std::vector< double > group::coord_diff (
            const group & G ) const
```

Returns coordinate difference between this group and another one, with proper care of boundaries. First N_ entries are theta, then r_x, then r_y.

6.2.3.66 fill_partition()

```
void group::fill_partition ( )
```

Fills, i.e. computes the partition.

6.2.3.67 generate_neighbor_list()

```
void group::generate_neighbor_list ( )
```

Fills the variables nb_index_, nb_first_, nb_dist_ according to the current neighborhood situation. Strongly recommended for fmxy model, recommended for xy and fvm model with small system sizes.

6.2.3.68 get_boxsize()

```
double group::get_boxsize ( ) const  [inline]
```

Returns smallest box length.

6.2.3.69 get_coord()

```
std::vector< double > group::get_coord ( ) const
```

Returns vector of all coordinates (angles theta_ and poitions r_, length 3N)

6.2.3.70 get_cutoff()

```
double group::get_cutoff ( ) const  [inline]
```

Returns member variable cutoff_ (interaction cutoff length)

6.2.3.71 get_density()

```
double group::get_density ( ) const  [inline]
```

Returns density.

6.2.3.72 get_group_type()

```
std::string group::get_group_type ( ) const  [inline]
```

Returns member variable group_type_ (type of group)

6.2.3.73 get_I()

```
double group::get_I ( ) const  [inline]
```

Returns member variable I_ (spin inertia)

6.2.3.74 get_J()

```
double group::get_J ( ) const  [inline]
```

Returns member variable J_ (spin coupling strength)

6.2.3.75 get_L()

```
topology::Vector2d group::get_L ( ) const  [inline]
```

Returns simulation box size.

6.2.3.76 get_m()

```
double group::get_m ( ) const  [inline]
```

Returns member variable m_ (particle mass)

6.2.3.77 get_mom()

```
std::vector< double > group::get_mom ( ) const
```

Returns vector of all momenta (spin momenta w_ and linear momenta p_, length 3N)

6.2.3.78 get_N()

```
int group::get_N ( ) const  [inline]
```

Returns number of particles.

6.2.3.79 get_neighbors() [1/2]

```
std::vector< int > group::get_neighbors (
            int i,
            std::string cellselect,
            std::vector< double > & distances ) const
```

Returns indices of neighbors of the particle. Selection of cells possible.

Parameters

in	*i*	Particle index.
in	*cellselect*	Cell selection command. ¨all¨ is for all neighboring cells, ¨ur¨ is for the cell of the particle, the three cells above and the cell to the right ¨single¨ is just for the cell the particle is in.
out	*distances*	Stores the distances to all neighbors. Saves computation time. Only filled in case of mobile particles.

6.2.3.80 get_neighbors() [2/2]

```
std::vector< int > group::get_neighbors (
            int i,
            std::vector< double > & distances ) const  [inline]
```

Returns indices of neighbors of the particle. Uses the member variable nb_rule_ to determine which cells to select.

6.2.3.81 get_p() [1/2]

```
std::vector< topology::Vector2d > group::get_p ( ) const  [inline]
```

Returns member vector p_ (linear momenta). Length N.

6.2.3.82 get_p() [2/2]

```
topology::Vector2d group::get_p (
            int i ) const  [inline]
```

Returns linear momentum p_[i] of particle i.

6.2.3.83 get_r() [1/2]

```
std::vector< topology::Vector2d > group::get_r ( ) const  [inline]
```

Returns member vector r_ (positions). Length N.

6.2.3.84 get_r() [2/2]

```
topology::Vector2d group::get_r (
            int i ) const  [inline]
```

Returns position r_[i] of particle i.

6.2.3.85 get_sqrtN()

```
int group::get_sqrtN ( ) const  [inline]
```

Returns sqrt of number of particles.

6.2.3.86 get_theta() [1/2]

```
std::vector< double > group::get_theta ( ) const  [inline]
```

Returns member vector theta_ (spin angles). Length N.

6.2.3.87 get_theta() [2/2]

```
double group::get_theta (
            int i ) const  [inline]
```

Returns spin angle theta_[i] of particle i.

6.2.3.88 get_vm_eta()

```
double group::get_vm_eta ( ) const  [inline]
```

Returns member variable vm_eta_ (Vicsek model noise strength)

6.2.3.89 get_vm_v()

```
double group::get_vm_v ( ) const  [inline]
```

Returns member variable vm_v_ (Vicsek model velocity)

6.2.3.90 get_volume()

```
double group::get_volume ( ) const  [inline]
```

Returns volume.

6.2.3.91 get_w() [1/2]

```
std::vector< double > group::get_w ( ) const  [inline]
```

Returns member vector w_ (spin momenta). Length N.

6.2.3.92 get_w() [2/2]

```
double group::get_w (
            int i ) const  [inline]
```

Returns spin momentum w_[i] of particle i.

6.2.3.93 initialize()

```
void group::initialize (
            const parameters & par )
```

Initializes particle data for the group based on parameters given.

6.2.3.94 initialize_random()

```
void group::initialize_random (
            double kbT = 0 )
```

Initializes the mobile group with random particle positions and fills the partition.

6.2.3.95 initialize_zero()

```
void group::initialize_zero ( )
```

Sets all particles to zero.

6.2.3.96 J_pot()

```
double group::J_pot (
            double dist ) const  [inline]
```

Returns spin interaction potential (distance-dependence)

6.2.3.97 J_pot_prime()

```
double group::J_pot_prime (
            double dist ) const  [inline]
```

Returns derivative of spin interaction potential (distance-dependence)

6.2.3.98 J_pot_primeprime()

```
double group::J_pot_primeprime (
            double dist ) const  [inline]
```

Returns second derivative of spin interaction potential (distance-dependence)

6.2.3.99 mom_to_zero()

```
void group::mom_to_zero ( )
```

Sets momenta to zero by shifts.

6.2.3.100 operator*=()

```
group & group::operator*= (
            const double a )
```

Multiplies particles by constant (used for adding time derivatives and such).

6.2.3.101 operator+=()

```
group & group::operator+= (
            const group & G )
```

Adds particle entries (used for adding time derivatives and such).

6.2.3.102 periodic_distance()

```
double group::periodic_distance (
            int i,
            int j ) const  [inline]
```

Returns distance between particle i and j considering periodic boundaries (square box). Taking sqrt takes more time than returning the squared quantity by dist_periodic_squared.

6.2.3.103 periodic_distance_squared()

```
double group::periodic_distance_squared (
            int i,
            int j ) const  [inline]
```

Returns squared distance between particle i and j considering periodic boundaries (square box). Squared function faster to calculate.

6.2.3.104 periodic_distance_vector()

```
topology::Vector2d group::periodic_distance_vector (
            int i,
            int j ) const  [inline]
```

Returns distance vector between particle i and j considering periodic boundaries (square box).

6.2.3.105 plaquette()

```
std::vector< int > group::plaquette (
            int i ) const
```

Return the plaquette the particle i belongs to. i is in the lower left corner. Only works for lattice-based models.

6.2.3.106 print_group()

```
void group::print_group (
            std::ofstream & outputfile ) const
```

Prints the entire group to the outputfile.

6.2.3.107 print_r()

```
void group::print_r (
            std::ofstream & outputfile ) const
```

Prints only position coordinates of group to the outputfile.

6.2.3.108 r_to_lattice()

```
void group::r_to_lattice ( )
```

Sets positions to lattice. Decides which lattice depending on lattice_type_ member variable.

6.2.3.109 r_to_squarelattice()

```
void group::r_to_squarelattice ( )
```

Sets positions to square lattice.

6.2.3.110 r_to_trigonallattice()

```
void group::r_to_trigonallattice ( )
```

Sets positions to trigonal lattice. CAREFUL! Trigonal lattice does not fit well into square box.

6.2.3.111 randomize_particles()

```
void group::randomize_particles (
            double kbT = 0 )
```

Sets particles to random values.

6.2.3.112 read_from_snapshot()

```
void group::read_from_snapshot (
            std::string snapshotname )
```

Reads coordinates and momenta from file snapshotname.

6.2.3.113 scale_from_subgroup() [1/2]

```
void group::scale_from_subgroup (
            const group & G )
```

Takes a subgroup (smaller group) and scales it up to the correct size of the group by copying.

Subgroups must be smaller by powers of 4. For proper results, this-object must be initialized.

6.2.3.114 scale_from_subgroup() [2/2]

```
void group::scale_from_subgroup (
            std::string snapshotname )
```

Reads a subgroup (smaller group) from a file and scales it up to the correct size of the group by copying.

Takes a subgroup (smaller group) and scales it up to the correct size of the group by copying. Subgroups must be smaller by powers of 4. For proper results, this-object must be initialized. Unlike the same function that takes a group as input, this function needs only a coordinate file. It is thus simpler to use.

6.2.3.115 scale_mom()

```
void group::scale_mom (
            double a )
```

Scales all momenta (w_, p_) by a factor a.

6.2.3.116 set_all_p()

```
void group::set_all_p (
            topology::Vector2d p )
```

Sets all p to given value (useful for setting T = 0)

6.2.3.117 set_all_theta()

```
void group::set_all_theta (
            double theta )
```

Sets all theta to given value (useful for perfect spin alignment)

6.2.3.118 set_all_w()

```
void group::set_all_w (
            double w )
```

Sets all w to given value (useful for setting T = 0)

6.2.3.119 set_p()

```
void group::set_p (
            topology::Vector2d p,
            int i )
```

Gives p_ of particle i a specified value.

6.2.3.120 set_particle()

```
void group::set_particle (
            double theta,
            double w,
            topology::Vector2d r,
            topology::Vector2d p,
            int i )
```

Sets all values theta_, w_, r_, p_ of particle i to the designated values.

6.2.3.121 set_px()

```
void group::set_px (
            double px,
            int i )
```

Gives x-component of p_ of particle i a specified value.

6.2.3.122 set_py()

```
void group::set_py (
            double py,
            int i )
```

Gives y-component of p_ of particle i a specified value.

6.2.3.123 set_r()

```
void group::set_r (
            topology::Vector2d r,
            int i )
```

Gives r_ of particle i a specified value.

6.2.3.124 set_r_to_pbc()

```
void group::set_r_to_pbc ( )
```

Sets particle positions according to boundary conditions.

6.2.3.125 set_rx()

```
void group::set_rx (
            double x,
            int i )
```

Gives x-component of r_ of particle i a specified value.

6.2.3.126 set_ry()

```
void group::set_ry (
            double y,
            int i )
```

Gives y-component of r_ of particle i a specified value.

6.2.3.127 set_temperature() [1/2]

```
void group::set_temperature (
            double kT )
```

Randomizes momenta to be in agreement with given kT of all particles.

6.2.3.128 set_temperature() [2/2]

```
void group::set_temperature (
            double kT,
            int i )
```

Randomizes momenta to be in agreement with given kT of particle i.

6.2.3.129 set_temperature_p() [1/2]

```
void group::set_temperature_p (
            double kT )
```

Randomizes linear momenta to be in agreement with given kT of all particles.

6.2.3.130 set_temperature_p() [2/2]

```
void group::set_temperature_p (
            double kT,
            int i )
```

Randomizes linear momenta to be in agreement with given kT of particle i.

6.2.3.131 set_temperature_w() [1/2]

```
void group::set_temperature_w (
            double kT )
```

Randomizes spin momenta to be in agreement with given kT of all particles.

6.2.3.132 set_temperature_w() [2/2]

```
void group::set_temperature_w (
            double kT,
            int i )
```

Randomizes spin momenta to be in agreement with given kT of particle i.

6.2.3.133 set_theta()

```
void group::set_theta (
            double theta,
            int i )
```

Gives theta_ of particle i a specified value.

6.2.3.134 set_theta_to_interval()

```
void group::set_theta_to_interval ( )
```

Sets theta_ values to interval (-pi, pi)

6.2.3.135 set_w()

```
void group::set_w (
            double w,
            int i )
```

Gives w_ of particle i a specified value.

6.2.3.136 size()

```
int group::size ( ) const  [inline]
```

Same as get_N()

6.2.3.137 stream_along_spin()

```
void group::stream_along_spin (
            double v )
```

Streams along spin, r_new = r + v ∗ spin(theta)

6.2.3.138 sum_e_squared()

```
double group::sum_e_squared ( ) const
```

Total energy squared, basically N_<e_i^2>. Extensive.

6.2.3.139 sum_eint_squared()

```
double group::sum_eint_squared ( ) const
```

Interaction energy squared, basically N_<e_{i,int}^2>. Extensive.

6.2.3.140 sum_ekin_squared()

```
double group::sum_ekin_squared ( ) const
```

Kinetic energy squared, basically N_<e_{i,kin}^2>. Extensive.

6.2.3.141 sum_p()

```
topology::Vector2d group::sum_p ( ) const
```

Total momentum. Basically N_<p>. Extensive.

6.2.3.142 sum_p_4()

```
double group::sum_p_4 ( ) const
```

Sum over momentum to the fourth power. Basically N_<p^4>. Extensive.

6.2.3.143 sum_p_squared()

```
double group::sum_p_squared ( ) const
```

Sum over momentum squared. Basically N_<p^2>. Extensive.

6.2.3.144 sum_s()

```
topology::Vector2d group::sum_s ( ) const
```

Magnetization. Basically N_<s>. Extensive.

6.2.3.145 sum_s_4()

```
double group::sum_s_4 ( ) const  [inline]
```

Magnetization to the fourth power. Basically N_<s>^4. Extensive.

6.2.3.146 sum_s_squared()

```
double group::sum_s_squared ( ) const  [inline]
```

Magnetization squared. Basically N_<s>^2. Extensive.

6.2.3.147 sum_theta()

```
double group::sum_theta ( ) const
```

Returns sum over all theta, basically N_<theta>. Extensive. Probably pointless.

6.2.3.148 sum_w()

```
double group::sum_w ( ) const
```

Returns sum over omega, basically N_<w>. Extensive.

6.2.3.149 sum_w_4()

```
double group::sum_w_4 ( ) const
```

Omega to the fourth power, basically N_<w^4>. Proportional to kinetic energy. Extensive.

6.2.3.150 sum_w_squared()

```
double group::sum_w_squared ( ) const
```

Omega squared, basically N_<w^2>. Proportional to kinetic energy. Extensive.

6.2.3.151 theta_diff()

```
double group::theta_diff (
            int i,
            int j ) const  [inline]
```

Difference in angles of two different particles. θ_{ij} in Bore paper.

6.2.3.152 time_derivative()

```
group group::time_derivative ( ) const
```

Returns time derivative of the entire group.

6.2.3.153 time_derivative_coord()

```
std::vector< double > group::time_derivative_coord ( ) const
```

Returns coordinate time derivative (first N_ entries are theta, then r_x, then r_y).

6.2.3.154 time_derivative_mom()

```
std::vector< double > group::time_derivative_mom ( ) const
```

Returns momenta time derivative (first N_ entries are omega, then p_x, then p_y).

6.2.3.155 time_derivative_p()

```
std::vector< double > group::time_derivative_p ( ) const
```

Returns p (linear momentum) time derivative. First N_ entries are x direction, N_+1 to 2N_ is y direction (splitting useful for leapfrog)

6.2.3.156 time_derivative_r()

```
std::vector< double > group::time_derivative_r ( ) const
```

Returns r (particle position) time derivative. First N_ entries are x direction, N_+1 to 2N_ is y direction (splitting useful for leapfrog)

6.2.3.157 time_derivative_theta()

```
std::vector< double > group::time_derivative_theta ( ) const
```

Returns theta (spin angle) time derivative (splitting useful for leapfrog)

6.2.3.158 time_derivative_w()

```
std::vector< double > group::time_derivative_w ( ) const
```

Returns omega (spin momentum) time derivative (splitting useful for leapfrog)

6.2.3.159 U_pot()

```
double group::U_pot (
            double dist ) const  [inline]
```

Returns spatial interaction potential.

6.2.3.160 U_pot_prime()

```
double group::U_pot_prime (
            double dist ) const  [inline]
```

Returns derivative of spatial interaction potential.

6.2.3.161 U_pot_primeprime()

```
double group::U_pot_primeprime (
            double dist ) const  [inline]
```

Returns second derivative of spatial interaction potential.

6.2.4 Member Data Documentation

6.2.4.1 cutoff_

```
double group::cutoff_ = 1  [protected]
```

Interaction cutoff radius.

6.2.4.2 group_type_

```
std::string group::group_type_  [protected]
```

Type of the group. Can be ¨xy¨ for the XY model, ¨mxy¨ for the mobile XY model, ¨fmxy¨ for a mobile XY model frozen in place, ¨vm¨ for the Vicsek model and ¨fvm¨ for the frozen (static) Vicsek model.

6.2.4.3 I_

```
double group::I_ = 1  [protected]
```

Spin inertia.

6.2.4.4 J_

```
double group::J_ = 1  [protected]
```

Nearest neighbor interaction strength.

6.2.4.5 L_

```
topology::Vector2d group::L_  [protected]
```

Size of the box (some functions only defined for square boxes yet).

6.2.4.6 lattice_type_

```
char group::lattice_type_  [protected]
```

lattice type. (type 's': square, type 't': trigonal, type 'n': none (mxy model etc))

6.2.4.7 m_

```
double group::m_ = 1  [protected]
```

Mass.

6.2.4.8 N_

```
int group::N_  [protected]
```

Size of the group.

6.2.4.9 nb_list_

```
neighbor_list* group::nb_list_  [protected]
```

neighbor-list

6.2.4.10 nb_mult_factor_

```
double group::nb_mult_factor_  [protected]
```

If neighbor rule leads to double counting, this factor has to be .5, otherwise 1.

6.2.4.11 nb_rule_

```
std::string group::nb_rule_  [protected]
```

Neighbor calculation rule. Possible values: ¨bruteforce¨, ¨all¨, ¨ur¨ for full, (partition with) all and (partition with) upper right neighbors.

6.2.4.12 p_

```
std::vector<topology::Vector2d> group::p_  [protected]
```

Particle momenta/velocities in the group.

6.2.4.13 partition_

```
partition group::partition_  [protected]
```

Partition (cell list) for neighborhood interaction.

6.2.4.14 r_

```
std::vector<topology::Vector2d> group::r_  [protected]
```

Particle positions in the group.

6.2.4.15 sqrtN_

```
int group::sqrtN_  [protected]
```

Square root of the group size (often useful).

6.2.4.16 theta_

```
std::vector<double> group::theta_  [protected]
```

Particle spin angles in the group.

6.2.4.17 U_

```
double group::U_ = 1  [protected]
```

Spatial repulsion interaction strength.

6.2.4.18 vm_eta_

```
double group::vm_eta_ [protected]
```

Vicsek model parameter eta: Angle for random noise.

6.2.4.19 vm_v_

```
double group::vm_v_ [protected]
```

Vicsek model parameter v: Streaming velocity.

6.2.4.20 w_

```
std::vector<double> group::w_ [protected]
```

Particle spin momenta in the group.

The documentation for this class was generated from the following files:

- group.h
- group.cpp

6.3 integrator Class Reference

Defines various integration methods for groups. Also includes thermostats. Integrators include: fourth order Runge-Kutta (rk4) and Leapfrog (lf)

```
#include <integrator.h>
```

Public Member Functions

- integrator ()
  Empty constructor.
- integrator (double dtin, std::string type)
  Constructor with given time steps.
- void integrator_eq (const parameters &par)
  Sets integrator for equilibration run.
- void integrator_sample (const parameters &par)
  Sets integrator for sampling run.
- void initialize (const parameters &par, const group &G)
  Initializes integrator from given parameters and a given group.
- void initialize_parameters (const parameters &par)
  Initializes integrator from given parameters.
- double get_dt () const
  Returns dt.
- double get_H_0 () const
  Returns H_0_.

- double get_pi () const

  Returns pi_.
- double get_eta () const

  Returns eta_.
- double get_s () const

  Returns s_.
- double berendsen_thermostat (group &G, double T_desired, double berendsen_tau)

  Implements the Berendsen thermostat.
- group integrate (const group &G, std::vector< double > &momdot)

  Performs an integration time step.
- group vm_rule (const group &G)

  Implements Vicsek model time-evolution rules.
- group rk4 (const group &G)

  Implements the fourth-order Runge-Kutta method.
- group leapfrog (const group &G, std::vector< double > &momdot)

  Implements leapfrog solver. Can get initial momdot for reduced computation time.
- group leapfrog_active (const group &G, std::vector< double > &momdot, double activity)

  Implements leapfrog solver with added activity. Can get initial momdot for reduced computation time.
- group np (const group &G, double kT, double &s, double &pi, double Q, double H_0)

  Nosé-Poincare integration solved via velocity Verlet.
- group nh (const group &G)

  Nosé-Hoover integration solved via velocity Verlet.
- group np (const group &G)

  Nosé-Poincare integration solved via velocity Verlet.
- group langevin (const group &G, std::vector< double > &momdot)

  Langevin dynamics solver.
- group langevin_active (const group &G, std::vector< double > &momdot, double activityc)

  Langevin dynamics solver including active motion of particles along spin.
- double NoseHamiltonian (group &G, double kT, double s, double pi, double Q) const

  The Hamiltonian for Nosé-Hoover and Nosé-Poincaré schemes.
- void add_activity (group &G, double activity) const

  Adds activity by moving the particles along spin direction times activity times dt.

6.3.1 Detailed Description

Defines various integration methods for groups. Also includes thermostats. Integrators include: fourth order Runge-Kutta (rk4) and Leapfrog (lf)

Author

Thomas Bissinger, RK4 implementation by Mathias Hoefler

Date

Created: 2019-04-12

Last Updated: 2023-08-06

6.3.2 Constructor & Destructor Documentation

6.3.2.1 integrator() [1/2]

```
integrator::integrator ( )  [inline]
```

Empty constructor.

6.3.2.2 integrator() [2/2]

```
integrator::integrator (
            double dtin,
            std::string type )
```

Constructor with given time steps.

6.3.3 Member Function Documentation

6.3.3.1 add_activity()

```
void integrator::add_activity (
            group & G,
            double activity ) const
```

Adds activity by moving the particles along spin direction times activity times dt.

6.3.3.2 berendsen_thermostat()

```
double integrator::berendsen_thermostat (
            group & G,
            double T_desired,
            double berendsen_tau )
```

Implements the Berendsen thermostat.

6.3.3.3 get_dt()

```
double integrator::get_dt ( ) const  [inline]
```

Returns dt.

6.3.3.4 get_eta()

```
double integrator::get_eta ( ) const  [inline]
```

Returns eta_.

6.3.3.5 get_H_0()

```
double integrator::get_H_0 ( ) const  [inline]
```

Returns H_0_.

6.3.3.6 get_pi()

```
double integrator::get_pi ( ) const  [inline]
```

Returns pi_.

6.3.3.7 get_s()

```
double integrator::get_s ( ) const  [inline]
```

Returns s_.

6.3.3.8 initialize()

```
void integrator::initialize (
            const parameters & par,
            const group & G )
```

Initializes integrator from given parameters and a given group.

The group is needed to initialize some further values like the reference energy H_0 in a Nosé-Poincaré scheme. Makes a call to initialize_parameters for the parameter readout.

6.3.3.9 initialize_parameters()

```
void integrator::initialize_parameters (
            const parameters & par )
```

Initializes integrator from given parameters.

6.3.3.10 integrate()

```
group integrator::integrate (
            const group & G,
            std::vector< double > & momdot )
```

Performs an integration time step.

Depending on integrator_type_, different solvers are called applied. For details, see also definition of integrator_↩ type_.

Not every method requires a variable momdot, yet some make use of a staggered time grid or the velocity at a previous time step, these need momdot.

6.3.3.11 integrator_eq()

```
void integrator::integrator_eq (
            const parameters & par )
```

Sets integrator for equilibration run.

6.3.3.12 integrator_sample()

```
void integrator::integrator_sample (
            const parameters & par )
```

Sets integrator for sampling run.

6.3.3.13 langevin()

```
group integrator::langevin (
            const group & G,
            std::vector< double > & momdot )
```

Langevin dynamics solver.

Can get initial momdot for reduced computation time. Details in Allen and Tildesley, ¨Computer Simulation of Liquids¨, Chapter 12

6.3.3.14 langevin_active()

```
group integrator::langevin_active (
            const group & G,
            std::vector< double > & momdot,
            double activityc )
```

Langevin dynamics solver including active motion of particles along spin.

Can get initial momdot for reduced computation time. Details in Allen and Tildesley, ¨Computer Simulation of Liquids¨, Chapter 12

6.3.3.15 leapfrog()

```
group integrator::leapfrog (
            const group & G,
            std::vector< double > & momdot )
```

Implements leapfrog solver. Can get initial momdot for reduced computation time.

6.3.3.16 leapfrog_active()

```
group integrator::leapfrog_active (
            const group & G,
            std::vector< double > & momdot,
            double activity )
```

Implements leapfrog solver with added activity. Can get initial momdot for reduced computation time.

6.3.3.17 nh()

```
group integrator::nh (
            const group & G )
```

Nosé-Hoover integration solved via velocity Verlet.

For some details, see https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2577381/, jchemphys 2008 128(24), Kleinerman et al., eq (6-17)

6.3.3.18 NoseHamiltonian()

```
double integrator::NoseHamiltonian (
            group & G,
            double kT,
            double s,
            double pi,
            double Q ) const  [inline]
```

The Hamiltonian for Nosé-Hoover and Nosé-Poincaré schemes.

6.3.3.19 np() [1/2]

```
group integrator::np (
            const group & G )  [inline]
```

Nosé-Poincare integration solved via velocity Verlet.

For some details, see https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2577381/, jchemphys 2008 128(24), Kleinerman et al., eq (21-28)

6.3.3.20 np() [2/2]

```
group integrator::np (
            const group & G,
            double kT,
            double & s,
            double & pi,
            double Q,
            double H_0 )
```

Nosé-Poincare integration solved via velocity Verlet.

For some details, see https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2577381/, jchemphys 2008 128(24), Kleinerman et al., eq (21-28)

6.3.3.21 rk4()

```
group integrator::rk4 (
            const group & G )
```

Implements the fourth-order Runge-Kutta method.

6.3.3.22 vm_rule()

```
group integrator::vm_rule (
            const group & G )
```

Implements Vicsek model time-evolution rules.

The documentation for this class was generated from the following files:

- integrator.h
- integrator.cpp

6.4 neighbor_list Class Reference

Defines the neighbor_list class. Can be used to extract all information about neighborhood in a group, that is the neighbor pairs and all the distances. Neighbors are those other particles within the cutoff radius or the nearest neighbors in case of a lattice system.

```
#include <neighbor_list.h>
```

Public Member Functions

- neighbor_list ()
  Empty constructor.
- neighbor_list (const group &G)
  Standard constructor.
- void clear ()
  Clears the neighbor_list.
- bool is_empty () const
  Returns whether or not the neighbor_list is empty.
- std::vector< int > get_neighbors (int i) const
  Returns all neighbors of particle i (elements of nb_indices_)
- std::vector< double > get_dist (int i) const
  Returns all distances between neighbors and particle i (elements of nb_dist_)

Protected Attributes

- int N_ = 0
  Number of particles.
- std::vector< int > nb_indices_
  Indices of neighbors. Only efficient for systems with fixed neighbors.
- std::vector< int > nb_first_
  Index of first neighbor of each particle. Helpful when accessing nb_indices.
- std::vector< double > nb_dist_
  Distances to neighbors. Only efficient for systems with fixed neighbors.

6.4.1 Detailed Description

Defines the neighbor_list class. Can be used to extract all information about neighborhood in a group, that is the neighbor pairs and all the distances. Neighbors are those other particles within the cutoff radius or the nearest neighbors in case of a lattice system.

Author

Thomas Bissinger

6.4.2 Constructor & Destructor Documentation

6.4.2.1 neighbor_list() [1/2]

```
neighbor_list::neighbor_list ( )  [inline]
```

Empty constructor.

6.4.2.2 neighbor_list() [2/2]

```
neighbor_list::neighbor_list (
            const group & G )
```

Standard constructor.

Parameters

in	*group*	the group based on which the neighbor_list has to be intialized. WARNING: Off-lattice systems need a full partition.

6.4.3 Member Function Documentation

6.4.3.1 clear()

```
void neighbor_list::clear ( )
```

Clears the neighbor_list.

6.4.3.2 get_dist()

```
std::vector< double > neighbor_list::get_dist (
            int i ) const
```

Returns all distances between neighbors and particle i (elements of nb_dist_)

6.4.3.3 get_neighbors()

```
std::vector< int > neighbor_list::get_neighbors (
            int i ) const
```

Returns all neighbors of particle i (elements of nb_indices_)

6.4.3.4 is_empty()

```
bool neighbor_list::is_empty ( ) const  [inline]
```

Returns whether or not the neighbor_list is empty.

6.4.4 Member Data Documentation

6.4.4.1 N_

```
int neighbor_list::N_ = 0  [protected]
```

Number of particles.

6.4.4.2 nb_dist_

```
std::vector<double> neighbor_list::nb_dist_  [protected]
```

Distances to neighbors. Only efficient for systems with fixed neighbors.

6.4.4.3 nb_first_

```
std::vector<int> neighbor_list::nb_first_  [protected]
```

Index of first neighbor of each particle. Helpful when accessing nb_indices.

6.4.4.4 nb_indices_

```
std::vector<int> neighbor_list::nb_indices_  [protected]
```

Indices of neighbors. Only efficient for systems with fixed neighbors.

The documentation for this class was generated from the following files:

- neighbor_list.h
- neighbor_list.cpp

6.5 parameters Class Reference

Contains the run parameters of a simulation.

`#include <parameters.h>`

Public Member Functions

- parameters ()
  Empty constructor.
- int read_from_file (std::ifstream &infile)
  Reads parameters from an infile.
- int correct_values (std::ofstream &stdoutfile)
  Corrects potentially wrong input.
- void initialize_bins ()
  Initializes both bins.
- void initialize_qbin ()
  Initializes the qbin.
- void initialize_rbin (int N_rbin)
  Initializes the rbin.
- void scale_tau (double scale_factor)
  tau_berendsen can be scaled with this.
- std::string system () const
- std::string mode () const
  Returns system_.
- std::string job_id () const
  Returns mode_.
- std::string outfilename () const
  Returns job_id_.
- int N () const
  Returns outfilename_.
- int sqrtN () const
  Returns N_.
- double dof () const
  Returns sqrtN_.
- double L () const
  Returns dof_.
- double dt () const
  Returns L_.
- double Tmax () const
  Returns dt_.
- double samplestart () const
  Returns Tmax_.
- double samplestep () const
  Returns samplestart_.
- double av_time_spacing () const
  Returns samplestep_.
- int Nsamp () const
  Returns av_time_spacing_.
- int randomseed () const

Returns Nsamp_.

- double kT () const

  Returns randomseed_.
- double l () const

  Returns kT_.
- double m () const

  Returns l_.
- double J () const

  Returns m_.
- double U () const

  Returns J_.
- double cutoff () const

  Returns U_.
- char lattice_type () const

  Returns cutoff_.
- double activity () const

  Returns lattice_type_.
- double vm_v () const

  Returns activity_.
- double vm_eta () const

  Returns vm_v_.
- std::string init_mode () const

  Returns vm_eta_.
- std::string init_file () const

  Returns init_mode_.
- double init_kT () const

  Returns init_file_.
- double init_random_displacement () const

  Returns init_kT_.
- double init_random_angle () const

  Returns init_random_displacement_.
- std::string eq_mode () const

  Returns init_random_angle_.
- std::string eq_integrator_type () const

  Returns eq_mode_.
- double eq_Tmax () const

  Returns eq_integrator_type_.
- std::string eq_breakcond () const

  Returns eq_Tmax_.
- double eq_agreement_threshold () const

  Returns eq_breakcond_.
- double eq_av_time () const

  Returns eq_agreement_threshold_.
- double tau_berendsen () const

  Returns eq_av_time_.
- double eq_anneal_rate () const

  Returns tau_berendsen_.
- double eq_anneal_step () const

  Returns eq_anneal_rate_.
- double eq_Tprintstep () const

  Returns eq_anneal_step_.

- double eq_brownian_kT_p () const

  Returns eq_Tprintstep_.
- double eq_brownian_timestep () const

  Returns eq_brownian_kT_p_.
- double eq_brownian_kT_omega () const

  Returns eq_brownian_timestep_.
- bool eq_sampswitch () const

  Returns eq_brownian_kT_omega_.
- int eq_Nsamp () const

  Returns eq_sampswitch_.
- std::string eq_samp_time_sequence () const

  Returns eq_Nsamp_.
- std::string sample_integrator_type () const

  Returns eq_samp_time_sequence_.
- std::string ensemble () const

  Returns sample_integrator_type_.
- double nhnp_pi () const

  Returns ensemble_.
- double nhnp_Q () const

  Returns nhnp_pi_.
- double nhnp_tau () const

  Returns nhnp_Q_.
- double nh_eta () const

  Returns nhnp_tau_.
- double np_s () const

  Returns nh_eta_.
- double mc_steplength_theta () const

  Returns np_s_.
- double brownian_kT_omega () const

  Returns mc_steplength_theta_.
- double brownian_kT_p () const

  Returns brownian_kT_omega_.
- double brownian_timestep () const

  Returns brownian_kT_p_.
- double gamma_ld_om () const

  Returns brownian_timestep_.
- double gamma_ld_p () const

  Returns gamma_ld_om_.
- double mc_steplength_r () const

  Returns gamma_ld_p_.
- std::string sampling_time_sequence () const

  Returns mc_steplength_r_.
- int N_rbin () const

  Returns sampling_time_sequence_.
- int N_qbin () const

  Returns N_rbin_.
- double min_binwidth_r () const

  Returns N_qbin_.
- std::string qbin_type () const

  Returns min_binwidth_r_.
- double qmax () const

Returns qbin_type_.

- double min_binwidth_q () const

  Returns qmax_.
- std::vector< double > rbin () const

  Returns min_binwidth_q_.
- std::vector< double > qbin () const

  Returns rbin_.
- int qsamps_per_bin () const

  Returns qbin_.
- int n_rsamps () const

  Returns qsamps_per_bin_.
- double qfullmax () const

  Returns n_rsamps_.
- bool print_snapshots () const

  Returns qfullmax_.
- bool on_fly_sampling () const

  Returns print_snapshots_.
- std::string snap_overview_file () const

  Returns on_fly_sampling_.
- std::string output_folder () const

  Returns snap_overview_file_.

Protected Attributes

- std::string system_ = ¨xy¨

  Type of system.
- std::string mode_ = ¨none¨

  Run mode for the simulation.
- std::string init_mode_ = ¨random¨

  Initialization mode.
- std::string init_file_ = ¨input/init_snap.in¨

  Path to initialization file. Only relevant if init_mode is set to ¨file¨.
- double init_kT_ = -1

  Initialization temperature, temperature at which simulation should start. Has no effect when chosen negative.
- double init_random_displacement_ = 0

  Random particle displacement at initialization (for scaled initialization only)
- double init_random_angle_ = 0

  Random angle displacement at initialization (for scaled initialization only)
- std::string job_id_ = ¨¨

  Job identifier. If specified, data will be stored in data_'job_id'.out etc. Otherwise, it is just the default data.out.
- std::string outfilename_ = ¨output/data.out¨

  Name of outfile.
- int N_ = 256

  Number of particles.
- int sqrtN_ = 16

  Square root of the number of particles.
- double dof_

  Degrees of freedom. Calculated automatically from N_ and the system_.
- double L_ = 16

  Length of simulation box.

- double dt_ = 0.01

  Integration time step.
- double Tmax_ = 100

  Runtime for sampling.
- double samplestart_ = 0

  Time at which sampling starts in the integration run.
- double samplestep_ = 1

  Time separation of two samples.
- double av_time_spacing_ = 1e2

  Minimum time spacing for average calculations in sampling.
- int Nsamp_ = 30

  Number of samples computed in sampling. Overwrites samplestep.
- int randomseed_ = 1

  Random seed.
- double kT_ = .89

  Temperature (units of energy)
- double I_ = 1

  Spin inertia.
- double m_ = 1

  Mass.
- double J_ = 1

  Spin interaction strength.
- double U_ = 1

  Spatial interaction strength.
- double cutoff_ = 1

  Cutoff length for interaction potential (default value 1)
- char lattice_type_ = 's'

  In static system: Type of lattice.
- double activity_ = 0

  Activity in the mobile XY model (stream velocity)
- double vm_v_ = 0

  Vicsek model streaming velocity.
- double vm_eta_ = 0

  Vicsek model noise strength (between 0 and 1, actual noise interval will be [-eta/2∗Pi,eta/2 ∗ Pi]).
- std::string eq_integrator_type_

  Integrator used for equilibration.
- std::string eq_mode_ = ¨anneal¨

  Equilibration mode.
- std::string eq_breakcond_ = ¨temperature¨

  Break condition for equilibration.
- double eq_Tmax_ = 100

  Maximum equilibration time.
- double eq_agreement_threshold_ = 1e-2

  Temperature agreement threshold to check successful equilibration.
- double eq_av_time_ = 0

  Averaging time in equilibration (set to 0 for no averaging)
- double tau_berendsen_

  tau coefficient of the Berendsen thermostat
- double eq_anneal_rate_ = .999

  Annealing rate.
- double eq_anneal_step_ = 1e1

Annealing time step.

- double eq_Tprintstep_ = std::numeric_limits<double>::quiet_NaN()

  Temperature print interval. Prints intermediate data when equilibration reaches T.

- double eq_brownian_kT_omega_ = -1

  Temperature for spin momentum thermostat during equilibration in a Brownian dynamics simulation (applied when non-negative)

- double eq_brownian_kT_p_ = -1

  Temperature for momentum thermostat during equilibration in a Brownian dynamics simulation (applied when non-negative)

- double eq_brownian_timestep_ = 1.00

  Step length for Brownian timestep during equilibration (in units of system time)

- bool eq_sampswitch_ = 0

  switch controlling whether sampling data is stored in equilibration

- int eq_Nsamp_ = 100

  Number of time samplings performed during equilibration. Only has an effect if eq_sampswitch = 1.

- std::string eq_samp_time_sequence_ = ¨lin¨

  Sampling time sequence. 'lin' or 'log'.

- std::string sample_integrator_type_

  Integrator used for sampling.

- std::string ensemble_

  Desired ensemble. ¨nvt¨ only works with certain integrators. ¨nve¨ is good standard.

- double nhnp_pi_ = 0

  Nose-Hoover or Nose-Poincare pi value.

- double nhnp_Q_ = -1

  Nose-Hoover or Nose-Poincare Q value. Default value negative, only used if positive. Method using nhnp_tau is more recommended.

- double nhnp_tau_ = 0.01

  Nose-Hoover or Nose-Poincare tau value (time scale that determines Q by Q = g ∗ kT ∗ tau^ 2). Order of a microscopic collision time.

- double nh_eta_ = 0

  Nose-Hoover eta value.

- double np_s_ = 1

  Nose-Poincare s value (initial time scaling factor)

- double mc_steplength_theta_ = 0.1

  Maximal trial step length of MC algorithm for angle theta.

- double brownian_kT_omega_ = -1

  Temperature for spin momentum thermostat in a Brownian dynamics simulation (applied when non-negative)

- double brownian_kT_p_ = -1

  Temperature for momentum thermostat in a Brownian dynamics simulation (applied when non-negative)

- double brownian_timestep_ = 1.00

  Step length for Brownian timestep (in units of system time)

- double gamma_ld_p_ = 0

  Damping rate gamma of the Langevin dynamics. Associated to the linear angular momentum p.

- double gamma_ld_om_ = 0

  Damping rate gamma of the Langevin dynamics. Associated to the spin angular velocity omega.

- double mc_steplength_r_ = 0.1

  Maximal trial step length of MC algorithm for distance r.

- std::string sampling_time_sequence_ = ¨lin¨

  Specifies how sampling should be performed. Values are lin (linear sequence of sampling times) and log (logarithmic sequence of sampling times).

- int N_rbin_

  Number of bins for r values.

- double min_binwidth_r_

  Minimal width of r bins (should be around 1)
- std::string qbin_type_ = ¨mult¨

  Type of qbin. ¨all¨ uses all possible q values on the grid, ¨mult¨ uses only integer multiples of $2\pi/L$ (recommended)
- double qmax_ = 2 * M_PI

  Maximum value for q in bin. Default is 2π (used when qbin_type = 'all')
- double min_binwidth_q_ = .015

  Minimal width of q bins, in fractions of 2pi / L.
- int N_qbin_

  Number of bins for q values (used when qbin_type = 'mult')
- std::vector< double > rbin_

  Binning values for the vector r. Set internally.
- std::vector< double > qbin_

  Binning values for the vector q. Set internally.
- double qfullmax_

  q value for full coverage (everything beyond is randomly sampled)
- int qsamps_per_bin_

  Number of q samples on a circle.
- int n_rsamps_ = 30

  Number of (randomly chosen) r sampling points for the calculation of correlation functions in position space (like spin correlation functions or mean squared displacements etc.)
- bool print_snapshots_ = false

  Specifies if snapshots should be printed.
- bool on_fly_sampling_ = true

  Specifies if sampling should be performed on-the-fly.
- std::string output_folder_ = ¨output¨

  Path to output folder. Added by David Stadler.
- std::string snap_overview_file_ = ¨output/snapshot_overview.out¨

  File where the names of all snapshots are stored. Only relevant when run in sampling mode.

6.5.1 Detailed Description

Contains the run parameters of a simulation.

Author

Thomas Bissinger

Date

Created: 2019-04-12

Last Updated: 2023-08-02

6.5.2 Constructor & Destructor Documentation

6.5.2.1 parameters()

```
parameters::parameters ( )  [inline]
```

Empty constructor.

6.5.3 Member Function Documentation

6.5.3.1 activity()

```
double parameters::activity ( ) const  [inline]
```

Returns lattice_type_.

6.5.3.2 av_time_spacing()

```
double parameters::av_time_spacing ( ) const  [inline]
```

Returns samplestep_.

6.5.3.3 brownian_kT_omega()

```
double parameters::brownian_kT_omega ( ) const  [inline]
```

Returns mc_steplength_theta_.

6.5.3.4 brownian_kT_p()

```
double parameters::brownian_kT_p ( ) const  [inline]
```

Returns brownian_kT_omega_.

6.5.3.5 brownian_timestep()

```
double parameters::brownian_timestep ( ) const  [inline]
```

Returns brownian_kT_p_.

6.5.3.6 correct_values()

```
int parameters::correct_values (
            std::ofstream & stdoutfile )
```

Corrects potentially wrong input.

6.5.3.7 cutoff()

```
double parameters::cutoff ( ) const  [inline]
```

Returns U_.

6.5.3.8 dof()

```
double parameters::dof ( ) const  [inline]
```

Returns sqrtN_.

6.5.3.9 dt()

```
double parameters::dt ( ) const  [inline]
```

Returns L_.

6.5.3.10 ensemble()

```
std::string parameters::ensemble ( ) const  [inline]
```

Returns sample_integrator_type_.

6.5.3.11 eq_agreement_threshold()

```
double parameters::eq_agreement_threshold ( ) const  [inline]
```

Returns eq_breakcond_.

6.5.3.12 eq_anneal_rate()

```
double parameters::eq_anneal_rate ( ) const  [inline]
```

Returns tau_berendsen_.

6.5.3.13 eq_anneal_step()

```
double parameters::eq_anneal_step ( ) const  [inline]
```

Returns eq_anneal_rate_.

6.5.3.14 eq_av_time()

```
double parameters::eq_av_time ( ) const  [inline]
```

Returns eq_agreement_threshold_.

6.5.3.15 eq_breakcond()

```
std::string parameters::eq_breakcond ( ) const  [inline]
```

Returns eq_Tmax_.

6.5.3.16 eq_brownian_kT_omega()

```
double parameters::eq_brownian_kT_omega ( ) const  [inline]
```

Returns eq_brownian_timestep_.

6.5.3.17 eq_brownian_kT_p()

```
double parameters::eq_brownian_kT_p ( ) const  [inline]
```

Returns eq_Tprintstep_.

6.5.3.18 eq_brownian_timestep()

```
double parameters::eq_brownian_timestep ( ) const  [inline]
```

Returns eq_brownian_kT_p_.

6.5.3.19 eq_integrator_type()

```
std::string parameters::eq_integrator_type ( ) const  [inline]
```

Returns eq_mode_.

6.5.3.20 eq_mode()

```
std::string parameters::eq_mode ( ) const  [inline]
```

Returns init_random_angle_.

6.5.3.21 eq_Nsamp()

```
int parameters::eq_Nsamp ( ) const  [inline]
```

Returns eq_sampswitch_.

6.5.3.22 eq_samp_time_sequence()

```
std::string parameters::eq_samp_time_sequence ( ) const  [inline]
```

Returns eq_Nsamp_.

6.5.3.23 eq_sampswitch()

```
bool parameters::eq_sampswitch ( ) const  [inline]
```

Returns eq_brownian_kT_omega_.

6.5.3.24 eq_Tmax()

```
double parameters::eq_Tmax ( ) const  [inline]
```

Returns eq_integrator_type_.

6.5.3.25 eq_Tprintstep()

```
double parameters::eq_Tprintstep ( ) const  [inline]
```

Returns eq_anneal_step_.

6.5.3.26 gamma_ld_om()

```
double parameters::gamma_ld_om ( ) const  [inline]
```

Returns brownian_timestep_.

6.5.3.27 gamma_ld_p()

```
double parameters::gamma_ld_p ( ) const  [inline]
```

Returns gamma_ld_om_.

6.5.3.28 I()

```
double parameters::I ( ) const  [inline]
```

Returns kT_.

6.5.3.29 init_file()

```
std::string parameters::init_file ( ) const  [inline]
```

Returns init_mode_.

6.5.3.30 init_kT()

```
double parameters::init_kT ( ) const  [inline]
```

Returns init_file_.

6.5.3.31 init_mode()

```
std::string parameters::init_mode ( ) const  [inline]
```

Returns vm_eta_.

6.5.3.32 init_random_angle()

```
double parameters::init_random_angle ( ) const  [inline]
```

Returns init_random_displacement_.

6.5.3.33 init_random_displacement()

```
double parameters::init_random_displacement ( ) const  [inline]
```

Returns init_kT_.

6.5.3.34 initialize_bins()

```
void parameters::initialize_bins ( )
```

Initializes both bins.

6.5.3.35 initialize_qbin()

```
void parameters::initialize_qbin ( )
```

Initializes the qbin.

Also initializes qfullmax_ so that the first bin that is randomly selected contains more than 1.5 $*$ qsamps_per_bin_ q-values.

6.5.3.36 initialize_rbin()

```
void parameters::initialize_rbin (
            int N_rbin )
```

Initializes the rbin.

6.5.3.37 J()

```
double parameters::J ( ) const  [inline]
```

Returns m_.

6.5.3.38 job_id()

```
std::string parameters::job_id ( ) const  [inline]
```

Returns mode_.

6.5.3.39 kT()

```
double parameters::kT ( ) const  [inline]
```

Returns randomseed_.

6.5.3.40 L()

```
double parameters::L ( ) const  [inline]
```

Returns dof_.

6.5.3.41 lattice_type()

```
char parameters::lattice_type ( ) const  [inline]
```

Returns cutoff_.

6.5.3.42 m()

```
double parameters::m ( ) const  [inline]
```

Returns l_.

6.5.3.43 mc_steplength_r()

```
double parameters::mc_steplength_r ( ) const  [inline]
```

Returns gamma_ld_p_.

6.5.3.44 mc_steplength_theta()

```
double parameters::mc_steplength_theta ( ) const  [inline]
```

Returns np_s_.

6.5.3.45 min_binwidth_q()

```
double parameters::min_binwidth_q ( ) const  [inline]
```

Returns qmax_.

6.5.3.46 min_binwidth_r()

```
double parameters::min_binwidth_r ( ) const  [inline]
```

Returns N_qbin_.

6.5.3.47 mode()

```
std::string parameters::mode ( ) const  [inline]
```

Returns system_.

6.5.3.48 N()

```
int parameters::N ( ) const  [inline]
```

Returns outfilename_.

6.5.3.49 N_qbin()

```
int parameters::N_qbin ( ) const  [inline]
```

Returns N_rbin_.

6.5.3.50 N_rbin()

```
int parameters::N_rbin ( ) const  [inline]
```

Returns sampling_time_sequence_.

6.5.3.51 n_rsamps()

```
int parameters::n_rsamps ( ) const  [inline]
```

Returns qsamps_per_bin_.

6.5.3.52 nh_eta()

```
double parameters::nh_eta ( ) const  [inline]
```

Returns nhnp_tau_.

6.5.3.53 nhnp_pi()

```
double parameters::nhnp_pi ( ) const  [inline]
```

Returns ensemble_.

6.5.3.54 nhnp_Q()

```
double parameters::nhnp_Q ( ) const  [inline]
```

Returns nhnp_pi_.

6.5.3.55 nhnp_tau()

```
double parameters::nhnp_tau ( ) const  [inline]
```

Returns nhnp_Q_.

6.5.3.56 np_s()

```
double parameters::np_s ( ) const  [inline]
```

Returns nh_eta_.

6.5.3.57 Nsamp()

```
int parameters::Nsamp ( ) const  [inline]
```

Returns av_time_spacing_.

6.5.3.58 on_fly_sampling()

```
bool parameters::on_fly_sampling ( ) const  [inline]
```

Returns print_snapshots_.

6.5.3.59 outfilename()

```
std::string parameters::outfilename ( ) const  [inline]
```

Returns job_id_.

6.5.3.60 output_folder()

```
std::string parameters::output_folder ( ) const  [inline]
```

Returns snap_overview_file_.

6.5.3.61 print_snapshots()

```
bool parameters::print_snapshots ( ) const  [inline]
```

Returns qfullmax_.

6.5.3.62 qbin()

```
std::vector< double > parameters::qbin ( ) const  [inline]
```

Returns rbin_.

6.5.3.63 qbin_type()

```
std::string parameters::qbin_type ( ) const  [inline]
```

Returns min_binwidth_r_.

6.5.3.64 qfullmax()

```
double parameters::qfullmax ( ) const  [inline]
```

Returns n_rsamps_.

6.5.3.65 qmax()

```
double parameters::qmax ( ) const  [inline]
```

Returns qbin_type_.

6.5.3.66 qsamps_per_bin()

```
int parameters::qsamps_per_bin ( ) const  [inline]
```

Returns qbin_.

6.5.3.67 randomseed()

```
int parameters::randomseed ( ) const  [inline]
```

Returns Nsamp_.

6.5.3.68 rbin()

```
std::vector< double > parameters::rbin ( ) const  [inline]
```

Returns min_binwidth_q_.

6.5.3.69 read_from_file()

```
int parameters::read_from_file (
            std::ifstream & infile )
```

Reads parameters from an infile.

6.5.3.70 sample_integrator_type()

```
std::string parameters::sample_integrator_type ( ) const  [inline]
```

Returns eq_samp_time_sequence_.

6.5.3.71 samplestart()

```
double parameters::samplestart ( ) const  [inline]
```

Returns Tmax_.

6.5.3.72 samplestep()

```
double parameters::samplestep ( ) const  [inline]
```

Returns samplestart_.

6.5.3.73 sampling_time_sequence()

```
std::string parameters::sampling_time_sequence ( ) const  [inline]
```

Returns mc_steplength_r_.

6.5.3.74 scale_tau()

```
void parameters::scale_tau (
            double scale_factor )
```

tau_berendsen can be scaled with this.

6.5.3.75 snap_overview_file()

```
std::string parameters::snap_overview_file ( ) const  [inline]
```

Returns on_fly_sampling_.

6.5.3.76 sqrtN()

```
int parameters::sqrtN ( ) const  [inline]
```

Returns N_.

6.5.3.77 system()

```
std::string parameters::system ( ) const  [inline]
```

6.5.3.78 tau_berendsen()

```
double parameters::tau_berendsen ( ) const  [inline]
```

Returns eq_av_time_.

6.5.3.79 Tmax()

```
double parameters::Tmax ( ) const  [inline]
```

Returns dt_.

6.5.3.80 U()

```
double parameters::U ( ) const  [inline]
```

Returns J_.

6.5.3.81 vm_eta()

```
double parameters::vm_eta ( ) const  [inline]
```

Returns vm_v_.

6.5.3.82 vm_v()

```
double parameters::vm_v ( ) const  [inline]
```

Returns activity_.

6.5.4 Member Data Documentation

6.5.4.1 activity_

```
double parameters::activity_ = 0  [protected]
```

Activity in the mobile XY model (stream velocity)

6.5.4.2 av_time_spacing_

```
double parameters::av_time_spacing_ = 1e2  [protected]
```

Minimum time spacing for average calculations in sampling.

6.5.4.3 brownian_kT_omega_

```
double parameters::brownian_kT_omega_ = -1  [protected]
```

Temperature for spin momentum thermostat in a Brownian dynamics simulation (applied when non-negative)

6.5.4.4 brownian_kT_p_

```
double parameters::brownian_kT_p_ = -1  [protected]
```

Temperature for momentum thermostat in a Brownian dynamics simulation (applied when non-negative)

6.5.4.5 brownian_timestep_

```
double parameters::brownian_timestep_ = 1.00  [protected]
```

Step length for Brownian timestep (in units of system time)

6.5.4.6 cutoff_

```
double parameters::cutoff_ = 1  [protected]
```

Cutoff length for interaction potential (default value 1)

6.5.4.7 dof_

```
double parameters::dof_  [protected]
```

Degrees of freedom. Calculated automatically from N_ and the system_.

6.5.4.8 dt_

```
double parameters::dt_ = 0.01  [protected]
```

Integration time step.

6.5.4.9 ensemble_

```
std::string parameters::ensemble_  [protected]
```

Desired ensemble. ¨nvt¨ only works with certain integrators. ¨nve¨ is good standard.

6.5.4.10 eq_agreement_threshold_

```
double parameters::eq_agreement_threshold_ = 1e-2  [protected]
```

Temperature agreement threshold to check successful equilibration.

6.5.4.11 eq_anneal_rate_

```
double parameters::eq_anneal_rate_ = .999  [protected]
```

Annealing rate.

6.5.4.12 eq_anneal_step_

```
double parameters::eq_anneal_step_ = 1e1  [protected]
```

Annealing time step.

6.5.4.13 eq_av_time_

```
double parameters::eq_av_time_ = 0  [protected]
```

Averaging time in equilibration (set to 0 for no averaging)

6.5.4.14 eq_breakcond_

```
std::string parameters::eq_breakcond_ = ¨temperature¨  [protected]
```

Break condition for equilibration.

Possible values:

Table 6.16 Values of eq_breakcond_

eq_breakcond_ value	break condition
¨temperature¨	breaks when the desired temperature has been maintained for a certain time (default)
¨time¨	breaks when an equilibration time has passed and checks temperature, continues equilibration if temperature is not reached
¨time_hard¨	breaks when an equilibration time has passed, regardless of temperature
¨any¨	breaks if the time or the temperature condition is met

6.5.4.15 eq_brownian_kT_omega_

```
double parameters::eq_brownian_kT_omega_ = -1  [protected]
```

Temperature for spin momentum thermostat during equilibration in a Brownian dynamics simulation (applied when non-negative)

6.5.4.16 eq_brownian_kT_p_

```
double parameters::eq_brownian_kT_p_ = -1  [protected]
```

Temperature for momentum thermostat during equilibration in a Brownian dynamics simulation (applied when non-negative)

6.5.4.17 eq_brownian_timestep_

```
double parameters::eq_brownian_timestep_ = 1.00  [protected]
```

Step length for Brownian timestep during equilibration (in units of system time)

6.5.4.18 eq_integrator_type_

```
std::string parameters::eq_integrator_type_  [protected]
```

Integrator used for equilibration.

Possible values:

Table 6.17 Values of eq_integrator_type_

eq_integrator_type_ value	integrator used
¨lf¨	leap-frog scheme
¨rk4¨	4th order Runge-Kutta scheme
¨langevin¨	Langevin-thermostatted time-evolution (stochastic, not tested)
¨nh¨	Nosé-Hoover thermostatted integrator (not extensively tested)
¨np¨	Nosé-Poincaré thermostatted integrator (possibly bugged, not extensively tested)
¨mc¨	Monte-Carlo integration: Has not been implemented yet

TODO: Langevin dynamics uses same parameters as sampling integrator. Potentially different parameters for equilibration may be implemented

TODO: Monte Carlo does not work yet.

6.5.4.19 eq_mode_

```
std::string parameters::eq_mode_ = ¨anneal¨  [protected]
```

Equilibration mode.

Possible values:

Table 6.18 Values of eq_mode_

eq_mode_ value	mode of equilibration
¨anneal¨	reaches the desired temperature with simulated annealing (default)
¨berendsen¨	reaches the desired temperature with Berendsen thermostat
¨brownian¨	reaches the desired temperature by repeatedly setting it (hard Brownian thermostat)

TODO: Add options for Langevin thermostat.

TODO: Equilibrate without temperature, keep energy fixed.

6.5.4.20 eq_Nsamp_

```
int parameters::eq_Nsamp_ = 100  [protected]
```

Number of time samplings performed during equilibration. Only has an effect if eq_sampswitch = 1.

6.5.4.21 eq_samp_time_sequence_

```
std::string parameters::eq_samp_time_sequence_ = ¨lin¨  [protected]
```

Sampling time sequence. 'lin' or 'log'.

6.5.4.22 eq_sampswitch_

```
bool parameters::eq_sampswitch_ = 0  [protected]
```

switch controlling whether sampling data is stored in equilibration

6.5.4.23 eq_Tmax_

```
double parameters::eq_Tmax_ = 100  [protected]
```

Maximum equilibration time.

6.5.4.24 eq_Tprintstep_

```
double parameters::eq_Tprintstep_ = std::numeric_limits<double>::quiet_NaN()  [protected]
```

Temperature print interval. Prints intermediate data when equilibration reaches T.

6.5.4.25 gamma_ld_om_

```
double parameters::gamma_ld_om_ = 0  [protected]
```

Damping rate gamma of the Langevin dynamics. Associated to the spin angular velocity omega.

6.5.4.26 gamma_ld_p_

```
double parameters::gamma_ld_p_ = 0  [protected]
```

Damping rate gamma of the Langevin dynamics. Associated to the linear angular momentum p.

6.5.4.27 I_

```
double parameters::I_ = 1  [protected]
```

Spin inertia.

6.5.4.28 init_file_

```
std::string parameters::init_file_ = ¨input/init_snap.in¨  [protected]
```

Path to initialization file. Only relevant if init_mode is set to ¨file¨.

6.5.4.29 init_kT_

```
double parameters::init_kT_ = -1  [protected]
```

Initialization temperature, temperature at which simulation should start. Has no effect when chosen negative.

6.5.4.30 init_mode_

```
std::string parameters::init_mode_ = ¨random¨  [protected]
```

Initialization mode.

Possible values:

Table 6.19 Values of init_mode

init_mode value	initialization
¨random¨	all particles placed and oriented randomly
¨aligned¨	particles placed randomly, but oriented along some random direction
¨file¨	reads initial configuration from file

For initialization from file, the path has to be specified in the parameter init_file_

6.5.4.31 init_random_angle_

```
double parameters::init_random_angle_ = 0  [protected]
```

Random angle displacement at initialization (for scaled initialization only)

6.5.4.32 init_random_displacement_

```
double parameters::init_random_displacement_ = 0  [protected]
```

Random particle displacement at initialization (for scaled initialization only)

6.5.4.33 J_

```
double parameters::J_ = 1  [protected]
```

Spin interaction strength.

6.5.4.34 job_id_

```
std::string parameters::job_id_ = ""  [protected]
```

Job identifier. If specified, data will be stored in data_'job_id'.out etc. Otherwise, it is just the default data.out.

6.5.4.35 kT_

```
double parameters::kT_ = .89  [protected]
```

Temperature (units of energy)

6.5.4.36 L_

```
double parameters::L_ = 16  [protected]
```

Length of simulation box.

6.5.4.37 lattice_type_

```
char parameters::lattice_type_ = 's'  [protected]
```

In static system: Type of lattice.

Possible values:

Table 6.20 Values of lattice_type

lattice_typ value	lattice
's'	square lattice (default)
't'	trigonal lattice

6.5.4.38 m_

```
double parameters::m_ = 1  [protected]
```

Mass.

6.5.4.39 mc_steplength_r_

```
double parameters::mc_steplength_r_ = 0.1  [protected]
```

Maximal trial step length of MC algorithm for distance r.

6.5.4.40 mc_steplength_theta_

```
double parameters::mc_steplength_theta_ = 0.1  [protected]
```

Maximal trial step length of MC algorithm for angle theta.

6.5.4.41 min_binwidth_q_

```
double parameters::min_binwidth_q_ = .015  [protected]
```

Minimal width of q bins, in fractions of 2pi / L.

6.5.4.42 min_binwidth_r_

```
double parameters::min_binwidth_r_  [protected]
```

Minimal width of r bins (should be around 1)

6.5.4.43 mode_

```
std::string parameters::mode_ = ¨none¨  [protected]
```

Run mode for the simulation.

Possible values:

Table 6.21 Values of mode

mode value	What happens during run?
¨none¨	does nothing
¨test¨	like non, unless user specified. See main.cpp
¨integrate¨	performs time integration, possibly with storing
¨integrate_cont¨	picks up an aborted integration and continues (relevant in older version, not recommended)
¨samp¨	samples a list of stored snapshots (has to be provided)
¨equilibrate¨	similar to integrate, but performs equilibration run only

6.5.4.44 N_

```
int parameters::N_ = 256  [protected]
```

Number of particles.

6.5.4.45 N_qbin_

```
int parameters::N_qbin_  [protected]
```

Number of bins for q values (used when qbin_type = 'mult')

6.5.4.46 N_rbin_

```
int parameters::N_rbin_  [protected]
```

Number of bins for r values.

6.5.4.47 n_rsamps_

```
int parameters::n_rsamps_ = 30  [protected]
```

Number of (randomly chosen) r sampling points for the calculation of correlation functions in position space (like spin correlation functions or mean squared displacements etc.)

6.5.4.48 nh_eta_

```
double parameters::nh_eta_ = 0  [protected]
```

Nose-Hoover eta value.

6.5.4.49 nhnp_pi_

```
double parameters::nhnp_pi_ = 0  [protected]
```

Nose-Hoover or Nose-Poincare pi value.

6.5.4.50 nhnp_Q_

```
double parameters::nhnp_Q_ = -1  [protected]
```

Nose-Hoover or Nose-Poincare Q value. Default value negative, only used if positive. Method using nhnp_tau is more recommended.

6.5.4.51 nhnp_tau_

```
double parameters::nhnp_tau_ = 0.01  [protected]
```

Nose-Hoover or Nose-Poincare tau value (time scale that determines Q by Q = g $*$ kT $*$ tau^2). Order of a microscopic collision time.

6.5.4.52 np_s_

```
double parameters::np_s_ = 1  [protected]
```

Nose-Poincare s value (initial time scaling factor)

6.5.4.53 Nsamp_

```
int parameters::Nsamp_ = 30  [protected]
```

Number of samples computed in sampling. Overwrites samplestep.

6.5.4.54 on_fly_sampling_

```
bool parameters::on_fly_sampling_ = true  [protected]
```

Specifies if sampling should be performed on-the-fly.

6.5.4.55 outfilename_

```
std::string parameters::outfilename_ = ¨output/data.out¨  [protected]
```

Name of outfile.

6.5.4.56 output_folder_

```
std::string parameters::output_folder_ = ¨output¨  [protected]
```

Path to output folder. Added by David Stadler.

6.5.4.57 print_snapshots_

```
bool parameters::print_snapshots_ = false  [protected]
```

Specifies if snapshots should be printed.

6.5.4.58 qbin_

```
std::vector<double> parameters::qbin_  [protected]
```

Binning values for the vector q. Set internally.

6.5.4.59 qbin_type_

```
std::string parameters::qbin_type_ = ¨mult¨  [protected]
```

Type of qbin. ¨all¨ uses all possible q values on the grid, ¨mult¨ uses only integer multiples of $2\pi/L$ (recommended)

6.5.4.60 qfullmax_

```
double parameters::qfullmax_  [protected]
```

q value for full coverage (everything beyond is randomly sampled)

6.5.4.61 qmax_

```
double parameters::qmax_ = 2 * M_PI  [protected]
```

Maximum value for q in bin. Default is 2π (used when qbin_type = 'all')

6.5.4.62 qsamps_per_bin_

```
int parameters::qsamps_per_bin_  [protected]
```

Number of q samples on a circle.

6.5.4.63 randomseed_

```
int parameters::randomseed_ = 1  [protected]
```

Random seed.

6.5.4.64 rbin_

```
std::vector<double> parameters::rbin_  [protected]
```

Binning values for the vector r. Set internally.

6.5.4.65 sample_integrator_type_

```
std::string parameters::sample_integrator_type_  [protected]
```

Integrator used for sampling.

Possible values:

Table 6.22 Values of eq_integrator_type_

eq_integrator_type_ value	integrator used
¨lf¨	leap-frog scheme
¨rk4¨	4th order Runge-Kutta scheme
¨langevin¨	Langevin-thermostatted time-evolution (stochastic, not tested)
¨nh¨	Nosé-Hoover thermostatted integrator (not extensively tested)
¨np¨	Nosé-Poincaré thermostatted integrator (possibly bugged, not extensively tested)
¨mc¨	Monte-Carlo integration: Has not been implemented yet

TODO: Monte Carlo does not work yet.

6.5.4.66 samplestart_

```
double parameters::samplestart_ = 0  [protected]
```

Time at which sampling starts in the integration run.

6.5.4.67 samplestep_

```
double parameters::samplestep_ = 1  [protected]
```

Time separation of two samples.

6.5.4.68 sampling_time_sequence_

```
std::string parameters::sampling_time_sequence_ = ¨lin¨  [protected]
```

Specifies how sampling should be performed. Values are lin (linear sequence of sampling times) and log (logarithmic sequence of sampling times).

6.5.4.69 snap_overview_file_

```
std::string parameters::snap_overview_file_ = ¨output/snapshot_overview.out¨  [protected]
```

File where the names of all snapshots are stored. Only relevant when run in sampling mode.

6.5.4.70 sqrtN_

```
int parameters::sqrtN_ = 16  [protected]
```

Square root of the number of particles.

6.5.4.71 system_

```
std::string parameters::system_ = ¨xy¨  [protected]
```

Type of system.

Possible values:

Table 6.23 Values of system

system value	System chosen
¨xy¨	XY model
¨mxy¨	mobile XY (MXY) model
¨fmxy¨	disordered XY (DXY) model / frozen mobile XY model
¨vm¨	Vicsek model (not fully tested)
¨fvm¨	frozen Vicsek model (not fully tested)

6.5.4.72 tau_berendsen_

```
double parameters::tau_berendsen_  [protected]
```

tau coefficient of the Berendsen thermostat

6.5.4.73 Tmax_

```
double parameters::Tmax_ = 100  [protected]
```

Runtime for sampling.

6.5.4.74 U_

```
double parameters::U_ = 1  [protected]
```

Spatial interaction strength.

6.5.4.75 vm_eta_

```
double parameters::vm_eta_ = 0  [protected]
```

Vicsek model noise strength (between 0 and 1, actual noise interval will be [-eta/2∗Pi,eta/2 ∗ Pi]).

6.5.4.76 vm_v_

```
double parameters::vm_v_ = 0  [protected]
```

Vicsek model streaming velocity.

The documentation for this class was generated from the following files:

- parameters.h
- parameters.cpp

6.6 partition Class Reference

Defines the partition class. The simulation box is partitioned into cells. The indices of a vector of particles (used for initialization) are sorted according to the cell they belong to. Has functions for printing and computing average velocities in a neighborhood.

```
#include <partition.h>
```

Public Member Functions

- partition ()

  Empty constructor.
- partition (const int &N, const double &cutoff, const topology::Vector2d &L)

  Standard constructor.
- partition (const int &N, const double &cutoff, const topology::Vector2d &L, const std::vector< topology::Vector2d > &positions)

  Standard constructor.
- ~partition ()

  Destructor.
- void clear ()

  Clears all entries.
- void fill (const std::vector< topology::Vector2d > &positions)

  Fills the partition.
- int get_cellelem (int m) const

  Returns int cellelem_[m].
- std::vector< int > get_cellelem () const

  Returns vector cellelem_.
- int get_cellnum () const

  Returns int cellnum_.
- std::vector< int > get_cellvec () const

  Returns vector cellvec.
- void print () const

  Print partition (only for troubleshooting).
- int find_cell (const topology::Vector2d &r) const

  Returns the index of the cell containing r.
- int find_first (int index) const

  Returns index of the first element in cellvec_ pertaining to the cell with index index.
- int neighbor_cell (int index, int lr, int du) const

  Returns index of neighboring cell to cell at index. Uses periodic boundary conditions.
- int neighbor_cell (int index, int lr, int du, topology::Vector2d &shift) const

  Returns index of neighboring cell to cell at index. Uses periodic boundary conditions. Contains a shift.
- std::vector< int > nb_cells_all (int index, std::vector< topology::Vector2d > &shifts) const

  Returns indices of all neighboring cells (including shifts)
- std::vector< int > nb_cells_all (const topology::Vector2d &r, double cutoffsquared, std::vector< topology::Vector2d > &shifts) const

  Returns indices of all neighboring cells (including shifts). Empty cells and cells out of reach of r are not included.
- std::vector< int > nb_cells_ur (int index, std::vector< topology::Vector2d > &shifts) const

  Returns indices the neighboring cells above and right of the cell and the cell itself (including shifts). Can be used to avoid double counting.
- std::vector< int > nb_cells_ur (const topology::Vector2d &r, double cutoffsquared, std::vector< topology::Vector2d > &shifts) const

  Returns indices the neighboring cells above and right of the cell and the cell itself (including shifts). Empty cells and cells out of reach of r are not included. Can be used to avoid double counting.
- topology::Vector2d corner (int index, int lr, int ud) const

  Get position of a corner of a cell. Which corner is determined by lr, ud.
- std::vector< int > part_in_cell (int index) const

  Returns a std::vector containing the indices of the particles in the box with index 'index'.
- std::vector< int > nb_in_cell_index (int index, const topology::Vector2d &r, double cutoffsquared, const std::vector< topology::Vector2d > &positions, std::vector< double > &distances, const topology::Vector2d shift=0) const

Returns the indices of the particles in a cell for all particles within a cutoff radius of r.

- std::vector< int > nb_in_cell_index_above (int index, const topology::Vector2d &r, double cutoff-squared, const std::vector< topology::Vector2d > &positions, std::vector< double > &distances, const topology::Vector2d shift=0) const

  Calculates only neighbors above particle (reduces computation time).
- void cluster_analysis (std::vector< int > &cluster_sizes, std::vector< int > &cluster_NoP, const int &rho_min) const

  Returns a vector of cluster sizes and the corresponding vector of the number of particles (NoP) per cluster. The variable rho_min determines the minimum number a cell has to contain to be considered part of a cluster.
- std::vector< int > cluster_recursion (int current_index, std::vector< int > indices, const int &rho_min) const

  Recursive function, returns vector containing all the indices being part of a cluster. In principle, current_index is not necessary, one could also use the last entry of indices. But it doesn't disturb performance and is a little more readable.

Protected Attributes

- int N_

  Size of the group.
- topology::Vector2d L_

  Size of simulation box.
- topology::Vector2d cellwidth_

  Width of a cell in the cell list. Typically the cutoff radius of the interaction.
- std::vector< int > M_ = std::vector<int>(2)

  Number of cells per dimension. Dim = d = 2.
- int cellnum_

  Total number of cells.
- std::vector< int > firsts_

  Indices of the first element pertaining to a cell. Dim = N_;.
- std::vector< int > cellelem_

  *Integers indicating the number of elements per cell. Dim = M_[0] * ... * M_[d];.*
- std::vector< int > cellvec_

  Vector of particle indices. Dim = N_. Easier to access than the list, and since it has to be accessed a lot, we save it here.

6.6.1 Detailed Description

Defines the partition class. The simulation box is partitioned into cells. The indices of a vector of particles (used for initialization) are sorted according to the cell they belong to. Has functions for printing and computing average velocities in a neighborhood.

Author

Thomas Bissinger

Note

A previous version of this class used pointers to the actual particles instead of their indices. This was changed due to memory access problems. The code is also simpler. On the other hand, one has to store the particle indices and combine the stored indices with the actual vector to access elements. Therefore, some functions need the original particle vector as a call-by-reference input.

Date

Created: late 2017, presumably

Last Updated: 2023-08-06

6.6.2 Constructor & Destructor Documentation

6.6.2.1 partition() [1/3]

```
partition::partition ( )  [inline]
```

Empty constructor.

6.6.2.2 partition() [2/3]

```
partition::partition (
            const int & N,
            const double & cutoff,
            const topology::Vector2d & L )
```

Standard constructor.

Parameters

in	*N*	Number of particles.
in	*cutoff*	Cutoff radius and width of cell.
in	*L*	Size of simulation box, [0, L].

6.6.2.3 partition() [3/3]

```
partition::partition (
            const int & N,
            const double & cutoff,
            const topology::Vector2d & L,
            const std::vector< topology::Vector2d > & positions )
```

Standard constructor.

Parameters

in	*N*	Number of particles.
in	*cutoff*	Cutoff radius and width of cell.
in	*L*	Size of simulation box, [0, L].
in	*positions*	Vector of positions to be sorted.

6.6.2.4 ~partition()

```
partition::~partition ( )
```

Destructor.

6.6.3 Member Function Documentation

6.6.3.1 clear()

```
void partition::clear ( )
```

Clears all entries.

6.6.3.2 cluster_analysis()

```
void partition::cluster_analysis (
            std::vector< int > & cluster_sizes,
            std::vector< int > & cluster_NoP,
            const int & rho_min ) const
```

Returns a vector of cluster sizes and the corresponding vector of the number of particles (NoP) per cluster. The variable rho_min determines the minimum number a cell has to contain to be considered part of a cluster.

Parameters

in	*cluster_sizes*	Stores size (number of pertaining cells) of each cluster. std::vector<int>
in	*cluster_NoP*	Stores the number of particles of each cluster. std::vector<int>
in	*rho_min*	Minimal density of cluster, i.e. minimal number of particles in a cell to be considered part of the cluster. int

Note

The function was originally implemented to check for large number fluctuations in Visek swarms. Hasn't been used since preliminary tests in the early 2018s. Might be useful to someone, but not recommended for use by the author. Analyze particle clusters contained in the partition.

6.6.3.3 cluster_recursion()

```
std::vector< int > partition::cluster_recursion (
            int current_index,
            std::vector< int > indices,
            const int & rho_min ) const
```

Recursive function, returns vector containing all the indices being part of a cluster. In principle, current_index is not necessary, one could also use the last entry of indices. But it doesn't disturb performance and is a little more readable.

Parameters

in	*current_index*	Index that is currently being viewed. int
in	*indices*	Indices that are already part of the cluster. std::vector<int>
in	*rho_min*	Minimal density of cluster, i.e. minimal number of particles in a cell to be considered part of the cluster. int Recursive function, finds all cells being part of a cluster.

6.6.3.4 corner()

```
topology::Vector2d partition::corner (
            int index,
            int lr,
            int ud ) const  [inline]
```

Get position of a corner of a cell. Which corner is determined by lr, ud.

6.6.3.5 fill()

```
void partition::fill (
            const std::vector< topology::Vector2d > & positions )
```

Fills the partition.

Assigns each particle to a cell in the partition.

Parameters

in	*positions*	Vector of positions to be sorted.

6.6.3.6 find_cell()

```
int partition::find_cell (
            const topology::Vector2d & r ) const
```

Returns the index of the cell containing r.

Warning

Has no error output if r is not in the simulation box.

6.6.3.7 find_first()

```
int partition::find_first (
            int index ) const  [inline]
```

Returns index of the first element in cellvec_ pertaining to the cell with index index.

6.6.3.8 get_cellelem() [1/2]

```
std::vector< int > partition::get_cellelem ( ) const  [inline]
```

Returns vector cellelem_.

6.6.3.9 get_cellelem() [2/2]

```
int partition::get_cellelem (
            int m ) const  [inline]
```

Returns int cellelem_[m].

6.6.3.10 get_cellnum()

```
int partition::get_cellnum ( ) const  [inline]
```

Returns int cellnum_.

6.6.3.11 get_cellvec()

```
std::vector< int > partition::get_cellvec ( ) const  [inline]
```

Returns vector cellvec.

6.6.3.12 nb_cells_all() [1/2]

```
std::vector< int > partition::nb_cells_all (
            const topology::Vector2d & r,
            double cutoffsquared,
            std::vector< topology::Vector2d > & shifts ) const
```

Returns indices of all neighboring cells (including shifts). Empty cells and cells out of reach of r are not included.

6.6.3.13 nb_cells_all() [2/2]

```
std::vector< int > partition::nb_cells_all (
            int index,
            std::vector< topology::Vector2d > & shifts ) const
```

Returns indices of all neighboring cells (including shifts)

These functions could be used if the old index is known. But the comparisons it saves are probably similarly expensive as the modulus calculations necessary to return to the original index and check whether the index is still accurate.

6.6.3.14 nb_cells_ur() [1/2]

```
std::vector< int > partition::nb_cells_ur (
            const topology::Vector2d & r,
            double cutoffsquared,
            std::vector< topology::Vector2d > & shifts ) const
```

Returns indices the neighboring cells above and right of the cell and the cell itself (including shifts). Empty cells and cells out of reach of r are not included. Can be used to avoid double counting.

6.6.3.15 nb_cells_ur() [2/2]

```
std::vector< int > partition::nb_cells_ur (
            int index,
            std::vector< topology::Vector2d > & shifts ) const
```

Returns indices the neighboring cells above and right of the cell and the cell itself (including shifts). Can be used to avoid double counting.

6.6.3.16 nb_in_cell_index()

```
std::vector< int > partition::nb_in_cell_index (
            int index,
            const topology::Vector2d & r,
            double cutoffsquared,
            const std::vector< topology::Vector2d > & positions,
            std::vector< double > & distances,
            const topology::Vector2d shift = 0 ) const
```

Returns the indices of the particles in a cell for all particles within a cutoff radius of r.

Parameters

in	*index*	Cell index
in	*r*	Position of particle.
in	*cutoffsquared*	Square of the cutoff radius (square because no roots have to be taken).
in	*positions*	Vector of positions to be accessed.
out	*distances*	Stores the distances to all neighbors. Saves computation time.
in	*shift*	Shifts position of vector r. Optional, default is 0.

ATTENTION: The integer neighbors is not re-initialized, so the number of neighbors is added to the value that is already stored in neighbors. This simplifies calculations with more than one group, but one has to bear that in mind.

6.6.3.17 nb_in_cell_index_above()

```
std::vector< int > partition::nb_in_cell_index_above (
            int index,
            const topology::Vector2d & r,
            double cutoffsquared,
            const std::vector< topology::Vector2d > & positions,
            std::vector< double > & distances,
            const topology::Vector2d shift = 0 ) const
```

Calculates only neighbors above particle (reduces computation time).

6.6.3.18 neighbor_cell() [1/2]

```
int partition::neighbor_cell (
            int index,
```

```
            int lr,
            int du ) const
```

Returns index of neighboring cell to cell at index. Uses periodic boundary conditions.

lr and ur are used to describe where the neighboring cell is. E.g. a cell to the top left would correspond to lr = -1 (left) and ud = 1 (up).

Parameters

in	*index*	index of cell.
in	*lr*	neighboring cell position in left-right (x) direction. In {-1,0,1}. -1 is left.
in	*du*	neighboring cell position in down-up (y) direction. In {-1,0,1}. -1 is down.

6.6.3.19 neighbor_cell() [2/2]

```
int partition::neighbor_cell (
            int index,
            int lr,
            int du,
            topology::Vector2d & shift ) const
```

Returns index of neighboring cell to cell at index. Uses periodic boundary conditions. Contains a shift.

lr and ur are used to describe where the neighboring cell is. E.g. a cell to the top left would correspond to lr = -1 (left) and ud = 1 (up).

Parameters

in	*index*	index of cell.
in	*lr*	neighboring cell position in left-right (x) direction. In {-1,0,1} = (left, center, right).
in	*du*	neighboring cell position in down-up (y) direction. In {-1,0,1} = (below, center, above).
in,out	*shift*	if the neighboring cell is on the other side of the periodic box, a nonzero shift is stored here. The shift is designed such that the cell at index is next to its neighboring cell if shift is subtracted from the positions of all particles in the cell at index. (Later function calls with r-shift)

6.6.3.20 part_in_cell()

```
std::vector< int > partition::part_in_cell (
            int index ) const
```

Returns a std::vector containing the indices of the particles in the box with index 'index'.

6.6.3.21 print()

```
void partition::print ( ) const
```

Print partition (only for troubleshooting).

6.6.4 Member Data Documentation

6.6.4.1 cellelem_

```
std::vector<int> partition::cellelem_  [protected]
```

Integers indicating the number of elements per cell. Dim = M_[0] ∗ ... ∗ M_[d];.

6.6.4.2 cellnum_

```
int partition::cellnum_  [protected]
```

Total number of cells.

6.6.4.3 cellvec_

```
std::vector<int> partition::cellvec_  [protected]
```

Vector of particle indices. Dim = N_. Easier to access than the list, and since it has to be accessed a lot, we save it here.

6.6.4.4 cellwidth_

```
topology::Vector2d partition::cellwidth_  [protected]
```

Width of a cell in the cell list. Typically the cutoff radius of the interaction.

6.6.4.5 firsts_

```
std::vector<int> partition::firsts_  [protected]
```

Indices of the first element pertaining to a cell. Dim = N_;.

Set to the first element of the next cell if the cell is empty.

6.6.4.6 L_

```
topology::Vector2d partition::L_  [protected]
```

Size of simulation box.

Box has to be at least twice the size of the cutoff radius, else particles can have double-interaction. This case is not an error caught by the simulation so far. Note that the simulation box is from 0 to L_.

6.6.4.7 M_

```
std::vector<int> partition::M_ = std::vector<int>(2)  [protected]
```

Number of cells per dimension. Dim = d = 2.

6.6.4.8 N_

```
int partition::N_  [protected]
```

Size of the group.

The documentation for this class was generated from the following files:

- partition.h
- partition.cpp

6.7 sampler Class Reference

Stores and handles all data sampling performed during a run or in a later diagnostic.

```
#include <sampler.h>
```

Public Member Functions

- sampler ()
  Empty constructor.
- sampler (parameters par)
  Constructor. Assigns par_ to par.
- sampler (parameters par, std::vector< topology::Vector2d > qvals)
  Constructor. Assigns par_ to par and qvals_ to qvals.
- void set_parameters (parameters par)
  Sets parameter member variable.
- void set_qvals (std::vector< topology::Vector2d > qvals)
  Sets qvals (redundant)
- void switches_from_vector (const std::vector< bool > &boolvec)
  Reads switches from vector.
- void switches_from_file (std::ifstream &infile, std::ofstream &stdoutfile)
  Reads switches from file.
- void all_switches_on ()
  Turns all switches on.
- void all_switches_off ()
  Turns all switches off.
- void print_snapshots_on ()
  Turns print_snaphshots_ switch on.
- void print_snapshots_off ()
  Turns print_snaphshots_ switch off.
- void check_on_fly_sampling ()

Turns all switches off if on_fly_sampling is off.

- std::vector< bool > get_switches () const

  Returns vector of sample switches. Is for checking.
- bool print_snapshots () const
- std::vector< topology::Vector2d > get_qvals () const
- void refresh_qvals (const topology::Vector2d &boxsize)

  Chooses new values for q (not recommended, relic of old implementation of qbin)
- std::vector< double > bin_qvals_to_q (std::vector< double > vals) const

  Sorts the qvals at which correlations were computed into a bin (double input)
- std::vector< std::complex< double > > bin_qvals_to_q (std::vector< std::complex< double > > vals) const

  Sorts the qvals at which correlations were computed into a bin (complex input)
- double get_last_temperature ()

  Returns last temperature that was calculated. Careful, no check if calcultaion was performed.
- void sample_MSD (const std::vector< double > &MSD)

  Samples MSD.
- void sample (const group &G, const group &G_new, double t)

  Samples all static properties and dynamic properties according to switches.
- void sample_static (const group &G, double t)

  Samples all static properties that have an activated sample switch.
- void sample_TCF (const group &G_initial, const group &G_new, const double t)

  Samples all time correlation functions that have an activated sample switch.
- void sample_snapshots (const group &G, const double t, std::string snapfile_name, std::ofstream &outfile)

  Samples (i.e. stores) a snapshot to a file.
- void average ()

  appends averages to the last entry of each nonempty sampling vector
- void print_matlab (std::ofstream &outfile)

  Prints in a format that can be directly read in from matlab.
- void print_averages_matlab (std::ofstream &outfile)

  Prints averages in a format that can be directly read in from matlab.

Protected Attributes

- parameters par_

  Simulation parameters.
- int NumberOfSwitches_ = 15

  Total number of switches. Could be useful.
- int nsamp_ = 0

  Number of samples (for time averages later on)
- int nsnap_ = 0

  Number of snapshots.
- std::vector< double > averaging_times_

  Stores sampling times (for averaging)
- std::vector< double > TCF_times_

  Stores sampling times (for time correlation function)
- std::vector< double > H_

  Stores energies.
- std::vector< double > H_2_

  Stores individual energies squared (<e_i^2>)
- std::vector< double > Hkin_2_

  Stores individual kinetic energies squared.

- std::vector< double > Hint_2_

  Stores individual interaction energies squared.
- std::vector< double > W_

  Stores momenta.
- std::vector< double > W_2_

  Stores momenta squared.
- std::vector< topology::Vector2d > M_

  Stores magnetizations.
- std::vector< double > M_2_

  Stores magnetizations squared.
- std::vector< double > M_4_

  Stores magnetizations to the fourth power.
- std::vector< double > absM_

  Stores absolute magnetizations.
- std::vector< double > M_angle_

  Stores angle of the magnetization (with respect to the spin x-axis)
- std::vector< double > Theta_

  Stores mean theta (with respect to the spin x-axis)
- std::vector< double > Theta_2_

  Stores mean theta^2 (with respect to the spin x-axis)
- std::vector< double > Theta_4_

  Stores mean theta^4 (with respect to the spin x-axis)
- std::vector< double > Theta_rel_to_M_

  Stores mean theta relative to orientation of magnetization.
- std::vector< double > Theta_rel_to_M_2_

  Stores mean theta^2 relative to orientation of magnetization.
- std::vector< double > Theta_rel_to_M_4_

  Stores mean theta^2 relative to orientation of magnetization.
- std::vector< double > temperature_

  Stores temperature.
- std::vector< double > temperature_squared_

  Stores temperature squared.
- std::vector< double > temperature_omega_

  Stores spin momentum temperature.
- std::vector< double > temperature_omega_squared_

  Stores spin momentum temperature squared.
- std::vector< double > temperature_p_

  Stores linear momentum temperature.
- std::vector< double > temperature_p_squared_

  Stores linear momentum temperature squared.
- std::vector< topology::Vector2d > P_

  Stores momentum components.
- std::vector< double > P_2_

  Stores momentum squared.
- std::vector< double > P_4_

  Stores momentum to the fourth power.
- std::vector< double > H_x_

  H_x as defined for the derivation of the helicity Upsilon.
- std::vector< double > H_y_

  H_y as defined for the derivation of the helicity Upsilon.
- std::vector< double > I_x_

l_x as defined for the derivation of the helicity Upsilon.

- std::vector< double > l_y_

  l_y as defined for the derivation of the helicity Upsilon.

- std::vector< double > l_x_2_

  l_x^2 as defined for the derivation of the helicity Upsilon.

- std::vector< double > l_y_2_

  l_y^2 as defined for the derivation of the helicity Upsilon.

- std::vector< double > Upsilon_

  Helicity Upsilon.

- std::vector< double > abs_vortices_

  Stores total vortex density.

- std::vector< double > signed_vortices_

  Stores total signed vortex density (i.e. positive minus negative density).

- std::vector< double > coordination_number_

  Coordination number in mobile model (avg. number of neighbors in interaction radius)

- std::vector< topology::Vector2d > qvals_

  Values of q for evaluation of field fluctuations (changes over sampling process)

- std::vector< std::complex< double > > mxq_

  Stores the $m_{x,q}$ (x-spin field fluctuation)

- std::vector< std::complex< double > > myq_

  Stores the $m_{y,q}$ (y-spin field fluctuation)

- std::vector< std::complex< double > > mparq_

  Stores the $m_{\parallel,q}$ (spin field fluctuation parallel to spontaneous M)

- std::vector< std::complex< double > > mperpq_

  Stores the $m_{\perp,q}$ (spin field fluctuation perpendicular to spontaneous M)

- std::vector< std::complex< double > > eq_

  Stores the e_q (energy field fluctuation)

- std::vector< std::complex< double > > wq_

  Stores the w_q (omega field fluctuation)

- std::vector< std::complex< double > > teq_

  Stores the θ_q (theta field fluctuation)

- std::vector< std::complex< double > > rq_

  Stores the ρ_q (density field fluctuation)

- std::vector< std::complex< double > > jparq_

  Stores the $j_{\parallel,q}$ (parallel momentum field fluctuation)

- std::vector< std::complex< double > > jperpq_

  Stores the $j_{\perp,q}$ (perpendicular momentum field fluctuation)

- std::vector< std::complex< double > > lq_

  Stores the l_q (spatial angular momentum field fluctuation)

- std::vector< std::complex< double > > mxq_cur_

  Stores the $m_{x,q}$ at the current time (avoids repeated computation, which is costly)

- std::vector< std::complex< double > > myq_cur_

  Stores the $m_{y,q}$ at the current time (avoids repeated computation, which is costly)

- std::vector< std::complex< double > > mparq_cur_

  Stores the $m_{\parallel,q}$ at the current time (avoids repeated computation, which is costly)

- std::vector< std::complex< double > > mperpq_cur_

  Stores the $m_{\perp,q}$ at the current time (avoids repeated computation, which is costly)

- std::vector< std::complex< double > > eq_cur_

  Stores the e_q at the current time (avoids repeated computation, which is costly)

- std::vector< std::complex< double > > wq_cur_

  Stores the w_q at the current time (avoids repeated computation, which is costly)

- std::vector< std::complex< double > > teq_cur_

  Stores the θ_q at the current time (avoids repeated computation, which is costly)
- std::vector< std::complex< double > > rq_cur_

  Stores the ρ_q at the current time.
- std::vector< std::complex< double > > jparq_cur_

  Stores the $j_{\parallel,q}$ at the current time.
- std::vector< std::complex< double > > jperpq_cur_

  Stores the $j_{\perp,q}$ at the current time.
- std::vector< std::complex< double > > lq_cur_

  Stores the l_q at the current time.
- std::vector< std::complex< double > > convol_wmx_cur_

  Stores the convolution $w_q \star m_{x,q}$.
- std::vector< std::complex< double > > convol_wmy_cur_

  Stores the convolution $w_q \star m_{y,q}$.
- std::vector< std::complex< double > > convol_jparmx_cur_

  Stores the convolution $j_{L,q} \star m_{x,q}$.
- std::vector< std::complex< double > > convol_jparmy_cur_

  Stores the convolution $j_{L,q} \star m_{y,q}$.
- std::vector< std::complex< double > > mxq_initial_

  Stores the $m_{x,q}$ at the initial time (avoids repeated computation, only usable if q is not refreshed)
- std::vector< std::complex< double > > myq_initial_

  Stores the $m_{y,q}$ at the initial time (avoids repeated computation, only usable if q is not refreshed)
- std::vector< std::complex< double > > mparq_initial_

  Stores the $m_{\parallel,q}$ at the initial time (avoids repeated computation, only usable if q is not refreshed)
- std::vector< std::complex< double > > mperpq_initial_

  Stores the $m_{\perp,q}$ at the initial time (avoids repeated computation, only usable if q is not refreshed)
- std::vector< std::complex< double > > eq_initial_

  Stores the e_q at the initial time (avoids repeated computation, only usable if q is not refreshed)
- std::vector< std::complex< double > > wq_initial_

  Stores the w_q at the initial time (avoids repeated computation, only usable if q is not refreshed)
- std::vector< std::complex< double > > teq_initial_

  Stores the θ_q at the initial time (avoids repeated computation, only usable if q is not refreshed)
- std::vector< std::complex< double > > rq_initial_

  Stores the ρ_q at the initial time (avoids repeated computation, only usable if q is not refreshed)
- std::vector< std::complex< double > > jparq_initial_

  Stores the $j_{\parallel,q}$ at the initial time (avoids repeated computation, only usable if q is not refreshed)
- std::vector< std::complex< double > > jperpq_initial_

  Stores the $j_{\perp,q}$ at the initial time (avoids repeated computation, only usable if q is not refreshed)
- std::vector< std::complex< double > > lq_initial_

  Stores the l_q at the initial time (avoids repeated computation, only usable if q is not refreshed)
- std::vector< std::complex< double > > convol_wmx_initial_

  Stores the convolution $w_q \star m_{x,q}$ at the initial time.
- std::vector< std::complex< double > > convol_wmy_initial_

  Stores the convolution $w_q \star m_{y,q}$ at the initial time.
- std::vector< std::complex< double > > convol_jparmx_initial_

  Stores the convolution $j_{\parallel,q} \star m_{x,q}$ at the initial time.
- std::vector< std::complex< double > > convol_jparmy_initial_

  Stores the convolution $j_{\parallel,q} \star m_{y,q}$ at the initial time.
- std::vector< double > chimxq_

  Stores the $\chi_{mx,q}$.
- std::vector< double > chimyq_

Stores the $\chi_{my,q}$.

- std::vector< double > chimparq_

  Stores the $\chi_{m\parallel,q}$.
- std::vector< double > chimperpq_

  Stores the $\chi_{m\perp,q}$.
- std::vector< double > chieq_

  Stores the $\chi_{e,q}$.
- std::vector< double > chiwq_

  Stores the $\chi_{w,q}$.
- std::vector< double > chiteq_

  Stores the $\chi_{\theta,q}$.
- std::vector< double > chirq_

  Stores the $\chi_{\rho,q}$.
- std::vector< double > chijparq_

  Stores the $\chi_{jL,q}$.
- std::vector< double > chijperpq_

  Stores the $\chi_{jT,q}$.
- std::vector< double > chilq_

  Stores the $\chi_{l,q}$.
- std::vector< std::complex< double > > SCFq_xy_

  Stores the $\langle m^*_{x,q} m_{y,q} \rangle$ static correlation.
- std::vector< std::complex< double > > SCFq_xw_

  Stores the $\langle m^*_{x,q} w_q \rangle$ static correlation.
- std::vector< std::complex< double > > SCFq_xe_

  Stores the $\langle m^*_{x,q} e_q \rangle$ static correlation.
- std::vector< std::complex< double > > SCFq_yw_

  Stores the $\langle m^*_{y,q} w_q \rangle$ static correlation.
- std::vector< std::complex< double > > SCFq_ye_

  Stores the $\langle m^*_{y,q} e_q \rangle$ static correlation.
- std::vector< std::complex< double > > SCFq_we_

  Stores the $\langle w^*_q e_q \rangle$ static correlation.
- std::vector< std::complex< double > > SCFq_mparmperp_

  Stores the $\langle m^*_{\parallel,q} m_{\perp,q} \rangle$ static correlation.
- std::vector< std::complex< double > > SCFq_re_

  Stores the $\langle r^*_q e_q \rangle$ static correlation.
- std::vector< double > SCF_Spin_

  Stores the static spin correlation function.
- std::vector< double > SCF_Spin_par_

  Stores the static spin correlation function.
- std::vector< double > SCF_Spin_perp_

  Stores the static spin correlation function.
- std::vector< double > SCF_anglediff_

  Stores the static angle difference correlation function.
- std::vector< double > SCF_W_

  Stores the static spin momentum correlation function.
- std::vector< double > SCF_P_

  Stores the static linear momentum correlation function.
- std::vector< double > SCF_g_

  Stores the pair distribution function.
- std::vector< double > SCF_E_

  Stores the static total energy correlation function.

- std::vector< double > SCF_Ekin_

  Stores the static kinetic energy correlation function.
- std::vector< double > SCF_Eint_

  Stores the static interaction energy correlation function.
- std::vector< double > ACF_Spin_

  Stores the spin autocorrelation function.
- std::vector< double > ACF_anglediff_

  Stores the angle difference autocorrelation function.
- std::vector< double > ACF_q0_M_

  Stores the magnetization autocorrelation function (q=0 limit)
- std::vector< double > ACF_q0_absM_

  Stores the autocorrelation function of the absolute magnetization (q=0 limit), basically the limit q=0 for mparq.
- std::vector< double > ACF_Sx_

  Stores the spin autocorrelation function in x direction.
- std::vector< double > ACF_Sy_

  Stores the spin autocorrelation function in y direction.
- std::vector< double > ACF_W_

  Stores the omega autocorrelation function in y direction.
- std::vector< double > ACF_E_

  Stores the energy autocorrelation function in y direction.
- std::vector< double > ACF_Eint_

  Stores the interaction energy autocorrelation function in y direction.
- std::vector< double > ACF_Ekin_

  Stores the kinetic energy autocorrelation function in y direction.
- std::vector< double > ACF_P_

  Stores the momentum autocorrelation function in y direction.
- std::vector< double > ACF_Ppar_

  Stores the momentum autocorrelation function parallel to the magnetization.
- std::vector< double > ACF_Pperp_

  Stores the spin autocorrelation function perpendicular to the magnetization.
- std::vector< double > ACF_MSD_

  Stores the positional mean squared displacement.
- std::vector< double > ACF_ang_MSD_

  Stores the angular mean squared displacement.
- std::vector< double > MSD_aux_

  Auxiliary MSD storage. Stores all distances and angles (depending on group type) that the particles have travelled.
- std::vector< std::complex< double > > gxx_

  Stores the $C_{xx}(q,t) = g_{x,x,q}(t) = \langle m^*_{x,q} m_{x,q}(t) \rangle$ *(time correlation function)*
- std::vector< std::complex< double > > gxy_

  Stores the $C_{xy}(q,t) = g_{x,y,q}(t) = \langle m^*_{x,q} m_{y,q}(t) \rangle$ *(time correlation function)*
- std::vector< std::complex< double > > gxw_

  Stores the $C_{xw}(q,t) = g_{x,w,q}(t) = \langle m^*_{x,q} w_q(t) \rangle$ *(time correlation function)*
- std::vector< std::complex< double > > gxe_

  Stores the $C_{we}(q,t) = g_{x,e,q}(t) = \langle m^*_{x,q} e_q(t) \rangle$ *(time correlation function)*
- std::vector< std::complex< double > > gyy_

  Stores the $C_{yy}(q,t) = g_{y,y,q}(t) = \langle m^*_{y,q} m_{y,q}(t) \rangle$ *(time correlation function)*
- std::vector< std::complex< double > > gyw_

  Stores the $C_{yw}(q,t) = g_{y,w,q}(t) = \langle m^*_{y,q} w_q(t) \rangle$ *(time correlation function)*
- std::vector< std::complex< double > > gye_

  Stores the $C_{ye}(q,t) = g_{y,e,q}(t) = \langle m^*_{y,q} e_q(t) \rangle$ *(time correlation function)*
- std::vector< std::complex< double > > gww_

Stores the $C_{ww}(q,t) = g_{w,w,q}(t) = \langle w_q^* w_q(t) \rangle$ (time correlation function)

- std::vector< std::complex< double > > gwe_

  Stores the $C_{we}(q,t) = g_{w,e,q}(t) = \langle w_q^* e_q(t) \rangle$ (time correlation function)
- std::vector< std::complex< double > > gee_

  Stores the $C_{ee}(q,t) = g_{e,e,q}(t) = \langle e_q^* e_q(t) \rangle$ (time correlation function)
- std::vector< std::complex< double > > gmparmpar_

  Stores the $C_{m\parallel,m\parallel}(q,t) = g_{m\parallel,m\parallel,q}(t) = \langle m_{\parallel,q}^* m_{\parallel,q}(t) \rangle$ (time correlation function)
- std::vector< std::complex< double > > gmperpmperp_

  Stores the $C_{m\perp,m\perp}(q,t) = g_{m\perp,m\perp,q}(t) = \langle m_{\perp,q}^* m_{\perp,q}(t) \rangle$ (time correlation function)
- std::vector< std::complex< double > > gmparmperp_

  Stores the $C_{m\parallel,m\perp}(q,t) = g_{m\parallel,m\perp,q}(t) = \langle m_{\perp,q}^* m_{\parallel,q}(t) \rangle$ (time correlation function)
- std::vector< std::complex< double > > gre_

  Stores the $C_{\rho e}(q,t) = g_{r,e,q}(t) = \langle \rho_q^* e_q(t) \rangle$ (time correlation function)
- std::vector< std::complex< double > > grr_

  Stores the $C_{\rho\rho}(q,t) = g_{r,r,q}(t) = \langle \rho_q^* \rho_q(t) \rangle$ (time correlation function, intermediate scattering function)
- std::vector< std::complex< double > > gjparjpar_

  Stores the $C_{jL,jL}(q,t) = g_{j\parallel,j\parallel,q}(t) = \langle j_{L,q}^* j_{L,q}(t) \rangle$ (time correlation function)
- std::vector< std::complex< double > > gjperpjperp_

  Stores the $C_{jT,jT}(q,t) = g_{j\perp,j\perp,q}(t) = \langle j_{T,q}^* j_{T,q}(t) \rangle$ (time correlation function)
- std::vector< std::complex< double > > gll_

  Stores the $C_{ll}(q,t) = g_{l,l,q}(t) = \langle l_q^* l_q(t) \rangle$ (time correlation function)
- std::vector< std::complex< double > > gtt_

  Stores the $C_{\theta\theta}(q,t) = g_{\theta,\theta,q}(t) = \langle \theta_q^* \theta_q(t) \rangle$ (time correlation function)
- std::vector< double > TransCoeff_J1_

  Transport Coefficient: $\langle J(r_{ij}) \rangle$ (see old notes, not included in thesis)
- std::vector< double > TransCoeff_J1_cos1_

  Transport Coefficient: $\langle J(r_{ij}) cos(te_{ij}) \rangle$ (see old notes, not included in thesis)
- std::vector< double > TransCoeff_J1_cos2_

  Transport Coefficient: $\langle J(r_{ij}) cos^2(te_{ij}) \rangle$ (see old notes, not included in thesis)
- std::vector< double > TransCoeff_J1_cos2_r2_

  Transport Coefficient: $\langle J(r_{ij}) cos^2(te_{ij}) r_{ij}^2 \rangle$ (see old notes, not included in thesis)
- std::vector< double > TransCoeff_J1_cos1_r2_

  Transport Coefficient: $\langle J(r_{ij}) cos(te_{ij}) r_{ij}^2 \rangle$ (see old notes, not included in thesis)
- std::vector< double > TransCoeff_J1_sin1_te_

  Transport Coefficient: $\langle J(r_{ij}) sin(te_{ij}) te_{ij} \rangle$ (see old notes, not included in thesis)
- std::vector< double > TransCoeff_Up_

  Transport Coefficient: $\langle \mathcal{U}'(r_{ij}) \rangle$ (see old notes, not included in thesis)
- std::vector< double > TransCoeff_Up_rinv_

  Transport Coefficient: $\langle \mathcal{U}'(r_{ij}) r_{ij}^{-1} \rangle$ (see old notes, not included in thesis)
- std::vector< double > TransCoeff_Upp_

  Transport Coefficient: $\langle \mathcal{U}''(r_{ij}) \rangle$ (see old notes, not included in thesis)
- std::vector< double > TransCoeff_Up_cos1_

  Transport Coefficient: $\langle \mathcal{U}'(r_{ij}) cos(te_{ij}) \rangle$ (see old notes, not included in thesis)
- std::vector< double > TransCoeff_Up_rinv_cos1_

  Transport Coefficient: $\langle \mathcal{U}'(r_{ij}) r_{ij}^{-1} cos(te_{ij}) \rangle$ (see old notes, not included in thesis)
- std::vector< double > TransCoeff_Upp_cos1_

  Transport Coefficient: $\langle \mathcal{U}''(r_{ij}) cos(te_{ij}) \rangle$ (see old notes, not included in thesis)
- std::vector< double > TransCoeff_cos1_

  Transport Coefficient: $\langle cos(te_{ij}) \rangle$ (see old notes, not included in thesis)
- std::vector< double > TransCoeff_Up_rinv_te2_

  Transport Coefficient: $\langle \mathcal{U}'(r_{ij}) r_{ij}^{-1} te_{ij}^2 \rangle$ (see old notes, not included in thesis)

- std::vector< double > TransCoeff_Upp_te2_

  Transport Coefficient: $\langle \mathcal{U}"(r_{ij}) te_{ij}^2 \rangle$ (see old notes, not included in thesis)
- std::vector< double > TransCoeff_times_

  Stores times used in calculation of transport coefficients.
- std::vector< topology::Vector2d > tau_

  Stores momentum flux tau, which is the q to 0 limit of τ'_q in eq. (1.31b). Required for ZM transport Coefficients. (see old notes, not included in thesis)
- std::vector< topology::Vector2d > je_

  Stores energy flux je, which is the q to 0 limit of $j'^\wedge e_q$ in eq. (1.31a). Required for ZM transport Coefficients. (see old notes, not included in thesis)
- topology::Vector2d tau_initial_

  Stores initial momentum flux tau (see old notes, not included in thesis)
- topology::Vector2d je_initial_

  Stores initial energy flux tau (see old notes, not included in thesis)
- std::vector< double > kappa_TransCoeff_

  Stores kappa transport coefficient, see Bissinger notes (sec. 1.2.7), (see old notes, not included in thesis)
- std::vector< double > Gamma_TransCoeff_

  Stores Gamma transport coefficient, see Bissinger notes (sec. 1.2.7), (see old notes, not included in thesis)
- std::vector< double > kappa_TransCoeff_new_

  Stores different kappa transport coefficient, see Bissinger notes (sec. 1.2.7), (see old notes, not included in thesis)
- std::vector< double > Gamma_TransCoeff_new_

  Stores different Gamma transport coefficient, see Bissinger notes (sec. 1.2.7), (see old notes, not included in thesis)
- std::vector< std::complex< double > > K_convol_wmx_

  Stores the memory kernel of the convolution $w_q \star m_{x,q}$ with itself.
- std::vector< std::complex< double > > K_convol_wmy_

  Stores the memory kernel of the convolution $w_q \star m_{y,q}$ with itself.
- std::vector< std::complex< double > > K_convol_jmx_

  Stores the memory kernel of the convolution $j_{L,q} \star m_{x,q}$ with itself.
- std::vector< std::complex< double > > K_convol_jmy_

  Stores the memory kernel of the convolution $j_{L,q} \star m_{y,q}$ with itself.
- std::vector< std::complex< double > > K_convol_wmx_jmy_

  Stores the memory kernel of the convolution $w_q \star m_{x,q}$ with $j_{L,q} \star m_{y,q}$.
- std::vector< std::complex< double > > K_convol_wmy_jmx_

  Stores the memory kernel of the convolution $w_q \star mxy$ with $j_{L,q} \star m_{x,q}$.
- std::vector< std::complex< double > > K_wmx_

  Stores the memory kernel of the quadratic form $w_q m_{x,q}$ with itself.
- std::vector< std::complex< double > > K_wmy_

  Stores the memory kernel of the quadratic form $w_q m_{y,q}$ with itself.
- std::vector< std::complex< double > > K_jmx_

  Stores the memory kernel of the quadratic form $jqparm_{x,q}$ with itself.
- std::vector< std::complex< double > > K_jmy_

  Stores the memory kernel of the quadratic form $jqparm_{y,q}$ with itself.
- std::vector< std::complex< double > > K_wmx_jmy_

  Stores the memory kernel of the quadratic form $w_q \star m_{x,q}$ with the quadratic form $j_{L,q} m_{y,q}$.
- std::vector< std::complex< double > > K_wmy_jmx_

  Stores the memory kernel of the quadratic form $w_q \star mxy$ with the quadratic form $j_{L,q} m_{x,q}$.
- bool store_static_ = 0

  Decides whether or not static quantities are calculated.
- bool store_vortices_ = 0

  Decides whether or not vortices_ is calculated.
- bool store_mxq_ = 0

Decides whether or not $m_{x,q}$ is calculated.

- bool store_myq_ = 0

  Decides whether or not $m_{y,q}$ is calculated.
- bool store_eq_ = 0

  Decides whether or not e_q is calculated.
- bool store_wq_ = 0

  Decides whether or not w_q is calculated.
- bool store_rq_ = 0

  Decides whether or not ρ_q is calculated.
- bool store_jparq_ = 0

  Decides whether or not $j_{L,q}$ is calculated.
- bool store_jperpq_ = 0

  Decides whether or not $j_{T,q}$ is calculated.
- bool store_lq_ = 0

  Decides whether or not l_q is calculated.
- bool store_teq_ = 0

  Decides whether or not θ_q is calculated.
- bool store_SCF_ = 0

  Decides whether or not the static correlation functions are calculated.
- bool store_TCF_ = 0

  Decides whether or not the time correlation functions (autocorrelations and so forth) are calculated.
- bool refresh_q_ = 0

  Decides whether or not to choose new random q values for each sampling.
- bool store_TransCoeff_ = 0

  Decides whether or not the transport coefficients are calculated.
- bool store_MemoryKernels_ = 0

  Decides whether or not the memory kernels (MCT) are calculated.
- bool print_snapshots_ = 0

  Decides whether trajectories should be sampled.

6.7.1 Detailed Description

Stores and handles all data sampling performed during a run or in a later diagnostic.

Has a wide variety of quantities that may be of physical interest during the run. Can compute these quantities when called upon and stores them in vecotrs. Most of these vectors can get time averaged at the end of the run.

All results are printed in matlab-readable form and can be further processed with matlab code.

TODO: Alternate output than matlab, ideally binary or CSV.

Date

Created on 2020-01-20

Author

Thomas Bissinger

6.7.2 Constructor & Destructor Documentation

6.7.2.1 sampler() [1/3]

```
sampler::sampler ( )  [inline]
```

Empty constructor.

6.7.2.2 sampler() [2/3]

```
sampler::sampler (
            parameters par )  [inline]
```

Constructor. Assigns par_ to par.

6.7.2.3 sampler() [3/3]

```
sampler::sampler (
            parameters par,
            std::vector< topology::Vector2d > qvals )  [inline]
```

Constructor. Assigns par_ to par and qvals_ to qvals.

6.7.3 Member Function Documentation

6.7.3.1 all_switches_off()

```
void sampler::all_switches_off ( )
```

Turns all switches off.

6.7.3.2 all_switches_on()

```
void sampler::all_switches_on ( )
```

Turns all switches on.

6.7.3.3 average()

```
void sampler::average ( )
```

appends averages to the last entry of each nonempty sampling vector

6.7.3.4 bin_qvals_to_q() [1/2]

```
std::vector< double > sampler::bin_qvals_to_q (
            std::vector< double > vals ) const
```

Sorts the qvals at which correlations were computed into a bin (double input)

6.7.3.5 bin_qvals_to_q() [2/2]

```
std::vector< std::complex< double > > sampler::bin_qvals_to_q (
            std::vector< std::complex< double > > vals ) const
```

Sorts the qvals at which correlations were computed into a bin (complex input)

6.7.3.6 check_on_fly_sampling()

```
void sampler::check_on_fly_sampling ( )
```

Turns all switches off if on_fly_sampling is off.

6.7.3.7 get_last_temperature()

```
double sampler::get_last_temperature ( )  [inline]
```

Returns last temperature that was calculated. Careful, no check if calcultaion was performed.

6.7.3.8 get_qvals()

```
std::vector< topology::Vector2d > sampler::get_qvals ( ) const  [inline]
```

6.7.3.9 get_switches()

```
std::vector< bool > sampler::get_switches ( ) const
```

Returns vector of sample switches. Is for checking.

6.7.3.10 print_averages_matlab()

```
void sampler::print_averages_matlab (
            std::ofstream & outfile )
```

Prints averages in a format that can be directly read in from matlab.

6.7.3.11 print_matlab()

```
void sampler::print_matlab (
            std::ofstream & outfile )
```

Prints in a format that can be directly read in from matlab.

6.7.3.12 print_snapshots()

```
bool sampler::print_snapshots ( ) const  [inline]
```

6.7.3.13 print_snapshots_off()

```
void sampler::print_snapshots_off ( )
```

Turns print_snaphshots_ switch off.

6.7.3.14 print_snapshots_on()

```
void sampler::print_snapshots_on ( )
```

Turns print_snaphshots_ switch on.

6.7.3.15 refresh_qvals()

```
void sampler::refresh_qvals (
            const topology::Vector2d & boxsize )
```

Chooses new values for q (not recommended, relic of old implementation of qbin)

6.7.3.16 sample()

```
void sampler::sample (
            const group & G,
            const group & G_new,
            double t )
```

Samples all static properties and dynamic properties according to switches.

6.7.3.17 sample_MSD()

```
void sampler::sample_MSD (
            const std::vector< double > & MSD )
```

Samples MSD.

6.7.3.18 sample_snapshots()

```
void sampler::sample_snapshots (
            const group & G,
            const double t,
            std::string snapfile_name,
            std::ofstream & outfile )
```

Samples (i.e. stores) a snapshot to a file.

6.7.3.19 sample_static()

```
void sampler::sample_static (
            const group & G,
            double t )
```

Samples all static properties that have an activated sample switch.

6.7.3.20 sample_TCF()

```
void sampler::sample_TCF (
            const group & G_initial,
            const group & G_new,
            const double t )
```

Samples all time correlation functions that have an activated sample switch.

6.7.3.21 set_parameters()

```
void sampler::set_parameters (
            parameters par )
```

Sets parameter member variable.

6.7.3.22 set_qvals()

```
void sampler::set_qvals (
            std::vector< topology::Vector2d > qvals )
```

Sets qvals (redundant)

6.7.3.23 switches_from_file()

```
void sampler::switches_from_file (
            std::ifstream & infile,
            std::ofstream & stdoutfile )
```

Reads switches from file.

6.7.3.24 switches_from_vector()

```
void sampler::switches_from_vector (
            const std::vector< bool > & boolvec )
```

Reads switches from vector.

6.7.4 Member Data Documentation

6.7.4.1 abs_vortices_

```
std::vector<double> sampler::abs_vortices_  [protected]
```

Stores total vortex density.

6.7.4.2 absM_

```
std::vector<double> sampler::absM_  [protected]
```

Stores absolute magnetizations.

6.7.4.3 ACF_ang_MSD_

```
std::vector<double> sampler::ACF_ang_MSD_  [protected]
```

Stores the angular mean squared displacement.

6.7.4.4 ACF_anglediff_

```
std::vector<double> sampler::ACF_anglediff_  [protected]
```

Stores the angle difference autocorrelation function.

6.7.4.5 ACF_E_

```
std::vector<double> sampler::ACF_E_  [protected]
```

Stores the energy autocorrelation function in y direction.

6.7.4.6 ACF_Eint_

```
std::vector<double> sampler::ACF_Eint_  [protected]
```

Stores the interaction energy autocorrelation function in y direction.

6.7.4.7 ACF_Ekin_

```
std::vector<double> sampler::ACF_Ekin_  [protected]
```

Stores the kinetic energy autocorrelation function in y direction.

6.7.4.8 ACF_MSD_

```
std::vector<double> sampler::ACF_MSD_  [protected]
```

Stores the positional mean squared displacement.

6.7.4.9 ACF_P_

```
std::vector<double> sampler::ACF_P_  [protected]
```

Stores the momentum autocorrelation function in y direction.

6.7.4.10 ACF_Ppar_

```
std::vector<double> sampler::ACF_Ppar_  [protected]
```

Stores the momentum autocorrelation function parallel to the magnetization.

6.7.4.11 ACF_Pperp_

```
std::vector<double> sampler::ACF_Pperp_  [protected]
```

Stores the spin autocorrelation function perpendicular to the magnetization.

6.7.4.12 ACF_q0_absM_

```
std::vector<double> sampler::ACF_q0_absM_  [protected]
```

Stores the autocorrelation function of the absolute magnetization (q=0 limit), basically the limit q=0 for mparq.

6.7.4.13 ACF_q0_M_

```
std::vector<double> sampler::ACF_q0_M_  [protected]
```

Stores the magnetization autocorrelation function (q=0 limit)

6.7.4.14 ACF_Spin_

```
std::vector<double> sampler::ACF_Spin_  [protected]
```

Stores the spin autocorrelation function.

6.7.4.15 ACF_Sx_

```
std::vector<double> sampler::ACF_Sx_ [protected]
```

Stores the spin autocorrelation function in x direction.

6.7.4.16 ACF_Sy_

```
std::vector<double> sampler::ACF_Sy_ [protected]
```

Stores the spin autocorrelation function in y direction.

6.7.4.17 ACF_W_

```
std::vector<double> sampler::ACF_W_ [protected]
```

Stores the omega autocorrelation function in y direction.

6.7.4.18 averaging_times_

```
std::vector<double> sampler::averaging_times_ [protected]
```

Stores sampling times (for averaging)

6.7.4.19 chieq_

```
std::vector<double> sampler::chieq_ [protected]
```

Stores the $\chi_{e,q}$.

6.7.4.20 chijparq_

```
std::vector<double> sampler::chijparq_ [protected]
```

Stores the $\chi_{jL,q}$.

6.7.4.21 chijperpq_

```
std::vector<double> sampler::chijperpq_ [protected]
```

Stores the $\chi_{jT,q}$.

6.7.4.22 chilq_

```
std::vector<double> sampler::chilq_ [protected]
```

Stores the $\chi_{l,q}$.

6.7.4.23 chimparq_

```
std::vector<double> sampler::chimparq_ [protected]
```

Stores the $\chi_{m\parallel,q}$.

6.7.4.24 chimperpq_

```
std::vector<double> sampler::chimperpq_ [protected]
```

Stores the $\chi_{m\perp,q}$.

6.7.4.25 chimxq_

```
std::vector<double> sampler::chimxq_ [protected]
```

Stores the $\chi_{mx,q}$.

6.7.4.26 chimyq_

```
std::vector<double> sampler::chimyq_ [protected]
```

Stores the $\chi_{my,q}$.

6.7.4.27 chirq_

```
std::vector<double> sampler::chirq_ [protected]
```

Stores the $\chi_{\rho,q}$.

6.7.4.28 chiteq_

```
std::vector<double> sampler::chiteq_ [protected]
```

Stores the $\chi_{\theta,q}$.

6.7.4.29 chiwq_

```
std::vector<double> sampler::chiwq_ [protected]
```

Stores the $\chi_{w,q}$.

6.7.4.30 convol_jparmx_cur_

```
std::vector<std::complex<double> > sampler::convol_jparmx_cur_ [protected]
```

Stores the convolution $j_{L,q} \star m_{x,q}$.

6.7.4.31 convol_jparmx_initial_

```
std::vector<std::complex<double> > sampler::convol_jparmx_initial_ [protected]
```

Stores the convolution $j_{\parallel,q} \star m_{x,q}$ at the initial time.

6.7.4.32 convol_jparmy_cur_

```
std::vector<std::complex<double> > sampler::convol_jparmy_cur_ [protected]
```

Stores the convolution $j_{L,q} \star m_{y,q}$.

6.7.4.33 convol_jparmy_initial_

```
std::vector<std::complex<double> > sampler::convol_jparmy_initial_ [protected]
```

Stores the convolution $j_{\parallel,q} \star m_{y,q}$ at the initial time.

6.7.4.34 convol_wmx_cur_

```
std::vector<std::complex<double> > sampler::convol_wmx_cur_ [protected]
```

Stores the convolution $w_q \star m_{x,q}$.

6.7.4.35 convol_wmx_initial_

```
std::vector<std::complex<double> > sampler::convol_wmx_initial_ [protected]
```

Stores the convolution $w_q \star m_{x,q}$ at the initial time.

6.7.4.36 convol_wmy_cur_

```
std::vector<std::complex<double> > sampler::convol_wmy_cur_ [protected]
```

Stores the convolution $w_q \star m_{y,q}$.

6.7.4.37 convol_wmy_initial_

```
std::vector<std::complex<double> > sampler::convol_wmy_initial_ [protected]
```

Stores the convolution $w_q \star m_{y,q}$ at the initial time.

6.7.4.38 coordination_number_

```
std::vector<double> sampler::coordination_number_ [protected]
```

Coordination number in mobile model (avg. number of neighbors in interaction radius)

6.7.4.39 eq_

```
std::vector<std::complex<double> > sampler::eq_ [protected]
```

Stores the e_q (energy field fluctuation)

6.7.4.40 eq_cur_

```
std::vector<std::complex<double> > sampler::eq_cur_ [protected]
```

Stores the e_q at the current time (avoids repeated computation, which is costly)

6.7.4.41 eq_initial_

```
std::vector<std::complex<double> > sampler::eq_initial_ [protected]
```

Stores the e_q at the initial time (avoids repeated computation, only usable if q is not refreshed)

6.7.4.42 Gamma_TransCoeff_

```
std::vector<double> sampler::Gamma_TransCoeff_ [protected]
```

Stores Gamma transport coefficient, see Bissinger notes (sec. 1.2.7), (see old notes, not included in thesis)

6.7.4.43 Gamma_TransCoeff_new_

```
std::vector<double> sampler::Gamma_TransCoeff_new_ [protected]
```

Stores different Gamma transport coefficient, see Bissinger notes (sec. 1.2.7), (see old notes, not included in thesis)

6.7.4.44 gee_

```
std::vector<std::complex<double> > sampler::gee_ [protected]
```

Stores the $C_{ee}(q,t) = g_{e,e,q}(t) = \langle e_q^* e_q(t) \rangle$ (time correlation function)

6.7.4.45 gjparjpar_

```
std::vector<std::complex<double> > sampler::gjparjpar_ [protected]
```

Stores the $C_{jL,jL}(q,t) = g_{j\parallel,j\parallel,q}(t) = \langle j_{L,q}^* j_{L,q}(t) \rangle$ (time correlation function)

6.7.4.46 gjperpjperp_

```
std::vector<std::complex<double> > sampler::gjperpjperp_ [protected]
```

Stores the $C_{jT,jT}(q,t) = g_{j\perp,j\perp,q}(t) = \langle j_{T,q}^* j_{T,q}(t) \rangle$ (time correlation function)

6.7.4.47 gll_

`std::vector<std::complex<double> > sampler::gll_ [protected]`

Stores the $C_{ll}(q,t) = g_{l,l,q}(t) = \langle l_q^* l_q(t) \rangle$ (time correlation function)

6.7.4.48 gmparmpar_

`std::vector<std::complex<double> > sampler::gmparmpar_ [protected]`

Stores the $C_{m\parallel,m\parallel}(q,t) = g_{m\parallel,m\parallel,q}(t) = \langle m_{\parallel,q}^* m_{\parallel,q}(t) \rangle$ (time correlation function)

6.7.4.49 gmparmperp_

`std::vector<std::complex<double> > sampler::gmparmperp_ [protected]`

Stores the $C_{m\parallel,m\perp}(q,t) = g_{m\parallel,m\perp,q}(t) = \langle m_{\perp,q}^* m_{\parallel,q}(t) \rangle$ (time correlation function)

6.7.4.50 gmperpmperp_

`std::vector<std::complex<double> > sampler::gmperpmperp_ [protected]`

Stores the $C_{m\perp,m\perp}(q,t) = g_{m\perp,m\perp,q}(t) = \langle m_{\perp,q}^* m_{\perp,q}(t) \rangle$ (time correlation function)

6.7.4.51 gre_

`std::vector<std::complex<double> > sampler::gre_ [protected]`

Stores the $C_{\rho e}(q,t) = g_{r,e,q}(t) = \langle \rho_q^* e_q(t) \rangle$ (time correlation function)

6.7.4.52 grr_

`std::vector<std::complex<double> > sampler::grr_ [protected]`

Stores the $C_{\rho\rho}(q,t) = g_{r,r,q}(t) = \langle \rho_q^* \rho_q(t) \rangle$ (time correlation function, intermediate scattering function)

6.7.4.53 gtt_

`std::vector<std::complex<double> > sampler::gtt_ [protected]`

Stores the $C_{\theta\theta}(q,t) = g_{\theta,\theta,q}(t) = \langle \theta_q^* \theta_q(t) \rangle$ (time correlation function)

6.7.4.54 gwe_

`std::vector<std::complex<double> > sampler::gwe_ [protected]`

Stores the $C_{we}(q,t) = g_{w,e,q}(t) = \langle w_q^* e_q(t) \rangle$ (time correlation function)

6.7.4.55 gww_

```
std::vector<std::complex<double> > sampler::gww_ [protected]
```

Stores the $C_{ww}(q,t) = g_{w,w,q}(t) = \langle w_q^* w_q(t) \rangle$ (time correlation function)

6.7.4.56 gxe_

```
std::vector<std::complex<double> > sampler::gxe_ [protected]
```

Stores the $C_{we}(q,t) = g_{x,e,q}(t) = \langle m_{x,q}^* e_q(t) \rangle$ (time correlation function)

6.7.4.57 gxw_

```
std::vector<std::complex<double> > sampler::gxw_ [protected]
```

Stores the $C_{xw}(q,t) = g_{x,w,q}(t) = \langle m_{x,q}^* w_q(t) \rangle$ (time correlation function)

6.7.4.58 gxx_

```
std::vector<std::complex<double> > sampler::gxx_ [protected]
```

Stores the $C_{xx}(q,t) = g_{x,x,q}(t) = \langle m_{x,q}^* m_{x,q}(t) \rangle$ (time correlation function)

6.7.4.59 gxy_

```
std::vector<std::complex<double> > sampler::gxy_ [protected]
```

Stores the $C_{xy}(q,t) = g_{x,y,q}(t) = \langle m_{x,q}^* m_{y,q}(t) \rangle$ (time correlation function)

6.7.4.60 gye_

```
std::vector<std::complex<double> > sampler::gye_ [protected]
```

Stores the $C_{ye}(q,t) = g_{y,e,q}(t) = \langle m_{y,q}^* e_q(t) \rangle$ (time correlation function)

6.7.4.61 gyw_

```
std::vector<std::complex<double> > sampler::gyw_ [protected]
```

Stores the $C_{yw}(q,t) = g_{y,w,q}(t) = \langle m_{y,q}^* w_q(t) \rangle$ (time correlation function)

6.7.4.62 gyy_

```
std::vector<std::complex<double> > sampler::gyy_ [protected]
```

Stores the $C_{yy}(q,t) = g_{y,y,q}(t) = \langle m_{y,q}^* m_{y,q}(t) \rangle$ (time correlation function)

6.7.4.63 H_

```
std::vector<double> sampler::H_ [protected]
```

Stores energies.

6.7.4.64 H_2_

```
std::vector<double> sampler::H_2_ [protected]
```

Stores individual energies squared (<e_i^2>)

6.7.4.65 H_x_

```
std::vector<double> sampler::H_x_ [protected]
```

H_x as defined for the derivation of the helicity Upsilon.

6.7.4.66 H_y_

```
std::vector<double> sampler::H_y_ [protected]
```

H_y as defined for the derivation of the helicity Upsilon.

6.7.4.67 Hint_2_

```
std::vector<double> sampler::Hint_2_ [protected]
```

Stores individual interaction energies squared.

6.7.4.68 Hkin_2_

```
std::vector<double> sampler::Hkin_2_ [protected]
```

Stores individual kinetic energies squared.

6.7.4.69 I_x_

```
std::vector<double> sampler::I_x_ [protected]
```

I_x as defined for the derivation of the helicity Upsilon.

6.7.4.70 I_x_2_

```
std::vector<double> sampler::I_x_2_ [protected]
```

I_x^2 as defined for the derivation of the helicity Upsilon.

6.7.4.71 I_y_

```
std::vector<double> sampler::I_y_ [protected]
```

I_y as defined for the derivation of the helicity Upsilon.

6.7.4.72 I_y_2_

```
std::vector<double> sampler::I_y_2_ [protected]
```

I_y^2 as defined for the derivation of the helicity Upsilon.

6.7.4.73 je_

```
std::vector<topology::Vector2d> sampler::je_ [protected]
```

Stores energy flux je, which is the q to 0 limit of $\j'^e_q$ in eq. (1.31a). Required for ZM transport Coefficients. (see old notes, not included in thesis)

6.7.4.74 je_initial_

```
topology::Vector2d sampler::je_initial_ [protected]
```

Stores initial energy flux tau (see old notes, not included in thesis)

6.7.4.75 jparq_

```
std::vector<std::complex<double> > sampler::jparq_ [protected]
```

Stores the $j_{\parallel,q}$ (parallel momentum field fluctuation)

6.7.4.76 jparq_cur_

```
std::vector<std::complex<double> > sampler::jparq_cur_ [protected]
```

Stores the $j_{\parallel,q}$ at the current time.

6.7.4.77 jparq_initial_

```
std::vector<std::complex<double> > sampler::jparq_initial_ [protected]
```

Stores the $j_{\parallel,q}$ at the initial time (avoids repeated computation, only usable if q is not refreshed)

6.7.4.78 jperpq_

`std::vector<std::complex<double> > sampler::jperpq_ [protected]`

Stores the $j_{\perp,q}$ (perpendicular momentum field fluctuation)

6.7.4.79 jperpq_cur_

`std::vector<std::complex<double> > sampler::jperpq_cur_ [protected]`

Stores the $j_{\perp,q}$ at the current time.

6.7.4.80 jperpq_initial_

`std::vector<std::complex<double> > sampler::jperpq_initial_ [protected]`

Stores the $j_{\perp,q}$ at the initial time (avoids repeated computation, only usable if q is not refreshed)

6.7.4.81 K_convol_jmx_

`std::vector<std::complex<double> > sampler::K_convol_jmx_ [protected]`

Stores the memory kernel of the convolution $j_{L,q} \star m_{x,q}$ with itself.

6.7.4.82 K_convol_jmy_

`std::vector<std::complex<double> > sampler::K_convol_jmy_ [protected]`

Stores the memory kernel of the convolution $j_{L,q} \star m_{y,q}$ with itself.

6.7.4.83 K_convol_wmx_

`std::vector<std::complex<double> > sampler::K_convol_wmx_ [protected]`

Stores the memory kernel of the convolution $w_q \star m_{x,q}$ with itself.

6.7.4.84 K_convol_wmx_jmy_

`std::vector<std::complex<double> > sampler::K_convol_wmx_jmy_ [protected]`

Stores the memory kernel of the convolution $w_q \star m_{x,q}$ with $j_{L,q} \star m_{y,q}$.

6.7.4.85 K_convol_wmy_

`std::vector<std::complex<double> > sampler::K_convol_wmy_ [protected]`

Stores the memory kernel of the convolution $w_q \star m_{y,q}$ with itself.

6.7.4.86 K_convol_wmy_jmx_

```
std::vector<std::complex<double> > sampler::K_convol_wmy_jmx_ [protected]
```

Stores the memory kernel of the convolution $w_q \star mxy$ with $j_{L,q} \star m_{x,q}$.

6.7.4.87 K_jmx_

```
std::vector<std::complex<double> > sampler::K_jmx_ [protected]
```

Stores the memory kernel of the quadratic form $jqparm_{x,q}$ with itself.

6.7.4.88 K_jmy_

```
std::vector<std::complex<double> > sampler::K_jmy_ [protected]
```

Stores the memory kernel of the quadratic form $jqparm_{y,q}$ with itself.

6.7.4.89 K_wmx_

```
std::vector<std::complex<double> > sampler::K_wmx_ [protected]
```

Stores the memory kernel of the quadratic form $w_q m_{x,q}$ with itself.

6.7.4.90 K_wmx_jmy_

```
std::vector<std::complex<double> > sampler::K_wmx_jmy_ [protected]
```

Stores the memory kernel of the quadratic form $w_q \star m_{x,q}$ with the quadratic form $j_{L,q} m_{y,q}$.

6.7.4.91 K_wmy_

```
std::vector<std::complex<double> > sampler::K_wmy_ [protected]
```

Stores the memory kernel of the quadratic form $w_q m_{y,q}$ with itself.

6.7.4.92 K_wmy_jmx_

```
std::vector<std::complex<double> > sampler::K_wmy_jmx_ [protected]
```

Stores the memory kernel of the quadratic form $w_q \star mxy$ with the quadratic form $j_{L,q} m_{x,q}$.

6.7.4.93 kappa_TransCoeff_

```
std::vector<double> sampler::kappa_TransCoeff_ [protected]
```

Stores kappa transport coefficient, see Bissinger notes (sec. 1.2.7), (see old notes, not included in thesis)

6.7.4.94 kappa_TransCoeff_new_

```
std::vector<double> sampler::kappa_TransCoeff_new_ [protected]
```

Stores different kappa transport coefficient, see Bissinger notes (sec. 1.2.7), (see old notes, not included in thesis)

6.7.4.95 lq_

```
std::vector<std::complex<double> > sampler::lq_ [protected]
```

Stores the l_q (spatial angular momentum field fluctuation)

6.7.4.96 lq_cur_

```
std::vector<std::complex<double> > sampler::lq_cur_ [protected]
```

Stores the l_q at the current time.

6.7.4.97 lq_initial_

```
std::vector<std::complex<double> > sampler::lq_initial_ [protected]
```

Stores the l_q at the initial time (avoids repeated computation, only usable if q is not refreshed)

6.7.4.98 M_

```
std::vector<topology::Vector2d> sampler::M_ [protected]
```

Stores magnetizations.

6.7.4.99 M_2_

```
std::vector<double> sampler::M_2_ [protected]
```

Stores magnetizations squared.

6.7.4.100 M_4_

```
std::vector<double> sampler::M_4_ [protected]
```

Stores magnetizations to the fourth power.

6.7.4.101 M_angle_

```
std::vector<double> sampler::M_angle_ [protected]
```

Stores angle of the magnetization (with respect to the spin x-axis)

6.7.4.102 mparq_

```
std::vector<std::complex<double> > sampler::mparq_ [protected]
```

Stores the $m_{\parallel,q}$ (spin field fluctuation parallel to spontaneous M)

6.7.4.103 mparq_cur_

```
std::vector<std::complex<double> > sampler::mparq_cur_ [protected]
```

Stores the $m_{\parallel,q}$ at the current time (avoids repeated computation, which is costly)

6.7.4.104 mparq_initial_

```
std::vector<std::complex<double> > sampler::mparq_initial_ [protected]
```

Stores the $m_{\parallel,q}$ at the initial time (avoids repeated computation, only usable if q is not refreshed)

6.7.4.105 mperpq_

```
std::vector<std::complex<double> > sampler::mperpq_ [protected]
```

Stores the $m_{\perp,q}$ (spin field fluctuation perpendicular to spontaneous M)

6.7.4.106 mperpq_cur_

```
std::vector<std::complex<double> > sampler::mperpq_cur_ [protected]
```

Stores the $m_{\perp,q}$ at the current time (avoids repeated computation, which is costly)

6.7.4.107 mperpq_initial_

```
std::vector<std::complex<double> > sampler::mperpq_initial_ [protected]
```

Stores the $m_{\perp,q}$ at the initial time (avoids repeated computation, only usable if q is not refreshed)

6.7.4.108 MSD_aux_

```
std::vector<double> sampler::MSD_aux_ [protected]
```

Auxiliary MSD storage. Stores all distances and angles (depending on group type) that the particles have travelled.

6.7.4.109 mxq_

```
std::vector<std::complex<double> > sampler::mxq_ [protected]
```

Stores the $m_{x,q}$ (x-spin field fluctuation)

6.7.4.110 mxq_cur_

```
std::vector<std::complex<double> > sampler::mxq_cur_ [protected]
```

Stores the $m_{x,q}$ at the current time (avoids repeated computation, which is costly)

6.7.4.111 mxq_initial_

```
std::vector<std::complex<double> > sampler::mxq_initial_ [protected]
```

Stores the $m_{x,q}$ at the initial time (avoids repeated computation, only usable if q is not refreshed)

6.7.4.112 myq_

```
std::vector<std::complex<double> > sampler::myq_ [protected]
```

Stores the $m_{y,q}$ (y-spin field fluctuation)

6.7.4.113 myq_cur_

```
std::vector<std::complex<double> > sampler::myq_cur_ [protected]
```

Stores the $m_{y,q}$ at the current time (avoids repeated computation, which is costly)

6.7.4.114 myq_initial_

```
std::vector<std::complex<double> > sampler::myq_initial_ [protected]
```

Stores the $m_{y,q}$ at the initial time (avoids repeated computation, only usable if q is not refreshed)

6.7.4.115 nsamp_

```
int sampler::nsamp_ = 0 [protected]
```

Number of samples (for time averages later on)

6.7.4.116 nsnap_

```
int sampler::nsnap_ = 0 [protected]
```

Number of snapshots.

6.7.4.117 NumberOfSwitches_

```
int sampler::NumberOfSwitches_ = 15 [protected]
```

Total number of switches. Could be useful.

6.7.4.118 P_

```
std::vector<topology::Vector2d> sampler::P_  [protected]
```

Stores momentum components.

6.7.4.119 P_2_

```
std::vector<double> sampler::P_2_  [protected]
```

Stores momentum squared.

6.7.4.120 P_4_

```
std::vector<double> sampler::P_4_  [protected]
```

Stores momentum to the fourth power.

6.7.4.121 par_

```
parameters sampler::par_  [protected]
```

Simulation parameters.

6.7.4.122 print_snapshots_

```
bool sampler::print_snapshots_ = 0  [protected]
```

Decides whether trajectories should be sampled.

6.7.4.123 qvals_

```
std::vector<topology::Vector2d> sampler::qvals_  [protected]
```

Values of q for evaluation of field fluctuations (changes over sampling process)

6.7.4.124 refresh_q_

```
bool sampler::refresh_q_ = 0  [protected]
```

Decides whether or not to choose new random q values for each sampling.

6.7.4.125 rq_

```
std::vector<std::complex<double> > sampler::rq_  [protected]
```

Stores the ρ_q (density field fluctuation)

6.7.4.126 rq_cur_

```
std::vector<std::complex<double> > sampler::rq_cur_ [protected]
```

Stores the ρ_q at the current time.

6.7.4.127 rq_initial_

```
std::vector<std::complex<double> > sampler::rq_initial_ [protected]
```

Stores the ρ_q at the initial time (avoids repeated computation, only usable if q is not refreshed)

6.7.4.128 SCF_anglediff_

```
std::vector<double> sampler::SCF_anglediff_ [protected]
```

Stores the static angle difference correlation function.

6.7.4.129 SCF_E_

```
std::vector<double> sampler::SCF_E_ [protected]
```

Stores the static total energy correlation function.

6.7.4.130 SCF_Eint_

```
std::vector<double> sampler::SCF_Eint_ [protected]
```

Stores the static interaction energy correlation function.

6.7.4.131 SCF_Ekin_

```
std::vector<double> sampler::SCF_Ekin_ [protected]
```

Stores the static kinetic energy correlation function.

6.7.4.132 SCF_g_

```
std::vector<double> sampler::SCF_g_ [protected]
```

Stores the pair distribution function.

6.7.4.133 SCF_P_

```
std::vector<double> sampler::SCF_P_ [protected]
```

Stores the static linear momentum correlation function.

6.7.4.134 SCF_Spin_

```
std::vector<double> sampler::SCF_Spin_ [protected]
```

Stores the static spin correlation function.

6.7.4.135 SCF_Spin_par_

```
std::vector<double> sampler::SCF_Spin_par_ [protected]
```

Stores the static spin correlation function.

6.7.4.136 SCF_Spin_perp_

```
std::vector<double> sampler::SCF_Spin_perp_ [protected]
```

Stores the static spin correlation function.

6.7.4.137 SCF_W_

```
std::vector<double> sampler::SCF_W_ [protected]
```

Stores the static spin momentum correlation function.

6.7.4.138 SCFq_mparmperp_

```
std::vector<std::complex<double> > sampler::SCFq_mparmperp_ [protected]
```

Stores the $\langle m^*_{\parallel,q} m_{\perp,q} \rangle$ static correlation.

6.7.4.139 SCFq_re_

```
std::vector<std::complex<double> > sampler::SCFq_re_ [protected]
```

Stores the $\langle r^*_q e_q \rangle$ static correlation.

6.7.4.140 SCFq_we_

```
std::vector<std::complex<double> > sampler::SCFq_we_ [protected]
```

Stores the $\langle w^*_q e_q \rangle$ static correlation.

6.7.4.141 SCFq_xe_

```
std::vector<std::complex<double> > sampler::SCFq_xe_ [protected]
```

Stores the $\langle m^*_{x,q} e_q \rangle$ static correlation.

6.7.4.142 SCFq_xw_

```
std::vector<std::complex<double> > sampler::SCFq_xw_ [protected]
```

Stores the $\langle m^*_{x,q} w_q \rangle$ static correlation.

6.7.4.143 SCFq_xy_

```
std::vector<std::complex<double> > sampler::SCFq_xy_ [protected]
```

Stores the $\langle m^*_{x,q} m_{y,q} \rangle$ static correlation.

6.7.4.144 SCFq_ye_

```
std::vector<std::complex<double> > sampler::SCFq_ye_ [protected]
```

Stores the $\langle m^*_{y,q} e_q \rangle$ static correlation.

6.7.4.145 SCFq_yw_

```
std::vector<std::complex<double> > sampler::SCFq_yw_ [protected]
```

Stores the $\langle m^*_{y,q} w_q \rangle$ static correlation.

6.7.4.146 signed_vortices_

```
std::vector<double> sampler::signed_vortices_ [protected]
```

Stores total signed vortex density (i.e. positive minus negative density).

6.7.4.147 store_eq_

```
bool sampler::store_eq_ = 0 [protected]
```

Decides whether or not e_q is calculated.

6.7.4.148 store_jparq_

```
bool sampler::store_jparq_ = 0 [protected]
```

Decides whether or not $j_{L,q}$ is calculated.

6.7.4.149 store_jperpq_

```
bool sampler::store_jperpq_ = 0 [protected]
```

Decides whether or not $j_{T,q}$ is calculated.

6.7.4.150 store_lq_

```
bool sampler::store_lq_ = 0  [protected]
```

Decides whether or not l_q is calculated.

6.7.4.151 store_MemoryKernels_

```
bool sampler::store_MemoryKernels_ = 0  [protected]
```

Decides whether or not the memory kernels (MCT) are calculated.

6.7.4.152 store_mxq_

```
bool sampler::store_mxq_ = 0  [protected]
```

Decides whether or not $m_{x,q}$ is calculated.

6.7.4.153 store_myq_

```
bool sampler::store_myq_ = 0  [protected]
```

Decides whether or not $m_{y,q}$ is calculated.

6.7.4.154 store_rq_

```
bool sampler::store_rq_ = 0  [protected]
```

Decides whether or not ρ_q is calculated.

6.7.4.155 store_SCF_

```
bool sampler::store_SCF_ = 0  [protected]
```

Decides whether or not the static correlation functions are calculated.

6.7.4.156 store_static_

```
bool sampler::store_static_ = 0  [protected]
```

Decides whether or not static quantities are calculated.

6.7.4.157 store_TCF_

```
bool sampler::store_TCF_ = 0  [protected]
```

Decides whether or not the time correlation functions (autocorrelations and so forth) are calculated.

6.7.4.158 store_teq_

```
bool sampler::store_teq_ = 0  [protected]
```

Decides whether or not θ_q is calculated.

6.7.4.159 store_TransCoeff_

```
bool sampler::store_TransCoeff_ = 0  [protected]
```

Decides whether or not the transport coefficients are calculated.

6.7.4.160 store_vortices_

```
bool sampler::store_vortices_ = 0  [protected]
```

Decides whether or not vortices_ is calculated.

6.7.4.161 store_wq_

```
bool sampler::store_wq_ = 0  [protected]
```

Decides whether or not w_q is calculated.

6.7.4.162 tau_

```
std::vector<topology::Vector2d> sampler::tau_  [protected]
```

Stores momentum flux tau, which is the q to 0 limit of τ'_q in eq. (1.31b). Required for ZM transport Coefficients. (see old notes, not included in thesis)

6.7.4.163 tau_initial_

```
topology::Vector2d sampler::tau_initial_  [protected]
```

Stores initial momentum flux tau (see old notes, not included in thesis)

6.7.4.164 TCF_times_

```
std::vector<double> sampler::TCF_times_  [protected]
```

Stores sampling times (for time correlation function)

6.7.4.165 temperature_

```
std::vector<double> sampler::temperature_ [protected]
```

Stores temperature.

6.7.4.166 temperature_omega_

```
std::vector<double> sampler::temperature_omega_ [protected]
```

Stores spin momentum temperature.

6.7.4.167 temperature_omega_squared_

```
std::vector<double> sampler::temperature_omega_squared_ [protected]
```

Stores spin momentum temperature squared.

6.7.4.168 temperature_p_

```
std::vector<double> sampler::temperature_p_ [protected]
```

Stores linear momentum temperature.

6.7.4.169 temperature_p_squared_

```
std::vector<double> sampler::temperature_p_squared_ [protected]
```

Stores linear momentum temperature squared.

6.7.4.170 temperature_squared_

```
std::vector<double> sampler::temperature_squared_ [protected]
```

Stores temperature squared.

6.7.4.171 teq_

```
std::vector<std::complex<double> > sampler::teq_ [protected]
```

Stores the θ_q (theta field fluctuation)

6.7.4.172 teq_cur_

```
std::vector<std::complex<double> > sampler::teq_cur_ [protected]
```

Stores the θ_q at the current time (avoids repeated computation, which is costly)

6.7.4.173 teq_initial_

```
std::vector<std::complex<double> > sampler::teq_initial_ [protected]
```

Stores the θ_q at the initial time (avoids repeated computation, only usable if q is not refreshed)

6.7.4.174 Theta_

```
std::vector<double> sampler::Theta_ [protected]
```

Stores mean theta (with respect to the spin x-axis)

6.7.4.175 Theta_2_

```
std::vector<double> sampler::Theta_2_ [protected]
```

Stores mean theta^2 (with respect to the spin x-axis)

6.7.4.176 Theta_4_

```
std::vector<double> sampler::Theta_4_ [protected]
```

Stores mean theta^4 (with respect to the spin x-axis)

6.7.4.177 Theta_rel_to_M_

```
std::vector<double> sampler::Theta_rel_to_M_ [protected]
```

Stores mean theta relative to orientation of magnetization.

6.7.4.178 Theta_rel_to_M_2_

```
std::vector<double> sampler::Theta_rel_to_M_2_ [protected]
```

Stores mean theta^2 relative to orientation of magnetization.

6.7.4.179 Theta_rel_to_M_4_

```
std::vector<double> sampler::Theta_rel_to_M_4_ [protected]
```

Stores mean theta^2 relative to orientation of magnetization.

6.7.4.180 TransCoeff_cos1_

```
std::vector<double> sampler::TransCoeff_cos1_ [protected]
```

Transport Coefficient: $\langle cos(te_{ij}) \rangle$ (see old notes, not included in thesis)

6.7.4.181 TransCoeff_J1_

```
std::vector<double> sampler::TransCoeff_J1_ [protected]
```

Transport Coefficient: $\langle J(r_{ij}) \rangle$ (see old notes, not included in thesis)

6.7.4.182 TransCoeff_J1_cos1_

```
std::vector<double> sampler::TransCoeff_J1_cos1_ [protected]
```

Transport Coefficient: $\langle J(r_{ij}) cos(te_{ij}) \rangle$ (see old notes, not included in thesis)

6.7.4.183 TransCoeff_J1_cos1_r2_

```
std::vector<double> sampler::TransCoeff_J1_cos1_r2_ [protected]
```

Transport Coefficient: $\langle J(r_{ij}) cos(te_{ij}) r_{ij}^2 \rangle$ (see old notes, not included in thesis)

6.7.4.184 TransCoeff_J1_cos2_

```
std::vector<double> sampler::TransCoeff_J1_cos2_ [protected]
```

Transport Coefficient: $\langle J(r_{ij}) cos^2(te_{ij}) \rangle$ (see old notes, not included in thesis)

6.7.4.185 TransCoeff_J1_cos2_r2_

```
std::vector<double> sampler::TransCoeff_J1_cos2_r2_ [protected]
```

Transport Coefficient: $\langle J(r_{ij}) cos^2(te_{ij}) r_{ij}^2 \rangle$ (see old notes, not included in thesis)

6.7.4.186 TransCoeff_J1_sin1_te_

```
std::vector<double> sampler::TransCoeff_J1_sin1_te_ [protected]
```

Transport Coefficient: $\langle J(r_{ij}) sin(te_{ij}) te_{ij} \rangle$ (see old notes, not included in thesis)

6.7.4.187 TransCoeff_times_

```
std::vector<double> sampler::TransCoeff_times_ [protected]
```

Stores times used in calculation of transport coefficients.

6.7.4.188 TransCoeff_Up_

```
std::vector<double> sampler::TransCoeff_Up_ [protected]
```

Transport Coefficient: $\langle \mathcal{U}'(r_{ij}) \rangle$ (see old notes, not included in thesis)

6.7.4.189 TransCoeff_Up_cos1_

```
std::vector<double> sampler::TransCoeff_Up_cos1_ [protected]
```

Transport Coefficient: $\langle \mathcal{U}'(r_{ij}) cos(te_{ij}) \rangle$ (see old notes, not included in thesis)

6.7.4.190 TransCoeff_Up_rinv_

```
std::vector<double> sampler::TransCoeff_Up_rinv_ [protected]
```

Transport Coefficient: $\langle \mathcal{U}'(r_{ij}) r_{ij}^{-1} \rangle$ (see old notes, not included in thesis)

6.7.4.191 TransCoeff_Up_rinv_cos1_

```
std::vector<double> sampler::TransCoeff_Up_rinv_cos1_ [protected]
```

Transport Coefficient: $\langle \mathcal{U}'(r_{ij}) r_{ij}^{-1} cos(te_{ij}) \rangle$ (see old notes, not included in thesis)

6.7.4.192 TransCoeff_Up_rinv_te2_

```
std::vector<double> sampler::TransCoeff_Up_rinv_te2_ [protected]
```

Transport Coefficient: $\langle \mathcal{U}'(r_{ij}) r_{ij}^{-1} te_{ij}^2 \rangle$ (see old notes, not included in thesis)

6.7.4.193 TransCoeff_Upp_

```
std::vector<double> sampler::TransCoeff_Upp_ [protected]
```

Transport Coefficient: $\langle \mathcal{U}''(r_{ij}) \rangle$ (see old notes, not included in thesis)

6.7.4.194 TransCoeff_Upp_cos1_

```
std::vector<double> sampler::TransCoeff_Upp_cos1_ [protected]
```

Transport Coefficient: $\langle \mathcal{U}''(r_{ij}) cos(te_{ij}) \rangle$ (see old notes, not included in thesis)

6.7.4.195 TransCoeff_Upp_te2_

```
std::vector<double> sampler::TransCoeff_Upp_te2_ [protected]
```

Transport Coefficient: $\langle \mathcal{U}''(r_{ij}) te_{ij}^2 \rangle$ (see old notes, not included in thesis)

6.7.4.196 Upsilon_

```
std::vector<double> sampler::Upsilon_ [protected]
```

Helicity Upsilon.

6.7.4.197 W_

```
std::vector<double> sampler::W_  [protected]
```

Stores momenta.

6.7.4.198 W_2_

```
std::vector<double> sampler::W_2_  [protected]
```

Stores momenta squared.

6.7.4.199 wq_

```
std::vector<std::complex<double> > sampler::wq_  [protected]
```

Stores the w_q (omega field fluctuation)

6.7.4.200 wq_cur_

```
std::vector<std::complex<double> > sampler::wq_cur_  [protected]
```

Stores the w_q at the current time (avoids repeated computation, which is costly)

6.7.4.201 wq_initial_

```
std::vector<std::complex<double> > sampler::wq_initial_  [protected]
```

Stores the w_q at the initial time (avoids repeated computation, only usable if q is not refreshed)

The documentation for this class was generated from the following files:

- sampler.h
- sampler.cpp

6.8 topology::Vector2d Class Reference

Mathematical 2d vectors. Can be added, multiplied by a scalar, norm computation is possible. There are print-to-file and print-to-command-line functions available.

```
#include <topology.h>
```

Public Member Functions

- Vector2d ()

  Constructor without argument.
- Vector2d (double *v)

  Constructor from double array.
- Vector2d (const double &x)

  Constructor from a double (same arguments).
- Vector2d (const int &x)

  Constructor from an int (same arguments).
- Vector2d (const double x, const double y)

  Constructor from two doubles, x and y argument.
- Vector2d (const Vector2d &w)

  Copy constructor from two doubles.
- void print (std::ostream &outfile) const

  Print to file in xyz format.
- void print () const

  Print to command line in xy format.
- void print_xyz () const

  Print to command line in xyz format.
- double get_x () const

  Return x component.
- double get_y () const

  Return y component.
- void set_x (double x)

  Assign value x to component x.
- void set_y (double y)

  Assign value y to component y.
- const int size () const

  Returns size of the vector (i.e. 2).
- bool is_zero () const

  Returns 1 if vector is zero-vector.
- double & operator[ ] (int index)

  Element access - only recommended in sepcific situation, use get_x(), get_y() whenever possible.
- const double & operator[ ] (int index) const

  const element access - only recommended in sepcific situation, use get_x(), get_y() whenever possible.
- Vector2d & operator+= (const Vector2d &w)

  Vector-Vector addition.
- Vector2d & operator-= (const Vector2d &w)

  Vector-Vector subtraction.
- Vector2d & operator*= (const double a)

  Vector-scalar multiplication (double)
- Vector2d & operator*= (const int a)

  Vector-scalar multiplication (int)
- Vector2d & operator/= (const double a)

  Vector-scalar division (double)
- Vector2d & operator/= (const int a)

  Vector-scalar division (int)
- Vector2d operator- () const

  Unary additive inversion.
- void rotate (const double theta)

Rotates vector by angle theta.

- double norm2 () const

  Returns squared L2 norm of vector.

- double angle ()

  Returns the orientation angle of a vector.

- void normalized ()

  Normalizes vector.

- void periodic_box (const Vector2d &minima, const Vector2d &maxima)

  Periodic boundary conditions in box.

- void periodic_box (const Vector2d &maxima)

  Periodic boundary conditions in box from 0 to maxima.

- Vector2d get_boundary_handler_periodic_box (const Vector2d &maxima, const double cutoff) const

  Boundary handler within [0,maxima]. For more details, see the namesake function with arbitrary boundaries.

- Vector2d get_boundary_handler_periodic_box (const Vector2d &minima, const Vector2d &maxima, const double cutoff) const

  Boundary handler within [minima, maxima].

Protected Attributes

- std::array< double, 2 > v_

  The two components of the 2d vector.

6.8.1 Detailed Description

Mathematical 2d vectors. Can be added, multiplied by a scalar, norm computation is possible. There are print-to-file and print-to-command-line functions available.

The vector can be accessed by the standard ¨VEC[]¨ command. Some additional functions are included to handle periodic boundary conditions in a box for position vectors, angles with respect to the x-axis, inner products, rotations and the like.

Author

Thomas Bissinger

Date

Created: early 2017

Last Updated: 2023-08-01

6.8.2 Constructor & Destructor Documentation

6.8.2.1 Vector2d() [1/6]

```
topology::Vector2d::Vector2d ( )
```

Constructor without argument.

6.8.2.2 Vector2d() [2/6]

```
topology::Vector2d::Vector2d (
            double * v )
```

Constructor from double array.

6.8.2.3 Vector2d() [3/6]

```
topology::Vector2d::Vector2d (
            const double & x )
```

Constructor from a double (same arguments).

6.8.2.4 Vector2d() [4/6]

```
topology::Vector2d::Vector2d (
            const int & x )
```

Constructor from an int (same arguments).

6.8.2.5 Vector2d() [5/6]

```
topology::Vector2d::Vector2d (
            const double x,
            const double y )
```

Constructor from two doubles, x and y argument.

6.8.2.6 Vector2d() [6/6]

```
topology::Vector2d::Vector2d (
            const Vector2d & w )
```

Copy constructor from two doubles.

6.8.3 Member Function Documentation

6.8.3.1 angle()

```
double topology::Vector2d::angle ( )  [inline]
```

Returns the orientation angle of a vector.

6.8.3.2 get_boundary_handler_periodic_box() [1/2]

```
topology::Vector2d topology::Vector2d::get_boundary_handler_periodic_box (
            const Vector2d & maxima,
            const double cutoff ) const
```

Boundary handler within [0,maxima]. For more details, see the namesake function with arbitrary boundaries.

6.8.3.3 get_boundary_handler_periodic_box() [2/2]

```
topology::Vector2d topology::Vector2d::get_boundary_handler_periodic_box (
            const Vector2d & minima,
            const Vector2d & maxima,
            const double cutoff ) const
```

Boundary handler within [minima, maxima].

A boundary handler is a Vector2d that determines whether the vector is close to the boundary of a periodic box. If it is within the cutoff radius of the minimum (or 0) in one component, this component will read 0. If it is within the cutoff radius to the maximum, the component will read 1. If neither is the case, i.e. the vector does is well within the volume, the component reads 0.

6.8.3.4 get_x()

```
double topology::Vector2d::get_x ( ) const  [inline]
```

Return x component.

6.8.3.5 get_y()

```
double topology::Vector2d::get_y ( ) const  [inline]
```

Return y component.

6.8.3.6 is_zero()

```
bool topology::Vector2d::is_zero ( ) const  [inline]
```

Returns 1 if vector is zero-vector.

6.8.3.7 norm2()

```
double topology::Vector2d::norm2 ( ) const  [inline]
```

Returns squared L2 norm of vector.

6.8.3.8 normalized()

```
void topology::Vector2d::normalized ( )
```

Normalizes vector.

6.8.3.9 operator*=() [1/2]

```
topology::Vector2d & topology::Vector2d::operator*= (
            const double a )
```

Vector-scalar multiplication (double)

6.8.3.10 operator*=() [2/2]

```
topology::Vector2d & topology::Vector2d::operator*= (
            const int a )
```

Vector-scalar multiplication (int)

6.8.3.11 operator+=()

```
topology::Vector2d & topology::Vector2d::operator+= (
            const Vector2d & w )
```

Vector-Vector addition.

6.8.3.12 operator-()

```
topology::Vector2d topology::Vector2d::operator- ( ) const
```

Unary additive inversion.

6.8.3.13 operator-=()

```
topology::Vector2d & topology::Vector2d::operator-= (
            const Vector2d & w )
```

Vector-Vector subtraction.

6.8.3.14 operator/=() [1/2]

```
topology::Vector2d & topology::Vector2d::operator/= (
            const double a )
```

Vector-scalar division (double)

6.8.3.15 operator/=() [2/2]

```
topology::Vector2d & topology::Vector2d::operator/= (
            const int a )
```

Vector-scalar division (int)

6.8.3.16 operator[]() [1/2]

```
double & topology::Vector2d::operator[] (
            int index )
```

Element access - only recommended in sepcific situation, use get_x(), get_y() whenever possible.

6.8.3.17 operator[]() [2/2]

```
const double & topology::Vector2d::operator[] (
            int index ) const
```

const element access - only recommended in sepcific situation, use get_x(), get_y() whenever possible.

6.8.3.18 periodic_box() [1/2]

```
void topology::Vector2d::periodic_box (
            const Vector2d & maxima )
```

Periodic boundary conditions in box from 0 to maxima.

6.8.3.19 periodic_box() [2/2]

```
void topology::Vector2d::periodic_box (
            const Vector2d & minima,
            const Vector2d & maxima )
```

Periodic boundary conditions in box.

PERIODIC_BOX: Changes the vector according to periodic boundary conditions in a box. Can either take two Vector2d arguments, indicating the minima and maxima of the box. If only one argument is provided, the minima are set to zero.

6.8.3.20 print() [1/2]

```
void topology::Vector2d::print ( ) const
```

Print to command line in xy format.

6.8.3.21 print() [2/2]

```
void topology::Vector2d::print (
            std::ostream & outfile ) const
```

Print to file in xyz format.

6.8.3.22 print_xyz()

```
void topology::Vector2d::print_xyz ( ) const
```

Print to command line in xyz format.

6.8.3.23 rotate()

```
void topology::Vector2d::rotate (
            const double theta )
```

Rotates vector by angle theta.

6.8.3.24 set_x()

```
void topology::Vector2d::set_x (
            double x )
```

Assign value x to component x.

6.8.3.25 set_y()

```
void topology::Vector2d::set_y (
            double y )
```

Assign value y to component y.

6.8.3.26 size()

```
const int topology::Vector2d::size ( ) const  [inline]
```

Returns size of the vector (i.e. 2).

6.8.4 Member Data Documentation

6.8.4.1 v_

```
std::array<double,2> topology::Vector2d::v_  [protected]
```

The two components of the 2d vector.

The documentation for this class was generated from the following files:

- topology.h
- topology.cpp

Chapter 7

File Documentation

7.1 computations.cpp File Reference

cpp-File to computations.h, implementation of the functions.

```
#include ¨computations.h¨
#include <iostream>
#include <stdlib.h>
```

Functions

- double mod (double val, const double min, const double max)

  Modulus function between min and max.
- template<typename T , typename A >
  T vector_mean (std::vector< T, A > v)

  Calculates mean of vector. Template, instantiated with double, int and std::complex<double> in computations.cpp.
- template double vector_mean (std::vector< double, std::allocator< double > > v)
- template int vector_mean (std::vector< int, std::allocator< int > > v)
- template std::complex< double > vector_mean (std::vector< std::complex< double >, std::allocator< std↩ ::complex< double > > > v)
- template topology::Vector2d vector_mean (std::vector< topology::Vector2d, std::allocator< topology::Vector2d > > v)
- template<typename T , typename A >
  T row_mean (std::vector< T, A > v, int M, int N, int i)

  Calculates mean of row of matrix (vector is M matrix rows of length N together). Template, instantiated with double, int and std::complex<double> in computations.cpp.
- template double row_mean (std::vector< double, std::allocator< double > > v, int M, int N, int i)
- template int row_mean (std::vector< int, std::allocator< int > > v, int M, int N, int i)
- template std::complex< double > row_mean (std::vector< std::complex< double >, std::allocator< std↩ ::complex< double > > > v, int M, int N, int i)
- template topology::Vector2d row_mean (std::vector< topology::Vector2d, std::allocator< topology::Vector2d > > v, int M, int N, int i)
- template<typename T , typename A >
  T column_mean (std::vector< T, A > v, int M, int N, int j)

  Calculates mean of column of matrix (vector is M matrix rows of length N together). Template, instantiated with double, int and std::complex<double> in computations.cpp.
- template double column_mean (std::vector< double, std::allocator< double > > v, int M, int N, int j)

- template int column_mean (std::vector< int, std::allocator< int > > v, int M, int N, int j)
- template std::complex< double > column_mean (std::vector< std::complex< double >, std::allocator< std::complex< double > > > v, int M, int N, int j)
- template topology::Vector2d column_mean (std::vector< topology::Vector2d, std::allocator< topology::Vector2d > > v, int M, int N, int j)
- template<typename T , typename A >
  T selective_vector_mean (std::vector< T, A > yvals, std::vector< double > xvals, double xsep_min)

  Calculates mean of vector, but only for entries that are at least xsep_min apart.
- template double selective_vector_mean (std::vector< double, std::allocator< double > > yvals, std::vector< double > xvals, double xsep_min)
- template int selective_vector_mean (std::vector< int, std::allocator< int > > yvals, std::vector< double > xvals, double xsep_min)
- template std::complex< double > selective_vector_mean (std::vector< std::complex< double >, std::allocator< std::complex< double > > > yvals, std::vector< double > xvals, double xsep_min)
- template topology::Vector2d selective_vector_mean (std::vector< topology::Vector2d, std::allocator< topology::Vector2d > > yvals, std::vector< double > xvals, double xsep_min)
- template<typename T , typename A >
  T selective_row_mean (std::vector< T, A > yvals, std::vector< double > xvals, double xsep_min, int M, int N, int i)

  Calculates row mean of vector, but only for entries that are at least xsep_min apart.
- template double selective_row_mean (std::vector< double, std::allocator< double > > yvals, std::vector< double > xvals, double xsep_min, int M, int N, int i)
- template int selective_row_mean (std::vector< int, std::allocator< int > > yvals, std::vector< double > xvals, double xsep_min, int M, int N, int i)
- template std::complex< double > selective_row_mean (std::vector< std::complex< double >, std::allocator< std::complex< double > > > yvals, std::vector< double > xvals, double xsep_min, int M, int N, int i)
- template topology::Vector2d selective_row_mean (std::vector< topology::Vector2d, std::allocator< topology::Vector2d > > yvals, std::vector< double > xvals, double xsep_min, int M, int N, int i)
- template<typename T , typename A >
  T selective_column_mean (std::vector< T, A > yvals, std::vector< double > xvals, double xsep_min, int M, int N, int j)

  Calculates column mean of vector, but only for entries that are at least xsep_min apart.
- template double selective_column_mean (std::vector< double, std::allocator< double > > yvals, std::vector< double > xvals, double xsep_min, int M, int N, int j)
- template int selective_column_mean (std::vector< int, std::allocator< int > > yvals, std::vector< double > xvals, double xsep_min, int M, int N, int j)
- template std::complex< double > selective_column_mean (std::vector< std::complex< double >, std::allocator< std::complex< double > > > yvals, std::vector< double > xvals, double xsep_min, int M, int N, int j)
- template topology::Vector2d selective_column_mean (std::vector< topology::Vector2d, std::allocator< topology::Vector2d > > yvals, std::vector< double > xvals, double xsep_min, int M, int N, int j)
- double vector_variance (std::vector< double > v, double &mean)

  Calculates variance of vector, also stores mean.
- template<typename T , typename A >
  std::vector< T, A > vector_scale (const std::vector< T, A > &v, double a)

  Scales vector by factor a.
- template std::vector< double, std::allocator< double > > vector_scale (const std::vector< double, std::allocator< double > > &v, double a)
- template std::vector< int, std::allocator< int > > vector_scale (const std::vector< int, std::allocator< int > > &v, double a)
- template std::vector< std::complex< double >, std::allocator< std::complex< double > > > vector_scale (const std::vector< std::complex< double >, std::allocator< std::complex< double > > > &v, double a)
- template std::vector< topology::Vector2d, std::allocator< topology::Vector2d > > vector_scale (const std::vector< topology::Vector2d, std::allocator< topology::Vector2d > > &v, double a)

- template<typename T , typename A >
  std::vector< T, A > vector_sum (const std::vector< T, A > &v, const std::vector< T, A > &w)

  Adds two vectors v and w.
- template std::vector< double, std::allocator< double > > vector_sum (const std::vector< double, std::allocator< double > > &v, const std::vector< double, std::allocator< double > > &w)
- template std::vector< int, std::allocator< int > > vector_sum (const std::vector< int, std::allocator< int > > &v, const std::vector< int, std::allocator< int > > &w)
- template std::vector< std::complex< double >, std::allocator< std::complex< double > > > vector_sum (const std::vector< std::complex< double >, std::allocator< std::complex< double > > > &v, const std::vector< std::complex< double >, std::allocator< std::complex< double > > > &w)
- template std::vector< topology::Vector2d, std::allocator< topology::Vector2d > > vector_sum (const std::vector< topology::Vector2d, std::allocator< topology::Vector2d > > &v, const std::vector< topology::Vector2d, std::allocator< topology::Vector2d > > &w)
- template<typename T , typename A >
  T vector_inpr (const std::vector< T, A > &v, const std::vector< T, A > &w)

  Inner product of two vectors v and w.
- template double vector_inpr (const std::vector< double, std::allocator< double > > &v, const std::vector< double, std::allocator< double > > &w)
- template int vector_inpr (const std::vector< int, std::allocator< int > > &v, const std::vector< int, std::allocator< int > > &w)
- std::complex< double > vector_inpr (const std::vector< std::complex< double > > &v, const std::vector< std::complex< double > > &w)

  Inner product of two complex vectors v and w, where the complex conjugate of the components of v is used.
- template<typename T , typename A >
  std::vector< T, A > vector_pw_mult (const std::vector< T, A > &v, const std::vector< T, A > &w)

  Pointwise multiplication of two vectors v and w.
- template std::vector< double, std::allocator< double > > vector_pw_mult (const std::vector< double, std::allocator< double > > &v, const std::vector< double, std::allocator< double > > &w)
- template std::vector< int, std::allocator< int > > vector_pw_mult (const std::vector< int, std::allocator< int > > &v, const std::vector< int, std::allocator< int > > &w)
- std::vector< std::complex< double > > vector_pw_mult (const std::vector< std::complex< double > > &v, const std::vector< std::complex< double > > &w)

  Pointwise multiplication of two complex vectors v and w, where the complex conjugate of the components of v is used.
- std::vector< std::complex< double > > vector_pw_mult (const std::vector< std::complex< double > > &v_1, const std::vector< std::complex< double > > &v_2, const std::vector< std::complex< double > > &v_3, const std::vector< std::complex< double > > &v_4)

  Pointwise multiplication of four complex vectors v and w, where the complex conjugate of the components of v_1 and v_2 is used.
- std::vector< double > vector_pw_norm (const std::vector< std::complex< double > > &v)

  Pointwise norm value of a complex vector v, i.e. sum v[i]^* v[i].
- std::vector< double > log_bin (double binmin, double binmax, double diffmin, int Nbins)

  Makes logarithmic bin.
- int find_bin (const std::vector< double > &bin, const double &value)
- double random_angle (double min, double max)

  functions for random angle, designed such that only -pi:pi is allowed (automatical limits when values are not allowed)
- double random_angle (double max)

  Random angle between -max and max (or -max and 0 if max is negative.)

7.1.1 Detailed Description

cpp-File to computations.h, implementation of the functions.

For more details, see computations.h.

Author

Thomas Bissinger

Date

Created: mid-2017

Last Update: 23-08-02

7.1.2 Function Documentation

7.1.2.1 column_mean() [1/5]

```
template double column_mean (
            std::vector< double, std::allocator< double > > v,
            int M,
            int N,
            int j )
```

7.1.2.2 column_mean() [2/5]

```
template int column_mean (
            std::vector< int, std::allocator< int > > v,
            int M,
            int N,
            int j )
```

7.1.2.3 column_mean() [3/5]

```
template std::complex< double > column_mean (
            std::vector< std::complex< double >, std::allocator< std::complex< double > >
> v,
            int M,
            int N,
            int j )
```

7.1.2.4 column_mean() [4/5]

```
template<typename T , typename A >
T column_mean (
            std::vector< T, A > v,
            int M,
            int N,
            int j )
```

Calculates mean of column of matrix (vector is M matrix rows of length N together). Template, instantiated with double, int and std::complex<double> in computations.cpp.

7.1.2.5 column_mean() [5/5]

```
template topology::Vector2d column_mean (
            std::vector< topology::Vector2d, std::allocator< topology::Vector2d > > v,
            int M,
            int N,
            int j )
```

7.1.2.6 find_bin()

```
int find_bin (
            const std::vector< double > & bin,
            const double & value )
```

7.1.2.7 log_bin()

```
std::vector< double > log_bin (
            double binmin,
            double binmax,
            double diffmin,
            int Nbins )
```

Makes logarithmic bin.

7.1.2.8 mod()

```
double mod (
            double val,
            const double min,
            const double max )
```

Modulus function between min and max.

7.1.2.9 random_angle() [1/2]

```
double random_angle (
            double max )
```

Random angle between -max and max (or -max and 0 if max is negative.)

7.1.2.10 random_angle() [2/2]

```
double random_angle (
            double min,
            double max )
```

functions for random angle, designed such that only -pi:pi is allowed (automatical limits when values are not allowed)

7.1.2.11 row_mean() [1/5]

```
template double row_mean (
            std::vector< double, std::allocator< double > > v,
            int M,
            int N,
            int i )
```

7.1.2.12 row_mean() [2/5]

```
template int row_mean (
            std::vector< int, std::allocator< int > > v,
            int M,
            int N,
            int i )
```

7.1.2.13 row_mean() [3/5]

```
template std::complex< double > row_mean (
            std::vector< std::complex< double >, std::allocator< std::complex< double > >
> v,
            int M,
            int N,
            int i )
```

7.1.2.14 row_mean() [4/5]

```
template<typename T , typename A >
T row_mean (
            std::vector< T, A > v,
            int M,
            int N,
            int i )
```

Calculates mean of row of matrix (vector is M matrix rows of length N together). Template, instantiated with double, int and std::complex<double> in computations.cpp.

7.1.2.15 row_mean() [5/5]

```
template topology::Vector2d row_mean (
            std::vector< topology::Vector2d, std::allocator< topology::Vector2d > > v,
            int M,
            int N,
            int i )
```

7.1.2.16 selective_column_mean() [1/5]

```
template double selective_column_mean (
            std::vector< double, std::allocator< double > > yvals,
            std::vector< double > xvals,
            double xsep_min,
            int M,
            int N,
            int j )
```

7.1.2.17 selective_column_mean() [2/5]

```
template int selective_column_mean (
            std::vector< int, std::allocator< int > > yvals,
            std::vector< double > xvals,
            double xsep_min,
            int M,
            int N,
            int j )
```

7.1.2.18 selective_column_mean() [3/5]

```
template std::complex< double > selective_column_mean (
            std::vector< std::complex< double >, std::allocator< std::complex< double > >
> yvals,
            std::vector< double > xvals,
            double xsep_min,
            int M,
            int N,
            int j )
```

7.1.2.19 selective_column_mean() [4/5]

```
template<typename T , typename A >
T selective_column_mean (
            std::vector< T, A > yvals,
            std::vector< double > xvals,
            double xsep_min,
            int M,
            int N,
            int j )
```

Calculates column mean of vector, but only for entries that are at least xsep_min apart.

7.1.2.20 selective_column_mean() [5/5]

```
template topology::Vector2d selective_column_mean (
            std::vector< topology::Vector2d, std::allocator< topology::Vector2d > > yvals,
            std::vector< double > xvals,
            double xsep_min,
            int M,
            int N,
            int j )
```

7.1.2.21 selective_row_mean() [1/5]

```
template double selective_row_mean (
            std::vector< double, std::allocator< double > > yvals,
            std::vector< double > xvals,
            double xsep_min,
            int M,
            int N,
            int i )
```

7.1.2.22 selective_row_mean() [2/5]

```
template int selective_row_mean (
            std::vector< int, std::allocator< int > > yvals,
            std::vector< double > xvals,
            double xsep_min,
            int M,
            int N,
            int i )
```

7.1.2.23 selective_row_mean() [3/5]

```
template std::complex< double > selective_row_mean (
            std::vector< std::complex< double >, std::allocator< std::complex< double > >
> yvals,
            std::vector< double > xvals,
            double xsep_min,
            int M,
            int N,
            int i )
```

7.1.2.24 selective_row_mean() [4/5]

```
template<typename T , typename A >
T selective_row_mean (
            std::vector< T, A > yvals,
            std::vector< double > xvals,
            double xsep_min,
            int M,
            int N,
            int i )
```

Calculates row mean of vector, but only for entries that are at least xsep_min apart.

7.1.2.25 selective_row_mean() [5/5]

```
template topology::Vector2d selective_row_mean (
            std::vector< topology::Vector2d, std::allocator< topology::Vector2d > > yvals,
            std::vector< double > xvals,
            double xsep_min,
            int M,
            int N,
            int i )
```

7.1.2.26 selective_vector_mean() [1/5]

```
template double selective_vector_mean (
            std::vector< double, std::allocator< double > > yvals,
            std::vector< double > xvals,
            double xsep_min )
```

7.1.2.27 selective_vector_mean() [2/5]

```
template int selective_vector_mean (
            std::vector< int, std::allocator< int > > yvals,
            std::vector< double > xvals,
            double xsep_min )
```

7.1.2.28 selective_vector_mean() [3/5]

```
template std::complex< double > selective_vector_mean (
            std::vector< std::complex< double >, std::allocator< std::complex< double > >
> yvals,
            std::vector< double > xvals,
            double xsep_min )
```

7.1.2.29 selective_vector_mean() [4/5]

```
template<typename T , typename A >
T selective_vector_mean (
            std::vector< T, A > yvals,
            std::vector< double > xvals,
            double xsep_min )
```

Calculates mean of vector, but only for entries that are at least xsep_min apart.

7.1.2.30 selective_vector_mean() [5/5]

```
template topology::Vector2d selective_vector_mean (
            std::vector< topology::Vector2d, std::allocator< topology::Vector2d > > yvals,
            std::vector< double > xvals,
            double xsep_min )
```

7.1.2.31 vector_inpr() [1/4]

```
template double vector_inpr (
            const std::vector< double, std::allocator< double > > & v,
            const std::vector< double, std::allocator< double > > & w )
```

7.1.2.32 vector_inpr() [2/4]

```
template int vector_inpr (
            const std::vector< int, std::allocator< int > > & v,
            const std::vector< int, std::allocator< int > > & w )
```

7.1.2.33 vector_inpr() [3/4]

```
std::complex< double > vector_inpr (
            const std::vector< std::complex< double > > & v,
            const std::vector< std::complex< double > > & w )
```

Inner product of two complex vectors v and w, where the complex conjugate of the components of v is used.

7.1.2.34 vector_inpr() [4/4]

```
template<typename T , typename A >
T vector_inpr (
            const std::vector< T, A > & v,
            const std::vector< T, A > & w )
```

Inner product of two vectors v and w.

7.1.2.35 vector_mean() [1/5]

```
template double vector_mean (
            std::vector< double, std::allocator< double > > v )
```

7.1.2.36 vector_mean() [2/5]

```
template int vector_mean (
            std::vector< int, std::allocator< int > > v )
```

7.1.2.37 vector_mean() [3/5]

```
template std::complex< double > vector_mean (
            std::vector< std::complex< double >, std::allocator< std::complex< double > >
> v )
```

7.1.2.38 vector_mean() [4/5]

```
template<typename T , typename A >
T vector_mean (
            std::vector< T, A > v )
```

Calculates mean of vector. Template, instantiated with double, int and std::complex<double> in computations.cpp.

7.1.2.39 vector_mean() [5/5]

```
template topology::Vector2d vector_mean (
            std::vector< topology::Vector2d, std::allocator< topology::Vector2d > > v )
```

7.1.2.40 vector_pw_mult() [1/5]

```
template std::vector< double, std::allocator< double > > vector_pw_mult (
            const std::vector< double, std::allocator< double > > & v,
            const std::vector< double, std::allocator< double > > & w )
```

7.1.2.41 vector_pw_mult() [2/5]

```
template std::vector< int, std::allocator< int > > vector_pw_mult (
            const std::vector< int, std::allocator< int > > & v,
            const std::vector< int, std::allocator< int > > & w )
```

7.1.2.42 vector_pw_mult() [3/5]

```
std::vector< std::complex< double > > vector_pw_mult (
            const std::vector< std::complex< double > > & v,
            const std::vector< std::complex< double > > & w )
```

Pointwise multiplication of two complex vectors v and w, where the complex conjugate of the components of v is used.

7.1.2.43 vector_pw_mult() [4/5]

```
std::vector< std::complex< double > > vector_pw_mult (
            const std::vector< std::complex< double > > & v_1,
            const std::vector< std::complex< double > > & v_2,
            const std::vector< std::complex< double > > & v_3,
            const std::vector< std::complex< double > > & v_4 )
```

Pointwise multiplication of four complex vectors v and w, where the complex conjugate of the components of v_1 and v_2 is used.

7.1.2.44 vector_pw_mult() [5/5]

```
template<typename T , typename A >
std::vector< T, A > vector_pw_mult (
            const std::vector< T, A > & v,
            const std::vector< T, A > & w )
```

Pointwise multiplication of two vectors v and w.

7.1.2.45 vector_pw_norm()

```
std::vector< double > vector_pw_norm (
            const std::vector< std::complex< double > > & v )
```

Pointwise norm value of a complex vector v, i.e. sum v[i]^∗ v[i].

7.1.2.46 vector_scale() [1/5]

```
template std::vector< double, std::allocator< double > > vector_scale (
            const std::vector< double, std::allocator< double > > & v,
            double a )
```

7.1.2.47 vector_scale() [2/5]

```
template std::vector< int, std::allocator< int > > vector_scale (
            const std::vector< int, std::allocator< int > > & v,
            double a )
```

7.1.2.48 vector_scale() [3/5]

```
template std::vector< std::complex< double >, std::allocator< std::complex< double > > >
vector_scale (
            const std::vector< std::complex< double >, std::allocator< std::complex< double
> > > & v,
            double a )
```

7.1.2.49 vector_scale() [4/5]

```
template<typename T , typename A >
std::vector< T, A > vector_scale (
            const std::vector< T, A > & v,
            double a )
```

Scales vector by factor a.

7.1.2.50 vector_scale() [5/5]

```
template std::vector< topology::Vector2d, std::allocator< topology::Vector2d > > vector_scale
(
            const std::vector< topology::Vector2d, std::allocator< topology::Vector2d > > &
v,
            double a )
```

7.1.2.51 vector_sum() [1/5]

```
template std::vector< double, std::allocator< double > > vector_sum (
            const std::vector< double, std::allocator< double > > & v,
            const std::vector< double, std::allocator< double > > & w )
```

7.1.2.52 vector_sum() [2/5]

```
template std::vector< int, std::allocator< int > > vector_sum (
            const std::vector< int, std::allocator< int > > & v,
            const std::vector< int, std::allocator< int > > & w )
```

7.1.2.53 vector_sum() [3/5]

```
template std::vector< std::complex< double >, std::allocator< std::complex< double > > >
vector_sum (
            const std::vector< std::complex< double >, std::allocator< std::complex< double
> > > & v,
            const std::vector< std::complex< double >, std::allocator< std::complex< double
> > > & w )
```

7.1.2.54 vector_sum() [4/5]

```
template<typename T , typename A >
std::vector< T, A > vector_sum (
            const std::vector< T, A > & v,
            const std::vector< T, A > & w )
```

Adds two vectors v and w.

7.1.2.55 vector_sum() [5/5]

```
template std::vector< topology::Vector2d, std::allocator< topology::Vector2d > > vector_sum (
            const std::vector< topology::Vector2d, std::allocator< topology::Vector2d > > &
v,
            const std::vector< topology::Vector2d, std::allocator< topology::Vector2d > > &
w )
```

7.1.2.56 vector_variance()

```
double vector_variance (
            std::vector< double > v,
            double & mean )
```

Calculates variance of vector, also stores mean.

Does not make much sense to use templates here, since the variance of an int vector is not much use and the variance of std::complex would require complex conjugation.

7.2 computations.h File Reference

Contains various computation methods that do not belong to a particular class.

```
#include <stdlib.h>
#include <vector>
#include <complex>
#include <math.h>
#include <numeric>
#include <algorithm>
#include ¨topology.h¨
```

Macros

- #define M_PI 3.14159265358979323846

  Manual introduction of M_PI, signifying π up to computer precision.

Functions

- double mod (const double val, const double min, const double max)

  Modulus function between min and max.

- template<typename T , typename A >
  T vector_mean (std::vector< T, A > v)

  Calculates mean of vector. Template, instantiated with double, int and std::complex<double> in computations.cpp.

- template<typename T , typename A >
  T row_mean (std::vector< T, A > v, int M, int N, int i)

  Calculates mean of row of matrix (vector is M matrix rows of length N together). Template, instantiated with double, int and std::complex<double> in computations.cpp.

- template<typename T , typename A >
  T column_mean (std::vector< T, A > v, int M, int N, int j)

  Calculates mean of column of matrix (vector is M matrix rows of length N together). Template, instantiated with double, int and std::complex<double> in computations.cpp.

- template<typename T , typename A >
  T selective_vector_mean (std::vector< T, A > yvals, std::vector< double > xvals, double xsep_min)

  Calculates mean of vector, but only for entries that are at least xsep_min apart.

- template<typename T , typename A >
  T selective_row_mean (std::vector< T, A > yvals, std::vector< double > xvals, double xsep_min, int M, int N, int i)

  Calculates row mean of vector, but only for entries that are at least xsep_min apart.

- template<typename T , typename A >
  T selective_column_mean (std::vector< T, A > yvals, std::vector< double > xvals, double xsep_min, int M, int N, int j)

  Calculates column mean of vector, but only for entries that are at least xsep_min apart.

- double vector_variance (std::vector< double > v, double &mean)

  Calculates variance of vector, also stores mean.

- template<typename T , typename A >
  std::vector< T, A > vector_scale (const std::vector< T, A > &v, double a)

  Scales vector by factor a.

- template<typename T , typename A >
  std::vector< T, A > vector_sum (const std::vector< T, A > &v, const std::vector< T, A > &w)

  Adds two vectors v and w.

- template<typename T , typename A >
  T vector_inpr (const std::vector< T, A > &v, const std::vector< T, A > &w)

  Inner product of two vectors v and w.

- std::complex< double > vector_inpr (const std::vector< std::complex< double > > &v, const std::vector< std::complex< double > > &w)

  Inner product of two complex vectors v and w, where the complex conjugate of the components of v is used.

- template<typename T , typename A >
  std::vector< T, A > vector_pw_mult (const std::vector< T, A > &v, const std::vector< T, A > &w)

  Pointwise multiplication of two vectors v and w.

- std::vector< std::complex< double > > vector_pw_mult (const std::vector< std::complex< double > > &v, const std::vector< std::complex< double > > &w)

  Pointwise multiplication of two complex vectors v and w, where the complex conjugate of the components of v is used.

- std::vector< std::complex< double > > vector_pw_mult (const std::vector< std::complex< double > > &v_1, const std::vector< std::complex< double > > &v_2, const std::vector< std::complex< double > > &v_3, const std::vector< std::complex< double > > &v_4)

  Pointwise multiplication of four complex vectors v and w, where the complex conjugate of the components of v_1 and v_2 is used.
- std::vector< double > vector_pw_norm (const std::vector< std::complex< double > > &v)

  Pointwise norm value of a complex vector v, i.e. sum v[i]^* v[i].
- int matr_index (int M, int N, int i, int j)

  Gives vector index if a vector of size $M \cdot N$ stands for an $M \times N$ matrix.
- std::vector< double > log_bin (double binmin, double binmax, double diffmin, int Nbins)

  Makes logarithmic bin.
- int random_int (int min, int max)

  Random int between min and max.
- double random_double (double min, double max)

  Random double between min and max. min < max is not required.
- double random_double (double max)

  Random double between 0 and max.
- double random_boltzmann_double (double kbT)

  Return random boltzmann-distributed double (Box-Muller method, commented out is a version using the polar method.
- double random_angle ()

  functions for random angle between -pi and pi.
- double random_angle (double min, double max)

  functions for random angle, designed such that only -pi:pi is allowed (automatical limits when values are not allowed)
- double random_angle (double max)

  Random angle between -max and max (or -max and 0 if max is negative.)

7.2.1 Detailed Description

Contains various computation methods that do not belong to a particular class.

Main functionalities: Provides the modulus function, provides code for averaging over data stored in a std::vector with some selection rules, handles treating std::vectors as representing matrices and performing column and row averages. Also has a few methods for computing random variables.

Author

Thomas Bissinger

Date

Created: mid-2017

Last Update: 23-08-02

7.2.2 Macro Definition Documentation

7.2.2.1 M_PI

```
#define M_PI 3.14159265358979323846
```

Manual introduction of M_PI, signifying π up to computer precision.

Some (mostly windows) compilers have problems with the definition of M_PI in math.h. For them, it is manually introduced here.

7.2.3 Function Documentation

7.2.3.1 column_mean()

```
template<typename T , typename A >
T column_mean (
             std::vector< T, A > v,
             int M,
             int N,
             int j )
```

Calculates mean of column of matrix (vector is M matrix rows of length N together). Template, instantiated with double, int and std::complex<double> in computations.cpp.

7.2.3.2 log_bin()

```
std::vector< double > log_bin (
             double binmin,
             double binmax,
             double diffmin,
             int Nbins )
```

Makes logarithmic bin.

7.2.3.3 matr_index()

```
int matr_index (
             int M,
             int N,
             int i,
             int j )  [inline]
```

Gives vector index if a vector of size $M \cdot N$ stands for an $M \times N$ matrix.

In the current implementation, a matrix $\mathbf{A} \in \mathcal{R}^{M \times N}$ is related to a vector $\mathbf{a} \in \mathcal{R}^{M \cdot N}$ by identifying

$$a_k = A_{ij}$$

with the prescription $k(i,j) = i + N \cdot k$.

Therefore, the value of M is not important. For completeness, it is taken as input (if anyone wishes to change the mapping between k and i, j.

Careful, numbering starts at 0, i.e. last index is actually $i = M - 1, j = N - 1$.

7.2.3.4 mod()

```
double mod (
             const double val,
             const double min,
             const double max )
```

Modulus function between min and max.

7.2.3.5 random_angle() [1/3]

```
double random_angle ( )  [inline]
```

functions for random angle between -pi and pi.

7.2.3.6 random_angle() [2/3]

```
double random_angle (
            double max )
```

Random angle between -max and max (or -max and 0 if max is negative.)

7.2.3.7 random_angle() [3/3]

```
double random_angle (
            double min,
            double max )
```

functions for random angle, designed such that only -pi:pi is allowed (automatical limits when values are not allowed)

7.2.3.8 random_boltzmann_double()

```
double random_boltzmann_double (
            double kbT )  [inline]
```

Return random boltzmann-distributed double (Box-Muller method, commented out is a version using the polar method.

7.2.3.9 random_double() [1/2]

```
double random_double (
            double max )  [inline]
```

Random double between 0 and max.

7.2.3.10 random_double() [2/2]

```
double random_double (
            double min,
            double max )  [inline]
```

Random double between min and max. min $<$ max is not required.

7.2.3.11 random_int()

```
int random_int (
            int min,
            int max )  [inline]
```

Random int between min and max.

7.2.3.12 row_mean()

```
template<typename T , typename A >
T row_mean (
            std::vector< T, A > v,
            int M,
            int N,
            int i )
```

Calculates mean of row of matrix (vector is M matrix rows of length N together). Template, instantiated with double, int and std::complex<double> in computations.cpp.

7.2.3.13 selective_column_mean()

```
template<typename T , typename A >
T selective_column_mean (
            std::vector< T, A > yvals,
            std::vector< double > xvals,
            double xsep_min,
            int M,
            int N,
            int j )
```

Calculates column mean of vector, but only for entries that are at least xsep_min apart.

7.2.3.14 selective_row_mean()

```
template<typename T , typename A >
T selective_row_mean (
            std::vector< T, A > yvals,
            std::vector< double > xvals,
            double xsep_min,
            int M,
            int N,
            int i )
```

Calculates row mean of vector, but only for entries that are at least xsep_min apart.

7.2.3.15 selective_vector_mean()

```
template<typename T , typename A >
T selective_vector_mean (
            std::vector< T, A > yvals,
            std::vector< double > xvals,
            double xsep_min )
```

Calculates mean of vector, but only for entries that are at least xsep_min apart.

7.2.3.16 vector_inpr() [1/2]

```
std::complex< double > vector_inpr (
            const std::vector< std::complex< double > > & v,
            const std::vector< std::complex< double > > & w )
```

Inner product of two complex vectors v and w, where the complex conjugate of the components of v is used.

7.2.3.17 vector_inpr() [2/2]

```
template<typename T , typename A >
T vector_inpr (
            const std::vector< T, A > & v,
            const std::vector< T, A > & w )
```

Inner product of two vectors v and w.

7.2.3.18 vector_mean()

```
template<typename T , typename A >
T vector_mean (
            std::vector< T, A > v )
```

Calculates mean of vector. Template, instantiated with double, int and std::complex<double> in computations.cpp.

7.2.3.19 vector_pw_mult() [1/3]

```
std::vector< std::complex< double > > vector_pw_mult (
            const std::vector< std::complex< double > > & v,
            const std::vector< std::complex< double > > & w )
```

Pointwise multiplication of two complex vectors v and w, where the complex conjugate of the components of v is used.

7.2.3.20 vector_pw_mult() [2/3]

```
std::vector< std::complex< double > > vector_pw_mult (
            const std::vector< std::complex< double > > & v_1,
            const std::vector< std::complex< double > > & v_2,
            const std::vector< std::complex< double > > & v_3,
            const std::vector< std::complex< double > > & v_4 )
```

Pointwise multiplication of four complex vectors v and w, where the complex conjugate of the components of v_1 and v_2 is used.

7.2.3.21 vector_pw_mult() [3/3]

```
template<typename T , typename A >
std::vector< T, A > vector_pw_mult (
            const std::vector< T, A > & v,
            const std::vector< T, A > & w )
```

Pointwise multiplication of two vectors v and w.

7.2.3.22 vector_pw_norm()

```
std::vector< double > vector_pw_norm (
            const std::vector< std::complex< double > > & v )
```

Pointwise norm value of a complex vector v, i.e. sum v[i]^∗ v[i].

7.2.3.23 vector_scale()

```
template<typename T , typename A >
std::vector< T, A > vector_scale (
            const std::vector< T, A > & v,
            double a )
```

Scales vector by factor a.

7.2.3.24 vector_sum()

```
template<typename T , typename A >
std::vector< T, A > vector_sum (
            const std::vector< T, A > & v,
            const std::vector< T, A > & w )
```

Adds two vectors v and w.

7.2.3.25 vector_variance()

```
double vector_variance (
            std::vector< double > v,
            double & mean )
```

Calculates variance of vector, also stores mean.

Does not make much sense to use templates here, since the variance of an int vector is not much use and the variance of std::complex would require complex conjugation.

7.3 computations.h

Go to the documentation of this file.

```
00001
00018 #ifndef COMPUTATIONS_H
00019 #define COMPUTATIONS_H
00020 #include<stdlib.h>
00021 #include <vector>
00022 #include <complex>
00023 #include<math.h>
00024 #include <numeric>
00025 #include <algorithm>
00026 #include "topology.h"
00027
00028
00034 #ifndef M_PI
00035     #define M_PI 3.14159265358979323846
00036 #endif
00037
00038
```

```
00039
00040
00041 //
      ================================================================================================================
00042 // general functions
00043 //
      ================================================================================================================
00045 double mod(const double val, const double min, const double max);
00046
00047 //
      ================================================================================================================
00048 // functions for std::vector
00049 //
      ================================================================================================================
00051 template<typename T, typename A>
00052 T vector_mean(std::vector<T,A> v);
00053
00055 template<typename T, typename A>
00056 T row_mean(std::vector<T,A> v, int M, int N, int i);
00057
00059 template<typename T, typename A>
00060 T column_mean(std::vector<T,A> v, int M, int N, int j);
00061
00063 template<typename T, typename A>
00064 T selective_vector_mean(std::vector<T,A> yvals, std::vector<double> xvals, double xsep_min);
00065
00067 template<typename T, typename A>
00068 T selective_row_mean(std::vector<T,A> yvals, std::vector<double> xvals, double xsep_min, int M, int N,
      int i);
00069
00070 template<typename T, typename A>
00071 T selective_column_mean(std::vector<T,A> yvals, std::vector<double> xvals, double xsep_min, int M, int
      N, int j);
00072
00074
00078 double vector_variance(std::vector<double> v, double& mean);
00079
00081 template<typename T, typename A>
00082 std::vector<T,A> vector_scale(const std::vector<T,A>& v, double a);
00083
00085 template<typename T, typename A>
00086 std::vector<T,A> vector_sum(const std::vector<T,A>& v, const std::vector<T,A>& w);
00087
00089 template<typename T, typename A>
00090 T vector_inpr(const std::vector<T,A>& v, const std::vector<T,A>& w);
00091
00093 std::complex<double> vector_inpr(const std::vector<std::complex<double> >& v, const
      std::vector<std::complex<double> >& w);
00094
00096 template<typename T, typename A>
00097 std::vector<T,A> vector_pw_mult(const std::vector<T,A>& v, const std::vector<T,A>& w);
00098
00100 std::vector<std::complex<double> > vector_pw_mult(const std::vector<std::complex<double> >& v, const
      std::vector<std::complex<double> >& w);
00101
00103 std::vector<std::complex<double> > vector_pw_mult(const std::vector<std::complex<double> >& v_1, const
      std::vector<std::complex<double> >& v_2,
00104         const std::vector<std::complex<double> >& v_3, const std::vector<std::complex<double> >& v_4);
00105
00107 std::vector<double > vector_pw_norm(const std::vector<std::complex<double> >& v);
00108
00123 inline int matr_index(int M, int N, int i, int j) { return i * N + j; };
00124
00125 //
      ================================================================================================================
00126 // functions for binning
00127 //
      ================================================================================================================
00138 std::vector<double> log_bin(double binmin, double binmax, double diffmin, int Nbins);
00139
00140
00141
00142
00143 //std::vector<double> lin_log_bin(double qmin, double qmax, double linsep, int Nlin, int Nlog); ///<
      Makes linear-logarithmic bin.
00144 // /*!<
00145 // * \fn std::vector<double> lin_log_bin()
00146 // * \param qmin        Minimum q (starting value) [double]
00147 // * \param qmax        Maximum q (will not be exceeded, may not be reached) [double]
00148 // * \param linsep  Separation for the linear part of the bin [double]
00149 // * \param Nlin        number of linear points [int]
00150 // * \param Nlog        number of logarithmic points. Due to rounding errors, the logarithmic length
      can be one shorter than Nlog. [int]
00151 // * \return           Vector of bins (can be interpreted as left boundaries of bins or mid points).
      [std::vector<double>]
00152 // */
00153
```

```
00156 int find_bin(const std::vector<double>& bin, const double& value);
00165 //
      ================================================================================================
00166 //
      ================================================================================================
00167 // Random functions
00168 //
      ================================================================================================
00169 //
      ================================================================================================
00171 inline int random_int(int min, int max) { return rand() % (max - min) + min; };
00172
00173
00175 inline double random_double(double min, double max) { return min + (double)rand() / RAND_MAX * (max -
      min); };
00177 inline double random_double(double max) { return (double)rand() / RAND_MAX * (max); };
00178
00180 inline double random_boltzmann_double(double kbT){   return
      sqrt(kbT)*sqrt(-2*log(random_double(1)))*cos(2*M_PI*random_double(1)); };
00181
00183 inline double random_angle() { return random_double(-M_PI, M_PI); };
00185 double random_angle(double min, double max);
00187 double random_angle(double max);
00188
00189 #endif
```

7.4 group.cpp File Reference

cpp-File to class declaration of group. Implements routines for the group.

```
#include ¨group.h¨
```

7.4.1 Detailed Description

cpp-File to class declaration of group. Implements routines for the group.

Implements the functions declared in file group.h. Most code should be self-explanatory, see the documentation in group.h for an overview of what each function does.

Author

Thomas Bissinger, additional contributions by Mathias Hoefler

Date

Created: 2020-02-29 (full rewrite)

Last Updated: 2023-07-23

7.5 group.h File Reference

Header-File to class declaration of group. Introduces the group, the central data structure.

```
#include ¨computations.h¨
#include ¨topology.h¨
#include ¨partition.h¨
#include ¨neighbor_list.h¨
#include ¨parameters.h¨
```

```
#include <fstream>
#include <vector>
#include <algorithm>
#include <math.h>
#include <stdio.h>
#include <string>
#include <stdlib.h>
#include <complex>
#include <random>
```

Classes

- class group

  A group of polar particles. Stores vectors with particle positions, velocities, spin orientations and spin rotation velocity, as well as further group properties.

Functions

- group operator+ (const group &G, const group &G2)

  Addition operator. Adds all particle entries of two groups.
- group operator∗ (const group &G, const double a)

  Right multiplication operator, multiplies all group elements by a scalar.
- group operator∗ (const double a, const group &G)

  Left multiplication operator, multiplies all group elements by a scalar.

7.5.1 Detailed Description

Header-File to class declaration of group. Introduces the group, the central data structure.

Date

Created: 2020-02-29 (full rewrite)

Last Updated: 2023-07-23

7.5.2 Function Documentation

7.5.2.1 operator∗() [1/2]

```
group operator* (
            const double a,
            const group & G )  [inline]
```

Left multiplication operator, multiplies all group elements by a scalar.

7.5.2.2 operator∗() [2/2]

```
group operator* (
            const group & G,
            const double a )  [inline]
```

Right multiplication operator, multiplies all group elements by a scalar.

7.5.2.3 operator+()

```
group operator+ (
            const group & G,
            const group & G2 )  [inline]
```

Addition operator. Adds all particle entries of two groups.

7.6 group.h

Go to the documentation of this file.

```
00001
00045 #ifndef GROUP_H_
00046 #define GROUP_H_
00047 #include "computations.h"
00048 #include "topology.h"
00049 #include "partition.h"
00050 #include "neighbor_list.h"
00051 #include "parameters.h"
00052
00053 #include <fstream>
00054 #include <vector>
00055 #include <algorithm>
00056 #include <math.h>
00057 #include <stdio.h>
00058 #include <string>
00059 #include <stdlib.h>
00060 #include <complex>
00061 #include <random>
00062
00063
00064 using namespace std::complex_literals; // makes 1i the complex unit.
00065
00066 // Forward declaration because of interdependencies between classes.
00067 class neighbor_list;
00068
00069 class group {
00070 public:
00071     group() {};
00072     //
     ==========================================================================================
00073     //   Constructors:
00074     //
     ==========================================================================================
00075     group(const parameters& par);
00076
00077     group(const int N, const std::string group_type);
00078     //
     ==========================================================================================
00079
00080     //
     ==========================================================================================
00081     //   Clearing, initialization, partition handling
00082     //
     ==========================================================================================
00084     void clear();
00085
00087     void initialize(const parameters& par);
00089     void initialize_random(double kbT = 0);
00091     void randomize_particles(double kbT = 0);
00093     void mom_to_zero();
00095     void r_to_squarelattice();
00097     void r_to_trigonallattice();
00099     void r_to_lattice();
00101     void initialize_zero();
00103     void fill_partition();
00104     //
     ==========================================================================================
00105
00106     //
     ==========================================================================================
00107     //   Reading and copying from other groups
00108     //
     ==========================================================================================
00110     void read_from_snapshot(std::string snapshotname);
00111
00113
```

```
00115     // TODO Works for MXY model, still problems with lattice models.
00116     void scale_from_subgroup(const group& G);
00117
00118
00119
00124     // TODO Works for MXY model, still problems with lattice models.
00125     void scale_from_subgroup(std::string snapshotname);
00127     //
      ============================================================================================
00128
00129     //
      ============================================================================================
00130     //   Printing functions
00131     //
      ============================================================================================
00133     void print_group(std::ofstream &outputfile) const;
00135     void print_r(std::ofstream &outputfile) const;
00136
00137     //
      ============================================================================================
00138     //   Extracting information
00139     //
      ============================================================================================
00141     inline int get_N() const { return N_; };
00143     inline int size() const { return N_; };
00145     inline int get_sqrtN() const { return sqrtN_; };
00147     inline topology::Vector2d get_L() const { return L_; };
00149     inline double get_boxsize() const { return std::min(L_.get_x(),L_.get_y()); };
00151     inline double get_volume() const { return L_.get_x() * L_.get_y() ; };
00153     inline double get_density() const { return N_ / get_volume(); };
00155     inline double get_I() const { return I_; };
00157     inline double get_J() const { return J_; };
00159     inline double get_m() const { return m_; };
00161     inline double get_cutoff() const { return cutoff_; };
00163     inline double get_vm_v() const { return vm_v_; };
00165     inline double get_vm_eta() const { return vm_eta_; };
00167     inline std::string get_group_type() const { return group_type_; };
00168
00170     inline std::vector<double> get_theta() const { return theta_; };
00172     inline std::vector<double> get_w() const { return w_; };
00174     inline std::vector<topology::Vector2d> get_r() const { return r_; };
00176     inline std::vector<topology::Vector2d> get_p() const { return p_; };
00177
00179     std::vector<double> get_coord() const;
00181     std::vector<double> get_mom() const;
00182
00184     inline double get_theta(int i) const { return theta_[i]; };
00186     inline double get_w(int i) const { return w_[i]; };
00188     inline topology::Vector2d get_r(int i) const { return r_[i]; };
00190     inline topology::Vector2d get_p(int i) const { return p_[i]; };
00191
00192
00194     inline double J_pot(double dist) const {  return J_ * std::pow(1 - dist, 2); };
00196     inline double U_pot(double dist) const {  return 4 * U_ * std::pow(1 - dist, 2); };
00198     inline double J_pot_prime(double dist) const {  return 2 * J_ * (dist - 1); };
00200     inline double U_pot_prime(double dist) const {  return 8 * U_ * (dist - 1);  };
00202     inline double J_pot_primeprime(double dist) const {  return 2 * J_ ; };
00204     inline double U_pot_primeprime(double dist) const {  return 8 * U_ ;  };
00205
00206
00207     //
      ============================================================================================
00208     //   Neighborhood determination
00209     //
      ============================================================================================
00211
00218     std::vector<int> get_neighbors(int i, std::string cellselect, std::vector<double>& distances )
      const;
00220     inline std::vector<int> get_neighbors(int i, std::vector<double>& distances ) const {
00221         return get_neighbors(i, nb_rule_, distances );
00222     };
00223
00225     void generate_neighbor_list();
00226     //
      ============================================================================================
00227
00228     //
      ============================================================================================
00229     //   Calculating differences
00230     //
      ============================================================================================
00232     inline double theta_diff(int i, int j) const { return get_theta(j) - get_theta(i); };
00234     inline double periodic_distance_squared(int i, int j) const {
00235         return topology::periodic_distance_squared(get_r(j), get_r(i), L_);
00236     };
00238     inline double periodic_distance(int i, int j) const { return
      std::sqrt(periodic_distance_squared(i,j)); };
```

```
00240     inline topology::Vector2d periodic_distance_vector(int i, int j) const {
00241         return topology::periodic_distance_vector(get_r(i), get_r(j), L_);
00242     };
00243 
00244     //
    ========================================================================================================
00245     //   Setting and scaling values
00246     //
    ========================================================================================================
00247     void set_theta(double theta, int i) ;
00248     void set_w(double w, int i) ;
00249     void set_r(topology::Vector2d r, int i) ;
00250     void set_rx(double x, int i) ;
00251     void set_ry(double y, int i) ;
00252     void set_p(topology::Vector2d p, int i) ;
00253     void set_px(double px, int i) ;
00254     void set_py(double py, int i) ;
00255     void set_particle(double theta, double w, topology::Vector2d r, topology::Vector2d p,int i) ;
00257     void set_all_w(double w);
00258     void set_all_p(topology::Vector2d p);
00259     void set_all_theta(double theta);
00260     void set_temperature(double kT, int i);
00261     void set_temperature(double kT);
00262     void set_temperature_p(double kT, int i);
00263     void set_temperature_p(double kT);
00264     void set_temperature_w(double kT, int i);
00265     void set_temperature_w(double kT);
00266 
00267     void scale_mom(double a);
00268     //
    ========================================================================================================
00269 
00270     //
    ========================================================================================================
00271     //   Adding to variables
00272     //
    ========================================================================================================
00273     void add_to_theta(const std::vector<double>& theta, double factor = 1);
00274     void add_to_r(const std::vector<topology::Vector2d>& r, double factor = 1);
00275     void add_to_r(const std::vector<double>& r, double factor = 1);
00276     void add_to_coord(const std::vector<double>& coord, double factor = 1);
00277     void add_to_coord_inertialscaling(const std::vector<double>& coord, double factor = 1);
00278     void add_to_w(const std::vector<double>& w, double factor = 1);
00279     void add_to_p(const std::vector<topology::Vector2d>& p, double factor = 1);
00280     void add_to_p(const std::vector<double>& p, double factor = 1);
00281     void add_to_mom(const std::vector<double>& mom, double factor = 1);
00282     void add_random_angle(double angmax);
00283     void add_random_displacement(double rmax);
00284     void stream_along_spin(double v);
00285 
00286     //
    ========================================================================================================
00287     //  Peridic boundary handling
00288     //
    ========================================================================================================
00289     void set_theta_to_interval();
00290     void set_r_to_pbc();
00291     //
    ========================================================================================================
00292 
00293     //
    ========================================================================================================
00294     //  Summing over particles
00295     //
    ========================================================================================================
00297     double sum_w() const;
00299     double sum_w_squared() const ;
00301     double sum_w_4() const ;
00303     double sum_theta() const;
00305     topology::Vector2d sum_s() const;
00307     inline double sum_s_squared() const { return topology::norm2(sum_s()); };
00309     inline double sum_s_4() const { return std::pow(sum_s_squared(),2); };
00311     topology::Vector2d sum_p() const;
00313     double sum_p_squared() const ;
00315     double sum_p_4() const ;
00317     double sum_e_squared() const ;
00319     double sum_ekin_squared() const ;
00321     double sum_eint_squared() const ;
00322     //
    ========================================================================================================
00323 
00324     //
    ========================================================================================================
00325     //  Calculation of physical properties
00326     //
    ========================================================================================================
00328     inline double binder_cumulant() const { return 1 - N_ * sum_s_4() / ( 3.0 *
```

```
      std::pow(sum_s_squared(), 2.0) ) ; } ;
00330     double calc_interaction_energy() const;
00332     double calc_interaction_energy(int i) const;
00334     double calc_kinetic_energy() const;
00336     double calc_kinetic_energy(int i) const;
00338     inline double calc_energy() const { return calc_interaction_energy() + calc_kinetic_energy(); };
00340     inline double calc_energy(int i) const { return calc_kinetic_energy(i) +
      calc_interaction_energy(i); };
00342     double calc_temperature() const;
00344     inline double calc_temperature_w() const {
00345         return (sum_w_squared() - pow(sum_w(), 2.0)/(N_ * I_) )/(N_);
00346     };
00348     inline double calc_temperature_p() const {
00349         return (sum_p_squared() - topology::norm2(sum_p())/(2 * N_ * m_))/(2 * N_);
00350     };
00351
00353     std::vector<int> plaquette(int i) const;
00355     double calc_vorticity(int index) const;
00357     double calc_vortexdensity_unsigned() const;
00359     double calc_vortexdensity_signed() const;
00361     double calc_space_angular_mom() const ;
00363     inline double calc_space_angular_mom(int i) const { return r_[i].get_x() * p_[i].get_y() -
      r_[i].get_y() * p_[i].get_x(); };
00365
00371     double calc_neighbor_mean(double te_pow, double r_pow, double cos_pow, double sin_pow, double
      J_pow, double Up_pow, double Upp_pow) const;
00372
00374     std::vector<double> calc_helicity(double beta) const;
00375
00376     //
      ============================================================================================================
00377     //  Functions implemented by Mathias Hoefler. Redundant and with old group structure
00378     //
      ============================================================================================================
00379     //  double ang_MSD(xygroup &initial) const;
00380     //  double ang_MSD_nonsat(const xygroup &initial) const;
00381     //  double spin_autocorrelation(const xygroup &corr_G) const;
00382
00383
00384     //
      ============================================================================================================
00385     // Calculation of field fluctuations
00386     //
      ============================================================================================================
00388     inline std::complex<double> calc_eiqr(const topology::Vector2d q, int i) const {
00389         return std::complex<double>(std::cos(topology::innerproduct(q,get_r(i))),
00390                 std::sin(topology::innerproduct(q,get_r(i))));
00391     };
00393     std::complex<double> calc_mxq(const topology::Vector2d q, double Mx_0) const;
00395     std::complex<double> calc_myq(const topology::Vector2d q, double My_0) const;
00397     std::complex<double> calc_wq(const topology::Vector2d q, double W_0) const;
00399     std::complex<double> calc_eq(const topology::Vector2d q, double E_0) const;
00401     std::complex<double> calc_teq(const topology::Vector2d q, double Te_0) const;
00403     std::complex<double> calc_rq(const topology::Vector2d q) const;
00405     std::vector<std::complex<double> > calc_jq(const topology::Vector2d q, topology::Vector2d J_0)
      const;
00407     std::complex<double> calc_jqpar(const topology::Vector2d q, topology::Vector2d J_0) const;
00409     std::complex<double> calc_jqperp(const topology::Vector2d q, topology::Vector2d J_0) const;
00411     std::complex<double> calc_lq(const topology::Vector2d q, double L_0) const;
00412
00414
00416     double calc_fieldfluct_average(std::string fluctname, topology::Vector2d q = 0) const;
00418
00420     double calc_one_particle_density(int index, std::string fluctname, topology::Vector2d q = 0)
      const;
00422
00451     std::vector<std::complex<double> > calc_fieldfluct(const std::vector<topology::Vector2d> qvals,
      std::string fluctname) const;
00453     std::vector<std::complex<double> > calc_fieldfluct_convolution(const
      std::vector<topology::Vector2d> qvals, std::string fluctname_1, std::string fluctname_2) const;
00454
00455     //
      ============================================================================================================
00456     // Functions to calculate transport coefficients. These do not work properly
00457     //
      ============================================================================================================
00459     topology::Vector2d calc_tau() const;
00461     topology::Vector2d calc_je() const;
00463     topology::Vector2d calc_current(std::string currentname) const;
00464
00465     //
      ============================================================================================================
00466     //   Static correlation functions
00467     //
      ============================================================================================================
00469
00479     std::vector<double> calc_SCF_S_individual( const int index, const std::vector<double> rbin,
```

```
    std::vector<int>& counts) const;
00480
00482
00492     std::vector<double> calc_SCF_S_oriented_individual( const int index, const std::vector<double>
    rbin, const double& orientation_angle, std::vector<int>& counts) const;
00493
00495
00503     std::vector<double> calc_SCF_g_individual( const int index, const std::vector<double> rbin) const;
00504
00506
00513     std::vector<double> calc_SCF_g( const std::vector<double> rbin, int number_of_points) const;
00514
00516
00526     std::vector<double> calc_SCF_anglediff_individual( const int index, const std::vector<double>
    rbin, std::vector<int>& counts) const;
00527
00529
00539     std::vector<double> calc_SCF_E_individual( const int index, const std::vector<double> rbin,
    std::vector<int>& counts) const;
00540
00542
00545     std::vector<double> calc_SCF_Ekin_individual( const int index, const std::vector<double> rbin,
    std::vector<int>& counts) const;
00546
00548
00558     std::vector<double> calc_SCF_Eint_individual( const int index, const std::vector<double> rbin,
    std::vector<int>& counts) const;
00559
00561
00564     std::vector<double> calc_SCF_P_individual( const int index, const std::vector<double> rbin,
    std::vector<int>& counts) const;
00566
00576     std::vector<double> calc_SCF_W_individual( const int index, const std::vector<double> rbin,
    std::vector<int>& counts) const;
00577
00579
00616     std::vector<double> calc_SCF_averaged( const std::vector<double> rbin, int number_of_points, const
    std::string name) const;
00617
00618
00619     //
    ================================================================================================
00620     //   Time correlation functions
00621     //
    ================================================================================================
00623
00628     double calc_ACF_S(const group& G_initial) const;
00629
00631
00637     double calc_ACF_anglediff(const group& G_initial) const;
00638
00640
00684     double calc_ACF_sp(const group& G_initial, const std::string name) const;
00685
00687
00731     double calc_ACF_q0(const group& G_initial, const std::string name) const;
00732
00733
00735
00757     std::vector<std::complex<double> > calc_TCF(const group& G_initial,
    std::vector<topology::Vector2d> qvals,
00758             std::string fluctname_initial, std::string fluctname_current) const;
00759
00760
00761     //
    ================================================================================================
00762     // Calculation of time derivatives and coordinate differences
00763     //
    ================================================================================================
00765     std::vector<double> time_derivative_theta() const;
00767     std::vector<double> time_derivative_w() const;
00769     std::vector<double> time_derivative_r() const;
00771     std::vector<double> time_derivative_p() const;
00772
00774     std::vector<double> time_derivative_coord() const;
00776     std::vector<double> time_derivative_mom() const;
00778     group time_derivative() const;
00779
00781     std::vector<double> coord_diff(const group& G) const;
00783     void accumulative_MSD(std::vector<double>& MSD, const group& last_G) const;
00784
00785
00786     //
    ================================================================================================
00787     //   Addition operator of groups and multiplication operator of a group by a double value
00788     //
    ================================================================================================
```

```
00790     group& operator+=(const group& G);
00792     group& operator*=(const double a);
00793
00794 protected:
00795
00797     std::string group_type_;
00798
00800     int N_;
00802     int sqrtN_;
00803
00805     topology::Vector2d L_;
00806
00808     double I_ = 1;
00810     double m_ = 1;
00811
00813     double J_ = 1;
00815     double U_ = 1;
00816
00818     double cutoff_ = 1;
00819
00820     std::vector<topology::Vector2d> r_;
00821     std::vector<topology::Vector2d> p_;
00822     std::vector<double> theta_;
00823     std::vector<double> w_;
00824
00826     partition partition_;
00827
00828
00829     std::string nb_rule_;
00830     double nb_mult_factor_;
00831
00832     neighbor_list* nb_list_;
00833
00835     char lattice_type_;
00836
00838     double vm_eta_;
00839
00841     double vm_v_;
00842
00843
00844
00845 };
00846
00848 inline group operator+(const group& G, const group& G2){
00849     return  group(G) += G2;
00850 };
00851
00852
00854 inline group operator*(const group& G, const double a){
00855     return group(G) *= a;
00856 };
00857
00859 inline group operator*(const double a, const group& G){
00860     return group(G) *= a;
00861 };
00862
00863 #endif /* GROUP_H_ */
```

7.7 inputoutput.cpp File Reference

Implements the routines declared in inputoutput.h.

```
#include ¨inputoutput.h¨
```

Functions

- void print_stdvector (std::vector< double > vec, std::ostream &outfile)

  Prints a std::vector (doubles) to an outflow-stream.
- void print_stdvector (std::vector< double > vec, std::ostream &outfile, std::string startstring, std::string endstring)

  Prints a std::vector (doubles) to an outflow-stream. Includes prefix and suffix.

- void print_stdvector (std::vector< double > vec, std::ostream &outfile, std::string startstring, std::string endstring, int index_start, int index_end, int stepskip)

  Prints a std::vector (doubles) to an outflow-stream. Includes prefix and suffix. The entries to be printed can be chosen manually.
- void print_stdvector (std::vector< int > vec, std::ostream &outfile, std::string startstring, std::string endstring)

  Prints a std::vector (integers) to an outflow-stream. Includes prefix and suffix.
- void print_stdvector (std::vector< std::complex< double > > vec, std::ostream &outfile, std::string startstring, std::string endstring, int index_start, int index_end, int stepskip)

  Prints a std::vector (complex numbers) to an outflow-stream. Includes prefix and suffix. The entries to be printed can be chosen manually.

7.7.1 Detailed Description

Implements the routines declared in inputoutput.h.

Detailed description can be found in inputoutput.h.

Author

Thomas Bissinger

Date

Created: mid-2017

Last Update: 23-08-02

7.7.2 Function Documentation

7.7.2.1 print_stdvector() [1/5]

```
void print_stdvector (
            std::vector< double > vec,
            std::ostream & outfile )
```

Prints a std::vector (doubles) to an outflow-stream.

The result is comma-separated.

7.7.2.2 print_stdvector() [2/5]

```
void print_stdvector (
            std::vector< double > vec,
            std::ostream & outfile,
            std::string startstring,
            std::string endstring )
```

Prints a std::vector (doubles) to an outflow-stream. Includes prefix and suffix.

The result is comma-separated. The startstring and endstring can be used to specify a format of the vector.

7.7.2.3 print_stdvector() [3/5]

```
void print_stdvector (
            std::vector< double > vec,
            std::ostream & outfile,
            std::string startstring,
            std::string endstring,
            int index_start,
            int index_end,
            int stepskip )
```

Prints a std::vector (doubles) to an outflow-stream. Includes prefix and suffix. The entries to be printed can be chosen manually.

Does not put a line-break at the end (if desired, include in endstring).

Does not check for errors – e.g. a negative index_start will not be caught.

The result is comma-separated. The startstring and endstring can be used to specify a format of the vector. Output for the vector $v = (v_1, \ldots, v_N)$ is of the form:

¨startstring¨ + ¨\iline 54 _form#70@_fakenl¨ + ¨endstring¨

if $\texttt{index}_e nd \neq \texttt{index}_e start + n \cdot \texttt{stepskip for some } n\texttt{, the next-lowest value is chosen with an } n$

7.7.2.4 print_stdvector() [4/5]

```
void print_stdvector (
            std::vector< int > vec,
            std::ostream & outfile,
            std::string startstring,
            std::string endstring )
```

Prints a std::vector (integers) to an outflow-stream. Includes prefix and suffix.

The result is comma-separated. The startstring and endstring can be used to specify a format of the vector.

7.7.2.5 print_stdvector() [5/5]

```
void print_stdvector (
            std::vector< std::complex< double > > vec,
            std::ostream & outfile,
            std::string startstring,
            std::string endstring,
            int index_start,
            int index_end,
            int stepskip )
```

Prints a std::vector (complex numbers) to an outflow-stream. Includes prefix and suffix. The entries to be printed can be chosen manually.

Does not put a line-break at the end (if desired, include in endstring).

Does not check for errors – e.g. a negative index_start will not be caught.

The result is comma-separated. The startstring and endstring can be used to specify a format of the vector. Output for the vector $v = (v_1, \ldots, v_N)$ is of the form:

¨startstring¨ + ¨\iline 88 _form#70@_fakenl¨ + ¨endstring¨

if index$_e$nd $\neq$ index$_e$start $+ n \cdot$ stepskip for some n, the next-lowest value is chosen with an n s

7.8 inputoutput.h File Reference

Provides routines for printing std::vectors.

```
#include ¨topology.h¨
#include ¨math.h¨
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>
#include <complex>
```

Functions

- void print_stdvector (std::vector< double > vec, std::ostream &outfile)

  Prints a std::vector (doubles) to an outflow-stream.
- void print_stdvector (std::vector< double > vec, std::ostream &outfile, std::string startstring, std::string endstring)

  Prints a std::vector (doubles) to an outflow-stream. Includes prefix and suffix.
- void print_stdvector (std::vector< double > vec, std::ostream &outfile, std::string startstring, std::string endstring, int index_start, int index_end, int stepskip)

  Prints a std::vector (doubles) to an outflow-stream. Includes prefix and suffix. The entries to be printed can be chosen manually.
- void print_stdvector (std::vector< std::complex< double > > vec, std::ostream &outfile, std::string startstring, std::string endstring, int index_start, int index_end, int stepskip)

  Prints a std::vector (complex numbers) to an outflow-stream. Includes prefix and suffix. The entries to be printed can be chosen manually.
- void print_stdvector (std::vector< int > vec, std::ostream &outfile, std::string startstring, std::string endstring)

  Prints a std::vector (integers) to an outflow-stream. Includes prefix and suffix.

7.8.1 Detailed Description

Provides routines for printing std::vectors.

Used to be a bigger suite of programs that would also handle data input. The task of reading input has been shifted to the parameter class that handles the entirety of input. Now it's a small set of routines suited for adjustable printing conditions.

Author

Thomas Bissinger

Date

Created: mid-2017

Last Update: 23-08-02

7.8.2 Function Documentation

7.8.2.1 print_stdvector() [1/5]

```
void print_stdvector (
            std::vector< double > vec,
            std::ostream & outfile )
```

Prints a std::vector (doubles) to an outflow-stream.

The result is comma-separated.

7.8.2.2 print_stdvector() [2/5]

```
void print_stdvector (
            std::vector< double > vec,
            std::ostream & outfile,
            std::string startstring,
            std::string endstring )
```

Prints a std::vector (doubles) to an outflow-stream. Includes prefix and suffix.

The result is comma-separated. The startstring and endstring can be used to specify a format of the vector.

7.8.2.3 print_stdvector() [3/5]

```
void print_stdvector (
             std::vector< double > vec,
             std::ostream & outfile,
             std::string startstring,
             std::string endstring,
             int index_start,
             int index_end,
             int stepskip )
```

Prints a std::vector (doubles) to an outflow-stream. Includes prefix and suffix. The entries to be printed can be chosen manually.

Does not put a line-break at the end (if desired, include in endstring).

Does not check for errors – e.g. a negative index_start will not be caught.

The result is comma-separated. The startstring and endstring can be used to specify a format of the vector. Output for the vector $v = (v_1, \ldots, v_N)$ is of the form:

¨startstring¨ + ¨\iline 54 _form#70@_fakenl¨ + ¨endstring¨

if $\texttt{index}_e nd \neq \texttt{index}_e start + n \cdot \texttt{stepskip for some } n\texttt{, the next-lowest value is chosen with an } n$

7.8.2.4 print_stdvector() [4/5]

```
void print_stdvector (
              std::vector< int > vec,
              std::ostream & outfile,
              std::string startstring,
              std::string endstring )
```

Prints a std::vector (integers) to an outflow-stream. Includes prefix and suffix.

The result is comma-separated. The startstring and endstring can be used to specify a format of the vector.

7.8.2.5 print_stdvector() [5/5]

```
void print_stdvector (
              std::vector< std::complex< double > > vec,
              std::ostream & outfile,
              std::string startstring,
              std::string endstring,
              int index_start,
              int index_end,
              int stepskip )
```

Prints a std::vector (complex numbers) to an outflow-stream. Includes prefix and suffix. The entries to be printed can be chosen manually.

Does not put a line-break at the end (if desired, include in endstring).

Does not check for errors – e.g. a negative index_start will not be caught.

The result is comma-separated. The startstring and endstring can be used to specify a format of the vector. Output for the vector $v = (v_1, \ldots, v_N)$ is of the form:

¨startstring¨ + ¨\iline 88 _form#70@_fakenl¨ + ¨endstring¨

if index$_e nd \neq$ index$_e start + n \cdot$ stepskip for some n, the next-lowest value is chosen with an n s

7.9 inputoutput.h

Go to the documentation of this file.

```
00001
00015 #ifndef INPUTOUTPUT_H
00016 #define INPUTOUTPUT_H
00017
00018 #include"topology.h"
00019
00020 #include "math.h"
00021 #include<stdio.h>
00022 #include<stdlib.h>
00023 #include <iostream>
00024 #include <fstream>
00025 #include <iomanip>
00026 #include <string>
00027 #include <complex>
00028
```

```
00029
00034 void print_stdvector(std::vector<double> vec, std::ostream& outfile);
00035
00041 void print_stdvector(std::vector<double> vec, std::ostream& outfile, std::string startstring,
      std::string endstring);
00042
00074 void print_stdvector(std::vector<double> vec, std::ostream& outfile, std::string startstring,
      std::string endstring,
00075         int index_start, int index_end, int stepskip);
00076
00108 void print_stdvector(std::vector<std::complex<double> > vec, std::ostream& outfile, std::string
      startstring, std::string endstring,
00109         int index_start, int index_end, int stepskip);
00110
00111
00117 void print_stdvector(std::vector<int> vec, std::ostream& outfile, std::string startstring, std::string
      endstring);
00118
00119 #endif
```

7.10 integrator.cpp File Reference

cpp-file to class declaration of integrator. Implements the routines declared in integrator.h

```
#include <vector>
#include ¨integrator.h¨
```

7.10.1 Detailed Description

cpp-file to class declaration of integrator. Implements the routines declared in integrator.h

Mostly comment-free. Some referenes to the equations given in the papers referenced in integrator.h.

Author

Thomas Bissinger, with contributions by Mathias Höfler

Date

Created: 2019-04-12

Last Updated: 2023-08-06

7.11 integrator.h File Reference

Header-file to class declaration of integrator. Introduces the integrator, the data structure associated with discrete time evolution. Incorporates multiple ODE solvers, deterministic as well as stochastic.

```
#include <iostream>
#include <fstream>
#include <vector>
#include ¨topology.h¨
#include ¨computations.h¨
#include ¨parameters.h¨
#include ¨group.h¨
#include <chrono>
```

Classes

- class integrator

  Defines various integration methods for groups. Also includes thermostats. Integrators include: fourth order Runge-Kutta (rk4) and Leapfrog (lf)

7.11.1 Detailed Description

Header-file to class declaration of integrator. Introduces the integrator, the data structure associated with discrete time evolution. Incorporates multiple ODE solvers, deterministic as well as stochastic.

Author

Thomas Bissinger, with contributions by Mathias Höfler

Date

Created: 2019-04-12

Last Updated: 2023-08-06

7.12 integrator.h

Go to the documentation of this file.

```
00001
00023 #ifndef INTEGRATOR_H
00024 #define INTEGRATOR_H
00025
00026
00027 #include <iostream>
00028 #include <fstream>
00029 #include <vector>
00030 #include "topology.h"
00031 #include "computations.h"
00032 #include "parameters.h"
00033 #include "group.h"
00034
00035
00036 #include <chrono>
00037
00038 class integrator {
00039 private:
00061     std::string integrator_type_;
00062
00063     double dt_;
00064     const std::vector<double> a_ {1.0/2.0, 1.0/2.0, 1.0};
00065     const std::vector<double> b_ {1.0/6.0, 1.0/3.0, 1.0/3.0, 1.0/6.0};
00066
00067
00068     double kT_;
00069     double kT_te_;
00070     double kT_r_;
00071     double activity_;
00072
00073     double g_;
00074     double Q_;
00075     double H_0_;
00076     double pi_;
00077     double eta_;
00078     double s_;
00079     double tau_;
00080
00081     double gamma_ld_om_;
00082     double gamma_ld_p_;
00083
00084
00085     double mc_steplength_theta_;
00086     double mc_steplength_r_;
```

```
00087
00088 public:
00089 //
      ================================================================================================
00090 //   Constructors and intialization
00091 //
      ================================================================================================//
      ////////////////////////////
00092
00094 integrator(){};
00096 integrator(double dtin, std::string type);
00098 void integrator_eq(const parameters& par);
00100 void integrator_sample(const parameters& par);
00101
00108 void initialize(const parameters& par, const group& G);
00110 void initialize_parameters(const parameters& par);
00111
00112
00113
00114 //
      ================================================================================================
00115 //   Simple get-functions
00116 //
      ================================================================================================//
      ////////////////////////////
00118 inline double get_dt() const { return dt_; } ;
00120 inline double get_H_0() const { return H_0_; } ;
00122 inline double get_pi() const { return pi_; } ;
00124 inline double get_eta() const { return eta_; } ;
00126 inline double get_s() const { return s_; } ;
00127
00128 //
      ================================================================================================
00129 //   Thermostats
00130 //
      ================================================================================================//
      ////////////////////////////
00132 double berendsen_thermostat(group& G, double T_desired, double berendsen_tau) ;
00133
00134 //
      ================================================================================================
00135 //   Solvers
00136 //
      ================================================================================================//
      ////////////////////////////
00146 group integrate(const group& G, std::vector<double>& momdot) ;
00147
00149 group vm_rule(const group& G) ;
00150
00152 group rk4(const group& G) ;
00153
00155 group leapfrog(const group& G, std::vector<double>& momdot) ;
00157 group leapfrog_active(const group& G, std::vector<double>& momdot, double activity) ;
00158
00159
00160
00165 group np(const group& G, double kT, double& s, double& pi, double Q, double H_0) ;
00166
00171 group nh(const group& G) ;
00172
00177 inline group np(const group& G) ;
00178
00179
00184 group langevin(const group& G, std::vector<double>& momdot) ;
00185
00190 group langevin_active(const group& G, std::vector<double>& momdot, double activityc ) ;
00191
00192 // TODO mc
00193 // template<class GROUP>
00194 // GROUP mc(const GROUP& G) ; ///< Monte-Carlo integration solver (not working yet)
00195
00196
00198 inline double NoseHamiltonian(group& G, double kT, double s , double pi, double Q) const {
00199     return G.calc_kinetic_energy() / (s * s) + G.calc_interaction_energy() + .5 * pi * pi / Q + 2 *
      G.get_N() * kT * std::log(s);
00200 };
00201
00203 void add_activity(group& G, double activity) const ;
00204
00205
00206
00207 };
00208 #endif /* INTEGRATOR_H_ */
```

7.13 main.cpp File Reference

Main-file. Every computation starts here.

```
#include ¨computations.h¨
#include ¨partition.h¨
#include ¨inputoutput.h¨
#include ¨topology.h¨
#include ¨routines.h¨
#include ¨integrator.h¨
#include ¨parameters.h¨
#include ¨sampler.h¨
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <cmath>
#include ¨math.h¨
#include <chrono>
#include <fstream>
#include <list>
#include <vector>
#include <ctime>
#include <complex>
```

Functions

- int main ()

7.13.1 Detailed Description

Main-file. Every computation starts here.

The main-File progresses as follows:

1. Read the file at input/infile.in (called infile for short)
2. Initialize parameter class object par
3. Read the infile into par. In case of input errors, abort the run. Return value is the error code thrown by the read function.
4. Depending on the value of the input parameter mode, choose
   - mode = none. Do nothing.
   - mode = test. User-specified tests can be performed.
   - mode = integrate, mode=integrate_cont, mode=equilibrate. Run specific integration routines from the routines namespace.
   - mode = samp. Run specific sampling routines from the routines namespace.
5. End the run, return value is the return value of the routine performed.

Author

Generated by Doxygen Thomas Bissinger

Date

7.13.2 Function Documentation

7.13.2.1 main()

```
int main ( )
```

7.14 neighbor_list.cpp File Reference

Cpp-file to class declaration of neighbor_list. Implements routines declared in neighbor_list.h.

```
#include ¨neighbor_list.h¨
```

7.14.1 Detailed Description

Cpp-file to class declaration of neighbor_list. Implements routines declared in neighbor_list.h.

For more details, see neighbor_list.h

Author

Thomas Bissinger

Date

Created: mid 2019

Last Updated: 2023-08-06

7.15 neighbor_list.h File Reference

Header-file to class declaration of neighbor_list. Introduces a data structure storing the neighbors to each particle. Useful when neighborhoods stay static.

```
#include ¨topology.h¨
#include ¨partition.h¨
#include ¨group.h¨
#include <fstream>
#include <list>
#include <vector>
#include <algorithm>
```

Classes

- class neighbor_list

  Defines the neighbor_list class. Can be used to extract all information about neighborhood in a group, that is the neighbor pairs and all the distances. Neighbors are those other particles within the cutoff radius or the nearest neighbors in case of a lattice system.

7.15.1 Detailed Description

Header-file to class declaration of neighbor_list. Introduces a data structure storing the neighbors to each particle. Useful when neighborhoods stay static.

Works especially well for disordered XY models and the like. Can also be used for lattice-based models, though the memory access for calls to the neighbor_list may be more expensive than the straightforward modulo calculations associated with neighbor calculations in lattice-based simulations

Author

Thomas Bissinger

Date

Created: mid 2019

Last Updated: 2023-08-06

7.16 neighbor_list.h

Go to the documentation of this file.

```
00001
00034 #ifndef NEIGHBOR_LIST_H
00035 #define NEIGHBOR_LIST_H
00036 #include "topology.h"
00037 #include "partition.h"
00038 #include "group.h"
00039
00040 #include <fstream>
00041 #include <list>
00042 #include <vector>
00043 #include <algorithm>
00044
00045
00046 class group ;
00047
00048 class neighbor_list {
00049     protected:
00050
00051         int N_ = 0;
00052
00053         std::vector<int> nb_indices_;
00054         std::vector<int> nb_first_;
00055         std::vector<double> nb_dist_;
00056     public:
00057         //
     ==========================================================================================
00058         // Constructors, destructors.
00059         //
     ==========================================================================================
00061         neighbor_list() {} ;
00062
00067         neighbor_list(const group& G);
00068
00069
00070         //
     ==========================================================================================
00071         // Clear function
00072         //
     ==========================================================================================
00074         void clear();
00075
00076         //
     ==========================================================================================
00077         // Get functions.
00078         //
     ==========================================================================================
00080         inline bool is_empty() const { return N_ == 0;} ;
00082         std::vector<int> get_neighbors(int i) const ;
00084         std::vector<double> get_dist(int i) const ;
00085
00086
00088 } ;
00089
00090 #endif
```

7.17 parameters.cpp File Reference

cpp-File to class declaration of parameters. Implements the routines defined in parameters.h. For details, check there.

```
#include <vector>
#include ¨parameters.h¨
#include ¨computations.h¨
```

7.17.1 Detailed Description

cpp-File to class declaration of parameters. Implements the routines defined in parameters.h. For details, check there.

Date

Created: 2019-04-12

Last Updated: 2023-08-02

7.18 parameters.h File Reference

Header-File to class declaration of parameters. Introduces parameters, the data structure associated with input data that governs the simulation run.

```
#include ¨computations.h¨
#include <vector>
#include <iostream>
#include <fstream>
#include <limits>
#include <cmath>
```

Classes

- class parameters

  Contains the run parameters of a simulation.

7.18.1 Detailed Description

Header-File to class declaration of parameters. Introduces parameters, the data structure associated with input data that governs the simulation run.

Contains functionalities for reading input files, setting and sharing parameter values.

Date

Created: 2019-04-12

Last Updated: 2023-08-02

7.19 parameters.h

Go to the documentation of this file.

```
00001
00025 #ifndef PARAMETERS_H
00026 #define PARAMETERS_H
00027
00028 #include "computations.h"
00029 #include <vector>
00030 #include <iostream>
00031 #include <fstream>
00032 #include <limits>
00033 #include <cmath>
00034
00035
00036 class parameters {
00037 protected:
00053     std::string system_ = "xy";
00067     std::string mode_ = "none";
00080     std::string init_mode_ = "random";
00081     std::string init_file_ = "input/init_snap.in";
00082     double init_kT_ = -1;
00083     double init_random_displacement_ = 0;
00084     double init_random_angle_ = 0;
00085
00086
00087     std::string job_id_ = "";
00088     std::string outfilename_ = "output/data.out";
00089     int N_ = 256;
00090     int sqrtN_ = 16;
00091     double dof_;
00092     double L_ = 16;
00093     double dt_ = 0.01;
00094     double Tmax_ = 100;
00095     double samplestart_ = 0;
00096     double samplestep_ = 1;
00097     double av_time_spacing_ = 1e2;
00098     int Nsamp_ = 30;
00099     int randomseed_ = 1;
00100     double kT_ = .89;
00101     double I_ = 1;
00102     double m_ = 1;
00103     double J_ = 1;
00104     double U_ = 1;
00105     double cutoff_ = 1;
00115     char lattice_type_ = 's';
00116     double activity_ = 0;
00117     double vm_v_ = 0;
00118     double vm_eta_ = 0;
00119
00138     std::string eq_integrator_type_;
00153     std::string eq_mode_ = "anneal";
00166     std::string eq_breakcond_ = "temperature";
00167     double eq_Tmax_ = 100;
00168     double eq_agreement_threshold_ = 1e-2;
00169     double eq_av_time_ = 0;
00170     double tau_berendsen_;
00171     double eq_anneal_rate_ = .999;
00172     double eq_anneal_step_ = 1e1;
00173     double eq_Tprintstep_ =  std::numeric_limits<double>::quiet_NaN();
00174     double eq_brownian_kT_omega_ = -1;
00175     double eq_brownian_kT_p_ = -1;
00176     double eq_brownian_timestep_ = 1.00;
00177     bool eq_sampswitch_ = 0;
00178     int eq_Nsamp_ = 100;
00179     std::string eq_samp_time_sequence_ = "lin";
00180
00181
00197     std::string sample_integrator_type_;
00198     std::string ensemble_;
00199     double nhnp_pi_ = 0;
00200     double nhnp_Q_ = -1;
00201     double nhnp_tau_ = 0.01;
00202     double nh_eta_ = 0;
00203     double np_s_ = 1;
00204     double mc_steplength_theta_ = 0.1;
00205     double brownian_kT_omega_ = -1;
00206     double brownian_kT_p_ = -1;
00207     double brownian_timestep_ = 1.00;
00208     double gamma_ld_p_ = 0;
00209     double gamma_ld_om_ = 0;
00210     double mc_steplength_r_ = 0.1;
00211
00212     std::string sampling_time_sequence_ = "lin";
00213     int N_rbin_;
```

```
00214     double min_binwidth_r_;
00215     std::string qbin_type_ = "mult";
00216     double qmax_ = 2 * M_PI;
00217     double min_binwidth_q_ = .015;
00218     int N_qbin_;
00219     std::vector<double> rbin_;
00220     std::vector<double> qbin_;
00221     double qfullmax_;
00222     int qsamps_per_bin_;
00223     int n_rsamps_ = 30;
00224     bool print_snapshots_ = false;
00225     bool on_fly_sampling_ = true;
00226     std::string output_folder_ = "output";
00227     std::string snap_overview_file_ = "output/snapshot_overview.out";
00228
00229
00230 public:
00231 // constructors
00233     parameters(){};
00234
00236     // Filling possibilities
00239     int read_from_file(std::ifstream& infile);
00241     int correct_values(std::ofstream& stdoutfile);
00243     void initialize_bins();
00249     void initialize_qbin();
00251     void initialize_rbin(int N_rbin);
00253     void scale_tau(double scale_factor);
00254
00255
00256
00258     // get functions
00260     inline std::string system() const { return system_; };
00261     inline std::string mode() const { return mode_; };
00262     inline std::string job_id() const { return job_id_; };
00263     inline std::string outfilename() const { return outfilename_; };
00264
00265     inline int N() const { return N_; };
00266     inline int sqrtN() const { return sqrtN_; };
00267     inline double dof() const { return dof_; };
00268     inline double L() const { return L_; };
00269     inline double dt() const { return dt_; };
00270     inline double Tmax() const { return Tmax_; };
00271
00272     inline double samplestart() const { return samplestart_; };
00273     inline double samplestep() const { return samplestep_; };
00274     inline double av_time_spacing() const { return av_time_spacing_; };
00275     inline int Nsamp() const { return Nsamp_; };
00276     inline int randomseed() const { return randomseed_; };
00277     inline double kT() const { return kT_; };
00278     inline double I() const { return I_; };
00279     inline double m() const { return m_; };
00280     inline double J() const { return J_; };
00281     inline double U() const { return U_; };
00282     inline double cutoff() const { return cutoff_; };
00283     inline char lattice_type() const { return lattice_type_; };
00284     inline double activity() const { return activity_; };
00285     inline double vm_v() const { return vm_v_; };
00286     inline double vm_eta() const { return vm_eta_; };
00287
00288     inline std::string init_mode() const { return init_mode_; };
00289     inline std::string init_file() const { return init_file_; };
00290     inline double init_kT() const { return init_kT_; };
00291     inline double init_random_displacement() const { return init_random_displacement_; };
00292     inline double init_random_angle() const { return init_random_angle_; };
00293
00294
00295     inline std::string eq_mode() const { return eq_mode_; };
00296     inline std::string eq_integrator_type() const { return eq_integrator_type_; };
00297     inline double eq_Tmax() const { return eq_Tmax_; };
00298     inline std::string eq_breakcond() const { return eq_breakcond_; };
00299     inline double eq_agreement_threshold() const { return eq_agreement_threshold_; };
00300     inline double eq_av_time() const { return eq_av_time_; };
00301     inline double tau_berendsen() const { return tau_berendsen_; };
00302     inline double eq_anneal_rate() const { return eq_anneal_rate_; };
00303     inline double eq_anneal_step() const { return eq_anneal_step_; };
00304     inline double eq_Tprintstep() const { return eq_Tprintstep_; };
00305     inline double eq_brownian_kT_p() const { return eq_brownian_kT_p_; };
00306     inline double eq_brownian_timestep() const { return eq_brownian_timestep_; };
00307     inline double eq_brownian_kT_omega() const { return eq_brownian_kT_omega_; };
00308     inline bool eq_sampswitch() const { return eq_sampswitch_; };
00309     inline int eq_Nsamp() const { return eq_Nsamp_; };
00310     inline std::string eq_samp_time_sequence() const { return eq_samp_time_sequence_; };
00311
00312
00313     inline std::string sample_integrator_type() const { return sample_integrator_type_; };
00314     inline std::string ensemble() const { return ensemble_; };
00315     inline double nhnp_pi() const { return nhnp_pi_; };
```

```
00316     inline double nhnp_Q() const { return nhnp_Q_; };
00317     inline double nhnp_tau() const { return nhnp_tau_; };
00318     inline double nh_eta() const { return nh_eta_; };
00319     inline double np_s() const { return np_s_; };
00320     inline double mc_steplength_theta() const { return mc_steplength_theta_; };
00321     inline double brownian_kT_omega() const { return brownian_kT_omega_; };
00322     inline double brownian_kT_p() const { return brownian_kT_p_; };
00323     inline double brownian_timestep() const { return brownian_timestep_; };
00324     inline double gamma_ld_om() const { return gamma_ld_om_; };
00325     inline double gamma_ld_p() const { return gamma_ld_p_; };
00326     inline double mc_steplength_r() const { return mc_steplength_r_; };
00327
00328
00329     inline std::string sampling_time_sequence() const { return sampling_time_sequence_; };
00330     inline int N_rbin() const { return N_rbin_; };
00331     inline int N_qbin() const { return N_qbin_; };
00332     inline double min_binwidth_r() const { return min_binwidth_r_; };
00333     inline std::string qbin_type() const { return qbin_type_; };
00334     inline double qmax() const { return qmax_; };
00335     inline double min_binwidth_q() const { return min_binwidth_q_; };
00336     inline std::vector<double> rbin() const { return rbin_; };
00337     inline std::vector<double> qbin() const { return qbin_; };
00338     inline int qsamps_per_bin() const { return qsamps_per_bin_; };
00339     inline int n_rsamps() const { return n_rsamps_; };
00340     inline double qfullmax() const { return qfullmax_; };
00341     inline bool print_snapshots() const { return print_snapshots_; };
00342     inline bool on_fly_sampling() const { return on_fly_sampling_; };
00343     inline std::string snap_overview_file() const { return snap_overview_file_; };
00344     inline std::string output_folder() const { return output_folder_; };
00345
00346
00347
00348 };
00349 #endif /* PARAMETERS_H_ */
```

7.20 partition.cpp File Reference

Cpp-file to class declaration of partition. Implements routines defined in partition.h.

```
#include ¨partition.h¨
#include <iomanip>
#include <fstream>
#include <chrono>
```

7.20.1 Detailed Description

Cpp-file to class declaration of partition. Implements routines defined in partition.h.

For details, see partition.h. Most code here is more or less eslf-explanatory. Some use of standard library functions is made for vector handling etc.

Author

Thomas Bissinger

Date

Created: late 2017

Last Updated: 2023-08-06

7.21 partition.h File Reference

Header-file to class declaration of partition. Introduces the partition, a cell list data structure that greatly facilitates neighbor calculation.

```
#include ¨topology.h¨
#include <fstream>
#include <list>
#include <vector>
#include <algorithm>
```

Classes

- class partition

  Defines the partition class. The simulation box is partitioned into cells. The indices of a vector of particles (used for initialization) are sorted according to the cell they belong to. Has functions for printing and computing average velocities in a neighborhood.

7.21.1 Detailed Description

Header-file to class declaration of partition. Introduces the partition, a cell list data structure that greatly facilitates neighbor calculation.

The partition is the preferred method for neighborhood calculation. The class neighbor_list is an alternative, but not recommended when neighborhoods change dynamically in each time step.

Author

Thomas Bissinger

Date

Created: late 2017

Last Updated: 2023-08-06

7.22 partition.h

Go to the documentation of this file.

```
00001
00034 #ifndef PARTITION_H
00035 #define PARTITION_H
00036 #include "topology.h"
00037 #include <fstream>
00038 #include <list>
00039 #include <vector>
00040 #include <algorithm>
00041
00042
00043 class partition {
00044    protected:
00045
00046       int N_;
00047
00054       topology::Vector2d L_;
00055
00056       topology::Vector2d cellwidth_;
```

```
00057
00058
00059       std::vector<int> M_ = std::vector<int>(2);
00060       int cellnum_;
00065       std::vector<int> firsts_;
00066
00067
00069       std::vector<int> cellelem_;
00070
00072       std::vector<int> cellvec_;
00073
00074    public:
00075         //
     ================================================================================================
00076         //   Constructors, destructors.
00077         //
     ================================================================================================
00079         partition() : N_(0), L_(0.0), cellnum_(0) { clear(); } ;
00080
00081
00088         partition(const int& N, const double& cutoff, const topology::Vector2d& L);
00089
00097         partition(const int& N, const double& cutoff,
00098               const topology::Vector2d& L,
00099               const std::vector<topology::Vector2d>& positions);
00100
00101
00102         ~partition();
00103         void clear();
00104
00111         void fill(const std::vector<topology::Vector2d>& positions);
00112
00113
00114         //
     ================================================================================================
00115         //   get-functions
00116         //
     ================================================================================================
00118         inline int get_cellelem(int m) const { return cellelem_[m];};
00120         inline std::vector<int> get_cellelem() const { return cellelem_;};
00122         inline int get_cellnum() const { return cellnum_;};
00124         inline std::vector<int> get_cellvec() const { return cellvec_;};
00125
00126
00127         //
     ================================================================================================
00128         //   print-function
00129         //
     ================================================================================================
00130         void print() const ;
00131
00132         //
     ================================================================================================
00133         //   finding cells corresponding to a coordinate
00134         //
     ================================================================================================
00139         int find_cell(const topology::Vector2d& r) const ;
00141         inline int find_first(int index) const { return firsts_[index];};
00142
00143         //
     ================================================================================================
00144         //   finding adjacent cells
00145         //
     ================================================================================================
00154         int neighbor_cell(int index, int lr, int du) const ;
00155
00167         int neighbor_cell(int index, int lr, int du, topology::Vector2d& shift) const ;
00174         std::vector<int> nb_cells_all(int index, std::vector<topology::Vector2d>& shifts) const;
00176         std::vector<int> nb_cells_all(const topology::Vector2d& r, double cutoffsquared,
     std::vector<topology::Vector2d>& shifts) const;
00178         std::vector<int> nb_cells_ur(int index, std::vector<topology::Vector2d>& shifts) const;
00180         std::vector<int> nb_cells_ur(const topology::Vector2d& r, double cutoffsquared,
     std::vector<topology::Vector2d>& shifts) const;
00181
00182
00184         inline topology::Vector2d corner(int index, int lr, int ud) const { return
     topology::Vector2d(cellwidth_.get_x() * ((index % M_[0])+( 1 + lr )/2), cellwidth_.get_y() *
     (floor((double)(index)/M_[0])+( 1 + ud )/2)); };
00185
00187         std::vector<int> part_in_cell(int index) const ;
00188
00202         std::vector<int> nb_in_cell_index(int index, const topology::Vector2d& r, double
     cutoffsquared,
00203               const std::vector<topology::Vector2d >& positions, std::vector<double>& distances,
00204               const topology::Vector2d shift = 0) const ;
00205
00206         std::vector<int> nb_in_cell_index_above(int index, const topology::Vector2d& r, double
```

```
    cutoffsquared,
00207               const std::vector<topology::Vector2d >& positions, std::vector<double>& distances,
00208               const topology::Vector2d shift = 0) const ;
00209
00210         //
    ============================================================================================================
00211         //   Cluster analysis functions (haven't been used in a while,)
00212         //
    ============================================================================================================
00226         void cluster_analysis(std::vector<int>& cluster_sizes, std::vector<int>& cluster_NoP, const
    int& rho_min) const ;
00228
00238         std::vector<int> cluster_recursion(int current_index, std::vector<int> indices, const int&
    rho_min) const ;
00240
00241
    //=============================================================================================================
00242
00243
00244 } ;
00245
00246
00247 #endif
```

7.23 README.md File Reference

7.24 routines.cpp File Reference

Cpp-File to the namespace routines. Implements the functions from routines.h.

```
#include ¨routines.h¨
```

7.24.1 Detailed Description

Cpp-File to the namespace routines. Implements the functions from routines.h.

Detailed descriptions can be found in routines.h. This file contains some further comments on the precise way the funcitons are implemented. It is recommended to read the code here in order to better understand what the functions do. (

Date

Created: 2019-12-11

Last Updated: 2023-08-06

Author

Thomas Bissinger

7.25 routines.h File Reference

Header-File to the namespace routines. The namespace contains simulation routines that manage setting up the simulation, running it and communicating the results.

```
#include <vector>
#include ¨sampler.h¨
#include ¨inputoutput.h¨
#include ¨integrator.h¨
#include ¨computations.h¨
#include ¨topology.h¨
#include <iostream>
#include <fstream>
#include <chrono>
```

Namespaces

- namespace routines

  Different calculation routines with xygroups and mxygroups. Carries out simulation tasks.

Functions

- int routines::integration (parameters par)

  basic integration routine
- int routines::sampling (parameters par)

  basic sampling routine (no integration performed)
- int routines::equilibrate (group &G, const parameters &par, sampler &samp, const double Tmax, double &t, const std::string breakcond, std::ofstream &stdoutfile)

  equilibration routine
- void routines::integrate_snapshots (group &G, const parameters &par, sampler &samp, std::ofstream &stdoutfile)

  integration routine (the one that does the work)
- void routines::initprint (std::string routine_name, std::ofstream &outfile)

  Initial print of each routine. States the routine name.
- void routines::terminateprint (std::string routine_name, std::ofstream &outfile)

  Terminal print of each routine. States the routine name.

7.25.1 Detailed Description

Header-File to the namespace routines. The namespace contains simulation routines that manage setting up the simulation, running it and communicating the results.

These routines are called by the file main.cpp and are the crucial building blocks of the simulation. (

Date

Created: 2019-12-11

Last Updated: 2023-08-06

7.26 routines.h

Go to the documentation of this file.

```
00001
00020 #include <vector>
00021 #include "sampler.h"
00022 #include "inputoutput.h"
00023 #include "integrator.h"
00024 #include "computations.h"
00025 #include "topology.h"
00026 #include <iostream>
00027 #include <fstream>
00028
00029 #include <chrono>
00030
00031 #ifndef ROUTINES_H_
00032 #define ROUTINES_H_
00033
00034
00035 namespace routines {
00036
00037     //
     ================================================================================================
00038     //   General routines
00039     //
     ================================================================================================
00059     int integration(parameters par);
00060
00081     int sampling(parameters par);
00082
00083
00084     //
     ================================================================================================
00085     //   General routines
00086     //
     ================================================================================================
00127     int equilibrate(group& G, const parameters& par, sampler& samp,
00128                 const double Tmax, double& t,
00129                 const std::string breakcond, std::ofstream& stdoutfile);
00130
00131
00132     //
     ================================================================================================
00133     //   Routines for sampling
00134     //
     ================================================================================================
00135
00165     void integrate_snapshots(group& G, const parameters& par, sampler& samp,
00166                 std::ofstream& stdoutfile);
00167
00168     //
     ================================================================================================
00169     //   Internal routines (input/output etc)
00170     //
     ================================================================================================
00172     void initprint(std::string routine_name, std::ofstream& outfile);
00174     void terminateprint(std::string routine_name, std::ofstream& outfile);
00175 };
00176 #endif /* ROUTINES_H_ */
```

7.27 sampler.cpp File Reference

Cpp-file to class declaration of sampler. Implements the routines declared insampler.h.

```
#include ¨sampler.h¨
```

7.27.1 Detailed Description

Cpp-file to class declaration of sampler. Implements the routines declared insampler.h.

Author

Thomas Bissinger

Date

Created: 2020-01-20

Last Updated: 2023-08-06

7.28 sampler.h File Reference

Header-file to class declaration of sampler. Introduces a data structure calculating and storing different properties of the system during the runtime.

```
#include <iostream>
#include <fstream>
#include <vector>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <complex>
#include ¨topology.h¨
#include ¨parameters.h¨
#include ¨inputoutput.h¨
#include ¨group.h¨
#include <chrono>
```

Classes

- class sampler

  Stores and handles all data sampling performed during a run or in a later diagnostic.

7.28.1 Detailed Description

Header-file to class declaration of sampler. Introduces a data structure calculating and storing different properties of the system during the runtime.

This data structure is central to sampling. See description of class sampler

Author

Thomas Bissinger

Date

Created: 2020-01-20

Last Updated: 2023-08-06

7.29 sampler.h

Go to the documentation of this file.

```
00001
00034 #ifndef SAMPLER_H
00035 #define SAMPLER_H
00036
00037
00038
00039 #include <iostream>
00040 #include <fstream>
00041 #include <vector>
00042 #include <math.h>
00043 #include <stdio.h>
00044 #include <stdlib.h>
00045 #include <complex>
00046
00047 #include "topology.h"
00048 #include "parameters.h"
00049 #include "inputoutput.h"
00050 #include "group.h"
00051
00052 #include <chrono> // Not necessary in code, just for testing.
00053
00054 class sampler {
00055 protected:
00056     parameters par_;
00057
00058     int NumberOfSwitches_ = 15;
00059
00060     int nsamp_ = 0;
00061     int nsnap_ = 0;
00062
00063     std::vector<double> averaging_times_;
00064     std::vector<double> TCF_times_;
00065
00066     std::vector<double> H_;
00067     std::vector<double> H_2_;
00068     std::vector<double> Hkin_2_;
00069     std::vector<double> Hint_2_;
00070     std::vector<double> W_;
00071     std::vector<double> W_2_;
00072     std::vector<topology::Vector2d> M_;
00073     std::vector<double> M_2_;
00074     std::vector<double> M_4_;
00075     std::vector<double> absM_;
00076     std::vector<double> M_angle_;
00077     std::vector<double> Theta_;
00078     std::vector<double> Theta_2_;
00079     std::vector<double> Theta_4_;
00080     std::vector<double> Theta_rel_to_M_;
00081     std::vector<double> Theta_rel_to_M_2_;
00082     std::vector<double> Theta_rel_to_M_4_;
00083     std::vector<double> temperature_;
00084     std::vector<double> temperature_squared_;
00085     std::vector<double> temperature_omega_;
00086     std::vector<double> temperature_omega_squared_;
00087     std::vector<double> temperature_p_;
00088     std::vector<double> temperature_p_squared_;
00089     std::vector<topology::Vector2d> P_;
00090     std::vector<double> P_2_;
00091     std::vector<double> P_4_;
00092
00093     std::vector<double> H_x_;
00094     std::vector<double> H_y_;
00095     std::vector<double> I_x_;
00096     std::vector<double> I_y_;
00097     std::vector<double> I_x_2_;
00098     std::vector<double> I_y_2_;
00099     std::vector<double> Upsilon_;
00100
00101     std::vector<double> abs_vortices_;
00102     std::vector<double> signed_vortices_;
00103
00104     std::vector<double> coordination_number_;
00105
00106     std::vector<topology::Vector2d> qvals_;
00107
00108     std::vector<std::complex<double> > mxq_;
00109     std::vector<std::complex<double> > myq_;
00110     std::vector<std::complex<double> > mparq_;
00111     std::vector<std::complex<double> > mperpq_;
00112     std::vector<std::complex<double> > eq_;
00113     std::vector<std::complex<double> > wq_;
00114     std::vector<std::complex<double> > teq_;
```

```
00115     std::vector<std::complex<double> > rq_;
00116     std::vector<std::complex<double> > jparq_;
00117     std::vector<std::complex<double> > jperpq_;
00118     std::vector<std::complex<double> > lq_;
00119
00120     std::vector<std::complex<double> > mxq_cur_;
00121     std::vector<std::complex<double> > myq_cur_;
00122     std::vector<std::complex<double> > mparq_cur_;
00123     std::vector<std::complex<double> > mperpq_cur_;
00124     std::vector<std::complex<double> > eq_cur_;
00125     std::vector<std::complex<double> > wq_cur_;
00126     std::vector<std::complex<double> > teq_cur_;
00127     std::vector<std::complex<double> > rq_cur_;
00128     std::vector<std::complex<double> > jparq_cur_;
00129     std::vector<std::complex<double> > jperpq_cur_;
00130     std::vector<std::complex<double> > lq_cur_;
00131
00132     std::vector<std::complex<double> > convol_wmx_cur_;
00133     std::vector<std::complex<double> > convol_wmy_cur_;
00134     std::vector<std::complex<double> > convol_jparmx_cur_;
00135     std::vector<std::complex<double> > convol_jparmy_cur_;
00136
00137
00138     std::vector<std::complex<double> > mxq_initial_;
00139     std::vector<std::complex<double> > myq_initial_;
00140     std::vector<std::complex<double> > mparq_initial_;
00141     std::vector<std::complex<double> > mperpq_initial_;
00142     std::vector<std::complex<double> > eq_initial_;
00143     std::vector<std::complex<double> > wq_initial_;
00144     std::vector<std::complex<double> > teq_initial_;
00145     std::vector<std::complex<double> > rq_initial_;
00146     std::vector<std::complex<double> > jparq_initial_;
00147     std::vector<std::complex<double> > jperpq_initial_;
00148     std::vector<std::complex<double> > lq_initial_;
00149
00150     std::vector<std::complex<double> > convol_wmx_initial_;
00151     std::vector<std::complex<double> > convol_wmy_initial_;
00152     std::vector<std::complex<double> > convol_jparmx_initial_;
00153     std::vector<std::complex<double> > convol_jparmy_initial_;
00154
00155
00156     std::vector<double> chimxq_;
00157     std::vector<double> chimyq_;
00158     std::vector<double> chimparq_;
00159     std::vector<double> chimperpq_;
00160     std::vector<double> chieq_;
00161     std::vector<double> chiwq_;
00162     std::vector<double> chiteq_;
00163     std::vector<double> chirq_;
00164     std::vector<double> chijparq_;
00165     std::vector<double> chijperpq_;
00166     std::vector<double> chilq_;
00167
00168     std::vector<std::complex<double> > SCFq_xy_;
00169     std::vector<std::complex<double> > SCFq_xw_;
00170     std::vector<std::complex<double> > SCFq_xe_;
00171     std::vector<std::complex<double> > SCFq_yw_;
00172     std::vector<std::complex<double> > SCFq_ye_;
00173     std::vector<std::complex<double> > SCFq_we_;
00174     std::vector<std::complex<double> > SCFq_mparmperp_;
00175     std::vector<std::complex<double> > SCFq_re_;
00176
00177
00178     std::vector<double> SCF_Spin_;
00179     std::vector<double> SCF_Spin_par_;
00180     std::vector<double> SCF_Spin_perp_;
00181     std::vector<double> SCF_anglediff_;
00182     std::vector<double> SCF_W_;
00183     std::vector<double> SCF_P_;
00184     std::vector<double> SCF_g_;
00185     std::vector<double> SCF_E_;
00186     std::vector<double> SCF_Ekin_;
00187     std::vector<double> SCF_Eint_;
00188
00189     std::vector<double> ACF_Spin_;
00190     std::vector<double> ACF_anglediff_;
00191
00192
00193     std::vector<double> ACF_q0_M_;
00194     std::vector<double> ACF_q0_absM_;
00195
00196     std::vector<double> ACF_Sx_;
00197     std::vector<double> ACF_Sy_;
00198     std::vector<double> ACF_W_;
00199     std::vector<double> ACF_E_;
00200     std::vector<double> ACF_Eint_;
00201     std::vector<double> ACF_Ekin_;
```

```
00202     std::vector<double> ACF_P_;
00203     std::vector<double> ACF_Ppar_;
00204     std::vector<double> ACF_Pperp_;
00205     std::vector<double> ACF_MSD_;
00206     std::vector<double> ACF_ang_MSD_;
00207     std::vector<double> MSD_aux_;
00208
00209
00210     std::vector<std::complex<double> > gxx_;
00211     std::vector<std::complex<double> > gxy_;
00212     std::vector<std::complex<double> > gxw_;
00213     std::vector<std::complex<double> > gxe_;
00214     std::vector<std::complex<double> > gyy_;
00215     std::vector<std::complex<double> > gyw_;
00216     std::vector<std::complex<double> > gye_;
00217     std::vector<std::complex<double> > gww_;
00218     std::vector<std::complex<double> > gwe_;
00219     std::vector<std::complex<double> > gee_;
00220     std::vector<std::complex<double> > gmparmpar_;
00221     std::vector<std::complex<double> > gmperpmperp_;
00222     std::vector<std::complex<double> > gmparmperp_;
00223     std::vector<std::complex<double> > gre_;
00224     std::vector<std::complex<double> > grr_;
00225     std::vector<std::complex<double> > gjparjpar_;
00226     std::vector<std::complex<double> > gjperpjperp_;
00227     std::vector<std::complex<double> > gll_;
00228     std::vector<std::complex<double> > gtt_;
00229
00230     std::vector<double> TransCoeff_J1_;
00231     std::vector<double> TransCoeff_J1_cos1_;
00232     std::vector<double> TransCoeff_J1_cos2_;
00233     std::vector<double> TransCoeff_J1_cos2_r2_;
00234     std::vector<double> TransCoeff_J1_cos1_r2_;
00235     std::vector<double> TransCoeff_J1_sin1_te_;
00236     std::vector<double> TransCoeff_Up_;
00237     std::vector<double> TransCoeff_Up_rinv_;
00238     std::vector<double> TransCoeff_Upp_;
00239     std::vector<double> TransCoeff_Up_cos1_;
00240     std::vector<double> TransCoeff_Up_rinv_cos1_;
00241     std::vector<double> TransCoeff_Upp_cos1_;
00242     std::vector<double> TransCoeff_cos1_;
00243     std::vector<double> TransCoeff_Up_rinv_te2_;
00244     std::vector<double> TransCoeff_Upp_te2_;
00245
00246     std::vector<double> TransCoeff_times_;
00247
00248     std::vector<topology::Vector2d> tau_;
00249     std::vector<topology::Vector2d> je_;
00250     topology::Vector2d tau_initial_;
00251     topology::Vector2d je_initial_;
00252     std::vector<double> kappa_TransCoeff_;
00253     std::vector<double> Gamma_TransCoeff_;
00254     std::vector<double> kappa_TransCoeff_new_;
00255     std::vector<double> Gamma_TransCoeff_new_;
00256
00257     std::vector<std::complex<double> > K_convol_wmx_;
00258     std::vector<std::complex<double> > K_convol_wmy_;
00259     std::vector<std::complex<double> > K_convol_jmx_;
00260     std::vector<std::complex<double> > K_convol_jmy_;
00261     std::vector<std::complex<double> > K_convol_wmx_jmy_;
00262     std::vector<std::complex<double> > K_convol_wmy_jmx_;
00263     std::vector<std::complex<double> > K_wmx_;
00264     std::vector<std::complex<double> > K_wmy_;
00265     std::vector<std::complex<double> > K_jmx_;
00266     std::vector<std::complex<double> > K_jmy_;
00267     std::vector<std::complex<double> > K_wmx_jmy_;
00268     std::vector<std::complex<double> > K_wmy_jmx_;
00269
00270
00271     bool store_static_ = 0;
00272
00273     bool store_vortices_ = 0;
00274
00275     bool store_mxq_ = 0;
00276     bool store_myq_ = 0;
00277     bool store_eq_ = 0;
00278     bool store_wq_ = 0;
00279     bool store_rq_ = 0;
00280     bool store_jparq_ = 0;
00281     bool store_jperpq_ = 0;
00282     bool store_lq_ = 0;
00283
00284     bool store_teq_ = 0;
00285
00286     bool store_SCF_ = 0;
00287     bool store_TCF_ = 0;
00288     bool refresh_q_ = 0;
```

```
00289
00290     bool store_TransCoeff_ = 0;
00291     bool store_MemoryKernels_ = 0;
00292
00293     bool print_snapshots_ = 0;
00294
00295 public:
00296     //
    ========================================================================================================
00297     //   Constructors
00298     //
    ========================================================================================================
00300     sampler(){};
00302     sampler(parameters par) : par_(par) {} ;
00304     sampler(parameters par, std::vector<topology::Vector2d> qvals) : par_(par), qvals_(qvals) {} ;
00305
00306
00307     //
    ========================================================================================================
00308     //   Intialization
00309     //
    ========================================================================================================
00311     void set_parameters(parameters par);
00313     void set_qvals(std::vector<topology::Vector2d> qvals);
00314
00315     //
    ========================================================================================================
00316     //   Switches management
00317     //
    ========================================================================================================
00318
00320     void switches_from_vector(const std::vector<bool>& boolvec);
00322     void switches_from_file(std::ifstream& infile, std::ofstream& stdoutfile);
00323
00325     void all_switches_on();
00327     void all_switches_off();
00328
00330     void print_snapshots_on();
00332     void print_snapshots_off();
00334     void check_on_fly_sampling();
00335
00336     //
    ========================================================================================================
00337     //   get functions
00338     //
    ========================================================================================================
00339     std::vector<bool> get_switches() const;
00340     inline bool print_snapshots() const { return print_snapshots_; };
00341     std::vector<topology::Vector2d> get_qvals() const { return qvals_; };
00342
00343     //
    ========================================================================================================
00344     //   binning functions and qvalue handling
00345     //
    ========================================================================================================
00347     void refresh_qvals(const topology::Vector2d& boxsize);
00349     std::vector<double> bin_qvals_to_q(std::vector<double> vals) const ;
00351     std::vector<std::complex<double> > bin_qvals_to_q(std::vector<std::complex<double> > vals) const ;
00352
00353
00354     //
    ========================================================================================================
00355     //   Data access (only for very useful data, shouldn't be overdone)
00356     //
    ========================================================================================================
00358     inline double get_last_temperature() { return temperature_.back(); };
00359
00360     //
    ========================================================================================================
00361     //   sampling functions
00362     //
    ========================================================================================================
00364     void sample_MSD(const std::vector<double>& MSD);
00365
00367     void sample(const group& G, const group& G_new, double t);
00369     void sample_static(const group& G, double t);
00371     void sample_TCF(const group& G_initial, const group& G_new, const double t);
00372
00374     void sample_snapshots(const group& G, const double t, std::string snapfile_name, std::ofstream&
    outfile);
00375
00377     void average();
00378
00379     //
    ========================================================================================================
00380     //   printing functions
00381     //
```

```
    ============================================================================================
00383     void print_matlab(std::ofstream& outfile);
00385     void print_averages_matlab(std::ofstream& outfile);
00386
00387
00388 };
00389 #endif /* SAMPLER_H_ */
```

7.30 topology.cpp File Reference

Cpp-File to declaration of namespace topology. Implements routines.

```
#include ¨topology.h¨
#include ¨computations.h¨
#include ¨math.h¨
#include <stdio.h>
#include <iomanip>
#include <iostream>
#include <fstream>
```

7.30.1 Detailed Description

Cpp-File to declaration of namespace topology. Implements routines.

Provides features for handling spatial vectors both for position and spin vectors. Not many detailed comments, most comments can be found in the corresponding file topology.h.

7.31 topology.h File Reference

Header-File to declaration of namespace topology. Defines classes Vector2d and angle2d (the latter redundant).

```
#include <stdlib.h>
#include <iostream>
#include <fstream>
#include <iomanip>
#include <vector>
#include <array>
#include ¨math.h¨
```

Classes

- class topology::Vector2d

  Mathematical 2d vectors. Can be added, multiplied by a scalar, norm computation is possible. There are print-to-file and print-to-command-line functions available.
- class topology::angle2d

  A (double-valued) angle in 2d space with the possibility of identifying it with its corresponding Vector2d unit vector. In the current state of the simulation, this is redundant.

Namespaces

- namespace topology

  Contains the vector classes and vector functions and operations. So far, only the class Vector2d is used in further routines and fully implemented.

Functions

- Vector2d topology::operator+ (const Vector2d &v, const Vector2d &w)
- Vector2d topology::operator- (const Vector2d &v, const Vector2d &w)
- Vector2d topology::operator∗ (const Vector2d &v, const double a)

  scalar ∗ operator
- Vector2d topology::operator∗ (const double a, const Vector2d &v)

  scalar ∗ operator
- Vector2d topology::operator∗ (const Vector2d &v, const int a)

  scalar ∗ operator
- Vector2d topology::operator∗ (const int a, const Vector2d &v)

  scalar ∗ operator
- Vector2d topology::operator/ (const Vector2d &v, const double a)

  scalar / operator
- Vector2d topology::operator/ (const Vector2d &v, const int a)

  scalar / operator
- double topology::norm2 (Vector2d v)

  Returns the L2 norm of a vector.
- Vector2d topology::normalized (Vector2d v)

  Normalizes a vector.
- double topology::periodic_distance_squared (const Vector2d &v, const Vector2d &w, const double &L)

  Returns squared distance |w - v|^2 considering periodic boundaries (square box, length L). Squared function faster to calculate.
- double topology::periodic_distance (const Vector2d &v, const Vector2d &w, const double &L)

  Returns distance |w - v| considering periodic boundaries (square box, length L). Taking sqrt takes more time than returning the squared quantity by dist_periodic_squared.
- Vector2d topology::periodic_distance_vector (const Vector2d &v, const Vector2d &w, const double &L)

  Returns distance vector w - v considering periodic boundaries (square box, length L).
- double topology::periodic_distance_squared (const Vector2d &v, const Vector2d &w, const Vector2d &L)

  Returns squared distance |w - v|^2 considering periodic boundaries (rectangular box, widths stored in L). Squared function faster to calculate.
- double topology::periodic_distance (const Vector2d &v, const Vector2d &w, const Vector2d &L)

  Returns squared distance |w - v| considering periodic boundaries (rectangular box, widths stored in L). Taking sqrt takes more time than returning the squared quantity by dist_periodic_squared.
- Vector2d topology::periodic_distance_vector (const Vector2d &v, const Vector2d &w, const Vector2d &L)

  Returns distance vector w - v considering periodic boundaries (rectangular box, widths stored in L).
- Vector2d topology::rotate (Vector2d v, double theta)

  Rotates a vector.
- Vector2d topology::rotate_orthogonal (Vector2d v)

  Rotates 90 degrees.
- double topology::innerproduct (Vector2d v, Vector2d w)

  Inner product.
- double topology::parallel_projection (Vector2d v, Vector2d w)

  Parallel projection.
- double topology::orthogonal_projection (Vector2d v, Vector2d w)

Parallel projection.

- Vector2d topology::random_vector (const Vector2d &minima, const Vector2d &maxima)

  Returns a random vector within a volume [minima, maxima].
- Vector2d topology::random_vector (const Vector2d &maxima)

  Returns a random vector within a volume [0,maxima].
- Vector2d topology::random_gaussian_vector (const double &sigma_squared)

  Returns a random vector with Gaussian distribution in each component.
- Vector2d topology::nearest_gridvec (Vector2d v, double gridsep)

  Returns nearest neighbor to 2D-Vector v on grid of grid point separation gridsep.
- Vector2d topology::nearest_gridvec (Vector2d v, Vector2d gridseps)

  Returns nearest neighbor to 2D-Vector v on grid of anisotropic grid point separation gridsep.
- std::vector< Vector2d > topology::qvalues_within_radius (double qmin, double qmax, Vector2d gridseps, int qsamps_per_bin)

  Creates a std::vector containing qsamps_per_bin 2D-vectors that lie on a grid with modulus between qmin and qmax. No vector appears double.
- Vector2d topology::random_vector_first_quadrant (const double length)

  Returns a random vector in the first quadrant.
- Vector2d topology::random_vector_sector (const double length, const double thetamin, const double thetamax)

  Returns a random vector within a sector given by thetamin, thetamax.
- Vector2d topology::random_velocity (const double r)

  Returns a random vector on a sphere surface of radius r.
- Vector2d topology::vector_from_angle (const double angle, const double r)

  Returns a vector of length r and orientation angle.
- Vector2d topology::vector_from_angle (const double angle)

  Returns a vector of unit length and orientation angle.
- double topology::angle_from_vector (const topology::Vector2d &v)

  Returns the orientation angle of a vector.
- Vector2d topology::spin (double theta)

  Returns a spin vector, i.e. unit vector, with given angle to x-axis.
- Vector2d topology::orthospin (double theta)

  Returns an orthogonal spin vector, i.e. a unit vector rotated 90^ °counterclockwise from the vector of spin(theta)
- Vector2d topology::vector_on_squarelattice (int index, int Nx, int Ny, double spacing)

  Index-dependent position vector for spin at index, square lattice.
- Vector2d topology::vector_on_trigonallattice (int index, int Nx, int Ny, double spacing)

  Index-dependent position vector for spin at index, trigonal lattice.
- void topology::print_matlab (const std::vector< Vector2d > &v, std::string name, std::ostream &outfile)

  Prints a list of vectors to in matlab-readable form.
- angle2d topology::operator+ (const angle2d &v, const angle2d &a)

  Addition operator.
- angle2d topology::operator- (const angle2d &v, const angle2d &w)

  Subtraction operator.
- angle2d topology::operator* (const angle2d &v, const double a)

  Multiplication operator (double, right)
- angle2d topology::operator* (const double a, const angle2d &v)

  Multiplication operator (double, left)
- angle2d topology::operator/ (const angle2d &v, const double a)

  Division operator (double)

7.31.1 Detailed Description

Header-File to declaration of namespace topology. Defines classes Vector2d and angle2d (the latter redundant).

Provides features for handling spatial vectors both for position and spin vectors.

7.32 topology.h

Go to the documentation of this file.

```
00001 #ifndef TOPOLOGY_H
00002 #define TOPOLOGY_H
00003 #include<stdlib.h>
00004 #include <iostream>
00005 #include <fstream>
00006 #include <iomanip>
00007
00008 #include<vector>
00009 #include<array>
00010 #include"math.h"
00011
00036 namespace topology
00037 {
00038
00052 class Vector2d{
00053 protected:
00054     std::array<double,2> v_;
00055
00056 public:
00057     //
      ============================================================================================
00058     //   Constructors
00059     //
      ============================================================================================
00060     Vector2d();
00061     Vector2d(double* v);
00062     Vector2d(const double& x);
00063     Vector2d(const int& x);
00064     Vector2d(const double x, const double y);
00065
00066     Vector2d(const Vector2d& w);
00067
00068     //
      ============================================================================================
00069     //   Print functions
00070     //
      ============================================================================================
00072     void print(std::ostream &outfile) const;
00074     void print() const;
00076     void print_xyz() const;
00078     inline double get_x() const { return v_[0]; };
00080     inline double get_y() const { return v_[1]; };
00081
00082     void set_x(double x);
00083     void set_y(double y);
00084
00086     inline const int size() const { return 2; }
00088     inline bool is_zero() const { return ( v_[0] == 0 && v_[1] == 0 ); };
00089
00090     //
      ============================================================================================
00091     //   Pointers to element
00092     //
      ============================================================================================
00093     double& operator[](int index);
00094     const double& operator[](int index) const;
00095
00096     //
      ============================================================================================
00097     //   Arithmetic operations
00098     //
      ============================================================================================
00099     Vector2d& operator+=(const Vector2d& w);
00100     Vector2d& operator-=(const Vector2d& w);
00101     Vector2d& operator*=(const double a);
00102     Vector2d& operator*=(const int a);
00103     // double operator*=(const Vector2d& w);    ///< Scalar product (not recommended for use)
00104     Vector2d& operator/=(const double a);
00105     Vector2d& operator/=(const int a);
```

```
00106     Vector2d operator-() const;
00107
00108     //
    ============================================================================================================
00109     //   Rotation, norms, angles, normalization
00110     //
    ============================================================================================================
00112     void rotate(const double theta);
00113
00115     inline double norm2() const { return v_[0] * v_[0] + v_[1] * v_[1]; }
00116
00118     inline double angle() { return std::atan2(this->v_[1],this->v_[0]); };
00119
00121     void normalized();
00122
00123     //
    ============================================================================================================
00124     //   Boundary conditions handling
00125     //
    ============================================================================================================
00126
00128
00132     void periodic_box(const Vector2d& minima, const Vector2d& maxima);
00134     void periodic_box(const Vector2d& maxima);
00135
00137     Vector2d get_boundary_handler_periodic_box(const Vector2d& maxima, const double cutoff) const;
00138
00140
00146     Vector2d get_boundary_handler_periodic_box(const Vector2d& minima,const Vector2d& maxima, const
    double cutoff) const;
00147
00148 };
00149
00150 //
    ============================================================================================================
00151 //   Typical linear and algebraic operations.
00152 //
    ============================================================================================================
00153 Vector2d operator+(const Vector2d& v, const Vector2d& w);
00154 Vector2d operator-(const Vector2d& v, const Vector2d& w);
00155 Vector2d operator*(const Vector2d& v, const double a);
00156 Vector2d operator*(const double a, const Vector2d& v);
00157 Vector2d operator*(const Vector2d& v, const int a);
00158 Vector2d operator*(const int a, const Vector2d& v);
00159 Vector2d operator/(const Vector2d& v, const double a);
00160 Vector2d operator/(const Vector2d& v, const int a);
00161
00162
00163 //
    ============================================================================================================
00164 //   Norms, rotations, distances
00165 //
    ============================================================================================================
00167 inline double norm2(Vector2d v) {
00168   return v.norm2();
00169 }
00170
00172 inline Vector2d normalized(Vector2d v) {
00173   return v/sqrt(v.norm2());
00174 }
00175
00177 double periodic_distance_squared(const Vector2d& v, const Vector2d& w, const double& L);
00179 inline double periodic_distance(const Vector2d& v, const Vector2d& w, const double& L) { return
    std::sqrt(periodic_distance_squared(v,w,L)); };
00181 Vector2d periodic_distance_vector(const Vector2d& v, const Vector2d& w, const double& L);
00182
00184 double periodic_distance_squared(const Vector2d& v, const Vector2d& w, const Vector2d& L);
00186 inline double periodic_distance(const Vector2d& v, const Vector2d& w, const Vector2d& L) { return
    std::sqrt(periodic_distance_squared(v,w,L)); };
00188 Vector2d periodic_distance_vector(const Vector2d& v, const Vector2d& w, const Vector2d& L);
00189
00191 inline Vector2d rotate(Vector2d v, double theta) {
00192   return Vector2d(cos(theta)*v.get_x()-sin(theta)*v.get_y(),
00193           sin(theta)*v.get_x()+cos(theta)*v.get_y());
00194 }
00195
00197 inline Vector2d rotate_orthogonal(Vector2d v) {
00198     return Vector2d(-v.get_y(),v.get_x());
00199 }
00200
00202 inline double innerproduct(Vector2d v, Vector2d w) {
00203   return v.get_x() * w.get_x() + v.get_y() * w.get_y();
00204 }
00205
00207 inline double parallel_projection(Vector2d v, Vector2d w) {
00208     return innerproduct(v,normalized(w));
00209 }
```

```
00210
00212 inline double orthogonal_projection(Vector2d v, Vector2d w) {
00213     return parallel_projection(v,rotate_orthogonal(w));
00214 }
00216 Vector2d random_vector(const Vector2d& minima, const Vector2d& maxima);
00218 Vector2d random_vector(const Vector2d& maxima);
00220 Vector2d random_gaussian_vector(const double& sigma_squared);
00221
00222
00224 Vector2d nearest_gridvec(Vector2d v, double gridsep);
00226 Vector2d nearest_gridvec(Vector2d v, Vector2d gridseps);
00228 std::vector<Vector2d> qvalues_within_radius(double qmin, double qmax, Vector2d gridseps, int
     qsamps_per_bin);
00229
00231 Vector2d random_vector_first_quadrant(const double length);
00232
00234 Vector2d random_vector_sector(const double length, const double thetamin, const double thetamax);
00235
00236
00238 Vector2d random_velocity(const double r);
00239
00241 Vector2d vector_from_angle(const double angle, const double r);
00242
00244 inline Vector2d vector_from_angle(const double angle) { return vector_from_angle(angle,1.0); };
00245
00247 inline double angle_from_vector(const topology::Vector2d& v) { return std::atan2(v.get_y(),v.get_x());
     };
00248
00250 inline Vector2d spin(double theta) { return Vector2d(cos(theta), sin(theta)); };
00252 inline Vector2d orthospin(double theta) { return Vector2d(-sin(theta), cos(theta)); };
00253
00254 Vector2d vector_on_squarelattice(int index, int Nx, int Ny, double spacing);
00255 Vector2d vector_on_trigonallattice(int index, int Nx, int Ny, double spacing);
00256
00257
00259 void print_matlab(const std::vector<Vector2d>& v, std::string name, std::ostream &outfile);
00260
00261
00262 //
     ================================================================================================================================
00263
00279 class angle2d {
00280 protected:
00282     double theta_;
00283
00284 public:
00286     angle2d() {} ;
00288     angle2d(const double& x);
00290     angle2d(const angle2d& w);
00291
00293     inline operator double() const { return theta_ ;}
00295     inline operator Vector2d() const { return Vector2d(cos(theta_), sin(theta_)) ;}
00296
00298     inline Vector2d spin() { return Vector2d(cos(theta_), sin(theta_)); };
00300     inline Vector2d orthospin() { return Vector2d(-sin(theta_), cos(theta_)); };
00301
00303     void boundary();
00304
00306     angle2d& operator+=(const double& a);
00308     angle2d& operator-=(const double& a);
00310     angle2d& operator*=(const double a);
00312     angle2d& operator/=(const double a);
00314     angle2d operator-() const;
00315 };
00316
00318 angle2d operator+(const angle2d& v, const angle2d& a);
00320 angle2d operator-(const angle2d& v, const angle2d& w);
00322 angle2d operator*(const angle2d& v, const double a);
00324 angle2d operator*(const double a, const angle2d& v);
00326 angle2d operator/(const angle2d& v, const double a);
00327 }
00328 #endif
```