Evolutionary Algorithms for Financial Optimisation

Generated by Doxygen 1.8.13

Contents

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:

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

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

Chapter 4

Class Index

4.1 Class List

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

Chapter 5

File Index

5.1 File List

Here is a list of all files with brief descriptions:

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.

```
{
    bpp,
    bpy
};
```

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

filename	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.

```
{
    std::vector<Bond<T>> bonds;
    std::ifstream input(filename);
    for (std::string line; getline(input, line); )
    {
        T coupon_percentage;
        T price;
        T nominal_value;
        T frequency;
        std::string settlement_date;
        std::string maturity_date;
        std::istringstream stream(line);
        stream >> coupon_percentage >> price >> nominal_value >> frequency >> settlement_date >> maturity_date;
        const Bond<T> bond{ coupon_percentage, price, nominal_value, frequency, settlement_date, maturity_date };
        bonds.push_back(bond);
    }
    return bonds;
}
```

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 PSOl

  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< PSOl, 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()  )
```

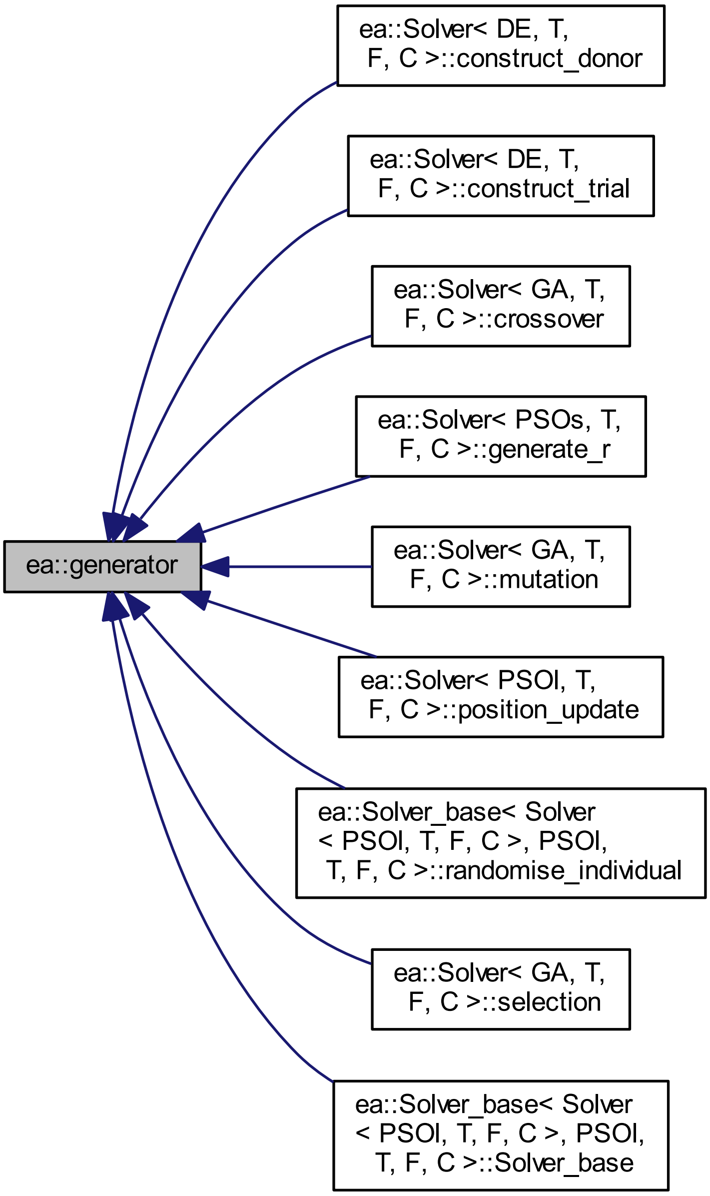
Pseudo-random number generator.

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< PSOl, T, F, C >::position_update(), ea::Solver_base< Solver< PSOl, T, F, C >, PSOl, T, F, C >::randomise_individual(), ea::Solver< GA, T, F, C >::selection(), and ea::Solver_base< Solver< PSOl, T, F, C >, PSOl, 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

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

Returns

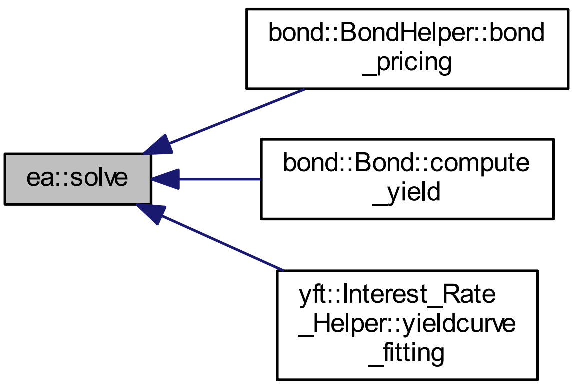
The solution vector

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().

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

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

r	Rate
period	The period the rate was recorded
df_type	The method used to calculate the discount factor

Returns

The discount factor

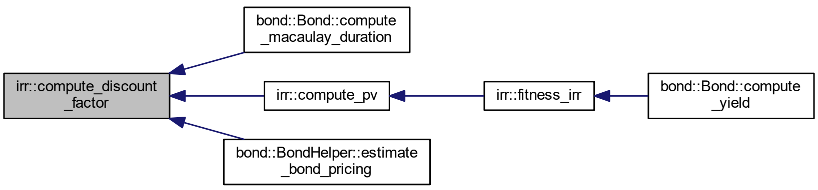
Definition at line 25 of file irr.h.

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

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

```
    {
        switch (df_type)
        {
        case (DF_type::frac): return 1 / std::pow((1 + r), period);
        case (DF_type::exp): return std::exp(-r * period);
            default: std::abort();
        }
    }
```

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

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

Returns

The present value of the investment

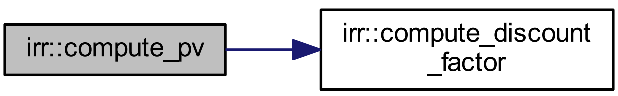
Definition at line 74 of file irr.h.

References compute_discount_factor().

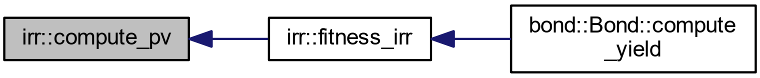
Referenced by fitness_irr().

```
{
    assert(cash_flows.size() == time_periods.size());
    const size_t& num_time_periods =time_periods.size();
    T sum = 0.0;
    for (size_t i = 0; i < num_time_periods; ++i)
    {
        sum = sum + cash_flows[i] * compute_discount_factor(r, time_periods[i], df_type);
    }
    return sum + nominal_value * compute_discount_factor(r, time_periods.back(), df_type);
}
```

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

solution	Internal Rate of Return candindate solution
constraints_type	Type of constraints used

Returns

True if constraints are satisfied, false otherwise

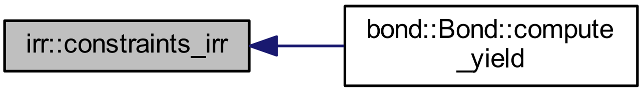
Definition at line 42 of file irr.h.

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

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

```
{
    switch (constraints_type)
    {
    case(Constraints_type::normal):
    {
        const T& r = solution[0];
        if (r > 0 && r < 1)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    case(Constraints_type::tight): return true;
    case(Constraints_type::none): return true;
        default: std::abort();
    }
}
```

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

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

Returns

The fitness cost of IRR

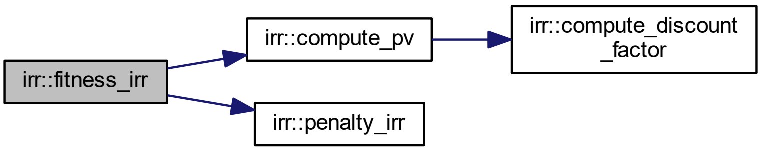
Definition at line 116 of file irr.h.

References compute_pv(), and penalty_irr().

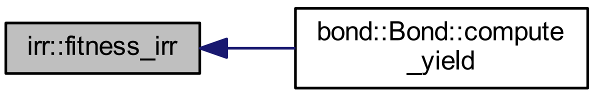
Referenced by bond::Bond< T >::compute_yield().

```
    {
        T sum_of_squares = 0;
        sum_of_squares = sum_of_squares + std::pow(price - compute_pv(solution[0], nominal_value,
    cash_flows, time_periods, df_type), 2);
        if (use_penalty_method)
        {
            return sum_of_squares + penalty_irr(solution[0]);
        }
        else
        {
            return sum_of_squares;
        }
    }
```

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

r	Candidate solution for the Internal Rate of Return

Returns

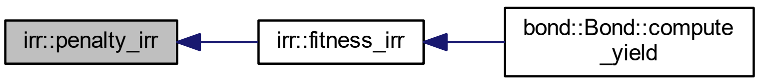
A penalty value, if constraints are not satisfied

Definition at line 92 of file irr.h.

Referenced by fitness_irr().

```
{
    T sum = 0;
    const T C = 1000;
    if (r < 0 || r > 1)
    {
        sum = sum + C * std::pow(std::abs(r), 2);
    }
     return sum;
 }
```

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

solution	NSS parameters candindate solution
constraints_type	Type of constraints used

Returns

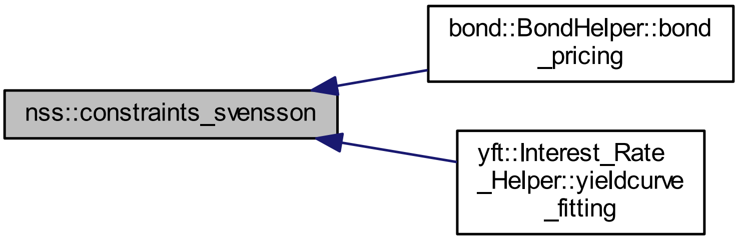
True if constraints are satisfied, false otherwise

Definition at line 18 of file svensson.h.

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

```
{
    switch (constraints_type)
    {
    case(Constraints_type::normal):
    {
        const T& b0 = solution[0];
        const T& b1 = solution[1];
        const T& tau1 = solution[4];
        const T& tau2 = solution[5];
        if (b0 > 0 && b0 + b1 > 0 && tau1 > 0 && tau2 > 0)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    case(Constraints_type::tight):
    {
        const T& b0 = solution[0];
        const T& b1 = solution[1];
        const T& b2 = solution[2];
        const T& b3 = solution[3];
        const T& tau1 = solution[4];
        const T& tau2 = solution[5];
        if ((b0 > 0 && b0 < 15)
            && (b1 > -15 && b1 < 30)
            && (b2 > -30 && b2 < 30)
            && (b3 > -30 && b3 < 30)
            && (tau1 > 0 && tau1 < 2.5)
            && (tau2 > 2.5 && tau2 < 5.5))
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    case(Constraints_type::none): return true;
    default: std::abort();
    }
}
```

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

solution	Candidate solution for the parameters of NSS

Returns

A penalty value, if constraints are not satisfied

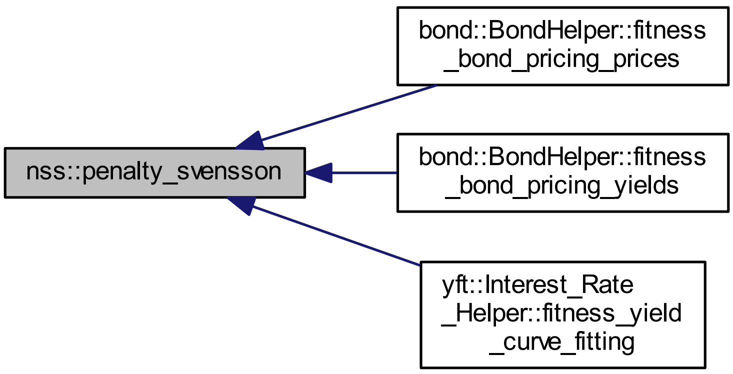
Definition at line 99 of file svensson.h.

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().

```
{
    T sum = 0;
    const T& b0 = solution[0];
    const T& b1 = solution[1];
    const T& b2 = solution[2];
    const T& b3 = solution[3];
    const T& tau1 = solution[4];
    const T& tau2 = solution[5];
    const T C = 100000;
    if (b0 < 0 || b0 > 15)
    {
        sum = sum + C * std::pow(std::abs(b0), 2);
    }
    if (b0 + b1 < 0)
    {
        sum = sum + C * std::pow(std::abs(b0 + b1), 2);
    }
    if (b1 < -15 || b1 > 30)
```

```
        {
            sum = sum + C * std::pow(std::abs(b1), 2);
        }
        if (b2 < -30 || b2 > 30)
        {
            sum = sum + C * std::pow(std::abs(b1), 2);
        }
        if (b3 < -30 || b3 > 30)
        {
            sum = sum + C * std::pow(std::abs(b1), 2);
        }
        if (tau1 < 0 || tau1 > 2.5)
        {
            sum = sum + C * std::pow(std::abs(tau2), 2);
        }
        if (tau2 < 2.5 || tau2 > 5.5)
        {
            sum = sum + C * std::pow(std::abs(tau2), 2);
        }
        return sum;
    }
```

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

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

Returns

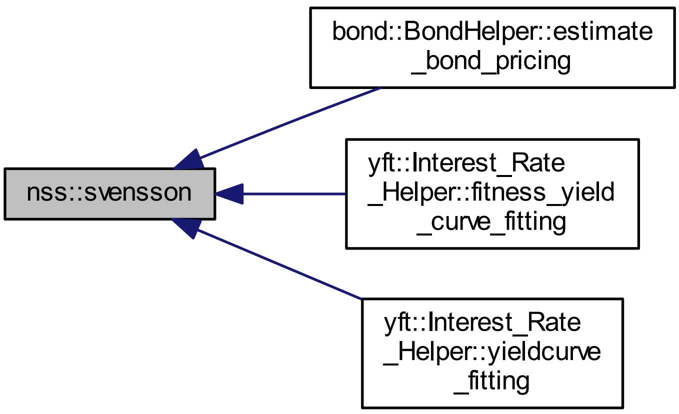
The spot interest rate at term m

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().

```
    {
        const T& b0 = solution[0];
        const T& b1 = solution[1];
        const T& b2 = solution[2];
        const T& b3 = solution[3];
        const T& tau1 = solution[4];
        const T& tau2 = solution[5];
        if (m == 0)
        {
            return b0 + b1;
        }
        else
        {
            T result = b0;
            result = result + b1 * ((1 - (std::exp(-m / tau1))) / (m / tau1));
            result = result + b2 * (((1 - (std::exp(-m / tau1))) / (m / tau1)) - std::exp(-m / tau1));
            result = result + b3 * (((1 - (std::exp(-m / tau2))) / (m / tau2)) - std::exp(-m / tau2));
            return result;
        }
    }
```

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.

```
{
    frac,
    exp
};
```

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

stream	An out stream
vector	A vector

Returns

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

Definition at line 38 of file utilities.h.

```
{
    if (!vector.empty())
    {
        stream << "[ ";
        std::copy(vector.begin(), vector.end(), std::ostream_iterator<T>(stream, " "));
        stream << "]";
    }
    return stream;
}
```

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

filename	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.

```
{
    std::vector<Interest_Rate<T>> ir_vec;
    std::ifstream input(filename);
    for (std::string line; getline(input, line); )
    {
        T period;
        T rate;
        std::istringstream stream(line);
        stream >> period >> rate;
        const Interest_Rate<T> ir{ period, rate };
        ir_vec.push_back(ir);
    }
    return ir_vec;
}
```

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

Returns

A Bond<T> object

Definition at line 48 of file bond.h.

```
                                                                     :
            coupon_percentage{ i_coupon_percentage },
            price{ i_price },
            nominal_value{ i_nominal_value },
            frequency{ i_frequency },
            coupon_value{ coupon_percentage * nominal_value / frequency },
            yield{ 0 },
            duration{ 0 }
        {
            assert(price > 0);
            assert(coupon_percentage > 0 && coupon_percentage < 1);
            assert(nominal_value > 0);
            assert(frequency > 0);
            std::tm t1 {};
            std::tm t2 {};
            std::stringstream s1;
            std::stringstream s2;
            s1 << i_settlement_date;
            s2 << i_maturity_date;
            s1 >> std::get_time(&t1, "%Y-%m-%d");
            s2 >> std::get_time(&t2, "%Y-%m-%d");
            settlement_date = date::year(t1.tm_year) / (t1.tm_mon+1) / t1.tm_mday;
            maturity_date = date::year(t2.tm_year) / (t2.tm_mon+1) / t2.tm_mday;
            cash_flows = compute_cash_flows();
            time_periods.resize(cash_flows.size());
            for (size_t i = 0; i < time_periods.size(); ++i)
            {
                time_periods[i] = static_cast<T>(i + 1) / frequency;
            }
        }
```

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.

```
    {
        assert(settlement_date < maturity_date);
        const T number_of_days_coupon = 365 / frequency;
        const auto days_difference = (maturity_date - settlement_date).count();
        const auto time_periods = static_cast<T>(days_difference) / number_of_days_coupon;
        const size_t num_time_periods = static_cast<size_t>(std::ceil(time_periods));
        std::vector<T> cash_flows(num_time_periods);
        for (auto& p : cash_flows)
        {
            p = coupon_value;
        }
        return cash_flows;
    }
```

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

df_type	The type of discount factor method

Returns

The Macaulay Duration of the bond

Discount factor

Prest cash flows

Present value

Macaulay duration

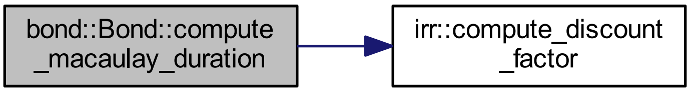
Definition at line 174 of file bond.h.

References irr::compute_discount_factor().

```
    {
        assert(yield > 0 && yield < 1);
        assert(cash_flows.size() > 0);
        assert(nominal_value > 0);
        assert(frequency > 0);
        T discount_factor = 0.0;
        T denominator = 0.0;
        T pv = 0.0;
```

```
        T numerator = 0.0;
        for (size_t i = 0; i < time_periods.size(); ++i)
        {
            discount_factor = compute_discount_factor(
    yield, time_periods[i], df_type);
            pv = coupon_value * discount_factor;
            numerator = numerator + pv * time_periods[i];
            denominator = denominator + pv;
        }
        pv = nominal_value * discount_factor;
        numerator = numerator + pv * time_periods.back();
        denominator = denominator + pv;
        T duration = numerator / denominator;
        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_price	The price of the bond
solver	The parameter structure of the solver that is going to be used to estimate the yield of maturity
df_type	The type of discount factor method

Returns

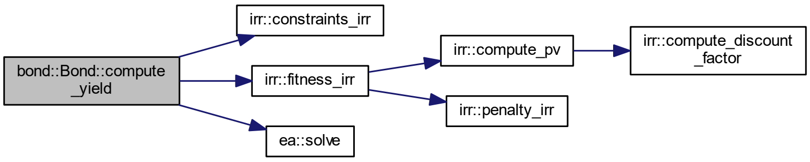
The yield-to-maturity of the bond

Definition at line 150 of file bond.h.

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

```
    {
        assert(solver.ndv == 1);
        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); };
        auto c = [&,constraints_type = solver.constraints_type](const auto& solution) { return constraints_irr(solution, constraints_type); };
        auto res = solve(f, c, solver, "YTM");
        T yield = res[0];
        return yield;
    }
```

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

Returns

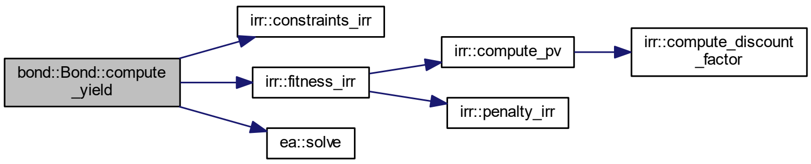
The yield-to-maturity of the bond

Definition at line 162 of file bond.h.

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

```
    {
        assert(solver.ndv == 1);
        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); };
        auto c = [&, constraints_type = solver.constraints_type](const auto& solution) { return
    constraints_irr(solution, constraints_type); };
        std::string problem = "YTM";
        auto res = solve(f, c, solver, problem.append(bonds_identifier));
        T yield = res[0];
        return yield;
    }
```

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_bonds	A vector of Bond<T> objects
i_df_type	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

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

Returns

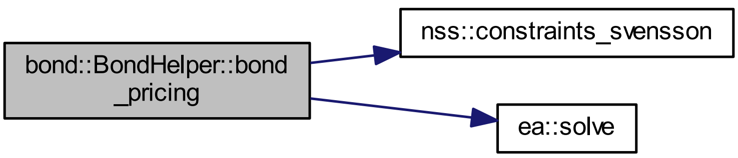
void

Definition at line 235 of file bondhelper.h.

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

```
{
    assert(solver.ndv == 6);
    for (const auto& p : bonds)
    {
        assert(p.yield > 0 && p.yield < 1);
        assert(p.duration > 0);
    }
    switch (bond_pricing_type)
    {
    case(Bond_pricing_type::bpp):
    {
        auto f = [&, use_penalty_method = solver.use_penalty_method](const auto& solution) { return fitness_bond_pricing_prices(solution, use_penalty_method); };
        auto c = [&, constraints_type = solver.constraints_type](const auto& solution) { return constraints_svensson(solution, constraints_type); };
        std::cout << "Solving bond pricing using bond prices..." << "\n";
        auto res = solve(f, c, solver, "BPP");
        print_bond_pricing_results(res, solver_irr);
        break;
    }
    case(Bond_pricing_type::bpy):
    {
        auto f = [&, use_penalty_method = solver.use_penalty_method](const auto& solution) { return fitness_bond_pricing_yields(solution, solver_irr, use_penalty_method); };
        auto c = [&, constraints_type = solver.constraints_type](const auto& solution) { return constraints_svensson(solution, constraints_type); };
        std::cout << "Solving bond pricing using bond yields..." << "\n";
        auto res = solve(f, c, solver, "BPY");
        print_bond_pricing_results(res, solver_irr);
    }
    }
}
```

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

solution	NSS parameters candindate solution
coupon_value	The value of the coupon payment
nominal_value	The nominal value of the investment
time_periods	The time periods that correspond to the coupon payments

Returns

The price of the bond

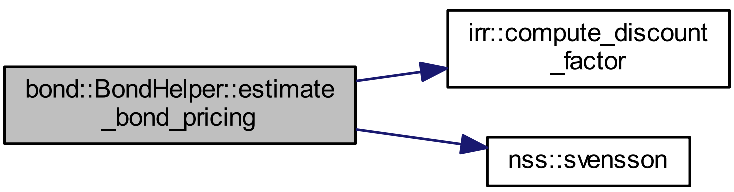
Call svensson for period

Definition at line 178 of file bondhelper.h.

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

```
{
    T sum = 0.0;
    for (const auto& t : time_periods)
    {
        sum = sum + coupon_value * compute_discount_factor(
svensson(solution, t), t, df_type);
    }
    T m = time_periods.back();
    sum = sum + nominal_value * compute_discount_factor(
svensson(solution, m), m, df_type);
    return sum;
}
```

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

solution	NSS parameters candindate solution
use_penalty_method	Whether to use the penalty method defined for NSS or not

Returns

The fitness cost of NSS for bond pricing

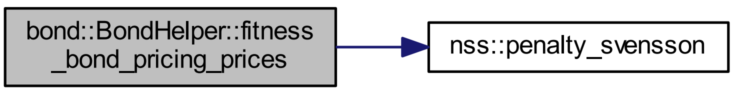
The sum of squares of errors between the actual bond price and the estimated price from estimate_bond_pricing

Definition at line 192 of file bondhelper.h.

References nss::penalty_svensson().

```
    {
        T sum_of_squares = 0.0;
        for (const auto& k : bonds)
        {
            T estimate = estimate_bond_pricing(solution, k.coupon_value, k.nominal_value, k.time_periods);
            sum_of_squares = sum_of_squares + std::pow((k.price/100 - estimate/100), 2) / std::sqrt(k.duration);
        }
        if (use_penalty_method)
        {
            return sum_of_squares + penalty_svensson(solution);
        }
        else
        {
            return sum_of_squares;
        }
    }
```

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

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

Returns

The fitness cost of NSS for bond pricing

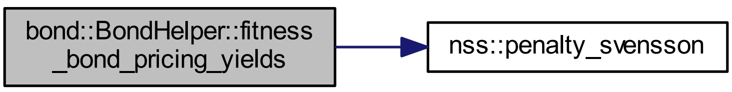
The sum of squares of errors between the actual bond yield to maturity and the estimated yield to maturity by svensson is used

Definition at line 213 of file bondhelper.h.

References nss::penalty_svensson().

```
    {
        T sum_of_squares = 0;
        for (const auto& k : bonds)
        {
            T estimate_price = estimate_bond_pricing(solution, k.coupon_value, k.nominal_value, k.time_periods);
            T estimate = k.compute_yield(estimate_price, solver_irr, df_type);
            sum_of_squares = sum_of_squares + std::pow(k.yield - estimate, 2);
        }
        if (use_penalty_method)
        {
            return sum_of_squares + penalty_svensson(solution);
        }
        else
        {
            return sum_of_squares;
        }
    }
```

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

res	The solution vector of NSS parameters
solver_irr	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.

```
    {
        T error = 0;
        //for (const auto& p : bonds)
        //{
            //T estimate_price = estimate_bond_pricing(res, p.coupon_value, p.nominal_value, p.time_periods);
            //T estimate = p.compute_yield(estimate_price, solver_irr, df_type);
            //error = error + std::pow(estimate - p.yield, 2);
            //std::cout << "Estimated yield: " << estimate << " Actual Yield: " << p.yield << "\n";
        //}
        //std::cout << "Yield Mean Squared Error: " << error << "\n";
        error = 0;
        for (const auto& p : bonds)
        {
            error = error + std::pow(estimate_bond_pricing(res, p.coupon_value, p.nominal_value, p.time_periods)/100 - p.price/100, 2);
            std::cout << "Estimated price: " << estimate_bond_pricing(res, p.coupon_value, p.nominal_value, p.time_periods) << " Actual Price: " << p.price << "\n";
        }
        std::cout << "Price Mean Squared Error: " << error << "\n";
    }
```

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

solver	The parameter structure of the solver that is going to be used

Returns

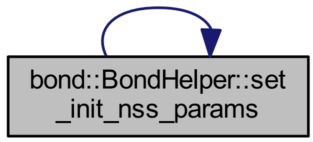
A vector of decision variables for NSS

Definition at line 69 of file bondhelper.h.

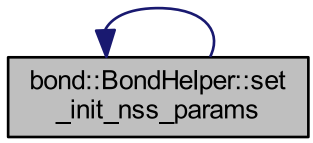
References bond::BondHelper< T >::set_init_nss_params().

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:

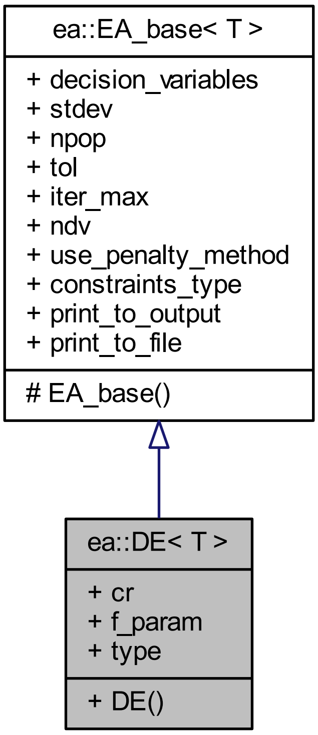
- bond.h
- bondhelper.h

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

```
                                                                            :
            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),
            cr( i_cr ),
            f_param( i_f_param )
        {
            assert(cr > 0 && cr <= 1);
            assert(f_param > 0 && f_param <= 1);
        }
```

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:

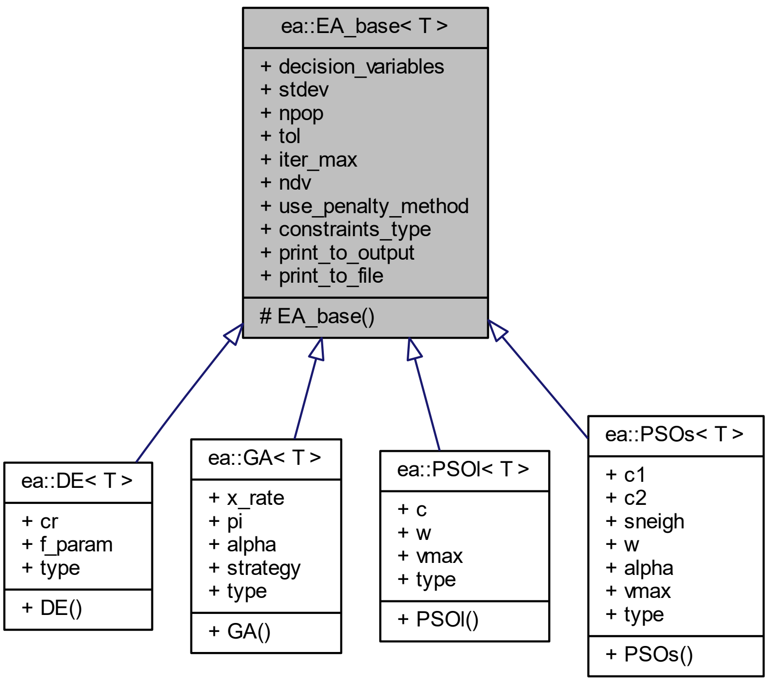
- differentialevo.h

7.4 ea::EA_base< T > Struct Template Reference

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

Returns

A EA_base<T> object

Definition at line 68 of file ealgorithm_base.h.

```
        : decision_variables{ i_decision_variables },
stdev{ i_stdev }, npop{ i_npop }, tol{ i_tol }, iter_max{ i_iter_max },
ndv{ i_decision_variables.size() },
        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 }
    {
        assert(decision_variables.size() > 0);
        assert(decision_variables.size() == stdev.size());
        for (const auto& p : stdev)
        {
            assert(p > 0);
        }
        assert(npop > 0);
        assert(tol > 0);
        assert(iter_max > 0);
    }
```

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::PSOl< T >::PSOl(), 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:

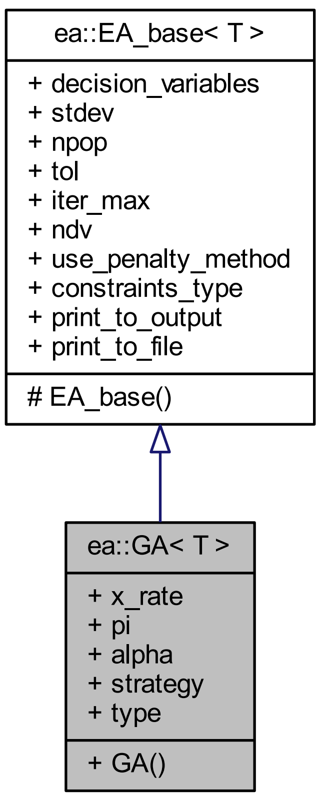
- ealgorithm_base.h

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

Returns

A GA<T> object

Definition at line 44 of file geneticalgo.h.

```
                                                                                        :
            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 ),
            x_rate ( i_x_rate ),
            pi ( i_pi ),
            alpha ( i_alpha ),
            strategy ( i_strategy )
        {
            assert(x_rate > 0 && x_rate <= 1);
            assert(pi > 0 && pi <= 1);
        }
```

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_period	The period the zero rate was recorded
i_rate	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

i_ir_vec	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

solution	NSS parameters candindate solution
use_penalty_method	Whether to use the penalty method defined for NSS or not

Returns

The fitness cost of NSS for yield curve fitting

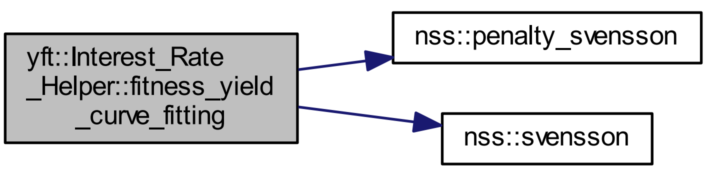
The sum of squares of errors betwwen the actual rates and the rates computed by svensson are used

Definition at line 105 of file yield_curve_fitting.h.

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

```
        {
            T sum_of_squares = 0;
            for (size_t i = 0; i < ir_vec.size(); ++i)
            {
                T estimate = svensson(solution, ir_vec[i].period);
                sum_of_squares = sum_of_squares + std::pow(ir_vec[i].rate - estimate, 2);
            }
            if (use_penalty_method)
            {
                return sum_of_squares + penalty_svensson(solution);
            }
            else
            {
                return sum_of_squares;
            }
        };
```

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

solver	The parameter structure of the solver that is going to be used for yield curve fitting

Returns

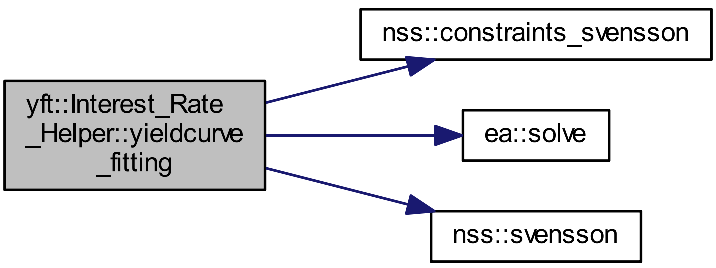
void

Definition at line 81 of file yield_curve_fitting.h.

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

```
    {
        assert(solver.ndv == 6);
        auto f = [&, use_penalty_method = solver.use_penalty_method](const auto& solution) { return fitness_yield_curve_fitting(solution, use_penalty_method); };
        auto c = [&, constraints_type = solver.constraints_type](const auto& solution) { return constraints_svensson(solution, constraints_type); };
        std::cout << "Yield Curve fitting." << "\n";
        auto res = solve(f, c, solver, "YFT");
        T error = 0;
        for (const auto& p : ir_vec)
        {
            error = error + std::pow(svensson(res, p.period) - p.rate, 2);
            //std::cout << "Estimated interest rates: " << svensson(res, p.period) << " Actual interest rates: " << p.rate << "\n";
        }
        std::cout << "Zero-rate Mean Squared Error: " << error / ir_vec.size() << "\n";
    };
```

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:

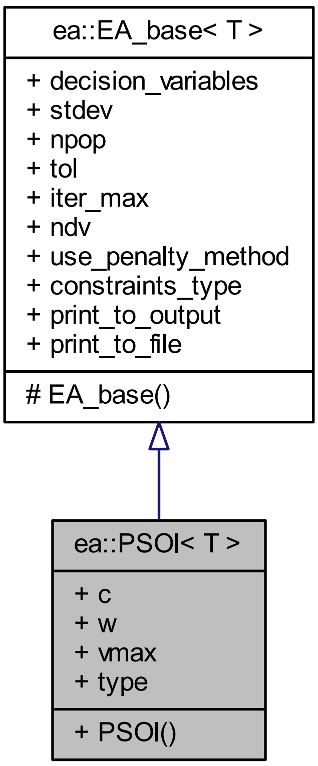
- yield_curve_fitting.h

7.8 ea::PSOl< 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::PSOl< T >:



Public Member Functions

- PSOl (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::PSOl< 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 PSOl()

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

Returns

A PSOl<T> object

Definition at line 41 of file lbestpso.h.

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

```
                                                                               :
            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),
            c(i_c),
            w(i_w),
            vmax(i_vmax)
        {
            assert(c > 0);
            assert(w > 0);
            for (const auto& p : vmax) { assert(p > 0); };
            assert(vmax.size() == this->ndv);
        }
```

7.8.3 Member Data Documentation

7.8.3.1 c

```
template<typename T>
const T ea::PSOl< T >::c
```

Parameter c for velocity update.

Definition at line 56 of file lbestpso.h.

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

7.8.3.2 type

```
template<typename T>
const std::string ea::PSOl< 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::PSOl< T >::vmax
```

Velocity Clamping Variant of PSO : Maximum Velocity.

Definition at line 60 of file lbestpso.h.

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

7.8.3.4 w

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

Inertia Variant of PSO : Inertia.

Definition at line 58 of file lbestpso.h.

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

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

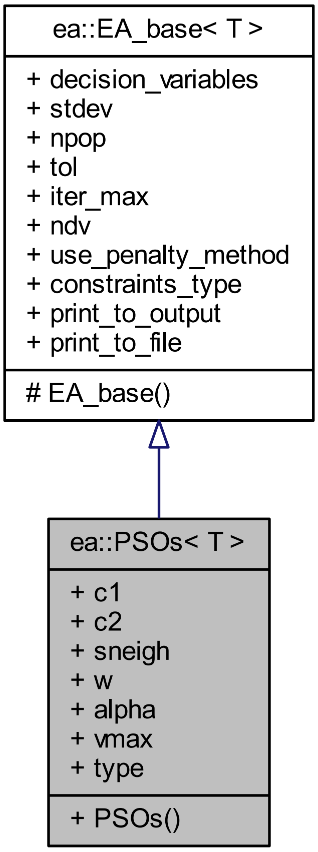
- lbestpso.h

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_c1	c1 parameter for velocity update
i_c2	c2 parameter for velocity update
i_sneigh	Number of neighbourhoods
i_w	Inertia parameter for velocity update
i_alpha	alpha parameter for maximum velocity decrease
i_vmax	Maximum velocity
i_decision_variables	The starting values of the decision variables
i_stdev	The standard deviation
i_npop	The population size
i_tol	The tolerance
i_iter_max	The maximum number of iterations
i_use_penalty_method	Whether to used penalties or not
i_constraints_type	What kind of constraints to use
i_print_to_output	Whether to print to terminal or not
i_print_to_file	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.

```
                                                                            :
            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),
            c1( i_c1 ),
            c2( i_c2 ),
            sneigh( i_sneigh ),
            w( i_w ),
            alpha( i_alpha ),
            vmax( i_vmax )
        {
            assert(c1 > 0);
            assert(c2 > 0);
            assert(sneigh > 0);
            assert(sneigh < i_npop);
            assert(w > 0);
            assert(alpha > 0);
            for (const auto& p : vmax) { assert(p > 0); };
            assert(vmax.size() == this->ndv);
        }
```

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:

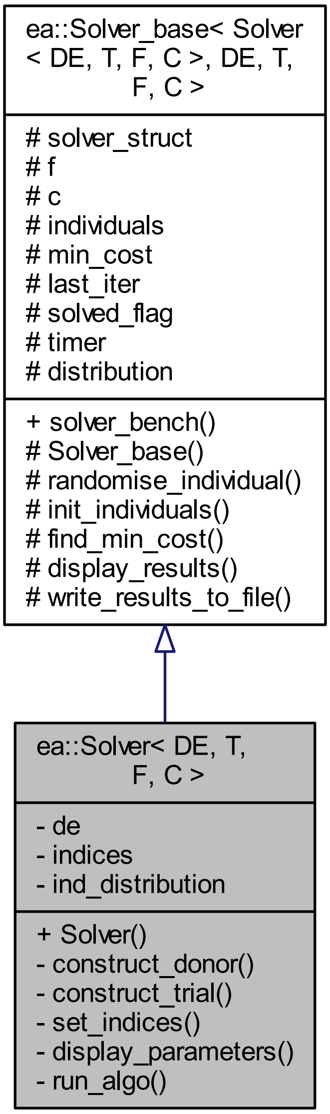
- ealgorithm_base.h

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_de	The differential evolution parameter structure that is used to construct the solver
f	A reference to the objective function
c	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.npop - 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

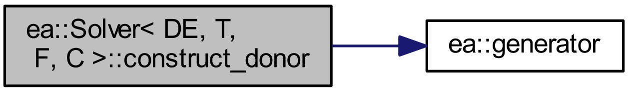
Check that the indices are not the same

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

target	Target vector (an individual)
donor	Donor vector produced from construct_donor()

Returns

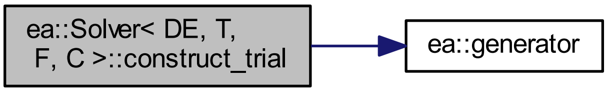
A trial vector that is compared with the current individual

Definition at line 150 of file differentialevo.h.

References ea::generator().

```
{
    std::vector<T> trial(de.ndv);
    std::vector<size_t> j_indices(de.ndv);
    for (size_t j = 0; j < de.ndv; ++j)
    {
        j_indices[j] = j;
    }
    std::uniform_int_distribution<size_t> j_ind_distribution(0, de.ndv - 1);
    for (size_t j = 0; j < de.ndv; ++j)
    {
        const T& epsilon = this->distribution(generator);
        const size_t& jrand = j_indices[j_ind_distribution(generator)];
        if (epsilon <= de.cr || j == jrand)
        {
            trial[j] = donor[j];
        }
        else
        {
            trial[j] = target[j];
        }
    }
    return trial;
}
```

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.

```
    {
        std::stringstream parameters;
        parameters << "Crossover Rate:" << "," << de.cr << ",";
        parameters << "Mutation Scale Factor:" << "," << de.f_param;
        return parameters;
    }
```

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.

```
    {
        std::vector< std::vector<T> > personal_best = this->individuals;
        for (size_t iter = 0; iter < de.iter_max; ++iter)
        {
            for (auto& p : this->individuals)
            {
                std::vector<T> donor = construct_donor();
                while (!this->c(donor))
                {
                    donor = construct_donor();
                }
                const std::vector<T>& trial = construct_trial(p, donor);
                if (this->f(trial) <= this->f(p))
                {
                    p = trial;
                }
            }
            this->find_min_cost();
            this->last_iter = iter;
            if (de.tol > std::abs(this->f(this->min_cost)))
            {
                this->solved_flag = true;
                break;
            }
        }
    }
```

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.

```
        {
            std::vector<size_t> indices;
            for (size_t i = 0; i < de.npop; ++i)
            {
                indices.push_back(i);
            }
            return indices;
        };
```

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:

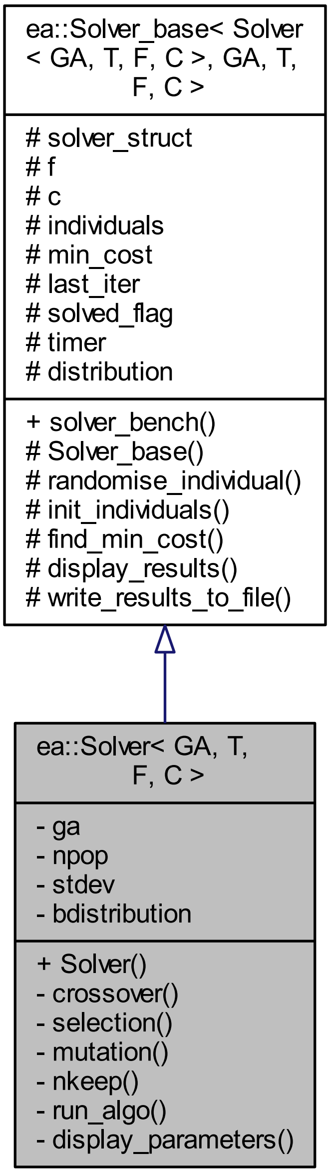
- differentialevo.h

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_ga	The genetic algorithms parameter structure that is used to construct the solver
f	A reference to the objective function
c	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

r,s	Parent individuals

Returns

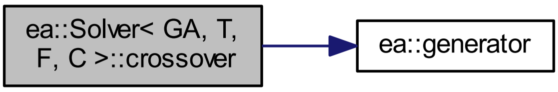
An offspring from the two parents r and s

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.

```
{
    std::stringstream parameters;
    parameters << "Natural Selection Rate:" << "," << ga.x_rate << ",";
    parameters << "Probability of Mutation:" << "," << ga.pi << ",";
    parameters << "Beta Distribution alpha:" << "," << ga.alpha << ",";
    parameters << "Strategy:" << ",";
    switch (ga.strategy)
    {
    case Strategy::keep_same: parameters << "Keep same individual"; break;
    case Strategy::re_mutate: parameters << "Re-mutate individual"; break;
    case Strategy::remove: parameters << "Remove individual"; break;
    default: parameters << "Do nothing"; break;
    }
    return parameters;
}
```

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

individual	An individual of the population

Returns

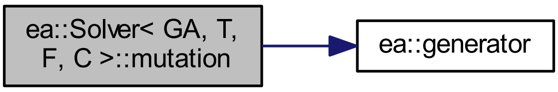
A mutated individual

Definition at line 178 of file geneticalgo.h.

References ea::generator().

```
{
    std::vector<T> mutated = individual;
    for (size_t j = 0; j < ga.ndv; ++j)
    {
        const T r = this->distribution(generator);
        if (ga.pi < r)
        {
            std::normal_distribution<T> ndistribution(0, stdev[j]);
            T epsilon = ndistribution(generator);
            mutated[j] = mutated[j] + epsilon;
        }
    }
    return mutated;
}
```

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.

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

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.

```
{
    auto comparator = [&](const std::vector<T>& l, const std::vector<T>& r)
    {
        return this->f(l) < this->f(r);
    };
    for (size_t iter = 0; iter < ga.iter_max; ++iter)
    {
        npop = this->individuals.size();
        std::sort(this->individuals.begin(), this->individuals.end(), comparator);
        this->individuals.erase(this->individuals.begin() + nkeep(), this->individuals.begin() + this->individuals.size());
        this->min_cost = this->individuals[0];
        this->last_iter = iter;
        if (ga.tol > std::abs(this->f(this->min_cost)))
        {
            this->solved_flag = true;
            break;
        }
        for (size_t i = 0; i < npop; ++i)
        {
            std::vector<T> offspring = selection();
            this->individuals.push_back(offspring);
        }
        if (this->individuals.size() > 1000)
        {
            //this->individuals.erase(this->individuals.begin() + 1000, this->individuals.begin() + this->individuals.size());
        }
        for (size_t i = 1; i < this->individuals.size(); ++i)
        {
            std::vector<T> mutated = mutation(this->individuals[i]);
            if (!this->c(mutated))
            {
                switch (ga.strategy)
                {
                case Strategy::keep_same: this->individuals[i] = this->individuals[i]; break;
                case Strategy::re_mutate:
                {
                    while (!this->c(mutated))
                    {
                        mutated = mutation(this->individuals[i]);
                    }
                    break;
                    this->individuals[i] = mutated;
                }
                case Strategy::remove:
                {
                    if (i == this->individuals.size() - 1)
                    {
                        this->individuals.pop_back();
                    }
                    else
                    {
                        this->individuals.erase(this->individuals.begin() + i, this->individuals.begin() + i + 1);
                    }
                    break;
                }
                case Strategy::none: break;
                default: std::abort();
                }
            }
```

```
                else
                {
                    this->individuals[i] = mutated;
                }
            }
            for (auto& p : stdev)
            {
                p = p + 0.02 * p;
            }
        }
    }
```

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

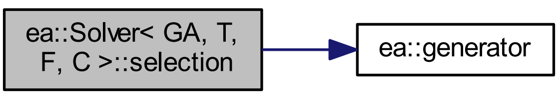
Produce offsrping using r and s indices by crossover

Definition at line 165 of file geneticalgo.h.

References ea::generator().

```
    {
        T xi = quantile(bdistribution, this->distribution(generator));
        size_t r = static_cast<size_t>(std::floor(static_cast<T>(nkeep()) * xi));
        xi = quantile(bdistribution, this->distribution(generator));
        size_t s = static_cast<size_t>(std::floor(static_cast<T>(nkeep()) * xi));
        std::vector<T> offspring = crossover(this->individuals[r], this->individuals[s]);
        return offspring;
    }
```

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:

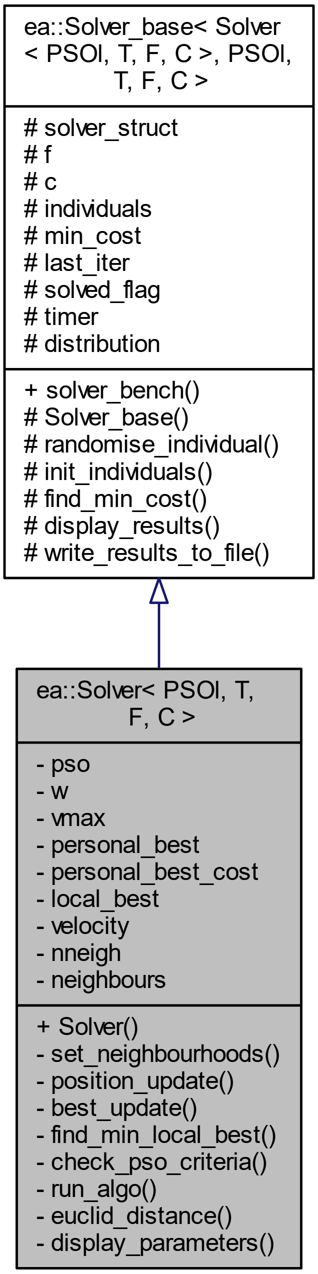
- geneticalgo.h

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

Local Best Particle Swarm Optimisation (PSO) Class.

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

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



Public Member Functions

- Solver (const PSOl< 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 PSOl< 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< PSOl, T, F, C >, PSOl, 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_pso	The particle swarm optimisation parameter structure that is used to construct the solver
f	A reference to the objective function
c	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)
```

```
            {
                personal_best.push_back(p);
                personal_best_cost.push_back(f(p));
            }
            for (auto& p : local_best)
            {
                p = personal_best[0];
            }
            for (size_t i = 0; i < pso.npop; ++i)
            {
                for (const auto& index : neighbours[i])
                {
                    if (personal_best_cost[index] < this->f(local_best[i]))
                    {
                        local_best[i] = personal_best[index];
                    }
                }
            }
            find_min_local_best();
        }
```

7.13.3 Member Function Documentation

7.13.3.1 best_update()

```
template<typename T , typename F , typename C >
void ea::Solver< PSOl, 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.

```
    {
        for (size_t i = 0; i < pso.npop; ++i)
        {
        }
    }
```

7.13.3.2 check_pso_criteria()

```
template<typename T , typename F , typename C >
bool ea::Solver< PSOl, 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.

```
    {
        std::vector<T> distance(pso.npop);
        for (size_t i = 0; i < pso.npop; ++i)
        {
            distance[i] = euclid_distance(this->individuals[i], this->min_cost);
        }
        T rmax = distance[0];
        for (size_t i = 0; i < pso.npop; ++i)
        {
            if (rmax < distance[i])
            {
                rmax = distance[i];
            }
            else
            {
            }
        }
        if (pso.tol > std::abs(this->f(this->min_cost)))// || rmax < pso.tol)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
```

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.

```
        {
            std::stringstream parameters;
            parameters << "C:" << "," << pso.c << ",";
            parameters << "Inertia:" << "," << pso.w << ",";
            parameters << "Maximum Velocity:" << "," << pso.vmax;
            return parameters;
        }
```

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

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

Returns

Distance as a floating-point number

Definition at line 172 of file lbestpso.h.

```
        {
            T sum = 0;
            for (size_t i = 0; i < x.size(); ++i)
            {
                sum = sum + std::pow(x[i] - y[i], 2);
            }
            return std::sqrt(sum);
        }
```

7.13.3.5 find_min_local_best()

```
template<typename T , typename F , typename C >
void ea::Solver< PSOl, 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.

```
    {
        for (size_t k = 0; k < nneigh; ++k)
        {
            if (this->f(local_best[k]) < this->f(this->min_cost))
            {
                this->min_cost = local_best[k];
            }
        }
    }
```

7.13.3.6 position_update()

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

Position update of the particles.

Returns

void

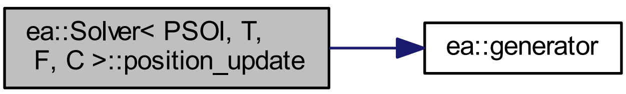
Checks that the candidate is feasible

Definition at line 221 of file lbestpso.h.

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

```
    {
        for (size_t i = 0; i < pso.npop; ++i)
        {
            for (size_t j = 0; j < pso.ndv; ++j)
            {
                std::uniform_real_distribution<double> c(0, pso.c);
                velocity[i][j] = w * velocity[i][j] + c(generator) * (personal_best[i][j] - this->individuals[i][j])
                    + c(generator) * (local_best[i][j] - this->individuals[i][j]);
                if (velocity[i][j] > vmax[j])
                {
                    velocity[i][j] = vmax[j];
                }
                this->individuals[i][j] = this->individuals[i][j] + velocity[i][j];
            }
            if (!this->c(this->individuals[i]))
            {
                this->individuals[i] = personal_best[i];
            }
            if (this->f(this->individuals[i]) < this->f(personal_best[i]))
            {
                personal_best[i] = this->individuals[i];
                personal_best_cost[i] = this->f(this->individuals[i]);
            }
            for (const auto& index : neighbours[i])
            {
                if (personal_best_cost[index] < this->f(local_best[i]))
                {
                    local_best[i] = personal_best[index];
                }
            }
        }
    }
```

Here is the call graph for this function:



7.13.3.7 run_algo()

```
template<typename T , typename F , typename C >
void ea::Solver< PSOl, 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::PSOl< T >::w.

```
{
    for (size_t iter = 0; iter < pso.iter_max; ++iter)
    {
        position_update();
        //best_update();
        find_min_local_best();
        //w = pso.w - (pso.w - 0.4) * (static_cast<T>(iter) / static_cast<T>(pso.iter_max));
        w = pso.w - (pso.w - 0.4) * std::pow((static_cast<T>(iter) / static_cast<T>(pso.iter_max)), inv_pi_sq<T>);
        //w = 0.729;
        //w = 0.5 + distribution(generator) / 2;
        this->last_iter = iter;
        if (check_pso_criteria())
        {
            this->solved_flag = true;
            break;
        }
    }
}
```

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< PSOl, 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.

```
{
    std::unordered_map<size_t, std::array<size_t, 3>> neighbours;
    for (size_t i = 0; i < pso.npop; ++i)
    {
        if (i == 0)
        {
            neighbours[i] = { pso.npop - 1, i, i + 1 };
        }
        else
        {
            if (i == pso.npop - 1)
            {
                neighbours[i] = { pso.npop - 2, i, 0 };
            }
            else
            {
                neighbours[i] = { i - 1, i, i + 1 };
            }
        }
    }
    return neighbours;
}
```

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:

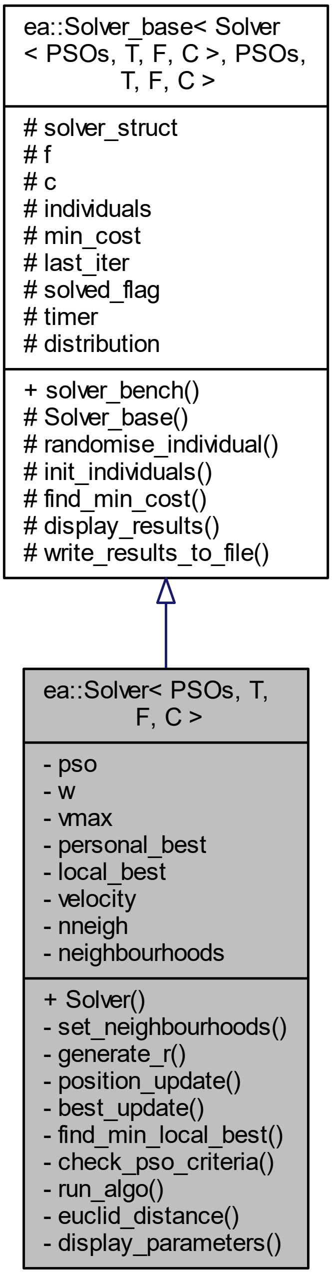
- lbestpso.h

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 r1 and r2 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_pso	The particle swarm optimisation parameter structure that is used to construct the solver
f	A reference to the objective function
c	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)
```

```
            {
                personal_best.push_back(p);
            }
            for (auto& p : local_best)
            {
                p = personal_best[0];
            }
            for (size_t i = 0; i < pso.npop; ++i)
            {
                if (f(personal_best[i]) < f(local_best[neighbourhoods[i]]))
                {
                    local_best[neighbourhoods[i]] = personal_best[i];
                }
            }
            find_min_local_best();
        }
```

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.

```
    {
        for (size_t i = 0; i < pso.npop; ++i)
        {
            if (!this->c(this->individuals[i]))
            {
                this->individuals[i] = personal_best[i];
            }
            if (this->f(this->individuals[i]) < this->f(personal_best[i]))
            {
                personal_best[i] = this->individuals[i];
            }
            if (this->f(personal_best[i]) < this->f(local_best[neighbourhoods[i]]))
            {
                local_best[neighbourhoods[i]] = personal_best[i];
            }
        }
    }
```

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.

```
    {
        std::vector<T> distance(pso.npop);
        for (size_t i = 0; i < pso.npop; ++i)
        {
            distance[i] = euclid_distance(this->individuals[i], this->min_cost);
        }
        T rmax = distance[0];
        for (size_t i = 0; i < pso.npop; ++i)
        {
            if (rmax < distance[i])
            {
                rmax = distance[i];
            }
            else
            {
            }
        }
        if (pso.tol > std::abs(this->f(this->min_cost))) //|| rmax < pso.tol)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
```

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.

```
        {
            std::stringstream parameters;
            parameters << "C1:" << "," << pso.c1 << ",";
            parameters << "C2:" << "," << pso.c2 << ",";
            parameters << "Neighbourhood size:" << "," << pso.sneigh << ",";
            parameters << "Inertia:" << "," << pso.w << ",";
            parameters << "Alpha parameter for inertia:" << "," << pso.alpha << ",";
            parameters << "Maximum Velocity:" << "," << pso.vmax;
            return parameters;
        }
```

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.

```
        {
            T sum = 0;
            for (size_t i = 0; i < x.size(); ++i)
            {
                sum = sum + std::pow(x[i] - y[i], 2);
            }
            return std::sqrt(sum);
        }
```

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.

```
    {
        for (size_t k = 0; k < nneigh; ++k)
        {
            if (this->f(local_best[k]) < this->f(this->min_cost))
            {
                this->min_cost = local_best[k];
            }
        }
    }
```

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

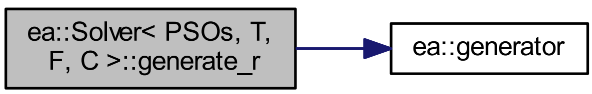
A vector containing r1 and r2

Definition at line 234 of file pso_sub_swarm.h.

References ea::generator().

```
{
    std::vector<std::vector<T>> r(3, std::vector<T>(pso.ndv));
    for (auto i = 0; i < 3; ++i)
    {
        for (size_t j = 0; j < pso.ndv; ++j)
        {
            r[i][j] = (this->distribution(generator));
        }
    }
    return r;
}
```

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.

```
{
    for (size_t i = 0; i < pso.npop; ++i)
    {
        for (size_t j = 0; j < pso.ndv; ++j)
        {
            const auto& r = generate_r();
            velocity[i][j] = 0.729 * velocity[i][j] + //pso.c1 * r[0][j] *
 (personal_best[i][j] - this->individuals[i][j])
                + 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]);
                ;
            if (velocity[i][j] > vmax[j])
            {
                velocity[i][j] = vmax[j];
            }
            this->individuals[i][j] = this->individuals[i][j] +
velocity[i][j];
        }
    }
}
```

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.

```
{
    for (size_t iter = 0; iter < pso.iter_max; ++iter)
    {
        position_update();
        best_update();
        find_min_local_best();
        w = pso.w - (pso.w - 0.4) * std::pow((static_cast<T>(iter) / static_cast<T>(
pso.iter_max)), inv_pi_sq_2<T>);
        this->last_iter = iter;
        if (check_pso_criteria())
        {
            this->solved_flag = true;
            break;
        }
    }
}
```

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.

```
{
    std::unordered_map<size_t, size_t> neighbourhoods;
    size_t neigh_index = 0;
    size_t counter = 0;
    for (size_t i = 0; i < pso.npop; ++i)
    {
        if (counter < pso.sneigh)
        {
            neighbourhoods[i] = neigh_index;
            counter = counter + 1;
        }
        else
        {
            neigh_index = neigh_index + 1;
            counter = 0;
        }
    }
    return neighbourhoods;
}
```

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 aroung 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_solver_struct	The parameter structure that is used to construct the solver
i_f	A reference to the objective function
i_c	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 << ",";
```

```
        results << "Elapsed Time:" << "," << timer << ",";
        results << "Starting Values:" << "," << solver_struct.decision_variables << ",";
        results << "Standard Deviation:" << "," << solver_struct.stdev << ",";
        results << "Initial Population:" << "," << solver_struct.npop << ",";
        results << "Tolerance:" << "," << solver_struct.tol << ",";
        results << "Maximum Iterations:" << "," << solver_struct.iter_max << ",";
        results << "Using Penalty Function:" << "," << solver_struct.use_penalty_method << ","
    ;
        results << "Using Constraints:" << ",";
        switch (solver_struct.constraints_type)
        {
        case(Constraints_type::normal): results << "Normal" << ","; break;
        case(Constraints_type::tight): results << "Tight" << ","; break;
        case(Constraints_type::none): results << "None" << ","; break;
        }
        results << static_cast<Derived*>(this)->display_parameters().str() << "\n";
        return results;
    }
```

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.

```
    {
        for (const auto& p : individuals)
        {
            if (f(min_cost) > f(p))
            {
                min_cost = p;
            }
        }
    }
```

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 aroung 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.

```
    {
        std::vector<std::vector<T>> individuals(solver_struct.npop, std::vector<T>(solver_struct.ndv));
        for (auto& p : individuals)
        {
            p = randomise_individual();
            while (!c(p))
            {
                p = randomise_individual();
            }
        }
        return individuals;
    }
```

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.

```
    {
        std::vector<T> individual = solver_struct.decision_variables;
        T epsilon = 0;
        for (size_t j = 0; j < solver_struct.ndv; ++j)
        {
            std::normal_distribution<T> ndistribution(0, solver_struct.stdev[j]);
            epsilon = ndistribution(generator);
            individual[j] = individual[j] + epsilon;
        }
        return individual;
    }
```

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

problem_name	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         {
280             const std::chrono::time_point<std::chrono::system_clock> start = std::chrono::system_clock::now();
281             static_cast<Derived*>(this)->run_algo();
282             const std::chrono::time_point<std::chrono::system_clock> end = std::chrono::system_clock::now();
283             const std::chrono::duration<double> elapsed_seconds = end - start;
284             timer = elapsed_seconds.count();
285         }
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

problem_name	The name of the problem in std::string form

Returns

void

Definition at line 258 of file ealgorithm_base.h.

```
{
    std::string filename;
    filename.append(problem_name);
    //filename.append("-");
    //filename.append(solver_struct.type);
    filename.append("-results.csv");
    std::ofstream out;
    out.open(filename, std::ofstream::out | std::ofstream::app);
    out << display_results().str();
}
```

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

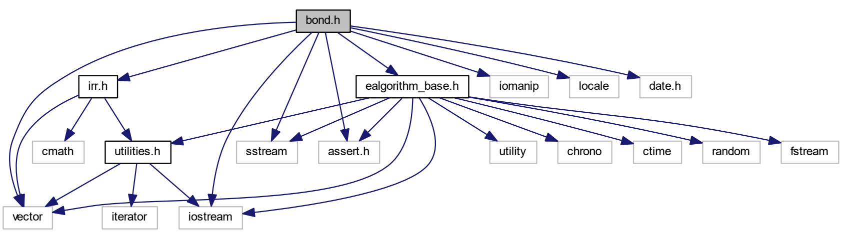
File Documentation

8.1 bond.h File Reference

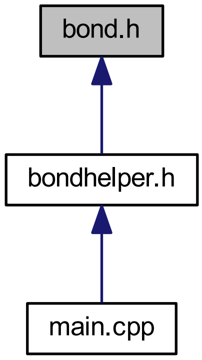
Classes and functions for bonds and their internal rate of return.

```
#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

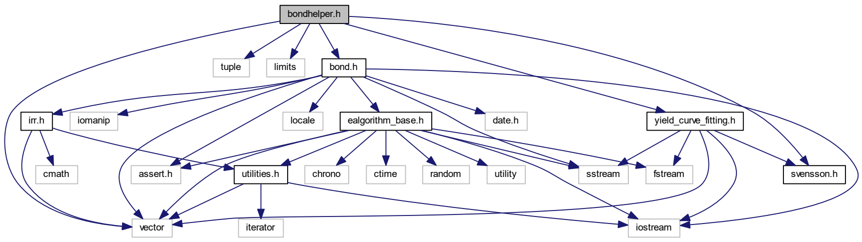
Ioannis Anagnostopoulos

8.2 bondhelper.h File Reference

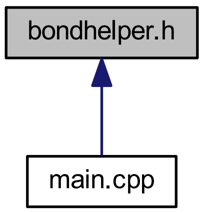
Classes and functions for the bond pricing problem.

```
#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.

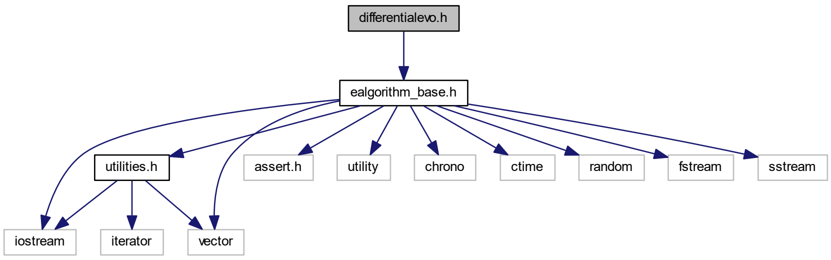
Author

Ioannis Anagnostopoulos

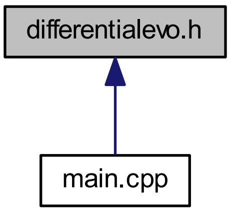
8.3 differentialevo.h File Reference

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

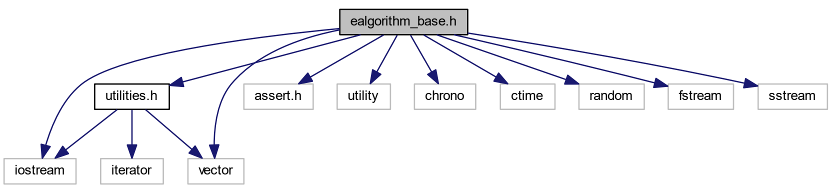
Ioannis Anagnostopoulos

8.4 ealgorithm_base.h File Reference

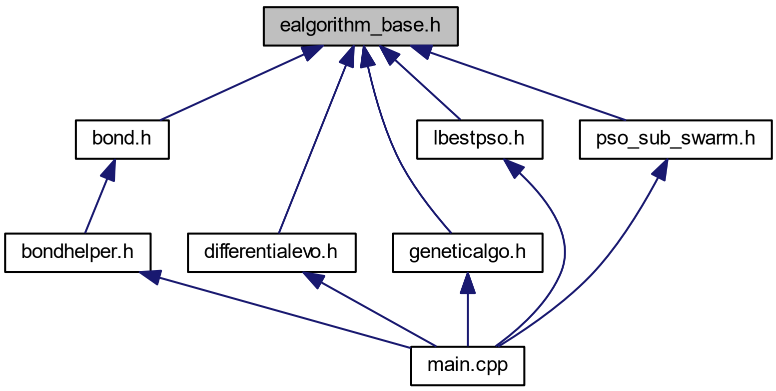
Classes and functions for the base of the solvers.

```
#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

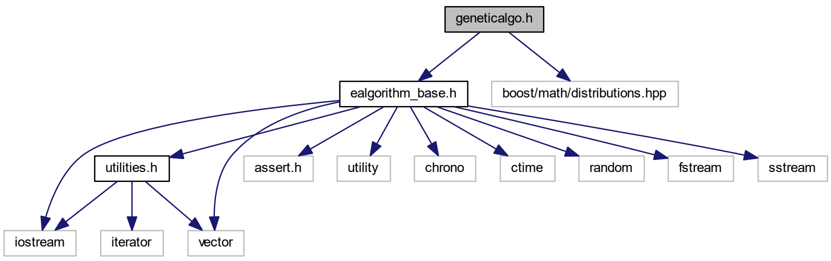
Ioannis Anagnostopoulos

8.5 geneticalgo.h File Reference

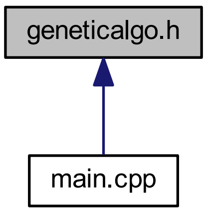
Classes and functions for Genetic Algorithms.

```
#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

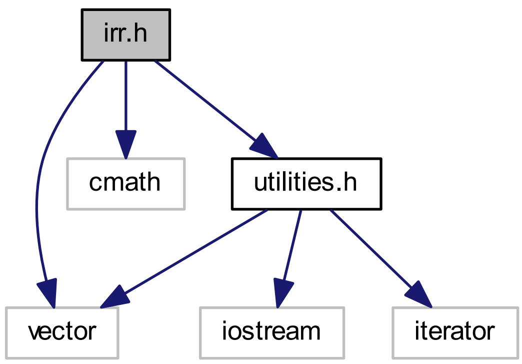
Ioannis Anagnostopoulos

8.6 irr.h File Reference

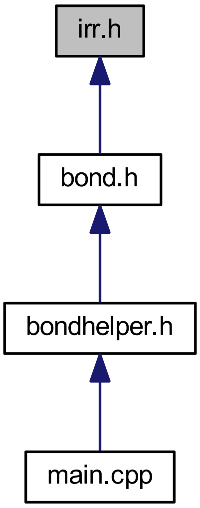
Functions for the Internal Rate of Return.

```
#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

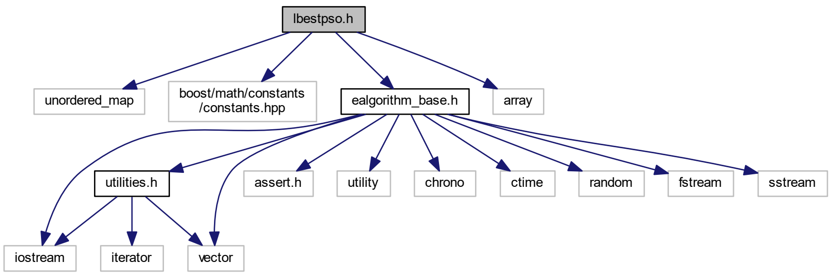
Ioannis Anagnostopoulos

8.7 lbestpso.h File Reference

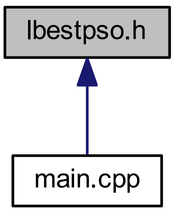
Classes and functions for the lbest (Local Best) Particle Swarm Optimisation with a ring topology.

```
#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::PSOl< T >

  Local Best Particle Swarm Optimisation Structure, used in the actual algorithm and for type deduction.
- class ea::Solver< PSOl, 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

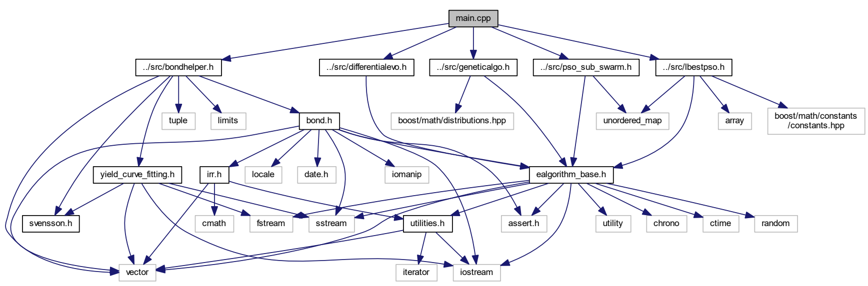
Ioannis Anagnostopoulos

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/pso_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, PSOl} with a specific floating-point number type, setting all of its parameters throught 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.

```
{
    using namespace yft;
    using namespace bond;
    const std::vector<double> stdev { 0.7, 0.7, 0.7, 0.7, 0.7, 0.7 };
    const std::vector<double> stdev_ga{ 0.5, 0.5, 0.5, 0.5, 0.5, 0.5 };
    double irr_tol = 0.00000001;
    double tol = 0.0001;
    double tol_f = 0.001;
    Interest_Rate_Helper<double> ir{ read_ir_from_file<double>("
   interest_rate_data_periods.txt") };
    BondHelper<double> de{ read_bonds_from_file<double>("bond_data.txt"), DF_type::exp };
    DE<double> de_irr{ 1, 0.6,{ 0.05 },{ 0.7 }, 10, irr_tol, 500, false, Constraints_type::normal
   , true, true };
    DE<double> de_irr_check{ 1, 0.6,{ 0.05 },{ 0.7 }, 10, irr_tol, 500, false,
   Constraints_type::normal, false, false };
    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};
    PSOl<double> pso_irr{ 1.49618, 0.9, { 1000000 },{ 0.05 },{ 0.7 }, 22, irr_tol, 3000, false,
   Constraints_type::normal, true, true};
    auto decision_variables = de.set_init_nss_params(de_irr);
    DE<double> de_pricing{ 1, 0.6, decision_variables, stdev, 60, tol, 500, false,
   Constraints_type::tight, true, true };
    DE<double> de_fitting{ 1, 0.6, decision_variables, stdev, 60, tol_f, 500, false,
```

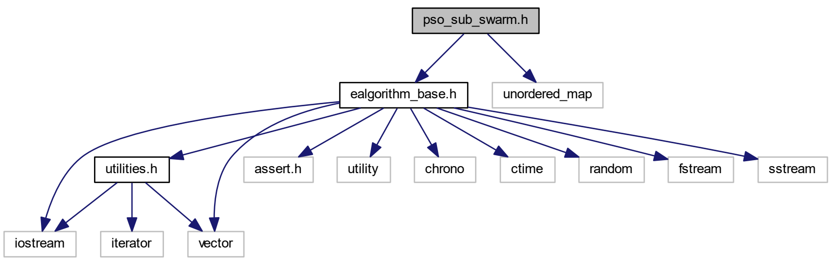
```
    Constraints_type::tight, true, true };
 GA<double> ga_pricing{ 0.4, 0.35, 6.0, decision_variables, stdev_ga, 250, tol, 2000, false,
    Constraints_type::tight, Strategy::remove, true, true };
 GA<double> ga_fitting{ 0.4, 0.35, 6.0, decision_variables, stdev_ga, 250, tol_f, 2000, false,
    Constraints_type::tight, Strategy::remove, true, true };
 PSOl<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 };
 PSOl<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 };
 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 };
  for (size_t i = 0; i < 100; ++i)
  {
      de.bond_pricing(ga_pricing, de_irr_check, Bond_pricing_type::bpp);
      ir.yieldcurve_fitting(ga_fitting);
      de.bond_pricing(de_pricing, de_irr_check, Bond_pricing_type::bpp);
      ir.yieldcurve_fitting(de_fitting);
      de.bond_pricing(pso_pricing, de_irr_check, Bond_pricing_type::bpp);
      ir.yieldcurve_fitting(pso_fitting);
      de.set_init_nss_params(de_irr);
      de.set_init_nss_params(pso_irr);
      de.set_init_nss_params(ga_irr);
  }
  return 0;
}
```

8.9 pso_sub_swarm.h File Reference

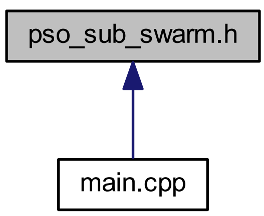
Classes and functions for the initial implementation of Sub-Swarm Particle Swarm Optimisation.

```
#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.

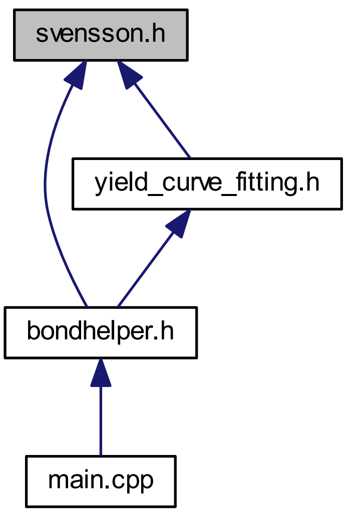
Author

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

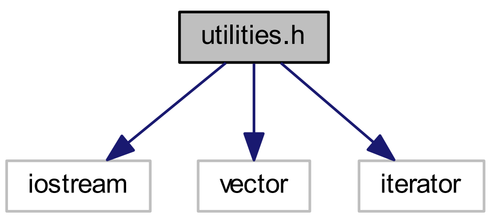
Ioannis Anagnostopoulos

8.11 utilities.h File Reference

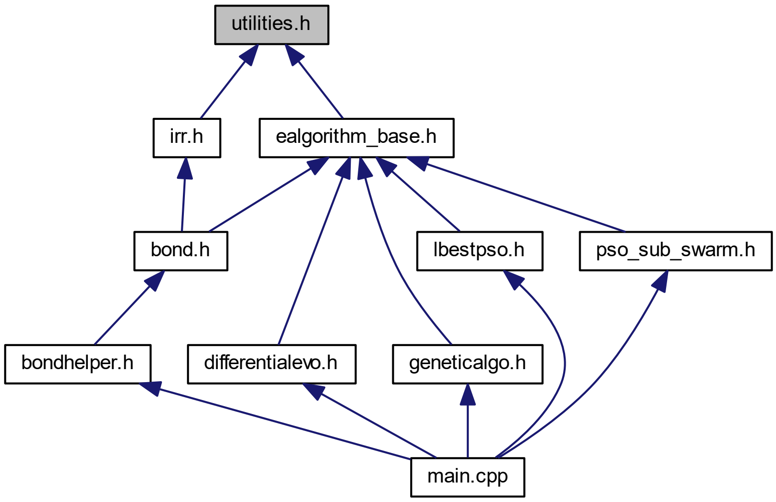
Enumerations and functions used from the rest of the project.

```
#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

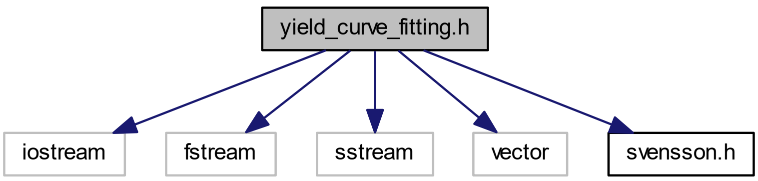
Ioannis Anagnostopoulos

8.12 yield_curve_fitting.h File Reference

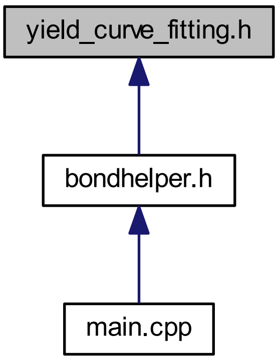
Class and functions for the yield curve fitting problem.

```
#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