

QMC2 Documentation

Generated by Doxygen 1.7.6.1

Mon Apr 1 2013 13:30:15

Contents

1	Class Index	1
1.1	Class Hierarchy	1
2	Class Index	3
2.1	Class List	3
3	Class Documentation	7
3.1	AlphaHarmonicOscillator Class Reference	7
3.1.1	Detailed Description	8
3.1.2	Member Function Documentation	8
3.1.2.1	get_parameter	8
3.1.2.2	get_qnums	8
3.1.2.3	get_variational_derivative	9
3.1.2.4	H	9
3.1.2.5	set_parameter	9
3.1.2.6	set_qnum_indie_terms	9
3.2	ASGD Class Reference	9
3.2.1	Detailed Description	11
3.2.2	Member Function Documentation	12
3.2.2.1	get_total_grad	12
3.2.2.2	get_variational_gradients	12
3.3	AtomCore Class Reference	12
3.3.1	Detailed Description	13
3.3.2	Member Function Documentation	13
3.3.2.1	get_pot_E	13
3.4	BasisFunctions Class Reference	13

3.4.1	Detailed Description	13
3.4.2	Member Function Documentation	14
3.4.2.1	eval	14
3.5	Blocking Class Reference	14
3.5.1	Member Function Documentation	15
3.5.1.1	block_data	15
3.5.1.2	get_initial_error	15
3.5.1.3	get_unique_blocks	15
3.6	Brute_Force Class Reference	16
3.6.1	Detailed Description	16
3.6.2	Member Function Documentation	16
3.6.2.1	copy_walker	16
3.6.2.2	get_necessities	17
3.6.2.3	reset_walker	17
3.6.2.4	update_walker	17
3.7	Coulomb Class Reference	17
3.7.1	Detailed Description	18
3.7.2	Member Function Documentation	18
3.7.2.1	get_pot_E	18
3.8	Diffusion Class Reference	18
3.8.1	Detailed Description	19
3.8.2	Member Function Documentation	20
3.8.2.1	call_RNG	20
3.8.2.2	get_g_ratio	20
3.8.2.3	get_GBfunc	20
3.8.2.4	get_new_pos	20
3.8.2.5	set_dt	21
3.9	Distribution Class Reference	21
3.9.1	Detailed Description	21
3.9.2	Constructor & Destructor Documentation	22
3.9.2.1	Distribution	22
3.9.3	Member Function Documentation	22
3.9.3.1	dump	22
3.9.3.2	finalize	22

3.9.3.3	rerun	22
3.10	DMC Class Reference	22
3.10.1	Detailed Description	24
3.10.2	Member Function Documentation	24
3.10.2.1	Evolve_walker	24
3.10.2.2	iterate_walker	25
3.10.2.3	move_authorized	25
3.10.2.4	node_comm	25
3.10.2.5	save_distribution	25
3.10.2.6	set_trial_positions	25
3.10.2.7	switch_souls	25
3.10.3	Member Data Documentation	26
3.10.3.1	K	26
3.11	DMCparams Struct Reference	26
3.11.1	Detailed Description	27
3.12	ErrorEstimator Class Reference	27
3.12.1	Detailed Description	28
3.12.2	Constructor & Destructor Documentation	28
3.12.2.1	ErrorEstimator	28
3.12.3	Member Function Documentation	28
3.12.3.1	combine_mean	29
3.12.3.2	combine_variance	29
3.12.3.3	finalize	29
3.12.3.4	init_file	29
3.12.3.5	node_comm_gather_data	29
3.12.3.6	node_comm_scatter_data	29
3.12.3.7	update_data	30
3.13	ExpandedBasis Class Reference	30
3.13.1	Member Function Documentation	31
3.13.1.1	del_phi	31
3.13.1.2	lapl_phi	31
3.13.1.3	phi	31
3.14	Fermions Class Reference	31
3.14.1	Detailed Description	33

3.14.2	Member Function Documentation	33
3.14.2.1	calc_for_newpos	33
3.14.2.2	copy_walker	33
3.14.2.3	get_spatial_grad	33
3.14.2.4	get_spatial_wf	33
3.14.2.5	initialize	33
3.14.2.6	make_merged_inv	33
3.14.2.7	reset_walker	34
3.14.2.8	update_walker	34
3.14.3	Member Data Documentation	34
3.14.3.1	I	34
3.15	Fokker_Planck Class Reference	34
3.15.1	Detailed Description	35
3.15.2	Member Function Documentation	35
3.15.2.1	get_g_ratio	35
3.15.2.2	get_new_pos	35
3.16	GeneralParams Struct Reference	36
3.16.1	Detailed Description	36
3.16.2	Member Data Documentation	37
3.16.2.1	systemConstant	37
3.17	Harmonic_osc Class Reference	37
3.17.1	Detailed Description	37
3.17.2	Member Function Documentation	37
3.17.2.1	get_pot_E	37
3.18	hydrogenicOrbitals Class Reference	38
3.18.1	Detailed Description	39
3.18.2	Member Function Documentation	39
3.18.2.1	get_dell_alpha_phi	39
3.18.2.2	get_parameter	39
3.18.2.3	set_parameter	40
3.18.2.4	set_qnum_indie_terms	40
3.19	Importance Class Reference	40
3.19.1	Detailed Description	41
3.19.2	Member Function Documentation	41

3.19.2.1	calculate_energy_necessities	41
3.19.2.2	copy_walker	41
3.19.2.3	get_necessities	41
3.19.2.4	reset_walker	41
3.19.2.5	update_necessities	42
3.19.2.6	update_walker	42
3.20	Jastrow Class Reference	42
3.20.1	Detailed Description	43
3.20.2	Member Function Documentation	43
3.20.2.1	get_derivative_num	43
3.20.2.2	get_dJ_matrix	44
3.20.2.3	get_dJ_matrix	44
3.20.2.4	get_grad	44
3.20.2.5	get_grad	44
3.20.2.6	get_j_ratio	44
3.20.2.7	get_lapl_sum	45
3.20.2.8	get_laplaciansum_num	45
3.20.2.9	get_parameter	45
3.20.2.10	get_val	45
3.20.2.11	get_variational_derivative	46
3.20.2.12	initialize	46
3.20.2.13	set_parameter	46
3.21	Minimizer Class Reference	46
3.21.1	Detailed Description	48
3.21.2	Constructor & Destructor Documentation	48
3.21.2.1	Minimizer	48
3.21.3	Member Function Documentation	48
3.21.3.1	dump_output	48
3.21.3.2	error_output	48
3.21.3.3	finalize_output	48
3.21.3.4	update_parameters	48
3.22	MinimizerParams Struct Reference	49
3.22.1	Detailed Description	49
3.23	No_Jastrow Class Reference	49

3.23.1	Detailed Description	50
3.23.2	Member Function Documentation	50
3.23.2.1	get_dJ_matrix	50
3.23.2.2	get_grad	51
3.23.2.3	get_grad	51
3.23.2.4	get_j_ratio	51
3.23.2.5	get_lapl_sum	51
3.23.2.6	get_parameter	51
3.23.2.7	get_val	52
3.23.2.8	get_variational_derivative	52
3.23.2.9	initialize	52
3.23.2.10	set_parameter	52
3.24	NO_STDOUT Class Reference	53
3.24.1	Detailed Description	53
3.25	Orbitals Class Reference	53
3.25.1	Detailed Description	55
3.25.2	Member Function Documentation	55
3.25.2.1	del_phi	55
3.25.2.2	get_parameter	55
3.25.2.3	get_variational_derivative	56
3.25.2.4	lapl_phi	56
3.25.2.5	num_ddiff	56
3.25.2.6	num_diff	56
3.25.2.7	phi	57
3.25.2.8	set_parameter	57
3.25.2.9	set_qnum_indie_terms	57
3.25.2.10	testDell	57
3.25.2.11	testLaplace	58
3.26	OutputHandler Class Reference	58
3.26.1	Detailed Description	59
3.26.2	Constructor & Destructor Documentation	59
3.26.2.1	OutputHandler	59
3.26.3	Member Function Documentation	60
3.26.3.1	dump	60

3.26.3.2	finalize	60
3.26.3.3	init_file	60
3.26.3.4	post_pointer_init	60
3.26.3.5	set_min_ptr	60
3.26.3.6	set_qmc_ptr	61
3.27	OutputParams Struct Reference	61
3.27.1	Detailed Description	61
3.28	Pade_Jastrow Class Reference	61
3.28.1	Detailed Description	62
3.28.2	Member Function Documentation	62
3.28.2.1	get_dJ_matrix	62
3.28.2.2	get_grad	63
3.28.2.3	get_grad	63
3.28.2.4	get_j_ratio	63
3.28.2.5	get_lapl_sum	63
3.28.2.6	get_parameter	64
3.28.2.7	get_val	64
3.28.2.8	get_variational_derivative	64
3.28.2.9	initialize	64
3.28.2.10	set_parameter	64
3.29	ParParams Struct Reference	65
3.29.1	Detailed Description	65
3.30	Potential Class Reference	65
3.30.1	Detailed Description	66
3.30.2	Member Function Documentation	66
3.30.2.1	get_pot_E	66
3.31	QMC Class Reference	66
3.31.1	Detailed Description	70
3.31.2	Constructor & Destructor Documentation	70
3.31.2.1	QMC	70
3.31.3	Member Function Documentation	70
3.31.3.1	calculate_energy_necessities	70
3.31.3.2	calculate_local_energy	70
3.31.3.3	copy_walker	70

3.31.3.4	diffuse_walker	71
3.31.3.5	dump_output	71
3.31.3.6	estimate_error	71
3.31.3.7	finalize_output	71
3.31.3.8	get_gradients	71
3.31.3.9	get_gradients	71
3.31.3.10	get_KE	71
3.31.3.11	get_lapsum	72
3.31.3.12	get_wf_value	72
3.31.3.13	metropolis_test	72
3.31.3.14	move_authorized	72
3.31.3.15	reset_walker	72
3.31.3.16	set_spin_state	73
3.31.3.17	set_trial_positions	73
3.31.3.18	test_gradients	73
3.31.3.19	test_ratios	73
3.31.3.20	update_walker	73
3.31.4	Member Data Documentation	74
3.31.4.1	cycle	74
3.31.4.2	n_w	74
3.32	Sampling Class Reference	74
3.32.1	Member Function Documentation	75
3.32.1.1	call_RNG	75
3.32.1.2	copy_walker	75
3.32.1.3	get_g_ratio	76
3.32.1.4	get_necessities	76
3.32.1.5	reset_walker	76
3.32.1.6	set_spin_state	76
3.32.1.7	update_pos	76
3.32.1.8	update_walker	76
3.32.2	Member Data Documentation	77
3.32.2.1	diffusion	77
3.32.2.2	end	77
3.32.2.3	start	77

3.33 Simple Class Reference	77
3.33.1 Detailed Description	78
3.33.2 Member Function Documentation	78
3.33.2.1 get_g_ratio	78
3.33.2.2 get_new_pos	78
3.34 SimpleVar Class Reference	79
3.34.1 Detailed Description	79
3.34.2 Member Function Documentation	80
3.34.2.1 update_data	80
3.35 STDOUT Class Reference	80
3.35.1 Detailed Description	80
3.36 stdoutASGD Class Reference	80
3.36.1 Detailed Description	81
3.36.2 Member Function Documentation	81
3.36.2.1 dump	81
3.36.2.2 post_pointer_init	81
3.37 stdoutDMC Class Reference	81
3.37.1 Detailed Description	82
3.37.2 Member Function Documentation	82
3.37.2.1 dump	82
3.38 System Class Reference	83
3.38.1 Detailed Description	84
3.38.2 Member Function Documentation	84
3.38.2.1 calc_for_newpos	84
3.38.2.2 copy_walker	84
3.38.2.3 get_potential_energy	85
3.38.2.4 get_spatial_grad	85
3.38.2.5 initialize	85
3.38.2.6 reset_walker	85
3.38.2.7 set_spin_state	85
3.38.2.8 update_walker	85
3.38.3 Member Data Documentation	86
3.38.3.1 end	86
3.38.3.2 start	86

3.39	SystemObjects Struct Reference	86
3.39.1	Detailed Description	86
3.40	VariationalParams Struct Reference	87
3.40.1	Detailed Description	87
3.41	VMC Class Reference	87
3.41.1	Detailed Description	88
3.41.2	Member Function Documentation	89
3.41.2.1	move_authorized	89
3.41.2.2	save_distribution	89
3.41.2.3	set_trial_positions	89
3.41.2.4	store_walkers	89
3.42	VMCparams Struct Reference	89
3.42.1	Detailed Description	90
3.43	Walker Class Reference	90
3.43.1	Detailed Description	91
3.43.2	Member Function Documentation	92
3.43.2.1	calc_r_i2	92
3.43.2.2	calc_r_rel	92
3.43.2.3	get_r_i	92
3.43.2.4	get_r_i2	92
3.43.2.5	kill	92
3.43.2.6	make_rel_matrix	93
3.43.2.7	print	93
3.43.2.8	recv_soul	93
3.43.2.9	ressurrect	93
3.43.2.10	send_soul	93

Chapter 1

Class Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

BasisFunctions	13
Diffusion	18
Fokker_Planck	34
Simple	77
DMCparams	26
ErrorEstimator	27
Blocking	14
SimpleVar	79
GeneralParams	36
Jastrow	42
No_Jastrow	49
Pade_Jastrow	61
Minimizer	46
ASGD	9
MinimizerParams	49
Orbitals	53
AlphaHarmonicOscillator	7
ExpandedBasis	30
hydrogenicOrbitals	38
OutputHandler	58
Distribution	21
stdoutASGD	80
stdoutDMC	81
OutputParams	61
ParParams	65
Potential	65
AtomCore	12

Coulomb	17
Harmonic_osc	37
QMC	66
DMC	22
VMC	87
Sampling	74
Brute_Force	16
Importance	40
STDOUT	80
NO_STDOUT	53
System	83
Fermions	31
SystemObjects	86
VariationalParams	87
VMCparams	89
Walker	90

Chapter 2

Class Index

2.1 Class List

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

AlphaHarmonicOscillator	Harmonic Oscillator single particle wave function class. Uses the HarmonicOscillator BasisFunction subclasses auto-generated by - SymPy through the orbitalsGenerator tool	7
ASGD	Implementation for the Adaptive Stochastic Gradient Descent method (ASGD) Used to find optimal variational parameters using adaptive step lengths	9
AtomCore	Implementation of the Atom Core potential. $-Z/r$	12
BasisFunctions	The Superclass shell for orbital basis functions	13
Blocking	14
Brute_Force	Implementation of the Brute Force QMC . Uses the Simple diffusion class. All methods are empty except for the energy necessities part which requires the gradients to be calculated (not using the Quantum Force)	16
Coulomb	Implementation of the Coulomb potential. $1/r_{ij}$	17
Diffusion	Class containing rules for walker movement based on diffusion models. Serves as class member in the Sampling class. Brute force implies the Simple diffusion model, while Importance Sampling implies the Fokker Planck diffusion	18
Distribution	Class for calculating distribution functions such as the one-body density. Does not collect data itself, but works merely as a control organ for the QMC class, calling it's methods for storing position data . . .	21

DMC	Implementation of the Diffusion Monte-Carlo Method. Very little needs to be added when the QMC superclass holds all the general functionality	22
DMCparams	Struct used to initialize DMC parameters	26
ErrorEstimator	Class handling error estimations of the QMC methods. The QMC class holds an object of this type, calling the <code>update_data</code> function in order to update the sampling pool. <code>finalize()</code> then either dumps the samples to file for later processing, or calculates an estimate	27
ExpandedBasis	30
Fermions	The Fermion system class	31
Fokker_Planck	Anisotropic diffusion by the Fokker-Planck equation	34
GeneralParams	Struct used to initialize general parameters	36
Harmonic_osc	Implementation of the Harmonic Oscillator potential. $0.5*w**2*r**2$	37
hydrogenicOrbitals	Hydrogen-like single particle wave function class. Uses the hydrogenic BasisFunction subclasses auto-generated by SymPy through the orbitalsGenerator tool	38
Importance	Implementation of Importance sampled QMC . Using the Fokker-Planck diffusion class. Introduces the Quantum Force	40
Jastrow	The class representing the Jastrow correlation functions Holds all data concerning the Jastrow function and it's influence on the QMC algorithm	42
Minimizer	Class for minimization methods used to obtain optimal variational parameters	46
MinimizerParams	Struct used to initialize Minimization parameters	49
No_Jastrow	Class loaded when no correlation factor is used	49
NO_STDOUT	Class for suppressing standard output. Every node but the master has this. If-tests around <code>cout</code> is avoided	53
Orbitals	Superclass for the single particle orbital classes. Handles everything specific regarding choice of single particle basis	53
OutputHandler	Class for handling output-methods. Designed to avoid rewriting code, as well as avoid if-tests if output is not desired	58
OutputParams	Struct used to initialize output parameters	61

Pade_Jastrow	The Pade Jastrow factor with a single variational parameter	61
ParParams	Struct used to initialize parallelization parameters	65
Potential	Superclass for potentials. Potentials are stores in a vector in the system object	65
QMC	The QMC superclass. Holds implementations of general functions for both VMC and DMC in order to avoid rewriting code and emphasize the similarities	66
Sampling	74
Simple	Simple Isotropic diffusion model	77
SimpleVar	Calculates the simple variance of the sampled values	79
STDOUT	Class for handling standard output. Only the master node has this object	80
stdoutASGD	Class for handling the output of ASGD . Ouputs values such as the variational gradients, step length, variational parameters, etc	80
stdoutDMC	Class for handling the output of DMC . Outputs values such as the trial energy, dmc energy, number of walkers, etc	81
System	The system class separating Fermions and Bosons. Designed to generalize the solver in terms of particle species	83
SystemObjects	Struct used to initialize system objects	86
VariationalParams	Struct used to initialize the varational parameters	87
VMC	Implementation of the Variational Monte-Carlo Method. Very little needs to be added when the QMC superclass holds all the general functionality	87
VMCparams	Struct used to initialize VMC parameters	89
Walker	Class representing a Random Walker . Holds position data, alive state, etc. Designed to lighten function arguments, and ease implementation of QMC methods involving multiple walkers. Alot of values are stored to avoid calculating the same value twice	90

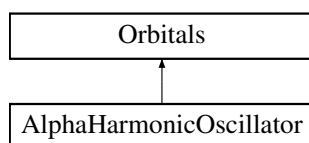
Chapter 3

Class Documentation

3.1 AlphaHarmonicOscillator Class Reference

Harmonic Oscillator single particle wave function class. Uses the HarmonicOscillator - BasisFunction subclasses auto-generated by SymPy through the orbitalsGenerator tool.

Inheritance diagram for AlphaHarmonicOscillator:



Public Member Functions

- **AlphaHarmonicOscillator** ([GeneralParams](#) &, [VariationalParams](#) &)
- **AlphaHarmonicOscillator** ([GeneralParams](#) &)
- void [set_qnum_indie_terms](#) (const [Walker](#) *walker, int i)

Protected Member Functions

- double [get_variational_derivative](#) (const [Walker](#) *walker, int n)
- void [get_qnums](#) ()
- double [H](#) (int n, double x) const
Method for calculating Hermite polynomials.
- double [get_parameter](#) (int n)
- void [set_parameter](#) (double parameter, int n)

Protected Attributes

- double `w`
The oscillator frequency.
- double * `alpha`
Pointer to the variational parameter alpha. Shared address with all the BasisFunction subclasses.
- double * `k`
*Pointer to $\sqrt{\alpha*w}$. Shared address with all the BasisFunction subclasses.*
- double * `k2`
*Pointer to $\alpha*w$. Shared address with all the BasisFunction subclasses.*
- double * `w_over_a`
- double * `exp_factor`
Pointer to a factor precalculated by `set_qnum_indie_terms()`. Shared address with all the BasisFunction subclasses.
- arma::Mat< int > `qnums`
Quantum number matrix needed by the variational derivatives.

Friends

- class **ExpandedBasis**

3.1.1 Detailed Description

Harmonic Oscillator single particle wave function class. Uses the HarmonicOscillator - BasisFunction subclasses auto-generated by SymPy through the orbitalsGenerator tool.

3.1.2 Member Function Documentation

3.1.2.1 `double AlphaHarmonicOscillator::get_parameter (int n)` [`inline`, `protected`, `virtual`]

Returns

The variational parameter alpha.

Implements [Orbitals](#).

3.1.2.2 `void AlphaHarmonicOscillator::get_qnums ()` [`protected`]

Calculates the quantum numbers of the oscillator and stores them in the matrix qnums.

3.1.2.3 `double AlphaHarmonicOscillator::get_variational_derivative (const Walker * walker, int n)` `[protected, virtual]`

Overridden superclass method implementing closed form expressions using Hermite polynomials.

Reimplemented from [Orbitals](#).

3.1.2.4 `double AlphaHarmonicOscillator::H (int n, double x) const` `[protected]`

Method for calculating Hermite polynomials.

Parameters

<i>n</i>	The degree of the Hermite polynomial.
<i>x</i>	The argument for evaluating the polynomial.

3.1.2.5 `void AlphaHarmonicOscillator::set_parameter (double parameter, int n)` `[inline, protected, virtual]`

Sets a new value for the alpha and updates all the pointer values.

Implements [Orbitals](#).

3.1.2.6 `void AlphaHarmonicOscillator::set_qnum_indie_terms (const Walker * walker, int i)` `[inline, virtual]`

Calculates the exponential term shared by all oscillator function once pr. particle to save CPU-time.

See also

[Orbitals::set_qnum_indie_terms\(\)](#)

Reimplemented from [Orbitals](#).

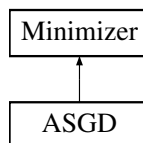
The documentation for this class was generated from the following files:

- `src/Orbitals/AlphaHarmonicOscillator/AlphaHarmonicOscillator.h`
- `src/Orbitals/AlphaHarmonicOscillator/AlphaHarmonicOscillator.cpp`

3.2 ASGD Class Reference

Implementation for the Adaptive Stochastic Gradient Descent method ([ASGD](#)) Used to find optimal variational parameters using adaptive step lengths.

Inheritance diagram for ASGD:



Public Member Functions

- **ASGD** (*VMC* *, *MinimizerParams* &, const *ParParams* &)
- void *minimize* ()

Method for executing the minimization main solver.

Protected Member Functions

- void *get_total_grad* ()
Method for calculating the total gradient.
- void *update_parameters* ()
Calculates the step and updates parameters.
- void *output_cycle* ()
Standard output of the progress.
- void *thermalize_walkers* ()
Thermalizes a set of walkers before the main loop.
- double *f* (double x)
Function for calculating the adaptive step.
- void *get_variational_gradients* (*Walker* *walker, double e_local)
Method for updating the vectors needed to calculate the total variational derivative.

Protected Attributes

- int *n_c*
The correlation length between storing two walkers after thermalization.
- int *n_c_SGD*
The number of samples used to estimate expectation values.
- int *SGDsamples*
The number of ASGD cycles.
- int *n_walkers*
The number of walkers.
- int *thermalization*
The number of thermalization cycles used on walkers.
- int *sample*
The current ASGD cycle.
- double *t_prev*

The previous t.

- double `t`

The current t.

- double `step`

The current calculates step.

- double `max_step`

The maximum threshold on a step.

- double `E`

The energy summation variable used to calculate the mean.

- double `a`

ASGD step parameter.

- double `A`

ASGD step parameter.

- double `f_min`

ASGD step parameter.

- double `f_max`

ASGD step parameter.

- double `w`

ASGD step parameter.

- `Walker` ** `walkers`

The walkers used to sample expectation values.

- `Walker` ** `trial_walkers`

- `arma::rowvec` `parameter`

- `arma::rowvec` `gradient`

Sumamtion vector for the trial wave function's variational derivatives.

- `arma::rowvec` `gradient_local`

Sumamtion vector for the trial wave function's variational derivatives times the energy.

- `arma::rowvec` `gradient_old`

The previous total gradient.

- `arma::rowvec` `gradient_tot`

The current total gradient.

Friends

- class `stdoutASGD`

3.2.1 Detailed Description

Implementation for the Adaptive Stochastic Gradient Descent method ([ASGD](#)) Used to find optimal variational parameters using adaptive step lengths.

3.2.2 Member Function Documentation

3.2.2.1 void ASGD::get_total_grad () [protected]

Method for calculating the total gradient.

Updates the error estimator with statistics.

3.2.2.2 void ASGD::get_variational_gradients (Walker * walker, double e_local) [protected]

Method for updating the vectors needed to calculate the total variational derivative.

Calculates the single particle variational derivatives V and accumulates V and $V \cdot e_{\text{local}}$.

Parameters

<i>e_local</i>	The local energy of the current walker at the current time step.
----------------	--

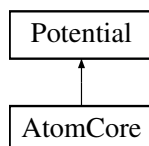
The documentation for this class was generated from the following files:

- src/Minimizer/ASGD/ASGD.h
- src/Minimizer/ASGD/ASGD.cpp

3.3 AtomCore Class Reference

Implementation of the Atom Core potential. $-Z/r$.

Inheritance diagram for AtomCore:



Public Member Functions

- **AtomCore** ([GeneralParams](#) &)
- double [get_pot_E](#) (const [Walker](#) *walker) const
Method for calculating a walker's potential energy.

Protected Attributes

- int [Z](#)
The core charge.

3.3.1 Detailed Description

Implementation of the Atom Core potential. $-Z/r$.

3.3.2 Member Function Documentation

3.3.2.1 `double AtomCore::get_pot_E (const Walker * walker) const` `[virtual]`

Method for calculating a walker's potential energy.

Method overridden by subclasses.

Parameters

<code>walker</code>	The walker for which the potential energy should be calculated.
---------------------	---

Returns

The potential energy.

Implements [Potential](#).

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

- `src/Potential/AtomCore/AtomCore.h`
- `src/Potential/AtomCore/AtomCore.cpp`

3.4 BasisFunctions Class Reference

The Superclass shell for orbital basis functions.

Public Member Functions

- virtual double [eval](#) (const [Walker](#) *walker, int i)=0
The method representing the orbitals functional expression.

3.4.1 Detailed Description

The Superclass shell for orbital basis functions.

Each single particle orbital has it's own implementation as a subclass of this class. A set of orbitals can then be loaded into the [Orbitals](#) basis_function vectors. An orbital-Generator script is supplied to autogenerate CPP files using this class.

See also

[Orbitals::basis_functions](#)
[Orbitals::dell_basis_functions](#)
[Orbitals::lapl_basis_functions](#)

3.4.2 Member Function Documentation

3.4.2.1 `virtual double BasisFunctions::eval (const Walker * walker, int i)` [pure virtual]

The method representing the orbitals functional expression.

Parameters

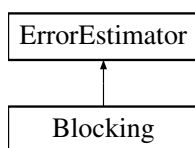
<i>walker</i>	The Walker whose position the orbital is to be evaluted at.
<i>i</i>	The particle to be evaluated (Single particle function).

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

- src/BasisFunctions/BasisFunctions.h
- src/BasisFunctions/BasisFunctions.cpp

3.5 Blocking Class Reference

Inheritance diagram for Blocking:



Public Member Functions

- **Blocking** (int [n_c](#), [ParParams](#) &pp, std::string filename="blocking_out", std::string path=".", int n_b=100, int maxb=10000, int minb=10, bool [rerun](#)=false)
- **Blocking** (int [n_c](#), std::string filename="blocking_out", std::string path=".", int n_b=100, int maxb=10000, int minb=10)
- double [estimate_error](#) ()
Estimates the error based on the subclass implementation.
- void [get_initial_error](#) ()
- void [get_unique_blocks](#) (arma::Row< int > &block_sizes, int &n)
Calculates the block sizes.

Protected Member Functions

- void [block_data](#) (int block_size, double &var, double &mean)
Calculates the variance and mean of the dataset with the specified block size.

Protected Attributes

- arma::rowvec **local_block**
- int [min_block_size](#)
The minimum amount of samples in one block.
- int [max_block_size](#)
The maximum amount of samples in one block.
- int [n_block_samples](#)
The total amount of different block sizes.

3.5.1 Member Function Documentation

3.5.1.1 void **Blocking::block_data** (int *block_size*, double & *var*, double & *mean*)
[protected]

Calculates the variance and mean of the dataset with the specified block size.

Parameters

<i>block_size</i>	The number of samples in each block.
<i>var</i>	Reference to the variance of the block's means.
<i>mean</i>	Reference to the mean of the block's means. Needed to combine the variances from different processes.

3.5.1.2 void **Blocking::get_initial_error** ()

Calculates the variance as in [SimpleVar](#)

3.5.1.3 void **Blocking::get_unique_blocks** (arma::Row< int > & *block_sizes*, int & *n*)

Calculates the block sizes.

Due to integer division, alot of sizes becomes equal. Only unique block sizes are returned.

Parameters

<i>block_sizes</i>	Vector containing the block sizes
<i>n</i>	The number of unqiue block sizes

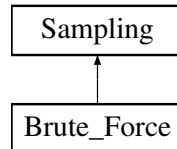
The documentation for this class was generated from the following files:

- src/ErrorEstimator/Blocking/Blocking.h
- src/ErrorEstimator/Blocking/Blocking.cpp

3.6 Brute_Force Class Reference

Implementation of the Brute Force [QMC](#). Uses the Simle diffusion class. All methods are empty except for the energy necessities part which requires the gradients to be calculated (not using the Quantum Force).

Inheritance diagram for Brute_Force:



Public Member Functions

- **Brute_Force** ([GeneralParams](#) &)
- void [update_walker](#) ([Walker](#) *walker_pre, const [Walker](#) *walker_post, int particle) const
- void [get_necessities](#) ([Walker](#) *walker)
 - Method for calculating the sampling specific necessary values.*
- void [update_necessities](#) (const [Walker](#) *walker_pre, [Walker](#) *walker_post, int particle) const
 - Method for updating the sampling specific necessary values.*
- void [calculate_energy_necessities](#) ([Walker](#) *walker) const
 - Method for calculating the sampling specific necessary values in order to compute the local energy.*
- void [copy_walker](#) (const [Walker](#) *parent, [Walker](#) *child) const
 - Method for copying the sampling specific parts of a walker.*
- void [reset_walker](#) (const [Walker](#) *walker_pre, [Walker](#) *walker_post, int particle) const

3.6.1 Detailed Description

Implementation of the Brute Force [QMC](#). Uses the Simle diffusion class. All methods are empty except for the energy necessities part which requires the gradients to be calculated (not using the Quantum Force).

3.6.2 Member Function Documentation

3.6.2.1 void [Brute_Force::copy_walker](#) (const [Walker](#) * *parent*, [Walker](#) * *child*) const
 [inline, virtual]

Method for copying the sampling specific parts of a walker.

See also

[QMC::copy_walker\(\)](#)

Implements [Sampling](#).

```
3.6.2.2 void Brute_Force::get_necessities ( Walker * walker ) [inline,  
virtual]
```

Method for calculating the sampling specific necessary values.

Called after a trial position is set.

Implements [Sampling](#).

```
3.6.2.3 void Brute_Force::reset_walker ( const Walker * walker_pre, Walker *  
walker_post, int particle ) const [inline, virtual]
```

See also

[QMC::reset_walker\(\)](#)

Implements [Sampling](#).

```
3.6.2.4 void Brute_Force::update_walker ( Walker * walker_pre, const Walker *  
walker_post, int particle ) const [inline, virtual]
```

See also

[QMC::update_walker\(\)](#)

Implements [Sampling](#).

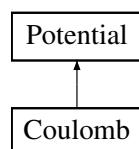
The documentation for this class was generated from the following files:

- src/Sampling/Brute_Force/Brute_Force.h
- src/Sampling/Brute_Force/Brute_Force.cpp

3.7 Coulomb Class Reference

Implementation of the [Coulomb](#) potential. $1/r_{ij}$.

Inheritance diagram for Coulomb:



Public Member Functions

- **Coulomb** ([GeneralParams](#) &)
- double [get_pot_E](#) (const [Walker](#) *walker) const
Method for calculating a walker's potential energy.

3.7.1 Detailed Description

Implementation of the [Coulomb](#) potential. $1/r_{ij}$.

3.7.2 Member Function Documentation

3.7.2.1 double **Coulomb::get_pot_E** (const [Walker](#) * *walker*) const [virtual]

Method for calculating a walker's potential energy.

Method overridden by subclasses.

Parameters

<i>walker</i>	The walker for which the potential energy should be calculated.
---------------	---

Returns

The potential energy.

Implements [Potential](#).

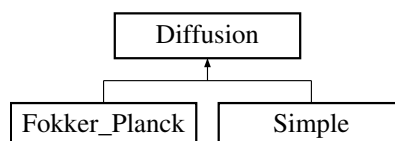
The documentation for this class was generated from the following files:

- src/Potential/Coulomb/Coulomb.h
- src/Potential/Coulomb/Coulomb.cpp

3.8 Diffusion Class Reference

Class containing rules for walker movement based on diffusion models. Serves as class member in the [Sampling](#) class. Brute force implies the [Simple](#) diffusion model, while - [Importance Sampling](#) implies the Fokker Planck diffusion.

Inheritance diagram for Diffusion:



Public Member Functions

- **Diffusion** (int n_p, int dim, double timestep, seed_type random_seed, double D)
- virtual double **get_new_pos** (const Walker *walker, int i, int j)
Virtual function returning the new position.
- virtual double **get_g_ratio** (const Walker *walker_post, const Walker *walker_pre) const =0
Calculates the Diffusion Green's function ratio needed by metropolis.
- double **get_GBfunc** (double E_x, double E_y, double E_T) const
Calculates the Branching Green's function ratio needed by DMC.
- double **call_RNG** ()
Calls a uniform random number generator.
- void **set_qmc_ptr** (QMC *qmc)
- void **set_dt** (double dt)
Function for altering the time step.
- double **get_dt** () const
- double **get_std** () const

Protected Attributes

- int **n_p**
- int **dim**
- QMC * **qmc**
The qmc main solver object. Not needed?
- double **timestep**
The discrete time step.
- double **D**
The diffusion constant.
- long **random_seed**
The random seed. Needs to be stored for some RNGs to work.
- double **std**
The standard deviation from QMC stored for efficiency. $\sqrt{2D \cdot \text{timestep}}$.

3.8.1 Detailed Description

Class containing rules for walker movement based on diffusion models. Serves as class member in the [Sampling](#) class. Brute force implies the [Simple](#) diffusion model, while [Importance Sampling](#) implies the Fokker Planck diffusion.

See also

[Brute_Force](#), [Importance](#).

3.8.2 Member Function Documentation

3.8.2.1 `double Diffusion::call_RNG ()`

Calls a uniform random number generator.

Returns a random uniform number on [0,1).

3.8.2.2 `virtual double Diffusion::get_g_ratio (const Walker * walker_post, const Walker * walker_pre) const` [pure virtual]

Calculates the [Diffusion](#) Green's function ratio needed by metropolis.

Parameters

<code>walker_post</code>	Walker at current time step.
<code>walker_pre</code>	Walker at previous time step.

Returns

The [Diffusion](#) Green's function ratio.

Implemented in [Simple](#), and [Fokker_Planck](#).

3.8.2.3 `double Diffusion::get_GBfunc (double E_x, double E_y, double E_T) const` [inline]

Calculates the Branching Green's function ratio needed by [DMC](#).

Parameters

<code>E_x</code>	Energy at current time step
<code>E_y</code>	Energy at previous time step

Returns

The Branching Green's function ratio

3.8.2.4 `double Diffusion::get_new_pos (const Walker * walker, int i, int j)` [virtual]

Virtual function returning the new position.

Returns the simple diffusion step if not overridden.

Parameters

<code>i</code>	Particle number.
<code>j</code>	dimension (x,y,z).

Returns

The new position (relative to the old).

Reimplemented in [Fokker_Planck](#), and [Simple](#).

3.8.2.5 void Diffusion::set_dt (double dt) [inline]

Function for altering the time step.

Takes care of consequences. Time step should only be altered using this function.

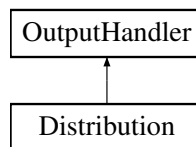
The documentation for this class was generated from the following files:

- src/Diffusion/Diffusion.h
- src/Diffusion/Diffusion.cpp

3.9 Distribution Class Reference

Class for calculating distribution functions such as the one-body density. Does not collect data itself, but works merely as a control organ for the [QMC](#) class, calling it's methods for storing position data.

Inheritance diagram for Distribution:

**Public Member Functions**

- [Distribution](#) ([ParParams](#) &, std::string path, std::string name)
Constructor.
- void [dump](#) ()
- void [finalize](#) ()
Method for calculating the distribution.
- void [rerun](#) (int n_p, int N, double bin_edge)
Method for re-calculating the distribution given a stores set of position data.

3.9.1 Detailed Description

Class for calculating distribution functions such as the one-body density. Does not collect data itself, but works merely as a control organ for the [QMC](#) class, calling it's methods for storing position data.

3.9.2 Constructor & Destructor Documentation

3.9.2.1 `Distribution::Distribution (ParParams & pp, std::string path, std::string name)`

Constructor.

Parameters

<i>path</i>	The path where output is stored (or read).
<i>name</i>	Name of the file.

3.9.3 Member Function Documentation

3.9.3.1 `void Distribution::dump () [virtual]`

Signals [QMC](#) solver to store position data.

Implements [OutputHandler](#).

3.9.3.2 `void Distribution::finalize () [virtual]`

Method for calculating the distribution.

Overrides the superclass implementation.

Reimplemented from [OutputHandler](#).

3.9.3.3 `void Distribution::rerun (int n_p, int N, double bin_edge)`

Method for re-calculating the distribution given a stores set of position data.

Scatters the data across nodes.

Parameters

<i>n_p</i>	Number of particles in the set.
<i>N</i>	Number of mesh points used in the histogram.
<i>bin_edge</i>	The Cartesian position of the end points of the histogram.

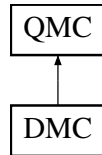
The documentation for this class was generated from the following files:

- `src/OutputHandler/Distribution/Distribution.h`
- `src/OutputHandler/Distribution/Distribution.cpp`

3.10 DMC Class Reference

Implementation of the [Diffusion](#) Monte-Carlo Method. Very little needs to be added when the [QMC](#) superclass holds all the general functionality.

Inheritance diagram for DMC:



Public Member Functions

- [DMC](#) ([GeneralParams](#) &, [DMCparams](#) &, [SystemObjects](#) &, [ParParams](#) &, [VMC](#) *vmc, bool dist_out)
Constructor.
- void [run_method](#) ()
Method used for executing the solver.
- void [output](#) ()
Method for standard output.

Static Public Attributes

- static const int [K](#) = 50
Factor of empty space for walkers over initial walkers.
- static const int **check_thresh** = 25
- static const int **sendcount_thresh** = 20

Protected Member Functions

- void [set_trial_positions](#) ()
Method for setting the trial position of all [DMC](#) walkers.
- void [iterate_walker](#) (int k, int n_b=1)
Method for iterating a walker one time step.
- void [Evolve_walker](#) (int k, double GB)
Method for the birth/death process of a walker.
- void [bury_the_dead](#) ()
Method for cleaning up the dead walkers and compact the list.
- void [update_energies](#) ()
Method for updating the [DMC](#) energy and calculating the new trial energy.
- bool [move_authorized](#) (double A)
- void **reset_parameters** ()
- void [node_comm](#) ()
- void [save_distribution](#) ()
Method for storing positional data for the Distribution.
- void [switch_souls](#) (int root, int root_id, int dest, int dest_id)

Method for sending a walker between two nodes.

- void [normalize_population](#) ()

Method for evening out the number of walkers on each node.

Protected Attributes

- int [n_w_last](#)

The amount of walkers at the time the walker loop was initiated.

- int [n_w_tot](#)

The total number of walkers across all nodes.

- arma::uvec [n_w_list](#)

List of the number of walkers of each node.

- int **deaths**

- int **block_size**

- int **samples**

- unsigned long **tot_samples**

- double **E_tot**

- double [dmc_E](#)

The [DMC](#) energy.

- double [dmc_E_unscaled](#)

The accumulative [DMC](#) energy: The sum of all previous trial energies.

- double [E_T](#)

The trial energy at the current cycle.

- double [E](#)

The accumulative energy for each cycle.

Friends

- class **stdoutDMC**

3.10.1 Detailed Description

Implementation of the [Diffusion](#) Monte-Carlo Method. Very little needs to be added when the [QMC](#) superclass holds all the general functionality.

3.10.2 Member Function Documentation

3.10.2.1 void **DMC::Evolve_walker** (int *k*, double *GB*) [protected]

Method for the birth/death process of a walker.

Parameters

<i>GB</i>	The branching Green's Function.
<i>k</i>	The index of the walker.

3.10.2.2 `void DMC::iterate_walker (int k, int n_b = 1)` [protected]

Method for iterating a walker one time step.

Parameters

<i>n_b</i>	The number of block samples used in the iteration.
<i>k</i>	The index of the walker.

3.10.2.3 `bool DMC::move_authorized (double A)` [inline, protected, virtual]

In case of [DMC](#), we must let the system have the possibility to override the metropolis test (fixed node approximation in case of a Fermion system)

Implements [QMC](#).

3.10.2.4 `void DMC::node_comm ()` [protected, virtual]

For each process: -Calculates the total number of walkers. -Sums up the energies sampled. -Sums up the total number of samples made.

Implements [QMC](#).

3.10.2.5 `void DMC::save_distribution ()` [protected, virtual]

Method for storing positional data for the Distribution.

Stores the position data from all currently alive [DMC](#) walkers every `dist_tresh` cycle.

Implements [QMC](#).

3.10.2.6 `void DMC::set_trial_positions ()` [protected, virtual]

Method for setting the trial position of all [DMC](#) walkers.

In case [VMC](#) is not run prior to [DMC](#), trial positions must be set.

Implements [QMC](#).

3.10.2.7 `void DMC::switch_souls (int root, int root_id, int dest, int dest_id)` [protected]

Method for sending a walker between two nodes.

Parameters

<i>root</i>	The node from which the walker is sent.
<i>root_id</i>	The index of the walker being sent from root.
<i>dest</i>	The node which receives the walker.
<i>dest_id</i>	The index where the walker is to be received.

See also

[Walker::send_soul\(\)](#), [Walker::recv_soul\(\)](#)

3.10.3 Member Data Documentation

3.10.3.1 `const int DMC::K = 50` `[static]`

Factor of empty space for walkers over initial walkers.

See also

[QMC::QMC\(\)](#)

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

- `src/QMC/DMC/DMC.h`
- `src/QMC/DMC/DMC.cpp`

3.11 DMCparams Struct Reference

Struct used to initialize [DMC](#) parameters.

Public Attributes

- `int n_c`
The number of cycles.
- `int therm`
Thermalization cycles.
- `int n_b`
Number of block samples pr. walker pr. cycle.
- `int n_w`
Number of walkers.
- `double dt`
Time step.

3.11.1 Detailed Description

Struct used to initialize [DMC](#) parameters.

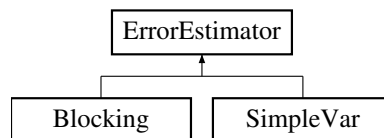
The documentation for this struct was generated from the following file:

- `src/QMChaders.h`

3.12 ErrorEstimator Class Reference

Class handling error estimations of the [QMC](#) methods. The [QMC](#) class holds an object of this type, calling the `update_data` function in order to update the sampling pool. [finalize\(\)](#) then either dumps the samples to file for later processing, or calculates an estimate.

Inheritance diagram for ErrorEstimator:



Public Member Functions

- [ErrorEstimator](#) (int [n_c](#), std::string filename, std::string path, bool parallel, int node, int n_nodes, bool [rerun](#)=false)

Constructor.

- double [combine_mean](#) (double mean, int n, int n_tot=0)

Calculates the combined mean of n_nodes means.

- double [combine_variance](#) (double var, double mean=0, int n=0)

Calculates the combined variance of n_nodes variances.

- void [finalize](#) ()
- void [node_comm_gather_data](#) ()
- void [node_comm_scatter_data](#) ()
- void [init_file](#) ()

Opens a file with filename at path supplied in constructor.

- virtual double [estimate_error](#) ()=0

Estimates the error based on the subclass implementation.

- virtual void [update_data](#) (double val)

Adds values to the data vector.

Public Attributes

- bool [data_to_file](#)

If true, the data vector are stored to file.

- bool `output_to_file`

If `init_file()` method is called, this flag is true.

Protected Attributes

- int `n_c`

Size of the data vector.

- int `i`

Count variable for the data vector.

- bool `parallel`

- bool `is_master`

- int `node`

- int `n_nodes`

- bool `rerun`

If false, data is assumed to already exist.

- std::string `filename`

- std::string `path`

- std::ofstream `file`

- arma::rowvec `data`

The vector containing the samples used in error calculation.

3.12.1 Detailed Description

Class handling error estimations of the `QMC` methods. The `QMC` class holds an object of this type, calling the `update_data` function in order to update the sampling pool. `finalize()` then either dumps the samples to file for later processing, or calculates an estimate.

3.12.2 Constructor & Destructor Documentation

3.12.2.1 `ErrorEstimator::ErrorEstimator (int n_c, std::string filename, std::string path, bool parallel, int node, int n_nodes, bool rerun = false)`

Constructor.

Parameters

<i>n_c</i>	The expected number of samples to be stored.
<i>filename</i>	The name of the file. Only necessary if <code>init_file()</code> is called.
<i>path</i>	The path where the data and/or the file is stored (or read).

3.12.3 Member Function Documentation

3.12.3.1 double ErrorEstimator::combine_mean (double *mean*, int *n*, int *n_tot* = 0)

Calculates the combined mean of *n_nodes* means.

Only useful for parallel calls.

Parameters

<i>mean</i>	The local mean on an individual node
<i>n</i>	The number of samples used to calculate the local mean.
<i>n_tot</i>	The total number of samples on all nodes. Calculated if not supplied.

3.12.3.2 double ErrorEstimator::combine_variance (double *var*, double *mean* = 0, int *n* = 0)

Calculates the combined variance of *n_nodes* variances.

Only useful for parallel calls.

Parameters

<i>var</i>	The local variance on an individual node
<i>mean</i>	The local mean on an individual node
<i>n_tot</i>	The total number of samples used on all nodes.

3.12.3.3 void ErrorEstimator::finalize ()

if [output_to_file]: Closes opened files if [data_to_file]: Stores accumulated data. if data vector was used, it's memory is freed.

3.12.3.4 void ErrorEstimator::init_file ()

Opens a file with filename at path supplied in constructor.

Subclass implementations can call this function. Superclass does not.

3.12.3.5 void ErrorEstimator::node_comm_gather_data ()

Gathers the data vectors from all processes into a single one on the master node.

3.12.3.6 void ErrorEstimator::node_comm_scatter_data ()

Exact reverse of [node_comm_gather_data\(\)](#)

3.12.3.7 void ErrorEstimator::update_data (double val) [virtual]

Adds values to the data vector.

Can be overridden if storage is not wanted.

Parameters

<i>val</i>	A local sample of the quantity of which the error is calculated
------------	---

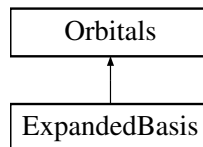
Reimplemented in [SimpleVar](#).

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

- src/ErrorEstimator/ErrorEstimator.h
- src/ErrorEstimator/ErrorEstimator.cpp

3.13 ExpandedBasis Class Reference

Inheritance diagram for ExpandedBasis:



Public Member Functions

- **ExpandedBasis** ([GeneralParams](#) &gp, [Orbitals](#) *basis, int m, std::string coeff-Path)
- double [phi](#) (const [Walker](#) *walker, int particle, int q_num)
Calculates the single particle wave function for a given walker's particle.
- double [del_phi](#) (const [Walker](#) *walker, int particle, int q_num, int d)
Calculates the single particle wave function derivative for a given walker's particle and dimension.
- double [lapl_phi](#) (const [Walker](#) *walker, int particle, int q_num)
Calculates the single particle wave function for a given walker's particle.

Protected Attributes

- int **basis_size**
- arma::mat **coeffs**
- [Orbitals](#) * **basis**

3.13.1 Member Function Documentation

3.13.1.1 **double ExpandedBasis::del_phi** (const Walker * *walker*, int *particle*, int *q_num*, int *d*) [virtual]

Calculates the single particle wave function derivative for a given walker's particle and dimension.

Parameters

<i>q_num</i>	The quantum number index.
<i>d</i>	The dimension for which the derivative should be calculated (x,y,z).

Reimplemented from [Orbitals](#).

3.13.1.2 **double ExpandedBasis::lapl_phi** (const Walker * *walker*, int *particle*, int *q_num*) [virtual]

Calculates the single particle wave function for a given walker's particle.

Parameters

<i>q_num</i>	The quantum number index.
--------------	---------------------------

Reimplemented from [Orbitals](#).

3.13.1.3 **double ExpandedBasis::phi** (const Walker * *walker*, int *particle*, int *q_num*) [virtual]

Calculates the single particle wave function for a given walker's particle.

Parameters

<i>q_num</i>	The quantum number index.
--------------	---------------------------

Reimplemented from [Orbitals](#).

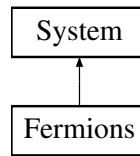
The documentation for this class was generated from the following files:

- src/Orbitals/ExpandedBasis/ExpandedBasis.h
- src/Orbitals/ExpandedBasis/ExpandedBasis.cpp

3.14 Fermions Class Reference

The Fermion system class.

Inheritance diagram for Fermions:



Public Member Functions

- **Fermions** ([GeneralParams](#) &, [Orbitals](#) *)
- void [get_spatial_grad](#) ([Walker](#) *walker, int particle) const
Method for calculating the changed part of the spatial gradient.
- void [get_spatial_grad_full](#) ([Walker](#) *walker) const
Method for calculating the full spatial gradient.
- double [get_spatial_ratio](#) (const [Walker](#) *walker_post, const [Walker](#) *walker_pre, int particle)
Method for calculating the spatial wave function ratios between two subsequent time steps.
- double [get_spatial_lapl_sum](#) (const [Walker](#) *walker) const
Method for calculating the sum of all Laplacians for a given walker.
- bool [allow_transition](#) ()
Fixed node approximation.
- void [copy_walker](#) (const [Walker](#) *parent, [Walker](#) *child) const
- void [update_walker](#) ([Walker](#) *walker_pre, const [Walker](#) *walker_post, int particle) const
- void [reset_walker](#) (const [Walker](#) *walker_pre, [Walker](#) *walker_post, int particle) const
- double [get_spatial_wf](#) (const [Walker](#) *walker)
- void [initialize](#) ([Walker](#) *walker)
- void [calc_for_newpos](#) (const [Walker](#) *walker_old, [Walker](#) *walker_new, int i)

Protected Member Functions

- void [make_merged_inv](#) ([Walker](#) *walker)
Method for calculating the Slater matrix inverse.
- void [update_inverse](#) (const [Walker](#) *walker_old, [Walker](#) *walker_new, int particle)
Method for updating the inverse given that we moved one particle.

Protected Attributes

- arma::rowvec [l](#)
The diagonal of the new slater matrix times the old slater inverse.
- bool [node_crossed](#)
True if the spatial ratio is negative.

3.14.1 Detailed Description

The Fermion system class.

3.14.2 Member Function Documentation

3.14.2.1 `void Fermions::calc_for_newpos (const Walker * walker_old, Walker * walker_new, int i)` `[inline, virtual]`

When a particle is moved, the inverse is updated.

Implements [System](#).

3.14.2.2 `void Fermions::copy_walker (const Walker * parent, Walker * child) const` `[inline, virtual]`

Copies the inverse.

Implements [System](#).

3.14.2.3 `void Fermions::get_spatial_grad (Walker * walker, int particle) const` `[virtual]`

Method for calculating the changed part of the spatial gradient.

Depending on which particle we moved, one of the spatial wave function parts (it is split) will be unchanged.

Implements [System](#).

3.14.2.4 `double Fermions::get_spatial_wf (const Walker * walker)` `[inline, virtual]`

The determinant of each spin value multiplied.

Implements [System](#).

3.14.2.5 `void Fermions::initialize (Walker * walker)` `[inline, virtual]`

Calculates the inverse.

Implements [System](#).

3.14.2.6 `void Fermions::make_merged_inv (Walker * walker)` `[protected]`

Method for calculating the Slater matrix inverse.

The merged inverse is made by concatenating the two slater matrix inverses. This way we can sum freely over particles without having to if-test on the spin.

3.14.2.7 `void Fermions::reset_walker (const Walker * walker_pre, Walker * walker_post, int particle) const` `[inline, virtual]`

Resets the inverse.

Implements [System](#).

3.14.2.8 `void Fermions::update_walker (Walker * walker_pre, const Walker * walker_post, int particle) const` `[inline, virtual]`

Updates the inverse.

Implements [System](#).

3.14.3 Member Data Documentation

3.14.3.1 `arma::rowvec Fermions::l` `[protected]`

The diagonal of the new slater matrix times the old slater inverse.

Needed for updating the inverse. Stored because only half of the vector is changed when moving one particle.

See also

[System::set_spin_state\(\)](#)

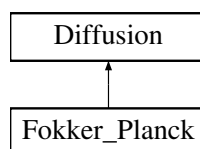
The documentation for this class was generated from the following files:

- `src/System/Fermions/Fermions.h`
- `src/System/Fermions/Fermions.cpp`

3.15 Fokker_Planck Class Reference

Anisotropic diffusion by the Fokker-Planck equation.

Inheritance diagram for Fokker_Planck:



Public Member Functions

- **Fokker_Planck** (int n_p, int dim, double [timestep](#), seed_type [random_seed](#), double [D](#)=0.5)
- double [get_g_ratio](#) (const [Walker](#) *walker_post, const [Walker](#) *walker_pre) const
Calculates the [Diffusion](#) Green's function ratio needed by metropolis.
- double [get_new_pos](#) (const [Walker](#) *walker, int i, int j)
Virtual function returning the new position.

3.15.1 Detailed Description

Anisotropic diffusion by the Fokker-Planck equation.

3.15.2 Member Function Documentation

3.15.2.1 double [Fokker_Planck::get_g_ratio](#) (const [Walker](#) * *walker_post*, const [Walker](#) * *walker_pre*) const [virtual]

Calculates the [Diffusion](#) Green's function ratio needed by metropolis.

Parameters

<i>walker_post</i>	Walker at current time step.
<i>walker_pre</i>	Walker at previous time step.

Returns

The [Diffusion](#) Green's function ratio.

Implements [Diffusion](#).

3.15.2.2 double [Fokker_Planck::get_new_pos](#) (const [Walker](#) * *walker*, int *i*, int *j*)
[inline, virtual]

Virtual function returning the new position.

Returns the simple diffusion step if not overridden.

Parameters

<i>i</i>	Particle number.
<i>j</i>	dimension (x,y,z).

Returns

The new position (relative to the old).

Reimplemented from [Diffusion](#).

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

- `src/Diffusion/Fokker_Planck/Fokker_Planck.h`
- `src/Diffusion/Fokker_Planck/Fokker_Planck.cpp`

3.16 GeneralParams Struct Reference

Struct used to initialize general parameters.

Public Attributes

- `int n_p`
The number of particles.
- `int dim`
The dimension.
- `seed_type random_seed`
The random number generator's seed.
- `double systemConstant`
The constant used in systems.
- `bool doMIN`
- `bool doVMC`
- `bool doDMC`
- `bool do_blocking`
- `bool use_jastrow`
- `bool use_coulomb`
- `std::string system`
String specifying the system type, e.g. "Atoms".
- `std::string sampling`
String specifying the sampling type, e.g. "IS".
- `std::string runpath`
The directory which the simulation is set to run.

3.16.1 Detailed Description

Struct used to initialize general parameters.

3.16.2 Member Data Documentation

3.16.2.1 double GeneralParams::systemConstant

The constant used in systems.

e.g. charge for atoms and oscillator frequency for quantum dots.

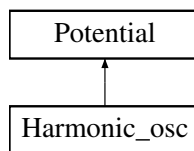
The documentation for this struct was generated from the following file:

- src/QMHeaders.h

3.17 Harmonic_osc Class Reference

Implementation of the Harmonic Oscillator potential. $0.5*w**2*r**2$.

Inheritance diagram for Harmonic_osc:



Public Member Functions

- **Harmonic_osc** ([GeneralParams](#) &)
- double [get_pot_E](#) (const [Walker](#) *walker) const
Method for calculating a walker's potential energy.

Protected Attributes

- double [w](#)
The oscillator frequency.

3.17.1 Detailed Description

Implementation of the Harmonic Oscillator potential. $0.5*w**2*r**2$.

3.17.2 Member Function Documentation

3.17.2.1 double Harmonic_osc::get_pot_E (const Walker * walker) const [virtual]

Method for calculating a walker's potential energy.

Method overridden by subclasses.

Parameters

<i>walker</i>	The walker for which the potential energy should be calculated.
---------------	---

Returns

The potential energy.

Implements [Potential](#).

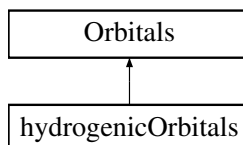
The documentation for this class was generated from the following files:

- src/Potential/Harmonic_osc/Harmonic_osc.h
- src/Potential/Harmonic_osc/Harmonic_osc.cpp

3.18 hydrogenicOrbitals Class Reference

Hydrogen-like single particle wave function class. Uses the hydrogenic BasisFunction subclasses auto-generated by SymPy through the orbitalsGenerator tool.

Inheritance diagram for hydrogenicOrbitals:



Public Member Functions

- **hydrogenicOrbitals** ([GeneralParams](#) &, [VariationalParams](#) &)
- **hydrogenicOrbitals** ([GeneralParams](#) &)
- void [set_qnum_indie_terms](#) (const [Walker](#) *walker, int i)

Protected Member Functions

- double [get_variational_derivative](#) (const [Walker](#) *walker, int n)
! Overridden superclass method implementing closed form expressions generated by SymPy.
- double [get_dell_alpha_phi](#) (const [Walker](#) *walker, int qnum, int i)
Method for calculating a single particle wave functions variational derivative.
- double [get_parameter](#) (int n)
- void [set_parameter](#) (double parameter, int n)

Protected Attributes

- int [Z](#)
The charge of the core.
- double * [alpha](#)
Pointer to the variational parameter alpha. Shared address with all the BasisFunction subclasses.
- double * [k](#)
*Pointer to $\alpha * Z$. Shared address with all the BasisFunction subclasses.*
- double * [k2](#)
*Pointer to $(\alpha * Z)^2$. Shared address with all the BasisFunction subclasses.*
- double * [r22d](#)
- double * [r2d](#)
- double * [exp_factor_n1](#)
Pointer to a factor precalculated by [set_qnum_indie_terms\(\)](#). Shared address with all the BasisFunction subclasses.
- double * [exp_factor_n2](#)
Pointer to a factor precalculated by [set_qnum_indie_terms\(\)](#). Shared address with all the BasisFunction subclasses.
- arma::Mat< int > [qnums](#)

Friends

- class [ExpandedBasis](#)

3.18.1 Detailed Description

Hydrogen-like single particle wave function class. Uses the hydrogenic BasisFunction subclasses auto-generated by SymPy through the orbitalsGenerator tool.

3.18.2 Member Function Documentation

3.18.2.1 `double hydrogenicOrbitals::get_dell_alpha_phi (const Walker * walker, int qnum, int i)` [protected]

Method for calculating a single particle wave functions variational derivative.

Parameters

<i>i</i>	The particle number.
----------	----------------------

3.18.2.2 `double hydrogenicOrbitals::get_parameter (int n)` [inline, protected, virtual]

Returns

The variational parameter alpha.

Implements [Orbitals](#).

3.18.2.3 `void hydrogenicOrbitals::set_parameter (double parameter, int n)`
`[inline, protected, virtual]`

Sets a new value for the alpha and updates all the pointer values.

Implements [Orbitals](#).

3.18.2.4 `void hydrogenicOrbitals::set_qnum_indie_terms (const Walker * walker, int i)`
`[virtual]`

Calculates the exponential terms $\exp(-r/n)$ for all needed *n* once pr. particle to save CPU-time.

See also

[Orbitals::set_qnum_indie_terms\(\)](#)

Reimplemented from [Orbitals](#).

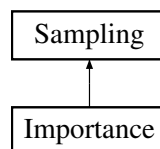
The documentation for this class was generated from the following files:

- `src/Orbitals/hydrogenicOrbitals/hydrogenicOrbitals.h`
- `src/Orbitals/hydrogenicOrbitals/hydrogenicOrbitals.cpp`

3.19 Importance Class Reference

Implementation of [Importance](#) sampled [QMC](#). Using the Fokker-Planck diffusion class. Introduces the Quantum Force.

Inheritance diagram for Importance:

**Public Member Functions**

- **Importance** ([GeneralParams](#) &)

- void [update_walker](#) ([Walker](#) *walker_pre, const [Walker](#) *walker_post, int particle) const
- void [reset_walker](#) (const [Walker](#) *walker_pre, [Walker](#) *walker_post, int particle) const
- void [get_necessities](#) ([Walker](#) *walker)
- void [update_necessities](#) (const [Walker](#) *walker_pre, [Walker](#) *walker_post, int particle) const
- void [calculate_energy_necessities](#) ([Walker](#) *walker) const
- void [copy_walker](#) (const [Walker](#) *parent, [Walker](#) *child) const

3.19.1 Detailed Description

Implementation of [Importance](#) sampled [QMC](#). Using the Fokker-Planck diffusion class. Introduces the Quantum Force.

3.19.2 Member Function Documentation

3.19.2.1 void [Importance::calculate_energy_necessities](#) ([Walker](#) * *walker*) const
[inline, virtual]

No energy necessities (they are already calculated).

Implements [Sampling](#).

3.19.2.2 void [Importance::copy_walker](#) (const [Walker](#) * *parent*, [Walker](#) * *child*) const
[inline, virtual]

The gradients and the Quantum force is copied.

Implements [Sampling](#).

3.19.2.3 void [Importance::get_necessities](#) ([Walker](#) * *walker*) [inline, virtual]

The gradients and the Quantum force are calculated.

Implements [Sampling](#).

3.19.2.4 void [Importance::reset_walker](#) (const [Walker](#) * *walker_pre*, [Walker](#) * *walker_post*, int *particle*) const [virtual]

The parts of the gradients with the same spin as the moved particle are reset.

Implements [Sampling](#).

3.19.2.5 `void Importance::update_necessities (const Walker * walker_pre, Walker * walker_post, int particle) const` [inline, virtual]

The gradients are updated and the Quantum force is re-calculated.

Implements [Sampling](#).

3.19.2.6 `void Importance::update_walker (Walker * walker_pre, const Walker * walker_post, int particle) const` [virtual]

The parts of the gradients with the same spin as the moved particle are updated.

Implements [Sampling](#).

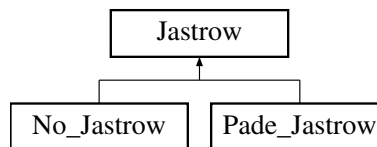
The documentation for this class was generated from the following files:

- src/Sampling/Importance/Importance.h
- src/Sampling/Importance/Importance.cpp

3.20 Jastrow Class Reference

The class representing the [Jastrow](#) correlation functions Holds all data concerning the [Jastrow](#) function and it's influence on the [QMC](#) algorithm.

Inheritance diagram for Jastrow:



Public Member Functions

- **Jastrow** (int n_p, int dim)
- virtual void [initialize](#) ()=0
- virtual double [get_val](#) (const [Walker](#) *walker) const =0
- virtual double [get_j_ratio](#) (const [Walker](#) *walker_new, const [Walker](#) *walker_old, int i) const =0
Calculates the ratio of the [Jastrow](#) factor needed by metropolis.
- virtual void [get_grad](#) ([Walker](#) *walker) const =0
- virtual void [get_grad](#) (const [Walker](#) *walker_pre, [Walker](#) *walker_post, int i) const =0
Updates the gradient for a new particle move.
- virtual void [get_dJ_matrix](#) ([Walker](#) *walker, int i) const =0
Updates the summation factors of [Jastrow](#) factors closed form expressions.

- void [get_dJ_matrix](#) ([Walker](#) *walker) const
Calculates the summation factors of [Jastrow](#) factors closed form expressions.
- virtual double [get_lapl_sum](#) ([Walker](#) *walker) const =0
Method for calculating the Laplacian.

Protected Member Functions

- virtual double [get_parameter](#) (int n)=0
Returns variational parameters.
- virtual void [set_parameter](#) (double param, int n)=0
Sets variational parameters.
- virtual double [get_variational_derivative](#) (const [Walker](#) *walker, int n)
Calculates the derivative of the [Jastrow](#) factor with respect to a variational parameter.
- double [get_derivative_num](#) ([Walker](#) *walker, int i, int d) const
Numerical Cartesian derivative.
- double [get_laplaciansum_num](#) ([Walker](#) *walker) const
Numerical Cartesian Laplacian.

Protected Attributes

- int **n_p**
- int **n2**
- int **dim**
- bool [active](#)
Parameter false if [No_Jastrow](#) is loaded.

Friends

- class **Minimizer**
- class **ASGD**
- class **stdoutASGD**

3.20.1 Detailed Description

The class representing the [Jastrow](#) correlation functions Holds all data concerning the [Jastrow](#) function and it's influence on the [QMC](#) algorithm.

3.20.2 Member Function Documentation

3.20.2.1 double [Jastrow::get_derivative_num](#) ([Walker](#) * *walker*, int *i*, int *d*) const
[protected]

Numerical Cartesian derivative.

For use in [get_grad\(\)](#) when no closed form expression is implemented.

Parameters

<i>i</i>	Particle number.
<i>d</i>	Dimension (x,y,z).

3.20.2.2 `virtual void Jastrow::get_dJ_matrix (Walker * walker, int i) const` [pure virtual]

Updates the summation factors of [Jastrow](#) factors closed form expressions.

Used to optimize the calculations as few of these terms change as we move a particle.

Parameters

<i>i</i>	Particle number.
----------	------------------

Implemented in [Pade_Jastrow](#), and [No_Jastrow](#).

3.20.2.3 `void Jastrow::get_dJ_matrix (Walker * walker) const`

Calculates the summation factors of [Jastrow](#) factors closed form expressions.

Used to optimize the calculations as few of these terms change as we move a particle.

3.20.2.4 `virtual void Jastrow::get_grad (Walker * walker) const` [pure virtual]

Calculates the entire Cartesian gradient.

Implemented in [Pade_Jastrow](#), and [No_Jastrow](#).

3.20.2.5 `virtual void Jastrow::get_grad (const Walker * walker_pre, Walker * walker_post, int i) const` [pure virtual]

Updates the gradient for a new particle move.

Parameters

<i>walker_post</i>	Walker at current time step
<i>walker_pre</i>	Walker at previous time step
<i>i</i>	Particle number.

Implemented in [Pade_Jastrow](#), and [No_Jastrow](#).

3.20.2.6 `virtual double Jastrow::get_j_ratio (const Walker * walker_new, const Walker * walker_old, int i) const` [pure virtual]

Calculates the ratio of the [Jastrow](#) factor needed by metropolis.

Parameters

<i>walker_new</i>	Walker at current time step
<i>walker_old</i>	Walker at previous time step
<i>i</i>	The particle number.

Implemented in [Pade_Jastrow](#), and [No_Jastrow](#).

3.20.2.7 `virtual double Jastrow::get_lapl_sum (Walker * walker) const` [pure virtual]

Method for calculating the Laplacian.

Calculates the sum of all particles Laplacians.

Implemented in [No_Jastrow](#), and [Pade_Jastrow](#).

3.20.2.8 `double Jastrow::get_laplaciansum_num (Walker * walker) const` [protected]

Numerical Cartesian Laplacian.

For use in [get_lapl_sum\(\)](#) when no closed form expression is implemented.

3.20.2.9 `virtual double Jastrow::get_parameter (int n)` [protected, pure virtual]

Returns variational parameters.

Parameters

<i>n</i>	The index of the sought variational parameter
----------	---

Returns

Variational parameter with index [n]

Implemented in [Pade_Jastrow](#), and [No_Jastrow](#).

3.20.2.10 `virtual double Jastrow::get_val (const Walker * walker) const` [pure virtual]

Calculates the value of the [Jastrow](#) Factor at the walker's position.

Implemented in [Pade_Jastrow](#), and [No_Jastrow](#).

3.20.2.11 `double Jastrow::get_variational_derivative (const Walker * walker, int n)`
`[protected, virtual]`

Calculates the derivative of the [Jastrow](#) factor with respect to a variational parameter.

Parameters

<i>n</i>	The index of the variational parameter for which the derivative is to be taken
<i>walker</i>	The walker holds the positions etc. needed to evaluate the derivative

Reimplemented in [No_Jastrow](#), and [Pade_Jastrow](#).

3.20.2.12 `virtual void Jastrow::initialize ()` `[pure virtual]`

Initializes the non-variational parameters needed by the [Jastrow](#) Factor.

Implemented in [Pade_Jastrow](#), and [No_Jastrow](#).

3.20.2.13 `virtual void Jastrow::set_parameter (double param, int n)`
`[protected, pure virtual]`

Sets variational parameters.

Parameters

<i>n</i>	The index of the sought variational parameter
<i>param</i>	The new value of parameter [<i>n</i>]

Implemented in [No_Jastrow](#), and [Pade_Jastrow](#).

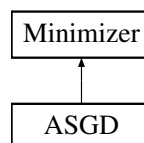
The documentation for this class was generated from the following files:

- `src/Jastrow/Jastrow.h`
- `src/Jastrow/Jastrow.cpp`

3.21 Minimizer Class Reference

Class for minimization methods used to obtain optimal variational parameters.

Inheritance diagram for Minimizer:



Public Member Functions

- [Minimizer](#) ([VMC](#) *vmc, const [ParParams](#) &, const arma::rowvec &alpha, const arma::rowvec &beta)
Constructor.
- void [add_output](#) ([OutputHandler](#) *output_handler)
Method used for loading the [stdoutASGD](#) object.
- [Orbitals](#) * [get_orbitals](#) ()
- [Jastrow](#) * [get_jastrow](#) ()
- virtual void [minimize](#) ()=0
Method for executing the minimization main solver.
- void [output](#) (std::string message, double number=-1)
Method for dumping variational parameter values to screen.
- void [add_error_estimator](#) ([ErrorEstimator](#) *error_estimator)
Method used to add error estimators.

Protected Member Functions

- void [dump_output](#) ()
- void [finalize_output](#) ()
- void [error_output](#) ()
- virtual void [update_parameters](#) ()=0
Method for updating the variational parameters based on the previous step.

Protected Attributes

- int [n_nodes](#)
- bool [is_master](#)
- [VMC](#) * [vmc](#)
Uses [VMC](#) methods to calculate stochastic variational gradients.
- [STDOUT](#) * [std_out](#)
Output object. Wraps and replaces std::cout.
- std::stringstream [s](#)
- int [Nspatial_params](#)
The number of variational parameters in the spatial trial wave function.
- int [Njastrow_params](#)
The number of variational parameters in the [Jastrow](#) factor.
- int [Nparams](#)
The total number of variational parameters.
- std::vector< [OutputHandler](#) * > [output_handler](#)
Either contains a [stdoutASGD](#) object or not.
- std::vector< [ErrorEstimator](#) * > [error_estimators](#)
One [ErrorEstimator](#) object pr. variational parameter.

3.21.1 Detailed Description

Class for minimization methods used to obtain optimal variational parameters.

3.21.2 Constructor & Destructor Documentation

3.21.2.1 `Minimizer::Minimizer (VMC * vmc, const ParParams & pp, const arma::rowvec & alpha, const arma::rowvec & beta)`

Constructor.

Parameters

<i>vmc</i>	The VMC object used for storing variational parameters and calculating stochastic gradients.
<i>alpha</i>	Vector of initial conditions of spatial variational parameters
<i>beta</i>	Vector of initial conditions of Jastrow variational parameters

3.21.3 Member Function Documentation

3.21.3.1 `void Minimizer::dump_output ()` [`protected`]

Iterates over the output objects in the `output_handler` vector. No if-tests.

3.21.3.2 `void Minimizer::error_output ()` [`protected`]

Estimates and finalizes the [ErrorEstimator](#) objects initialized in the `error_estimators` vector.

3.21.3.3 `void Minimizer::finalize_output ()` [`protected`]

Calls the `finalize` function for the object in the `output_handler` vector.

3.21.3.4 `virtual void Minimizer::update_parameters ()` [`protected`, `pure virtual`]

Method for updating the variational parameters based on the previous step.

Needs to be implemented by a subclass.

Implemented in [ASGD](#).

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

- `src/Minimizer/Minimizer.h`
- `src/Minimizer/Minimizer.cpp`

3.22 MinimizerParams Struct Reference

Struct used to initialize Minimization parameters.

Public Attributes

- double **max_step**
- double **f_max**
- double **f_min**
- double **omega**
- double **A**
- double **a**
- int **SGDsamples**
- int **n_w**
- int **therm**
- int **n_c**
- int **n_c_SGD**
- arma::rowvec [alpha](#)
Initial condition for the spatial variational parameter(s).
- arma::rowvec [beta](#)
Initial condition for the [Jastrow](#) variational parameter(s).

3.22.1 Detailed Description

Struct used to initialize Minimization parameters.

See also

[ASGD](#)

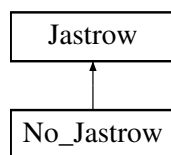
The documentation for this struct was generated from the following file:

- src/QMChheaders.h

3.23 No_Jastrow Class Reference

Class loaded when no correlation factor is used.

Inheritance diagram for No_Jastrow:



Public Member Functions

- void [get_grad](#) ([Walker](#) *walker) const
- void [get_grad](#) (const [Walker](#) *walker_pre, [Walker](#) *walker_post, int i) const
Updates the gradient for a new particle move.
- void [initialize](#) ()
- void [get_dJ_matrix](#) ([Walker](#) *walker, int i) const
Updates the summation factors of [Jastrow](#) factors closed form expressions.
- double [get_j_ratio](#) (const [Walker](#) *walker_post, const [Walker](#) *walker_pre, int i) const
Calculates the ratio of the [Jastrow](#) factor needed by metropolis.
- double [get_val](#) (const [Walker](#) *walker) const
- double [get_lapl_sum](#) ([Walker](#) *walker) const
Method for calculating the Laplacian.

Protected Member Functions

- double [get_parameter](#) (int n)
Returns variational parameters.
- void [set_parameter](#) (double param, int n)
Sets variational parameters.
- double [get_variational_derivative](#) (const [Walker](#) *walker, int n)
Calculates the derivative of the [Jastrow](#) factor with respect to a variational parameter.

3.23.1 Detailed Description

Class loaded when no correlation factor is used.

3.23.2 Member Function Documentation

3.23.2.1 void No_Jastrow::get_dJ_matrix (Walker * walker, int i) const [inline, virtual]

Updates the summation factors of [Jastrow](#) factors closed form expressions.

Used to optimize the calculations as few of these terms change as we move a particle.

Parameters

<i>i</i>	Particle number.
----------	------------------

Implements [Jastrow](#).

3.23.2.2 `void No_Jastrow::get_grad (Walker * walker) const` [inline, virtual]

Calculates the entire Cartesian gradient.

Implements [Jastrow](#).

3.23.2.3 `void No_Jastrow::get_grad (const Walker * walker_pre, Walker * walker_post, int i) const` [inline, virtual]

Updates the gradient for a new particle move.

Parameters

<i>walker_post</i>	Walker at current time step
<i>walker_pre</i>	Walker at previous time step
<i>i</i>	Particle number.

Implements [Jastrow](#).

3.23.2.4 `double No_Jastrow::get_j_ratio (const Walker * walker_new, const Walker * walker_old, int i) const` [inline, virtual]

Calculates the ratio of the [Jastrow](#) factor needed by metropolis.

Parameters

<i>walker_new</i>	Walker at current time step
<i>walker_old</i>	Walker at previous time step
<i>i</i>	The particle number.

Implements [Jastrow](#).

3.23.2.5 `double No_Jastrow::get_lapl_sum (Walker * walker) const` [inline, virtual]

Method for calculating the Laplacian.

Calculates the sum of all particles Laplacians.

Implements [Jastrow](#).

3.23.2.6 `double No_Jastrow::get_parameter (int n)` [inline, protected, virtual]

Returns variational parameters.

Parameters

<i>n</i>	The index of the sought variational parameter
----------	---

Returns

Variational parameter with index [n]

Implements [Jastrow](#).

3.23.2.7 `double No_Jastrow::get_val (const Walker * walker) const` [inline, virtual]

Calculates the value of the [Jastrow](#) Factor at the walker's position.

Implements [Jastrow](#).

3.23.2.8 `double No_Jastrow::get_variational_derivative (const Walker * walker, int n)` [inline, protected, virtual]

Calculates the derivative of the [Jastrow](#) factor with respect to a variational parameter.

Parameters

<i>n</i>	The index of the variational parameter for which the derivative is to be taken
<i>walker</i>	The walker holds the positions etc. needed to evaluate the derivative

Reimplemented from [Jastrow](#).

3.23.2.9 `void No_Jastrow::initialize ()` [inline, virtual]

Initializes the non-variational parameters needed by the [Jastrow](#) Factor.

Implements [Jastrow](#).

3.23.2.10 `void No_Jastrow::set_parameter (double param, int n)` [inline, protected, virtual]

Sets variational parameters.

Parameters

<i>n</i>	The index of the sought variational parameter
<i>param</i>	The new value of parameter [n]

Implements [Jastrow](#).

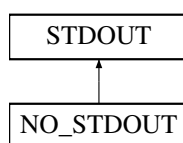
The documentation for this class was generated from the following files:

- src/Jastrow/No_Jastrow/No_Jastrow.h
- src/Jastrow/No_Jastrow/No_Jastrow.cpp

3.24 NO_STDOUT Class Reference

Class for suppressing standard output. Every node but the master has this. If-tests around cout is avoided.

Inheritance diagram for NO_STDOUT:



Public Member Functions

- virtual void **cout** (std::stringstream &a)

3.24.1 Detailed Description

Class for suppressing standard output. Every node but the master has this. If-tests around cout is avoided.

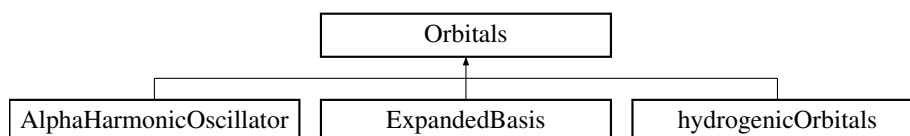
The documentation for this class was generated from the following file:

- src/QMHeaders.h

3.25 Orbitals Class Reference

Superclass for the single particle orbital classes. Handles everything specific regarding choice of single particle basis.

Inheritance diagram for Orbitals:



Public Member Functions

- **Orbitals** (int n_p, int dim)
- virtual void **set_qnum_indie_terms** (const [Walker](#) *walker, int i)
Calculates single particle wave function terms which are independent of the quantum numbers.
- virtual double **phi** (const [Walker](#) *walker, int particle, int q_num)
Calculates the single particle wave function for a given walker's particle.
- virtual double **del_phi** (const [Walker](#) *walker, int particle, int q_num, int d)
Calculates the single particle wave function derivative for a given walker's particle and dimension.
- virtual double **lapl_phi** (const [Walker](#) *walker, int particle, int q_num)
Calculates the single particle wave function for a given walker's particle.
- void **set_qmc_ptr** ([QMC](#) *qmc)

Protected Member Functions

- virtual double **get_parameter** (int n)=0
A method for retrieving variational parameters.
- virtual void **set_parameter** (double parameter, int n)=0
A method for setting variational parameters.
- virtual double **get_variational_derivative** (const [Walker](#) *walker, int n)
A method for calculating the variational derivative.
- double **num_diff** (const [Walker](#) *walker, int particle, int q_num, int d)
Method for calculating the single particle derivative using a finite difference scheme.
- double **num_ddiff** (const [Walker](#) *walker, int particle, int q_num)
Method for calculating the single particle Laplacian using a finite difference scheme.
- void **testLaplace** (const [Walker](#) *walker, int particle, int q_num)
Method for validating closed form expressions for laplacians by comparing them to numerical calculations.
- void **testDell** (const [Walker](#) *walker, int particle, int q_num, int d)
Method for validating closed form expressions for derivatives by comparing them to numerical calculations.

Protected Attributes

- int **n_p**
- int **n2**
- int **dim**
- int **max_implemented**
The maximum number basis size supported for any system ##RYDD OPP.
- [QMC](#) * **qmc**
A pointer to the [QMC](#) solver object. Needed for numerical variational derivatives.
- double **h**

The step length for finite difference schemes.

- double **h2**
- double **two_h**
- [BasisFunctions](#) ** [basis_functions](#)

A vector mapping a quantum number index to a single particle wave function.

- [BasisFunctions](#) *** [dell_basis_functions](#)

A maxtrix mapping a quantum number- and dimension index to a single particle wave function derivative.

- [BasisFunctions](#) ** [lapl_basis_functions](#)

A vector mapping a quantum number index to a single particle wave function Laplacian.

Friends

- class **Minimizer**
- class **ASGD**
- class **stdoutASGD**

3.25.1 Detailed Description

Superclass for the single particle orbital classes. Handles everything specific regarding choice of single particle basis.

3.25.2 Member Function Documentation

3.25.2.1 `double Orbitals::del_phi (const Walker * walker, int particle, int q_num, int d)`
[virtual]

Calculates the single particle wave function derivative for a given walker's particle and dimension.

Parameters

<i>q_num</i>	The quantum number index.
<i>d</i>	The dimension for which the derivative should be calculated (x,y,z).

Reimplemented in [ExpandedBasis](#).

3.25.2.2 `virtual double Orbitals::get_parameter (int n)` [protected, pure virtual]

A method for retrieving variational parameters.

Parameters

<i>n</i>	Index of the sought variational parameter.
----------	--

Implemented in [AlphaHarmonicOscillator](#), and [hydrogenicOrbitals](#).

3.25.2.3 `double Orbitals::get_variational_derivative (const Walker * walker, int n)`
`[protected, virtual]`

A method for calculating the variational derivative.

By default uses a finite difference scheme. Can be overridden to evaluate a closed form expression.

Parameters

<i>n</i>	Index of the sought variational parameter.
----------	--

Reimplemented in [AlphaHarmonicOscillator](#), and [hydrogenicOrbitals](#).

3.25.2.4 `double Orbitals::lapl_phi (const Walker * walker, int particle, int q_num)`
`[virtual]`

Calculates the single particle wave function for a given walker's particle.

Parameters

<i>q_num</i>	The quantum number index.
--------------	---------------------------

Reimplemented in [ExpandedBasis](#).

3.25.2.5 `double Orbitals::num_ddiff (const Walker * walker, int particle, int q_num)`
`[protected]`

Method for calculating the single particle Laplacian using a finite difference scheme.

Method [lapl_phi\(\)](#) can be overridden to use this method in case no closed form expressions are implemented.

Parameters

<i>q_num</i>	The quantum number index.
--------------	---------------------------

3.25.2.6 `double Orbitals::num_diff (const Walker * walker, int particle, int q_num, int d)`
`[protected]`

Method for calculating the single particle derivative using a finite difference scheme.

Method [del_phi\(\)](#) can be overridden to use this method in case no closed form expressions are implemented.

Parameters

<i>d</i>	The dimension for which the derivative should be calculated (x,y,z).
----------	--

3.25.2.7 `double Orbitals::phi (const Walker * walker, int particle, int q_num)`
`[virtual]`

Calculates the single particle wave function for a given walker's particle.

Parameters

<i>q_num</i>	The quantum number index.
--------------	---------------------------

Reimplemented in [ExpandedBasis](#).

3.25.2.8 `virtual void Orbitals::set_parameter (double parameter, int n)`
`[protected, pure virtual]`

A method for setting variational parameters.

Parameters

<i>n</i>	Index of the sought variational parameter.
<i>parameter</i>	The new value of the variational parameter.

Implemented in [AlphaHarmonicOscillator](#), and [hydrogenicOrbitals](#).

3.25.2.9 `virtual void Orbitals::set_qnum_indep_terms (const Walker * walker, int i)`
`[inline, virtual]`

Calculates single particle wave function terms which are independent of the quantum numbers.

If a term in the single particle functions are independent of the quantum number, this function can be overridden to calculate them beforehand (for each particle), and rather extract the value instead of recalculating.

Parameters

<i>i</i>	Particle number.
----------	------------------

Reimplemented in [AlphaHarmonicOscillator](#), and [hydrogenicOrbitals](#).

3.25.2.10 `void Orbitals::testDell (const Walker * walker, int particle, int q_num, int d)`
`[protected]`

Method for validating closed form expressions for derivatives by comparing them to numerical calculations.

Parameters

<i>q_num</i>	The quantum number index.
<i>d</i>	The dimension for which the derivative should be calculated (x,y,z).

3.25.2.11 void Orbitals::testLaplace (const Walker * walker, int particle, int q_num) [protected]

Method for validating closed form expressions for laplacians by comparing them to numerical calculations.

Parameters

<i>q_num</i>	The quantum number index.
--------------	---------------------------

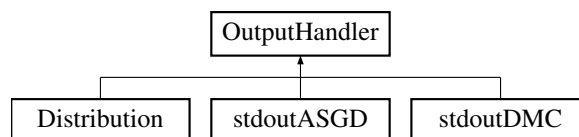
The documentation for this class was generated from the following files:

- src/Orbitals/Orbitals.h
- src/Orbitals/Orbitals.cpp

3.26 OutputHandler Class Reference

Class for handling output-methods. Designed to avoid rewriting code, as well as avoid if-tests if output is not desired.

Inheritance diagram for OutputHandler:



Public Member Functions

- [OutputHandler](#) (std::string filename, std::string path, bool parallel, int node, int n_nodes)
Constructor.
- virtual void [dump](#) ()=0
Methods for updating the output.
- virtual void [finalize](#) ()
Finalizes the output.
- void [set_qmc_ptr](#) (QMC *qmc)
- void [set_min_ptr](#) (Minimizer *min)

Protected Member Functions

- void [init_file](#) ()
- virtual void [post_pointer_init](#) ()

Method for initialization requires once the correct QMC/Min pointer type is set.

Protected Attributes

- bool [is_vmc](#)
Switch used to typecast the [QMC](#) object to a [VMC](#) object.
- bool [is_dmc](#)
Switch used to typecast the [QMC](#) object to a [DMC](#) object.
- bool [is_ASGD](#)
Switch used to typecast the [Min](#) object to an [ASGD](#) object.
- bool **parallel**
- int **node**
- int **n_nodes**
- bool [use_file](#)
If [init_file\(\)](#) is called, this flag is set true. Assures correct finalization.
- std::stringstream **s**
- std::string **filename**
- std::string **path**
- std::ofstream **file**
- [QMC](#) * **qmc**
- [DMC](#) * **dmc**
- [VMC](#) * **vmc**
- [Minimizer](#) * **min**
- [ASGD](#) * **asgd**

3.26.1 Detailed Description

Class for handling output-methods. Designed to avoid rewriting code, as well as avoid if-tests if output is not desired.

See also

[QMC::output_handler](#), [Minimizer::output_handler](#)

3.26.2 Constructor & Destructor Documentation

- 3.26.2.1 [OutputHandler::OutputHandler](#) ([std::string filename](#), [std::string path](#), [bool parallel](#), [int node](#), [int n_nodes](#))

Constructor.

Parameters

<i>filename</i>	The name of the output file.
<i>path</i>	The path of the output.

3.26.3 Member Function Documentation

3.26.3.1 virtual void **OutputHandler::dump** () [pure virtual]

Methods for updating the output.

Typically retrieves information through the solver pointers (given correct accessibility levels/friend)

Implemented in [Distribution](#), [stdoutDMC](#), and [stdoutASGD](#).

3.26.3.2 void **OutputHandler::finalize** () [virtual]

Finalizes the output.

Closes file if use_file flag is true. Can be overridden if more complex tasks needs to be done, such as calculating histograms etc.

See also

[Distribution::finalize\(\)](#)

Reimplemented in [Distribution](#).

3.26.3.3 void **OutputHandler::init_file** () [protected]

Opens a file with filename at path supplied in constructor. Subclass implementations can call this function. Superclass does not.

3.26.3.4 virtual void **OutputHandler::post_pointer_init** () [inline, protected, virtual]

Method for initialization requires once the correct QMC/Min pointer type is set.

Defaults to nothing.

Reimplemented in [stdoutASGD](#).

3.26.3.5 void **OutputHandler::set_min_ptr** (**Minimizer** * *min*)

Sets the Min pointer and typecasts it according to the minimizer flags.

3.26.3.6 void OutputHandler::set_qmc_ptr (QMC * qmc)

Sets the [QMC](#) pointer and typecasts it according to the solver flags.

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

- src/OutputHandler/OutputHandler.h
- src/OutputHandler/OutputHandler.cpp

3.27 OutputParams Struct Reference

Struct used to initialize output parameters.

Public Attributes

- bool [dist_out](#)
If true, distributions are calculated for VMC/DMC.
- bool [dmc_out](#)
If true, [DMC](#) outputs data to file each cycle.
- bool [ASGD_out](#)
If true, [ASGD](#) outputs data to file each cycle.

3.27.1 Detailed Description

Struct used to initialize output parameters.

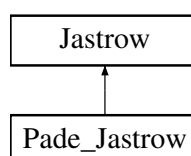
The documentation for this struct was generated from the following file:

- src/QMChaders.h

3.28 Pade_Jastrow Class Reference

The Pade [Jastrow](#) factor with a single variational parameter.

Inheritance diagram for Pade_Jastrow:



Public Member Functions

- **Pade_Jastrow** ([GeneralParams](#) &, [VariationalParams](#) &)
- void [initialize](#) ()
- void [get_grad](#) ([Walker](#) *walker) const
- void [get_grad](#) (const [Walker](#) *walker_pre, [Walker](#) *walker_post, int i) const
Updates the gradient for a new particle move.
- void [get_dJ_matrix](#) ([Walker](#) *walker, int i) const
Updates the summation factors of [Jastrow](#) factors closed form expressions.
- double [get_j_ratio](#) (const [Walker](#) *walker_new, const [Walker](#) *walker_old, int i) const
Calculates the ratio of the [Jastrow](#) factor needed by metropolis.
- double [get_val](#) (const [Walker](#) *walker) const
- double [get_lapl_sum](#) ([Walker](#) *walker) const
Method for calculating the Laplacian.

Protected Member Functions

- double [get_variational_derivative](#) (const [Walker](#) *walker, int n)
Calculates the derivative of the [Jastrow](#) factor with respect to a variational parameter.
- void [set_parameter](#) (double param, int n)
Sets variational parameters.
- double [get_parameter](#) (int n)
Returns variational parameters.

Protected Attributes

- double [beta](#)
The variational parameter.
- arma::mat [a](#)
The spin-dependent variables taking care of the cusp condition.

3.28.1 Detailed Description

The Pade [Jastrow](#) factor with a single variational parameter.

3.28.2 Member Function Documentation

3.28.2.1 void Pade_Jastrow::get_dJ_matrix ([Walker](#) * walker, int i) const [virtual]

Updates the summation factors of [Jastrow](#) factors closed form expressions.

Used to optimize the calculations as few of these terms change as we move a particle.

Parameters

<i>i</i>	Particle number.
----------	------------------

Implements [Jastrow](#).

3.28.2.2 `void Pade_Jastrow::get_grad (Walker * walker) const` [virtual]

Calculates the entire Cartesian gradient.

Implements [Jastrow](#).

3.28.2.3 `void Pade_Jastrow::get_grad (const Walker * walker_pre, Walker * walker_post, int i) const` [virtual]

Updates the gradient for a new particle move.

Parameters

<i>walker_post</i>	Walker at current time step
<i>walker_pre</i>	Walker at previous time step
<i>i</i>	Particle number.

Implements [Jastrow](#).

3.28.2.4 `double Pade_Jastrow::get_j_ratio (const Walker * walker_new, const Walker * walker_old, int i) const` [virtual]

Calculates the ratio of the [Jastrow](#) factor needed by metropolis.

Parameters

<i>walker_new</i>	Walker at current time step
<i>walker_old</i>	Walker at previous time step
<i>i</i>	The particle number.

Implements [Jastrow](#).

3.28.2.5 `double Pade_Jastrow::get_lapl_sum (Walker * walker) const` [virtual]

Method for calculating the Laplacian.

Calculates the sum of all particles Laplacians.

Implements [Jastrow](#).

3.28.2.6 `double Pade_Jastrow::get_parameter (int n)` [inline, protected, virtual]

Returns variational parameters.

Parameters

<i>n</i>	The index of the sought variational parameter
----------	---

Returns

Variational parameter with index [*n*]

Implements [Jastrow](#).

3.28.2.7 `double Pade_Jastrow::get_val (const Walker * walker) const` [virtual]

Calculates the value of the [Jastrow](#) Factor at the walker's position.

Implements [Jastrow](#).

3.28.2.8 `double Pade_Jastrow::get_variational_derivative (const Walker * walker, int n)` [protected, virtual]

Calculates the derivative of the [Jastrow](#) factor with respect to a variational parameter.

Parameters

<i>n</i>	The index of the variational parameter for which the derivative is to be taken
<i>walker</i>	The walker holds the positions etc. needed to evaluate the derivative

Reimplemented from [Jastrow](#).

3.28.2.9 `void Pade_Jastrow::initialize ()` [virtual]

In case of Pade [Jastrow](#), initializing means setting up the a matrix.

Implements [Jastrow](#).

3.28.2.10 `void Pade_Jastrow::set_parameter (double param, int n)` [inline, protected, virtual]

Sets variational parameters.

Parameters

<i>n</i>	The index of the sought variational parameter
<i>param</i>	The new value of parameter [n]

Implements [Jastrow](#).

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

- src/Jastrow/Pade_Jastrow/Pade_Jastrow.h
- src/Jastrow/Pade_Jastrow/Pade_Jastrow.cpp

3.29 ParParams Struct Reference

Struct used to initialize parallelization parameters.

Public Attributes

- bool [is_master](#)
True for the master node.
- bool [parallel](#)
True if $n_nodes > 1$.
- int [node](#)
The process' rank.
- int [n_nodes](#)
The total number of processes.

3.29.1 Detailed Description

Struct used to initialize parallelization parameters.

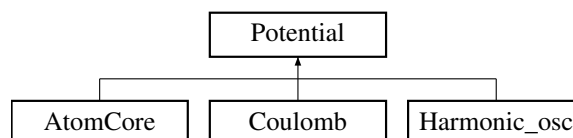
The documentation for this struct was generated from the following file:

- src/QMChheaders.h

3.30 Potential Class Reference

Superclass for potentials. Potentials are stores in a vector in the system object.

Inheritance diagram for Potential:



Public Member Functions

- **Potential** (int n_p, int dim)
- virtual double [get_pot_E](#) (const [Walker](#) *walker) const =0
Method for calculating a walker's potential energy.

Protected Attributes

- int **n_p**
- int **dim**

3.30.1 Detailed Description

Superclass for potentials. Potentials are stores in a vector in the system object.

See also

[System::potentials](#), [System::get_potential_energy\(\)](#)

3.30.2 Member Function Documentation

3.30.2.1 virtual double **Potential::get_pot_E** (const [Walker](#) * *walker*) const [pure virtual]

Method for calculating a walker's potential energy.

Method overridden by subclasses.

Parameters

<i>walker</i>	The walker for which the potential energy should be calculated.
---------------	---

Returns

The potential energy.

Implemented in [Harmonic_osc](#), [AtomCore](#), and [Coulomb](#).

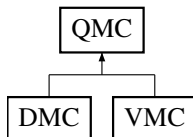
The documentation for this class was generated from the following files:

- src/Potential/Potential.h
- src/Potential/Potential.cpp

3.31 QMC Class Reference

The [QMC](#) superclass. Holds implementations of general functions for both [VMC](#) and [DMC](#) in order to avoid rewriting code and emphasize the similarities.

Inheritance diagram for QMC:



Public Member Functions

- [QMC](#) ([GeneralParams](#) &, int [n_c](#), [SystemObjects](#) &, [ParParams](#) &, int [n_w](#), int [K=1](#))
Constructor.
- virtual void [run_method](#) ()=0
Method used for executing the solver.
- void [get_QF](#) ([Walker](#) *walker) const
Method for calculating the Quantum Force.
- void [get_gradients](#) (const [Walker](#) *walker_pre, [Walker](#) *walker_post, int particle) const
Method for calculating the gradients after moving a particle.
- void [get_gradients](#) ([Walker](#) *walker) const
Method for calculating the full gradients.
- void [get_lapsum](#) ([Walker](#) *walker) const
Method for calculating the Laplacian of all walkers.
- double [get_wf_value](#) (const [Walker](#) *walker) const
Method for calculating the wave functions value at a given walker's position.
- double [calculate_local_energy](#) (const [Walker](#) *walker) const
Method for calculating the local energy.
- void [get_accepted_ratio](#) ()
Method for calculating the acceptance ratio.
- void [add_output](#) ([OutputHandler](#) *output_handler)
Method used for loading the output_handler with objects.
- void [set_error_estimator](#) ([ErrorEstimator](#) *error_estimator)
Method for setting the error estimator.
- virtual void [output](#) ()=0
Method for standard output.
- [System](#) * [get_system_ptr](#) () const
- [Sampling](#) * [get_sampling_ptr](#) () const
- [Jastrow](#) * [get_jastrow_ptr](#) () const
- [Orbitals](#) * [get_orbitals_ptr](#) () const

Protected Member Functions

- virtual void `set_trial_positions` ()=0
Method for setting the trial position of the QMC method's walkers.
- void `diffuse_walker` (`Walker` *original, `Walker` *trial)
Method for diffusing a walker one time step.
- double `get_acceptance_ratio` (const `Walker` *walker_pre, const `Walker` *walker_post, int particle) const
Method for calculating the acceptance ratio used in the Metropolis test.
- virtual bool `move_authorized` (double A)=0
Method for deciding whether or not to accept a move.
- bool `metropolis_test` (double A)
Method for performing the metropolis test after when diffusing a walker.
- void `update_walker` (`Walker` *walker_pre, const `Walker` *walker_post, int particle) const
Method for updating the walker after an accepted step.
- void `reset_walker` (const `Walker` *walker_pre, `Walker` *walker_post, int particle) const
Method for resetting the walker after a rejected step.
- void `copy_walker` (const `Walker` *parent, `Walker` *child) const
Method for (hard) copying a walker object.
- void `calculate_energy_necessities` (`Walker` *walker) const
Method for calculating the necessary quantities needed in order to calculate the local energy.
- double `get_KE` (const `Walker` *walker) const
Method for storing positional data.
- virtual void `save_distribution` ()=0
- virtual void `node_comm` ()=0
Method for performing node communication.
- void `dump_output` ()
- void `finalize_output` ()
- void `estimate_error` () const
- void `set_spin_state` (int particle) const
- void `test_ratios` (const `Walker` *walker_pre, const `Walker` *walker_post, int particle, double R_qmc) const
Method used for testing the optimized ratio calculation.
- void `test_gradients` (`Walker` *walker)
Method for testing the optimized gradients calculation.

Protected Attributes

- `STDOUT` * `std_out`
Output object. Wraps and replaces std::cout.
- `std::stringstream` `s`

- `std::string` [runpath](#)
The directory which the simulation is set to run.
- `std::string` [dist_path](#)
The path where the distribution are saved.
- `arma::mat` [dist](#)
Matrix holding positional data for the distribution.
- `int` [last_inserted](#)
Index of last inserted positional data.
- `int` [dist_tresh](#)
Amount of cycles to skip in between storing position data.
- `bool` **is_master**
- `bool` **parallel**
- `int` **node**
- `int` **n_nodes**
- `int` [n_c](#)
The number of Monte-Carlo cycles.
- `int` [thermalization](#)
The number of thermalization steps.
- `int` [cycle](#)
- `int` [n_w](#)
The number of walkers.
- `int` **n_p**
- `int` **n2**
- `int` **dim**
- `unsigned long int` [accepted](#)
Number of accepted moves.
- `unsigned long int` [total_samples](#)
Total number of moves.
- `double` [local_E](#)
The last calculated local energy.
- `Walker *` [trial_walker](#)
The trial walker used to test a move.
- `Walker **` [original_walkers](#)
*A list of `n_w` walkers used in *DMC*.*
- `Jastrow *` [jastrow](#)
*The *Jastrow* object.*
- `Sampling *` [sampling](#)
*The *Sampling* object.*
- `System *` [system](#)
The system object.
- `ErrorEstimator *` [error_estimator](#)
The error estimator.
- `std::vector<` [OutputHandler](#) `>` [output_handler](#)
*Can hold *stdoutDMC* (in case of *DMC*), *Distribution*, both or none.*

Friends

- class **Distribution**

3.31.1 Detailed Description

The [QMC](#) superclass. Holds implementations of general functions for both [VMC](#) and [DMC](#) in order to avoid rewriting code and emphasize the similarities.

3.31.2 Constructor & Destructor Documentation

3.31.2.1 **QMC::QMC** ([GeneralParams](#) & *gP*, int *n_c*, [SystemObjects](#) & *sO*, [ParParams](#) & *pp*, int *n_w*, int *K* = 1)

Constructor.

K K times *n_w* walkers are initialized. K != 0 only sensible in [DMC](#).

3.31.3 Member Function Documentation

3.31.3.1 **void QMC::calculate_energy_necessities** ([Walker](#) * *walker*) const [protected]

Method for calculating the necessary quantities needed in order to calculate the local energy.

See also

[Sampling::calculate_energy_necessities\(\)](#)

3.31.3.2 **double QMC::calculate_local_energy** (const [Walker](#) * *walker*) const

Method for calculating the local energy.

See also

[get_KE\(\)](#), [System::get_potential_energy\(\)](#)

3.31.3.3 **void QMC::copy_walker** (const [Walker](#) * *parent*, [Walker](#) * *child*) const [protected]

Method for (hard) copying a walker object.

Parameters

<i>parent, child</i>	The parent is copied to the child.
----------------------	------------------------------------

3.31.3.4 `void QMC::diffuse_walker (Walker * original, Walker * trial)`
`[protected]`

Method for diffusing a walker one time step.

The trial walker must equal the original walker in input. The original walker is updated on output.

3.31.3.5 `void QMC::dump_output ()` `[protected]`

Iterates over the output objects in the output_handler vector. No if-tests.

3.31.3.6 `void QMC::estimate_error () const` `[protected]`

Estimates and finalizes the [ErrorEstimator](#) object initialized in the error_estimator vector.

3.31.3.7 `void QMC::finalize_output ()` `[protected]`

Calls the finalize function for the object in the output_handler vector.

3.31.3.8 `void QMC::get_gradients (const Walker * walker_pre, Walker * walker_post,
int particle) const`

Method for calculating the gradients after moving a particle.

See also

[Jastrow::get_grad\(\)](#), [System::get_spatial_grad\(\)](#)

3.31.3.9 `void QMC::get_gradients (Walker * walker) const`

Method for calculating the full gradients.

See also

[Jastrow::get_grad\(\)](#), [System::get_spatial_grad\(\)](#)

3.31.3.10 `double QMC::get_KE (const Walker * walker) const` `[protected]`

Method for storing positional data.

Stored in the dist matrix. Used by OutputHandler::Distribution.

See also

[Distribution::dump\(\)](#) Method for calculating the kinetic energy of a walker.

3.31.3.11 `void QMC::get_lapsum (Walker * walker) const` `[inline]`

Method for calculating the Laplacian of all walkers.

See also

[System::get_spatial_lapl_sum\(\)](#), [Jastrow::get_lapl_sum\(\)](#)

3.31.3.12 `double QMC::get_wf_value (const Walker * walker) const` `[inline]`

Method for calculating the wave functions value at a given walker's position.

See also

[System::get_spatial_wf\(\)](#), [Jastrow::get_val\(\)](#)

3.31.3.13 `bool QMC::metropolis_test (double A)` `[protected]`

Method for performing the metropolis test after when diffusing a walker.

Parameters

<i>A</i>	The acceptance ratio calculated by get_acceptance_ratio() .
----------	---

3.31.3.14 `virtual bool QMC::move_authorized (double A)` `[protected, pure virtual]`

Method for deciding whether or not to accept a move.

Wraps the metropolis sampling with possibilities of overriding.

See also

[System::allow_transition\(\)](#)

Implemented in [DMC](#), and [VMC](#).

3.31.3.15 `void QMC::reset_walker (const Walker * walker_pre, Walker * walker_post, int particle) const` `[protected]`

Method for resetting the walker after a rejected step.

Given a particle number, the method only resets the changed values.

Parameters

<i>walker_post</i>	Walker at current time step
<i>walker_pre</i>	Walker at previous time step

3.31.3.16 void **QMC::set_spin_state** (int *particle*) const [protected]

Since the spatial wave function is split, certain values are unchanged if the moved particle has opposite spin. Assuming a two-level system, the first half of the particles are assumed to have one spin value, and the second half the other.

This method sets the start and end position of the block that needs to be changed.

Parameters

<i>start,end</i>	See the <code>System::Start</code> <code>System::End</code>
------------------	---

3.31.3.17 virtual void **QMC::set_trial_positions** () [protected, pure virtual]

Method for setting the trial position of the [QMC](#) method's walkers.

See also

[Sampling::set_trial_pos\(\)](#)

Implemented in [DMC](#), and [VMC](#).

3.31.3.18 void **QMC::test_gradients** (Walker * *walker*) [protected]

Method for testing the optimized gradients calculation.

Compares with finite difference calculation.

3.31.3.19 void **QMC::test_ratios** (const Walker * *walker_pre*, const Walker * *walker_post*, int *particle*, double *R_qmc*) const [protected]

Method used for testing the optimized ratio calculation.

Compares to brute force computation of the wave function values. *R_qmc* The optimized trial wave function ratio (spatial and [Jastrow](#)).

3.31.3.20 void **QMC::update_walker** (Walker * *walker_pre*, const Walker * *walker_post*, int *particle*) const [protected]

Method for updating the walker after an accepted step.

Given a particle number, the method only updates the changed values.

Parameters

<i>walker_post</i>	Walker at current time step
<i>walker_pre</i>	Walker at previous time step

3.31.4 Member Data Documentation

3.31.4.1 `int QMC::cycle` [protected]

The current Monte-Carlo cycle.

3.31.4.2 `int QMC::n_w` [protected]

The number of walkers.

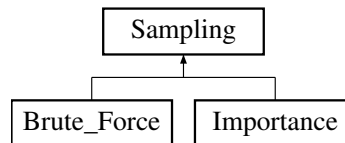
[VMC](#) stores this many cycles in case of [DMC](#)

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

- `src/QMC/QMC.h`
- `src/QMC/QMC.cpp`

3.32 Sampling Class Reference

Inheritance diagram for Sampling:



Public Member Functions

- **Sampling** (int n_p, int dim)
- void [update_pos](#) (const [Walker](#) *walker_pre, [Walker](#) *walker_post, int particle) const
Method for updating the position of a walker's particle.
- virtual void [update_necessities](#) (const [Walker](#) *walker_pre, [Walker](#) *walker_post, int particle) const =0
Method for updating the sampling specific necessary values.
- virtual void [update_walker](#) ([Walker](#) *walker_pre, const [Walker](#) *walker_post, int particle) const =0
- virtual void [reset_walker](#) (const [Walker](#) *walker_pre, [Walker](#) *walker_post, int particle) const =0
- void [set_trial_pos](#) ([Walker](#) *walker, bool set_pos=true)
Method for setting the trial position for a given walker.
- void [set_trial_states](#) ([Walker](#) *walker)
Method for setting up the single particle orbitals and it's derivatives for a given walker.
- virtual void [get_necessities](#) ([Walker](#) *walker)=0

Method for calculating the sampling specific necessary values.

- virtual void [calculate_energy_necessities](#) ([Walker](#) *walker) const =0

Method for calculating the sampling specific necessary values in order to compute the local energy.

- virtual void [copy_walker](#) (const [Walker](#) *parent, [Walker](#) *child) const =0

Method for copying the sampling specific parts of a walker.

- virtual double [get_g_ratio](#) (const [Walker](#) *walker_post, const [Walker](#) *walker_pre) const

Method for calculating the diffusion Green's function ratios.

- double [get_branching_Gfunc](#) (double E_x, double E_y, double E_T) const
- double [get_spatialjast_ratio](#) (const [Walker](#) *walker_post, const [Walker](#) *walker_pre, int particle) const
- void [set_qmc_ptr](#) ([QMC](#) *qmc)
- void [set_dt](#) (double dt)
- double [get_dt](#) () const
- double [get_std](#) () const
- double [call_RNG](#) ()

Calls a uniform random number generator.

- void [set_spin_state](#) (int start, int end)

Protected Attributes

- int [n_p](#)
- int [n2](#)
- int [dim](#)
- int [start](#)
- int [end](#)
- [Diffusion](#) * [diffusion](#)

The [Diffusion](#) object.

- [QMC](#) * [qmc](#)

The [QMC](#) main solver object. Needed to access e.g. the system object.

3.32.1 Member Function Documentation

3.32.1.1 double [Sampling::call_RNG](#) () [\[inline\]](#)

Calls a uniform random number generator.

Returns a random uniform number on [0,1).

3.32.1.2 virtual void [Sampling::copy_walker](#) (const [Walker](#) * [parent](#), [Walker](#) * [child](#)) const [\[pure virtual\]](#)

Method for copying the sampling specific parts of a walker.

See also

[QMC::copy_walker\(\)](#)

Implemented in [Importance](#), and [Brute_Force](#).

3.32.1.3 virtual double **Sampling::get_g_ratio** (const Walker * *walker_post*, const Walker * *walker_pre*) const [inline, virtual]

Method for calculating the diffusion Green's function ratios.

See the [Diffusion](#) class for documentation.

3.32.1.4 virtual void **Sampling::get_necessities** (Walker * *walker*) [pure virtual]

Method for calculating the sampling specific necessary values.

Called after a trial position is set.

Implemented in [Importance](#), and [Brute_Force](#).

3.32.1.5 virtual void **Sampling::reset_walker** (const Walker * *walker_pre*, Walker * *walker_post*, int *particle*) const [pure virtual]

See also

[QMC::reset_walker\(\)](#)

Implemented in [Brute_Force](#), and [Importance](#).

3.32.1.6 void **Sampling::set_spin_state** (int *start*, int *end*) [inline]

See also

[QMC::set_spin_state\(\)](#)

3.32.1.7 void **Sampling::update_pos** (const Walker * *walker_pre*, Walker * *walker_post*, int *particle*) const

Method for updating the position of a walker's particle.

Sets a new position according to the diffusion rules, and calls all the functions necessary to get all the values updates, e.g. `System::calc_for_new_pos()`

3.32.1.8 virtual void **Sampling::update_walker** (Walker * *walker_pre*, const Walker * *walker_post*, int *particle*) const [pure virtual]

See also

[QMC::update_walker\(\)](#)

Implemented in [Importance](#), and [Brute_Force](#).

3.32.2 Member Data Documentation

3.32.2.1 `Diffusion* Sampling::diffusion` [protected]

The [Diffusion](#) object.

See also

[Diffusion](#)

3.32.2.2 `int Sampling::end` [protected]

See also

[System::end](#)

3.32.2.3 `int Sampling::start` [protected]

See also

[System::start](#)

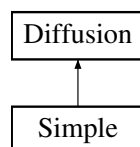
The documentation for this class was generated from the following files:

- `src/Sampling/Sampling.h`
- `src/Sampling/Sampling.cpp`

3.33 Simple Class Reference

[Simple](#) Isotropic diffusion model.

Inheritance diagram for Simple:



Public Member Functions

- **Simple** (int n_p, int dim, double [timestep](#), seed_type [random_seed](#), double D=0.-5)
- double [get_new_pos](#) (const [Walker](#) *walker, int i, int j)
Virtual function returning the new position.
- double [get_g_ratio](#) (const [Walker](#) *walker_post, const [Walker](#) *walker_pre) const
Calculates the [Diffusion](#) Green's function ratio needed by metropolis.

3.33.1 Detailed Description

[Simple](#) Isotropic diffusion model.

3.33.2 Member Function Documentation

3.33.2.1 `double Simple::get_g_ratio (const Walker * walker_post, const Walker * walker_pre) const` `[inline, virtual]`

Calculates the [Diffusion](#) Green's function ratio needed by metropolis.

Parameters

<i>walker_post</i>	Walker at current time step.
<i>walker_pre</i>	Walker at previous time step.

Returns

The [Diffusion](#) Green's function ratio.

Implements [Diffusion](#).

3.33.2.2 `double Simple::get_new_pos (const Walker * walker, int i, int j)` `[inline, virtual]`

Virtual function returning the new position.

Returns the simple diffusion step if not overridden.

Parameters

<i>i</i>	Particle number.
<i>j</i>	dimension (x,y,z).

Returns

The new position (relative to the old).

Reimplemented from [Diffusion](#).

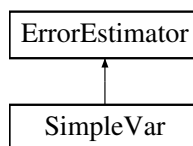
The documentation for this class was generated from the following files:

- `src/Diffusion/Simple/Simple.h`
- `src/Diffusion/Simple/Simple.cpp`

3.34 SimpleVar Class Reference

Calculates the simple variance of the sampled values.

Inheritance diagram for SimpleVar:



Public Member Functions

- **SimpleVar** ([ParParams](#) &)
- double [estimate_error](#) ()
Estimates the error based on the subclass implementation.
- void [update_data](#) (double val)
- void **normalize** ()

Protected Attributes

- double [f](#)
sum variable used to calculate the mean
- double [f2](#)
sum variable used to calculate the mean of squares.

3.34.1 Detailed Description

Calculates the simple variance of the sampled values.

3.34.2 Member Function Documentation

3.34.2.1 void SimpleVar::update_data (double *val*) [virtual]

Overrides the default described in the superclass. Does not store values in memory, but rather use sum variables.

Reimplemented from [ErrorEstimator](#).

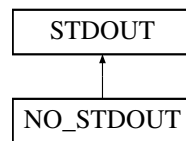
The documentation for this class was generated from the following files:

- src/ErrorEstimator/SimpleVar/SimpleVar.h
- src/ErrorEstimator/SimpleVar/SimpleVar.cpp

3.35 STDOUT Class Reference

Class for handling standard output. Only the master node has this object.

Inheritance diagram for STDOUT:



Public Member Functions

- virtual void **cout** (std::stringstream &a)

3.35.1 Detailed Description

Class for handling standard output. Only the master node has this object.

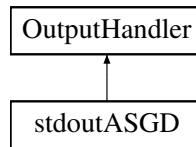
The documentation for this class was generated from the following file:

- src/QMChaders.h

3.36 stdoutASGD Class Reference

Class for handling the output of [ASGD](#). Outputs values such as the variational gradients, step length, variational parameters, etc.

Inheritance diagram for stdoutASGD:



Public Member Functions

- **stdoutASGD** (std::string path, std::string filename="ASGD_out")
- void [dump](#) ()
Methods for updating the output.
- void [post_pointer_init](#) ()

3.36.1 Detailed Description

Class for handling the output of [ASGD](#). Outputs values such as the variational gradients, step length, variational parameters, etc.

3.36.2 Member Function Documentation

3.36.2.1 void stdoutASGD::dump () [virtual]

Methods for updating the output.

Typically retrieves information through the solver pointers (given correct accessibility levels/friend)

Implements [OutputHandler](#).

3.36.2.2 void stdoutASGD::post_pointer_init () [inline, virtual]

Initializes the correct size of the variational gradient once the min pointer has been cast to [ASGD](#).

Reimplemented from [OutputHandler](#).

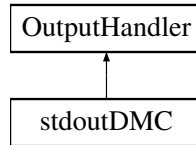
The documentation for this class was generated from the following files:

- src/OutputHandler/stdoutASGD/stdoutASGD.h
- src/OutputHandler/stdoutASGD/stdoutASGD.cpp

3.37 stdoutDMC Class Reference

Class for handling the output of [DMC](#). Outputs values such as the trial energy, dmc energy, number of walkers, etc.

Inheritance diagram for stdoutDMC:



Public Member Functions

- **stdoutDMC** (std::string path, std::string filename="DMC_out")
- void [dump](#) ()

Methods for updating the output.

Protected Attributes

- int [n](#)

Number of times the [dump\(\)](#) method has been called.

- double [sumE](#)

Sum of the [DMC](#) energy used to calculate the trailing average.

- double [sumN](#)

Sum of the number of walkers used to calculate the trailing average.

3.37.1 Detailed Description

Class for handling the output of [DMC](#). Outputs values such as the trial energy, dmc energy, number of walkers, etc.

3.37.2 Member Function Documentation

3.37.2.1 void stdoutDMC::dump () [virtual]

Methods for updating the output.

Typically retrieves information through the solver pointers (given correct accessibility levels/friend)

Implements [OutputHandler](#).

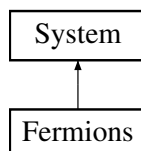
The documentation for this class was generated from the following files:

- src/OutputHandler/stdoutDMC/stdoutDMC.h
- src/OutputHandler/stdoutDMC/stdoutDMC.cpp

3.38 System Class Reference

The system class separating [Fermions](#) and Bosons. Designed to generalize the solver in terms of particle species.

Inheritance diagram for System:



Public Member Functions

- **System** (int n_p, int dim, [Orbitals](#) *orbital)
- virtual void [initialize](#) ([Walker](#) *walker)=0
Initializes the system before the main solver loop starts.
- void [add_potential](#) ([Potential](#) *pot)
Method for adding a potential to the system.
- double [get_potential_energy](#) (const [Walker](#) *walker)
Method for calculating the total potential energy.
- virtual void [update_walker](#) ([Walker](#) *walker_pre, const [Walker](#) *walker_post, int particle) const =0
- virtual void [reset_walker](#) (const [Walker](#) *walker_pre, [Walker](#) *walker_post, int particle) const =0
- virtual void [calc_for_newpos](#) (const [Walker](#) *walker_old, [Walker](#) *walker_new, int particle)=0
Method for calculating the necessary values needed by the walker after a new step is made.
- virtual double [get_spatial_ratio](#) (const [Walker](#) *walker_pre, const [Walker](#) *walker_post, int particle)=0
Method for calculating the spatial wave function ratios between two subsequent time steps.
- virtual double [get_spatial_wf](#) (const [Walker](#) *walker)=0
Method for calculating the spatial wave function's value at a given walkers position.
- virtual void [get_spatial_grad](#) ([Walker](#) *walker, int particle) const =0
Method for calculating the changed part of the spatial gradient.
- virtual void [get_spatial_grad_full](#) ([Walker](#) *walker) const =0
Method for calculating the full spatial gradient.
- virtual double [get_spatial_lapl_sum](#) (const [Walker](#) *walker) const =0
Method for calculating the sum of all Laplacians for a given walker.
- virtual void [copy_walker](#) (const [Walker](#) *parent, [Walker](#) *child) const =0
Method for copying the system specific parts of a walker.
- virtual bool [allow_transition](#) ()=0

Method allowing the system to override the Metropolis test.

- [Orbitals](#) * **get_orbital_ptr** ()
- void **set_spin_state** (int [start](#), int [end](#))

Protected Attributes

- int **n_p**
- int **n2**
- int **dim**
- int [start](#)

The start point of separable calculations.

- int [end](#)

The end point of separable calculations.

- std::vector< [Potential](#) * > [potentials](#)

A vector of potentials.

- [Orbitals](#) * [orbital](#)

The single particle wave function object.

3.38.1 Detailed Description

The system class separating [Fermions](#) and Bosons. Designed to generalize the solver in terms of particle species.

3.38.2 Member Function Documentation

3.38.2.1 virtual void **System::calc_for_newpos** (const Walker * *walker_old*, Walker * *walker_new*, int *particle*) [pure virtual]

Method for calculating the necessary values needed by the walker after a new step is made.

Given a particle number, the method does not recompute unchanged values.

Parameters

<i>walker_old</i>	Walker at current time step.
<i>walker_new</i>	Walker at previous time step.

Implemented in [Fermions](#).

3.38.2.2 virtual void **System::copy_walker** (const Walker * *parent*, Walker * *child*) const [pure virtual]

Method for copying the system specific parts of a walker.

See also

[QMC::copy_walker\(\)](#)

Implemented in [Fermions](#).

3.38.2.3 double System::get_potential_energy (const Walker * walker)

Method for calculating the total potential energy.

Iterates over all objects in the potentials vector and accumulates their potential energies for the given walker.

3.38.2.4 virtual void System::get_spatial_grad (Walker * walker, int particle) const [pure virtual]

Method for calculating the changed part of the spatial gradient.

Depending on which particle we moved, one of the spatial wave function parts (it is split) will be unchanged.

Implemented in [Fermions](#).

3.38.2.5 virtual void System::initialize (Walker * walker) [pure virtual]

Initializes the system before the main solver loop starts.

Called by the [Sampling](#) class when trial positions are set.

Implemented in [Fermions](#).

3.38.2.6 virtual void System::reset_walker (const Walker * walker_pre, Walker * walker_post, int particle) const [pure virtual]

See also

[QMC::reset_walker\(\)](#)

Implemented in [Fermions](#).

3.38.2.7 void System::set_spin_state (int start, int end) [inline]

See also

[QMC::set_spin_state\(\)](#)

3.38.2.8 virtual void System::update_walker (Walker * walker_pre, const Walker * walker_post, int particle) const [pure virtual]

See also

[QMC::update_walker\(\)](#)

Implemented in [Fermions](#).

3.38.3 Member Data Documentation

3.38.3.1 `int System::end` [protected]

The end point of separable calculations.

Either N/2 or N. Since the spatial wave function is split, particles with spin not equal that of the moved particle is unchanged and does not need to be recalculated.

3.38.3.2 `int System::start` [protected]

The start point of separable calculations.

Either 0 or N/2. Since the spatial wave function is split, particles with spin not equal that of the moved particle is unchanged and does not need to be recalculated.

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

- `src/System/System.h`
- `src/System/System.cpp`

3.39 SystemObjects Struct Reference

Struct used to initialize system objects.

Public Attributes

- [Orbitals](#) * **SP_basis**
- [Potential](#) * **onebody_pot**
- [System](#) * **SYSTEM**
- [Sampling](#) * **sample_method**
- [Jastrow](#) * **jastrow**

3.39.1 Detailed Description

Struct used to initialize system objects.

The memory addresses allocated here will not change throughout the run.

See also

[Orbitals](#), [Potential](#), [System](#), [Sampling](#), [Jastrow](#)

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

- `src/QMHeaders.h`

3.40 VariationalParams Struct Reference

Struct used to initialize the varational parameters.

Public Attributes

- double [alpha](#)
The spatial variational parameter.
- double [beta](#)
The [Jastrow](#) variational parameter.

3.40.1 Detailed Description

Struct used to initialize the varational parameters.

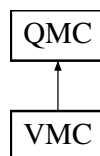
The documentation for this struct was generated from the following file:

- `src/QMHeaders.h`

3.41 VMC Class Reference

Implementation of the Variational Monte-Carlo Method. Very little needs to be added when the [QMC](#) superclass holds all the general functionality.

Inheritance diagram for VMC:



Public Member Functions

- [VMC](#) ([GeneralParams](#) &, [VMCparams](#) &, [SystemObjects](#) &, [ParParams](#) &, int `n_w`, bool `dist_out`)

Constructor.

- void **set_e** (double E)
- double **get_energy** () const
- void [run_method](#) ()

Method used for executing the solver.

- void [output](#) ()

Method for standard output.

Protected Member Functions

- void [set_trial_positions](#) ()
- void [store_walkers](#) ()

Method for storing walkers for DMC.

- void [save_distribution](#) ()

Method for storing positional data for the Distribtuon.

- bool [move_authorized](#) (double A)
- void **scale_values** ()
- void [node_comm](#) ()

Method for performing node communication.

Protected Attributes

- int [pop_tresh](#)

The amount of cycles between storing walkers for DMC.

- int [offset](#)

The amount of cycles before starting to store walkers for DMC.

- int [last_walker](#)

Count variable for the last walker stores for DMC.

- double [vmc_E](#)

The VMC energy.

- [Walker](#) * [original_walker](#)

The VMC walker.

Friends

- class **DMC**
- class **Minimizer**
- class **ASGD**
- class **BlockingData**

3.41.1 Detailed Description

Implementation of the Variational Monte-Carlo Method. Very little needs to be added when the [QMC](#) superclass holds all the general functionality.

3.41.2 Member Function Documentation

3.41.2.1 `bool VMC::move_authorized (double A) [inline, protected, virtual]`

In [VMC](#), only the metropolis test is performed.

Implements [QMC](#).

3.41.2.2 `void VMC::save_distribution () [protected, virtual]`

Method for storing positional data for the Distribtuon.

Stores the position data of the single [VMC](#) walker every `dist_tresh` cycle after thermalization.

Implements [QMC](#).

3.41.2.3 `void VMC::set_trial_positions () [protected, virtual]`

Sets the trial position for the single walker.

Implements [QMC](#).

3.41.2.4 `void VMC::store_walkers () [protected]`

Method for storing walkers for [DMC](#).

Stores the single [VMC](#) walker every `pop_thresh` cycle after offset cycles.

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

- `src/QMC/VMC/VMC.h`
- `src/QMC/VMC/VMC.cpp`

3.42 VMCparams Struct Reference

Struct used to initialize [VMC](#) parameters.

Public Attributes

- `int n_c`
The number of cycles.
- `double dt`
The time step.

3.42.1 Detailed Description

Struct used to initialize [VMC](#) parameters.

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

- `src/QMHeaders.h`

3.43 Walker Class Reference

Class representing a Random [Walker](#). Holds position data, alive state, etc. Designed to lighten function arguments, and ease implementation of [QMC](#) methods involving multiple walkers. A lot of values are stored to avoid calculating the same value twice.

Public Member Functions

- **Walker** (int n_p, int dim, bool do_init=true)
- void [calc_r_i2](#) (int i)
Method for calculating the radius squared for one particle.
- void [calc_r_i2](#) ()
Method for calculating the radius squared for all particles.
- double [calc_r_rel](#) (int i, int j) const
Method for calculating the relative distance between two particles.
- void [make_rel_matrix](#) ()
- void [send_soul](#) (int source)
- void [recv_soul](#) (int root)
- double [get_r_i2](#) (int i) const
Method for fetching the squared radius of a particle.
- double [get_r_i](#) (int i) const
Method for calculating the radius of a particle.
- void [kill](#) ()
- bool [is_dead](#) ()
- bool [is_alive](#) ()
- void [ressurrect](#) ()
- void [set_E](#) (double E)
- double [get_E](#) () const
- void [print](#) (std::string header="----") const
Prints out all the walkers information.

Public Attributes

- double [spatial_ratio](#)
The ratio of the spatial wave function (stored in the newest walker).
- double [lapl_sum](#)

The sum of the Laplacians of all particles.

- double [E](#)

The energy of the given configuration (stored to speed up [DMC](#)).

- arma::mat [r](#)

The positions of all particles.

- arma::mat [r_rel](#)

The relative positions of all particles.

- arma::mat [qforce](#)

The Quantum Force for all particles.

- arma::mat [spatial_grad](#)

The gradient of the Spatial Wave function for all particles.

- arma::mat [jast_grad](#)

The gradient of the [Jastrow](#) Factor for all particles.

- arma::mat [inv](#)

The inverse of the Slater matrix (given fermion system)

- arma::mat [phi](#)

The single particle wave functions for all particles and quantum numbers.

- arma::field< arma::mat > [dell_phi](#)

The derivatives of the single particle wave functions for all particles and quantum numbers.

- arma::cube [dJ](#)

Cube used for storing sum terms for the [Jastrow](#) Factor's closed form expressions.

- arma::rowvec [r2](#)

The radius squared for all particles.

Protected Attributes

- int [n_p](#)
- int [n2](#)
- int [dim](#)
- bool [is_murdered](#)

If true, the walker will be deleted and removed ([DMC](#) only).

3.43.1 Detailed Description

Class representing a Random [Walker](#). Holds position data, alive state, etc. Designed to lighten function arguments, and ease implementation of [QMC](#) methods involving multiple walkers. A lot of values are stored to avoid calculating the same value twice.

3.43.2 Member Function Documentation

3.43.2.1 void Walker::calc_r_i2 (int *i*)

Method for calculating the radius squared for one particle.

Parameters

<i>i</i>	The particle number.
----------	----------------------

3.43.2.2 double Walker::calc_r_rel (int *i*, int *j*) const

Method for calculating the relative distance between two particles.

Parameters

<i>i,j</i>	The particle numbers.
------------	-----------------------

3.43.2.3 double Walker::get_r_i (int *i*) const [inline]

Method for calculating the radius of a particle.

Parameters

<i>i</i>	Particle number.
----------	------------------

3.43.2.4 double Walker::get_r_i2 (int *i*) const [inline]

Method for fetching the squared radius of a particle.

Used in order to avoid calculating the same radius twice.

Parameters

<i>i</i>	Particle number.
----------	------------------

3.43.2.5 void Walker::kill () [inline]

Flags the walker for destruction.

See also

[DMC::bury_the_dead\(\)](#)

3.43.2.6 void Walker::make_rel_matrix ()

Creates the relative position matrix.

3.43.2.7 void Walker::print (std::string *header* = "----") const

Prints out all the walkers information.

Extremely handy for debugging.

Parameters

<i>header</i>	A header for the printout in order to distinguish several printouts easily.
---------------	---

3.43.2.8 void Walker::recv_soul (int *root*)

Receives a walker from a different node.

Parameters

<i>root</i>	The rank of the node from which the walker was sent.
-------------	--

3.43.2.9 void Walker::ressurect () [inline]

Sets the destruction flag to false.

3.43.2.10 void Walker::send_soul (int *source*)

Send a walker to a different node.

Parameters

<i>source</i>	The receiving node's rank.
---------------	----------------------------

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

- src/Walker/Walker.h
- src/Walker/Walker.cpp