

Evolutionary Algorithms for Financial Optimisation

Generated by Doxygen 1.8.13

Contents

1	Main Page	1
1.1	Introduction	1
1.2	Template Parameters	1
1.3	Compilation	1
1.3.1	External Libraries	1
1.4	Showcase	2
2	Namespace Index	3
2.1	Namespace List	3
3	Hierarchical Index	5
3.1	Class Hierarchy	5
4	Class Index	7
4.1	Class List	7
5	File Index	9
5.1	File List	9

6	Namespace Documentation	11
6.1	bond Namespace Reference	11
6.1.1	Detailed Description	11
6.1.2	Enumeration Type Documentation	11
6.1.2.1	Bond_pricing_type	11
6.1.3	Function Documentation	12
6.1.3.1	read_bonds_from_file()	12
6.2	ea Namespace Reference	12
6.2.1	Detailed Description	14
6.2.2	Enumeration Type Documentation	14
6.2.2.1	Strategy	14
6.2.3	Function Documentation	14
6.2.3.1	generator()	14
6.2.3.2	solve()	15
6.2.4	Variable Documentation	16
6.2.4.1	inv_pi_sq	17
6.2.4.2	inv_pi_sq_2	17
6.2.4.3	rd	17
6.3	irr Namespace Reference	17
6.3.1	Detailed Description	18
6.3.2	Function Documentation	18
6.3.2.1	compute_discount_factor()	18
6.3.2.2	compute_pv()	19
6.3.2.3	constraints_irr()	20
6.3.2.4	fitness_irr()	21
6.3.2.5	penalty_irr()	22
6.4	nss Namespace Reference	23
6.4.1	Detailed Description	23
6.4.2	Function Documentation	23
6.4.2.1	constraints_svensson()	24
6.4.2.2	penalty_svensson()	25
6.4.2.3	svensson()	26
6.5	utilities Namespace Reference	27
6.5.1	Detailed Description	28
6.5.2	Enumeration Type Documentation	28
6.5.2.1	Constraints_type	28
6.5.2.2	DF_type	28
6.5.3	Function Documentation	29
6.5.3.1	operator<<()	29
6.6	yft Namespace Reference	29
6.6.1	Detailed Description	30
6.6.2	Function Documentation	30
6.6.2.1	read_ir_from_file()	30

7	Class Documentation	31
7.1	bond::Bond< T > Class Template Reference	31
7.1.1	Detailed Description	32
7.1.2	Constructor & Destructor Documentation	32
7.1.2.1	Bond()	32
7.1.3	Member Function Documentation	33
7.1.3.1	compute_cash_flows()	33
7.1.3.2	compute_macaulay_duration()	34
7.1.3.3	compute_yield() [1/2]	35
7.1.3.4	compute_yield() [2/2]	36
7.1.4	Friends And Related Function Documentation	37
7.1.4.1	BondHelper< T >	37
7.1.5	Member Data Documentation	37
7.1.5.1	cash_flows	37
7.1.5.2	coupon_percentage	38
7.1.5.3	coupon_value	38
7.1.5.4	duration	38
7.1.5.5	frequency	38
7.1.5.6	maturity_date	38
7.1.5.7	nominal_value	39
7.1.5.8	price	39
7.1.5.9	settlement_date	39
7.1.5.10	time_periods	39
7.1.5.11	yield	39
7.2	bond::BondHelper< T > Class Template Reference	40
7.2.1	Detailed Description	40
7.2.2	Constructor & Destructor Documentation	41
7.2.2.1	BondHelper()	41
7.2.3	Member Function Documentation	41
7.2.3.1	bond_pricing()	41

7.2.3.2	estimate_bond_pricing()	42
7.2.3.3	fitness_bond_pricing_prices()	43
7.2.3.4	fitness_bond_pricing_yields()	44
7.2.3.5	print_bond_pricing_results()	45
7.2.3.6	set_init_nss_params()	46
7.2.4	Member Data Documentation	47
7.2.4.1	bonds	47
7.2.4.2	df_type	47
7.3	ea::DE< T > Struct Template Reference	48
7.3.1	Detailed Description	49
7.3.2	Constructor & Destructor Documentation	49
7.3.2.1	DE()	49
7.3.3	Member Data Documentation	50
7.3.3.1	cr	50
7.3.3.2	f_param	50
7.3.3.3	type	51
7.4	ea::EA_base< T > Struct Template Reference	51
7.4.1	Detailed Description	52
7.4.2	Member Typedef Documentation	52
7.4.2.1	fp_type	52
7.4.3	Constructor & Destructor Documentation	53
7.4.3.1	EA_base()	53
7.4.4	Member Data Documentation	54
7.4.4.1	constraints_type	54
7.4.4.2	decision_variables	54
7.4.4.3	iter_max	54
7.4.4.4	ndv	54
7.4.4.5	npop	55
7.4.4.6	print_to_file	55
7.4.4.7	print_to_output	55

7.4.4.8	stdev	55
7.4.4.9	tol	55
7.4.4.10	use_penalty_method	56
7.5	ea::GA< T > Struct Template Reference	56
7.5.1	Detailed Description	57
7.5.2	Constructor & Destructor Documentation	57
7.5.2.1	GA()	57
7.5.3	Member Data Documentation	58
7.5.3.1	alpha	58
7.5.3.2	pi	59
7.5.3.3	strategy	59
7.5.3.4	type	59
7.5.3.5	x_rate	59
7.6	yft::Interest_Rate< T > Struct Template Reference	60
7.6.1	Detailed Description	60
7.6.2	Constructor & Destructor Documentation	60
7.6.2.1	Interest_Rate()	60
7.6.3	Member Data Documentation	61
7.6.3.1	period	61
7.6.3.2	rate	61
7.7	yft::Interest_Rate_Helper< T > Class Template Reference	61
7.7.1	Detailed Description	62
7.7.2	Constructor & Destructor Documentation	62
7.7.2.1	Interest_Rate_Helper()	62
7.7.3	Member Function Documentation	62
7.7.3.1	fitness_yield_curve_fitting()	63
7.7.3.2	yieldcurve_fitting()	64
7.7.4	Member Data Documentation	64
7.7.4.1	ir_vec	65
7.8	ea::PSO< T > Struct Template Reference	65

7.8.1	Detailed Description	66
7.8.2	Constructor & Destructor Documentation	66
7.8.2.1	PSOI()	66
7.8.3	Member Data Documentation	67
7.8.3.1	c	67
7.8.3.2	type	68
7.8.3.3	vmax	68
7.8.3.4	w	68
7.9	ea::PSOs< T > Struct Template Reference	69
7.9.1	Detailed Description	70
7.9.2	Constructor & Destructor Documentation	70
7.9.2.1	PSOs()	70
7.9.3	Member Data Documentation	71
7.9.3.1	alpha	71
7.9.3.2	c1	72
7.9.3.3	c2	72
7.9.3.4	sneigh	72
7.9.3.5	type	72
7.9.3.6	vmax	73
7.9.3.7	w	73
7.10	ea::Solver< S, T, F, C > Class Template Reference	73
7.10.1	Detailed Description	73
7.11	ea::Solver< DE, T, F, C > Class Template Reference	74
7.11.1	Detailed Description	75
7.11.2	Constructor & Destructor Documentation	75
7.11.2.1	Solver()	75
7.11.3	Member Function Documentation	76
7.11.3.1	construct_donor()	76
7.11.3.2	construct_trial()	77
7.11.3.3	display_parameters()	78

7.11.3.4	run_algo()	78
7.11.3.5	set_indices()	79
7.11.4	Friends And Related Function Documentation	79
7.11.4.1	Solver_base< Solver< DE, T, F, C >, DE, T, F, C >	79
7.11.5	Member Data Documentation	80
7.11.5.1	de	80
7.11.5.2	ind_distribution	80
7.11.5.3	indices	80
7.12	ea::Solver< GA, T, F, C > Class Template Reference	81
7.12.1	Detailed Description	82
7.12.2	Constructor & Destructor Documentation	82
7.12.2.1	Solver()	82
7.12.3	Member Function Documentation	83
7.12.3.1	crossover()	83
7.12.3.2	display_parameters()	84
7.12.3.3	mutation()	84
7.12.3.4	nkeep()	85
7.12.3.5	run_algo()	86
7.12.3.6	selection()	87
7.12.4	Friends And Related Function Documentation	88
7.12.4.1	Solver_base< Solver< GA, T, F, C >, GA, T, F, C >	88
7.12.5	Member Data Documentation	88
7.12.5.1	bdistribution	88
7.12.5.2	ga	88
7.12.5.3	npop	88
7.12.5.4	stdev	89
7.13	ea::Solver< PSOI, T, F, C > Class Template Reference	89
7.13.1	Detailed Description	92
7.13.2	Constructor & Destructor Documentation	92
7.13.2.1	Solver()	92

7.13.3	Member Function Documentation	93
7.13.3.1	best_update()	93
7.13.3.2	check_pso_criteria()	93
7.13.3.3	display_parameters()	94
7.13.3.4	euclid_distance()	94
7.13.3.5	find_min_local_best()	95
7.13.3.6	position_update()	96
7.13.3.7	run_algo()	97
7.13.3.8	set_neighbourhoods()	97
7.13.4	Friends And Related Function Documentation	98
7.13.4.1	Solver_base< Solver< PSOI, T, F, C >, PSOI, T, F, C >	98
7.13.5	Member Data Documentation	98
7.13.5.1	local_best	98
7.13.5.2	neighbours	98
7.13.5.3	nneigh	98
7.13.5.4	personal_best	99
7.13.5.5	personal_best_cost	99
7.13.5.6	pso	99
7.13.5.7	velocity	99
7.13.5.8	vmax	99
7.13.5.9	w	100
7.14	ea::Solver< PSOs, T, F, C > Class Template Reference	100
7.14.1	Detailed Description	103
7.14.2	Constructor & Destructor Documentation	103
7.14.2.1	Solver()	103
7.14.3	Member Function Documentation	104
7.14.3.1	best_update()	104
7.14.3.2	check_pso_criteria()	105
7.14.3.3	display_parameters()	105
7.14.3.4	euclid_distance()	106

7.14.3.5	find_min_local_best()	106
7.14.3.6	generate_r()	107
7.14.3.7	position_update()	107
7.14.3.8	run_algo()	108
7.14.3.9	set_neighbourhoods()	109
7.14.4	Friends And Related Function Documentation	109
7.14.4.1	Solver_base< Solver< PSOs, T, F, C >, PSOs, T, F, C >	109
7.14.5	Member Data Documentation	109
7.14.5.1	local_best	109
7.14.5.2	neighbourhoods	110
7.14.5.3	nneigh	110
7.14.5.4	personal_best	110
7.14.5.5	pso	110
7.14.5.6	velocity	110
7.14.5.7	vmax	111
7.14.5.8	w	111
7.15	ea::Solver_base< Derived, S, T, F, C > Class Template Reference	111
7.15.1	Detailed Description	112
7.15.2	Constructor & Destructor Documentation	112
7.15.2.1	Solver_base()	112
7.15.3	Member Function Documentation	113
7.15.3.1	display_results()	113
7.15.3.2	find_min_cost()	114
7.15.3.3	init_individuals()	114
7.15.3.4	randomise_individual()	115
7.15.3.5	solver_bench()	115
7.15.3.6	write_results_to_file()	116
7.15.4	Member Data Documentation	117
7.15.4.1	c	117
7.15.4.2	distribution	117
7.15.4.3	f	117
7.15.4.4	individuals	118
7.15.4.5	last_iter	118
7.15.4.6	min_cost	118
7.15.4.7	solved_flag	118
7.15.4.8	solver_struct	118
7.15.4.9	timer	118

8 File Documentation	119
8.1 bond.h File Reference	119
8.1.1 Detailed Description	120
8.2 bondhelper.h File Reference	120
8.2.1 Detailed Description	122
8.3 differentialevo.h File Reference	122
8.3.1 Detailed Description	123
8.4 ealgorithm_base.h File Reference	123
8.4.1 Detailed Description	125
8.5 geneticalgo.h File Reference	125
8.5.1 Detailed Description	126
8.6 irr.h File Reference	126
8.6.1 Detailed Description	128
8.7 lbestpso.h File Reference	128
8.7.1 Detailed Description	129
8.8 main.cpp File Reference	129
8.8.1 Detailed Description	130
8.8.2 Function Documentation	130
8.8.2.1 main()	130
8.9 pso_sub_swarm.h File Reference	131
8.9.1 Detailed Description	132
8.10 svensson.h File Reference	132
8.10.1 Detailed Description	133
8.11 utilities.h File Reference	133
8.11.1 Detailed Description	134
8.12 yield_curve_fitting.h File Reference	135
8.12.1 Detailed Description	136
Index	137

Chapter 1

Main Page

1.1 Introduction

This project implements solvers for three financial optimisation problems: **Yield Curve Fitting**, **Internal Rate of Return Estimation** and **Bond Pricing** problems.

Some general guidelines and remarks are mentioned below.

1.2 Template Parameters

A template parameter named **T** denotes a floating-point number type.

A template parameter named **F** denotes a lambda function (or function object/functor) type of the objective function

A template parameter named **C** denotes a lambda function (or function object/functor) type of the constraints function

A template parameter named **S** denotes a solver structure.

1.3 Compilation

Compilation is possible with the three major compilers, gcc, clang and msvc.

Since some of the features used in the project are from the C++11/C++14 standards, the source has to be compiled with at least **-std=c++14**.

For gcc and clang, **-std=c++17** can be used as well, as the source is compatible with the newest C++ ISO standard.

1.3.1 External Libraries

The external libraries used by the project are: **Boost** (<http://www.boost.org/>) and **date** (<https://github.com/HowardHinnant/date>).

Boost is used for the Beta Distribution implementation and the pi constant.

date is used for date handling of the settlement and maturity dates.

Both can be included only as their header versions.

1.4 Showcase

A showcase of the project is provided under [tests/main.cpp](#).

Be sure that the resulting executable will be run in the same working directory as the data files, otherwise the executable will crash abruptly, since exceptions are not implemented yet.

If other data files are used, be sure that their content has the correct format.

For bond data files, the input data have to be of the form: **coupon rate (in percentage) price nominal value frequency settlement date (in yyyy-mm-dd form) and maturity date (in yyyy-mm-dd form)**.

For example: **0.06 101.657 100 2 2016-03-30 2019-06-24** has the correct format.

For interest rate data files, the input data have to be of the form: **period (as a decimal) zero rate (in percentage)**.

For example: **0.25 0.079573813** has the correct format.

Chapter 2

Namespace Index

2.1 Namespace List

Here is a list of all namespaces with brief descriptions:

bond	Bond Class and Utilities	11
ea	Evolutionary Algorithms	12
irr	Internal Rate of Return (IRR) namespace	17
nss	Nelson-Siegel-Svensson (NSS) model namespace	23
utilities	Utilities namespace	27
yft	Yield Curve Fitting namespace	29

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

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

bond::Bond< T >	31
bond::BondHelper< T >	40
ea::EA_base< T >	51
ea::DE< T >	48
ea::GA< T >	56
ea::PSOI< T >	65
ea::PSOs< T >	69
yft::Interest_Rate< T >	60
yft::Interest_Rate_Helper< T >	61
ea::Solver< S, T, F, C >	73
ea::Solver_base< Derived, S, T, F, C >	111
ea::Solver_base< Solver< DE, T, F, C >, DE, T, F, C >	111
ea::Solver< DE, T, F, C >	74
ea::Solver_base< Solver< GA, T, F, C >, GA, T, F, C >	111
ea::Solver< GA, T, F, C >	81
ea::Solver_base< Solver< PSOI, T, F, C >, PSOI, T, F, C >	111
ea::Solver< PSOI, T, F, C >	89
ea::Solver_base< Solver< PSOs, T, F, C >, PSOs, T, F, C >	111
ea::Solver< PSOs, T, F, C >	100

Chapter 4

Class Index

4.1 Class List

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

bond::Bond< T >	
Bond Class definition	31
bond::BondHelper< T >	
A class for the bond pricing problem as well as finding the yield-to-maturities of bonds	40
ea::DE< T >	
Differential Evolution Structure, used in the actual algorithm and for type deduction	48
ea::EA_base< T >	
Evolutionary algorithm stucture base	51
ea::GA< T >	
Genetic Algorithms Structure, used in the actual algorithm and for type deduction	56
yft::Interest_Rate< T >	
Structure for interest rates	60
yft::Interest_Rate_Helper< T >	
A class for the yield-curve-fitting problem	61
ea::PSOI< T >	
Local Best Particle Swarm Optimisation Structure, used in the actual algorithm and for type deduction	65
ea::PSOs< T >	
Particle Swarm Optimisation Structure, used in the actual algorithm and for type deduction	69
ea::Solver< S, T, F, C >	
Template Class for Solvers	73
ea::Solver< DE, T, F, C >	
Differential Evolution Algorithm (DE) Class	74
ea::Solver< GA, T, F, C >	
Genetic Algorithms (GA) Class	81
ea::Solver< PSOI, T, F, C >	
Local Best Particle Swarm Optimisation (PSO) Class	89
ea::Solver< PSOs, T, F, C >	
Sub-Swarm Particle Swarm Optimisation (PSO) Class	100
ea::Solver_base< Derived, S, T, F, C >	
Base Class for Evolutionary Algorithms	111

Chapter 5

File Index

5.1 File List

Here is a list of all files with brief descriptions:

bond.h	Classes and functions for bonds and their internal rate of return	119
bondhelper.h	Classes and functions for the bond pricing problem	120
differentialevo.h	Classes and functions for Differential Evolution	122
ealgorithm_base.h	Classes and functions for the base of the solvers	123
geneticalgo.h	Classes and functions for Genetic Algorithms	125
irr.h	Functions for the Internal Rate of Return	126
lbestpso.h	Classes and functions for the lbest (Local Best) Particle Swarm Optimisation with a ring topology	128
main.cpp	A showcase of the application of the solvers on the Yield Curve Fitting, Internal Rate of Return Estimation and Bond Pricing Problems	129
pso_sub_swarm.h	Classes and functions for the initial implementation of Sub-Swarm Particle Swarm Optimisation	131
svensson.h	Functions for the Nelson-Siegel-Svensson model	132
utilities.h	Enumerations and functions used from the rest of the project	133
yield_curve_fitting.h	Class and functions for the yield curve fitting problem	135

Chapter 6

Namespace Documentation

6.1 bond Namespace Reference

[Bond](#) Class and Utilities.

Classes

- class [Bond](#)
[Bond](#) Class definition.
- class [BondHelper](#)
A class for the bond pricing problem as well as finding the yield-to-maturities of bonds.

Enumerations

- enum [Bond_pricing_type](#) { [Bond_pricing_type::bpp](#), [Bond_pricing_type::bpy](#) }
Enumeration for type of bondpricing, using yields or prices.

Functions

- template<typename T >
std::vector< [Bond](#)< T > > [read_bonds_from_file](#) (const std::string &filename)
Reads bond data from a file.

6.1.1 Detailed Description

[Bond](#) Class and Utilities.

6.1.2 Enumeration Type Documentation

6.1.2.1 Bond_pricing_type

```
enum bond::Bond\_pricing\_type [strong]
```

Enumeration for type of bondpricing, using yields or prices.

Enumerator

bpp	Use bond prices
bpy	Use bond yields-to-maturities

Definition at line 21 of file bondhelper.h.

```

22     {
23         bpp,
24         bpy
25     };

```

6.1.3 Function Documentation**6.1.3.1 read_bonds_from_file()**

```

template<typename T >
bond::read_bonds_from_file (
    const std::string & filename )

```

Reads bond data from a file.

Parameters

<i>filename</i>	The name of the input file as an std::string
-----------------	--

Returns

A vector of Bond<T> objects

Definition at line 33 of file bondhelper.h.

```

34     {
35         std::vector<Bond<T>> bonds;
36         std::ifstream input(filename);
37         for (std::string line; getline(input, line); )
38         {
39             T coupon_percentage;
40             T price;
41             T nominal_value;
42             T frequency;
43             std::string settlement_date;
44             std::string maturity_date;
45             std::istreamstream stream(line);
46             stream >> coupon_percentage >> price >> nominal_value >> frequency >> settlement_date >>
maturity_date;
47             const Bond<T> bond{ coupon_percentage, price, nominal_value, frequency, settlement_date,
maturity_date };
48             bonds.push_back(bond);
49         }
50         return bonds;
51     }

```

6.2 ea Namespace Reference

Evolutionary Algorithms.

Classes

- struct [DE](#)
Differential Evolution Structure, used in the actual algorithm and for type deduction.
- struct [EA_base](#)
Evolutionary algorithm stucture base.
- struct [GA](#)
Genetic Algorithms Structure, used in the actual algorithm and for type deduction.
- struct [PSOI](#)
Local Best Particle Swarm Optimisation Structure, used in the actual algorithm and for type deduction.
- struct [PSOs](#)
Particle Swarm Optimisation Structure, used in the actual algorithm and for type deduction.
- class [Solver](#)
Template Class for Solvers.
- class [Solver< DE, T, F, C >](#)
Differential Evolution Algorithm (DE) Class.
- class [Solver< GA, T, F, C >](#)
Genetic Algorithms (GA) Class.
- class [Solver< PSOI, T, F, C >](#)
Local Best Particle Swarm Optimisation (PSO) Class.
- class [Solver< PSOs, T, F, C >](#)
Sub-Swarm Particle Swarm Optimisation (PSO) Class.
- class [Solver_base](#)
Base Class for Evolutionary Algorithms.

Enumerations

- enum [Strategy](#) { [Strategy::keep_same](#), [Strategy::re_mutate](#), [Strategy::remove](#), [Strategy::none](#) }
Replacing or remove individuals strategies during mutation.

Functions

- `std::mt19937_64` [generator](#) (`rd()`)
Pseudo-random number generator.
- `template<typename F, typename C, template< typename > class S, typename T >`
`std::vector< T > solve (const F &f, const C &c, const S< T > &solver_struct, const std::string &problem_
name)`
[Solver](#) wrapper function, interface to solvers : free function used for benchmarks.

Variables

- `std::random_device` [rd](#)
Random device / Random number generator.
- `template<typename T >`
`const double inv_pi_sq = 1 / std::pow(boost::math::constants::pi<T>(), 2)`
Inverse square of pi constant.
- `template<typename T >`
`const double inv_pi_sq_2 = 1 / std::pow(boost::math::constants::pi<T>(), 2)`
Inverse square of pi constant.

6.2.1 Detailed Description

Evolutionary Algorithms.

6.2.2 Enumeration Type Documentation

6.2.2.1 Strategy

```
enum ea::Strategy [strong]
```

Replacing or remove individuals strategies during mutation.

Enumerator

keep_same	
re_mutate	
remove	
none	

Definition at line 15 of file geneticalgo.h.

```
15 { keep_same, re_mutate, remove, none };
```

6.2.3 Function Documentation

6.2.3.1 generator()

```
ea::generator (
    rd() )
```

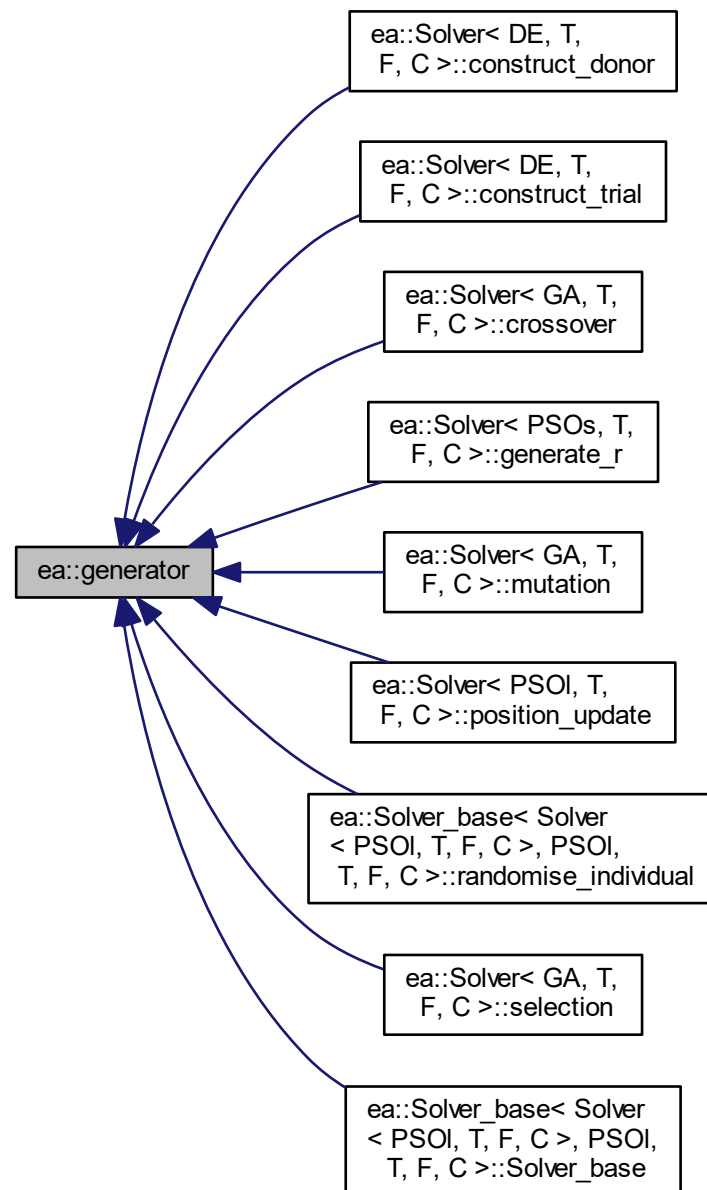
Pseudo-random number generator.

Returns

A random number

Referenced by `ea::Solver< DE, T, F, C >::construct_donor()`, `ea::Solver< DE, T, F, C >::construct_trial()`, `ea::Solver< GA, T, F, C >::crossover()`, `ea::Solver< PSOs, T, F, C >::generate_r()`, `ea::Solver< GA, T, F, C >::mutation()`, `ea::Solver< PSOI, T, F, C >::position_update()`, `ea::Solver_base< Solver< PSOI, T, F, C >, PSOI, T, F, C >::randomise_individual()`, `ea::Solver< GA, T, F, C >::selection()`, and `ea::Solver_base< Solver< PSOI, T, F, C >, PSOI, T, F, C >::Solver_base()`.

Here is the caller graph for this function:



6.2.3.2 solve()

```

template<typename F , typename C , template< typename > class S, typename T >
ea::solve (
    const F & f,
    const C & c,

```

```
const S< T > & solver_struct,
const std::string & problem_name )
```

Solver wrapper function, interface to solvers : free function used for benchmarks.

Parameters

<i>f</i>	The objective function
<i>c</i>	The constraints function
<i>solver_struct</i>	The parameter structure of the solver
<i>problem_name</i>	The name of the problem in std::string form. It is used to print results to file.

Returns

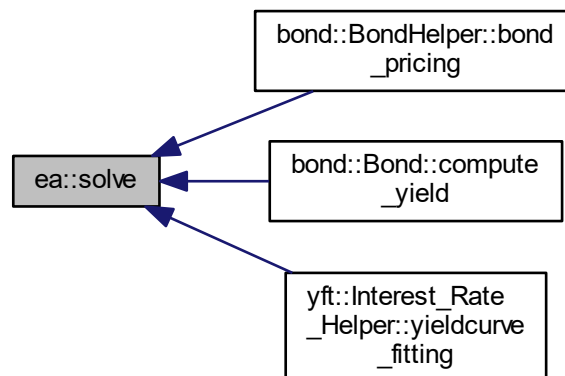
The solution vector

Definition at line 309 of file ealgorithm_base.h.

Referenced by `bond::BondHelper< T >::bond_pricing()`, `bond::Bond< T >::compute_yield()`, and `yft::Interest_Rate_Helper< T >::yieldcurve_fitting()`.

```
310 {
311     Solver<S, T, F, C> solver{ solver_struct, f, c };
312     return solver.solver_bench(problem_name);
313 }
```

Here is the caller graph for this function:



6.2.4 Variable Documentation

6.2.4.1 inv_pi_sq

```
template<typename T >
const double ea::inv_pi_sq = 1 / std::pow(boost::math::constants::pi<T>(), 2)
```

Inverse square of pi constant.

Definition at line 307 of file lbestpso.h.

6.2.4.2 inv_pi_sq_2

```
template<typename T >
const double ea::inv_pi_sq_2 = 1 / std::pow(boost::math::constants::pi<T>(), 2)
```

Inverse square of pi constant.

Definition at line 331 of file pso_sub_swarm.h.

6.2.4.3 rd

```
std::random_device ea::rd
```

Random device / Random number generator.

Definition at line 85 of file ealgorithm_base.h.

6.3 irr Namespace Reference

Internal Rate of Return (IRR) namespace.

Functions

- template<typename T >
T [compute_discount_factor](#) (const T &r, const T &period, const [DF_type](#) &df_type)
Calculates discount factors.
- template<typename T >
bool [constraints_irr](#) (const std::vector< T > &solution, const [Constraints_type](#) &constraints_type)
Constraints function for Internal Rate of Return.
- template<typename T >
T [compute_pv](#) (const T &r, const T &nominal_value, const std::vector< T > &cash_flows, const std::vector< T > &time_periods, const [DF_type](#) &df_type)
Returns the present value of an investment.
- template<typename T >
T [penalty_irr](#) (const T &r)
Penalty function for IRR.
- template<typename T >
T [fitness_irr](#) (const std::vector< T > &solution, const T &price, const T &nominal_value, const std::vector< T > &cash_flows, const std::vector< T > &time_periods, const [DF_type](#) &df_type, const bool &use_penalty_method)
This is the fitness function for finding the internal rate of return of a bond, in this case it is equal to its yield to maturity.

6.3.1 Detailed Description

Internal Rate of Return (IRR) namespace.

6.3.2 Function Documentation

6.3.2.1 compute_discount_factor()

```
template<typename T >
irr::compute_discount_factor (
    const T & r,
    const T & period,
    const DF_type & df_type )
```

Calculates discount factors.

Parameters

<i>r</i>	Rate
<i>period</i>	The period the rate was recorded
<i>df_type</i>	The method used to calculate the discount factor

Returns

The discount factor

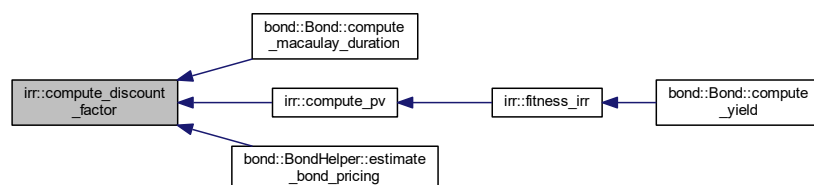
Definition at line 25 of file irr.h.

References utilities::exp, and utilities::frac.

Referenced by bond::Bond< T >::compute_macaulay_duration(), compute_pv(), and bond::BondHelper< T >::estimate_bond_pricing().

```
26     {
27         switch (df_type)
28         {
29             case (DF_type::frac): return 1 / std::pow((1 + r), period);
30             case (DF_type::exp):  return std::exp(-r * period);
31             default: std::abort();
32         }
33     }
```

Here is the caller graph for this function:



6.3.2.2 compute_pv()

```
template<typename T >
irr::compute_pv (
    const T & r,
    const T & nominal_value,
    const std::vector< T > & cash_flows,
    const std::vector< T > & time_periods,
    const DF_type & df_type )
```

Returns the present value of an investment.

Parameters

<i>r</i>	Internal Rate of Return
<i>nominal_value</i>	The nominal value of the investment
<i>cash_flows</i>	The cash flows of the investment
<i>time_periods</i>	The time periods that correspond to the cash flows of the investment
<i>df_type</i>	The method used to calculate the discount factor

Returns

The present value of the investment

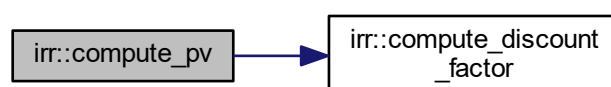
Definition at line 74 of file irr.h.

References `compute_discount_factor()`.

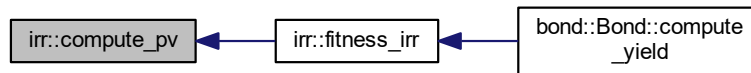
Referenced by `fitness_irr()`.

```
75     {
76         assert(cash_flows.size() == time_periods.size());
77         const size_t& num_time_periods = time_periods.size();
78         T sum = 0.0;
79         for (size_t i = 0; i < num_time_periods; ++i)
80         {
81             sum = sum + cash_flows[i] * compute_discount_factor(r, time_periods[i],
82 df_type);
83         }
84         return sum + nominal_value * compute_discount_factor(r, time_periods.back(),
df_type);
85     }
```

Here is the call graph for this function:



Here is the caller graph for this function:



6.3.2.3 constraints_irr()

```

template<typename T >
irr::constraints_irr (
    const std::vector< T > & solution,
    const Constraints_type & constraints_type )
  
```

Constraints function for Internal Rate of Return.

Parameters

<i>solution</i>	Internal Rate of Return candindate solution
<i>constraints_type</i>	Type of constraints used

Returns

True if constraints are satisfied, false otherwise

Definition at line 42 of file irr.h.

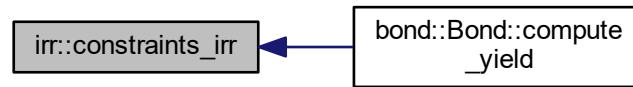
References `utilities::none`, `utilities::normal`, and `utilities::tight`.

Referenced by `bond::Bond< T >::compute_yield()`.

```

43     {
44         switch (constraints_type)
45         {
46             case(Constraints_type::normal):
47             {
48                 const T& r = solution[0];
49                 if (r > 0 && r < 1)
50                 {
51                     return true;
52                 }
53                 else
54                 {
55                     return false;
56                 }
57             }
58             case(Constraints_type::tight): return true;
59             case(Constraints_type::none): return true;
60             default: std::abort();
61         }
62     }
  
```


Here is the caller graph for this function:



6.3.2.4 fitness_irr()

```

template<typename T >
irr::fitness_irr (
    const std::vector< T > & solution,
    const T & price,
    const T & nominal_value,
    const std::vector< T > & cash_flows,
    const std::vector< T > & time_periods,
    const DF_type & df_type,
    const bool & use_penalty_method )
  
```

This is the fitness function for finding the internal rate of return of a bond, in this case it is equal to its yield to maturity.

Parameters

<i>solution</i>	Internal Rate of Return candindate solution
<i>price</i>	The present value of the investment
<i>nominal_value</i>	The nominal value of the investment
<i>cash_flows</i>	The cash flows of the investment
<i>time_periods</i>	The time periods that correspond to the cash flows of the investment
<i>df_type</i>	The method used to calculate the discount factor
<i>use_penalty_method</i>	Whether to use the penalty method defined for IRR or not

Returns

The fitness cost of IRR

Definition at line 116 of file irr.h.

References `compute_pv()`, and `penalty_irr()`.

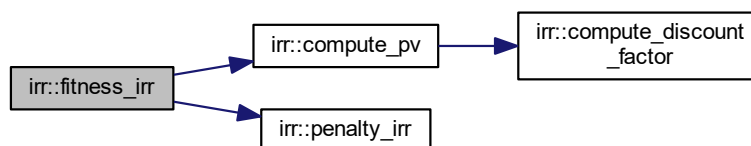
Referenced by `bond::Bond< T >::compute_yield()`.

```

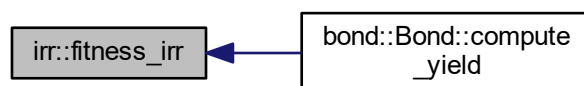
118     {
119         T sum_of_squares = 0;
120         sum_of_squares = sum_of_squares + std::pow(price - compute_pv(solution[0], nominal_value,
121         cash_flows, time_periods, df_type), 2);
122         if (use_penalty_method)
123         {
124             return sum_of_squares + penalty_irr(solution[0]);
125         }
126         else
127         {
128             return sum_of_squares;
129         }
130     }

```

Here is the call graph for this function:



Here is the caller graph for this function:



6.3.2.5 penalty_irr()

```

template<typename T >
irr::penalty_irr (
    const T & r )

```

Penalty function for IRR.

Parameters

<i>r</i>	Candidate solution for the Internal Rate of Return
----------	--

Returns

A penalty value, if constraints are not satisfied

Definition at line 92 of file irr.h.

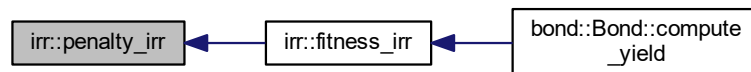
Referenced by `fitness_irr()`.

```

93     {
94         T sum = 0;
95         const T C = 1000;
96         if (r < 0 || r > 1)
97         {
98             sum = sum + C * std::pow(std::abs(r), 2);
99         }
100         return sum;
101     }

```

Here is the caller graph for this function:



6.4 nss Namespace Reference

Nelson-Siegel-Svensson (NSS) model namespace.

Functions

- `template<typename T >`
`bool constraints_svensson` (`const std::vector< T > &solution`, `const Constraints_type &constraints_type`)
Constraints function for the NSS model.
- `template<typename T >`
`T svensson` (`const std::vector< T > &solution`, `const T &m`)
Spot interest rate at term m using the NSS model.
- `template<typename T >`
`T penalty_svensson` (`const std::vector< T > &solution`)
Penalty function for NSS.

6.4.1 Detailed Description

Nelson-Siegel-Svensson (NSS) model namespace.

6.4.2 Function Documentation

6.4.2.1 constraints_svensson()

```
template<typename T >
nss::constraints_svensson (
    const std::vector< T > & solution,
    const Constraints_type & constraints_type )
```

Constraints function for the NSS model.

Parameters

<i>solution</i>	NSS parameters candindate solution
<i>constraints_type</i>	Type of constraints used

Returns

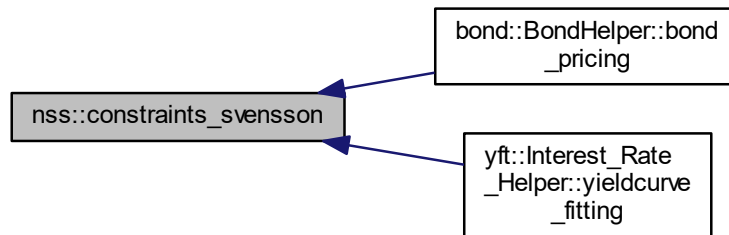
True if constraints are satisfied, false otherwise

Definition at line 18 of file svensson.h.

Referenced by `bond::BondHelper< T >::bond_pricing()`, and `yft::Interest_Rate_Helper< T >::yieldcurve_fitting()`.

```
19     {
20         switch (constraints_type)
21         {
22             case(Constraints_type::normal):
23             {
24                 const T& b0 = solution[0];
25                 const T& b1 = solution[1];
26                 const T& tau1 = solution[4];
27                 const T& tau2 = solution[5];
28                 if (b0 > 0 && b0 + b1 > 0 && tau1 > 0 && tau2 > 0)
29                 {
30                     return true;
31                 }
32                 else
33                 {
34                     return false;
35                 }
36             }
37             case(Constraints_type::tight):
38             {
39                 const T& b0 = solution[0];
40                 const T& b1 = solution[1];
41                 const T& b2 = solution[2];
42                 const T& b3 = solution[3];
43                 const T& tau1 = solution[4];
44                 const T& tau2 = solution[5];
45                 if ((b0 > 0 && b0 < 15)
46                     && (b1 > -15 && b1 < 30)
47                     && (b2 > -30 && b2 < 30)
48                     && (b3 > -30 && b3 < 30)
49                     && (tau1 > 0 && tau1 < 2.5)
50                     && (tau2 > 2.5 && tau2 < 5.5))
51                 {
52                     return true;
53                 }
54                 else
55                 {
56                     return false;
57                 }
58             }
59             case(Constraints_type::none): return true;
60             default: std::abort();
61         }
62     }
```

Here is the caller graph for this function:



6.4.2.2 penalty_svensson()

```
template<typename T >
nss::penalty_svensson (
    const std::vector< T > & solution )
```

Penalty function for NSS.

Parameters

<i>solution</i>	Candidate solution for the parameters of NSS
-----------------	--

Returns

A penalty value, if constraints are not satisfied

Definition at line 99 of file `svensson.h`.

Referenced by `bond::BondHelper< T >::fitness_bond_pricing_prices()`, `bond::BondHelper< T >::fitness_bond_pricing_yields()`, and `yft::Interest_Rate_Helper< T >::fitness_yield_curve_fitting()`.

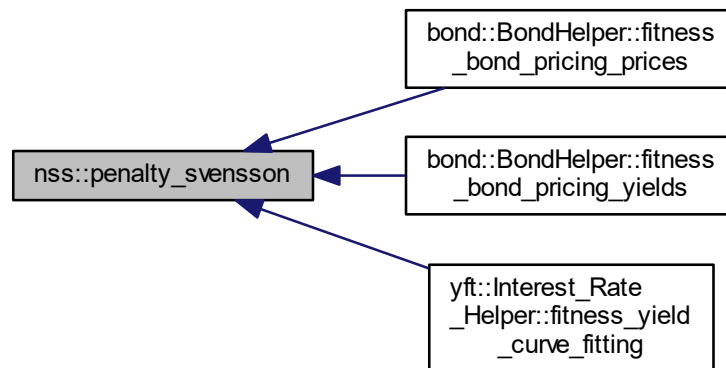
```
100     {
101         T sum = 0;
102         const T& b0 = solution[0];
103         const T& b1 = solution[1];
104         const T& b2 = solution[2];
105         const T& b3 = solution[3];
106         const T& tau1 = solution[4];
107         const T& tau2 = solution[5];
108         const T C = 100000;
109         if (b0 < 0 || b0 > 15)
110         {
111             sum = sum + C * std::pow(std::abs(b0), 2);
112         }
113         if (b0 + b1 < 0)
114         {
115             sum = sum + C * std::pow(std::abs(b0 + b1), 2);
116         }
117         if (b1 < -15 || b1 > 30)
```

```

118     {
119         sum = sum + C * std::pow(std::abs(b1), 2);
120     }
121     if (b2 < -30 || b2 > 30)
122     {
123         sum = sum + C * std::pow(std::abs(b1), 2);
124     }
125     if (b3 < -30 || b3 > 30)
126     {
127         sum = sum + C * std::pow(std::abs(b1), 2);
128     }
129     if (tau1 < 0 || tau1 > 2.5)
130     {
131         sum = sum + C * std::pow(std::abs(tau2), 2);
132     }
133     if (tau2 < 2.5 || tau2 > 5.5)
134     {
135         sum = sum + C * std::pow(std::abs(tau2), 2);
136     }
137     return sum;
138 }

```

Here is the caller graph for this function:



6.4.2.3 `svensson()`

```

template<typename T >
nss::svensson (
    const std::vector< T > & solution,
    const T & m )

```

Spot interest rate at term `m` using the NSS model.

Parameters

<i>solution</i>	Candidate solution for the parameters of NSS
<i>m</i>	The term at which the spot interest rate is recorded

Returns

The spot interest rate at term m

Definition at line 71 of file `svensson.h`.

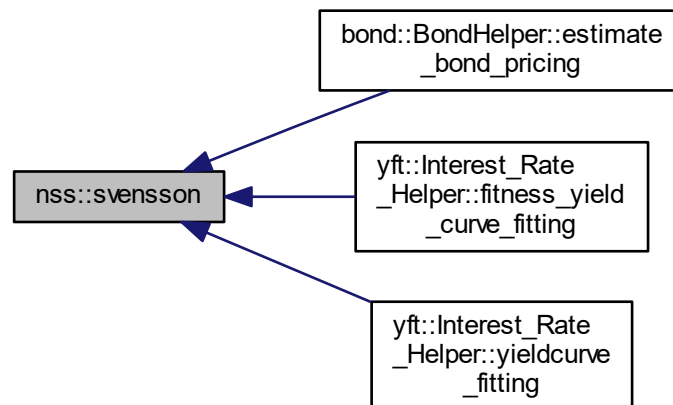
Referenced by `bond::BondHelper< T >::estimate_bond_pricing()`, `yft::Interest_Rate_Helper< T >::fitness_yield_curve_fitting()`, and `yft::Interest_Rate_Helper< T >::yieldcurve_fitting()`.

```

72     {
73         const T& b0 = solution[0];
74         const T& b1 = solution[1];
75         const T& b2 = solution[2];
76         const T& b3 = solution[3];
77         const T& tau1 = solution[4];
78         const T& tau2 = solution[5];
79         if (m == 0)
80         {
81             return b0 + b1;
82         }
83         else
84         {
85             T result = b0;
86             result = result + b1 * ((1 - (std::exp(-m / tau1))) / (m / tau1));
87             result = result + b2 * (((1 - (std::exp(-m / tau1))) / (m / tau1)) - std::exp(-m / tau1));
88             result = result + b3 * (((1 - (std::exp(-m / tau2))) / (m / tau2)) - std::exp(-m / tau2));
89             return result;
90         }
91     }

```

Here is the caller graph for this function:



6.5 utilities Namespace Reference

Utilities namespace.

Enumerations

- enum `DF_type` { `DF_type::frac`, `DF_type::exp` }
Enumeration for discount factor types/methods.
- enum `Constraints_type` { `Constraints_type::normal`, `Constraints_type::tight`, `Constraints_type::none` }
Enumeration for types of constraints for the optimisation problems.

Functions

- `template<typename T>`
`std::ostream & operator<< (std::ostream &stream, const std::vector< T > &vector)`
Overload the operator << for printing vectors.

6.5.1 Detailed Description

Utilities namespace.

6.5.2 Enumeration Type Documentation

6.5.2.1 Constraints_type

```
enum utilities::Constraints_type [strong]
```

Enumeration for types of constraints for the optimisation problems.

Enumerator

normal	Use the normal constraints
tight	Use tighter constraints
none	Ignore constraints

Definition at line 25 of file utilities.h.

```
26     {
27         normal,
28         tight,
29         none
30     };
```

6.5.2.2 DF_type

```
enum utilities::DF_type [strong]
```

Enumeration for discount factor types/methods.

Enumerator

frac	Use fractional form to calculate the discount factor
exp	Use the exponential form to calculate the discount factor

Definition at line 17 of file utilities.h.

```

18     {
19         frac,
20         exp
21     };

```

6.5.3 Function Documentation

6.5.3.1 operator<<()

```

template<typename T >
utilities::operator<< (
    std::ostream & stream,
    const std::vector< T > & vector )

```

Overload the operator << for printing vectors.

Parameters

<i>stream</i>	An out stream
<i>vector</i>	A vector

Returns

An out stream of the vector in the form [...]

Definition at line 38 of file utilities.h.

```

39     {
40         if (!vector.empty())
41         {
42             stream << "[ ";
43             std::copy(vector.begin(), vector.end(), std::ostream_iterator<T>(stream, " "));
44             stream << "]";
45         }
46         return stream;
47     }

```

6.6 yft Namespace Reference

Yield Curve Fitting namespace.

Classes

- struct [Interest_Rate](#)
Structure for interest rates.
- class [Interest_Rate_Helper](#)
A class for the yield-curve-fitting problem.

Functions

- `template<typename T>`
`std::vector< Interest_Rate< T > > read_ir_from_file (const std::string &filename)`
Reads the interest rates and periods from file and constructs a vector of interest rate structs.

6.6.1 Detailed Description

Yield Curve Fitting namespace.

6.6.2 Function Documentation

6.6.2.1 `read_ir_from_file()`

```
template<typename T >
yft::read_ir_from_file (
    const std::string & filename )
```

Reads the interest rates and periods from file and constructs a vector of interest rate structs.

Parameters

<i>filename</i>	The name of the input file as an std::string
-----------------	--

Returns

A vector of [Interest_Rate](#) objects

Definition at line 45 of file `yield_curve_fitting.h`.

```
46     {
47         std::vector<Interest_Rate<T>> ir_vec;
48         std::ifstream input(filename);
49         for (std::string line; getline(input, line); )
50         {
51             T period;
52             T rate;
53             std::stringstream stream(line);
54             stream >> period >> rate;
55             const Interest_Rate<T> ir{ period, rate };
56             ir_vec.push_back(ir);
57         }
58         return ir_vec;
59     }
```

Chapter 7

Class Documentation

7.1 `bond::Bond< T >` Class Template Reference

`Bond` Class definition.

```
#include <bond.h>
```

Public Member Functions

- `Bond` (const T &i_coupon_percentage, const T &i_price, const T &i_nominal_value, const T &i_frequency, std::string &i_settlement_date, const std::string &i_maturity_date)

Constructor.

- template<typename S >
T `compute_yield` (const T &i_price, const S &solver, const `DF_type` &df_type) const

Calculates the yield-to-maturity using the supplied solver.

- template<typename S >
T `compute_yield` (const T &i_price, const S &solver, const `DF_type` &df_type, const std::string &bonds_↵ identifier) const

Calculates the yield-to-maturity using the supplied solver and passes the bond identifier to the solver.

- T `compute_macaulay_duration` (const `DF_type` &df_type) const

Calculates the Macaulay duration of the bond.

Private Member Functions

- std::vector< T > `compute_cash_flows` ()

Calculate the cash flows of the bond.

Private Attributes

- const T [coupon_percentage](#)
Bond's annual coupon rate.
- const T [price](#)
Bond's price.
- const T [nominal_value](#)
Bond's face value.
- const T [frequency](#)
Bond's coupon payment frequency.
- const T [coupon_value](#)
This is the annual coupon divided by the frequency.
- std::vector< T > [time_periods](#)
Coupon payment periods.
- std::vector< T > [cash_flows](#)
A vector with all the coupon payments corresponding to time periods.
- date::sys_days [settlement_date](#)
Settlement date of the bond.
- date::sys_days [maturity_date](#)
Maturity date of the bond.
- T [yield](#)
Yield-to-maturity of the bond.
- T [duration](#)
Macaulay duration of the bond.

Friends

- class [BondHelper](#)< T >
Friend class [BondHelper](#).

7.1.1 Detailed Description

```
template<typename T>
class bond::Bond< T >
```

[Bond](#) Class definition.

Definition at line 32 of file bond.h.

7.1.2 Constructor & Destructor Documentation

7.1.2.1 Bond()

```
template<typename T >
bond::Bond< T >::Bond (
    const T & i_coupon_percentage,
    const T & i_price,
    const T & i_nominal_value,
    const T & i_frequency,
    std::string & i_settlement_date,
    const std::string & i_maturity_date ) [inline]
```

Constructor.

Parameters

<i>i_coupon_percentage</i>	The coupon rate in %
<i>i_price</i>	The price of the bond
<i>i_nominal_value</i>	The nominal value of the bond
<i>i_frequency</i>	The frequency of coupon payments per year
<i>i_settlement_date</i>	The date the bond was bought
<i>i_maturity_date</i>	The date the bond expires

Returns

A Bond<T> object

Definition at line 48 of file bond.h.

```

49                                     :
50         coupon_percentage{ i_coupon_percentage },
51         price{ i_price },
52         nominal_value{ i_nominal_value },
53         frequency{ i_frequency },
54         coupon_value{ coupon_percentage *
nominal_value / frequency },
55         yield{ 0 },
56         duration{ 0 }
57     {
58         assert(price > 0);
59         assert(coupon_percentage > 0 && coupon_percentage < 1);
60         assert(nominal_value > 0);
61         assert(frequency > 0);
62         std::tm t1 {};
63         std::tm t2 {};
64         std::stringstream s1;
65         std::stringstream s2;
66         s1 << i_settlement_date;
67         s2 << i_maturity_date;
68         s1 >> std::get_time(&t1, "%Y-%m-%d");
69         s2 >> std::get_time(&t2, "%Y-%m-%d");
70         settlement_date = date::year(t1.tm_year) / (t1.tm_mon+1) / t1.tm_mday;
71         maturity_date = date::year(t2.tm_year) / (t2.tm_mon+1) / t2.tm_mday;
72         cash_flows = compute_cash_flows();
73         time_periods.resize(cash_flows.size());
74         for (size_t i = 0; i < time_periods.size(); ++i)
75         {
76             time_periods[i] = static_cast<T>(i + 1) / frequency;
77         }
78     }

```

7.1.3 Member Function Documentation

7.1.3.1 compute_cash_flows()

```

template<typename T >
std::vector< T > bond::Bond< T >::compute_cash_flows ( ) [private]

```

Calculate the cash flows of the bond.

Returns

The cash flows of the bonds (coupon payments)

Definition at line 133 of file bond.h.

```

134     {
135         assert(settlement_date < maturity_date);
136         const T number_of_days_coupon = 365 / frequency;
137         const auto days_difference = (maturity_date -
settlement_date).count();
138         const auto time_periods = static_cast<T>(days_difference) / number_of_days_coupon;
139         const size_t num_time_periods = static_cast<size_t>(std::ceil(
time_periods));
140         std::vector<T> cash_flows(num_time_periods);
141         for (auto& p : cash_flows)
142         {
143             p = coupon_value;
144         }
145         return cash_flows;
146     }

```

7.1.3.2 compute_macaulay_duration()

```

template<typename T >
T bond::Bond< T >::compute_macaulay_duration (
    const DF_type & df_type ) const

```

Calculates the Macaulay duration of the bond.

Parameters

<i>df_type</i>	The type of discount factor method
----------------	------------------------------------

Returns

The Macaulay Duration of the bond

Discount factor

Prest cash flows

Present value

Macaulay duration

Definition at line 174 of file bond.h.

References irr::compute_discount_factor().

```

175     {
176         assert(yield > 0 && yield < 1);
177         assert(cash_flows.size() > 0);
178         assert(nominal_value > 0);
179         assert(frequency > 0);
181         T discount_factor = 0.0;
183         T denominator = 0.0;
185         T pv = 0.0;

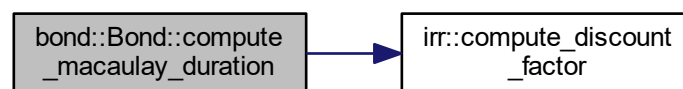
```

```

187         T numerator = 0.0;
188         for (size_t i = 0; i < time_periods.size(); ++i)
189         {
190             discount_factor = compute_discount_factor(
191                 yield, time_periods[i], df_type);
192             pv = coupon_value * discount_factor;
193             numerator = numerator + pv * time_periods[i];
194             denominator = denominator + pv;
195         }
196         pv = nominal_value * discount_factor;
197         numerator = numerator + pv * time_periods.back();
198         denominator = denominator + pv;
199         T duration = numerator / denominator;
200         return duration;

```

Here is the call graph for this function:



7.1.3.3 compute_yield() [1/2]

```

template<typename T >
template<typename S >
T bond::Bond< T >::compute_yield (
    const T & i_price,
    const S & solver,
    const DF_type & df_type ) const

```

Calculates the yield-to-maturity using the supplied solver.

Parameters

<i>i_price</i>	The price of the bond
<i>solver</i>	The parameter structure of the solver that is going to be used to estimate the yield of maturity
<i>df_type</i>	The type of discount factor method

Returns

The yield-to-maturity of the bond

Definition at line 150 of file bond.h.

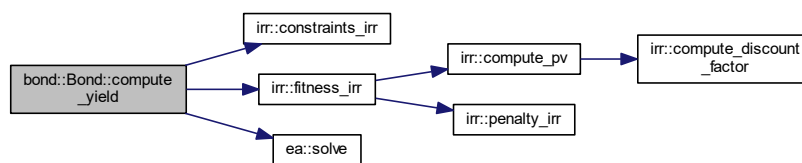
References irr::constraints_irr(), irr::fitness_irr(), and ea::solve().

```

151     {
152         assert(solver.ndv == 1);
153         auto f = [&,use_penalty_method = solver.use_penalty_method](const auto& solution) { return
fitness_irr(solution, i_price, nominal_value, cash_flows,
time_periods, df_type, use_penalty_method); };
154         auto c = [&,constraints_type = solver.constraints_type](const auto& solution) { return
constraints_irr(solution, constraints_type); };
155         auto res = solve(f, c, solver, "YTM");
156         T yield = res[0];
157         return yield;
158     }

```

Here is the call graph for this function:



7.1.3.4 compute_yield() [2/2]

```

template<typename T >
template<typename S >
T bond::Bond< T >::compute_yield (
    const T & i_price,
    const S & solver,
    const DF_type & df_type,
    const std::string & bonds_identifier ) const

```

Calculates the yield-to-maturity using the supplied solver and passes the bond identifier to the solver.

Parameters

<i>i_price</i>	The price of the bond
<i>solver</i>	The parameter structure of the solver that is going to be used to estimate the yield of maturity
<i>df_type</i>	The type of discount factor method
<i>bonds_identifier</i>	An identifier for the bond in std::string form

Returns

The yield-to-maturity of the bond

Definition at line 162 of file bond.h.

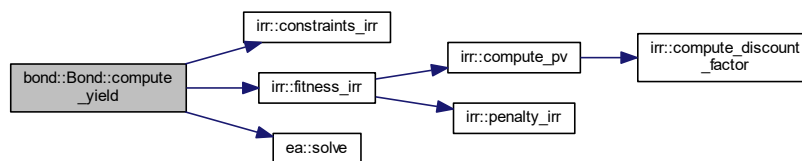
References irr::constraints_irr(), irr::fitness_irr(), and ea::solve().


```

163     {
164         assert(solver.ndv == 1);
165         auto f = [&, use_penalty_method = solver.use_penalty_method](const auto& solution) { return
fitness_irr(solution, i_price, nominal_value, cash_flows,
time_periods, df_type, use_penalty_method); };
166         auto c = [&, constraints_type = solver.constraints_type](const auto& solution) { return
constraints_irr(solution, constraints_type); };
167         std::string problem = "YTM";
168         auto res = solve(f, c, solver, problem.append(bonds_identifier));
169         T yield = res[0];
170         return yield;
171     }

```

Here is the call graph for this function:



7.1.4 Friends And Related Function Documentation

7.1.4.1 BondHelper< T >

```

template<typename T >
friend class BondHelper< T > [friend]

```

Friend class [BondHelper](#).

Definition at line 35 of file bond.h.

7.1.5 Member Data Documentation

7.1.5.1 cash_flows

```

template<typename T >
std::vector<T> bond::Bond< T >::cash_flows [private]

```

A vector with all the coupon payments corresponding to time periods.

Definition at line 116 of file bond.h.

7.1.5.2 coupon_percentage

```
template<typename T >
const T bond::Bond< T >::coupon_percentage [private]
```

Bond's annual coupon rate.

Definition at line 104 of file bond.h.

7.1.5.3 coupon_value

```
template<typename T >
const T bond::Bond< T >::coupon_value [private]
```

This is the annual coupon divided by the frequency.

Definition at line 112 of file bond.h.

7.1.5.4 duration

```
template<typename T >
T bond::Bond< T >::duration [private]
```

Macaulay duration of the bond.

Definition at line 124 of file bond.h.

7.1.5.5 frequency

```
template<typename T >
const T bond::Bond< T >::frequency [private]
```

Bond's coupon payment frequency.

Definition at line 110 of file bond.h.

7.1.5.6 maturity_date

```
template<typename T >
date::sys_days bond::Bond< T >::maturity_date [private]
```

Maturity date of the bond.

Definition at line 120 of file bond.h.

7.1.5.7 `nominal_value`

```
template<typename T >
const T bond::Bond< T >::nominal_value [private]
```

`Bond`'s face value.

Definition at line 108 of file `bond.h`.

7.1.5.8 `price`

```
template<typename T >
const T bond::Bond< T >::price [private]
```

`Bond`'s price.

Definition at line 106 of file `bond.h`.

7.1.5.9 `settlement_date`

```
template<typename T >
date::sys_days bond::Bond< T >::settlement_date [private]
```

Settlement date of the bond.

Definition at line 118 of file `bond.h`.

7.1.5.10 `time_periods`

```
template<typename T >
std::vector<T> bond::Bond< T >::time_periods [private]
```

Coupon payment periods.

Definition at line 114 of file `bond.h`.

7.1.5.11 `yield`

```
template<typename T >
T bond::Bond< T >::yield [private]
```

Yield-to-maturity of the bond.

Definition at line 122 of file `bond.h`.

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

- [bond.h](#)

7.2 `bond::BondHelper< T >` Class Template Reference

A class for the bond pricing problem as well as finding the yield-to-maturities of bonds.

```
#include <bond.h>
```

Public Member Functions

- `BondHelper` (const std::vector< `Bond`< T >> &i_bonds, const `DF_type` &i_df_type)
Constructor.
- template<typename S >
std::vector< T > `set_init_nss_params` (const S &solver)
This method sets the nss initial svensson parameters by computing the bond yields-to-maturity and Macaulay durations.
- template<typename S1 , typename S2 >
void `bond_pricing` (const S1 &solver, const S2 &solver_irr, const `Bond_pricing_type` &bond_pricing_type)
This methods solves the bond pricing problem using prices or yields and the supplied solver.
- template<typename S >
void `print_bond_pricing_results` (const std::vector< T > &res, const S &solver_irr)
This method prints to screen the bond pricing results.

Private Member Functions

- T `estimate_bond_pricing` (const std::vector< T > &solution, const T &coupon_value, const T &nominal_value, const std::vector< T > &time_periods)
Returns the bond prices using the estimated spot interest rates computed with svensson.
- template<typename S >
T `fitness_bond_pricing_yields` (const std::vector< T > &solution, const S &solver_irr, const bool &use_penalty_method)
This is the fitness function for bond pricing using the bonds' yields-to-maturity.
- T `fitness_bond_pricing_prices` (const std::vector< T > &solution, const bool &use_penalty_method)
This is the fitness function for bond pricing using the bonds' prices.

Private Attributes

- std::vector< `Bond`< T >> `bonds`
Vector of bonds.
- const `DF_type` `df_type`
Discount Factor type.

7.2.1 Detailed Description

```
template<typename T>
class bond::BondHelper< T >
```

A class for the bond pricing problem as well as finding the yield-to-maturities of bonds.

Definition at line 26 of file bond.h.

7.2.2 Constructor & Destructor Documentation

7.2.2.1 BondHelper()

```
template<typename T >
bond::BondHelper< T >::BondHelper (
    const std::vector< Bond< T >> & i_bonds,
    const DF_type & i_df_type = DF_type::exp ) [inline]
```

Constructor.

Parameters

<i>i_bonds</i>	A vector of Bond<T> objects
<i>i_df_type</i>	The type of discount factor method

Returns

A BondHelper<T> object

Definition at line 66 of file bondhelper.h.

```
66                                     :
67         bonds( i_bonds ),
68         df_type( i_df_type )
69     {};
```

7.2.3 Member Function Documentation

7.2.3.1 bond_pricing()

```
template<typename T >
template<typename S1 , typename S2 >
void bond::BondHelper< T >::bond_pricing (
    const S1 & solver,
    const S2 & solver_irr,
    const Bond_pricing_type & bond_pricing_type )
```

This methods solves the bond pricing problem using prices or yields and the supplied solver.

Parameters

<i>solver</i>	The parameter structure of the solver that is going to be used for bond pricing
<i>solver_irr</i>	The parameter structure of the solver that is going to be used to estimate the yield of maturity
<i>bond_pricing_type</i>	Whether to use bond yields-to-maturities or bond prices to find the NSS parameters

Returns

void

Definition at line 235 of file bondhelper.h.

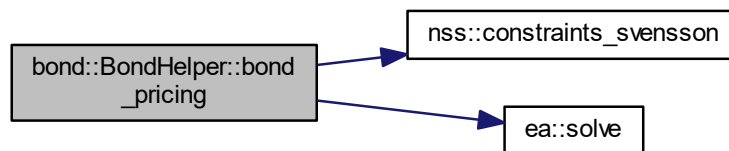
References `bond::bpp`, `bond::bpy`, `nss::constraints_svensson()`, and `ea::solve()`.

```

236     {
237         assert(solver.ndv == 6);
238         for (const auto& p : bonds)
239         {
240             assert(p.yield > 0 && p.yield < 1);
241             assert(p.duration > 0);
242         }
243         switch (bond_pricing_type)
244         {
245             case (Bond_pricing_type::bpp):
246             {
247                 auto f = [&, use_penalty_method = solver.use_penalty_method](const auto& solution) { return
248 fitness_bond_pricing_prices(solution, use_penalty_method); };
249                 auto c = [&, constraints_type = solver.constraints_type](const auto& solution) { return
250 constraints_svensson(solution, constraints_type); };
251                 std::cout << "Solving bond pricing using bond prices..." << "\n";
252                 auto res = solve(f, c, solver, "BPP");
253                 print_bond_pricing_results(res, solver_irr);
254                 break;
255             }
256             case (Bond_pricing_type::bpy):
257             {
258                 auto f = [&, use_penalty_method = solver.use_penalty_method](const auto& solution) { return
259 fitness_bond_pricing_yields(solution, solver_irr, use_penalty_method); };
260                 auto c = [&, constraints_type = solver.constraints_type](const auto& solution) { return
261 constraints_svensson(solution, constraints_type); };
262                 std::cout << "Solving bond pricing using bond yields..." << "\n";
263                 auto res = solve(f, c, solver, "BPY");
264                 print_bond_pricing_results(res, solver_irr);
265             }
266         }
267     }

```

Here is the call graph for this function:



7.2.3.2 estimate_bond_pricing()

```

template<typename T >
T bond::BondHelper< T >::estimate_bond_pricing (
    const std::vector< T > & solution,
    const T & coupon_value,
    const T & nominal_value,
    const std::vector< T > & time_periods ) [private]

```

Returns the bond prices using the estimated spot interest rates computed with `svensson`.

Parameters

<i>solution</i>	NSS parameters candidate solution
<i>coupon_value</i>	The value of the coupon payment
<i>nominal_value</i>	The nominal value of the investment
<i>time_periods</i>	The time periods that correspond to the coupon payments

Returns

The price of the bond

Call `svensson` for period

Definition at line 178 of file `bondhelper.h`.

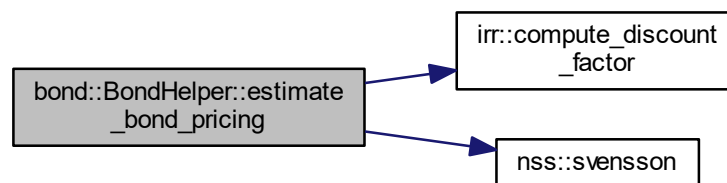
References `irr::compute_discount_factor()`, and `nss::svensson()`.

```

179     {
180         T sum = 0.0;
181         for (const auto& t : time_periods)
182         {
183             sum = sum + coupon_value * compute_discount_factor(
184                 svensson(solution, t), t, df_type);
185         }
186         T m = time_periods.back();
187         sum = sum + nominal_value * compute_discount_factor(
188             svensson(solution, m), m, df_type);
189         return sum;
190     }

```

Here is the call graph for this function:

7.2.3.3 `fitness_bond_pricing_prices()`

```

template<typename T >
T bond::BondHelper< T >::fitness_bond_pricing_prices (
    const std::vector< T > & solution,
    const bool & use_penalty_method ) [private]

```

This is the fitness function for bond pricing using the bonds' prices.

Parameters

<i>solution</i>	NSS parameters candindate solution
<i>use_penalty_method</i>	Whether to use the penalty method defined for NSS or not

Returns

The fitness cost of NSS for bond pricing

The sum of squares of errors between the actual bond price and the estimated price from `estimate_bond_pricing`

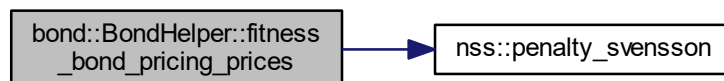
Definition at line 192 of file `bondhelper.h`.

References `nss::penalty_svensson()`.

```

193     {
194         T sum_of_squares = 0.0;
195         for (const auto& k : bonds)
196         {
197             T estimate = estimate_bond_pricing(solution, k.coupon_value, k.
nominal_value, k.time_periods);
198             sum_of_squares = sum_of_squares + std::pow((k.price/100 - estimate/100), 2) / std::sqrt(k.
duration);
199         }
200         if (use_penalty_method)
201         {
202             return sum_of_squares + penalty_svensson(solution);
203         }
204         else
205         {
206             return sum_of_squares;
207         }
208     }
209 }
```

Here is the call graph for this function:



7.2.3.4 fitness_bond_pricing_yields()

```

template<typename T >
template<typename S >
T bond::BondHelper< T >::fitness_bond_pricing_yields (
    const std::vector< T > & solution,
    const S & solver_irr,
    const bool & use_penalty_method ) [private]
```

This is the fitness function for bond pricing using the bonds' yields-to-maturity.

Parameters

<i>solution</i>	NSS parameters candindate solution
<i>solver_irr</i>	The parameter structure of the solver that is going to be used to estimate the yield of maturity
<i>use_penalty_method</i>	Whether to use the penalty method defined for NSS or not

Returns

The fitness cost of NSS for bond pricing

The sum of squares of errors between the actual bond yield to maturity and the estimated yield to maturity by svensson is used

Definition at line 213 of file bondhelper.h.

References `nss::penalty_svensson()`.

```

214     {
215         T sum_of_squares = 0;
216         for (const auto& k : bonds)
217         {
218             T estimate_price = estimate_bond_pricing(solution, k.coupon_value, k.
nominal_value, k.time_periods);
219             T estimate = k.compute_yield(estimate_price, solver_irr, df_type);
220             sum_of_squares = sum_of_squares + std::pow(k.yield - estimate, 2);
221         }
222         if (use_penalty_method)
223         {
224             return sum_of_squares + penalty_svensson(solution);
225         }
226         else
227         {
228             return sum_of_squares;
229         }
230     }
231 }
```

Here is the call graph for this function:



7.2.3.5 print_bond_pricing_results()

```

template<typename T >
template<typename S >
void bond::BondHelper< T >::print_bond_pricing_results (
    const std::vector< T > & res,
    const S & solver_irr )
```

This method prints to screen the bond pricing results.

Parameters

<i>res</i>	The solution vector of NSS parameters
<i>solver</i> ↵ <i>_irr</i>	The parameter structure of the solver that is going to be used to estimate the yield of maturity

Returns

void

Definition at line 267 of file bondhelper.h.

```

268     {
269         T error = 0;
270         //for (const auto& p : bonds)
271         //{
272             //T estimate_price = estimate_bond_pricing(res, p.coupon_value, p.nominal_value,
p.time_periods);
273             //T estimate = p.compute_yield(estimate_price, solver_irr, df_type);
274             //error = error + std::pow(estimate - p.yield, 2);
275             //std::cout << "Estimated yield: " << estimate << " Actual Yield: " << p.yield << "\n";
276             //}
277             //std::cout << "Yield Mean Squared Error: " << error << "\n";
278             error = 0;
279             for (const auto& p : bonds)
280             {
281                 error = error + std::pow(estimate_bond_pricing(res, p.coupon_value, p.
nominal_value, p.time_periods)/100 - p.price/100, 2);
282                 std::cout << "Estimated price: " << estimate_bond_pricing(res, p.
coupon_value, p.nominal_value, p.time_periods) << " Actual Price: " << p.price << "\n";
283             }
284             std::cout << "Price Mean Squared Error: " << error << "\n";
285         }

```

7.2.3.6 set_init_nss_params()

```

template<typename T >
template<typename S >
std::vector< T > bond::BondHelper< T >::set_init_nss_params (
    const S & solver )

```

This method sets the nss initial svensson parameters by computing the bond yields-to-maturity and Macaulay durations.

Parameters

<i>solver</i>	The parameter structure of the solver that is going to be used
---------------	--

Returns

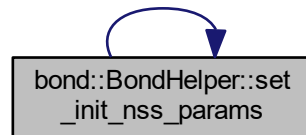
A vector of decision variables for NSS

Definition at line 69 of file bondhelper.h.

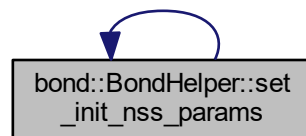
References bond::BondHelper< T >::set_init_nss_params().

Referenced by bond::BondHelper< T >::set_init_nss_params().

Here is the call graph for this function:



Here is the caller graph for this function:



7.2.4 Member Data Documentation

7.2.4.1 bonds

```
template<typename T >
std::vector<Bond<T> > bond::BondHelper< T >::bonds [private]
```

Vector of bonds.

Definition at line 93 of file bondhelper.h.

7.2.4.2 df_type

```
template<typename T >
const DF_type bond::BondHelper< T >::df_type [private]
```

Discount Factor type.

Definition at line 95 of file bondhelper.h.

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

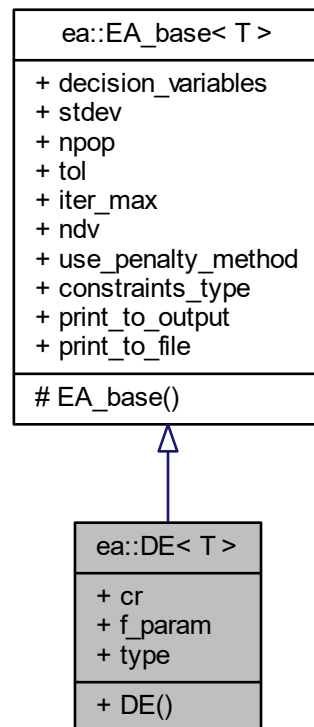
- [bond.h](#)
- [bondhelper.h](#)

7.3 ea::DE< T > Struct Template Reference

Differential Evolution Structure, used in the actual algorithm and for type deduction.

```
#include <differentialevo.h>
```

Inheritance diagram for ea::DE< T >:



Public Member Functions

- **DE** (const T &i_cr, const T &i_f_param, const std::vector< T > &i_decision_variables, const std::vector< T > &i_stdev, const size_t &i_npop, const T &i_tol, const size_t &i_iter_max, const bool &i_use_penalty_method, const [Constraints_type](#) &i_constraints_type, const bool &i_print_to_output, const bool &i_print_to_file)

Constructor.

Public Attributes

- const T [cr](#)
Crossover Rate.
- const T [f_param](#)
Mutation Scale Fuctor.
- const std::string [type](#) = "Differential Evolution"
Type of the algorithm.

Additional Inherited Members

7.3.1 Detailed Description

```
template<typename T>
struct ea::DE< T >
```

Differential Evolution Structure, used in the actual algorithm and for type deduction.

Definition at line 16 of file differentialevo.h.

7.3.2 Constructor & Destructor Documentation

7.3.2.1 DE()

```
template<typename T>
ea::DE< T >::DE (
    const T & i_cr,
    const T & i_f_param,
    const std::vector< T > & i_decision_variables,
    const std::vector< T > & i_stdev,
    const size_t & i_npop,
    const T & i_tol,
    const size_t & i_iter_max,
    const bool & i_use_penalty_method = false,
    const Constraints_type & i_constraints_type = Constraints_type::none,
    const bool & i_print_to_output = true,
    const bool & i_print_to_file = true ) [inline]
```

Constructor.

Parameters

<i>i_cr</i>	Crossover Rate
<i>i_f_param</i>	Mutation Scale Factor
<i>i_decision_variables</i>	The starting values of the decision variables
<i>i_stdev</i>	The standard deviation
<i>i_npop</i>	The population size
<i>i_tol</i>	The tolerance
<i>i_iter_max</i>	The maximum number of iterations
<i>i_use_penalty_method</i>	Whether to used penalties or not
<i>i_constraints_type</i>	What kind of constraints to use
<i>i_print_to_output</i>	Whether to print to terminal or not
<i>i_print_to_file</i>	Whether to print to a file or not

Returns

A DE<T> object

Definition at line 37 of file differentialevo.h.

References `ea::DE< T >::cr`, and `ea::DE< T >::f_param`.

```

40                                     :
41     EA_base<T>(i_decision_variables, i_stddev, i_npop, i_tol, i_iter_max, i_use_penalty_method,
i_constraints_type, i_print_to_output, i_print_to_file),
42     cr( i_cr ),
43     f_param( i_f_param )
44     {
45         assert( cr > 0 && cr <= 1 );
46         assert( f_param > 0 && f_param <= 1 );
47     }
```

7.3.3 Member Data Documentation

7.3.3.1 cr

```

template<typename T>
const T ea::DE< T >::cr
```

Crossover Rate.

Definition at line 49 of file differentialevo.h.

Referenced by `ea::DE< T >::DE()`, and `ea::Solver< DE, T, F, C >::display_parameters()`.

7.3.3.2 f_param

```

template<typename T>
const T ea::DE< T >::f_param
```

Mutation Scale Fuctor.

Definition at line 51 of file differentialevo.h.

Referenced by `ea::DE< T >::DE()`, and `ea::Solver< DE, T, F, C >::display_parameters()`.

7.3.3.3 type

```
template<typename T>
const std::string ea::DE< T >::type = "Differential Evolution"
```

Type of the algorithm.

Definition at line 53 of file differentialevo.h.

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

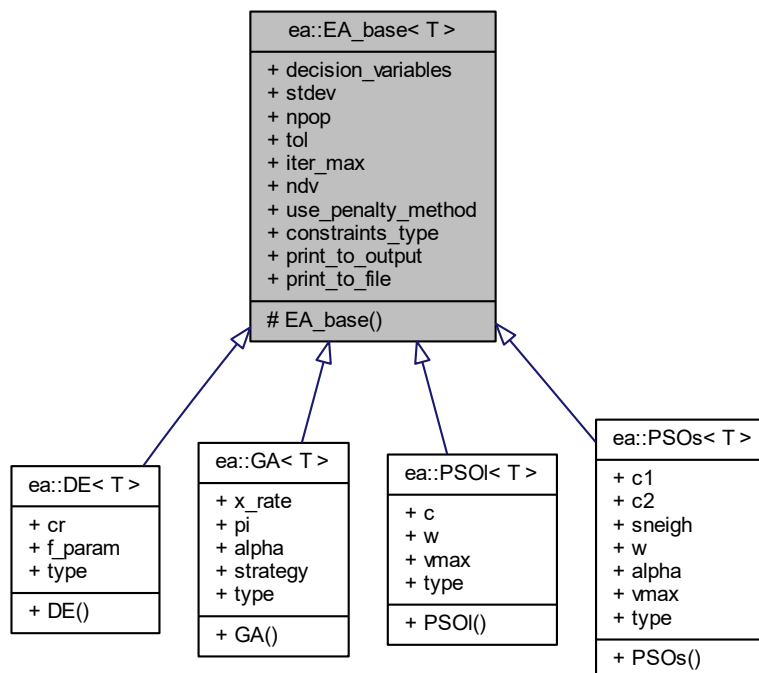
- [differentialevo.h](#)

7.4 ea::EA_base< T > Struct Template Reference

Evolutionary algorithm structure base.

```
#include <ealgorithm_base.h>
```

Inheritance diagram for ea::EA_base< T >:



Public Types

- using `fp_type` = T

The floating point number type used for type deduction.

Public Attributes

- const std::vector< T > [decision_variables](#)
Initial Decision Variables.
- const std::vector< T > [stdev](#)
Standard deviation of the decision variables.
- const size_t [npop](#)
Size of the population.
- const T [tol](#)
Tolerance.
- const size_t [iter_max](#)
Number of maximum iterations.
- const size_t [ndv](#)
Number of decision variables.
- const bool [use_penalty_method](#)
Use penalty function or not.
- const [Constraints_type](#) [constraints_type](#)
Constraints type.
- const bool [print_to_output](#)
Print to output or not.
- const bool [print_to_file](#)
Print to file or not.

Protected Member Functions

- [EA_base](#) (const std::vector< T > &i_decision_variables, const std::vector< T > &i_stdev, const size_t &i_npop, const T &i_tol, const size_t &i_iter_max, const bool &i_use_penalty_method, const [Constraints_type](#) &i_constraints_type, const bool &i_print_to_output, const bool &i_print_to_file)
Constructor.

7.4.1 Detailed Description

```
template<typename T>
struct ea::EA_base< T >
```

Evolutionary algorithm stucture base.

Definition at line 28 of file ealgorithm_base.h.

7.4.2 Member Typedef Documentation

7.4.2.1 fp_type

```
template<typename T >
using ea::EA_base< T >::fp_type = T
```

The floating point number type used for type deduction.

Definition at line 32 of file ealgorithm_base.h.

7.4.3 Constructor & Destructor Documentation

7.4.3.1 EA_base()

```
template<typename T >
ea::EA_base< T >::EA_base (
    const std::vector< T > & i_decision_variables,
    const std::vector< T > & i_stdev,
    const size_t & i_npop,
    const T & i_tol,
    const size_t & i_iter_max,
    const bool & i_use_penalty_method,
    const Constraints_type & i_constraints_type,
    const bool & i_print_to_output,
    const bool & i_print_to_file ) [inline], [protected]
```

Constructor.

Parameters

<i>i_decision_variables</i>	The starting values of the decision variables
<i>i_stdev</i>	The standard deviation
<i>i_npop</i>	The population size
<i>i_tol</i>	The tolerance
<i>i_iter_max</i>	The maximum number of iterations
<i>i_use_penalty_method</i>	Whether to used penalties or not
<i>i_constraints_type</i>	What kind of constraints to use
<i>i_print_to_output</i>	Whether to print to terminal or not
<i>i_print_to_file</i>	Whether to print to a file or not

Returns

A EA_base<T> object

Definition at line 68 of file ealgorithm_base.h.

```
70         : decision_variables{ i_decision_variables },
          stdev{ i_stdev }, npop{ i_npop }, tol{ i_tol }, iter_max{ i_iter_max },
          ndv{ i_decision_variables.size() },
71         use_penalty_method{ i_use_penalty_method },
          constraints_type{ i_constraints_type }, print_to_output{ i_print_to_output }
          , print_to_file{ i_print_to_file }
72     {
73         assert( decision_variables.size() > 0 );
74         assert( decision_variables.size() == stdev.size() );
75         for (const auto& p : stdev)
76         {
77             assert( p > 0 );
78         }
79         assert( npop > 0 );
80         assert( tol > 0 );
81         assert( iter_max > 0 );
82     }
```

7.4.4 Member Data Documentation

7.4.4.1 constraints_type

```
template<typename T >  
const Constraints_type ea::EA_base< T >::constraints_type
```

Constraints type.

Definition at line 48 of file ealgorithm_base.h.

7.4.4.2 decision_variables

```
template<typename T >  
const std::vector<T> ea::EA_base< T >::decision_variables
```

Initial Decision Variables.

Definition at line 34 of file ealgorithm_base.h.

7.4.4.3 iter_max

```
template<typename T >  
const size_t ea::EA_base< T >::iter_max
```

Number of maximum iterations.

Definition at line 42 of file ealgorithm_base.h.

7.4.4.4 ndv

```
template<typename T >  
const size_t ea::EA_base< T >::ndv
```

Number of decision variables.

Definition at line 44 of file ealgorithm_base.h.

Referenced by ea::PSOI< T >::PSOI(), and ea::PSOs< T >::PSOs().

7.4.4.5 npop

```
template<typename T >
const size_t ea::EA_base< T >::npop
```

Size of the population.

Definition at line 38 of file ealgorithm_base.h.

Referenced by ea::Solver< DE, T, F, C >::set_indices().

7.4.4.6 print_to_file

```
template<typename T >
const bool ea::EA_base< T >::print_to_file
```

Print to file or not.

Definition at line 52 of file ealgorithm_base.h.

7.4.4.7 print_to_output

```
template<typename T >
const bool ea::EA_base< T >::print_to_output
```

Print to output or not.

Definition at line 50 of file ealgorithm_base.h.

7.4.4.8 stdev

```
template<typename T >
const std::vector<T> ea::EA_base< T >::stdev
```

Standard deviation of the decision variables.

Definition at line 36 of file ealgorithm_base.h.

7.4.4.9 tol

```
template<typename T >
const T ea::EA_base< T >::tol
```

Tolerance.

Definition at line 40 of file ealgorithm_base.h.

7.4.4.10 use_penalty_method

```
template<typename T >
const bool ea::EA_base< T >::use_penalty_method
```

Use penalty function or not.

Definition at line 46 of file ealgorithm_base.h.

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

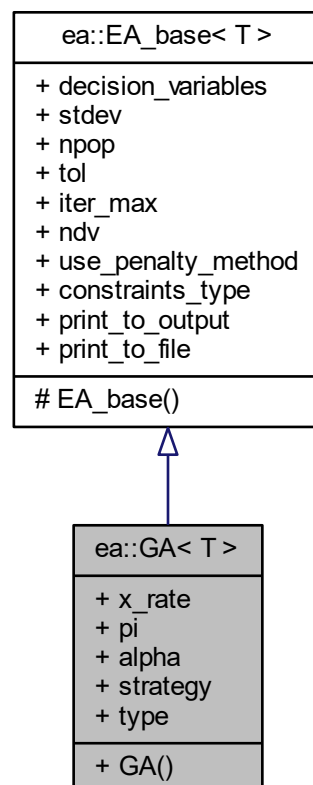
- [ealgorithm_base.h](#)

7.5 ea::GA< T > Struct Template Reference

Genetic Algorithms Structure, used in the actual algorithm and for type deduction.

```
#include <geneticalgo.h>
```

Inheritance diagram for ea::GA< T >:



Public Member Functions

- [GA](#) (const T &i_x_rate, const T &i_pi, const T &i_alpha, const std::vector< T > &i_decision_variables, const std::vector< T > &i_stdev, const size_t &i_npop, const T &i_tol, const size_t &i_iter_max, const bool &i_use_penalty_method, const [Constraints_type](#) &i_constraints_type, const [Strategy](#) &i_strategy, const bool &i_print_to_output, const bool &i_print_to_file)

Constructor.

Public Attributes

- const T [x_rate](#)
Natural Selection rate.
- const T [pi](#)
Probability of mutating.
- const T [alpha](#)
Parameter alpha for Beta distribution.
- const [Strategy](#) [strategy](#)
Replacing or remove individuals strategies during mutation.
- const std::string [type](#) = "Genetic Algorithms"
Type of the algorithm.

Additional Inherited Members

7.5.1 Detailed Description

```
template<typename T>
struct ea::GA< T >
```

Genetic Algorithms Structure, used in the actual algorithm and for type deduction.

Definition at line 21 of file geneticalgo.h.

7.5.2 Constructor & Destructor Documentation

7.5.2.1 GA()

```
template<typename T>
ea::GA< T >::GA (
    const T & i_x_rate,
    const T & i_pi,
    const T & i_alpha,
    const std::vector< T > & i_decision_variables,
    const std::vector< T > & i_stdev,
    const size_t & i_npop,
    const T & i_tol,
    const size_t & i_iter_max,
    const bool & i_use_penalty_method = false,
    const Constraints\_type & i_constraints_type = Constraints\_type::none,
    const Strategy & i_strategy = Strategy::keep\_same,
    const bool & i_print_to_output = true,
    const bool & i_print_to_file = true ) [inline]
```

Constructor.

Parameters

<i>i_x_rate</i>	Selection Rate or percentage of population to keep up to the next generation
<i>i_pi</i>	Probability of mutation
<i>i_alpha</i>	Alpha parameter of the Beta Distribution
<i>i_strategy</i>	The strategy used for handling constraints
<i>i_decision_variables</i>	The starting values of the decision variables
<i>i_stdev</i>	The standard deviation
<i>i_npop</i>	The population size
<i>i_tol</i>	The tolerance
<i>i_iter_max</i>	The maximum number of iterations
<i>i_use_penalty_method</i>	Whether to used penalties or not
<i>i_constraints_type</i>	What kind of constraints to use
<i>i_print_to_output</i>	Whether to print to terminal or not
<i>i_print_to_file</i>	Whether to print to a file or not

Returns

A GA<T> object

Definition at line 44 of file geneticalgo.h.

```

47                                     :
48     EA_base<T> ( i_decision_variables, i_stdev, i_npop, i_tol, i_iter_max, i_use_penalty_method,
    i_constraints_type, i_print_to_output, i_print_to_file ),
49     x_rate ( i_x_rate ),
50     pi ( i_pi ),
51     alpha ( i_alpha ),
52     strategy ( i_strategy )
53     {
54         assert(x_rate > 0 && x_rate <= 1);
55         assert(pi > 0 && pi <= 1);
56     }
```

7.5.3 Member Data Documentation

7.5.3.1 alpha

```

template<typename T>
const T ea::GA< T >::alpha
```

Parameter alpha for Beta distribution.

Definition at line 62 of file geneticalgo.h.

Referenced by ea::Solver< GA, T, F, C >::display_parameters().

7.5.3.2 pi

```
template<typename T>
const T ea::GA< T >::pi
```

Probability of mutating.

Definition at line 60 of file `geneticalgo.h`.

Referenced by `ea::Solver< GA, T, F, C >::display_parameters()`.

7.5.3.3 strategy

```
template<typename T>
const Strategy ea::GA< T >::strategy
```

Replacing or remove individuals strategies during mutation.

Definition at line 64 of file `geneticalgo.h`.

Referenced by `ea::Solver< GA, T, F, C >::display_parameters()`.

7.5.3.4 type

```
template<typename T>
const std::string ea::GA< T >::type = "Genetic Algorithms"
```

Type of the algorithm.

Definition at line 66 of file `geneticalgo.h`.

7.5.3.5 x_rate

```
template<typename T>
const T ea::GA< T >::x_rate
```

Natural Selection rate.

Definition at line 58 of file `geneticalgo.h`.

Referenced by `ea::Solver< GA, T, F, C >::display_parameters()`.

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

- [geneticalgo.h](#)

7.6 yft::Interest_Rate< T > Struct Template Reference

Structure for interest rates.

```
#include <yield_curve_fitting.h>
```

Public Member Functions

- [Interest_Rate](#) (const T &i_period, const T &i_rate)
Constructor.

Public Attributes

- const T [period](#)
Period is the time when the rate was recorder.
- const T [rate](#)
Interest rate.

7.6.1 Detailed Description

```
template<typename T>
struct yft::Interest_Rate< T >
```

Structure for interest rates.

Definition at line 22 of file yield_curve_fitting.h.

7.6.2 Constructor & Destructor Documentation

7.6.2.1 Interest_Rate()

```
template<typename T >
yft::Interest_Rate< T >::Interest_Rate (
    const T & i_period,
    const T & i_rate ) [inline]
```

Constructor.

Parameters

<i>i_period</i>	The period the zero rate was recorded
<i>i_rate</i>	Zero rate (spot interest rate)

Returns

An [Interest_Rate](#) object

Definition at line 30 of file `yield_curve_fitting.h`.

```

30                                     :
31         period{ i_period },
32         rate{ i_rate } {};
```

7.6.3 Member Data Documentation

7.6.3.1 period

```

template<typename T >
const T yft::Interest_Rate< T >::period
```

Period is the time when the rate was recorder.

Definition at line 32 of file `yield_curve_fitting.h`.

7.6.3.2 rate

```

template<typename T >
const T yft::Interest_Rate< T >::rate
```

Interest rate.

Definition at line 36 of file `yield_curve_fitting.h`.

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

- [yield_curve_fitting.h](#)

7.7 yft::Interest_Rate_Helper< T > Class Template Reference

A class for the yield-curve-fitting problem.

```
#include <yield_curve_fitting.h>
```

Public Member Functions

- [Interest_Rate_Helper](#) (const std::vector< [Interest_Rate](#)< T >> &i_ir_vec)
Constructor.
- template<typename S >
void [yieldcurve_fitting](#) (const S &solver)
Yield Curve Fitting using interest rates and recorded periods.

Private Member Functions

- `T fitness_yield_curve_fitting` (const std::vector< T > &solution, const bool &use_penalty_method)
This is the fitness function for yield-curve fitting using Interest Rates.

Private Attributes

- std::vector< [Interest_Rate](#)< T > > `ir_vec`
Vector of interest rates.

7.7.1 Detailed Description

```
template<typename T>
class yft::Interest_Rate_Helper< T >
```

A class for the yield-curve-fitting problem.

Definition at line 65 of file `yield_curve_fitting.h`.

7.7.2 Constructor & Destructor Documentation

7.7.2.1 Interest_Rate_Helper()

```
template<typename T >
yft::Interest_Rate_Helper< T >::Interest_Rate_Helper (
    const std::vector< Interest\_Rate< T >> & i_ir_vec ) [inline]
```

Constructor.

Parameters

<code>i_ir_vec</code>	A vector of Interest_Rate <T> objects
-----------------------	---

Returns

An [Interest_Rate_Helper](#) object

Definition at line 73 of file `yield_curve_fitting.h`.

```
73                                     :
74         ir\_vec{ i_ir_vec } {};
```

7.7.3 Member Function Documentation

7.7.3.1 fitness_yield_curve_fitting()

```
template<typename T >
yft::Interest_Rate_Helper< T >::fitness_yield_curve_fitting (
    const std::vector< T > & solution,
    const bool & use_penalty_method ) [inline], [private]
```

This is the fitness function for yield-curve fitting using Interest Rates.

Parameters

<i>solution</i>	NSS parameters candindate solution
<i>use_penalty_method</i>	Whether to use the penalty method defined for NSS or not

Returns

The fitness cost of NSS for yield curve fitting

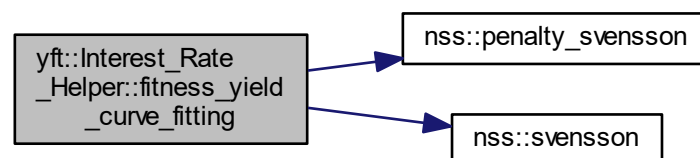
The sum of squares of errors between the actual rates and the rates computed by svensson are used

Definition at line 105 of file yield_curve_fitting.h.

References `nss::penalty_svensson()`, and `nss::svensson()`.

```
106     {
107         T sum_of_squares = 0;
108         for (size_t i = 0; i < ir_vec.size(); ++i)
109         {
110             T estimate = svensson(solution, ir_vec[i].period);
111             sum_of_squares = sum_of_squares + std::pow(ir_vec[i].rate - estimate, 2);
112         }
113         if (use_penalty_method)
114         {
115             return sum_of_squares + penalty_svensson(solution);
116         }
117         else
118         {
119             return sum_of_squares;
120         }
121     }
122     };
```

Here is the call graph for this function:



7.7.3.2 yieldcurve_fitting()

```
template<typename T >
template<typename S >
yft::Interest_Rate_Helper< T >::yieldcurve_fitting (
    const S & solver ) [inline]
```

Yield Curve Fitting using interest rates and recorded periods.

Parameters

<i>solver</i>	The parameter structure of the solver that is going to be used for yield curve fitting
---------------	--

Returns

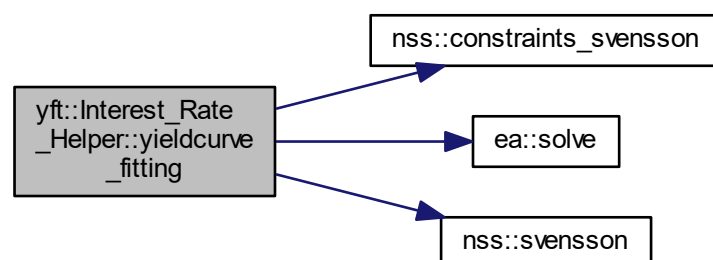
void

Definition at line 81 of file yield_curve_fitting.h.

References `nss::constraints_svensson()`, `ea::solve()`, and `nss::svensson()`.

```
82     {
83         assert(solver.ndv == 6);
84         auto f = [&, use_penalty_method = solver.use_penalty_method](const auto& solution) { return
85 fitness_yieldcurve_fitting(solution, use_penalty_method); };
86         auto c = [&, constraints_type = solver.constraints_type](const auto& solution) { return
87 constraints_svensson(solution, constraints_type); };
88         std::cout << "Yield Curve fitting." << "\n";
89         auto res = solve(f, c, solver, "YFT");
90         T error = 0;
91         for (const auto& p : ir_vec)
92         {
93             error = error + std::pow(svensson(res, p.period) - p.rate, 2);
94             //std::cout << "Estimated interest rates: " << svensson(res, p.period) << " Actual interest
95 rates: " << p.rate << "\n";
96         }
97         std::cout << "Zero-rate Mean Squared Error: " << error / ir_vec.size() << "\n";
98     };
```

Here is the call graph for this function:



7.7.4 Member Data Documentation

7.7.4.1 ir_vec

```
template<typename T >
std::vector<Interest_Rate<T> > yft::Interest_Rate_Helper< T >::ir_vec [private]
```

Vector of interest rates.

Definition at line 95 of file yield_curve_fitting.h.

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

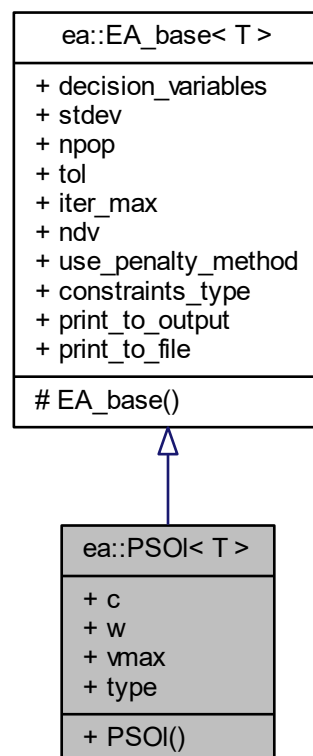
- [yield_curve_fitting.h](#)

7.8 ea::PSOI< T > Struct Template Reference

Local Best Particle Swarm Optimisation Structure, used in the actual algorithm and for type deduction.

```
#include <lbestpso.h>
```

Inheritance diagram for ea::PSOI< T >:



Public Member Functions

- **PSOI** (const T &i_c, const T &i_w, const std::vector< T > &i_vmax, const std::vector< T > &i_decision_variables, const std::vector< T > &i_stdev, const size_t &i_npop, const T &i_tol, const size_t &i_iter_max, const bool &i_use_penalty_method, const **Constraints_type** &i_constraints_type, const bool &i_print_to_output, const bool &i_print_to_file)

Constructor.

Public Attributes

- const T **c**
Parameter c for velocity update.
- const T **w**
Inertia Variant of PSO : Inertia.
- const std::vector< T > **vmax**
Velocity Clamping Variant of PSO : Maximum Velocity.
- const std::string **type** = "Local Best Particle Swarm Optimisation"
Type of the algorithm.

Additional Inherited Members

7.8.1 Detailed Description

```
template<typename T>
struct ea::PSOI< T >
```

Local Best Particle Swarm Optimisation Structure, used in the actual algorithm and for type deduction.

Definition at line 19 of file lbestpso.h.

7.8.2 Constructor & Destructor Documentation

7.8.2.1 PSOI()

```
template<typename T>
ea::PSOI< T >::PSOI (
    const T & i_c,
    const T & i_w,
    const std::vector< T > & i_vmax,
    const std::vector< T > & i_decision_variables,
    const std::vector< T > & i_stdev,
    const size_t & i_npop,
    const T & i_tol,
    const size_t & i_iter_max,
    const bool & i_use_penalty_method = false,
    const Constraints_type & i_constraints_type = Constraints_type::none,
    const bool & i_print_to_output = true,
    const bool & i_print_to_file = true ) [inline]
```

Constructor.

Parameters

<i>i_c</i>	c parameter for velocity update
<i>i_w</i>	Inertia parameter for velocity update
<i>i_vmax</i>	Maximum velocity
<i>i_decision_variables</i>	The starting values of the decision variables
<i>i_stdev</i>	The standard deviation
<i>i_npop</i>	The population size
<i>i_tol</i>	The tolerance
<i>i_iter_max</i>	The maximum number of iterations
<i>i_use_penalty_method</i>	Whether to used penalties or not
<i>i_constraints_type</i>	What kind of constraints to use
<i>i_print_to_output</i>	Whether to print to terminal or not
<i>i_print_to_file</i>	Whether to print to a file or not

Returns

A PSOI<T> object

Definition at line 41 of file lbestpso.h.

References ea::PSOI< T >::c, ea::EA_base< T >::ndv, ea::PSOI< T >::vmax, and ea::PSOI< T >::w.

```

44                                     :
45     EA_base<T>(i_decision_variables, i_stdev, i_npop, i_tol, i_iter_max, i_use_penalty_method,
i_constraints_type, i_print_to_output, i_print_to_file),
46     c(i_c),
47     w(i_w),
48     vmax(i_vmax)
49     {
50         assert(c > 0);
51         assert(w > 0);
52         for (const auto& p : vmax) { assert(p > 0); };
53         assert(vmax.size() == this->ndv);
54     }

```

7.8.3 Member Data Documentation

7.8.3.1 c

```

template<typename T>
const T ea::PSOI< T >::c

```

Parameter c for velocity update.

Definition at line 56 of file lbestpso.h.

Referenced by ea::Solver< PSOI, T, F, C >::display_parameters(), ea::Solver< PSOI, T, F, C >::position_update(), and ea::PSOI< T >::PSOI().

7.8.3.2 type

```
template<typename T>
const std::string ea::PSOI< T >::type = "Local Best Particle Swarm Optimisation"
```

Type of the algorithm.

Definition at line 62 of file lbestpso.h.

7.8.3.3 vmax

```
template<typename T>
const std::vector<T> ea::PSOI< T >::vmax
```

Velocity Clamping Variant of PSO : Maximum Velocity.

Definition at line 60 of file lbestpso.h.

Referenced by `ea::Solver< PSOI, T, F, C >::display_parameters()`, `ea::Solver< PSOI, T, F, C >::position_update()`, and `ea::PSOI< T >::PSOI()`.

7.8.3.4 w

```
template<typename T>
const T ea::PSOI< T >::w
```

Inertia Variant of PSO : Inertia.

Definition at line 58 of file lbestpso.h.

Referenced by `ea::Solver< PSOI, T, F, C >::display_parameters()`, `ea::Solver< PSOI, T, F, C >::position_update()`, `ea::PSOI< T >::PSOI()`, and `ea::Solver< PSOI, T, F, C >::run_algo()`.

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

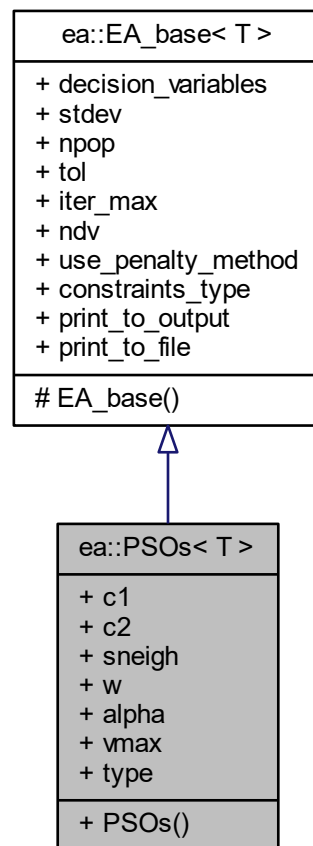
- [lbestpso.h](#)

7.9 ea::PSOs< T > Struct Template Reference

Particle Swarm Optimisation Structure, used in the actual algorithm and for type deduction.

```
#include <pso_sub_swarm.h>
```

Inheritance diagram for ea::PSOs< T >:



Public Member Functions

- [PSOs](#) (const T &i_c1, const T &i_c2, const size_t &i_sneigh, const T &i_w, const T &i_alpha, const std::vector< T > &i_vmax, const std::vector< T > &i_decision_variables, const std::vector< T > &i_stdev, const size_t &i_npop, const T &i_tol, const size_t &i_iter_max, const bool &i_use_penalty_method, const [Constraints_type](#) &i_constraints_type, const bool &i_print_to_output, const bool &i_print_to_file)

Constructor.

Public Attributes

- const T [c1](#)
Parameter c1 for velocity update.
- const T [c2](#)
Parameter c2 for velocity update.
- const size_t [sneigh](#)
Neighbourhood size.
- const T [w](#)
Inertia Variant of PSO : Inertia.
- const T [alpha](#)
Alpha Parameter for maximum velocity.
- const std::vector< T > [vmax](#)
Velocity Clamping Variant of PSO : Maximum Velocity.
- const std::string [type](#) = "Sub-swarm Particle Swarm Optimisation"
Type of the algorithm.

Additional Inherited Members

7.9.1 Detailed Description

```
template<typename T>
struct ea::PSOs< T >
```

Particle Swarm Optimisation Structure, used in the actual algorithm and for type deduction.

Definition at line 17 of file pso_sub_swarm.h.

7.9.2 Constructor & Destructor Documentation

7.9.2.1 PSOs()

```
template<typename T>
ea::PSOs< T >::PSOs (
    const T & i_c1,
    const T & i_c2,
    const size_t & i_sneigh,
    const T & i_w,
    const T & i_alpha,
    const std::vector< T > & i_vmax,
    const std::vector< T > & i_decision_variables,
    const std::vector< T > & i_stdev,
    const size_t & i_npop,
    const T & i_tol,
    const size_t & i_iter_max,
    const bool & i_use_penalty_method = false,
    const Constraints_type & i_constraints_type = Constraints_type::none,
    const bool & i_print_to_output = true,
    const bool & i_print_to_file = true ) [inline]
```

Constructor.

Parameters

<i>i_c1</i>	c1 parameter for velocity update
<i>i_c2</i>	c2 parameter for velocity update
<i>i_sneigh</i>	Number of neighbourhoods
<i>i_w</i>	Inertia parameter for velocity update
<i>i_alpha</i>	alpha parameter for maximum velocity decrease
<i>i_vmax</i>	Maximum velocity
<i>i_decision_variables</i>	The starting values of the decision variables
<i>i_stdev</i>	The standard deviation
<i>i_npop</i>	The population size
<i>i_tol</i>	The tolerance
<i>i_iter_max</i>	The maximum number of iterations
<i>i_use_penalty_method</i>	Whether to used penalties or not
<i>i_constraints_type</i>	What kind of constraints to use
<i>i_print_to_output</i>	Whether to print to terminal or not
<i>i_print_to_file</i>	Whether to print to a file or not

Returns

A PSO<T> object

Definition at line 42 of file pso_sub_swarm.h.

References `ea::PSOs< T >::alpha`, `ea::PSOs< T >::c1`, `ea::PSOs< T >::c2`, `ea::EA_base< T >::ndv`, `ea::PSOs< T >::sneigh`, `ea::PSOs< T >::vmax`, and `ea::PSOs< T >::w`.

```

45                                     :
46     EA_base<T>(i_decision_variables, i_stdev, i_npop, i_tol, i_iter_max, i_use_penalty_method,
47     i_constraints_type, i_print_to_output, i_print_to_file),
48     c1( i_c1 ),
49     c2( i_c2 ),
50     sneigh( i_sneigh ),
51     w( i_w ),
52     alpha( i_alpha ),
53     vmax( i_vmax )
54     {
55         assert(c1 > 0);
56         assert(c2 > 0);
57         assert(sneigh > 0);
58         assert(sneigh < i_npop);
59         assert(w > 0);
60         assert(alpha > 0);
61         for (const auto& p : vmax) { assert(p > 0); };
62         assert(vmax.size() == this->ndv);
63     }

```

7.9.3 Member Data Documentation

7.9.3.1 alpha

```

template<typename T>
const T ea::PSOs< T >::alpha

```

Alpha Parameter for maximum velocity.

Definition at line 72 of file pso_sub_swarm.h.

Referenced by `ea::Solver< PSOs, T, F, C >::display_parameters()`, and `ea::PSOs< T >::PSOs()`.

7.9.3.2 c1

```
template<typename T>
const T ea::PSOs< T >::c1
```

Parameter c1 for velocity update.

Definition at line 64 of file pso_sub_swarm.h.

Referenced by ea::Solver< PSOs, T, F, C >::display_parameters(), and ea::PSOs< T >::PSOs().

7.9.3.3 c2

```
template<typename T>
const T ea::PSOs< T >::c2
```

Parameter c2 for velocity update.

Definition at line 66 of file pso_sub_swarm.h.

Referenced by ea::Solver< PSOs, T, F, C >::display_parameters(), and ea::PSOs< T >::PSOs().

7.9.3.4 sneigh

```
template<typename T>
const size_t ea::PSOs< T >::sneigh
```

Neighbourhood size.

Definition at line 68 of file pso_sub_swarm.h.

Referenced by ea::Solver< PSOs, T, F, C >::display_parameters(), and ea::PSOs< T >::PSOs().

7.9.3.5 type

```
template<typename T>
const std::string ea::PSOs< T >::type = "Sub-swarm Particle Swarm Optimisation"
```

Type of the algorithm.

Definition at line 76 of file pso_sub_swarm.h.

7.9.3.6 vmax

```
template<typename T>
const std::vector<T> ea::PSOs< T >::vmax
```

Velocity Clamping Variant of PSO : Maximum Velocity.

Definition at line 74 of file pso_sub_swarm.h.

Referenced by ea::Solver< PSOs, T, F, C >::display_parameters(), ea::Solver< PSOs, T, F, C >::position_update(), and ea::PSOs< T >::PSOs().

7.9.3.7 w

```
template<typename T>
const T ea::PSOs< T >::w
```

Inertia Variant of PSO : Inertia.

Definition at line 70 of file pso_sub_swarm.h.

Referenced by ea::Solver< PSOs, T, F, C >::display_parameters(), ea::PSOs< T >::PSOs(), and ea::Solver< PSOs, T, F, C >::run_algo().

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

- [pso_sub_swarm.h](#)

7.10 ea::Solver< S, T, F, C > Class Template Reference

Template Class for Solvers.

```
#include <ealgorithm_base.h>
```

7.10.1 Detailed Description

```
template<template< typename > class S, typename T, typename F, typename C>
class ea::Solver< S, T, F, C >
```

Template Class for Solvers.

Definition at line 93 of file ealgorithm_base.h.

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

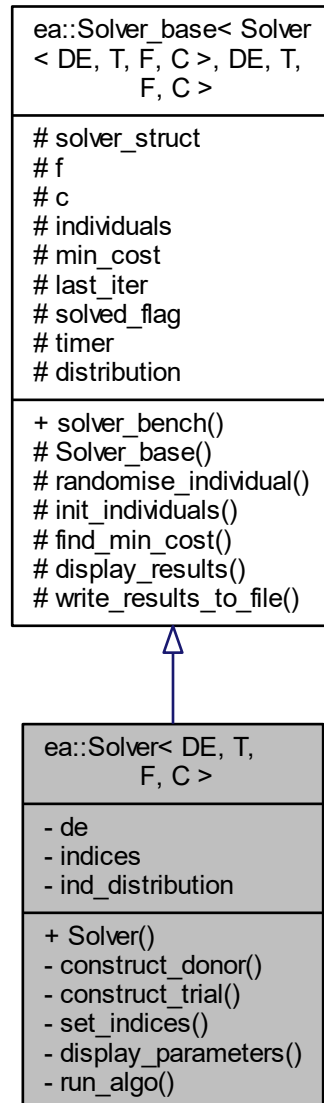
- [ealgorithm_base.h](#)

7.11 ea::Solver< DE, T, F, C > Class Template Reference

Differential Evolution Algorithm ([DE](#)) Class.

```
#include <differentialevo.h>
```

Inheritance diagram for ea::Solver< DE, T, F, C >:



Public Member Functions

- [Solver](#) (const [DE](#)< T > &i_de, const F &f, const C &c)
Constructor.

Private Member Functions

- `std::vector< T > construct_donor ()`
Method that constructs the donor vector.
- `std::vector< T > construct_trial (const std::vector< T > &target, const std::vector< T > &donor)`
Method that constructs the trial vector.
- `std::vector< size_t > set_indices ()`
Generate the indices.
- `std::stringstream display_parameters ()`
Display the parameters of [DE](#).
- `void run_algo ()`
Runs the algorithm until stopping criteria return void.

Private Attributes

- `const DE< T > & de`
Differential Evolution structure used internally (reference to [solver_struct](#))
- `const std::vector< size_t > indices`
Indices of population.
- `std::uniform_int_distribution< size_t > ind_distribution`
Uniform `size_t` distribution of the indices.

Friends

- `class Solver_base< Solver< DE, T, F, C >, DE, T, F, C >`

Additional Inherited Members

7.11.1 Detailed Description

```
template<typename T, typename F, typename C>
class ea::Solver< DE, T, F, C >
```

Differential Evolution Algorithm ([DE](#)) Class.

Definition at line 60 of file `differentialevo.h`.

7.11.2 Constructor & Destructor Documentation

7.11.2.1 Solver()

```
template<typename T , typename F , typename C >
ea::Solver< DE, T, F, C >::Solver (
    const DE< T > & i\_de,
    const F & f,
    const C & c ) [inline]
```

Constructor.

Parameters

<i>i_de</i>	The differential evolution parameter structure that is used to construct the solver
<i>f</i>	A reference to the objective function
<i>c</i>	A reference to the constraints function

Returns

A [Solver<DE, T, F, C>](#) object

Definition at line 71 of file differentialevo.h.

```

71         :
72         Solver_base<Solver<DE, T, F, C>, DE, T, F, C>( i_de, f, c ),
73         de( this->solver_struct ),
74         indices( set_indices() ),
75         ind_distribution( std::uniform_int_distribution<size_t>(0,
de.ndpop - 1) )
76     {
77     };

```

7.11.3 Member Function Documentation

7.11.3.1 construct_donor()

```

template<typename T , typename F , typename C >
std::vector< T > ea::Solver< DE, T, F, C >::construct_donor ( ) [private]

```

Method that constructs the donor vector.

Returns

Donor vector

Check that the indices are not the same

Definition at line 129 of file differentialevo.h.

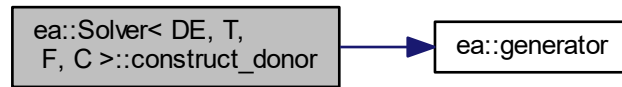
References [ea::generator\(\)](#).

```

130     {
131         std::vector<T> donor(de.ndv);
132         std::vector<size_t> r_i;
133         while (r_i.size() < 3)
134         {
135             r_i.push_back(indices[ind_distribution(
generator)]);
136             if (r_i.size() > 1 && r_i.end() [-1] == r_i.end() [-2])
137             {
138                 r_i.pop_back();
139             }
140         }
141         for (size_t j = 0; j < de.ndv; ++j)
142         {
143             donor[j] = this->individuals[r_i[0]][j] + de.f_param * (this->
individuals[r_i[1]][j] - this->individuals[r_i[2]][j]);
144         }
145         return donor;
146     }
147 }

```


Here is the call graph for this function:



7.11.3.2 construct_trial()

```

template<typename T , typename F , typename C >
std::vector< T > ea::Solver< DE, T, F, C >::construct_trial (
    const std::vector< T > & target,
    const std::vector< T > & donor ) [private]
  
```

Method that constructs the trial vector.

Parameters

<i>target</i>	Target vector (an individual)
<i>donor</i>	Donor vector produced from construct_donor()

Returns

A trial vector that is compared with the current individual

Definition at line 150 of file differentialevo.h.

References [ea::generator\(\)](#).

```

151     {
152         std::vector<T> trial(de.ndv);
153         std::vector<size_t> j_indices(de.ndv);
154         for (size_t j = 0; j < de.ndv; ++j)
155         {
156             j_indices[j] = j;
157         }
158         std::uniform_int_distribution<size_t> j_ind_distribution(0, de.ndv - 1);
159         for (size_t j = 0; j < de.ndv; ++j)
160         {
161             const T& epsilon = this->distribution(generator);
162             const size_t& jrand = j_indices[j_ind_distribution(generator)];
163             if (epsilon <= de.cr || j == jrand)
164             {
165                 trial[j] = donor[j];
166             }
167             else
168             {
169                 trial[j] = target[j];
170             }
171         }
172         return trial;
173     }
  
```

Here is the call graph for this function:



7.11.3.3 display_parameters()

```
template<typename T , typename F , typename C >
ea::Solver< DE, T, F, C >::display_parameters ( ) [inline], [private]
```

Display the parameters of [DE](#).

Returns

A `std::stringstream` of the parameters

Definition at line 114 of file `differentialevo.h`.

References `ea::DE< T >::cr`, and `ea::DE< T >::f_param`.

```
115     {
116         std::stringstream parameters;
117         parameters << "Crossover Rate:" << ", " << de.cr << ", ";
118         parameters << "Mutation Scale Factor:" << ", " << de.f_param;
119         return parameters;
120     }
```

7.11.3.4 run_algo()

```
template<typename T , typename F , typename C >
void ea::Solver< DE, T, F, C >::run_algo ( ) [private]
```

Runs the algorithm until stopping criteria return void.

Differential Evolution starts here

Construct donor and trial vectors

Recalculate minimum cost individual of the population

Stopping Criteria

Definition at line 176 of file `differentialevo.h`.

```

177     {
178         std::vector< std::vector<T> > personal_best = this->individuals;
179         for (size_t iter = 0; iter < de.iter_max; ++iter)
180         {
181             for (auto& p : this->individuals)
182             {
183                 std::vector<T> donor = construct_donor();
184                 while (!this->c(donor))
185                 {
186                     donor = construct_donor();
187                 }
188                 const std::vector<T>& trial = construct_trial(p, donor);
189                 if (this->f(trial) <= this->f(p))
190                 {
191                     p = trial;
192                 }
193             }
194         }
195         this->find_min_cost();
196         this->last_iter = iter;
197         if (de.tol > std::abs(this->f(this->min_cost)))
198         {
199             this->solved_flag = true;
200             break;
201         }
202     }
203 }
204
205 }
206

```

7.11.3.5 set_indices()

```

template<typename T , typename F , typename C >
ea::Solver< DE, T, F, C >::set_indices ( ) [inline], [private]

```

Generate the indices.

Returns

The vector of the population indices

Definition at line 101 of file differentialevo.h.

References ea::EA_base< T >::npop.

```

102     {
103         std::vector<size_t> indices;
104         for (size_t i = 0; i < de.npop; ++i)
105         {
106             indices.push_back(i);
107         }
108         return indices;
109     };

```

7.11.4 Friends And Related Function Documentation

7.11.4.1 Solver_base< Solver< DE, T, F, C >, DE, T, F, C >

```

template<typename T , typename F , typename C >
friend class Solver_base< Solver< DE, T, F, C >, DE, T, F, C > [friend]

```

Definition at line 63 of file differentialevo.h.

7.11.5 Member Data Documentation

7.11.5.1 de

```
template<typename T , typename F , typename C >  
const DE<T>& ea::Solver< DE, T, F, C >::de [private]
```

Differential Evolution structure used internally (reference to solver_struct)

Definition at line 77 of file differentialevo.h.

7.11.5.2 ind_distribution

```
template<typename T , typename F , typename C >  
std::uniform_int_distribution<size_t> ea::Solver< DE, T, F, C >::ind_distribution [private]
```

Uniform size_t distribution of the indices.

Definition at line 84 of file differentialevo.h.

7.11.5.3 indices

```
template<typename T , typename F , typename C >  
const std::vector<size_t> ea::Solver< DE, T, F, C >::indices [private]
```

Indices of population.

Definition at line 82 of file differentialevo.h.

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

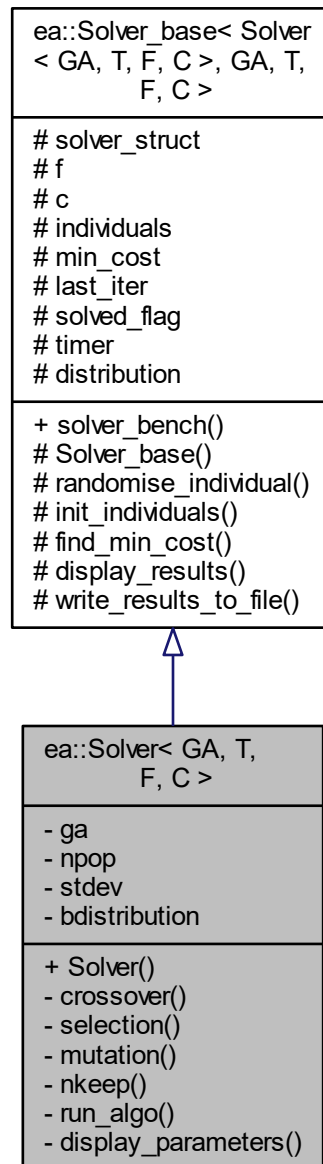
- [differentialevo.h](#)

7.12 ea::Solver< GA, T, F, C > Class Template Reference

Genetic Algorithms ([GA](#)) Class.

```
#include <geneticalgo.h>
```

Inheritance diagram for ea::Solver< GA, T, F, C >:



Public Member Functions

- [Solver](#) (const [GA](#)< T > &i_ga, F f, C c)
Constructor.

Private Member Functions

- `std::vector< T > crossover` (`std::vector< T > r`, `std::vector< T > s`)
Crossover step of GA.
- `std::vector< T > selection` ()
Selection step of GA.
- `std::vector< T > mutation` (`const std::vector< T > &individual`)
Mutation step of GA.
- `size_t nkeep` ()
Returns number of individuals to be kept in each generation.
- `void run_algo` ()
Runs the algorithm until stopping criteria return void.
- `std::stringstream display_parameters` ()
Display the parameters of GA.

Private Attributes

- `const GA< T > & ga`
Genetic Algorithms structure used internally (reference to solver_struct)
- `size_t npop`
Size of the population is mutable.
- `std::vector< T > stdev`
Standard deviation is mutable, so a copy is created.
- `boost::math::beta_distribution< T > bdistribution`
Beta distribution.

Friends

- `class Solver_base< Solver< GA, T, F, C >, GA, T, F, C >`

Additional Inherited Members

7.12.1 Detailed Description

```
template<typename T, typename F, typename C>
class ea::Solver< GA, T, F, C >
```

Genetic Algorithms (GA) Class.

Definition at line 73 of file `geneticalgo.h`.

7.12.2 Constructor & Destructor Documentation

7.12.2.1 Solver()

```
template<typename T , typename F , typename C >
ea::Solver< GA, T, F, C >::Solver (
    const GA< T > & i_ga,
    F f,
    C c ) [inline]
```

Constructor.

Parameters

<i>i_ga</i>	The genetic algorithms parameter structure that is used to construct the solver
<i>f</i>	A reference to the objective function
<i>c</i>	A reference to the constraints function

Returns

A [Solver<GA, T, F, C>](#) object

Definition at line 84 of file `geneticalgo.h`.

```

84                                     :
85     Solver_base<Solver<GA, T, F, C>, GA, T, F, C> ( i_ga, f, c ),
86     ga ( this->solver_struct ),
87     npop ( i_ga.npop ),
88     stdev ( i_ga.stdev ),
89     bdistribution (boost::math::beta_distribution<T>(1, ga.alpha))
90     {
91     }
```

7.12.3 Member Function Documentation

7.12.3.1 crossover()

```

template<typename T , typename F , typename C >
std::vector< T > ea::Solver< GA, T, F, C >::crossover (
    std::vector< T > r,
    std::vector< T > s ) [private]
```

Crossover step of [GA](#).

Parameters

<i>r,s</i>	Parent individuals
------------	--------------------

Returns

An offspring from the two parents *r* and *s*

Definition at line 152 of file `geneticalgo.h`.

References [ea::generator\(\)](#).

```

153     {
154         std::vector<T> offspring(ga.ndv);
155         std::vector<T> psi(ga.ndv);
156         for (size_t j = 0; j < ga.ndv; ++j)
157         {
158             psi[j] = this->distribution(generator);
159             offspring[j] = psi[j] * r[j] + (1 - psi[j]) * s[j];
160         }
161         return offspring;
162     }
```

Here is the call graph for this function:



7.12.3.2 display_parameters()

```
template<typename T , typename F , typename C >
ea::Solver< GA, T, F, C >::display_parameters ( ) [inline], [private]
```

Display the parameters of [GA](#).

Returns

A `std::stringstream` of the parameters

Definition at line 133 of file `geneticalgo.h`.

References `ea::GA< T >::alpha`, `ea::keep_same`, `ea::GA< T >::pi`, `ea::re_mutate`, `ea::remove`, `ea::GA< T >::strategy`, and `ea::GA< T >::x_rate`.

```
134     {
135         std::stringstream parameters;
136         parameters << "Natural Selection Rate:" << "," << ga.x_rate << ",";
137         parameters << "Probability of Mutation:" << "," << ga.pi << ",";
138         parameters << "Beta Distribution alpha:" << "," << ga.alpha << ",";
139         parameters << "Strategy:" << ",";
140         switch (ga.strategy)
141         {
142             case Strategy::keep_same: parameters << "Keep same individual"; break;
143             case Strategy::re_mutate: parameters << "Re-mutate individual"; break;
144             case Strategy::remove: parameters << "Remove individual"; break;
145             default: parameters << "Do nothing"; break;
146         }
147         return parameters;
148     }
```

7.12.3.3 mutation()

```
template<typename T , typename F , typename C >
std::vector< T > ea::Solver< GA, T, F, C >::mutation (
    const std::vector< T > & individual ) [private]
```

Mutation step of [GA](#).

Parameters

<i>individual</i>	An individual of the population
-------------------	---------------------------------

Returns

A mutated individual

Definition at line 178 of file `geneticalgo.h`.

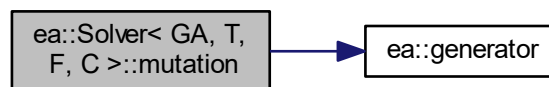
References `ea::generator()`.

```

179     {
180         std::vector<T> mutated = individual;
181         for (size_t j = 0; j < ga.ndv; ++j)
182         {
183             const T r = this->distribution(generator);
184             if (ga.pi < r)
185             {
186                 std::normal_distribution<T> ndistribution(0, stdev[j]);
187                 T epsilon = ndistribution(generator);
188                 mutated[j] = mutated[j] + epsilon;
189             }
190         }
191         return mutated;
192     }

```

Here is the call graph for this function:



7.12.3.4 nkeep()

```

template<typename T , typename F , typename C >
size_t ea::Solver< GA, T, F, C >::nkeep ( ) [private]

```

Returns number of individuals to be kept in each generation.

Returns

The new nkeep

Definition at line 195 of file `geneticalgo.h`.

```

196     {
197         return static_cast<size_t>(std::ceil(static_cast<T>(npop) * ga.x_rate));
198     }

```

7.12.3.5 run_algo()

```
template<typename T , typename F , typename C >
void ea::Solver< GA, T, F, C >::run_algo ( ) [private]
```

Runs the algorithm until stopping criteria return void.

Set the new population size which is previous population size + natural selection rate * population size

Standard Deviation is not constant in GA

Definition at line 201 of file geneticalgo.h.

References `ea::keep_same`, `ea::none`, `ea::re_mutate`, and `ea::remove`.

```
202     {
203         auto comparator = [&](const std::vector<T>& l, const std::vector<T>& r)
204         {
205             return this->f(l) < this->f(r);
206         };
207         for (size_t iter = 0; iter < ga.iter_max; ++iter)
208         {
209             npop = this->individuals.size();
210             std::sort(this->individuals.begin(), this->individuals.end(), comparator)
211         };
212         this->individuals.erase(this->individuals.begin() +
nkeep(), this->individuals.begin() + this->individuals.size());
213         this->min_cost = this->individuals[0];
214         this->last_iter = iter;
215         if (ga.tol > std::abs(this->f(this->min_cost)))
216         {
217             this->solved_flag = true;
218             break;
219         }
220         for (size_t i = 0; i < npop; ++i)
221         {
222             std::vector<T> offspring = selection();
223             this->individuals.push_back(offspring);
224         }
225         if (this->individuals.size() > 1000)
226         {
227             //this->individuals.erase(this->individuals.begin() + 1000, this->individuals.begin() +
this->individuals.size());
228         }
229         for (size_t i = 1; i < this->individuals.size(); ++i)
230         {
231             std::vector<T> mutated = mutation(this->individuals[i]);
232             if (!this->c(mutated))
233             {
234                 switch (ga.strategy)
235                 {
236                     case Strategy::keep_same: this->
individuals[i] = this->individuals[i]; break;
237                     case Strategy::re_mutate:
238                     {
239                         while (!this->c(mutated))
240                         {
241                             mutated = mutation(this->individuals[i]);
242                         }
243                         break;
244                         this->individuals[i] = mutated;
245                     }
246                     case Strategy::remove:
247                     {
248                         if (i == this->individuals.size() - 1)
249                         {
250                             this->individuals.pop_back();
251                         }
252                         else
253                         {
254                             this->individuals.erase(this->individuals.begin() + i,
this->individuals.begin() + i + 1);
255                         }
256                         break;
257                     }
258                     case Strategy::none: break;
259                     default: std::abort();
260                 }
261             }
262         }
263     }
```

```

262         else
263         {
264             this->individuals[i] = mutated;
265         }
266     }
267     for (auto& p : stdev)
268     {
269         p = p + 0.02 * p;
270     }
271 }
272 }
273 }

```

7.12.3.6 selection()

```

template<typename T , typename F , typename C >
std::vector< T > ea::Solver< GA, T, F, C >::selection ( ) [private]

```

Selection step of [GA](#).

Select two parents *r* and *s* using a Beta distribution and generates an offspring using the crossover method

Returns

An offspring from the two parents

Generate *r* and *s* indices

Produce offspring using *r* and *s* indices by crossover

Definition at line 165 of file `geneticalgo.h`.

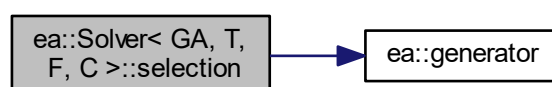
References [ea::generator\(\)](#).

```

166     {
167         T xi = quantile(bdistribution, this->distribution(
168             generator));
169         size_t r = static_cast<size_t>(std::floor(static_cast<T>(nkeep()) * xi));
170         xi = quantile(bdistribution, this->distribution(
171             generator));
172         size_t s = static_cast<size_t>(std::floor(static_cast<T>(nkeep()) * xi));
173         std::vector<T> offspring = crossover(this->individuals[r], this->
174             individuals[s]);
175         return offspring;
176     }

```

Here is the call graph for this function:



7.12.4 Friends And Related Function Documentation

7.12.4.1 Solver_base< Solver< GA, T, F, C >, GA, T, F, C >

```
template<typename T , typename F , typename C >
friend class Solver_base< Solver< GA, T, F, C >, GA, T, F, C > [friend]
```

Definition at line 76 of file geneticalgo.h.

7.12.5 Member Data Documentation

7.12.5.1 bdistribution

```
template<typename T , typename F , typename C >
boost::math::beta_distribution<T> ea::Solver< GA, T, F, C >::bdistribution [private]
```

Beta distribution.

Definition at line 100 of file geneticalgo.h.

7.12.5.2 ga

```
template<typename T , typename F , typename C >
const GA<T>& ea::Solver< GA, T, F, C >::ga [private]
```

Genetic Algorithms structure used internally (reference to solver_struct)

Definition at line 94 of file geneticalgo.h.

7.12.5.3 npop

```
template<typename T , typename F , typename C >
size_t ea::Solver< GA, T, F, C >::npop [private]
```

Size of the population is mutable.

Definition at line 96 of file geneticalgo.h.

7.12.5.4 stdev

```
template<typename T , typename F , typename C >
std::vector<T> ea::Solver< GA, T, F, C >::stdev [private]
```

Standard deviation is mutable, so a copy is created.

Definition at line 98 of file `geneticalgo.h`.

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

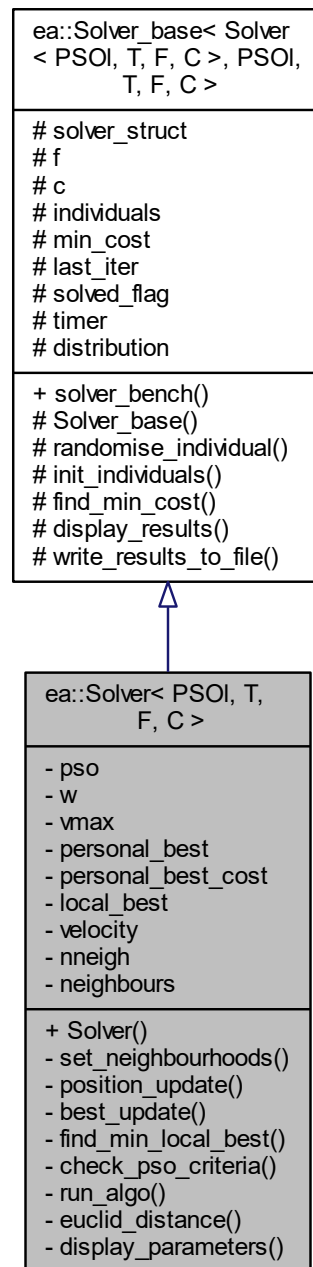
- [geneticalgo.h](#)

7.13 ea::Solver< PSOI, T, F, C > Class Template Reference

Local Best Particle Swarm Optimisation (PSO) Class.

```
#include <lbestpso.h>
```

Inheritance diagram for `ea::Solver< PSOI, T, F, C >`:



Public Member Functions

- [Solver](#) (const [PSOI](#)< T > &i_pso, F [f](#), C [c](#))

Constructor.

Private Member Functions

- `std::unordered_map< size_t, std::array< size_t, 3 > > set_neighbourhoods ()`
Set the neighbourhoods of the algorithm using particle indices.
- `void position_update ()`
Position update of the particles.
- `void best_update ()`
This method sets the personal and local best solutions.
- `void find_min_local_best ()`
This is a faster way to calculate the minimum cost unless there is only one neighbourhood, in which case it is the same as `find_min_cost(F f)`
- `bool check_pso_criteria ()`
Define the maximum radius stopping criterion.
- `void run_algo ()`
Runs the algorithm until stopping criteria return void.
- `T euclid_distance (const std::vector< T > &x, const std::vector< T > &y)`
Euclidean Distance of two vectors ..
- `std::stringstream display_parameters ()`
Display PSO parameters.

Private Attributes

- `const PSOI< T > & pso`
Particle Swarm Optimisation structure used internally (reference to solver_struct)
- `T w`
Inertia is mutable, so a copy is created.
- `std::vector< T > vmax`
Maximum Velocity is mutable, so a copy is created.
- `std::vector< std::vector< T > > personal_best`
Personal best vector of the particles, holds the best position recorded for each particle.
- `std::vector< T > personal_best_cost`
Personal best cost vector of the particles.
- `std::vector< std::vector< T > > local_best`
Local best vector, holds the best position recorded for each neighbourhood.
- `std::vector< std::vector< T > > velocity`
Velocity of the particles.
- `const size_t nneigh`
Number of neighbourhoods.
- `std::unordered_map< size_t, std::array< size_t, 3 > > neighbours`
Neighbours of each particle.

Friends

- `class Solver_base< Solver< PSOI, T, F, C >, PSOI, T, F, C >`

Additional Inherited Members

7.13.1 Detailed Description

```
template<typename T, typename F, typename C>
class ea::Solver< PSOL, T, F, C >
```

Local Best Particle Swarm Optimisation (PSO) Class.

Definition at line 69 of file lbestpso.h.

7.13.2 Constructor & Destructor Documentation

7.13.2.1 Solver()

```
template<typename T , typename F , typename C >
ea::Solver< PSOL, T, F, C >::Solver (
    const PSOL< T > & i_pso,
    F f,
    C c ) [inline]
```

Constructor.

Parameters

<i>i_pso</i>	The particle swarm optimisation parameter structure that is used to construct the solver
<i>f</i>	A reference to the objective function
<i>c</i>	A reference to the constraints function

Returns

A Solver<PSO, T, F, C> object

Definition at line 80 of file lbestpso.h.

```
80                                     :
81     Solver_base<Solver<PSOL, T, F, C>, PSOL, T, F, C>(i_pso, f, c),
82     pso(this->solver_struct),
83     w(i_pso.w),
84     vmax(i_pso.vmax),
85     nneigh(i_pso.npop),
86     neighbours(set_neighbourhoods())
87 {
88     velocity.resize(pso.npop, std::vector<T>(pso.ndv));
89     local_best.resize(nneigh);
90     for (auto& p : velocity)
91     {
92         for (auto& n : p)
93         {
94             n = 0.0;
95         }
96     }
97     for (const auto& p : this->individuals)
```



```

98         {
99             personal_best.push_back(p);
100             personal_best_cost.push_back(f(p));
101         }
102         for (auto& p : local_best)
103         {
104             p = personal_best[0];
105         }
106         for (size_t i = 0; i < pso.npop; ++i)
107         {
108             for (const auto& index : neighbours[i])
109             {
110                 if (personal_best_cost[index] < this->f(local_best[i]))
111                 {
112                     local_best[i] = personal_best[index];
113                 }
114             }
115         }
116         find_min_local_best();
117     }

```

7.13.3 Member Function Documentation

7.13.3.1 best_update()

```

template<typename T , typename F , typename C >
void ea::Solver< PSOI, T, F, C >::best_update ( ) [private]

```

This method sets the personal and local best solutions.

Returns

void

Definition at line 257 of file lbestpso.h.

```

258     {
259         for (size_t i = 0; i < pso.npop; ++i)
260         {
261             }
262     }

```

7.13.3.2 check_pso_criteria()

```

template<typename T , typename F , typename C >
bool ea::Solver< PSOI, T, F, C >::check_pso_criteria ( ) [private]

```

Define the maximum radius stopping criterion.

Returns

true if criteria are met, false otherwise

Definition at line 277 of file lbestpso.h.

```

278     {
279         std::vector<T> distance(pso.npop);
280         for (size_t i = 0; i < pso.npop; ++i)
281         {
282             distance[i] = euclid_distance(this->individuals[i], this->
min_cost);
283         }
284         T rmax = distance[0];
285         for (size_t i = 0; i < pso.npop; ++i)
286         {
287             if (rmax < distance[i])
288             {
289                 rmax = distance[i];
290             }
291             else
292             {
293             }
294         }
295         if (pso.tol > std::abs(this->f(this->min_cost))) // || rmax < pso.tol
296         {
297             return true;
298         }
299         else
300         {
301             return false;
302         }
303     }

```

7.13.3.3 display_parameters()

```

template<typename T , typename F , typename C >
ea::Solver< PSOL, T, F, C >::display_parameters ( ) [inline], [private]

```

Display PSO parameters.

Returns

A std::stringstream of the parameters

Definition at line 185 of file lbestpso.h.

References ea::PSOL< T >::c, ea::PSOL< T >::vmax, and ea::PSOL< T >::w.

```

186     {
187         std::stringstream parameters;
188         parameters << "C:" << "," << pso.c << ",";
189         parameters << "Inertia:" << "," << pso.w << ",";
190         parameters << "Maximum Velocity:" << "," << pso.vmax;
191         return parameters;
192     }

```

7.13.3.4 euclid_distance()

```

template<typename T , typename F , typename C >
ea::Solver< PSOL, T, F, C >::euclid_distance (
    const std::vector< T > & x,
    const std::vector< T > & y ) [inline], [private]

```

Euclidean Distance of two vectors ..

Parameters

<code>x,y</code>	The two vectors for which the distance is calculated
------------------	--

Returns

Distance as a floating-point number

Definition at line 172 of file lbestpso.h.

```

173     {
174         T sum = 0;
175         for (size_t i = 0; i < x.size(); ++i)
176         {
177             sum = sum + std::pow(x[i] - y[i], 2);
178         }
179         return std::sqrt(sum);
180     }

```

7.13.3.5 find_min_local_best()

```

template<typename T , typename F , typename C >
void ea::Solver< PSOI, T, F, C >::find_min_local_best ( ) [private]

```

This is a faster way to calculate the minimum cost unless there is only one neighbourhood, in which case it is the same as find_min_cost(F f)

Returns

void

Definition at line 265 of file lbestpso.h.

```

266     {
267         for (size_t k = 0; k < nneigh; ++k)
268         {
269             if (this->f(local_best[k]) < this->f(this->min_cost))
270             {
271                 this->min_cost = local_best[k];
272             }
273         }
274     }

```

7.13.3.6 position_update()

```
template<typename T , typename F , typename C >
void ea::Solver< PSOI, T, F, C >::position_update ( ) [private]
```

Position update of the particles.

Returns

void

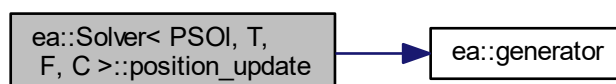
Checks that the candidate is feasible

Definition at line 221 of file lbestpso.h.

References `ea::PSOI< T >::c`, `ea::generator()`, `ea::PSOI< T >::vmax`, and `ea::PSOI< T >::w`.

```
222     {
223         for (size_t i = 0; i < pso.npop; ++i)
224         {
225             for (size_t j = 0; j < pso.ndv; ++j)
226             {
227                 std::uniform_real_distribution<double> c(0, pso.c);
228                 velocity[i][j] = w * velocity[i][j] + c(
229                     generator) * (personal_best[i][j] - this->individuals[i][j])
230                     + c(generator) * (local_best[i][j] - this->
231                     individuals[i][j]);
232                 if (velocity[i][j] > vmax[j])
233                 {
234                     velocity[i][j] = vmax[j];
235                 }
236                 this->individuals[i][j] = this->individuals[i][j] +
237                 velocity[i][j];
238             }
239             if (!this->c(this->individuals[i]))
240             {
241                 this->individuals[i] = personal_best[i];
242             }
243             if (this->f(this->individuals[i]) < this->f(
244                 personal_best[i]))
245             {
246                 personal_best[i] = this->individuals[i];
247                 personal_best_cost[i] = this->f(this->
248                 individuals[i]);
249             }
250             for (const auto& index : neighbours[i])
251             {
252                 if (personal_best_cost[index] < this->f(
253                     local_best[i]))
254                 {
255                     local_best[i] = personal_best[index];
256                 }
257             }
258         }
259     }
```

Here is the call graph for this function:



7.13.3.7 run_algo()

```
template<typename T , typename F , typename C >
void ea::Solver< PSOI, T, F, C >::run_algo ( ) [private]
```

Runs the algorithm until stopping criteria return void.

Local Best Particle Swarm starts here

Inertia weight is updated - Linear

Non-linear

Definition at line 310 of file lbestpso.h.

References ea::PSOI< T >::w.

```
311     {
313         for (size_t iter = 0; iter < pso.iter_max; ++iter)
314         {
315             position_update();
316             //best_update();
317             find_min_local_best();
319             //w = pso.w - (pso.w - 0.4) * (static_cast<T>(iter) / static_cast<T>(pso.iter_max));
321             w = pso.w - (pso.w - 0.4) * std::pow((static_cast<T>(iter) / static_cast<T>(
pso.iter_max)), inv_pi_sq<T>);
322             //w = 0.729;
323             //w = 0.5 + distribution(generator) / 2;
324             this->last_iter = iter;
325             if (check_pso_criteria())
326             {
327                 this->solved_flag = true;
328                 break;
329             }
330         }
331     }
```

7.13.3.8 set_neighbourhoods()

```
template<typename T , typename F , typename C >
std::unordered_map< size_t, std::array< size_t, 3 > > ea::Solver< PSOI, T, F, C >::set_↵
neighbourhoods ( ) [private]
```

Set the neighbourhoods of the algorithm using particle indices.

Returns

A map matching particle indices to neighbourhoods

Definition at line 196 of file lbestpso.h.

```
197     {
198         std::unordered_map<size_t, std::array<size_t, 3>> neighbours;
199         for (size_t i = 0; i < pso.npop; ++i)
200         {
201             if (i == 0)
202             {
203                 neighbours[i] = { pso.npop - 1, i, i + 1 };
204             }
205             else
206             {
207                 if (i == pso.npop - 1)
208                 {
209                     neighbours[i] = { pso.npop - 2, i, 0 };
210                 }
211                 else
212                 {
213                     neighbours[i] = { i - 1, i, i + 1 };
214                 }
215             }
216         }
217         return neighbours;
218     }
```

7.13.4 Friends And Related Function Documentation

7.13.4.1 Solver_base< Solver< PSOL, T, F, C >, PSOL, T, F, C >

```
template<typename T , typename F , typename C >
friend class Solver_base< Solver< PSOL, T, F, C >, PSOL, T, F, C > [friend]
```

Definition at line 72 of file lbestpso.h.

7.13.5 Member Data Documentation

7.13.5.1 local_best

```
template<typename T , typename F , typename C >
std::vector<std::vector<T> > ea::Solver< PSOL, T, F, C >::local_best [private]
```

Local best vector, holds the best position recorded for each neighbourhood.

Definition at line 130 of file lbestpso.h.

7.13.5.2 neighbours

```
template<typename T , typename F , typename C >
std::unordered_map<size_t, std::array<size_t, 3> > ea::Solver< PSOL, T, F, C >::neighbours
[private]
```

Neighbours of each particle.

Definition at line 136 of file lbestpso.h.

7.13.5.3 nneigh

```
template<typename T , typename F , typename C >
const size_t ea::Solver< PSOL, T, F, C >::nneigh [private]
```

Number of neighbourhoods.

Definition at line 134 of file lbestpso.h.

7.13.5.4 personal_best

```
template<typename T , typename F , typename C >
std::vector<std::vector<T> > ea::Solver< PSOL, T, F, C >::personal_best [private]
```

Personal best vector of the particles, holds the best position recorded for each particle.

Definition at line 126 of file lbestpso.h.

7.13.5.5 personal_best_cost

```
template<typename T , typename F , typename C >
std::vector<T> ea::Solver< PSOL, T, F, C >::personal_best_cost [private]
```

Personal best cost vector of the particles.

Definition at line 128 of file lbestpso.h.

7.13.5.6 pso

```
template<typename T , typename F , typename C >
const PSOL<T>& ea::Solver< PSOL, T, F, C >::pso [private]
```

Particle Swarm Optimisation structure used internally (reference to solver_struct)

Definition at line 120 of file lbestpso.h.

7.13.5.7 velocity

```
template<typename T , typename F , typename C >
std::vector<std::vector<T> > ea::Solver< PSOL, T, F, C >::velocity [private]
```

Velocity of the particles.

Definition at line 132 of file lbestpso.h.

7.13.5.8 vmax

```
template<typename T , typename F , typename C >
std::vector<T> ea::Solver< PSOL, T, F, C >::vmax [private]
```

Maximum Velocity is mutable, so a copy is created.

Definition at line 124 of file lbestpso.h.

7.13.5.9 w

```
template<typename T , typename F , typename C >  
T ea::Solver< PSOl, T, F, C >::w [private]
```

Inertia is mutable, so a copy is created.

Definition at line 122 of file lbestpso.h.

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

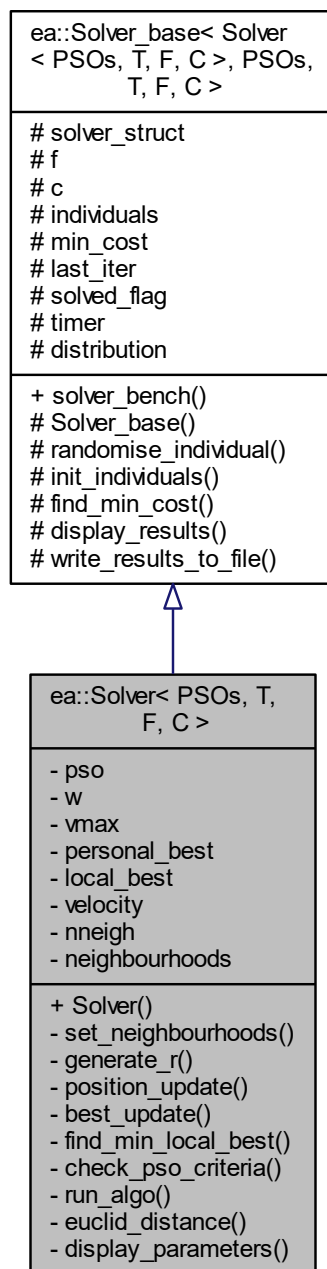
- [lbestpso.h](#)

7.14 ea::Solver< PSOs, T, F, C > Class Template Reference

Sub-Swarm Particle Swarm Optimisation (PSO) Class.

```
#include <pso_sub_swarm.h>
```


Inheritance diagram for ea::Solver< PSOs, T, F, C >:



Public Member Functions

- [Solver](#) (const [PSOs](#)< T > &i_pso, F f, C c)

Constructor.

Private Member Functions

- `std::unordered_map< size_t, size_t > set_neighbourhoods ()`
Set the neighbourhoods of the algorithm using particle indices.
- `std::vector< std::vector< T > > generate_r ()`
This method generates r_1 and r_2 for the velocity update rule.
- `void position_update ()`
Position update of the particles.
- `void best_update ()`
This method sets the personal and local best solutions.
- `void find_min_local_best ()`
This is a faster way to calculate the minimum cost unless there is only one neighbourhood, in which case it is the same as `find_min_cost(F f)`
- `bool check_pso_criteria ()`
Define the maximum radius stopping criterion.
- `void run_algo ()`
Runs the algorithm until stopping criteria return void.
- `T euclid_distance (const std::vector< T > &x, const std::vector< T > &y)`
Euclidean Distance of two vectors ..
- `std::stringstream display_parameters ()`
Display PSO parameters.

Private Attributes

- `const PSOs< T > & pso`
Particle Swarm Optimisation structure used internally (reference to `solver_struct`)
- `T w`
Inertia is mutable, so a copy is created.
- `std::vector< T > vmax`
Maximum Velocity is mutable, so a copy is created.
- `std::vector< std::vector< T > > personal_best`
Personal best vector of the particles, holds the best position recorded for each particle.
- `std::vector< std::vector< T > > local_best`
Local best vector, holds the best position recorded for each neighbourhood.
- `std::vector< std::vector< T > > velocity`
Velocity of the particles.
- `const size_t nneigh`
Number of neighbourhoods.
- `std::unordered_map< size_t, size_t > neighbourhoods`
Neighbourhoods.

Friends

- `class Solver_base< Solver< PSOs, T, F, C >, PSOs, T, F, C >`

Additional Inherited Members

7.14.1 Detailed Description

```
template<typename T, typename F, typename C>
class ea::Solver< PSOs, T, F, C >
```

Sub-Swarm Particle Swarm Optimisation (PSO) Class.

Definition at line 83 of file pso_sub_swarm.h.

7.14.2 Constructor & Destructor Documentation

7.14.2.1 Solver()

```
template<typename T , typename F , typename C >
ea::Solver< PSOs, T, F, C >::Solver (
    const PSOs< T > & i_pso,
    F f,
    C c ) [inline]
```

Constructor.

Parameters

<i>i_pso</i>	The particle swarm optimisation parameter structure that is used to construct the solver
<i>f</i>	A reference to the objective function
<i>c</i>	A reference to the constraints function

Returns

A [Solver<PSOs, T, F, C>](#) object

Definition at line 94 of file pso_sub_swarm.h.

```

94                                     :
95     Solver_base<Solver<PSOs, T, F, C>, PSOs, T, F, C>( i_pso, f, c ),
96     pso( this->solver_struct ),
97     w( i_pso.w ),
98     vmax( i_pso.vmax ),
99     nneigh( static_cast<size_t>(std::ceil(i_pso.npop / i_pso.sneigh)) ),
100     neighbourhoods( set_neighbourhoods() )
101 {
102     velocity.resize(pso.npop, std::vector<T>(pso.ndv));
103     local_best.resize(nneigh);
104     for (auto& p : velocity)
105     {
106         for (auto& n : p)
107         {
108             n = 0.0;
109         }
110     }
111     for (const auto& p : this->individuals)
```

```

112         {
113             personal_best.push_back(p);
114         }
115         for (auto& p : local_best)
116         {
117             p = personal_best[0];
118         }
119         for (size_t i = 0; i < pso.npop; ++i)
120         {
121             if (f(personal_best[i]) < f(local_best[
neighbourhoods[i]]))
122             {
123                 local_best[neighbourhoods[i]] = personal_best[i];
124             }
125         }
126         find_min_local_best();
127     }

```

7.14.3 Member Function Documentation

7.14.3.1 best_update()

```

template<typename T , typename F , typename C >
void ea::Solver< PSOs, T, F, C >::best_update ( ) [private]

```

This method sets the personal and local best solutions.

Returns

void

Checks that the candidate is feasible

Definition at line 268 of file pso_sub_swarm.h.

```

269     {
270         for (size_t i = 0; i < pso.npop; ++i)
271         {
272             if (!this->c(this->individuals[i]))
273             {
274                 this->individuals[i] = personal_best[i];
275             }
276             if (this->f(this->individuals[i]) < this->f(
personal_best[i]))
277             {
278                 personal_best[i] = this->individuals[i];
279             }
280             if (this->f(personal_best[i]) < this->f(local_best[
neighbourhoods[i]]))
281             {
282                 local_best[neighbourhoods[i]] =
personal_best[i];
283             }
284         }
285     }
286 }

```

7.14.3.2 check_pso_criteria()

```
template<typename T , typename F , typename C >
bool ea::Solver< PSOs, T, F, C >::check_pso_criteria ( ) [private]
```

Define the maximum radius stopping criterion.

Returns

true if criteria are met, false otherwise

Definition at line 301 of file pso_sub_swarm.h.

```
302     {
303         std::vector<T> distance(pso.npop);
304         for (size_t i = 0; i < pso.npop; ++i)
305         {
306             distance[i] = euclid_distance(this->individuals[i], this->
min_cost);
307         }
308         T rmax = distance[0];
309         for (size_t i = 0; i < pso.npop; ++i)
310         {
311             if (rmax < distance[i])
312             {
313                 rmax = distance[i];
314             }
315             else
316             {
317             }
318         }
319         if (pso.tol > std::abs(this->f(this->min_cost))) //|| rmax < pso.tol)
320         {
321             return true;
322         }
323         else
324         {
325             return false;
326         }
327     }
```

7.14.3.3 display_parameters()

```
template<typename T , typename F , typename C >
ea::Solver< PSOs, T, F, C >::display_parameters ( ) [inline], [private]
```

Display PSO parameters.

Returns

A std::stringstream of the parameters

Definition at line 198 of file pso_sub_swarm.h.

References ea::PSOs< T >::alpha, ea::PSOs< T >::c1, ea::PSOs< T >::c2, ea::PSOs< T >::sneigh, ea::PSOs< T >::vmax, and ea::PSOs< T >::w.

```
199     {
200         std::stringstream parameters;
201         parameters << "C1:" << "," << pso.c1 << ",";
202         parameters << "C2:" << "," << pso.c2 << ",";
203         parameters << "Neighbourhood size:" << "," << pso.sneigh << ",";
204         parameters << "Inertia:" << "," << pso.w << ",";
205         parameters << "Alpha parameter for inertia:" << "," << pso.alpha << ",";
206         parameters << "Maximum Velocity:" << "," << pso.vmax;
207         return parameters;
208     }
```

7.14.3.4 euclid_distance()

```
template<typename T , typename F , typename C >
ea::Solver< PSOs, T, F, C >::euclid_distance (
    const std::vector< T > & x,
    const std::vector< T > & y ) [inline], [private]
```

Euclidean Distance of two vectors ..

Parameters

x,y	The two vectors for which the distance is calculated
-----	--

Returns

Distance as a floating-point number

Definition at line 185 of file pso_sub_swarm.h.

```
186     {
187         T sum = 0;
188         for (size_t i = 0; i < x.size(); ++i)
189         {
190             sum = sum + std::pow(x[i] - y[i], 2);
191         }
192         return std::sqrt(sum);
193     }
```

7.14.3.5 find_min_local_best()

```
template<typename T , typename F , typename C >
void ea::Solver< PSOs, T, F, C >::find_min_local_best ( ) [private]
```

This is a faster way to calculate the minimum cost unless there is only one neighbourhood, in which case it is the same as find_min_cost(F f)

Returns

void

Definition at line 289 of file pso_sub_swarm.h.

```
290     {
291         for (size_t k = 0; k < nneigh; ++k)
292         {
293             if (this->f(local_best[k]) < this->f(this->min_cost))
294             {
295                 this->min_cost = local_best[k];
296             }
297         }
298     }
```

7.14.3.6 generate_r()

```
template<typename T , typename F , typename C >
std::vector< std::vector< T > > ea::Solver< PSOs, T, F, C >::generate_r ( ) [private]
```

This method generates r1 and r2 for the velocity update rule.

Returns

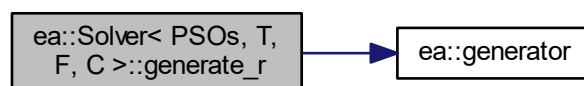
A vector containing r1 and r2

Definition at line 234 of file pso_sub_swarm.h.

References `ea::generator()`.

```
235     {
236         std::vector<std::vector<T>> r(3, std::vector<T>(pso.ndv));
237         for (auto i = 0; i < 3; ++i)
238         {
239             for (size_t j = 0; j < pso.ndv; ++j)
240             {
241                 r[i][j] = (this->distribution(generator));
242             }
243         }
244         return r;
245     }
```

Here is the call graph for this function:



7.14.3.7 position_update()

```
template<typename T , typename F , typename C >
void ea::Solver< PSOs, T, F, C >::position_update ( ) [private]
```

Position update of the particles.

Returns

void

Definition at line 248 of file pso_sub_swarm.h.

References `ea::PSOs< T >::vmax`.

```

249     {
250         for (size_t i = 0; i < pso.npop; ++i)
251         {
252             for (size_t j = 0; j < pso.ndv; ++j)
253             {
254                 const auto& r = generate_r();
255                 velocity[i][j] = 0.729 * velocity[i][j] + //pso.c1 * r[0][j] *
                (personal_best[i][j] - this->individuals[i][j])
256                 + pso.c2 * r[1][j] * (local_best[neighbourhoods[i]][j] -
                this->individuals[i][j]) //+(w / 2) * r[2][j]*(min_cost[j] - this->individuals[i][j]);
257             }
258             if (velocity[i][j] > vmax[j])
259             {
260                 velocity[i][j] = vmax[j];
261             }
262             this->individuals[i][j] = this->individuals[i][j] +
                velocity[i][j];
263         }
264     }
265 }

```

7.14.3.8 run_algo()

```

template<typename T , typename F , typename C >
void ea::Solver< PSOs, T, F, C >::run_algo ( ) [private]

```

Runs the algorithm until stopping criteria return void.

Local Best Particle Swarm starts here

Inertia is updated

Definition at line 334 of file pso_sub_swarm.h.

References `ea::PSOs< T >::w`.

```

335     {
337         for (size_t iter = 0; iter < pso.iter_max; ++iter)
338         {
339             position_update();
340             best_update();
341             find_min_local_best();
342             w = pso.w - (pso.w - 0.4) * std::pow((static_cast<T>(iter) / static_cast<T>(
343 pso.iter_max)), inv_pi_sq_2<T>);
344             this->last_iter = iter;
345             if (check_pso_criteria())
346             {
347                 this->solved_flag = true;
348                 break;
349             }
350         }
351     }

```


7.14.3.9 set_neighbourhoods()

```
template<typename T , typename F , typename C >
std::unordered_map< size_t, size_t > ea::Solver< PSOs, T, F, C >::set_neighbourhoods ( )
[private]
```

Set the neighbourhoods of the algorithm using particle indices.

Returns

A map matching particle indices to neighbourhoods

Definition at line 212 of file pso_sub_swarm.h.

```
213     {
214         std::unordered_map<size_t, size_t> neighbourhoods;
215         size_t neigh_index = 0;
216         size_t counter = 0;
217         for (size_t i = 0; i < pso.npop; ++i)
218         {
219             if (counter < pso.sneigh)
220             {
221                 neighbourhoods[i] = neigh_index;
222                 counter = counter + 1;
223             }
224             else
225             {
226                 neigh_index = neigh_index + 1;
227                 counter = 0;
228             }
229         }
230         return neighbourhoods;
231     }
```

7.14.4 Friends And Related Function Documentation

7.14.4.1 Solver_base< Solver< PSOs, T, F, C >, PSOs, T, F, C >

```
template<typename T , typename F , typename C >
friend class Solver_base< Solver< PSOs, T, F, C >, PSOs, T, F, C > [friend]
```

Definition at line 86 of file pso_sub_swarm.h.

7.14.5 Member Data Documentation

7.14.5.1 local_best

```
template<typename T , typename F , typename C >
std::vector<std::vector<T> > ea::Solver< PSOs, T, F, C >::local_best [private]
```

Local best vector, holds the best position recorded for each neighbourhood.

Definition at line 138 of file pso_sub_swarm.h.

7.14.5.2 neighbourhooDs

```
template<typename T , typename F , typename C >
std::unordered_map<size_t, size_t> ea::Solver< PSOs, T, F, C >::neighbourhooDs [private]
```

NeighbourhooDs.

Definition at line 144 of file pso_sub_swarm.h.

7.14.5.3 nneigh

```
template<typename T , typename F , typename C >
const size_t ea::Solver< PSOs, T, F, C >::nneigh [private]
```

Number of neighbourhooDs.

Definition at line 142 of file pso_sub_swarm.h.

7.14.5.4 personal_best

```
template<typename T , typename F , typename C >
std::vector<std::vector<T> > ea::Solver< PSOs, T, F, C >::personal_best [private]
```

Personal best vector of the particles, holds the best position recorded for each particle.

Definition at line 136 of file pso_sub_swarm.h.

7.14.5.5 pso

```
template<typename T , typename F , typename C >
const PSOs<T>& ea::Solver< PSOs, T, F, C >::pso [private]
```

Particle Swarm Optimisation structure used internally (reference to solver_struct)

Definition at line 130 of file pso_sub_swarm.h.

7.14.5.6 velocity

```
template<typename T , typename F , typename C >
std::vector<std::vector<T> > ea::Solver< PSOs, T, F, C >::velocity [private]
```

Velocity of the particles.

Definition at line 140 of file pso_sub_swarm.h.

7.14.5.7 vmax

```
template<typename T , typename F , typename C >
std::vector<T> ea::Solver< PSOs, T, F, C >::vmax [private]
```

Maximum Velocity is mutable, so a copy is created.

Definition at line 134 of file pso_sub_swarm.h.

7.14.5.8 w

```
template<typename T , typename F , typename C >
T ea::Solver< PSOs, T, F, C >::w [private]
```

Inertia is mutable, so a copy is created.

Definition at line 132 of file pso_sub_swarm.h.

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

- [pso_sub_swarm.h](#)

7.15 ea::Solver_base< Derived, S, T, F, C > Class Template Reference

Base Class for Evolutionary Algorithms.

```
#include <ealgorithm_base.h>
```

Public Member Functions

- `std::vector< T > solver_bench` (const std::string &problem_name)
Solve wrapper function for Solvers, used for benchmarks.

Protected Member Functions

- `Solver_base` (const S< T > &i_solver_struct, const F &i_f, const C &i_c)
Constructor.
- `std::vector< T > randomise_individual` ()
Returns a randomised individual using the initial decision variables and standard deviation.
- `std::vector< std::vector< T > > init_individuals` ()
Initialises the population by randomising around the decision variables using the given standard deviation.
- `void find_min_cost` ()
Find the minimum cost individual of the fitness function for the population.
- `std::stringstream display_results` ()
Display the results of execution of an algorithm as well as its parameters.
- `void write_results_to_file` (const std::string &problem_name)
Write the results to a file.

Protected Attributes

- `const S< T > solver_struct`
Internal copy of the structure used for parameters of the algorithm.
- `F f`
Copy of the fitness function passed as a lambda.
- `C c`
Copy of the constraints function passed as a lambda.
- `std::vector< std::vector< T > > individuals`
Population.
- `std::vector< T > min_cost`
Best solution / lowest fitness.
- `size_t last_iter`
Last iteration to solution.
- `bool solved_flag`
A flag which determines if the solver has already solved the problem.
- `T timer`
The timer used for benchmarks.
- `std::uniform_real_distribution< T > distribution`
Uniform real distribution.

7.15.1 Detailed Description

```
template<typename Derived, template< typename > class S, typename T, typename F, typename C>
class ea::Solver_base< Derived, S, T, F, C >
```

Base Class for Evolutionary Algorithms.

The fitness and constraints functions are copied so that even if they are not available in the current scope, the solver will still execute properly. At the same time, `std::function` could be have used, thus eliminating the need for template parameters F and C. However, that comes at a runtime cost, since calls to the functions would be virtual and there is a possibility that allocation could happen on the heap.

Definition at line 103 of file `ealgorithm_base.h`.

7.15.2 Constructor & Destructor Documentation

7.15.2.1 Solver_base()

```
template<typename Derived, template< typename > class S, typename T, typename F, typename C>
ea::Solver_base< Derived, S, T, F, C >::Solver_base (
    const S< T > & i_solver_struct,
    const F & i_f,
    const C & i_c ) [inline], [protected]
```

Constructor.

Parameters

<i>i_solver_struct</i>	The parameter structure that is used to construct the solver
<i>i_f</i>	A reference to the objective function
<i>i_c</i>	A reference to the constraints function

Returns

A Solver_base<Derived, S, T, F, C> object

Definition at line 120 of file ealgorithm_base.h.

```

120                                     :
121         solver_struct{ i_solver_struct },
122         f{ i_f },
123         c{ i_c },
124         individuals{ init_individuals() },
125         min_cost{ individuals[0] },
126         last_iter{ 0 },
127         solved_flag{ false },
128         timer{ 0 },
129         distribution{ std::uniform_real_distribution<T>(0.0, 1.0) }
130     {
131         generator.discard(700000);
132         find_min_cost();
133     }

```

7.15.3 Member Function Documentation

7.15.3.1 display_results()

```

template<typename Derived , template< typename > class S, typename T , typename F , typename
C >
std::stringstream ea::Solver_base< Derived, S, T, F, C >::display_results ( ) [protected]

```

Display the results of execution of an algorithm as well as its parameters.

Returns

A std::stringstream of the results

Definition at line 223 of file ealgorithm_base.h.

```

224     {
225         std::stringstream results;
226         results << "Algorithm:" << ", " << solver_struct.type << ", " << "Solved:" << ", ";
227         if (!solved_flag)
228         {
229             results << "False" << ", ";
230         }
231         else
232         {
233             results << "True" << ", ";
234         }
235         results << "Solution:" << ", " << min_cost << ", ";
236         results << "Fitness:" << ", " << f(min_cost) << ", ";
237         results << "Population:" << ", " << individuals.size() << ", ";
238         results << "Iterations:" << ", " << last_iter << ", ";

```

```

239     results << "Elapsed Time:" << "," << timer << ",";
240     results << "Starting Values:" << "," << solver_struct.decision_variables << ",";
241     results << "Standard Deviation:" << "," << solver_struct.stdev << ",";
242     results << "Initial Population:" << "," << solver_struct.npop << ",";
243     results << "Tolerance:" << "," << solver_struct.tol << ",";
244     results << "Maximum Iterations:" << "," << solver_struct.iter_max << ",";
245     results << "Using Penalty Function:" << "," << solver_struct.use_penalty_method << ","
;
246     results << "Using Constraints:" << ",";
247     switch (solver_struct.constraints_type)
248     {
249     case(Constraints_type::normal): results << "Normal" << ","; break;
250     case(Constraints_type::tight): results << "Tight" << ","; break;
251     case(Constraints_type::none): results << "None" << ","; break;
252     }
253     results << static_cast<Derived*>(this)->display_parameters().str() << "\n";
254     return results;
255 }

```

7.15.3.2 find_min_cost()

```

template<typename Derived , template< typename > class S, typename T , typename F , typename
C >
void ea::Solver_base< Derived, S, T, F, C >::find_min_cost ( ) [protected]

```

Find the minimum cost individual of the fitness function for the population.

Returns

void

Definition at line 211 of file ealgorithm_base.h.

```

212     {
213         for (const auto& p : individuals)
214         {
215             if (f(min_cost) > f(p))
216             {
217                 min_cost = p;
218             }
219         }
220     }

```

7.15.3.3 init_individuals()

```

template<typename Derived , template< typename > class S, typename T , typename F , typename
C >
std::vector< std::vector< T > > ea::Solver_base< Derived, S, T, F, C >::init_individuals ( )
[protected]

```

Initialises the population by randomising around the decision variables using the given standard deviation.

Returns

The population after checking the constraints of the optimisation problem

Check population constraints

Definition at line 195 of file ealgorithm_base.h.

```

196     {
197         std::vector<std::vector<T>> individuals(solver_struct.npop, std::vector<T>(
198             solver_struct.ndv));
199         for (auto& p : individuals)
200         {
201             p = randomise_individual();
202             while (!c(p))
203             {
204                 p = randomise_individual();
205             }
206         }
207         return individuals;
208     }

```

7.15.3.4 randomise_individual()

```

template<typename Derived , template< typename > class S, typename T , typename F , typename
C >
std::vector< T > ea::Solver_base< Derived, S, T, F, C >::randomise_individual ( ) [protected]

```

Returns a randomised individual using the initial decision variables and standard deviation.

Returns

A randomised individual of type std::vector<T>, where T is a floating-point number type.

Definition at line 181 of file ealgorithm_base.h.

```

182     {
183         std::vector<T> individual = solver_struct.decision_variables;
184         T epsilon = 0;
185         for (size_t j = 0; j < solver_struct.ndv; ++j)
186         {
187             std::normal_distribution<T> ndistribution(0, solver_struct.stdev[j]);
188             epsilon = ndistribution(generator);
189             individual[j] = individual[j] + epsilon;
190         }
191         return individual;
192     }

```

7.15.3.5 solver_bench()

```

template<typename Derived , template< typename > class S, typename T , typename F , typename
C >
std::vector< T > ea::Solver_base< Derived, S, T, F, C >::solver_bench (
    const std::string & problem_name )

```

Solve wrapper function for Solvers, used for benchmarks.

Parameters

<i>problem_name</i>	The name of the problem in std::string form
---------------------	---

Returns

The solution vector

Time the computation

Return minimum cost individual

Definition at line 271 of file ealgorithm_base.h.

```

272     {
273         if (solver_struct.tol > std::abs(f(min_cost)))
274         {
275             timer = 0;
276         }
277         else
278         {
279             const std::chrono::time_point<std::chrono::system_clock> start = std::chrono::system_clock::now
280 ();
281             static_cast<Derived*>(this)->run_algo();
282             const std::chrono::time_point<std::chrono::system_clock> end = std::chrono::system_clock::now()
283 ;
284             const std::chrono::duration<double> elapsed_seconds = end - start;
285             timer = elapsed_seconds.count();
286         }
287         if (solver_struct.print_to_output)
288         {
289             std::cout << display_results().str();
290         }
291         else {};
292         if (solver_struct.print_to_file)
293         {
294             write_results_to_file(problem_name);
295         }
296         else {};
297         return min_cost;
298     }

```

7.15.3.6 write_results_to_file()

```

template<typename Derived , template< typename > class S, typename T , typename F , typename
C >
void ea::Solver_base< Derived, S, T, F, C >::write_results_to_file (
    const std::string & problem_name ) [protected]

```

Write the results to a file.

Parameters

<i>problem_name</i>	The name of the problem in std::string form
---------------------	---

Returns

void

Definition at line 258 of file ealgorithm_base.h.

```

259     {
260         std::string filename;
261         filename.append(problem_name);
262         //filename.append("-");
263         //filename.append(solver_struct.type);
264         filename.append("-results.csv");
265         std::ofstream out;
266         out.open(filename, std::ofstream::out | std::ofstream::app);
267         out << display_results().str();
268     }

```

7.15.4 Member Data Documentation

7.15.4.1 c

```

template<typename Derived, template< typename > class S, typename T, typename F, typename C>
C ea::Solver_base< Derived, S, T, F, C >::c [protected]

```

Copy of the constraints function passed as a lambda.

Definition at line 139 of file ealgorithm_base.h.

7.15.4.2 distribution

```

template<typename Derived, template< typename > class S, typename T, typename F, typename C>
std::uniform_real_distribution<T> ea::Solver_base< Derived, S, T, F, C >::distribution [protected]

```

Uniform real distribution.

Definition at line 151 of file ealgorithm_base.h.

7.15.4.3 f

```

template<typename Derived, template< typename > class S, typename T, typename F, typename C>
F ea::Solver_base< Derived, S, T, F, C >::f [protected]

```

Copy of the fitness function passed as a lambda.

Definition at line 137 of file ealgorithm_base.h.

7.15.4.4 individuals

```
template<typename Derived, template< typename > class S, typename T, typename F, typename C>
std::vector<std::vector<T> > ea::Solver\_base< Derived, S, T, F, C >::individuals [protected]
```

Population.

Definition at line 141 of file `ealgorithm_base.h`.

7.15.4.5 last_iter

```
template<typename Derived, template< typename > class S, typename T, typename F, typename C>
size_t ea::Solver\_base< Derived, S, T, F, C >::last_iter [protected]
```

Last iteration to solution.

Definition at line 145 of file `ealgorithm_base.h`.

7.15.4.6 min_cost

```
template<typename Derived, template< typename > class S, typename T, typename F, typename C>
std::vector<T> ea::Solver\_base< Derived, S, T, F, C >::min_cost [protected]
```

Best solution / lowest fitness.

Definition at line 143 of file `ealgorithm_base.h`.

7.15.4.7 solved_flag

```
template<typename Derived, template< typename > class S, typename T, typename F, typename C>
bool ea::Solver\_base< Derived, S, T, F, C >::solved_flag [protected]
```

A flag which determines if the solver has already solved the problem.

Definition at line 147 of file `ealgorithm_base.h`.

7.15.4.8 solver_struct

```
template<typename Derived, template< typename > class S, typename T, typename F, typename C>
const S<T> ea::Solver\_base< Derived, S, T, F, C >::solver_struct [protected]
```

Internal copy of the structure used for parameters of the algorithm.

Definition at line 135 of file `ealgorithm_base.h`.

7.15.4.9 timer

```
template<typename Derived, template< typename > class S, typename T, typename F, typename C>
T ea::Solver\_base< Derived, S, T, F, C >::timer [protected]
```

The timer used for benchmarks.

Definition at line 149 of file `ealgorithm_base.h`.

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

- [ealgorithm_base.h](#)

Chapter 8

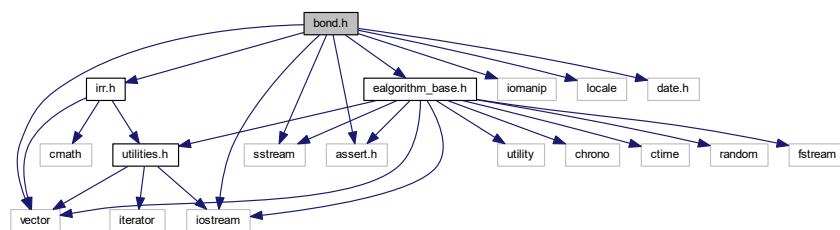
File Documentation

8.1 bond.h File Reference

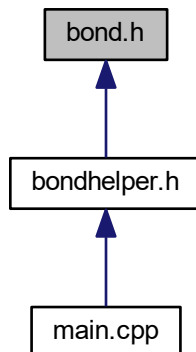
Classes and functions for bonds and their internal rate of return.

```
#include <iostream>
#include <vector>
#include <iomanip>
#include <sstream>
#include <locale>
#include <assert.h>
#include "date.h"
#include "irr.h"
#include "ealgorithm_base.h"
```

Include dependency graph for bond.h:



This graph shows which files directly or indirectly include this file:



Classes

- class [bond::BondHelper< T >](#)
A class for the bond pricing problem as well as finding the yield-to-maturities of bonds.
- class [bond::Bond< T >](#)
[Bond](#) Class definition.

Namespaces

- [bond](#)
[Bond](#) Class and Utilities.

8.1.1 Detailed Description

Classes and functions for bonds and their internal rate of return.

Author

Ioannis Anagnostopoulos

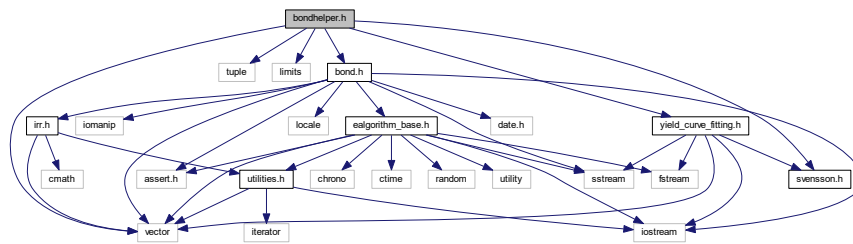
8.2 bondhelper.h File Reference

Classes and functions for the bond pricing problem.

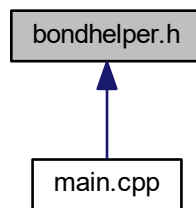
```
#include <vector>
#include <tuple>
#include <limits>
#include "bond.h"
#include "svensson.h"
```

```
#include "yield_curve_fitting.h"
```

Include dependency graph for bondhelper.h:



This graph shows which files directly or indirectly include this file:



Classes

- class [bond::BondHelper< T >](#)
A class for the bond pricing problem as well as finding the yield-to-maturities of bonds.

Namespaces

- [bond](#)
Bond Class and Utilities.

Enumerations

- enum [bond::Bond_pricing_type](#) { [bond::Bond_pricing_type::bpp](#), [bond::Bond_pricing_type::bpy](#) }
Enumeration for type of bondpricing, using yields or prices.

Functions

- template<typename T >
std::vector< Bond< T > > [bond::read_bonds_from_file](#) (const std::string &filename)
Reads bond data from a file.

8.2.1 Detailed Description

Classes and functions for the bond pricing problem.

Author

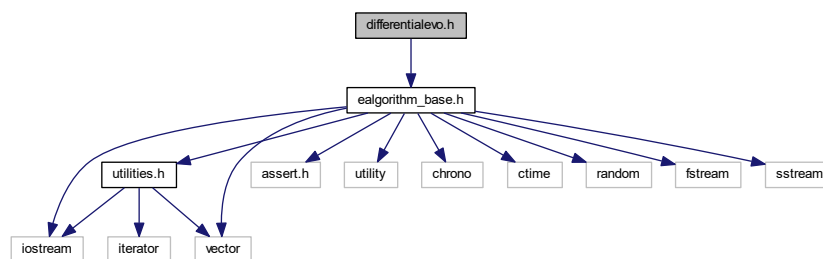
Ioannis Anagnostopoulos

8.3 differentialevo.h File Reference

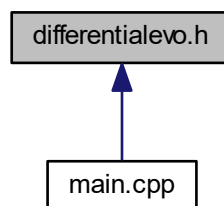
Classes and functions for Differential Evolution.

```
#include "ealgorithm_base.h"
```

Include dependency graph for differentialevo.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct `ea::DE< T >`
Differential Evolution Structure, used in the actual algorithm and for type deduction.
- class `ea::Solver< DE, T, F, C >`
Differential Evolution Algorithm (DE) Class.

Namespaces

- [ea](#)

Evolutionary Algorithms.

8.3.1 Detailed Description

Classes and functions for Differential Evolution.

Author

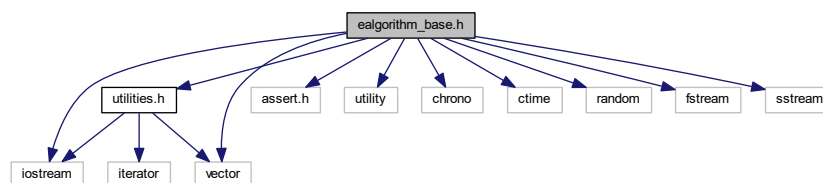
Ioannis Anagnostopoulos

8.4 ealgorithm_base.h File Reference

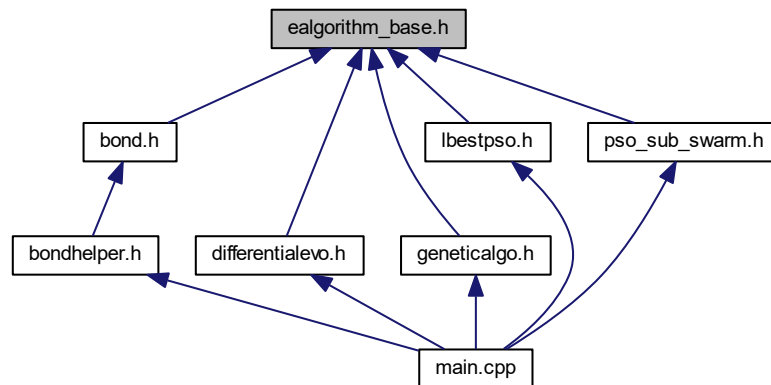
Classes and functions for the base of the solvers.

```
#include <iostream>
#include <vector>
#include <assert.h>
#include <utility>
#include <chrono>
#include <ctime>
#include <random>
#include <fstream>
#include <sstream>
#include "utilities.h"
```

Include dependency graph for ealgorithm_base.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct `ea::EA_base< T >`
Evolutionary algorithm stucture base.
- class `ea::Solver< S, T, F, C >`
Template Class for Solvers.
- class `ea::Solver_base< Derived, S, T, F, C >`
Base Class for Evolutionary Algorithms.

Namespaces

- `ea`
Evolutionary Algorithms.

Functions

- `std::mt19937_64 ea::generator (rd())`
Pseudo-random number generator.
- `template<typename F, typename C, template< typename > class S, typename T >
std::vector< T > ea::solve (const F &f, const C &c, const S< T > &solver_struct, const std::string &problem←
_name)`
Solver wrapper function, interface to solvers : free function used for benchmarks.

Variables

- `std::random_device ea::rd`
Random device / Random number generator.

8.4.1 Detailed Description

Classes and functions for the base of the solvers.

Author

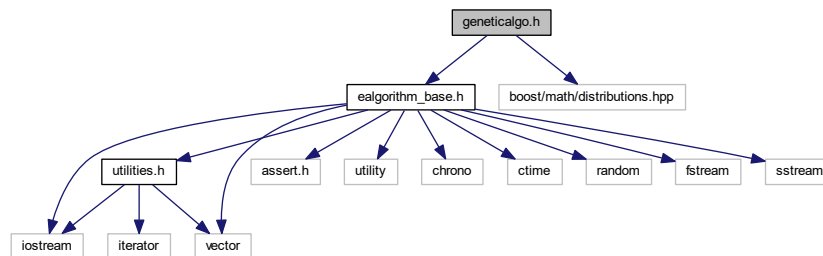
Ioannis Anagnostopoulos

8.5 geneticalgo.h File Reference

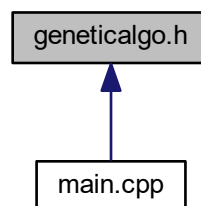
Classes and functions for Genetic Algorithms.

```
#include "ealgorithm_base.h"
#include <boost/math/distributions.hpp>
```

Include dependency graph for geneticalgo.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct `ea::GA< T >`
Genetic Algorithms Structure, used in the actual algorithm and for type deduction.
- class `ea::Solver< GA, T, F, C >`
Genetic Algorithms (GA) Class.

Namespaces

- [ea](#)

Evolutionary Algorithms.

Enumerations

- enum [ea::Strategy](#) { [ea::Strategy::keep_same](#), [ea::Strategy::re_mutate](#), [ea::Strategy::remove](#), [ea::Strategy::none](#) }

Replacing or remove individuals strategies during mutation.

8.5.1 Detailed Description

Classes and functions for Genetic Algorithms.

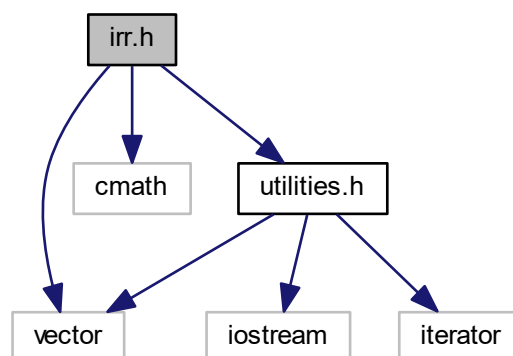
Author

Ioannis Anagnostopoulos

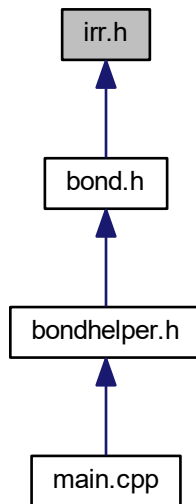
8.6 irr.h File Reference

Functions for the Internal Rate of Return.

```
#include <vector>
#include <cmath>
#include "utilities.h"
Include dependency graph for irr.h:
```



This graph shows which files directly or indirectly include this file:



Namespaces

- [irr](#)

Internal Rate of Return (IRR) namespace.

Functions

- `template<typename T >`
`T irr::compute_discount_factor` (const T &r, const T &period, const [DF_type](#) &df_type)
Calculates discount factors.
- `template<typename T >`
`bool irr::constraints_irr` (const std::vector< T > &solution, const [Constraints_type](#) &constraints_type)
Constraints function for Internal Rate of Return.
- `template<typename T >`
`T irr::compute_pv` (const T &r, const T &nominal_value, const std::vector< T > &cash_flows, const std::vector< T > &time_periods, const [DF_type](#) &df_type)
Returns the present value of an investment.
- `template<typename T >`
`T irr::penalty_irr` (const T &r)
Penalty function for IRR.
- `template<typename T >`
`T irr::fitness_irr` (const std::vector< T > &solution, const T &price, const T &nominal_value, const std::vector< T > &cash_flows, const std::vector< T > &time_periods, const [DF_type](#) &df_type, const bool &use_penalty_method)
This is the fitness function for finding the internal rate of return of a bond, in this case it is equal to its yield to maturity.

8.6.1 Detailed Description

Functions for the Internal Rate of Return.

Author

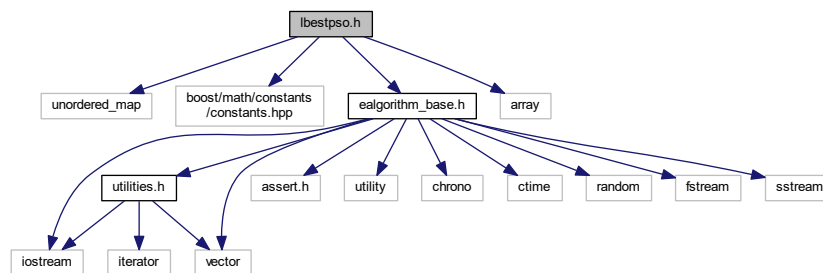
Ioannis Anagnostopoulos

8.7 lbestpso.h File Reference

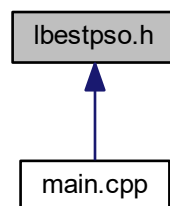
Classes and functions for the lbest (Local Best) Particle Swarm Optimisation with a ring topology.

```
#include <unordered_map>
#include <boost/math/constants/constants.hpp>
#include "ealgorithm_base.h"
#include <array>
```

Include dependency graph for lbestpso.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct [ea::PSO< T >](#)
Local Best Particle Swarm Optimisation Structure, used in the actual algorithm and for type deduction.
- class [ea::Solver< PSO, T, F, C >](#)
Local Best Particle Swarm Optimisation (PSO) Class.

Namespaces

- [ea](#)
Evolutionary Algorithms.

Variables

- `template<typename T>`
`const double ea::inv_pi_sq = 1 / std::pow(boost::math::constants::pi<T>(), 2)`
Inverse square of pi constant.

8.7.1 Detailed Description

Classes and functions for the lbest (Local Best) Particle Swarm Optimisation with a ring topology.

Author

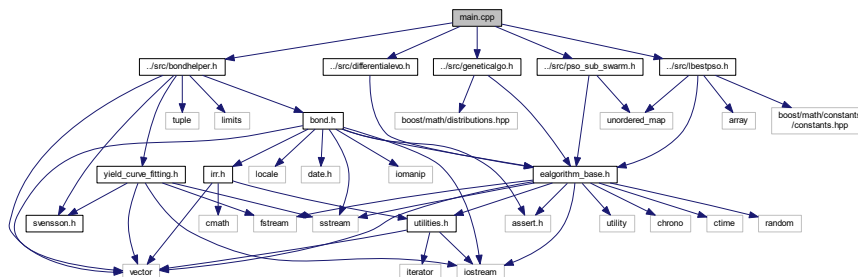
Ioannis Anagnostopoulos

8.8 main.cpp File Reference

A showcase of the application of the solvers on the Yield Curve Fitting, Internal Rate of Return Estimation and Bond Pricing Problems.

```
#include "../src/bondhelper.h"
#include "../src/geneticalgo.h"
#include "../src/psw_sub_swarm.h"
#include "../src/differentialevo.h"
#include "../src/lbestpso.h"
```

Include dependency graph for main.cpp:



Functions

- `int main ()`

8.8.1 Detailed Description

A showcase of the application of the solvers on the Yield Curve Fitting, Internal Rate of Return Estimation and Bond Pricing Problems.

Author

Ioannis Anagnostopoulos

Usage is the following:

Step 1: Create a solver structure object {GA, DE, PSO} with a specific floating-point number type, setting all of its parameters through its constructor.

Set `print_to_output` or `print_to_display` to false if there is no need for displaying the results to terminal or printing them to a file.

Step 2: Either use the common interface `solve` `solve(const F& f, const C& c, const S<T>& solver_struct, const std::string& problem_name)` passing the objective and constraint functions as lambda functions (anonymous functions) capturing all the required variables, such as bonds or use the public interfaces from the `Interest_Rate_Helper` (Yield Curve Fitting), `BondHelper` (Internal Rate of Return estimation and bond pricing for a number of bonds) and `Bond` (Internal Rate of Return Estimation and Macaulay Duration Estimation) classes after creating an object instance of those classes using their constructors.

8.8.2 Function Documentation

8.8.2.1 main()

```
int main ( )
```

Call benchmark functions

IRR solvers

Definition at line 75 of file main.cpp.

```
76 {
77     using namespace yft;
78     using namespace bond;
79     const std::vector<double> stdev { 0.7, 0.7, 0.7, 0.7, 0.7, 0.7 };
80     const std::vector<double> stdev_ga{ 0.5, 0.5, 0.5, 0.5, 0.5, 0.5 };
81     double irr_tol = 0.00000001;
82     double tol = 0.0001;
83     double tol_f = 0.001;
84     Interest_Rate_Helper<double> ir{ read_ir_from_file<double>("
interest_rate_data_periods.txt") };
85     BondHelper<double> de{ read_bonds_from_file<double>("bond_data.txt"), DF_type::exp };
86     DE<double> de_irr{ 1, 0.6, { 0.05 }, { 0.7 }, 10, irr_tol, 500, false, Constraints_type::normal
, true, true };
87     DE<double> de_irr_check{ 1, 0.6, { 0.05 }, { 0.7 }, 10, irr_tol, 500, false,
Constraints_type::normal, false, false };
88     GA<double> ga_irr{ 0.4, 0.35, 6.0, { 0.05 }, { 0.5 }, 42, irr_tol, 2000, false, Constraints_type::normal
, Strategy::remove, true, true };
89     PSO<double> pso_irr{ 1.49618, 0.9, { 1000000 }, { 0.05 }, { 0.7 }, 22, irr_tol, 3000, false,
Constraints_type::normal, true, true };
90     auto decision_variables = de.set_init_nss_params(de_irr);
91     DE<double> de_pricing{ 1, 0.6, decision_variables, stdev, 60, tol, 500, false,
Constraints_type::tight, true, true };
92     DE<double> de_fitting{ 1, 0.6, decision_variables, stdev, 60, tol_f, 500, false,
```

```

Constraints_type::tight, true, true };
95   GA<double> ga_pricing{ 0.4, 0.35, 6.0, decision_variables, stdev_ga, 250, tol, 2000, false,
Constraints_type::tight, Strategy::remove, true, true };
96   GA<double> ga_fitting{ 0.4, 0.35, 6.0, decision_variables, stdev_ga, 250, tol_f, 2000, false,
Constraints_type::tight, Strategy::remove, true, true };
97   PSO1<double> pso_pricing{ 1.49618, 0.9, { 100000, 100000, 100000, 100000, 100000, 100000 },
decision_variables, stdev, 130, tol, 3000, false, Constraints_type::tight, true, true };
98   PSO1<double> pso_fitting{ 1.49618, 0.9, { 100000, 100000, 100000, 100000, 100000, 100000 },
decision_variables, stdev, 130, tol_f, 3000, false, Constraints_type::tight, true, true };
99   PSOs<double> pso_pricing{ 2.05, 2.05, 6, 0.9, 1.0, { 100000, 100000, 100000, 100000, 100000, 100000 },
decision_variables, stdev, 24, tol, 1000, false, Constraints_type::none, true, true };
100   for (size_t i = 0; i < 100; ++i)
101   {
102       de.bond_pricing(ga_pricing, de_irr_check, Bond_pricing_type::bpp);
103       ir.yieldcurve_fitting(ga_fitting);
104       de.bond_pricing(de_pricing, de_irr_check, Bond_pricing_type::bpp);
105       ir.yieldcurve_fitting(de_fitting);
106       de.bond_pricing(pso_pricing, de_irr_check, Bond_pricing_type::bpp);
107       ir.yieldcurve_fitting(pso_fitting);
108       de.set_init_nss_params(de_irr);
109       de.set_init_nss_params(pso_irr);
110       de.set_init_nss_params(ga_irr);
111   }
112   return 0;
113 }

```

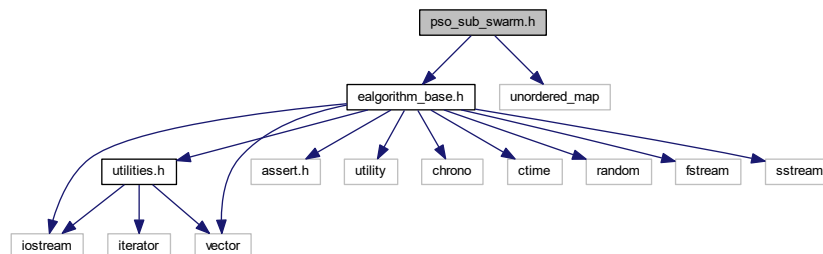
8.9 pso_sub_swarm.h File Reference

Classes and functions for the initial implementation of Sub-Swarm Particle Swarm Optimisation.

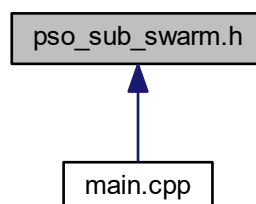
```
#include "ealgorithm_base.h"
```

```
#include <unordered_map>
```

Include dependency graph for pso_sub_swarm.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct `ea::PSOs< T >`
Particle Swarm Optimisation Structure, used in the actual algorithm and for type deduction.
- class `ea::Solver< PSOs, T, F, C >`
Sub-Swarm Particle Swarm Optimisation (PSO) Class.

Namespaces

- `ea`
Evolutionary Algorithms.

Variables

- `template<typename T >`
`const double ea::inv_pi_sq_2 = 1 / std::pow(boost::math::constants::pi<T>(), 2)`
Inverse square of pi constant.

8.9.1 Detailed Description

Classes and functions for the initial implementation of Sub-Swarm Particle Swarm Optimisation.

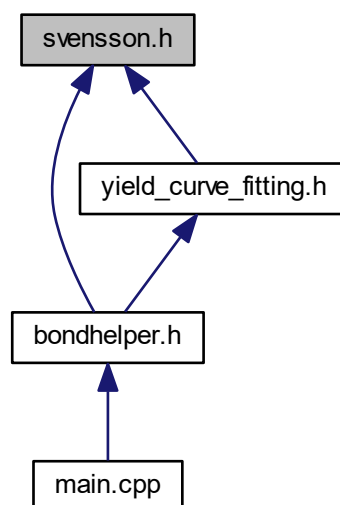
Author

Ioannis Anagnostopoulos

8.10 svensson.h File Reference

Functions for the Nelson-Siegel-Svensson model.

This graph shows which files directly or indirectly include this file:



Namespaces

- [nss](#)

Nelson-Siegel-Svensson (NSS) model namespace.

Functions

- `template<typename T >`
`bool nss::constraints_svensson (const std::vector< T > &solution, const Constraints_type &constraints_type)`
Constraints function for the NSS model.
- `template<typename T >`
`T nss::svensson (const std::vector< T > &solution, const T &m)`
Spot interest rate at term m using the NSS model.
- `template<typename T >`
`T nss::penalty_svensson (const std::vector< T > &solution)`
Penalty function for NSS.

8.10.1 Detailed Description

Functions for the Nelson-Siegel-Svensson model.

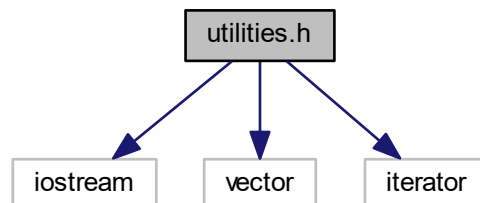
Author

Ioannis Anagnostopoulos

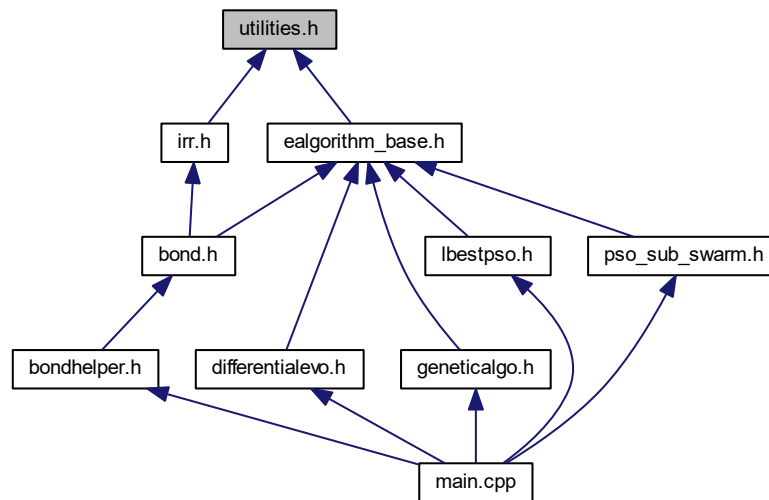
8.11 utilities.h File Reference

Enumerations and functions used from the rest of the project.

```
#include <iostream>
#include <vector>
#include <iterator>
Include dependency graph for utilities.h:
```



This graph shows which files directly or indirectly include this file:



Namespaces

- [utilities](#)

Utilities namespace.

Enumerations

- enum [utilities::DF_type](#) { [utilities::DF_type::frac](#), [utilities::DF_type::exp](#) }
Enumeration for discount factor types/methods.
- enum [utilities::Constraints_type](#) { [utilities::Constraints_type::normal](#), [utilities::Constraints_type::tight](#), [utilities::Constraints_type::none](#) }
Enumeration for types of constraints for the optimisation problems.

Functions

- template<typename T >
std::ostream & [utilities::operator<<](#) (std::ostream &stream, const std::vector< T > &vector)
Overload the operator << for printing vectors.

8.11.1 Detailed Description

Enumerations and functions used from the rest of the project.

Author

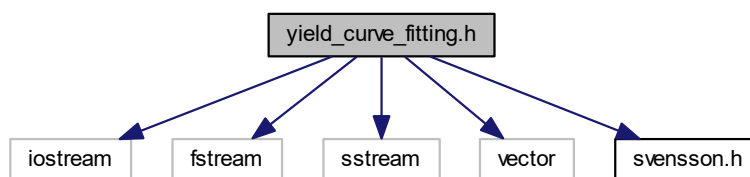
Ioannis Anagnostopoulos

8.12 yield_curve_fitting.h File Reference

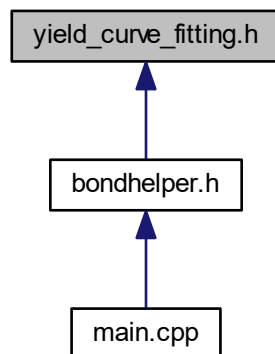
Class and functions for the yield curve fitting problem.

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include "svensson.h"
```

Include dependency graph for yield_curve_fitting.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct [yft::Interest_Rate< T >](#)
Structure for interest rates.
- class [yft::Interest_Rate_Helper< T >](#)
A class for the yield-curve-fitting problem.

Namespaces

- [yft](#)

Yield Curve Fitting namespace.

Functions

- `template<typename T >
std::vector< Interest_Rate< T > > yft::read_ir_from_file (const std::string &filename)`
Reads the interest rates and periods from file and constructs a vector of interest rate structs.

8.12.1 Detailed Description

Class and functions for the yield curve fitting problem.

Author

Ioannis Anagnostopoulos

Index

- alpha
 - ea::GA, [58](#)
 - ea::PSOs, [71](#)
- bdistribution
 - ea::Solver< GA, T, F, C >, [88](#)
- best_update
 - ea::Solver< PSOI, T, F, C >, [93](#)
 - ea::Solver< PSOs, T, F, C >, [104](#)
- Bond
 - bond::Bond, [32](#)
- bond, [11](#)
 - Bond_pricing_type, [11](#)
 - bpp, [12](#)
 - bpy, [12](#)
 - read_bonds_from_file, [12](#)
- bond.h, [119](#)
- bond::Bond
 - Bond, [32](#)
 - BondHelper< T >, [37](#)
 - cash_flows, [37](#)
 - compute_cash_flows, [33](#)
 - compute_macaulay_duration, [34](#)
 - compute_yield, [35](#), [36](#)
 - coupon_percentage, [37](#)
 - coupon_value, [38](#)
 - duration, [38](#)
 - frequency, [38](#)
 - maturity_date, [38](#)
 - nominal_value, [38](#)
 - price, [39](#)
 - settlement_date, [39](#)
 - time_periods, [39](#)
 - yield, [39](#)
- bond::Bond< T >, [31](#)
- bond::BondHelper
 - bond_pricing, [41](#)
 - BondHelper, [41](#)
 - bonds, [47](#)
 - df_type, [47](#)
 - estimate_bond_pricing, [42](#)
 - fitness_bond_pricing_prices, [43](#)
 - fitness_bond_pricing_yields, [44](#)
 - print_bond_pricing_results, [45](#)
 - set_init_nss_params, [46](#)
- bond::BondHelper< T >, [40](#)
- bond_pricing
 - bond::BondHelper, [41](#)
- Bond_pricing_type
 - bond, [11](#)
- BondHelper
 - bond::BondHelper, [41](#)
- BondHelper< T >
 - bond::Bond, [37](#)
- bondhelper.h, [120](#)
- bonds
 - bond::BondHelper, [47](#)
- bpp
 - bond, [12](#)
- bpy
 - bond, [12](#)
- c
 - ea::PSOI, [67](#)
 - ea::Solver_base, [117](#)
- c1
 - ea::PSOs, [71](#)
- c2
 - ea::PSOs, [72](#)
- cash_flows
 - bond::Bond, [37](#)
- check_pso_criteria
 - ea::Solver< PSOI, T, F, C >, [93](#)
 - ea::Solver< PSOs, T, F, C >, [104](#)
- compute_cash_flows
 - bond::Bond, [33](#)
- compute_discount_factor
 - irr, [18](#)
- compute_macaulay_duration
 - bond::Bond, [34](#)
- compute_pv
 - irr, [19](#)
- compute_yield
 - bond::Bond, [35](#), [36](#)
- constraints_irr
 - irr, [20](#)
- constraints_svensson
 - nss, [23](#)
- Constraints_type
 - utilities, [28](#)
- constraints_type
 - ea::EA_base, [54](#)
- construct_donor
 - ea::Solver< DE, T, F, C >, [76](#)
- construct_trial
 - ea::Solver< DE, T, F, C >, [77](#)
- coupon_percentage
 - bond::Bond, [37](#)
- coupon_value
 - bond::Bond, [38](#)

- cr
 - ea::DE, 50
- crossover
 - ea::Solver< GA, T, F, C >, 83
- DF_type
 - utilities, 28
- DE
 - ea::DE, 49
- de
 - ea::Solver< DE, T, F, C >, 80
- decision_variables
 - ea::EA_base, 54
- df_type
 - bond::BondHelper, 47
- differentiallevo.h, 122
- display_parameters
 - ea::Solver< DE, T, F, C >, 78
 - ea::Solver< GA, T, F, C >, 84
 - ea::Solver< PSOI, T, F, C >, 94
 - ea::Solver< PSOs, T, F, C >, 105
- display_results
 - ea::Solver_base, 113
- distribution
 - ea::Solver_base, 117
- duration
 - bond::Bond, 38
- EA_base
 - ea::EA_base, 53
- ea, 12
 - generator, 14
 - inv_pi_sq, 16
 - inv_pi_sq_2, 17
 - none, 14
 - rd, 17
 - remove, 14
 - solve, 15
 - Strategy, 14
- ea::DE< T >, 48
- ea::DE
 - cr, 50
 - DE, 49
 - f_param, 50
 - type, 50
- ea::EA_base
 - constraints_type, 54
 - decision_variables, 54
 - EA_base, 53
 - fp_type, 52
 - iter_max, 54
 - ndv, 54
 - npop, 54
 - print_to_file, 55
 - print_to_output, 55
 - stdev, 55
 - tol, 55
 - use_penalty_method, 55
- ea::EA_base< T >, 51
- ea::GA< T >, 56
- ea::GA
 - alpha, 58
 - GA, 57
 - pi, 58
 - strategy, 59
 - type, 59
 - x_rate, 59
- ea::PSOI
 - c, 67
 - PSOI, 66
 - type, 67
 - vmax, 68
 - w, 68
- ea::PSOI< T >, 65
- ea::PSOs
 - alpha, 71
 - c1, 71
 - c2, 72
 - PSOs, 70
 - sneigh, 72
 - type, 72
 - vmax, 72
 - w, 73
- ea::PSOs< T >, 69
- ea::Solver< DE, T, F, C >, 74
 - construct_donor, 76
 - construct_trial, 77
 - de, 80
 - display_parameters, 78
 - ind_distribution, 80
 - indices, 80
 - run_algo, 78
 - set_indices, 79
 - Solver, 75
 - Solver_base< Solver< DE, T, F, C >, DE, T, F, C >, 79
- ea::Solver< GA, T, F, C >, 81
 - bdistribution, 88
 - crossover, 83
 - display_parameters, 84
 - ga, 88
 - mutation, 84
 - nkeep, 85
 - npop, 88
 - run_algo, 85
 - selection, 87
 - Solver, 82
 - Solver_base< Solver< GA, T, F, C >, GA, T, F, C >, 88
 - stdev, 88
- ea::Solver< PSOI, T, F, C >, 89
 - best_update, 93
 - check_pso_criteria, 93
 - display_parameters, 94
 - euclid_distance, 94
 - find_min_local_best, 95
 - local_best, 98

- neighbours, 98
- nneigh, 98
- personal_best, 98
- personal_best_cost, 99
- position_update, 95
- pso, 99
- run_algo, 96
- set_neighbourhoods, 97
- Solver, 92
- Solver_base< Solver< PSOI, T, F, C >, PSOI, T, F, C >, 98
- velocity, 99
- vmax, 99
- w, 99
- ea::Solver< PSOs, T, F, C >, 100
 - best_update, 104
 - check_pso_criteria, 104
 - display_parameters, 105
 - euclid_distance, 105
 - find_min_local_best, 106
 - generate_r, 106
 - local_best, 109
 - neighbourhoods, 109
 - nneigh, 110
 - personal_best, 110
 - position_update, 107
 - pso, 110
 - run_algo, 108
 - set_neighbourhoods, 108
 - Solver, 103
 - Solver_base< Solver< PSOs, T, F, C >, PSOs, T, F, C >, 109
 - velocity, 110
 - vmax, 110
 - w, 111
- ea::Solver< S, T, F, C >, 73
- ea::Solver_base
 - c, 117
 - display_results, 113
 - distribution, 117
 - f, 117
 - find_min_cost, 114
 - individuals, 117
 - init_individuals, 114
 - last_iter, 118
 - min_cost, 118
 - randomise_individual, 115
 - solved_flag, 118
 - Solver_base, 112
 - solver_bench, 115
 - solver_struct, 118
 - timer, 118
 - write_results_to_file, 116
- ea::Solver_base< Derived, S, T, F, C >, 111
- ealgorithm_base.h, 123
- estimate_bond_pricing
 - bond::BondHelper, 42
- euclid_distance
 - ea::Solver< PSOI, T, F, C >, 94
 - ea::Solver< PSOs, T, F, C >, 105
- exp
 - utilities, 28
- f
 - ea::Solver_base, 117
- f_param
 - ea::DE, 50
- find_min_cost
 - ea::Solver_base, 114
- find_min_local_best
 - ea::Solver< PSOI, T, F, C >, 95
 - ea::Solver< PSOs, T, F, C >, 106
- fitness_bond_pricing_prices
 - bond::BondHelper, 43
- fitness_bond_pricing_yields
 - bond::BondHelper, 44
- fitness_irr
 - irr, 21
- fitness_yield_curve_fitting
 - yft::Interest_Rate_Helper, 62
- fp_type
 - ea::EA_base, 52
- frac
 - utilities, 28
- frequency
 - bond::Bond, 38
- GA
 - ea::GA, 57
- ga
 - ea::Solver< GA, T, F, C >, 88
- generate_r
 - ea::Solver< PSOs, T, F, C >, 106
- generator
 - ea, 14
- geneticalgo.h, 125
- ind_distribution
 - ea::Solver< DE, T, F, C >, 80
- indices
 - ea::Solver< DE, T, F, C >, 80
- individuals
 - ea::Solver_base, 117
- init_individuals
 - ea::Solver_base, 114
- Interest_Rate
 - yft::Interest_Rate, 60
- Interest_Rate_Helper
 - yft::Interest_Rate_Helper, 62
- inv_pi_sq
 - ea, 16
- inv_pi_sq_2
 - ea, 17
- ir_vec
 - yft::Interest_Rate_Helper, 64
- irr, 17
 - compute_discount_factor, 18

- compute_pv, [19](#)
 - constraints_irr, [20](#)
 - fitness_irr, [21](#)
 - penalty_irr, [22](#)
- irr.h, [126](#)
- iter_max
 - ea::EA_base, [54](#)
- last_iter
 - ea::Solver_base, [118](#)
- lbestpso.h, [128](#)
- local_best
 - ea::Solver< PSOI, T, F, C >, [98](#)
 - ea::Solver< PSOs, T, F, C >, [109](#)
- main
 - main.cpp, [130](#)
- main.cpp, [129](#)
 - main, [130](#)
- maturity_date
 - bond::Bond, [38](#)
- min_cost
 - ea::Solver_base, [118](#)
- mutation
 - ea::Solver< GA, T, F, C >, [84](#)
- ndv
 - ea::EA_base, [54](#)
- neighbourhoods
 - ea::Solver< PSOs, T, F, C >, [109](#)
- neighbours
 - ea::Solver< PSOI, T, F, C >, [98](#)
- nkeep
 - ea::Solver< GA, T, F, C >, [85](#)
- nneigh
 - ea::Solver< PSOI, T, F, C >, [98](#)
 - ea::Solver< PSOs, T, F, C >, [110](#)
- nominal_value
 - bond::Bond, [38](#)
- none
 - ea, [14](#)
 - utilities, [28](#)
- normal
 - utilities, [28](#)
- npop
 - ea::EA_base, [54](#)
 - ea::Solver< GA, T, F, C >, [88](#)
- nss, [23](#)
 - constraints_svensson, [23](#)
 - penalty_svensson, [25](#)
 - svensson, [26](#)
- operator<<
 - utilities, [29](#)
- PSOI
 - ea::PSOI, [66](#)
- PSOs
 - ea::PSOs, [70](#)
- penalty_irr
 - irr, [22](#)
- penalty_svensson
 - nss, [25](#)
- period
 - yft::Interest_Rate, [61](#)
- personal_best
 - ea::Solver< PSOI, T, F, C >, [98](#)
 - ea::Solver< PSOs, T, F, C >, [110](#)
- personal_best_cost
 - ea::Solver< PSOI, T, F, C >, [99](#)
- pi
 - ea::GA, [58](#)
- position_update
 - ea::Solver< PSOI, T, F, C >, [95](#)
 - ea::Solver< PSOs, T, F, C >, [107](#)
- price
 - bond::Bond, [39](#)
- print_bond_pricing_results
 - bond::BondHelper, [45](#)
- print_to_file
 - ea::EA_base, [55](#)
- print_to_output
 - ea::EA_base, [55](#)
- pso
 - ea::Solver< PSOI, T, F, C >, [99](#)
 - ea::Solver< PSOs, T, F, C >, [110](#)
- pso_sub_swarm.h, [131](#)
- randomise_individual
 - ea::Solver_base, [115](#)
- rate
 - yft::Interest_Rate, [61](#)
- rd
 - ea, [17](#)
- read_bonds_from_file
 - bond, [12](#)
- read_ir_from_file
 - yft, [30](#)
- remove
 - ea, [14](#)
- run_algo
 - ea::Solver< DE, T, F, C >, [78](#)
 - ea::Solver< GA, T, F, C >, [85](#)
 - ea::Solver< PSOI, T, F, C >, [96](#)
 - ea::Solver< PSOs, T, F, C >, [108](#)
- selection
 - ea::Solver< GA, T, F, C >, [87](#)
- set_indices
 - ea::Solver< DE, T, F, C >, [79](#)
- set_init_nss_params
 - bond::BondHelper, [46](#)
- set_neighbourhoods
 - ea::Solver< PSOI, T, F, C >, [97](#)
 - ea::Solver< PSOs, T, F, C >, [108](#)
- settlement_date
 - bond::Bond, [39](#)
- sneigh

- ea::PSOs, [72](#)
- solve
 - ea, [15](#)
- solved_flag
 - ea::Solver_base, [118](#)
- Solver
 - ea::Solver< DE, T, F, C >, [75](#)
 - ea::Solver< GA, T, F, C >, [82](#)
 - ea::Solver< PSOI, T, F, C >, [92](#)
 - ea::Solver< PSOs, T, F, C >, [103](#)
- Solver_base
 - ea::Solver_base, [112](#)
- Solver_base< Solver< DE, T, F, C >, DE, T, F, C >
 - ea::Solver< DE, T, F, C >, [79](#)
- Solver_base< Solver< GA, T, F, C >, GA, T, F, C >
 - ea::Solver< GA, T, F, C >, [88](#)
- Solver_base< Solver< PSOI, T, F, C >, PSOI, T, F, C >
 - ea::Solver< PSOI, T, F, C >, [98](#)
- Solver_base< Solver< PSOs, T, F, C >, PSOs, T, F, C >
 - ea::Solver< PSOs, T, F, C >, [109](#)
- solver_bench
 - ea::Solver_base, [115](#)
- solver_struct
 - ea::Solver_base, [118](#)
- stdev
 - ea::EA_base, [55](#)
 - ea::Solver< GA, T, F, C >, [88](#)
- Strategy
 - ea, [14](#)
- strategy
 - ea::GA, [59](#)
- svensson
 - nss, [26](#)
- svensson.h, [132](#)
- tight
 - utilities, [28](#)
- time_periods
 - bond::Bond, [39](#)
- timer
 - ea::Solver_base, [118](#)
- tol
 - ea::EA_base, [55](#)
- type
 - ea::DE, [50](#)
 - ea::GA, [59](#)
 - ea::PSOI, [67](#)
 - ea::PSOs, [72](#)
- use_penalty_method
 - ea::EA_base, [55](#)
- utilities, [27](#)
 - Constraints_type, [28](#)
 - DF_type, [28](#)
 - exp, [28](#)
 - frac, [28](#)
 - none, [28](#)
 - normal, [28](#)
 - operator<<, [29](#)
 - tight, [28](#)
- utilities.h, [133](#)
- velocity
 - ea::Solver< PSOI, T, F, C >, [99](#)
 - ea::Solver< PSOs, T, F, C >, [110](#)
- vmax
 - ea::PSOI, [68](#)
 - ea::PSOs, [72](#)
 - ea::Solver< PSOI, T, F, C >, [99](#)
 - ea::Solver< PSOs, T, F, C >, [110](#)
- w
 - ea::PSOI, [68](#)
 - ea::PSOs, [73](#)
 - ea::Solver< PSOI, T, F, C >, [99](#)
 - ea::Solver< PSOs, T, F, C >, [111](#)
- write_results_to_file
 - ea::Solver_base, [116](#)
- x_rate
 - ea::GA, [59](#)
- yft, [29](#)
 - read_ir_from_file, [30](#)
- yft::Interest_Rate
 - Interest_Rate, [60](#)
 - period, [61](#)
 - rate, [61](#)
- yft::Interest_Rate< T >, [60](#)
- yft::Interest_Rate_Helper
 - fitness_yield_curve_fitting, [62](#)
 - Interest_Rate_Helper, [62](#)
 - ir_vec, [64](#)
 - yieldcurve_fitting, [63](#)
- yft::Interest_Rate_Helper< T >, [61](#)
- yield
 - bond::Bond, [39](#)
- yield_curve_fitting.h, [135](#)
- yieldcurve_fitting
 - yft::Interest_Rate_Helper, [63](#)