OCTARISK Documentation

version 0.4.3

Stefan Schloegl (schinzilord@octarisk.com)

Table of Contents

octarisk Documentation

This manual is for the market risk measurement project octarisk (version 0.4.3).

1 Introduction

1.1 Why quantifying market risk?

This ongoing project is made for all investors who want to dig deeper into their portfolio than just looking at the yearly profit or loss. Although most financial reports of more sophisticated brokers contain risk measures like standard deviations, the volatility alone cannot cover the risk inherent in non-linear financial products like options. Moreover, potential investors care about their portfolio values under certain market conditions, e.g. they want to compare their perceived personal stress levels during the financial crisis but with the financial instruments in their portfolio losses during stress scenarios.

If questions like

- What happens to the portfolio value, if the ECB increases the interest rate by 100bp?
- What is the least amount of money, a given portfolio will loose in one out of 100 events during the next year?
- How would a given portfolio perform, if a crash comparable to Black Friday 1987 takes place?

are of potential interest for you, then the OCTARISK market risk project might be satisfying your needs for a professional risk modelling framework.

Since most investors do not give excessive credits to debtors or bear operational or liquidity risks, the OCTARISK project focuses on market risk only - all remaining types of risk which are relevant for your investment portfolio: equity risk, interest rate risk, volatility risk, commodity risk, ...

For the assessment of these market risk types, a sophisticated full valuation approach with Monte-Carlo based value-at-risk and expected shortfall calculation is performed. The underlying principles are state-of-the-art in financial institutions and are used in internal models to fulfill the requirements set by regulators (for Basel III and Solvency 2). The important concepts are adopted, the unnecessary overhead was skipped - resulting in a fast, lightweight yet flexible approach for quantifying market risk.

1.2 Features

The OCTARISK quantifying market risk projects features

- a full valuation approach for financial instruments
- Monte-Carlo method for scenario generation of underlying risk factors
- simple input interface via static and dynamic files
- a processable report incl. graphical representation of portfolio profits and losses as well as an overview over the riskiest instruments and positions (see See Section 3.4 [Output files], page 25 for examples)
- an easily customizable framework based on full implementation on Octave with compact source code

1.3 Prerequisites

The only requirement is GNU Octave (tested for versions > 4.0) with installed financial package and hardware with minimum of 4Gb of memory. Calculation time decreases significantly while using optimized linear algebra packages of OpenBLAS and LAPACK (or comparable). For automatic processing of the input data (e.g. to get actual market data from Quandl or Yahoo finance), to make the parameter estimation as well as process the report files, some programming language like Perl, Python and a running LaTeX environment are recommended, but not required.

Nevertheless, a basic understanding of a high level programming language like Octave is required to adjust the source code and to customize the calculation. Furthermore, a thorough understanding of financial markets, instruments and valuation will be needed in order to select appropriate models and to interpret results. The implemented models and the underlying concepts are explained in detail for example in following literature:

Risk Management and Financial Institutions, John C. Hull, 2015
Paul Wilmott on Quantitative Finance, 2nd Edition, Paul Wilmott, 2006
Options, Futures and other Derivatives, 7th Edition, John C. Hull, 2008

The next chapter contains the background and details needed for running the market risk valuation and aggregation software and understanding the risk measures.

2 User guide

This chapter gives a general introduction to market risk and how the market risk is captured by OCTARISK. Thereby the following convention is made: market movement means the movement around the mean value or, in other words, the possible deviation from the expected value as time passes. Stronger movement around the expected value means more risk and results in a higher standard deviation, the most important statistic parameter for capturing market risk.

2.1 Introducing market risk

Typically, market risk can be divided into several sub types. Most of the splitting is obvious, but some of the more exotic risk types remain very broad. The ideal breakdown depends heavily on the specific portfolio whose market risk shall be quantified. A basic approach fitting most portfolios of private investors, not too broad, not too granular, is chosen in OCTARISK. The following types of risk are defined:

- *FX*: Forex risk captures the market movements of exchange rate values. These movements relative to the reporting currency (in our case EUR) affect financial assets in foreign currencies only.
- *EQ*: Equity risk is related to the movement of equity markets. It will be typically broken down into sub-categories like countries, regions or other types of aggregation (e.g. developed market, emerging market, frontier markets).
- *COM*: Commodity risk. This is a rather broad type of risk, often correlated to other risk types (e.g. equity). Commodity risk tries to capture the movement of spot and future / forward commodity prices, as well as commodity linked equities.
- *IR*: Interest rate risk is related to the movement of interest rate values. This is the most important type of risk for cash flow bearing instruments.
- *SPREAD*: Spread risk is related to the movement of spread rates. This is the second most important type of risk for cash flow bearing instruments.
- *INFLATION*: Inflation risk is related to the not anticipated changes in inflation rates, e.g. changes in the inflation expectation curve. This risk type affects inflation linked instruments only.
- *RE*: Real estate risk, where typically it will be distinguished between movements of market values of REITs (Real Estate Investment Trusts) and housing prices. This risk type can also be broken down into sub-categories like different countries, regions or other categories (e.g. developed markets, emerging markets).
- *VOLA*: Implied volatility risk, e.g. the anticipated future movement of implied volatility of equity instruments or swaps.
- *ALT*: Alternative market risk, used as a container for every other type of risk not already captured (e.g. Bitcoins, infrastructure)

The OCTARISK projects focuses on these risk types. For each risk type, appropriate risk factors can be chosen. One risk factor is a typical representative of a particular risk type. Most often, several risk factors are needed to describe the granular behavior of a market risk type, e.g. it is necessary to describe the specific characteristics of countries, regions or currencies. For example, two risk factors can be used to describe the international equity market movements: Developed markets and emerging market. If desired, developed market can be further split into North America, Europe and Asia to describe risk diversification effects between these broad regions. After the selection of risk factors, stochastic models are chosen, calibrated and subsequently used for scenario generation.

2.2 Market risk measures

2.2.1 Value-at-Risk

Value-at-risk (VAR) is defined as the monetary loss which the portfolio won't exceed for a specific probability on a certain time horizon. As an example, a 250 trading day (one calender year) VAR of 1000 EUR at the 99% confidence interval means there is a probability of 99% that the portfolio loss within one year (250 trading days) is equal to or less than 1000 EUR. It is important to note that no forecast is made for the possible loss which can occur in 1% of the remaining cases. Furthermore, within a proper calibrated risk setup one **must** expect a loss greater than the VAR amount in 1% of all cases, that means a loss of more than 1000 EUR will occur in two to three trading days per year. Otherwise, if there are no trading days observed where the loss is greater than predicted by VAR, the risk is overstated, leaving room for better usage of risk capital.

2.2.2 Expected shortfall (ES)

Expected shortfall is an additional risk measure which is defined as the arithmetic average (mean) loss in the remaining tail of the sorted profit and loss distribution of all simulated MC scenarios, which are not covered by the 99% VAR. The ES should always be seen in context of the VAR and is a more coherent risk measure, which can make stronger predictions about diversification benefits of portfolios.

2.3 Scenario generation

A scenario is a specific set of shocks to risk factors. Typically, the directions of the shocks are correlated, and the value of the shock is dependent on the stochastic properties of the risk factors (e.g. volatility or mean reversion parameters). Although these properties can be also chosen customary, for evaluating risk measures like value-at-risk these parameters are typically extracted from past, real market movements.

2.3.1 Random number generation

During MC scenario generation, uniform distributed random numbers are used to generate either normal distributed or t-distributed numbers for risk factor shocks. Octave's built in pseudo-random number generator (Mersenne-Twister) is used for default. It is possible to specify a custom and / or stable seed in order to always get the same results (useful in regression testing). Moreover, OCTARISK allows to use Sobol quasi-random numbers for scenario generation. A manual seed and custom directoin numbers can be specified. Statistical tests (e.g. Jarque-Berra tests and distribution moment calculations) may be performed on the qualities of drawn random numbers to match the specified statistical properties.

2.3.2 Stochastic models

In order to describe movements of risk factors in time, a connection has to be made between statistical behavior of real time series and stochastic processes for modeling synthetic time series. OCTARISK concentrates on three stochastic processes: Wiener, Ornstein-Uhlenbeck and root-diffusion processes.

2.3.2.1 Random walk and the Wiener process

For the Wiener process, two different possible definitions exist. In the first case, both the drift and the normally-distributed random number W_t (the so called Wiener process) are proportional to the variable at the former time step, resulting in the process S_t which satisfies the following stochastic differential equation:

$dS_t = mu * S_{t-1} * dt + sigma * S_t * dW_t$
The following analytic solution to this stochastic differential equation is derived:

$S_t = S_0 * exp((mu - sigma^2/2) * t + sigma * W_t)$
This solution ensures positive values at all time steps.

In the second case, a continuous time random walk with independent, normally-distributed random numbers independent of the variable at the former time step is given by the following stochastic differential equation:
$dS_t = mu * dt + sigma * W_t$
This process shows self-similarity and scaling behavior.

2.3.2.2 Ornstein-Uhlenbeck process

The Wiener process can be extended to incorporate a serial dependency (like a memory) - tomorrows values are dependent on the level of todays values. The Ornstein-Uhlenbeck (OU) process has a mean-reversion term, which is directly proportional to the difference of the actual value from the mean reversion level:
$dX_t = mr_{rate} * (mr_{level} - X_{t-1}) * dt + sigma * dW_t$
where the mean reversion rate mr_{rate} can be seen as a proportional parameter of a restoring force. The increments dX_t tend to point to the mean-reversion level mr_{level}. The result of the formula is an addditive term to the risk factor depending on the past level and a stochastic term (modeled by the Wiener process).

2.3.2.3 Square-root diffusion process

In order to exclude negative values in the OU mean-reversion process, an additional repelling force is needed, which ensures that the level of the stochastic variables stays away from zero. The square-root diffusion process (SRD) has an additional term, which is multiplied with the standard deviation and the random variable. This term is identified as the square root of the variable at the former time step:
$dX_t = mr_{rate} * (mr_{level} - X_{t-1}) * dt + sigma * sqrt(X_{t-1}) * dW_t$
Negative values for the variables are excluded if the following equation is fulfilled:
$2 * mr_{rate} * mr_{level} >= sigma$

2.3.3 Financial models

The stochastic models which have been presented in the last section, are used as basis for financial models. In order to map stochastic processes to financial models, properties of financial models are identified and subsequently stochastic models chosen in order to generate simulated time series of risk factors with appropriate and desired behavior.

2.3.3.1 Geometric Brownian motion

The standard financial model for equity, real-estate and commodity risk factors is a geometric Brownian motion (GBM), which utilizes the extended Wiener process, where the drift and random variable are proportional to the risk factor value. Due to this proportionality the time series can not reach zero and the modeled risk factors values always stay positive. This is reflected in the real world behavior of equity or commodity prices, where the intrinsic value of these assets cannot fall below zero.

2.3.3.2 Brownian motion

In a Brownian motion (BM) model, the time-series increments are a function of drift, time and standard deviation, but not dependent on the actual level of the financial variable. For certain types of risk factors (e.g. interest rates), one assumes a Brownian motion so that the modeled price movements are given as additive shocks which are completely independent on the actual risk factor value. Therefore, negative values of the modeled risk factors are allowed.

2.3.3.3 Vasicek model

In the long run, the market movements of exchange rates and interest rates seem to be mean-reverting. This behavior can be modeled by a Ornstein-Uhlenbeck process resulting in a so called one factor short rate model proposed by Vasicek. The Ornstein-Uhlenbeck process allows for negative values, which reflects the real world behavior of short rates since the financial crisis, where interest rates of AAA-rated government bonds had negative yields for at east some time.

2.3.3.4 Cox-Ingersoll-Ross and Heston model

If one doesn't want to allow negative values for interest rates or other mean-reverting risk factors, a square-root diffusion process can be chosen as stochastic model. This results in the short-rate model of Cox-Ingersoll-Ross or the Heaston model for modelling at-the-money volatility used in option pricing.

2.3.4 Parameter estimation

Once an appropriate model is chosen for the risk factor, one has to define input parameters for the stochastic differential equations. One approach is to use historical data to estimate statistic parameters like volatility, correlations or mean-reversion parameters. Another approach is to apply expert judgment in selecting input parameters for the models. OCTARISK uses the specified parameters to generate Monte-Carlo scenarios - the selection of the parameter estimation approach is up to the reader.

Typically, parameters are extracted from historical time series on weekly or monthly data on the past three to five years. Unfortunately, availability of historical time series for

all risk factors is one of the main constraints in parameter estimation for private investors. Most often, one has to overcome problems of missing data and the need for interpolation or extrapolation. In future versions scripts for parameter estimation will be provided.
In an ideal world, at first appropriate risk types and risk factors are selected, then the validation of a stochastic model is performed and the appropriateness of the model and the assumptions (like length of historical time series) is verified in back-tests. Nevertheless, no stochastic model can completely describe the financial markets, giving rise to model error (both in parameter estimation and model selection).

2.3.5 Monte-Carlo Simulation

A Monte-Carlo approach is chosen to generate the risk factor shocks in all scenarios. Therefore a risk factor correlation matrix (e.g. estimated from historical time series) and additional parameters describing statistical distributions can be used to generate random numbers, which are utilized as input parameters to stochastic models and finally lead to correlated risk factor shocks with the desired properties.

In order to account for non-normal distributions and higher order correlation effects in the Monte-Carlo simulation, a copula approach is chosen to generate dependent, correlated random numbers. In a first step, the input correlation matrix is used to generate normally distributed, correlated random numbers with zero drift and unit variance. In a next step, either a Gaussian copula or a student-t copula is utilized to transform the normally distributed random variables to uniformly distributed random variables, while either the linear correlation dependence (for Gaussian copulas) or additionally the non-linear dependence structures (for student-t copulas) is preserved. In a last step, these random numbers are incorporated into a function which chooses for each risk factor the appropriate distribution in a certain way, that standard deviation, skewness and kurtosis are matched with the input parameters. Therefore the Pearson distribution system is used as a basis for generation of random variables. Subsequently, in each of the Monte-Carlo scenarios a specific set of correlated risk factor shocks, dependent on the selected financial models, is generated, while the marginal distributions have desired standard deviation, skewness and kurtosis.

A further potential problem is the model error from the Monte-Carlo method. Only a limited number of Monte-Carlo scenarios can be generated and valuated (in the range of 10000 to 100000), thus leaving space for not represented scenarios which could alter the risk measures.

2.3.6 Stress testing

A complementary method to the stochastic scenario generation is to directly define shocks to risk factors. This scenario analysis is normally done in order to calculate the portfolio behavior in well known historic scenarios (like Financial Crisis 2008, devaluation of Asian currencies in the mid 90s, terrorist attack on 9/11, Black Friday in October 1987) or in scenarios, where one only is interested in the behavior of the portfolio value in isolated shocks (like all equities decline by 30pct in value, a +100bp parallel shift in interest rates). Possible sources of scenarios are provided by regulators and can be easily adapted for personalized stress scenarios.

2.4 Instrument valuation

After the scenarios are generated, one has to calculate the behavior of financial instruments to movements in the underlying risk factors. Therefore two different approaches are chosen: the sensitivity approach applies relative valuation adjustments to the instrument base values proportional to the defined sensitivity. The valuation adjustments are derived from movements in value of the underlying risk factors. Secondly, in a full valuation approach, the scenario dependent input parameters are fed into a pricing function which (re)calculates the new absolute value of the instrument in each particular scenario.

2.4.1 Sensitivity approach

The sensitivity approach is performed for all instruments which have a linear dependency on the movement of underlying risk factor values, or where not enough data or no appropriate pricing function exists for a full valuation approach.

2.4.1.1 Linear dependency on risk factor shocks

For the risk assessment of equities, commodities or real estate instruments it is appropriate to take only the linear dependency on risk factor movements into account. Each risk factor has sensitivities to underlying risk factors. An instrument inherits the amount of shock proportional to the defined sensitivity value from the underlying risk factors.

A typical example is a developed market exchanged traded fund (ETF) with exposure to North America, Europe and Asia. The ETF will follow the price movements of these three risk factors, so the sensitivities are simply the relative exposure to the three equity markets (e.g. 0.5, 0.3 and 0.2). These sensitivities are then used to calculate the weighted shock that is applied to the actual instrument value. The same principle holds for single stock, where sensitivities to appropriate risk factors and to an idiosyncratic risk term (an uncorrelated random number) can be selected. A useful method for calibration is the multi-dimensional linear regression. The resulting betas from this regression can be taken for the sensitivities to regressed risk factors. The remaining alpha and estimation error resembles the sensitivity to the idiosyncratic risk. The exposure to the uncorrelated random number can be derived from one minus the adjusted R_square of the regression. Since the R_square gives the amount of variability that is explained by the regression model, one minus R_square is equal to the amount of uncorrelated random fluctuations which are not covered by the input parameters.

2.4.1.2 Approximation with sensitivity approach

For instruments with insufficient information one can also choose the sensitivity approach. One example are funds consisting mainly of bonds. Without look-through, one has no information about the exact cash flows of the underlying bonds. Instead, often the duration (and convexity) of the fund is known. These two types of sensitivity (duration and convexity) can then be used to calculate the change in value to interest rate shocks. Therefore, the absolute interest rate shock at the node, which equals the Macaulay duration, is calculated as absolute difference compared to the base rate. This change in interest rate level (dIR) is directly transformed to a relative value shock (dV) by the formula
$dV = duration * dIR + convexity * dIR^2$
incorporating both sensitivities.

2.4.2 Full valuation approach

The core competency of a quantitative risk measurement project is full valuation, where the absolute value of financial instruments is calculated from raw input parameters by a special pricing function. Some input parameters to the pricing function are scenario dependent, other are inherent to the instrument. The most important input parameters have to be modeled by stochastic processes and can subsequently be fed into the pricing function, where the new, scenario dependent absolute value of the instrument is calculated. At the moment, the following full valuation pricing functions are implemented in OCTARISK:

2.4.2.1 Option pricing

European plain-vanilla options are priced by the Black-Scholes model (See Section 5.61 [option_bs], page 102). The Black-Scholes equation provides a best estimate of the option price. The underlying financial instrument and implied volatility are modeled as risk factors in order to calculate the new option price in each scenario. The risk free rate will be also made scenario dependent in order to capture the interest rate sensitivity of the option price. Further information is provided by numerous textbooks.

For American options a more sophisticated model has to be used for pricing. Unfortunately, binomial models (like the Cox-Rubinstein-Ross model) or finite-difference models are not feasible for a full valuation Monte-Carlo based approach, since the computation time for a large amount of time steps and MC scenarios is too high due to missing parallelization opportunities. Instead, a Willow-Tree model is implemented to price American options. Within that model, instead of using the full binomial tree with increasing number of nodes per time step, a constant number of nodes at each pricing time step is utilized to approximate the movements of the underlying price. With optimized transition probabilities, the whole model relies on a smaller amount of total nodes which significantly decreases computation time and lowers memory consumption (See Section 5.63 [option_willowtree], page 103 implementation for further details).

A calibration is performed to align the model price based on provided input parameters with the observed market price. This calibration calculates an implied spread which is added to the modeled volatility as a constant offset.

The implied volatility itself is dependent on the option strike level and time to maturity (term). In order to grasp that behavior, the so called volatility smile is modeled by a moneyness vs. term volatility surface, where changes in the spot price lead to moneyness changes. Therefore, the actual implied volatility behavior (at-the-money implied volatility vs. moneyness vs. term) of the market is preserved for the pricing.

Amongst simple underlying instruments or indizes, baskets of several indizes or instrument can be used. A diversified basket volatility is calculated which serves as im plied volatility for the option derivative. Baskets itself are modeled as synthetic instruments.

Besides plain vanilla options, closed-form solutions are used for pricing of European Barrier and European Asian options. Continously geometric average asian options are priced by Kemna and Vorst model of 1990, while arithmetic average asian options are priced by Levy (1992) model.

2.4.2.2 Swaption pricing

European plain-vanilla swaptions are priced via the closed form solution of the Black-76 model or the Bachelier model. (See Section 5.76 [swaption_black76], page 107 and See Section 5.75 [swaption_bachelier], page 107 for details). Again, a calibration is performed to align the model price based on provided input parameters with the observed market price. The calibrated implied volatility spread is subsequently added to the modeled volatility as a constant offset. The volatility smile for the specific term is also given by volatility cubes. The interest rate implied volatility can be set up by three axes: underlying tenor, swaption term and moneyness. For each of these combinations an implied volatility can be set. For Swaption underlyings either discount curves or fixed and floating leg swaps can be used.

2.4.2.3 Forward and Future pricing

Equity and Bond forwards and futures can be valuated. Therefore market indizes can be set up, which serve as underlyings for the forwards. The value of the forward or future is calculated as the payoff at maturity (underlying value minus strike) discounted back to the valuation date. For futures, net basis and accrued interest can be taken into account. (See Section 5.65 [pricing_forward], page 104 for details).

2.4.2.4 Cash flow instrument pricing

Cash flow instruments are specified by the following sets of variables: cash flow dates and corresponding cash flow values. Moreover, each cash flow instrument has an actual market price and an underlying interest rate curve, which has to be provided as a separate risk factor. Before the full valuation can be carried out, the spread over yield is calculated to align the observed market price with the value given by the pricing function. The spread over yield is then assumed to be a constant offset to the scenario dependent interest rate spot curve. For each scenario, all cash flows are discounted with the appropriate interest rate and spread curve. The present value is then given by the sum of all discounted cash flows. Credit spreads are modeled as separate risk factors, thus capturing credit spread risk for cash flow instruments.

2.4.2.5 Bond instrument pricing

The Bond instrument class covers the full spectrum of plain vanilla bond instruments: fixed rate bonds, floating rate notes, fixed and floating swap legs, fixed rate amortizing bonds, mortage backed securities with prepayments, floating swap legs based on CMS rates, and many more. During pricing, at first the cash flows are rolled out. The forward rates for calculation of floating payments are scenario dependent. After the rollout is done, a spread over yield is calibrated in order to match the market price with the theoretical value. For calculation of the net present value of the bonds, a discount curve can be set. If the curves have attached risk factors, the pricing will be fully scenario dependent. CMS floating swaps can also have special cash flowst based on averages or capitalized CMS rates. Moreover, convexity adjustment (according to Hull or Hagan) can be taken into account.

Moreover, bonds with embedded call or put options can be modelled. Therefore a trinomial Hull-White tree is used to price these embedded European or American bond options. See (See Section 5.60 [option_bond_hw], page 101 for further details.

2.4.2.6 Cap and floor instrument pricing

The CapFloor class covers caps and floor instruments on discount curves. The cash flow rollout of these caps and floors also covers CMS rates and convexity adjustment. Both Black and Normal models are implemented, thus allowing for negative interest rates.

2.4.3 Synthetic instruments

In order to model a fixed share combination of instruments, the synthetic class was introduced. The value of the synthetic instrument is calculated as the linear combination of all underlying instrument. Synthetic instruments can be used to e.g. model portfolios with full look-through or to combine fixed and floating legs to swaps. Moreover, more complex instrument including option behaviour can be modeled, if e.g. a bond and a option is combined to an instrument with embedded call / put optionality.

2.4.4 Stochastic instruments

In order to pre-calculate cash flow values and instrument prices in other risk systems, the stochastic instrument class was introduced. Based on modelled risk factor values (e.g. correlated uniformly or normal distributed random variables), random numbers are used to determine quantile numbers and then draw cash flows or values from special curves or surfaces. These objects are used as storage containers for quantile dependent values.

2.5 Aggregation

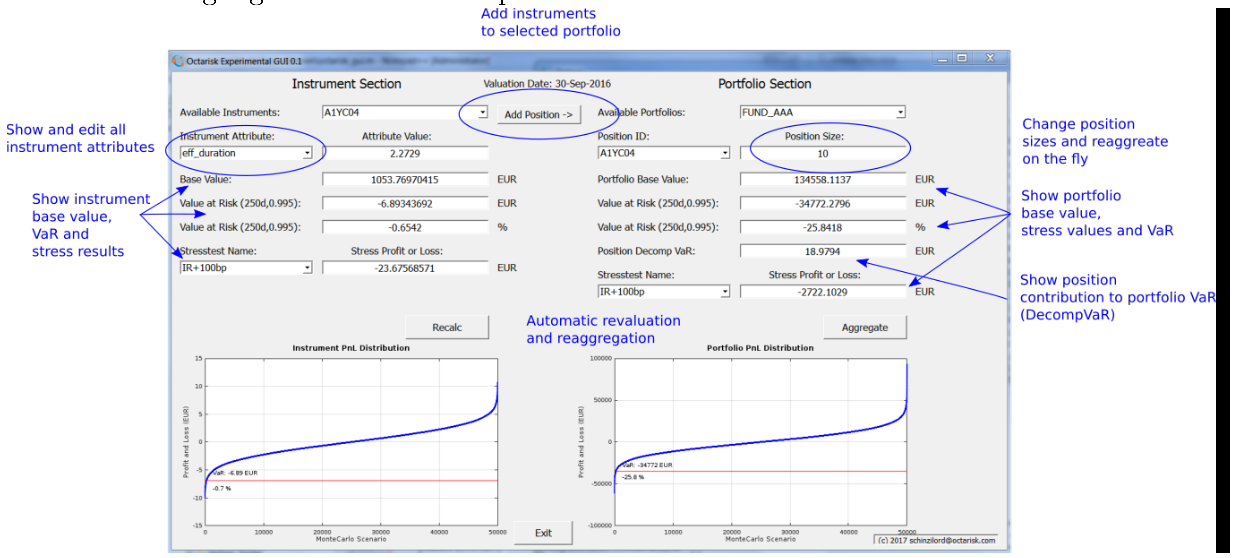
After the valuation of all instruments in each scenario, one has to aggregate all parameters of instruments contained in a fund. Therefore all fund position values are derived by multiplying the position size with the instrument value in each scenario. The resulting sorted fund profit and loss distribution is then used to calculate the value-at-risk at fund level.
If less than 50001 MC scenarios are used in OCTARISK, it is recommended to smooth the VAR by a Harrell-Davis estimator (See Section 5.36 [harrell_davis_weight], page 93 for details). A weighted average of the scenarios around the confidence interval scenarios is calculated. The HD VAR shall reduce the Monte-Carlo error. Aggregation keys (e.g. currency, ID, asset class) can be freely specified to get a full portfolio risk drill down.

2.6 Reporting

After the aggregation of instrument data to fund level, a risk report is generated. The report contains VAR on position level, the riskiest instruments and positions per fund, as well as the diversified (with observed correlation) and undiversified (correlation set to 1) VAR and ES on fund level. Moreover, stress test results per fund, profit and loss distributions and histograms on both the one day and 250 day time horizon are generated (See Section 3.4 [Output files], page 25 for examples)

2.7 Graphical user interface

Octarisk's graphical user interface (GUI) allows for full user interaction. Based on an existing session, where instruments, market data, risk factors, portfolios and stress tests are defined, instrument properties can be changed and portfolio contributions can be altered. Moreover, a full insight into single instruments is possible, where e.g. sensitivities like key rate durations or option's Greeks can be investigated. It is also possible to extract scenario values. Once instrument attributes or position sizes are modified, a full valuation of the selected instruments and a reaggregation of the total portfolio takes place to examine impacts on all risk figures. It is also possible to add instruments as new positions to the selected portfolio. In this case, the new position has a position size of zero. For removing positions, simply set the size to zero. The following image gives an overview of the GUI and highlights all interaction possibilities:


Add instruments to selected portfolio
Show and edit all instrument attributes
Show instrument base value, VaR and stress results
Change position sizes and reaggregate on the fly
Show portfolio base value, stress values and VaR
Show position contribution to portfolio VaR (DecompVaR)
Automatic revaluation and reaggregation
Octarisk Experimental GUI 0.1
Instrument Section
Portfolio Section
Add Position ->
Recalc
Aggregate
Exit
Instrument PnL Distribution
Portfolio PnL Distribution
Profit and Loss (EUR)
MonteCarlo Scenario
VaR: -6.89 EUR
-0.7 %
VaR: -34772 EUR
-25.8 %
(c) 2017 schinzilord@octarisk.com

Valuation Date: 30-Sep-2016

Available Instruments:	A1YC04	
Instrument Attribute: eff_duration	Attribute Value: 2.2729	
Base Value:	1053.76970415	EUR
Value at Risk (250d,0.995):	-6.89343692	EUR
Value at Risk (250d,0.995):	-0.6542	%
Stresstest Name: IR+100bp	Stress Profit or Loss: -23.67568571	EUR

Available Portfolios:	FUND_AAA	
Position ID: A1YC04	Position Size: 10	
Portfolio Base Value:	134558.1137	EUR
Value at Risk (250d,0.995):	-34772.2796	EUR
Value at Risk (250d,0.995):	-25.8418	%
Position Decomp VaR:	18.9794	EUR
Stresstest Name: IR+100bp	Stress Profit or Loss: -2722.1029	EUR

3 Developer guide

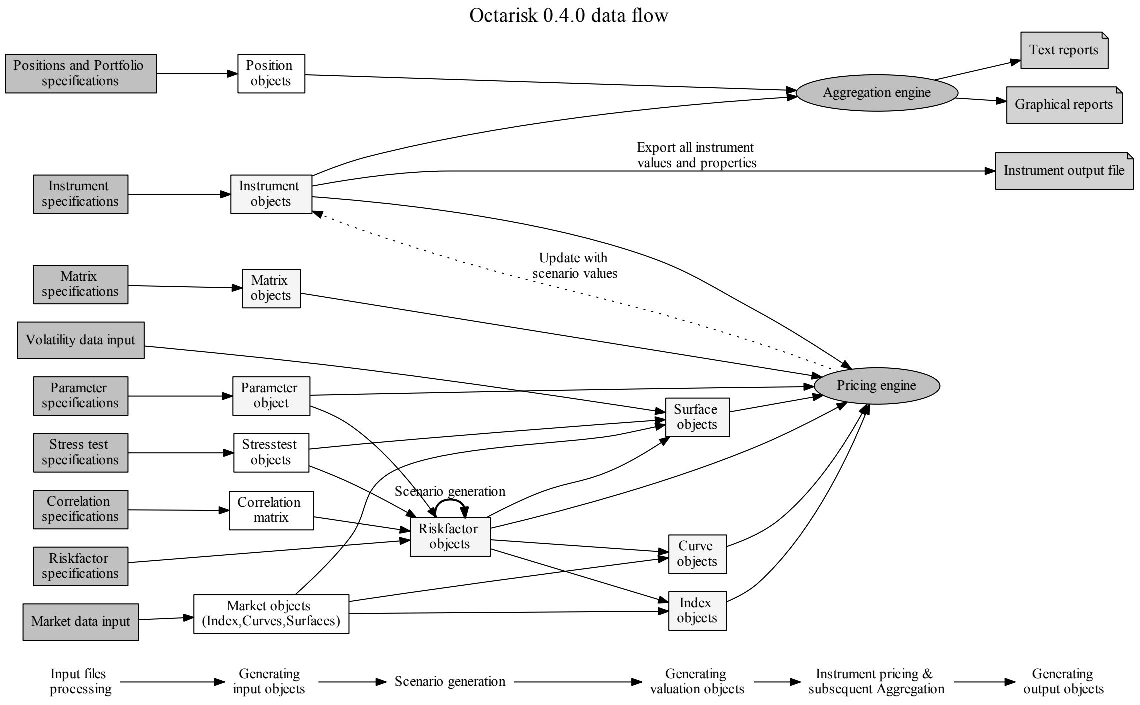
This chapter describes the actual implementation of the project. All calculation steps and the input and output files will be described in detail as well as examples are provided.

3.1 Implementation concept

A lightweight implementation concept was chosen for OCTARISK. One script is responsible for the complete work flow from input file parsing to aggregation and reporting. This script calls subfunctions to parse input data and construct objects, generate scenario dependent input values and call appropriate pricing functions. To assist developers, script are introduced for automatic generation of class diagrams (function print_class2dot) and function dependencies (get_dependencies). Moreover, all function and class descriptions, which are contained in the source files, are automatically extracted and appended to this document (see functions get_documentation and get_documentation_classes for details). A command line help for all functions (help functionname) and classes (Classname.help) is possible. To get performance insights, use the profiler (function profiler_analysis) to conveniently show detailled information.

3.1.1 Process overview

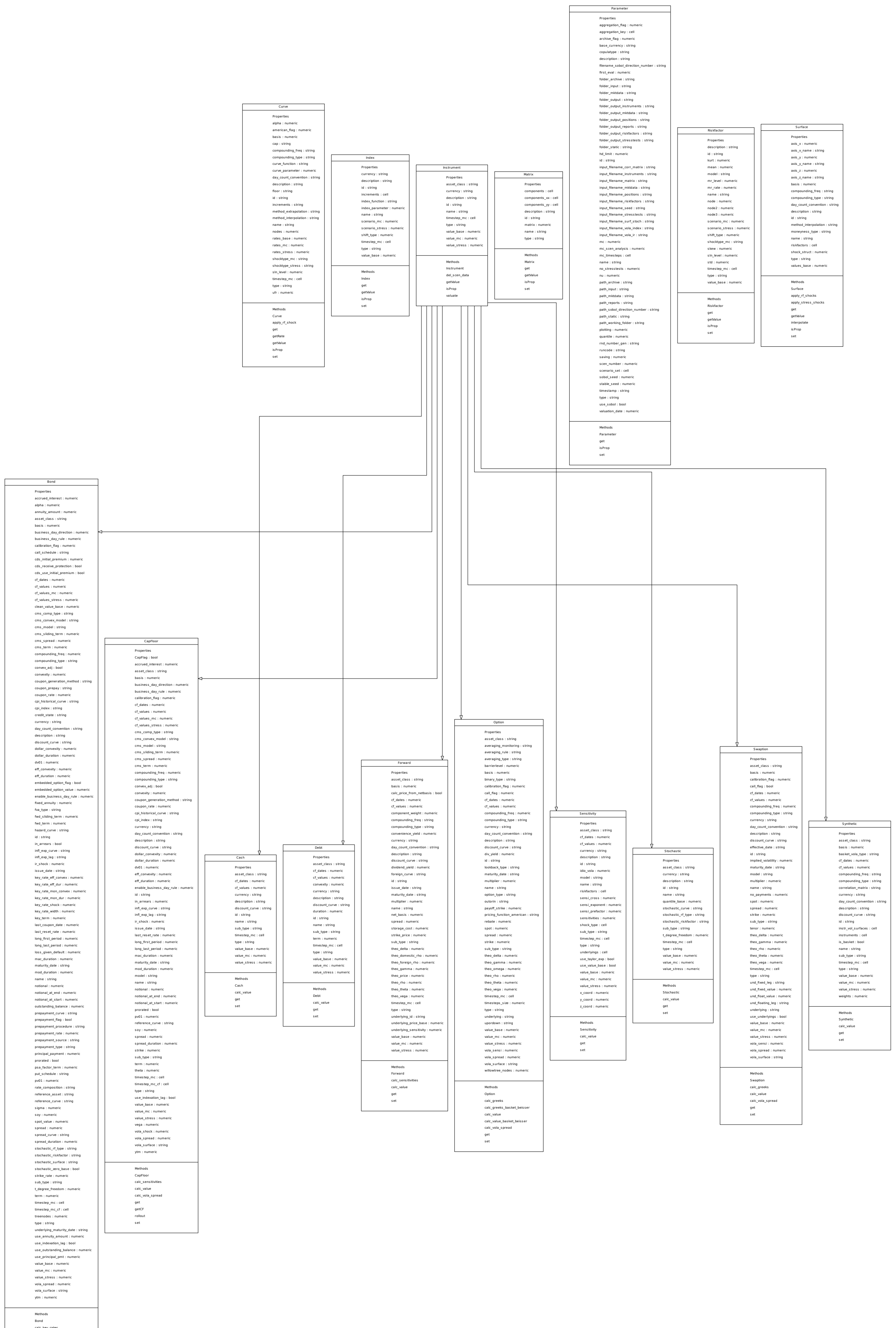
The following process summarizes the complete work flow:



Input files (See Section 3.3 [Input files], page 18) are parsed into structures, matrizes and objects. After parsing of these input files, Monte-Carlo and stress test scenarios are generated taking into account the correlation matrix and marginal risk factor distributions as well as custom stress test scenario configurations. The scenario dependent shocks are then stored for each risk factor object. The scenario shocks are then applied to all market data objects (mainly indizes, curves and exchange rates) which have attached risk factors. Scenario dependent values for each market data object are calculated by taken into account the scenario dependent shock, the market data base value and the risk factor stochastic model. In a next step instrument pricing takes place. For each instrument object the product type dependent calculation rule is triggered. If the sensitivity approach is chosen, scenario shocks (delta values) are applied to the instrument base value. In case of a full valuation approach, the instrument is priced with specified pricing engines taking into account all scenario dependent market object values. A mark-to-market procedure is applied, which results in equal theoretical and market base values (by setting an appropriate spread over yield or volatility spread). After all instruments have been priced, the aggregation starts for all portfolios and their positions. Final results are printed in graphical and text based reports reflecting the market risk measures.

3.1.2 Class diagram

OCTARISK was set up in an object oriented programming style for all objects like instruments, risk factors, curves, indizes and surfaces. Inside the methods of the aforementioned classes, pricing or interpolation functions are called. The following class diagram gives an overview of all classes:



See the class documentation in the next chapter for further information.

3.2 Implementation workflow

3.2.1 Overview

The following enumeration gives an overview of the main script *octarisk.m* (See Section 5.53 [octarisk], page 98 for details):

1. DEFINITION OF VARIABLES
    1. Parse parameter file
    2. read in general variables
    3. read in VAR specific variables
2. INPUT
    1. Processing Instruments data
    2. Processing Riskfactor data
    3. Processing Positions and Portfolio data
    4. Processing Stresstest data
    5. Processing Market data
3. CALCULATION
    1. Model Riskfactor Scenario Generation
        - Load input correlation matrix
        - Get distribution parameters from riskfactors
        - call MC scenario generations
    2. Monte Carlo Riskfactor Simulation for all timesteps
    3. Take risk factor stress scenarios from stressdefinition
    4. Process yield curves and volatility surface
    5. Update market data objects with risk factor shocks
    6. Full Valuation of all Instruments in all MC and stress scenarios
    7. Portfolio Aggregation
        - loop over all portfolios / positions
        - VaR Calculation
            - sort arrays
            - Get Value of confidence scenario
            - make vector with Harrel-Davis Weights
        - Calculate Expected Shortfall
    8. Print Report including position VaRs
    9. Plotting
4. HELPER FUNCTIONS

3.3 Input files

In the following, all required input files are introduced. The basic file format for instruments, positions, stresstests, risk factors and market data objects is comma separated with a variable number of header attributes. Each of these input files is further split into different sub types. Each sub type has his own header, which is directly followed by all entries belonging to this sub type. Each Header line is introduced by the string **Header** and without any space followed by the **SUBTYPE** in capital letters (e.g. **HeaderFRB** for fixed rate bonds of the instrument class). Each header attribute contains the name of the header and the type of the data: **NameTYPE**. There exist exactly four different attribut types:

- *CHAR*: any character combination without commas. For definition of lists (e.g. several riskfactors and weights), also type CHAR is used. The list entries are separated by | (pipe symbol): 365|730|1095|3650|7300 can be used for a definition of curve nodes.
- *DATE*: date in the format *DD-MMM-YYYY* (e.g. 29-Apr-2016) for the use of maturity dates or issue dates.
- *BOOL*: boolean variable. Either *1* or *0* or *TRUE* or *FALSE*
- *NMBR*: numeric variable (can also be complex). Can be an integer or float with double precision.

Empty attribute values (e.g. *ValueA,,1234*) are ignored during parsing of the input files.

3.3.1 Risk factors

The risk factors input file contains all risk factors which are modeled by stochastic processes. The shocks of these risk factors are then used as input to the calculation of the scenario dependent index, curve, volatility and instrument values which have these risk factors as attached risk drivers.
The columns of the risk factors file consist of the following entries:

- *HeaderRISKFACTOR*: Introduction of risk factors
- *nameCHAR*: Name of the riskfactors, this follows the convention RF_TYPE_XYZ (string).
- *idCHAR*: Unique ID of the risk factors. To keep it simple, just take name (string).
- *typeCHAR*: Risk factor types follow typical asset class conventions (string). These types are explained in Section 2.1 [Introducing market risk], page 4.
- *descriptionCHAR*: A short description of the risk factor. String maximum length of 255 characters.
- *modelCHAR*: Model ID of the underlying stochastic process (string). See section Models for further explanation.
- *meanNMBR*: Mean of the stochastic process used in scenario generation
- *stdNMBR*: Standard deviation (expected *annualized* volatility)
- *skewNMBR*: Skewness
- *kurtNMBR*: Kurtosis
- *value_baseNMBR*: Start value for the mean reversion (e.g. Ornstein-Uhlenbeck or square root diffusion) processes
- *mr_levelNMBR*: Mean reversion level

- *mr_rateNMBR*: Mean reversion rate
- *nodeNMBR*: Risk factor node of first dimension (e.g. term node of IR Curve or option term node of index vol surface)
- *node2NMBR*: Risk factor node of second dimension (e.g. moneyness of index vol surface or Underlying term of IR vol cube)
- *node3NMBR*: Risk factor node of second dimension (e.g. moneyness of IR vol cube)

An example of the input file is given:

```
HeaderRISKFACTOR,nameCHAR,idCHAR,typeCHAR,descriptionCHAR,modelCHAR,meanNMBR, ...
   ... stdNMBR,skewNMBR,kurtNMBR,value_baseNMBR,mr_levelNMBR,mr_rateNMBR,nodeNMBR,node2NMBR,no
Item,RF_EQ_DE,RF_EQ_DE,RF_EQ,Equity Germany,GBM,0,0.18,-0.5,5,9820,,,
Item,RF_EQ_EUR,RF_EQ_EUR,RF_EQ,Equity Euro,GBM,0,0.18,-0.5,5,,,,
Item,RF_FX_EURUSD,RF_FX_EURUSD,RF_FX,FX EUR USD,SRD,0,0.08,0,3,1.09,1.2,0.001,
Item,RF_VOLA_EQ_DE,RF_VOLA_EQ_DE,RF_VOLA,Impl Vol Ger,SRD,0,0.1,0,3,0.31,0.21,0.02,
Item,RF_IR_EUR_1Y,RF_IR_EUR_1Y,RF_IR,IR EUR 1year,BM,0,0.0011,0,3,0.0008,,,365
Item,RF_IR_EUR_10Y,RF_IR_EUR_10Y,RF_IR,IR EUR 10year,BM,0,0.0032,0,3,0.0026,,,3650
Item,RF_IR_EUR_20Y,RF_IR_EUR_20Y,RF_IR,IR EUR 20year,BM,0,0.006,0,3,0.015,,,7300
Item,RF_VOLA_COM_GOLD,RF_VOLA_COM_GOLD,RF_VOLA,VolGold,SRD,0,0.1,0,3,0.15,0.16,0.03,
Item,RF_SPPR_EUR_HY_5Y,RF_SPR_EUR_HY_5Y,RF_SPR,HY,BM,0,0.083,0.24,10,0.05,,,1825
```

3.3.2 Positions

The positions input file contains all portfolios and positions. Positions must point to instruments which are defined in the instruments input file. Both portfolios and positions are stored to structures. Thus, additional columns can be appended, which could then be used as positional attributes.
The columns of the portfolio contain the following characteristics :

- *HeaderPORTFOLIO*: Indicating a new header specifying a portfolio
- *idCHAR*: Unique ID of the portfolio. Used in the filename of the report
- *nameCHAR*: Portfolio name
- *descriptionCHAR*: Description of the portfolio (optional)
- *HeaderPOSITION*: Indicating a new header specifying a position
- *port_idCHAR*: ID of the portfolio, where the position belongs to. Must be valid portfolio ID.
- *idCHAR*: ID of the instrument
- *quantityNMBR*: quantity of the instrument in the portfolio

Example definitions for some positions (positive quantity: long position, negative quantity: short position) in two portfolios:

```
HeaderPORTFOLIO,idCHAR,nameCHAR,descriptionCHAR
Item,FUND_AAA,Global Diversified,Global diversified test portfolio
Item,FUND_BBB,Global Derivatives,Global derivatives test portfolio
HeaderPOSITION,port_idCHAR,idCHAR,quantityNMBR
Item,FUND_AAA,A0RFFT,65
Item,FUND_AAA,A1JB4Q,179
Item,FUND_AAA,A1J7CK,135
Item,FUND_AAA,A1YC04,20
Item,FUND_AAA,BTCOIN,1.98
Item,FUND_AAA,ODAXC20160318,-0.01
Item,FUND_AAA,EQFORW01,1
Item,FUND_AAA,SYNTH01,0.1
Item,FUND_AAA,CASH_EUR,1000
Item,FUND_BBB,A0LGQL,723
Item,FUND_BBB,ODAXC20160318,-0.1
Item,FUND_BBB,EQFORW01,1
Item,FUND_BBB,SYNTH01,1
Item,FUND_BBB,CASH_EUR,1000
```

3.3.3 Instruments

The instruments input file contains the specifications of all instruments which are priced during instrument valuation. The instrument universe is split into different product types. Each of the product type has his own file header, reflecting the huge differences in instrument specification.
The basic header attributes, which all instruments have in common, are given. For detailed information of additional columns see the class diagram.

- *HeaderXXYYZ*: Product type specific header attributes, e.g. CASH, FRB, FRN, SENSI, SYNTH, OPT, FWD, SWAPT, ...
- *nameCHAR*: Name of the instrument which will be used for the reports.
- *idCHAR*: Unique ID of the instrument. Used as a reference in position input file. Is used as default aggregation key for reporting.
- *value_baseNMBR*: Actual market value of the instrument.
- *typeCHAR*: Instrument type specifying the sub type inside the specified instrument class (e.g. EQFWD for equity forward or OPT_EUR_C for an European call option).
- *descriptionCHAR*: A short description of the instrument. Maximum length of 255 characters.
- *currencyCHAR*: Currency of the instrument. If no appropriate currency risk factor is defined, they are mapped to EUR. Is used as default aggregation key for reporting.
- *asset_classCHAR*: Instrument asset class. Is used as default aggregation key for reporting.

Example definitions for cash instruments:

```
HeaderCASH,nameCHAR,idCHAR,value_baseNMBR,typeCHAR,descriptionCHAR,currencyCHAR,asset_classCHA
Item,Cash Account EUR,CASH_EUR,1,CASH,EUR Cash Account,EUR,cash
```

3.3.4 Stress tests

The stress test input file contains the definition of all stress test. Each stress test describes the behavior of one or more risk factor in a particular scenario. The risk factor shock values are directly applied to all risk factor IDs which are selected through the regular expression. The columns of the stress test file consists of following entries:

- *HeaderSTRESSTESTS*: indicates stress tests
- *idCHAR*: Unique ID of the stresstest.
- *nameCHAR*: Name of the stresstest, used in reporting.
- *objectCHAR*: Object ID which will be shocked (curve, index, surface, not a risk factor)
- *objecttypeCHAR*: type of object to be shocked (e.g. curve or index)
- *shocktypeCHAR*: type of shock (e.g. absolute or relative shock or specific value used in stress scenario)
- *termCHAR*: term of curve be shocked (several terms are separated by pipes)
- *axis_xCHAR*: x coordinate of object (e.g. tenor of IR volatility cube)
- *axis_yCHAR*: y coordinate of object (e.g. term of IR volatility cube)
- *axis_zCHAR*: z coordinate of object (e.g. moneyness of IR volatility cube)
- *method_interpolationCHAR*: interpolation method of shocks (only applicable to curves and surface objects)
- *shockvalueCHAR*: shock value applied to objects (several shocks to all terms are separated by pipes)

An example for possible stress test definitions are given:

```
#Stresstests Specifications
HeaderSTRESSTESTS,idCHAR,nameCHAR,objectCHAR,objecttypeCHAR,shocktypeCHAR, ...
  ... termCHAR,axis_xCHAR,axis_yCHAR,axis_zCHAR,method_interpolationCHAR,shockvalueCHAR
Item,STRESS01,IR-100bp,IR_EUR,curve,absolute,365,,,,linear,-0.01
Item,STRESS01,IR-100bp,IR_USD,curve,absolute,365,,,,linear,-0.01
Item,STRESS02,IR+100bp,IR_EUR,curve,absolute,365,,,,linear,+0.01
Item,STRESS02,IR+100bp,IR_USD,curve,absolute,365,,,,linear,+0.01
Item,STRESS03,SPREAD-100bp,SPREAD_EUR_FIN_A,curve,absolute,365,,,,linear,-0.01
Item,STRESS04,SPREAD+100bp,SPREAD_EUR_FIN_A,curve,absolute,365,,,,linear,+0.01
Item,STRESS05,INFL-100bp,INFL_EXP_CURVE,curve,absolute,365,,,,linear,-0.01
Item,STRESS06,INFL+100bp,INFL_EXP_CURVE,curve,absolute,365,,,,linear,+0.01
Item,STRESS07,EQ-30Pct,EQ_DE,index,relative,,,,,linear,0.7
Item,STRESS08,EQ+30Pc,EQ_DE,index,relative,,,,,linear,1.3
Item,STRESS09,ASIANFLU,VOLA_IR_EUR_0.0002,surface,value,,365|3650,365|3650,,,0.015|0.025;0.01|
Item,STRESS10,IR_EURTwistPos,IR_EUR,curve,absolute,365|3650|7300,,,,linear,-0.01|0.01|0.03
```

New stress tests can be easily appended to the specification file.

3.3.5 Volatility surface

For all options and swaptions, the implied volatility is necessary to calculate the derivative theoretical value. In order to feed the implied volatility into the system, a term / moneyness surface has to be specified for index instruments and a underlying tenor / term / moneyness for IR instruments in a separate file. For all underlying index risk factors the impl. volatility data file has to be named like *vol_index_RF_XX_YY.dat* and *vol_ir_RF_XX_YY.dat* for all interest rate risk factor. The risk factor ID will be used to automatically identify the appropriate file. The structure of the file is a linearized version of the volatility surface (for term / moneyness instruments):

```
% term moneyness implied_vola
30 1.2 0.4
30 1.0 0.35
30 0.8 0.3
90 1.2 0.5
90 1.0 0.45
90 0.8 0.4
```

and the volatility cube for tenor term moneyness for interest rate instruments:

```
#tenor term moneyness implied_vola
365.00000 365.00000 1.00000 0.50000
365.00000 730.00000 1.00000 0.40000
365.00000 1095.00000 1.00000 0.30000
730.00000 730.00000 1.00000 0.35000
730.00000 365.00000 1.00000 0.30000
730.00000 1095.00000 1.00000 0.35000
365.00000 365.00000 1.25 0.50000
365.00000 730.00000 1.25 0.50000
365.00000 1095.00000 1.25 0.30000
730.00000 730.00000 1.25 0.35000
730.00000 365.00000 1.25 0.30000
730.00000 1095.00000 1.25 0.35000
```

All tenor and term values are given in days from valuation date.

During the full valuation approach the moneyness is a function of the underlying risk factor spot price over rate and the constant strike price over rate. A linear interpolation for the moneyness and a nearest neighbour mapping for tenors terms and constant extrapolation will be performed to calculate the new scenario dependent implied volatililty. Since the at-the-money volatility is itself a risk factor (modeled as a factor), the interpolated volatility will be adjusted by this scenario dependent factor. This process combines the conservation of volatility surface or cubes shape with the single factor stochastic modeling of the at-the-money volatility. Furthermore, it is also possible to generate a file with just one constant volatility.

3.3.6 Marketdata objects file

Market data objects store all relevant objects for full valuation instrument pricing. These objects can be interest and spread curves, aggregated curves (sum of curves), market indizes, exchange rates, volatility surfaces and cubes as well as call and put schedules. The base values of these objects can then be shocked by scenario dependent values, which were calculated for the attached risk factors. The market data objects are specified in a separate file following the same conventions as the instruments input file:

- *HeaderCURVE*: market object specific Header classification for curves
- *idCHAR*: Unique ID of the market curve. Note: if it is required that the curve will be shocked during stress tests or in MC scenarios, the ID of the market curve must have an attached risk factor starting with *RF_* followed by the market curve ID. Otherwise, an automatic mapping is not possible
- *nameCHAR*: Name of the market curve

- *typeCHAR*: Type of the curve (e.g. Spread Curve or Discount Curve)
- *descriptionCHAR*: Description of the object
- *method_interpolationCHAR*: Interpolation method for the market curve (e.g. linear or monotone-convex). This interpolation method can be different to the interpolation method from the attached risk factor. For all nodes of the market curve the relative of absolute scenario dependent shock values (Derived from the attached risk factor curve) is interpolated and then applied to the market curve node
- *nodesCHAR*: Pipe separated nodes of the market curve in days from the valuation date
- *rates_baseCHAR*: Pipe separated rates of the market curve. One rate needed for each specified node
- *compounding_typeCHAR*: Compounding type of curve (defaults to continuous)
- *compounding_freqCHAR*: Compounding frequency of curve (defaults to annual, only relevant if compounding type equals discrete. Otherwise, value will be neglected)
- *day_count_conventionCHAR*: Day count convention of curve (defaults to act/365)
- *floorNMBR*: floor rate for base and stress rates. The floor is applied when rates are set or, if there are already specified rates, the new floor is applied to old rates
- *capNMBR*: cap rate for base and stress rates
- *american_flagBOOL*: boolean flag for american call and put option schedule
- *HeaderAGGREGATEDCURVE*: market object specific Header classification for aggregated curves
- *idCHAR*: Unique ID of the aggregated curve. No stress or MC shocks are applied directly on the aggregated curve. The underlying curves have to be shocked directly.
- *nameCHAR*: Name of the aggregated market curve.
- *typeCHAR*: Type of the curve (Aggregated Curve)
- *descriptionCHAR*: Description of the object
- *method_interpolationCHAR*: Interpolation method for the market curve (e.g. linear or monotone-convex). This interpolation method can be different to the interpolation method from the attached risk factor. For all nodes of the market curve the relative of absolute scenario dependent shock values (Derived from the attached risk factor curve) is interpolated and then applied to the market curve node.
- *nodesCHAR*: Pipe separated nodes of the market curve in days from the valuation date.
- *incrementsCHAR*: Pipe separated ID of underlying curve increments. Specify all curve increments which are then used in curve stacking. An incremental spread model can be specified with this procedure
- *compounding_typeCHAR*: Compounding type of curve (defaults to continuous)
- *compounding_freqCHAR*: Compounding frequency of curve (defaults to annual, only relevant if compounding type equals discrete. Otherwise, value will be neglected)
- *day_count_conventionCHAR*: Day count convention of curve (defaults to act/365)
- *floorNMBR*: floor rate for base and stress rates. The floor is applied when rates are set or, if there are already specified rates, the new floor is applied to old rates

- *capNMBR*: cap rate for base and stress rates

Please note: If the curve increments of the aggregated curve has differenct settings for compounding type, frequency and day count convention, an automatic conversion will be performed.

- *HeaderINDEX*: market object specific Header classification for indizes and exchange rates. Basically, all market objects with exactly one scenario dependent value can be stored here
- *idCHAR*: Unique ID of the market index. Note: if it is required that the market index will be shocked during stress tests or in MC scenarios, the ID of the market curve must have an attached risk factor starting with *RF_* followed by the market curve ID. Otherwise, an automatic mapping is not possible
- *nameCHAR*: Name of the market index
- *typeCHAR*: Type of the index (e.g. Equity Index, Commodity Index, Exchange Rate)
- *currencyCHAR*: Currency of the object
- *descriptionCHAR*: Description of the object
- *value_baseNMBR*: Market value of the index
- *HeaderSURFACE*: market object specific Header classification for volatility surfaces and cubes.
- *idCHAR*: Unique ID of the market volatility surface.
- *nameCHAR*: Name of the volatility surface
- *typeCHAR*: Type of the surface (e.g. INDEX, IR)
- *descriptionCHAR*: Description of the object
- *moneyness_typeCHAR*: Moneyness type of volatility surface. Can be K/S for relative moneyness and K-S for absolute moneyness
- *method_interpolationCHAR*: interpolation method for volatility surface or cube (e.g. nearest or (bi- or tri-)linear)
- *riskfactorsCHAR*: Pipe separated string with ids of underlying risk factors

```
HeaderCURVE,idCHAR,nameCHAR,typeCHAR,descriptionCHAR,method_interpolationCHAR,nodesCHAR,rates_
Curve,IR_EUR,IR_EUR,Discount Curve,EUR-SWAP Curve,monotone-convex,365|1095|1825|3650|7300|1095
0.00519251|-0.00508595|-0.00367762|0.00185694|0.00776253|0.00999986|0.00123,,,,0.0001,0.1█
Curve,IR_USD,IR_USD,Discount Curve,USD-SWAP Curve,monotone-convex,365|1095|1825|3650|7300|1095
Curve,SPREAD_EUR_HY,SPREAD_EUR_HY,Spread Curve,SPREAD_EUR_High Yield Curve,linear,1825,0.02,di
HeaderAGGREGATEDCURVE,idCHAR,nameCHAR,typeCHAR,descriptionCHAR,method_interpolationCHAR,nodesC
Curve,AGGR_SPREAD_USD_BBB,AGGR_SPREAD_USD_BBB,Aggregated Curve,Aggregated Spread Curve USD BBB
Curve,AGGR_EUR_FIN_BBB,AGGR_EUR_FIN_BBB,Aggregated Curve,Aggregated Spread Curve EUR USD,monot
convex,30|91|365|730|1095,IR_EUR|SPREAD_EUR_HY,simple,annual,act/365
HeaderINDEX,idCHAR,nameCHAR,typeCHAR,currencyCHAR,descriptionCHAR,value_baseNMBR█
Index,EQ_DE,DAX30,Equity Index,EUR,DAX Equity Index German Blue Chips,9820
Index,COM_GOLD,Gold,Commodity Index,USD,Gold 1 Ounce USD,1073
Index,FX_EURUSD,EUR_USD,Exchange Rate,EUR,EUR USD Exchange rate,1.08
HeaderSURFACE,idCHAR,nameCHAR,typeCHAR,descriptionCHAR,moneyness_typeCHAR,method_interpolation
Surface,VOLA_IR_EUR,VOLA_IR_EUR,IR,Test description,K-S,linear,RF_VOLA_IR_EUR_730_365|RF_VOLA_
```

These two market objects (indizes and curves) reflect market objects with either just one scenario dependent value (a scalar like for indizes) or a certain constant number of dependent values per scenario (like a curve). For these objects it is possible to specify an arbitrary number of risk factors in order to get the most granular scenario dependent behaviour. For convenience reasons and because of the increased memory consumption, indizes and curves are up to now the only possible market objects. For volatility surface or cubes it is only possible to specify risk factors with just one scenario dependent value (at-the-money volatility). The market object (the volatility cube) is then offsetted by a constant value in each scenario, thus preserving the base value volatility smile. It is therefore not possible to model a scenario dependent smile.

3.3.7 Correlation matrix

The correlation file contains the correlations between risk factor in a linearized format. Only the lower (or upper) triangular matrix including the diagonal values has to be set. The first line is a header describing the three columns, but there is no possibility to . The order of the risk factors in the correlation file can be arbitrary. Only the subset of all risk factors, which are contained in the correlation file, are used during MC scenario generation. The risk factors specified in the correlation file must be a subset of risk factors specified in the risk factors file. During parsing of the file validation checks are performed in order to make sure, that all correlation pairs have been set. Otherwise an exception will be raised. After parsing, the correlation file is stored in Octaves built-in matrix format. The correlation matrix undergoes then a Cholesky decomposition to generate correlated random variables with a copula approach. Based on these correlated randum numbers, the four distribution moments (mean, standard deviation, skewness, kurtosis), which are given in the risk factor input file, are used to generate the marginal distributions for each risk factor based on the Pearson type I-VII distribution system. If the correlation matrix is not positive semi-definite, all negative eigenvalues are set to slightly positive numbers in an iterative approach, thus leading to positive semi-definiteness. Be aware, that the correlation settings may change. Further statistics on the correlation settings and on the statistical parameters of the marginal risk factor distributions may be automatically calculated if the appropriate flag is set in the settings. In this case, a correlation heat map shows the deviations of the final correlation settings compared to the input correlation settings. The correlation file contaings the following columns:

- *RF1CHAR*: ID of first risk factor
- *RF2CHAR*: Unique ID of second risk factor
- *CorrelationNMBR*: correlation between these two risk factors

An example is given:

```
RF1CHAR,RF2CHAR,CorrelationNMBR
RF_EQ_DE,RF_EQ_DE,1
RF_EQ_EUR,RF_EQ_DE,0.96479531
RF_EQ_EU,RF_EQ_DE,0.890684579999999
RF_EQ_NA,RF_EQ_DE,0.81541365
...
```

3.4 Output files

Overview of report summary and graphics.

3.4.1 VAR Report

For each fund, a VAR report is generated. The report shall give an overview of total risk measure, allow a break down to position level and estimate the diversification effect. An example for the report looks like:

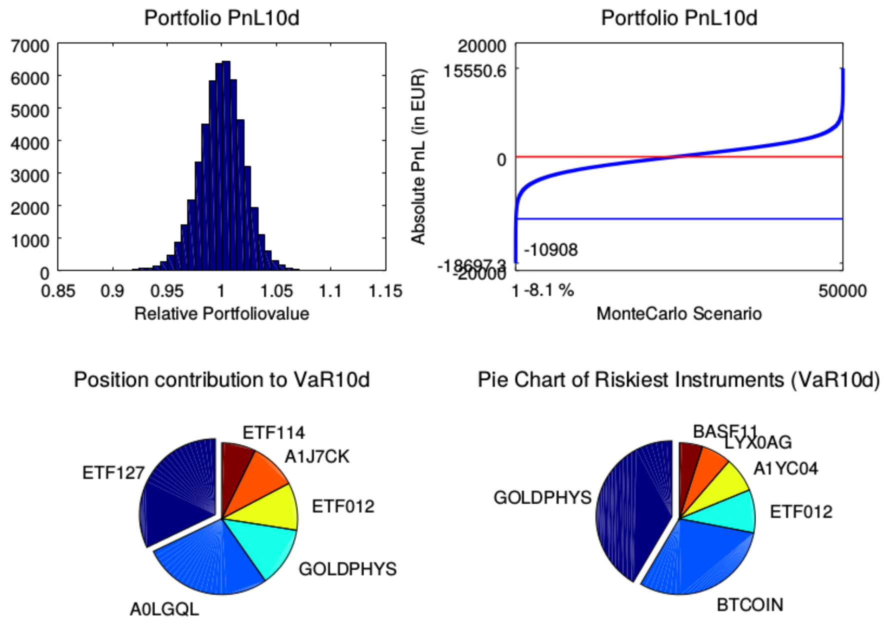
```
=== Value-At-Risk Report for Portfolio 1 ===
VaR calculated 99pct Confidence Intervall:
Number of Monte Carlo Scenarios: 500000
Confidence Scenarionummer: 5000
Valuation Date: 09-Oct-2015
VaR on Positional Level:
|VaR 1D for Position   |iShares JPMorgan $ EM|A0RFFT| = | 618.16 EUR|
|VaR 250D for Position |iShares JPMorgan $ EM|A0RFFT| = | 1755.21 EUR|
...
=== Total Portfolio VaR ===
|Portfolio VaR   1D@99Pct|   |    -1.32%|
|Portfolio VaR   1D@99Pct|   |  1779.97 EUR|
|Portfolio VaR 250D@99Pct|   |   -19.34%|
|Portfolio VaR 250D@99Pct|   | 25983.52 EUR|

|Expected Shortfall 1D@99Pct|   |    -1.51%|
|Expected Shortfall 1D@99Pct|   |  2027.10 EUR|
|Expected Shortfall 250D@99Pct|   |   -21.58%|
|Expected Shortfall 250D@99Pct|   | 28996.72 EUR|
```

The unique field separator '`|`' allows efficient use of the data in LaTeX based report files.

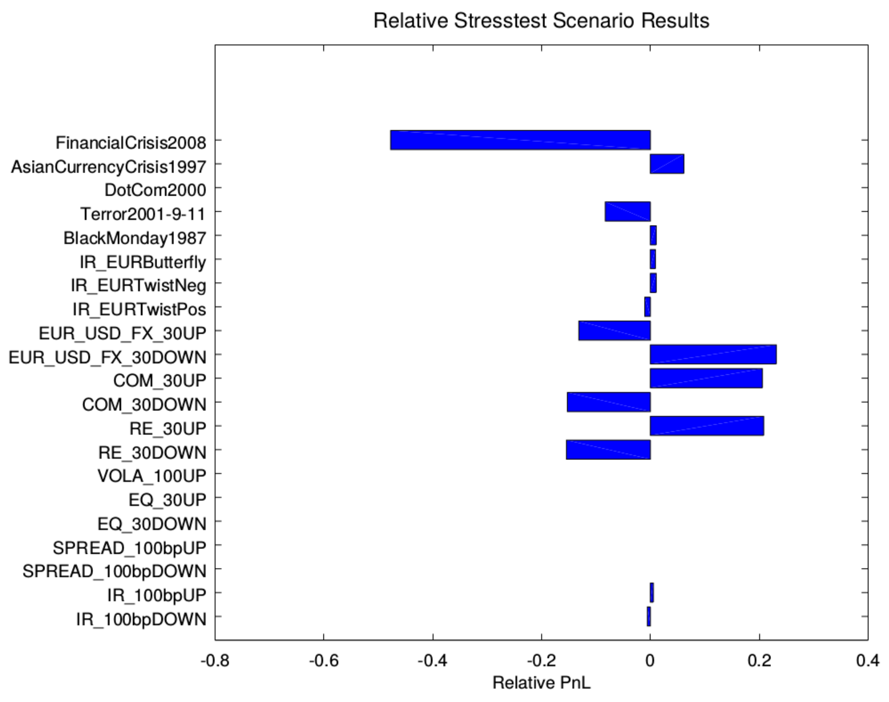
3.4.2 Overview images

Additionally to the printed report, overview images of the profit and loss distribution, both sorted and histogram as well as the riskiest instruments and positions are plotted.



Portfolio Value-at-Risk at the 99.9% confidence level on 10 day time horizon. There are shown four different results: On the top left corner a histogram of the profit and loss distribution is presented for the whole portfolio. On the x-axis, the relative portfolio value is given. On the top right corner the P'n'L distribution is shown as sorted absolute profits or losses for each of the MC scenarios. The red base line marks the base scenario. The blue line indicates the VAR scenario number, where 99.9% of all profits and losses are shown on the right side. Losses greater than the VAR occur in 0.1% of all cases. The two lower charts show the VAR contribution of the six riskiest positions (left chart) or instruments (right chart). The positional view is determined by weighing the instruments with their position size in the portfolio.

Moreover, stress test results are plotted in a bar chart, indicating the relative profit or loss of the fund in each stress test scenario:


Relative Stresstest Scenario Results
FinancialCrisis2008
AsianCurrencyCrisis1997
DotCom2000
Terror2001-9-11
BlackMonday1987
IR_EURButterfly
IR_EURTwistNeg
IR_EURTwistPos
EUR_USD_FX_30UP
EUR_USD_FX_30DOWN
COM_30UP
COM_30DOWN
RE_30UP
RE_30DOWN
VOLA_100UP
EQ_30UP
EQ_30DOWN
SPREAD_100bpUP
SPREAD_100bpDOWN
IR_100bpUP
IR_100bpDOWN
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
Relative PnL

4 Octave octarisk Classes

In the following sections you find the octarisk classes texinfo.

4.1 Instrument.help

`object = Instrument (name, id, description, type, currency, base_value, asset_class)` [Octarisk Class]

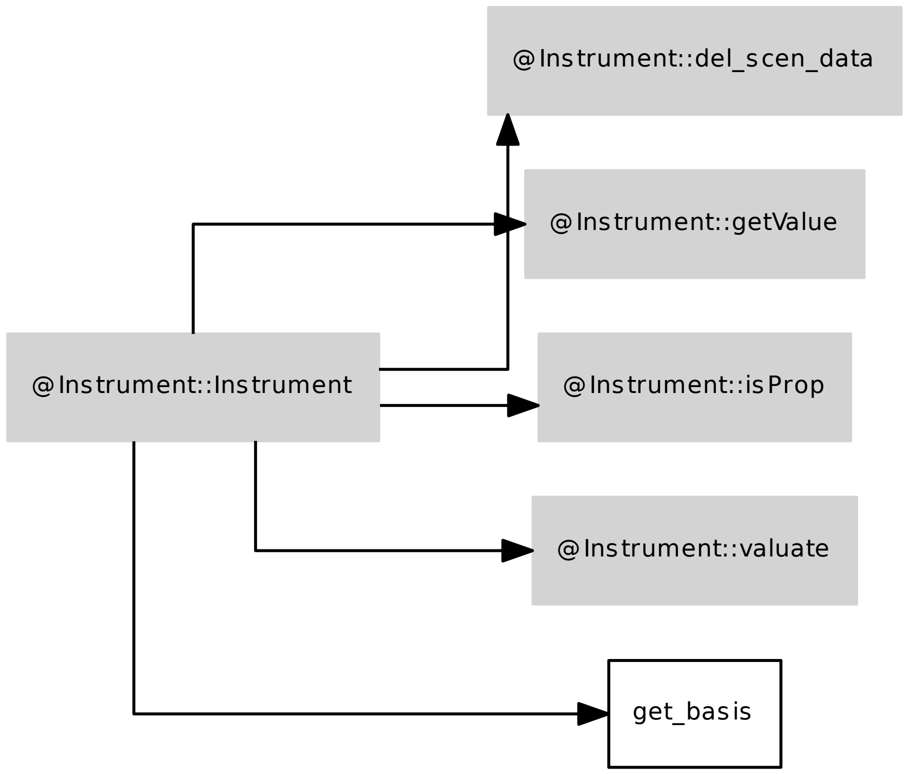
Superclass for all instrument objects.

- *name* (string): name of object
- *id* (string): id of object
- *description* (string): description of object
- *type* (string): instrument type in list [cash, bond, debt, forward, option, sensitivity, synthetic, capfloor, stochastic, swaption]
- *currency* (string): ISO code of currency
- *base_value* (float): actual base (spot) value of object
- *asset_class* (sring): instrument asset class

The constructor of the instrument class constructs an object with the following properties and inherits them to all sub classes:

- name: name of object
- id: id of object
- description: description of object
- value_base: actual base (spot) value of object
- currency: ISO code of currency
- asset_class: instrument asset class
- type: type of instrument class (Bond,Forward,...)
- value_stress: vector with values under stress scenarios
- value_mc: matrix with values under MC scenarios (values per timestep per column)
- timestep_mc: MC timestep per column (cell string)

Dependencies of class:



4.2 Matrix.help

`object` = *Matrix*(`id`) [Octarisk Class]
`object` = *Matrix*() [Octarisk Class]

Class for setting up Matrix objects.

This class contains all attributes and methods related to the following Matrix types:

- Correlation: specifies a symmetric correlation matrix.

In the following, all methods and attributes are explained and a code example is given.

Methods for Matrix object *obj*:

- Matrix(*id*) or Matrix(): Constructor of a Matrix object. *id* is optional and specifies id and name of new object.
- obj.set(*attribute*,*value*): Setter method. Provide pairs of attributes and values. Values are checked for format and constraints.
- obj.get(*attribute*): Getter method. Query the value of specified attribute.

- obj.getValue(*xx*,*yy*): Return matrix value for component on x-Axis *xx* and component on y-Axis *yy*. Component values are recognized as strings.
- Matrix.help(*format*,*returnflag*): show this message. Format can be [plain text, html or texinfo]. If empty, defaults to plain text. Returnflag is boolean: True returns documentation string, false (default) returns empty string. [static method]

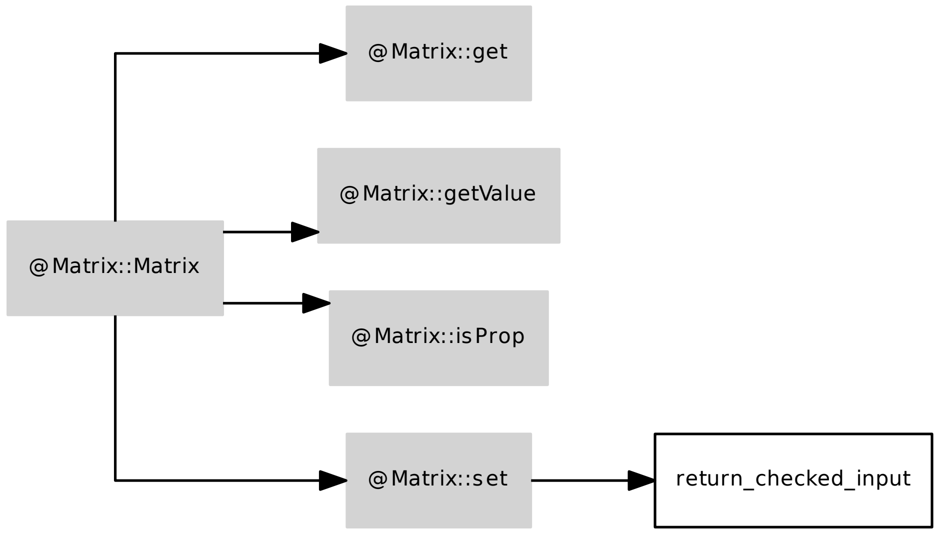
Attributes of Matrix objects:

- *id*: Matrix id. Has to be unique identifier. Default: empty string.
- *name*: Matrix name. Default: empty string.
- *description*: Matrix description. Default: empty string.
- *type*: Matrix type. Can be [Correlation]
- *components*: String cell specifying matrix components. For symmetric correlation matrizes x and y-axis components are equal.
- *matrix*: Matrix containing all elements. Has to be of dimension n x n, while n is length of *components* cell.
- *components_xx*: Set automatically while setting *components* cell
- *components_yy*: Set automatically while setting *components* cell

For illustration see the following example: A symmetric 3 x 3 correlation matrix is specified and one specific correlation for a set of components as well as the whole matrix is retrieved:

```
m = Matrix();
component_cell = cell;
component_cell(1) = 'INDEX_A';
component_cell(2) = 'INDEX_B';
component_cell(3) = 'INDEX_C';
m = m.set('id','BASKET_CORR','type','Correlation','components',component_cell);
m = m.set('matrix',[1.0,0.3,-0.2;0.3,1,0.1;-0.2,0.1,1]);
m.get('matrix')
corr_A_C = m.getValue('INDEX_A','INDEX_C')
```

Dependencies of class:



4.3 Curve.help

`object = Curve(id)` [Octarisk Class]
`object = Curve()` [Octarisk Class]

Class for setting up Curve objects.

This class contains all attributes and methods related to the following Curve types:

- Discount Curve: specify discount rates. Can be used as underlying for aggregated curves.
- Spread Curve: specify spread rates. Can be used as underlying for aggregated curves.
- Dummy Curve: used as default curve with no special meaning
- Aggregated Curve: stacked curve which combines underlying curves by aggregation functions (e.g. sum, factor)
- Prepayment Curve: specify prepayment rates per node (e.g. PSA curves).
- Call Schedule: curve used for Bonds with embedded call options
- Put Schedule: curve used for Bonds with embedded put options
- Hazard Curve: curve used for CDS. Gives default probability per cf date
- Historical Curve: curve with historical rates of index levels. Used for inflation linked or averaging instruments
- Inflation Expectation Curve: specify inflation expectation rates used for inflation linked instrument pricing

- Shock Curve: specify absolute or relative shocks per node, which can then be applied to other Curve types.

In the following, all methods and attributes are explained and a code example is given.

Methods for Curve object *obj*:

- Curve(*id*) or Curve(): Constructor of a Curve object. *id* is optional and specifies id and name of new object.
- obj.set(*attribute*,*value*): Setter method. Provide pairs of attributes and values. Values are checked for format and constraints.
- obj.get(*attribute*): Getter method. Query the value of specified attribute.
- obj.getRate(*scenario*,*node*): Return scenario curve values at given node (in days). Interpolation or Extrapolation is performed according to specified methods. *scenario* can be 'base', 'stress' or a certain MC timestep like '250d'.
- obj.getValue(*scenario*): Return all scenario curve values. *scenario* can be 'base', 'stress' or a certain MC timestep like '250d'.
- obj.apply_rf_shocks(*scenario*,*riskfactor_object*): Set shock curve values for *scenario* Scenario shocks from provided *riskfactor_object* are used
- obj.isProp(*attribute*): Return true, if attribute is a property of Curve class. Return false otherwise.
- Curve.help(*format*,*returnflag*): show this message. Format can be [plain text, html or texinfo]. If empty, defaults to plain text. Returnflag is boolean: True returns documentation string, false (default) returns empty string. [static method]

Attributes of Curve objects:

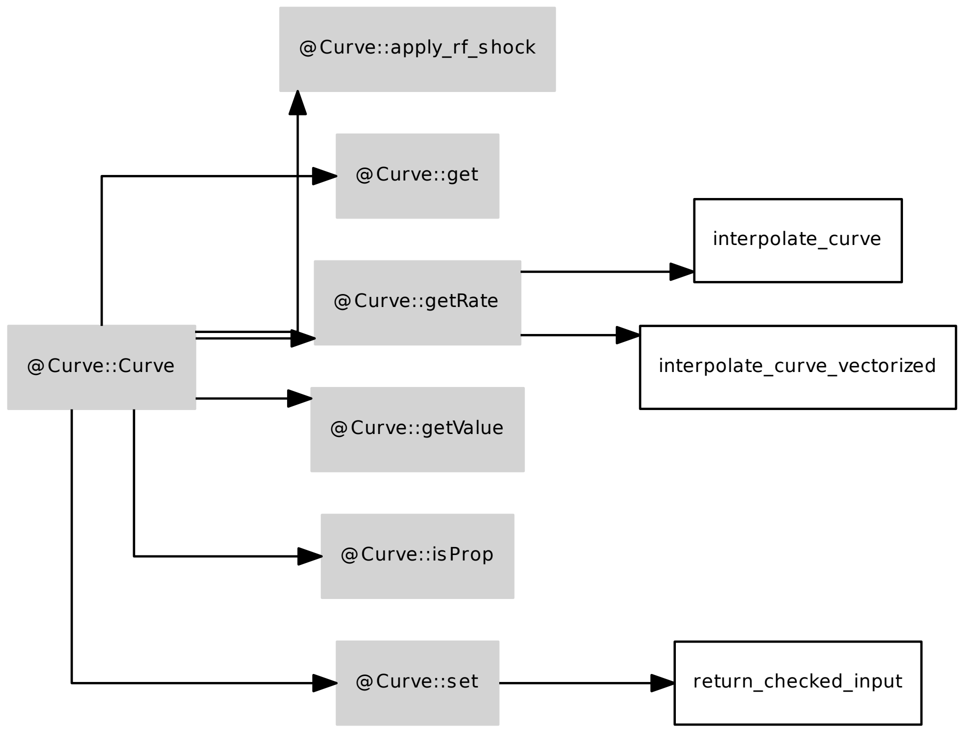
- *id*: Curve id. Has to be unique identifier. Default: empty string.
- *name*: Curve name. Default: empty string.
- *description*: Curve description. Default: empty string.
- *type*: Curve type. Can be [Discount Curve (default), Spread Curve, Dummy Curve, Aggregated Curve, Prepayment Curve, Call Schedule, Put Schedule, Historical Curve, Inflation Expectation Curve, Shock Curve]
- *day_count_convention*: Day count convention of curve. See 'help get_basis' for details. Default: 'act/365'\n
- *basis*: Basis belonging to day count convention. Value is set automatically.
- *compounding_type*: Compounding type. Can be continuous, discrete or simple. Default: 'cont'
- *compounding_freq*: Compounding frequency used for discrete compounding. Can be [daily, weekly, monthly, quarterly, semi-annual, annual]. Default: 'annual'
- *curve_function*: Type Aggregated Curve only: specifies how to aggregated curves, which are specified in attribute increments. Can be [sum, product, divide, factor]. [sum, product, divide] specifies mathematical operation applied on all curve increments. [factor] allows only one increment and uses *curve_parameter* for multiplication. Default: 'sum'
- *curve_parameter*: Type Aggregated Curve only: used as multiplication parameter for factor *curve_function*.

- *increments*: Type Aggregated Curve only: List of IDs of all underlying curves. Use *curve_function* to specify how to aggregated curves.
- *method_extrapolation*: Extrapolation method. Can be 'constant' (default) or 'linear'.
- *method_interpolation*: Interpolation method. See 'help interpolate_curve' for details. Default: 'linear'.
- *ufr*: Smith-Wilson Ultimate Forward Rate. Used for Smith-Wilson interpolation and extrapolation. Defaults to 0.042.
- *alpha*: Smith-Wilson Reversion parameter. Used for Smith-Wilson interpolation and extrapolation. Defaults to 0.19.
- *cap*: Cap rate. Cap rate is enforced on all set rates. Set to empty string for no cap rate. Default: empty string.
- *floor*: Floor rate. Floor rate is enforced on all existing and future rates. Set to empty string for no floor rate. Default: empty string.
- *nodes*: Vector with curve nodes.
- *rates_base*: Vector with curve rates. Has to be of same column size as *nodes*.
- *rates_mc*: Matrix with curve rates. Has to be of same column size as *nodes*. Columns: nodes, Lines: scenarios. MC rates for several MC timesteps are stored in layers.
- *rates_stress*: Matrix with curve rates. Has to be of same as *nodes*. Columns correspond to nodes, lines correspond to scenarios.
- *timestep_mc*: String Cell array with MC timesteps. Automatically appended if values for new timesteps are set.
- *shocktype_mc*: Specify how to apply risk factor shocks in Monte Carlo scenarios and for method apply_rf_shocks. Can be [absolute, relative, sln_relative]. Automatically set by scripts. Default: absolute
- *shocktype_stress*: Specify Stress risk factor shocks for method apply_rf_shocks. Can be [absolute, relative] by stree scenario configuration.
- *sln_level*: Vector with term specific shift level for risk factors modelled with shifted log-normal model. Automatically set by script during curve setup.
- *american_flag*: Flag for American (true) or European (false) call feature on bonds. Valid only if Curve type is call or put schedule. Default: false.

For illustration see the following example: A discount curve c is specified. A shock curve s provides absolute shocks for stress and relative shocks for MC scenarios, which are linearly interpolated and subsequently applied to the discount curve c. In the end, stress and MC discount rates are interpolated for given nodes with method getRate, while all curve rates are extracted with getValue.

```
c = Curve();
c = c.set('id','Discount_Curve','type','Discount Curve', ...
'nodes',[365,3650,7300],'rates_base',[0.01,0.02,0.04], ...
'method_interpolation','linear','compounding_type','continuous', ...
'day_count_convention','act/365');
s = Curve();
s = s.set('id','IR Shock','type','Shock Curve','nodes',[365,7300], ...
'rates_base',[],'rates_stress',[0.01,0.01;0.02,0.02;-0.01,-0.01;-0.01,0.01], ...
'rates_mc',[1.1,1.1;0.9,0.9;1.2,0.8;0.8,1.2],'timestep_mc','250d', ...
'method_interpolation','linear','shocktype_stress','absolute', ...
'shocktype_mc','relative');
c = c.apply_rf_shock('stress',s);
c = c.apply_rf_shock('250d',s);
c_base = c.getRate('base',1825)
c_rate_stress = c.getRate('stress',1825)
c_rate_250d = c.getRate('250d',1825)
c_rates_250d = c.getValue('250d')
```

Dependencies of class:



4.4 Forward.help

`object` = *Forward*(`id`) [Octarisk Class]
`object` = *Forward*() [Octarisk Class]

Class for setting up Forward and Future objects. Possible underlyings are bonds, equities and FX rates. Therefore the following Forward types are introduced:

- Bond: Forward on bond underlyings. Underlying instrument will be priced incl. accrued interest.
- Equity: Forward on equity underlyings like stocks or equity funds. Only Continuous dividends are possible.
- FX: Forward on currencies. Price is depending on underlying price and foreign and domestic discount factors.
- EquityFuture: Standardized contract on equity underlyings. A net basis can be specified.
- BondFuture: Standardized contract on Bond underlyings. A net basis can be specified.

In the following, all methods and attributes are explained and a code example is given.

Methods for Forward object *obj*:

- Forward(*id*) or Forward(): Constructor of a Forward object. *id* is optional and specifies id and name of new object.
- obj.set(*attribute*,*value*): Setter method. Provide pairs of attributes and values. Values are checked for format and constraints.
- obj.get(*attribute*): Getter method. Query the value of specified attribute.
- obj.calc_value(*valuation_date*,*scenario*, *discount_curve_object*, *underlying_object*, *und_curve_object*) Calculate the value of Forwards based on valuation date, scenario type, discount curve and underlying instruments. Underlying discount curve *und_curve_object* is used for Forwards on Bond or FX rates only.
- obj.getValue(*scenario*): Return Forward value for given *scenario*. Method inherited from Superclass *Instrument*
- obj.calc_sensitivities(*valuation_date*, *discount_curve_object*, *underlying_object*, *und_curve_object*) Calculate sensitivities (the Greeks) of all Forward and Future by numeric approximation.
- Forward.help(*format*,*returnflag*): show this message. Format can be [plain text, html or texinfo]. If empty, defaults to plain text. Returnflag is boolean: True returns documentation string, false (default) returns empty string. [static method]

Attributes of Forward objects:

- *id*: Instrument id. Has to be unique identifier. (Default: empty string)
- *name*: Instrument name. (Default: empty string)
- *description*: Instrument description. (Default: empty string)
- *value_base*: Base value of instrument of type real numeric. (Default: 0.0)

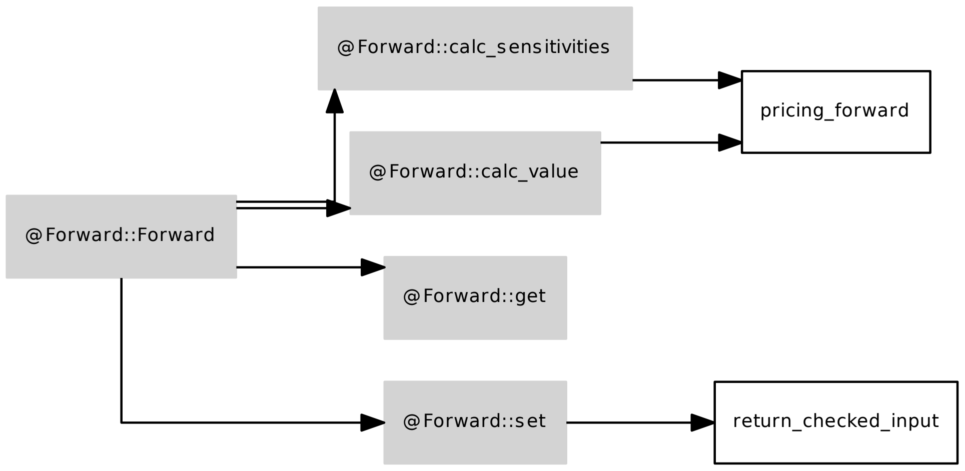
- *currency*: Currency of instrument of type string. (Default: 'EUR') During instrument valuation and aggregation, FX conversion takes place if corresponding FX rate is available.
- *asset_class*: Asset class of instrument. (Default: 'derivative')
- *type*: Type of instrument, specific for class. Set to 'Forward'.
- *value_stress*: Line vector with instrument stress scenario values.
- *value_mc*: Line vector with instrument scenario values. MC values for several *timestep_mc* are stored in columns.
- *timestep_mc*: String Cell array with MC timesteps. If new timesteps are set, values are automatically appended.
- *issue_date*: Issue date of Forward (date in format DD-MMM-YYYY)
- *maturity_date*: Maturity date of Forward (date in format DD-MMM-YYYY)
- *day_count_convention*: Day count convention of curve. See 'help get_basis' for details (Default: 'act/365')
- *compounding_type*: Compounding type. Can be continuous, discrete or simple. (Default: 'cont')
- *compounding_freq*: Compounding frequency used for discrete compounding. Can be [daily, weekly, monthly, quarterly, semi-annual, annual]. (Default: 'annual')
- *strike_price*: Strike price (Default: 0.0)
- *underlying_id*:ID of underlying object. (Default: ")
- *underlying_price_base*: Underlying base price. Used only if underlying object is a risk factor. Risk factor shocks are applied to underlying base price. (Default: 0.0)
- *underlying_sensitivity*: Underlying sensitivity used only, if underlying object is a risk factor. Risk factor shocks are scaled by this sensitivity (Default: 1.0)
- *discount_curve*: Discount curve (Default: 'IR_EUR')
- *foreign_curve*: Foreign curve, used for Bond and FX Forwards only (Default: 'IR_USD')
- *multiplier*: Multiplier. Used to scale price and value of one constract. (Default: 1)
- *dividend_yield*: Dividend yield is part of total cost of carry. Used for Equity Forwards only. (Default: 0.0)
- *convenience_yield*: Convenience yield is part of total cost of carry. Used for Equity Forwards only. (Default: 0.0)
- *storage_cost*: Storage cost (yield) is part of total cost of carry. Used for Equity Forwards only. (Default: 0.0)
- *spread*: Unsued: Spread of Forward (Default: 0.0)
- *cf_dates*: Unused: Cash flow dates (Default: [])
- *cf_values*: Unused: Cash flow values (Default: [])
- *component_weight*: Used for Bond futures only. Scale future future price.
- *net_basis*: Net basis of futures. Used only, if *calc_price_from_netbasis* is set to true.

- *calc_price_from_netbasis*: Boolean Flag. True: use *net_basis* to calculate future price. (Default: false).
- *theo_delta*: Sensitivity to changes in underlying's price. Calculated by method *calc_sensitivities*.
- *theo_gamma*: Sensitivity to changes in changes of underlying's price. Calculated by method *calc_sensitivities*.
- *theo_vega*: Sensitivity to changes in volatility. Calculated by method *calc_sensitivities*.
- *theo_theta*: Sensitivity to changes in remaining days to maturity. Calculated by method *calc_sensitivities*.
- *theo_rho*: Sensitivity to changes in risk free rate. Calculated by method *calc_sensitivities*.
- *theo_domestic_rho*: Sensitivity to changes in domestic interest rate. Calculated by method *calc_sensitivities*.
- *theo_foreign_rho*: Sensitivity to changes in foreign interest rate. Calculated by method *calc_sensitivities*.
- *theo_price*: Forward price. Calculated by method *calc_value*.

For illustration see the following example: An equity forward with 10 years to maturity, an underlying index and a discount curve are set up and the forward value (-27.2118960639903) is calculated and retrieved:

```
c = Curve();
c = c.set('id','IR_EUR','nodes',[365,3650,7300]);
c = c.set('rates_base',[0.0001002070,0.0045624391,0.009346842]);
c = c.set('method_interpolation','linear');
i = Index();
i = i.set('value_base',326.900);
f = Forward();
f = f.set('name','EQ_Forward_Index_Test','maturity_date','26-Mar-2036');
f = f.set('strike_price',426.900);
f = f.set('compounding_freq','annual');
f = f.calc_value('31-Mar-2016','base',c,i);
f.getValue('base')
```

Dependencies of class:



4.5 Option.help

`object = ` *Option*(`id`) [Octarisk Class]
`object = ` *Option*() [Octarisk Class]

Class for setting up Option objects. Possible underlyings are financial instruments or indizes. Therefore the following Option types are introduced:

- OPT_EUR_C: European Call option priced by Black-Scholes model.
- OPT_EUR_P: European Put option priced by Black-Scholes model.
- OPT_AM_C: American Call option priced by Willow tree model, CRR binomial tree or Bjerksund-Stensland approximation.
- OPT_AM_P: European Put option priced by Willow tree model, CRR binomial tree or Bjerksund-Stensland approximation.
- OPT_BAR_C: European Barrier Call option. Can have all combinations of out or in and up or down barrier types. Priced with Merton, Reiner, Rubinstein model.
- OPT_BAR_P: European Barrier Put option. Same restrictions as Barrier Call options.
- OPT_ASN_C: European Asian Call option. Average rate only. The following compounding types can be used: geometric continuous (Kemna-Vorst90 pricing model) or arithmetic continuous (Levy pricing model)
- OPT_ASN_P: European Asian Put option. Same restrictions as Asian Call options.
- OPT_BIN_C: European Binary Call option. Binary types gap, supershare, asset-or-nothing or cash-or-nothing. Priced with Reiner-Rubinstein model.

- OPT_BIN_P: European Binary Put option. Same restrictions as Binary Call options.
- OPT_LBK_C: European Lookback Call option. Lookback types floating_strike or fixed_strike. Priced with Conze and Viswanathan or Goldman, Sosin and Gatto model.
- OPT_LBP_P: European Lookback Put option. Same restrictions as Lookback Call options.

In the following, all methods and attributes are explained and a code example is given.

Methods for Option object *obj*:

- Option(*id*) or Option(): Constructor of a Option object. *id* is optional and specifies id and name of new object.
- obj.set(*attribute*,*value*): Setter method. Provide pairs of attributes and values. Values are checked for format and constraints.
- obj.get(*attribute*): Getter method. Query the value of specified attribute.
- obj.calc_value(*valuation_date*,*scenario*, *underlying*, *discount_curve*, *volatility_surface*, *path_static*) Calculate the value of Options based on valuation date, scenario type, discount curve, underlying instrument and volatility surface. The pricing model is chosen based on Option type and instrument model attributes. A path to precalculated Willow trees for pricing American options by Willowtree model can be provided.
- obj.calc_greeks(*valuation_date*,*scenario*, *underlying*, *discount_curve*, *volatility_surface*, *path_static*) Calculate sensitivities (the Greeks) for the given Option instrument. For plain-vanilla European Options the Greeks are calculated by Black-Scholes pricing. The Greeks of all other Option types will be calculated by numeric approximation.
- obj.calc_vola_spread(*valuation_date*, *underlying*, *discount_curve*, *volatility_surface*, *path_static*) Calibrate volatility spread in order to match the Option price with the market price. The volatility spread will be used for further pricing.
- obj.getValue(*scenario*): Return Option value for given *scenario*. Method inherited from Superclass *Instrument*
- Option.help(*format*,*returnflag*): show this message. Format can be [plain text, html or texinfo]. If empty, defaults to plain text. Returnflag is boolean: True returns documentation string, false (default) returns empty string. [static method]

Attributes of Option objects:

- *id*: Instrument id. Has to be unique identifier. (Default: empty string)
- *name*: Instrument name. (Default: empty string)
- *description*: Instrument description. (Default: empty string)
- *value_base*: Base value of instrument of type real numeric. (Default: 0.0)
- *currency*: Currency of instrument of type string. (Default: 'EUR') During instrument valuation and aggregation, FX conversion takes place if corresponding FX rate is available.

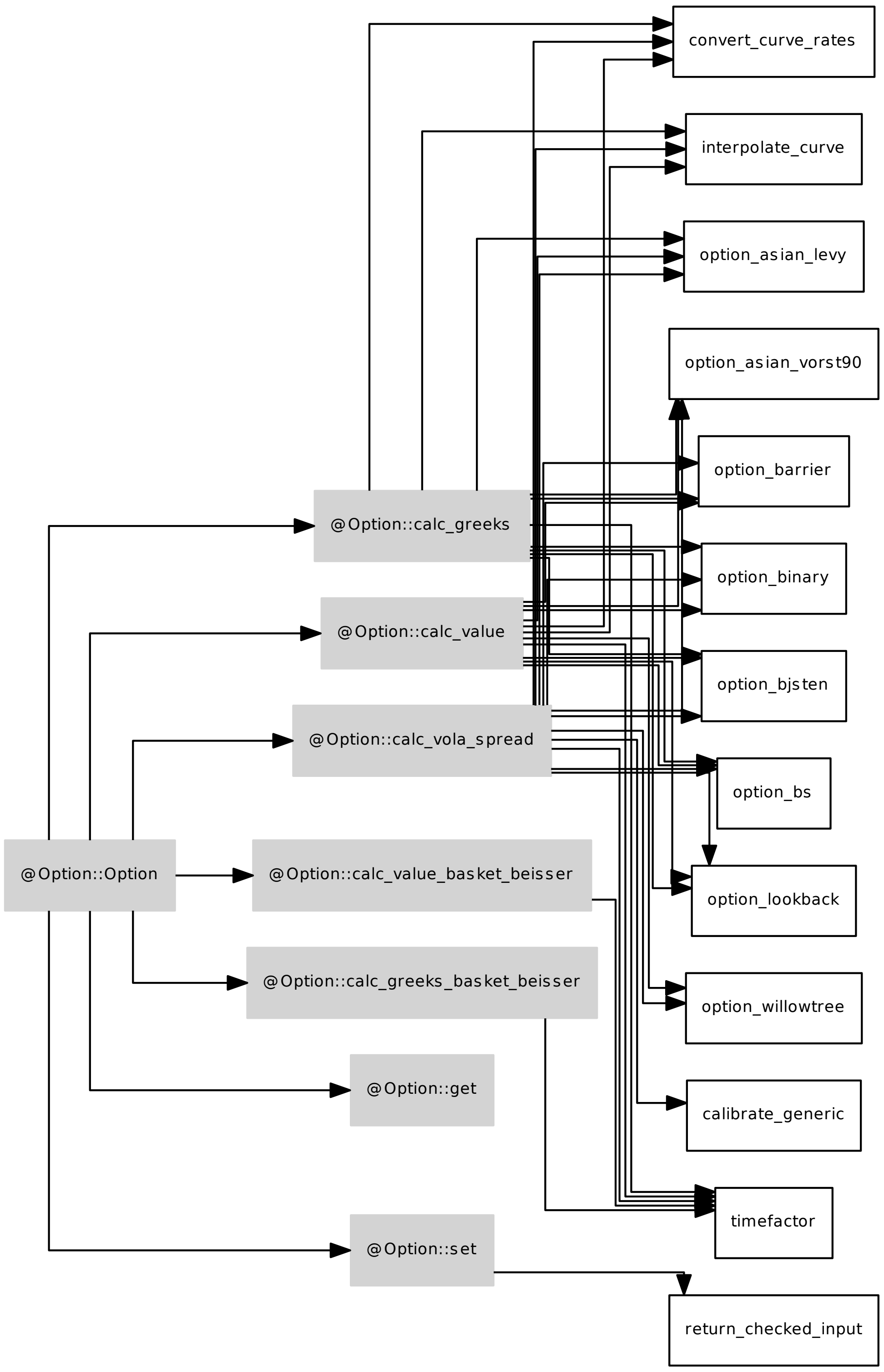
- *asset_class*: Asset class of instrument. (Default: 'derivative')
- *type*: Type of instrument, specific for class. Set to 'Option'.
- *value_stress*: Line vector with instrument stress scenario values.
- *value_mc*: Line vector with instrument scenario values. MC values for several *timestep_mc* are stored in columns.
- *timestep_mc*: String Cell array with MC timesteps. If new timesteps are set, values are automatically appended.
- *maturity_date*: Maturity date of Option (date in format DD-MMM-YYYY)
- *day_count_convention*: Day count convention of curve. See 'help get_basis' for details (Default: 'act/365')
- *compounding_type*: Compounding type. Can be continuous, discrete or simple. (Default: 'cont')
- *compounding_freq*: Compounding frequency used for discrete compounding. Can be [daily, weekly, monthly, quarterly, semi-annual, annual]. (Default: 'annual')
- *spread*: Interest rate spread used in calculating risk free interest rate. Default: 0.0;
- *discount_curve*: ID of discount curve. Default: empty string
- *underlying*: ID of underlying object (instrument or risk factor). Default: empty string
- *vola_surface*: ID of volatility surface. Default: empty string
- *vola_sensi*: Sensitivity scaling factor for volatility. Default: 1
- *strike*: Strike value of Option (used to set maximum and minimum values for lookback options). Default: 100
- *spot*: Spot value of underlying instrment. Only used, if underlying is risk factor.
- *multiplier*: Multiplier of Option. Resulting Option price is scales by this multiplier. Default: 5
- *div_yield*: Continuous dividend yield of underlying (act/365 day count convention assumed). Default: 0.0
- *timesteps_size*: American Willow Tree timestep size (in days). Default: 5
- *willowtree_nodes*: American Willow Tree nodes per timestep. Default: 20
- *pricing_function_american*: American pricing model [Willowtree,BjSten]. Default: 'BjSten'
- *binary_type*: Binary option type. Can be ['cash','gap','asset','supershare']. Defaults to 'cash'.
- *lookback_type*: Lookback option type. Can be ['floating_strike','fixed_strike']. Defaults to 'floating_strike'.
- *payoff_strike*: Binary Option payoff strike (used as e.g. upper bound or strike). Default: 100
- *upordown*: Barrier Up or Down description. Default: 'U'
- *outorin*: Barrier In or Out description. Default: 'out'
- *barrierlevel*: Barrier level. Default: 0.0

- *rebate*: Barrier rebate (payoff in case of a barrier event). Default: 0.0
- *averaging_type* Asian option averaging type ['rate','strike']. Defaults to 'rate'.
- *averaging_rule* = Asian option underlying distribution of average type ['geometric','arithmetic']. Defaults to 'geometric'
- *averaging_monitoring* Asian option average monitoring. Can only be 'continuous'
- *theo_delta*: Sensitivity to changes in underlying's price. Calculate by method *calc_greeks*.
- *theo_gamma*: Sensitivity to changes in changes of underlying's price. Calculate by method *calc_greeks*.
- *theo_vega*: Sensitivity to changes in volatility. Calculate by method *calc_greeks*.
- *theo_theta*: Sensitivity to changes in remaining days to maturity. Calculate by method *calc_greeks*.
- *theo_rho*: Sensitivity to changes in risk free rate. Calculate by method *calc_greeks*.
- *theo_omega*: Specified as *theo_delta* scaled by underlying value over option base value. Calculate by method *calc_greeks*.

For illustration see the following example: An American equity Option with 10 years to maturity, an underlying index, a volatility surface and a discount curve are set up and the Option value (123.043), volatility spread and the Greeks are calculated by the Willowtree model and retrieved:

```
disp('Pricing American Option Object (Willowtree)')
c = Curve();
c = c.set('id','IR_EUR','nodes',[730,3650,4380], ...
'rates_base',[0.0001001034,0.0045624391,0.0062559362], ...
'method_interpolation','linear');
v = Surface();
v = v.set('axis_x',3650,'axis_x_name','TERM','axis_y',1.1, ...
'axis_y_name','MONEYNESS');
v = v.set('values_base',0.210360082233);
v = v.set('type','IndexVol');
i = Index();
i = i.set('value_base',286.867623322,'currency','USD');
o = Option();
o = o.set('maturity_date','29-Mar-2026','currency','USD', ...
'timesteps_size',5,'willowtree_nodes',30);
o = o.set('strike',368.7362,'multiplier',1,'sub_Type','OPT_AM_P');
o = o.set('pricing_function_american','Willowtree');
o = o.calc_value('31-Mar-2016','base',i,c,v);
o = o.calc_greeks('31-Mar-2016','base',i,c,v);
value_base = o.getValue('base')
theo_omega = o.get('theo_omega')
disp('Calibrating volatility spread over yield:')
o = o.set('value_base',100);
o = o.calc_vola_spread('31-Mar-2016',i,c,v);
o.getValue('base')
```

Dependencies of class:



4.6 Cash.help

object = Cash(id) [Octarisk Class]
object = Cash() [Octarisk Class]

Class for setting up Cash objects.

This class contains all attributes and methods related to the following Cash types:

- Cash: Specify riskless cash instruments

In the following, all methods and attributes are explained and a code example is given.

Methods for Cash object *obj*:

- Cash(*id*) or Cash(): Constructor of a Cash object. *id* is optional and specifies id and name of new object.
- obj.set(*attribute*,*value*): Setter method. Provide pairs of attributes and values. Values are checked for format and constraints.
- obj.get(*attribute*): Getter method. Query the value of specified attribute.
- obj.calc_value(*scenario*,*scen_number*): Extends base value to vector of row size *scen_number* and stores vector for given *scenario*. Cash instruments are per definition risk free.
- obj.getValue(*scenario*): Return Cash value for given *scenario*. Method inherited from Superclass *Instrument*
- Cash.help(*format*,*returnflag*): show this message. Format can be [plain text, html or texinfo]. If empty, defaults to plain text. Returnflag is boolean: True returns documentation string, false (default) returns empty string. [static method]

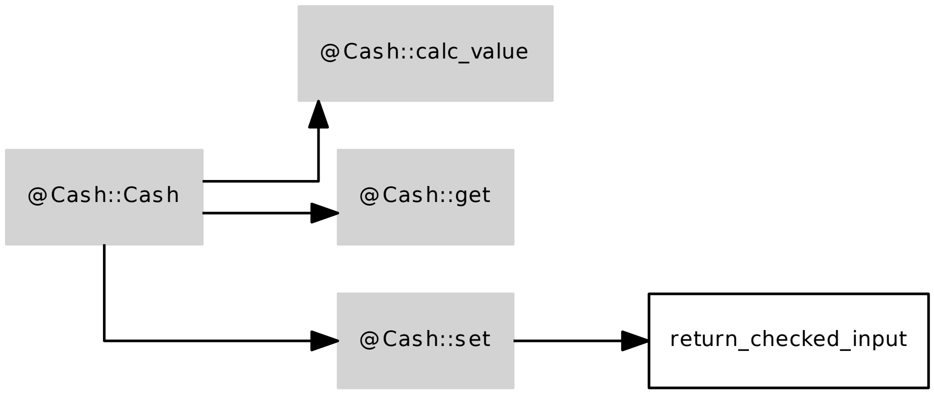
Attributes of Cash objects:

- *id*: Instrument id. Has to be unique identifier. Default: empty string.
- *name*: Instrument name. Default: empty string.
- *description*: Instrument description. Default: empty string.
- *value_base*: Base value of instrument of type real numeric. Default: 0.0.
- *currency*: Currency of instrument of type string. Default: 'EUR' During instrument valuation and aggregation, FX conversion takes place if corresponding FX rate is available.
- *asset_class*: Asset class of instrument. Default: 'unknown'
- *type*: Type of instrument, specific for class. Set to 'cash'.
- *value_stress*: Line vector with instrument stress scenario values.
- *value_mc*: Line vector with instrument scenario values. MC values for several *timestep_mc* are stored in columns.
- *timestep_mc*: String Cell array with MC timesteps. If new timesteps are set, values are automatically appended.

For illustration see the following example: A THB Cash instrument is being generated and during value calculation the stress and MC scenario values with 20 resp. 1000 scenarios are derived from the base value:

```
c = Cash();
c = c.set('id','THB_CASH','name','Cash Position THB');
c = c.set('asset_class','cash','currency','THB');
c = c.set('value_base',346234.1256);
c = c.calc_value('stress',20);
c = c.calc_value('250d',1000);
value_stress = c.getValue('stress');
```

Dependencies of class:



4.7 Debt.help

object = *Debt*(*id*) [Octarisk Class]
object = *Debt*() [Octarisk Class]

Class for setting up Debt objects. The idea of this class is to model baskets (funds) of bond instruments. The shocked value is derived from a sensitivity approach based on Modified duration and convexity. These sensitivities describe the total basket properties in terms of interest rate sensitivity. The following formula is applied to calculate the instrument shock:

```
 dP
--- = -D dY + 0.5 C dY^2
 P
```

If you want to model all underlying bonds directly, use Bond class for underlyings and Synthetic class for the basket (fund).
This class contains all attributes and methods related to the following Debt types:

- DBT: Standard debt type

In the following, all methods and attributes are explained and a code example is given.
Methods for Debt object *obj*:

- Debt(*id*) or Debt(): Constructor of a Debt object. *id* is optional and specifies id and name of new object.
- obj.set(*attribute*,*value*): Setter method. Provide pairs of attributes and values. Values are checked for format and constraints.
- obj.get(*attribute*): Getter method. Query the value of specified attribute.
- obj.calc_value(*discount_curve*,*scenario*): Calculate instrument shocked value based on interest rate sensitivity. Modified Duration and Convexity are used to predict change in value based on absolute shock discount curve at given term.
- obj.getValue(*scenario*): Return Debt value for given *scenario*. Method inherited from Superclass *Instrument*
- Debt.help(*format*,*returnflag*): show this message. Format can be [plain text, html or texinfo]. If empty, defaults to plain text. Returnflag is boolean: True returns documentation string, false (default) returns empty string. [static method]

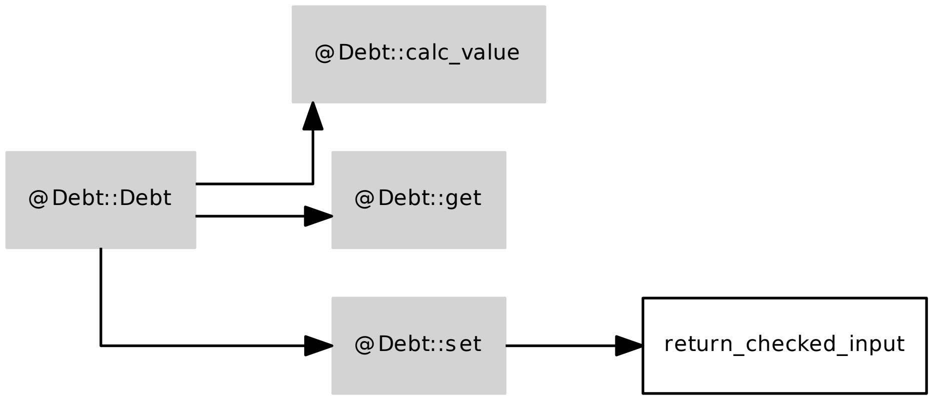
Attributes of Debt objects:

- *id*: Instrument id. Has to be unique identifier. Default: empty string.
- *name*: Instrument name. Default: empty string.
- *description*: Instrument description. Default: empty string.
- *value_base*: Base value of instrument of type real numeric. Default: 0.0.
- *currency*: Currency of instrument of type string. Default: 'EUR' During instrument valuation and aggregation, FX conversion takes place if corresponding FX rate is available.
- *asset_class*: Asset class of instrument. Default: 'debt'
- *type*: Type of instrument, specific for class. Set to 'debt'.
- *value_stress*: Line vector with instrument stress scenario values.
- *value_mc*: Line vector with instrument scenario values. MC values for several *timestep_mc* are stored in columns.
- *timestep_mc*: String Cell array with MC timesteps. If new timesteps are set, values are automatically appended.
- *discount_curve*: Discount curve is used as sensitivity curve to derive absolute shocks at term given by duration.
- *term*: Term of debt instrument (in years). Equals average maturity of all underlying cash flows.
- *duration*: Modified duration of debt instrument.
- *convexity*: Convexity of debt instrument.

For illustration see the following example: Stress values of a debt instrument with average maturity of underlyings of 8.35 years and given duration and convexity are calculated based on 100bp parallel down- and upshift scenarios of a given discount curve. Stress results are [108.5300000000000;91.1969278203125]

```
          c = Curve();
          c = c.set('id','IR_EUR','nodes',[730,3650,4380],'rates_base',[0.01,0.02,0.025],'m
          c = c.set('rates_stress',[0.00,0.01,0.015;0.02,0.031,0.035],'method_interpolation
          d = Debt();
          d = d.set('duration',8.35,'convexity',18,'term',8.35);
          d = d.calc_value(c,'stress');
          d.getValue('base')
          d.getValue('stress')
```

Dependencies of class:



4.8 Sensitivity.help

object = *Sensitivity*(*id*) [Octarisk Class]
object = *Sensitivity*() [Octarisk Class]

Class for setting up Sensitivity objects. This class contains two different instrument setups. The first idea of this class is to model an instrument whose shocks are derived from underlying risk factor shocks or idiosyncratic risk (for MC only). The shocks from these risk factors are then applied to the instrument base value under the assumption of a Geometric Brownian Motion or Brownian Motion. Basically, all real assets like Equity or Commodity can be modelled with this class. The combined shock is a linear combination of all underlying risk factor shocks:

```
V_shock = V_base * exp(Sum_i=1...n [dRF_i * w_i]) (Model: GBM)
V_shock = V_base + (Sum_i=1...n [dRF_i * w_i]) (Model: BM)
```

with the new shock Value V_shock, base V_base, risk factor shock dRF_i and risk factor weight w_i.

The second idea is to use this class to specify a polynomial function or taylor series of underlying instruments, risk factors, curves, surfaces or indizes and derive the sensitivity value with the following formulas. If Taylor expansion shall be used:

```
    V_shock = V_base + a1/b1 * x1^b1 + a2/b2 * x2^b2 + .. + an/bn * xn^bn * am/bm * x
```

The base value is used only if appropriate flag is set. Otherwise, a polynomial function can be set up:

```
    V_shock = V_base + a1 * x1^b1 + a2 * x2^b2 + .. + an * xn^bn * am * xm^bm
```

with the new shock Value V_shock, base V_base, and prefactors a, exponents b and a multiplicative combination of cross terms (term with equal cross terms are multiplied with each other, term with cross terms equal zero are added to the total value) All combined cross terms and all single terms are finally summed up.

This class contains all attributes and methods related to the following Sensitivity types:

- EQU: Equity sensitivity type
- RET: Real Estate sensitivity type
- COM: Commodity sensitivity type
- STK: Stock sensitivity type
- ALT: Alternative investments sensitivity type
- SENSI: Taylor series or polynomial equation of underlying objects

which stands for Equity, Real Estate, Commodity, Stock and Alternative Investments. All sensitivity types assume a geometric brownian motion or brownian motion as underlying stochastic process.

In the following, all methods and attributes are explained and a code example is given.

Methods for Sensitivity object *obj*:

- Sensitivity(*id*) or Sensitivity(): Constructor of a Sensitivity object. *id* is optional and specifies id and name of new object.
- obj.set(*attribute*,*value*): Setter method. Provide pairs of attributes and values. Values are checked for format and constraints.
- obj.get(*attribute*): Getter method. Query the value of specified attribute.
- obj.calc_value(*valuation_date*, *scenario*, *riskfactor_struct*, *instrument_struct*, *index_struct*, *curve_struct*, *surface_struct*, [*scen_number*]): Method for calculation of sensitivity value. Only structures with used objects need to be set.
- obj.valuate(*valuation_date*, *scenario*, *instrument_struct*, *surface_struct*, *matrix_struct*, *curve_struct*, *index_struct*, *riskfactor_struct*, [*para_struct*]) Generic instrument valuation method. All objects required for valuation of the instrument are taken from provided structures (e.g. curves, riskfactors, underlying indizes). Method inherited from Superclass *Instrument*.
- obj.getValue(*scenario*): Return Sensitivity value for given *scenario*. Method inherited from Superclass *Instrument*
- Sensitivity.help(*format*,*returnflag*): show this message. Format can be [plain text, html or texinfo]. If empty, defaults to plain text. Returnflag is boolean: True returns documentation string, false (default) returns empty string. [static method]

Attributes of Sensitivity objects:

- *id*: Instrument id. Has to be unique identifier. Default: empty string.

- *name*: Instrument name. Default: empty string.
- *description*: Instrument description. Default: empty string.
- *value_base*: Base value of instrument of type real numeric. Default: 0.0.
- *currency*: Currency of instrument of type string. Default: 'EUR' During instrument valuation and aggregation, FX conversion takes place if corresponding FX rate is available.
- *asset_class*: Asset class of instrument. Default: empty string
- *type*: Type of instrument, specific for class. Set to 'sensitivity'.
- *value_stress*: Line vector with instrument stress scenario values.
- *value_mc*: Line vector with instrument scenario values. MC values for several *timestep_mc* are stored in columns.
- *timestep_mc*: String Cell array with MC timesteps. If new timesteps are set, values are automatically appended.
- *riskfactors*: Cell with IDs of all underlying risk factors. Default: empy cell.
- *sensitivities*: Vector with weights of riskfactors of same length and order as riskfactors cell. Default: empty vector
- *idio_vola*: Idiosyncratic volatility of sensitivity instrument. Used only if one riskfactor is set to 'IDIO'. Applies a shock given by a normal distributed random variable with standard deviation taken from the value of attribute *idio_vola* and a mean of zero.
- *model*: Model specifies the stochastic process. Can be a Geometric Brownian Notion (GBM) or Brownian Notion (BM). (Default: 'GBM')
- *underlyings*: cell array of underlying indizes, curves, surfaces, instruments, risk factors
- *x_coord*: vector with x coordinates of underlying (curves, surfaces, cubes only)
- *y_coord*: vector y coordinate of underlying (surfaces, cubes only)
- *z_coord*: vector z coordinate of underlying (cubes only)
- *shock_type*: cell array of shock types for each underlying [value, relative, absolute]
- *sensi_prefactor*: vector with prefactors (a in a*x^b)
- *sensi_exponent*: vector with exponents (b in a*x^b)
- *sensi_cross*: vector with cross terms [0 = single; 1,2,3, ... link cross terms]
- *use_value_base*: boolean flag: use value_base for base valuation. Scenario shocks are added to base value (default: false)
- *use_taylor_exp*: boolean flag: if true, treat polynomial value as Taylor expansion(default: false)
- *cf_dates*: Unused: Cash flow dates (Default: [])
- *cf_values*: Unused: Cash flow values (Default: [])

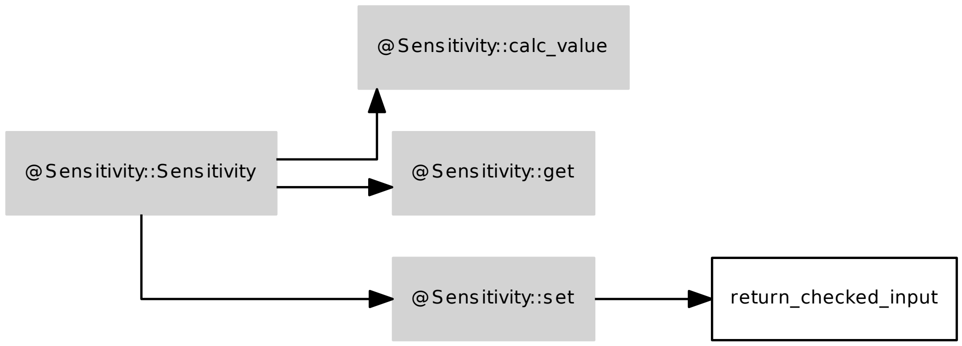
For illustration see the following example: An All Country World Index (ACWI) fund with base value of 100 USD shall be modelled with both instrument setups (Linear combination of risk factor shocks and a polynomial (linear) function with two single terms. Underlying risk factors are the MSCI Emerging Market and MSCI World Index. The sensitivities to both risk factors are equal to the weights of the subindices in the ACWI index. Both risk factors are shocked during a stress scenario and the total shock values for the fund are calculated:

```
 fprintf(' doc_instrument:  Pricing Sensitivity Instrument (Polyno-
mial Function)');
 r1 = Riskfactor();
  r1 = r1.set('id','MSCI_WORLD','scenario_stress',[20;-10], ...
        'model','GBM','shift_type',[1;1]);
  r2 = Riskfactor();
  r2 = r2.set('id','MSCI_EM','scenario_stress',[10;-20], ...
        'model','GBM','shift_type',[1;1] );
  riskfactor_struct = struct();
  riskfactor_struct(1).id = r1.id;
  riskfactor_struct(1).object = r1;
  riskfactor_struct(2).id = r2.id;
  riskfactor_struct(2).object = r2;
  s = Sensitivity();
  s = s.set('id','MSCI_ACWI_ETF','sub_type','SENSI', 'currency', 'USD' , ...
        'asset_class','Equity',  'value_base', 100, ...
        'underlyings',cellstr(['MSCI_WORLD';'MSCI_EM']), ...
        'x_coord',[0,0], ...
        'y_coord',[0,0.0], ...
        'z_coord',[0,0], ...
        'shock_type', cellstr(['absolute';'absolute']), ...
        'sensi_prefactor', [0.8,0.2], 'sensi_exponent', [1,1], ...
        'sensi_cross', [0,0], 'use_value_base',true,'use_taylor_exp',false);
  instrument_struct = struct();
  instrument_struct(1).id = s.id;
  instrument_struct(1).object = s;
s = s.calc_value('31-Dec-2016', 'base',riskfactor_struct,instrument_struct,[],[],
s.getValue('base')
s = s.calc_value('31-Dec-2016', 'stress',riskfactor_struct,instrument_struct,[],[
s.getValue('stress')

 fprintf(' doc_instrument:  Pricing Sensitivity Instrument (Riskfac-
tor linear combination)');
 r1 = Riskfactor();
  r1 = r1.set('id','MSCI_WORLD','scenario_stress',[20;-10], ...
        'model','BM','shift_type',[1;1]);
  r2 = Riskfactor();
  r2 = r2.set('id','MSCI_EM','scenario_stress',[10;-20], ...
        'model','BM','shift_type',[1;1] );
  riskfactor_struct = struct();
  riskfactor_struct(1).id = r1.id;
  riskfactor_struct(1).object = r1;
  riskfactor_struct(2).id = r2.id;
  riskfactor_struct(2).object = r2;
  s = Sensitivity();
  s = s.set('id','MSCI_ACWI_ETF','sub_type','EQU', 'currency', 'USD', ...
        'asset_class','Equity', 'model', 'BM', ...
        'riskfactors',cellstr(['MSCI_WORLD';'MSCI_EM']), ...
        'sensitivities',[0.8,0.2],'value_base',100.00);
  instrument_struct = struct();
  instrument_struct(1).id = s.id;
  instrument_struct(1).object = s;
  s = s.valuate('31-Dec-2016', 'stress', ...
```

Stress results are [118;88].

Dependencies of class:



4.9 Riskfactor.help

object = *Riskfactor*(*id*) [Octarisk Class]
object = *Riskfactor*() [Octarisk Class]

Class for setting up Riskfactor objects.
The mapping between e.g. curve or index objects and their corresponding risk factors is automatically done by using regular expressions to match the names. Riskfactors always have to begin with 'RF_' followed by the object name. If certain nodes of curves or surfaces are shocked, the name is followed by an additional node identifier, e.g. 'RF_IR_EUR_SWAP_1Y' for shocking an interest rate curve or 'RF_VOLA_IR_EUR_1825_3650' for shocking a certain point on the volatility tenor / term surface.
Riskfactors can be either shocked during stresses, where custom absolute or relative shocks can be defined. During Monte-Carlo scenario generation risk factor shocks are calculated by applying statistical processes according to specified stochastic model. The random numbers follow a match of given mean, standard deviation, skewness and kurtosis according to distributions selected by the Pearson Type I-VII distribution system.
This class contains all attributes and methods related to the following Riskfactor types:

- RF_IR: Interest rate risk factor.
- RF_SPREAD: Spread risk factor.
- RF_COM: Commodity risk factor.
- RF_RE: Real estate risk factor.
- RF_EQ: Equity risk factor.
- RF_VOLA: Volatility risk factor.
- RF_ALT: Alternative investment risk factor.

- RF_INFL: Inflation risk factor.
- RF_FX: Forex risk factor.

In the following, all methods and attributes are explained and a code example is given.
Methods for Riskfactor object *obj*:

- Riskfactor(*id*) or Riskfactor(): Constructor of a Riskfactor object. *id* is optional and specifies id and name of new object.
- obj.set(*attribute*,*value*): Setter method. Provide pairs of attributes and values. Values are checked for format and constraints.
- obj.get(*attribute*): Getter method. Query the value of specified attribute.
- obj.getValue(*scenario*, *abs_flag*, *sensitivity*): Return Riskfactor value according to scenario type. If optional parameter abs_flag is true returns Riskfactor scenario values. Therefore static method Riskfactor.get_abs_values will be called.
- Riskfactor.help(*format*,*returnflag*): show this message. Format can be [plain text, html or texinfo]. If empty, defaults to plain text. Returnflag is boolean: True returns documentation string, false (default) returns empty string. [static method]
- Riskfactor.get_abs_values(*model*, *scen_deltavec*, *value_base*, *sensitivity*): Calculate absolute scenario value for given base value, sensitivity, model and shock. [static method]
- Riskfactor.get_basis(*dcc_string*): Return basis integer value for given day count convention string.

Attributes of Riskfactor objects:

- *id*: Riskfactor id. Has to be unique identifier. Default: empty string.
- *name*: Riskfactor name. Default: empty string.
- *description*: Riskfactor description. Default: empty string.
- *type*: Riskfactor type. Can be [RF_IR, RF_SPREAD, RF_COM, RF_RE, RF_EQ, RF_VOLA, RF_ALT, RF_INFL or RF_FX]
- *model*: Stochastic risk factor model. Can be [Geometric Brownian Motion (GBM), Brownian Motioan (BM), Black-Karasinsky Model (BKM), Shifted Log-Normal (SLN), Ornstein-Uhlenbeck (OU), Square-Root Diffusion (SRD)]. Default: empty string.
- *mean*: Annualized targeted marginal mean (drift) of risk factor. Default: 0.0
- *std*: Annualized targeted marginal standard deviation of risk factor. Default: 0.0
- *skew*: Targeted marginal skewness of risk factor. Default: 0.0
- *kurt*: Targeted marginal kurtosis of risk factor. Default: 0.0
- *value_base*: Base value of risk factor (required for mean reverting stochastic models). Default: 0.0
- *mr_level*: Mean reversion level. Default: 0.0
- *mr_rate*: Mean reversion parameter. Default: 0.0
- *node*: Risk factor term value in first dimension (in days). For curves equals term in days at x-axis (term). Default: 0.0

- *node2*: Risk factor term value in second dimension. For interest rate surfaces equals term in days at y-axis (tenor or term). For index surfaces equals moneyness. Default: 0.0

- *node3*: Risk factor term value in third dimension. For volatility cubes equals moneyness at z-axis. Default: 0.0

- *sln_level*: Shift parameter (shift level) of shifted log-normal distribution. Default: 0.0

- *scenario_mc*: Vector with risk factor shock values. MC rates for several MC timesteps are stored in layers.

- *scenario_stress*: Vector with risk factor shock values.

- *timestep_mc*: String Cell array with MC timesteps. Automatically appended if values for new timesteps are set.

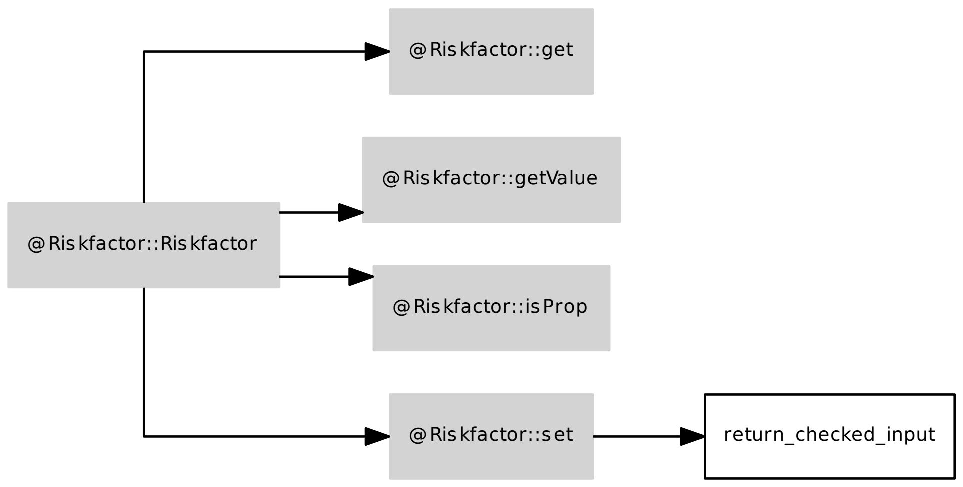
- *shocktype_mc*: Specify how to apply risk factor shocks in Monte Carlo scenarios. Can be [absolute, relative, sln_relative]. Automatically set by scripts. Default: absolute

- *shift_type*: Specify a vector specifying stress risk factor shift type . Can be either 0 (absolute) or 1 (relative) shift.

For illustration see the following example: A swap risk factor modelled by a shifted log-normal model at the three year node is set up and shifted in three stress scenarios (absolute up- and downshift, relative downshift):

```
disp('Setting up Swap(3650) risk factor')
r = Riskfactor();
r = r.set('id','RF_EUR-SWAP_3Y','name','RF_EUR-SWAP_3Y', ...
    'scenario_stress',[0.02;-0.01;0.8], ...
    'type','RF_IR','model','SLN','shift_type',[0;0;1], ...
    'mean',0.0,'std',0.117,'skew',0.0,'kurt',3, ...
    'node',1095,'sln_level',0.03)
```

Dependencies of class:



4.10 Index.help

`object = Index(id)` [Octarisk Class]
`object = Index()` [Octarisk Class]

Class for setting up Index objects.
Index class is used for specifying asset indizes, exchange rates and consumer price indizes. Indizes serve as underlyings for e.g. Options or Forwards, are used to set up forex rates or CPI indizes for inflation linked products. Indizes can be shocked with risk factors (e.g. risk factor types RF_EQ or RF_FX) or in MC scenarios.
This class contains all attributes and methods related to the following Index types:

- EQUITY INDEX
- BOND INDEX
- VOLATILITY INDEX
- COMMODITY INDEX
- REAL ESTATE INDEX
- EXCHANGE RATE
- CPI (Consumer Price index)
- AGGREGATED INDEX (consists of underlying indexes)

In the following, all methods and attributes are explained and a code example is given.
Methods for Index object *obj*:

- Index(*id*) or Index(): Constructor of a Index object. *id* is optional and specifies id and name of new object.

- obj.set(*attribute*,*value*): Setter method. Provide pairs of attributes and values. Values are checked for format and constraints.
- obj.get(*attribute*): Getter method. Query the value of specified attribute.
- obj.getValue(*scenario*): Return Index value according to scenario type.
- Index.help(*format*,*returnflag*): show this message. Format can be [plain text, html or texinfo]. If empty, defaults to plain text. Returnflag is boolean: True returns documentation string, false (default) returns empty string. [static method]
- Index.get_basis(*dcc_string*): Return basis integer value for given day count convention string. [static method]

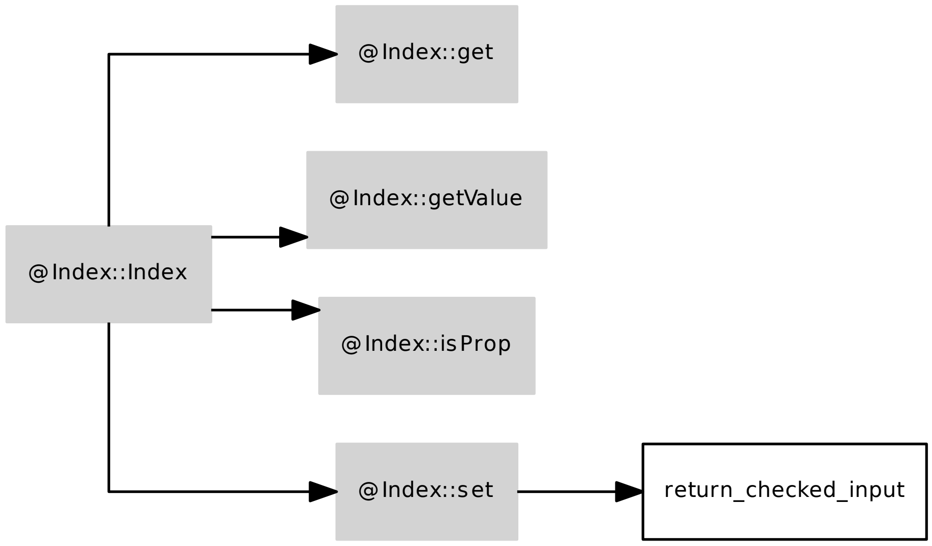
Attributes of Index objects:

- *id*: Index id. Has to be unique identifier. Default: empty string.
- *name*: Index name. Default: empty string.
- *description*: Index description. Default: empty string.
- *type*: Index type. Can be [EQUITY INDEX, BOND INDEX, VOLATILITY INDEX, COMMODITY INDEX, REAL ESTATE INDEX, EXCHANGE RATE, CPI]. Default: empty string.
- *value_base*: Base value of index. Default: 1.0
- *currency*: Index currency. Default: 'EUR'
- *index_function*: Type Aggregated Index only: specifies how to aggregated indexes, which are specified in attribute increments. Can be [sum, product, divide, factor]. [sum, product, divide] specifies mathematical operation applied on all curve increments. [factor] allows only one increment and uses *curve_parameter* for multiplication. Default: 'sum'
- *index_parameter*: Type Aggregated indexes only: used as multiplication parameter for factor *curve_function*.
- *increments*: Type Aggregated indexes only: List of IDs of all underlying indexes. Use *index_function* to specify how to aggregated indexes.
- *scenario_mc*: Vector with Monte Carlo index values. \n
- *scenario_stress*: Vector with Stress index values.
- *timestep_mc*: String Cell array with MC timesteps. Automatically appended if values for new timesteps are set.
- *shift_type*: (unused) Specify a vector specifying stress index shift type . Can be either 0 (absolute) or 1 (relative) shift.

For illustration see the following example:

```
disp('Setting up an equity index and Exchange Rate')
i = Index();
i = i.set('id','MSCIWORLD','value_base',1000, ...
    'scenario_stress',[2000;1333;800],'currency','USD');
fx = Index();
fx = fx.set('id','FX_EURUSD','value_base',1.1,  ...
    'scenario_stress',[1.2;1.18;1.23]);
```

Dependencies of class:



4.11 Synthetic.help

`object = Synthetic(id)` [Octarisk Class]
`object = Synthetic()` [Octarisk Class]

Class for setting up Synthetic objects. A Synthetic instrument is a linear combination of underlying instruments. The following Synthetic types are introduced:

- SYNTH: Synthetic instrument with underlyings. The Synthetic price is based on the linear combination of underlying instrument's prices.
- Basket: The same as type SYNTH, but additional attributes for specifying underlying volatility surface and volatility types are introduced to enable basket option valuation.

In the following, all methods and attributes are explained and a code example is given.

Methods for Synthetic object *obj*:

- Synthetic(*id*) or Synthetic(): Constructor of a Synthetic object. *id* is optional and specifies id and name of new object.
- obj.set(*attribute*,*value*): Setter method. Provide pairs of attributes and values. Values are checked for format and constraints.
- obj.get(*attribute*): Getter method. Query the value of specified attribute.
- obj.calc_value(*valuation_date*,*scenario*, *instrument_struct*, *index_struct*) Calculate the value of Synthetic instruments based on valuation date, scenario type,

underlying instruments and FX rates. The provided structures have to contain the referenced underlying instrument objects and FX rates.

- obj.getValue(*scenario*): Return Synthetic value for given *scenario*. Method inherited from Superclass *Instrument*
- Synthetic.help(*format*,*returnflag*): show this message. Format can be [plain text, html or texinfo]. If empty, defaults to plain text. Returnflag is boolean: True returns documentation string, false (default) returns empty string. [static method]

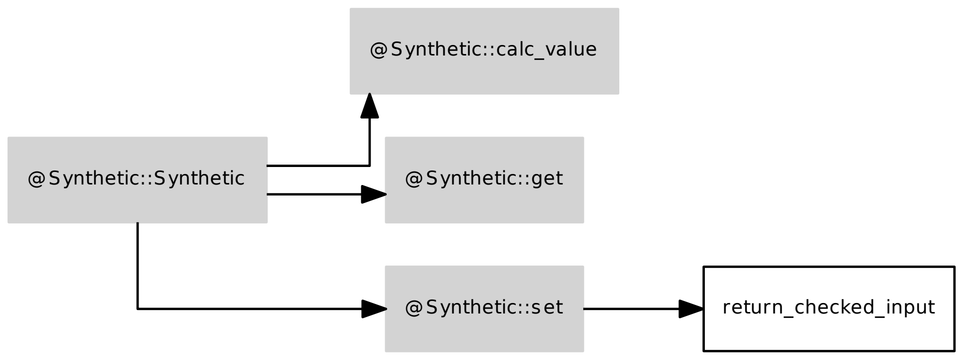
Attributes of Synthetic objects:

- *id*: Instrument id. Has to be unique identifier. (Default: empty string)
- *name*: Instrument name. (Default: empty string)
- *description*: Instrument description. (Default: empty string)
- *value_base*: Base value of instrument of type real numeric. (Default: 0.0)
- *currency*: Currency of instrument of type string. (Default: 'EUR') During instrument valuation and aggregation, FX conversion takes place if corresponding FX rate is available.
- *asset_class*: Asset class of instrument. (Default: 'derivative')
- *type*: Type of instrument, specific for class. Set to 'Synthetic'.
- *value_stress*: Line vector with instrument stress scenario values.
- *value_mc*: Line vector with instrument scenario values. MC values for several *timestep_mc* are stored in columns.
- *timestep_mc*: String Cell array with MC timesteps. If new timesteps are set, values are automatically appended.
- *day_count_convention*: Day count convention of curve. See 'help get_basis' for details (Default: 'act/365')
- *compounding_type*: Compounding type. Can be continuous, discrete or simple. (Default: 'cont')
- *compounding_freq*: Compounding frequency used for discrete compounding. Can be [daily, weekly, monthly, quarterly, semi-annual, annual]. (Default: 'annual')
- *instruments*: Underlying instrument identifiers (Cellstring)
- *weights*: Underlying instruments weights (Numeric vector)
- *discount_curve*: Discount curve (Default: empty string)
- *instr_vol_surfaces*: Required for Options on Baskets only: Underlying instruments volatility surface identifiers (Cellstring)
- *correlation_matrix*: Required for Options on Baskets only: Correlation matrix object of Basket underlyings (Default: empty string)
- *basket_vola_type*: Required for Options on Baskets only: Approximation method for basket volatility calculation [Levy, VCV, Beisser] (Default: 'Levy')

For illustration see the following example: A fund modelled as synthetic instrument with two underlying indizes (MSCI World and Euro Stoxx 50) is set up and the synthetic value (1909.090909) is calculated and retrieved:

```
fprintf('Pricing Synthetic Instrument');
s = Synthetic();
instrument_cell = cell;
instrument_cell(1) = 'EURO_STOXX_50';
instrument_cell(2) = 'MSCIWORLD';
s = s.set('id','TestSynthetic','instruments',instrument_cell);
s = s.set('weights',[1,1],'currency','EUR');
i1 = Index();
i1 = i1.set('id','EURO_STOXX_50','value_base',1000,'scenario_stress',2000);
i2 = Index();
i2 = i2.set('id','MSCIWORLD','value_base',1000);
i2 = i2.set('scenario_stress',2000,'currency','USD');
fx = Index();
fx = fx.set('id','FX_EURUSD','value_base',1.1,'scenario_stress',1.2);
instrument_struct = struct();
instrument_struct(1).id = i1.id;
instrument_struct(1).object = i1;
instrument_struct(2).id = i2.id;
instrument_struct(2).object = i2;
index_struct = struct();
index_struct(1).id = fx.id;
index_struct(1).object = fx;
valuation_date = datenum('31-Mar-2016');
s = s.calc_value(valuation_date,'base',instrument_struct,index_struct);
s.getValue('base')
```

Dependencies of class:



4.12 Surface.help

`object = Surface(id)` [Octarisk Class]

`object = ` *Surface*() [Octarisk Class]

Class for setting up Surface objects.

Surface class is used for specifying IndexVol, IRVol, Stochastic, Prepayment or Dummy Surfaces. A Surface (or Cube) stores two- or three-dimensional values (e.g. term, tenor and/or moneyness dependent volatility values. Surfaces can be shocked with risk factors (e.g. risk factor types RF_VOLA_EQ or RF_VOLA_IR) at any coordinates of the multi-dimensional space in MC or stress scenarios.

This class contains all attributes and methods related to the following Surface types:

- *IndexVol* two-dimensional surface (term vs. moneyness) for setting up Equity volatility values.
- *IRVol* two- or three-dimensional surface (term vs. moneyness) / cube (term vs. tenor vs. moneyness) for setting up Interest rate volatility values.
- *Stochastic* one- or two-dimensional curve / surface to store scenario dependent values (e.g. stochastic cash flow surface with values dependent on date and quantile)
- *Prepayment* two-dimensional prepayment surface with prepayment factors dependent on e.g. interest rate shock and coupon rates.
- *Dummy* Dummy curve for various purposes.

In the following, all methods and attributes are explained and a code example is given.

Methods for Surface object *obj*:

- Surface(*id*) or Surface(): Constructor of a Surface object. *id* is optional and specifies id and name of new object.
- obj.set(*attribute*,*value*): Setter method. Provide pairs of attributes and values. Values are checked for format and constraints.
- obj.get(*attribute*): Getter method. Query the value of specified attribute.
- obj.getValue(*scenario*, *x*, *y*, *z*): Return Surface value at given coordinates according to scenario type. Interpolate surface base value and risk factor shock (only possible after call of method *apply_rf_shocks*)
- Surface.help(*format*,*returnflag*): show this message. Format can be [plain text, html or texinfo]. If empty, defaults to plain text. Returnflag is boolean: True returns documentation string, false (default) returns empty string. [static method]
- obj.apply_rf_shocks(*riskfactor_struct*): Apply risk factor shocks to Surface base values and store the shocks for later use in attribute *shock_struct*. These shocks are then applied to the surface base value with method getValue. The risk factors are taken from the provided structure according to the surface risk factor IDs given by the attribute *riskfactors*.
- Surface.interpolate(*x*, [*y*, [*z*]]): Return Surface base value at given coordinates.

Attributes of Surface objects:

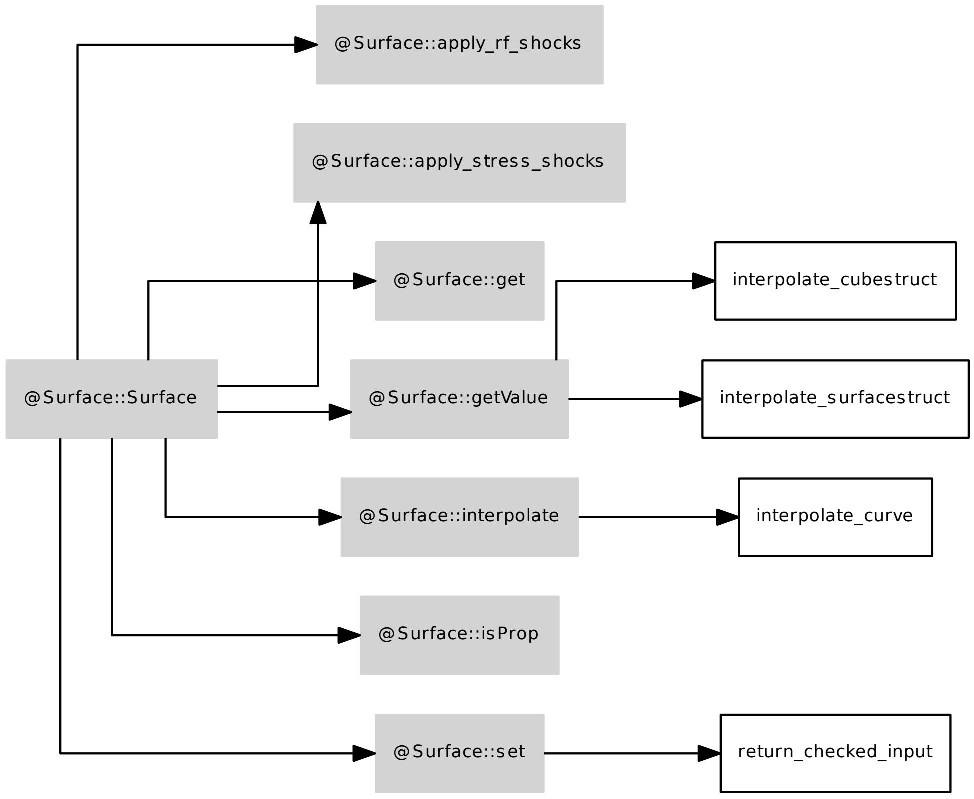
- *id*: Surface id. Has to be unique identifier. Default: empty string.
- *name*: Surface name. Default: empty string.
- *description*: Surface description. Default: empty string.
- *type*: Surface type. Can be [Index, IR, Stochastic, Prepayment, Dummy Surfaces]. Default: Index.

- *day_count_convention*: Day count convention of curve. See 'help get_basis' for details (Default: 'act/365')
- *compounding_type*: Compounding type. Can be continuous, discrete or simple. (Default: 'cont')
- *method_interpolation*: Interpolation method. Can be linear or nearest. Default: linear.
- *compounding_freq*: Compounding frequency used for discrete compounding. Can be [daily, weekly, monthly, quarterly, semi-annual, annual]. (Default: 'annual')
- *values_base*: Base values of Surface.
- *moneyness_type*: Moneyness type. Can be K/S for relative moneyness or K-S for absolute moneyness. (Default: 'K/S').
- *shock_struct*: Structure containing all risk factor shock specifications (e.g. model, risk factor coordinates, shock values and shift type)
- *riskfactors*: Cell specifying all risk factor IDs
- *axis_x*: x-axis coordinates
- *axis_y*: y-axis coordinates
- *axis_z*: z-axis coordinates
- *axis_x_name*: x-axis name
- *axis_y_name*: y-axis name
- *axis_z_name*: z-axis name

For illustration see the following example:

```
disp('Setting up an Index Surface and Risk factor, apply shocks and re-
trieve values:')
r1 = Riskfactor();
r1 = r1.set('id','V1','scenario_stress',[1.0;-0.5], ...
'model','GBM','shift_type',[1;1], ...
'node',730,'node2',1);
riskfactor_struct(1).id = r1.id;
riskfactor_struct(1).object = r1;
v = Surface();
v = v.set('id','V1','axis_x',[365,3650], ...
'axis_x_name','TERM','axis_y',[0.9,1.0,1.1], ...
'axis_y_name','MONEYNESS');
v = v.set('values_base',[0.25,0.36;0.22,0.32;0.26,0.34]);
riskfactor_cell = cell;
riskfactor_cell(1) = 'V1';
v = v.set('type','INDEX','riskfactors',riskfactor_cell);
v = v.apply_rf_shocks(riskfactor_struct);
base_value = v.interpolate(365,0.9)
base_value = v.getValue('base',365,0.9)
stress_value = v.getValue('stress',365,0.9)
```

Dependencies of class:



4.13 Swaption.help

`object = `*Swaption*(`id`) [Octarisk Class]
`object = `*Swaption*() [Octarisk Class]

Class for setting up Swaption objects. Possible underlyings are fixed and floating swap legs. Therefore the following Swaption types are introduced:

- SWAPT_REC: European Receiver Swaption priced by Black-Scholes or Bachelier normal model.
- SWAPT_PAY: European Payer Swaption priced by Black-Scholes or Bachelier normal model.

In the following, all methods and attributes are explained and a code example is given.
Methods for Swaption object *obj*:

- Swaption(*id*) or Swaption(): Constructor of a Swaption object. *id* is optional and specifies id and name of new object.

- obj.set(*attribute*,*value*): Setter method. Provide pairs of attributes and values. Values are checked for format and constraints.
- obj.get(*attribute*): Getter method. Query the value of specified attribute.
- obj.calc_value(*valuation_date*,*scenario*, *discount_curve*, *volatility_surface*, *underlying_fixed_leg*, *underlying_floating_leg*) Calculate the value of Swaptions based on valuation date, scenario type, discount curve, underlying instruments and volatility surface. The pricing model is chosen based on Swaption type and instrument model attributes.
- obj.calc_greeks(*valuation_date*,*scenario*, *discount_curve*, *volatility_surface*, *underlying_fixed_leg*, *underlying_floating_leg*) Calculate numerical sensitivities (the Greeks) for the given Swaption instrument.
- obj.calc_vola_spread(*valuation_date*,*scenario*, *discount_curve*, *volatility_surface*, *underlying_fixed_leg*, *underlying_floating_leg*) Calibrate volatility spread in order to match the Swaption price with the market price. The volatility spread will be used for further pricing.
- obj.getValue(*scenario*): Return Swaption value for given *scenario*. Method inherited from Superclass *Instrument*
- Swaption.help(*format*,*returnflag*): show this message. Format can be [plain text, html or texinfo]. If empty, defaults to plain text. Returnflag is boolean: True returns documentation string, false (default) returns empty string. [static method]

Attributes of Swaption objects:

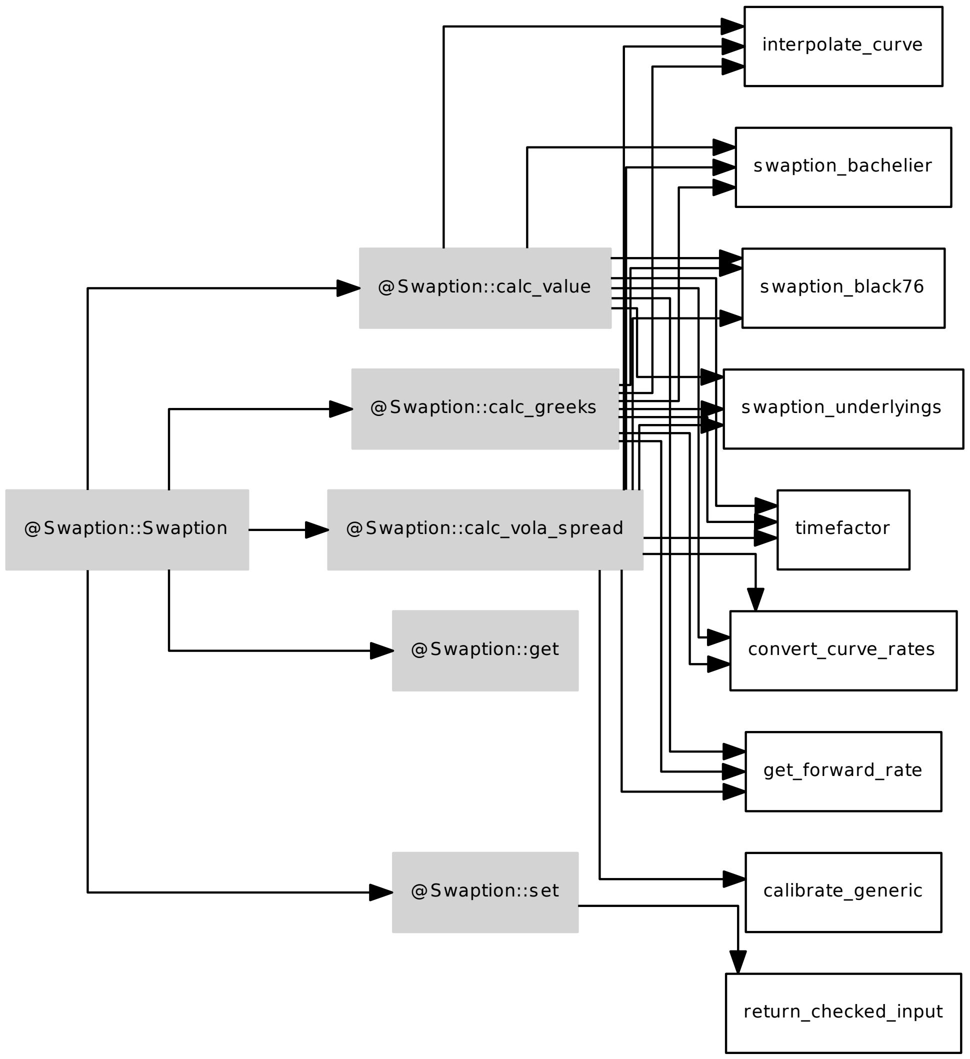
- *id*: Instrument id. Has to be unique identifier. (Default: empty string)
- *name*: Instrument name. (Default: empty string)
- *description*: Instrument description. (Default: empty string)
- *value_base*: Base value of instrument of type real numeric. (Default: 0.0)
- *currency*: Currency of instrument of type string. (Default: 'EUR') During instrument valuation and aggregation, FX conversion takes place if corresponding FX rate is available.
- *asset_class*: Asset class of instrument. (Default: 'derivative')
- *type*: Type of instrument, specific for class. Set to 'Swaption'.
- *value_stress*: Line vector with instrument stress scenario values.
- *value_mc*: Line vector with instrument scenario values. MC values for several *timestep_mc* are stored in columns.
- *timestep_mc*: String Cell array with MC timesteps. If new timesteps are set, values are automatically appended.
- *maturity_date*: Maturity date of Swaption (date in format DD-MMM-YYYY)
- *effective_date*: Effective date of Swaption (date in format DD-MMM-YYYY)
- *day_count_convention*: Day count convention of curve. See 'help get_basis' for details (Default: 'act/365')
- *compounding_type*: Compounding type. Can be continuous, discrete or simple. (Default: 'cont')
- *compounding_freq*: Compounding frequency used for discrete compounding. Can be [daily, weekly, monthly, quarterly, semi-annual, annual]. (Default: 'annual')

- *spread*: Interest rate spread used in calculating risk free interest rate. Default: 0.0;
- *discount_curve*: ID of discount curve. Default: empty string
- *underlying*: ID of underlying curve object for extracting forward rates. Default: empty string
- *vola_surface*: ID of volatility surface. Default: empty string
- *vola_sensi*: Sensitivity scaling factor for volatility. Default: 1
- *strike*: Strike rate of Swaption. Default: 100
- *spot*: Spot rate of underlying reference curve. Only used, if underlying is risk factor.
- *multiplier*: Multiplier of Swaption. Resulting Swaption price is scales by this multiplier. Default: 100
- *model*: Pricing model for Swaptions. Can be ['black','normal']. Default: 'black'
- *tenor*: Tenor of swaption contract.
- *no_payments*: Number of payments of swaption contract.
- *use_underlyings*: Boolean flag: if true, underlying swap values of fixed and floating legs are used for calculation of swaption spot price spot price (Default: 'false')
- *und_fixed_leg*: ID of underlying fixed swap leg. Object has to be a Bond(). Default: empty string
- *und_floating_leg*: ID of underlying floating swap leg. Object has to be a Bond(). Default: empty string
- *theo_delta*: Sensitivity to changes in underlying's price. Calculate by method *calc_greeks*.
- *theo_gamma*: Sensitivity to changes in changes of underlying's price. Calculate by method *calc_greeks*.
- *theo_vega*: Sensitivity to changes in volatility. Calculate by method *calc_greeks*.
- *theo_theta*: Sensitivity to changes in remaining days to maturity. Calculate by method *calc_greeks*.
- *theo_rho*: Sensitivity to changes in risk free rate. Calculate by method *calc_greeks*.
- *theo_omega*: Specified as *theo_delta* scaled by underlying value over Swaption base value. Calculate by method *calc_greeks*.

For illustration see the following example: A normal payer swaption with maturity in 20 years with underlying swaps starting in 20 years for 10 years, a volatility surface and a discount curve are priced. The resulting Swaption value (642.6867193851) is retrieved:

```
disp('Pricing Payer Swaption with underlyings (Normal Model)')
r = Curve();
r = r.set('id','EUR-SWAP-NOFLOOR','nodes', ...
[7300,7665,8030,8395,8760,9125,9490,9855,10220,10585,10900], ...
'rates_base',[0.02,0.01,0.0075,0.005,0.0025,-0.001, ...
-0.002,-0.003,-0.005,-0.0075,-0.01], ...
'method_interpolation','linear');
fix = Bond();
fix = fix.set('Name','SWAP_FIXED','coupon_rate',0.045, ...
'value_base',100,'coupon_generation_method','forward', ...
'sub_type','SWAP_FIXED');
fix = fix.set('maturity_date','24-Mar-2046','notional',100, ...
'compounding_type','simple','issue_date','26-Mar-2036', ...
'term',365,'notional_at_end',0);
fix = fix.rollout('base','31-Mar-2016');
fix = fix.rollout('stress','31-Mar-2016');
fix = fix.calc_value('31-Mar-2016','base',r);
fix = fix.calc_value('31-Mar-2016','stress',r);
float = Bond();
float = float.set('Name','SWAP_FLOAT','coupon_rate',0.00,'value_base',100, ...
'coupon_generation_method','forward','last_reset_rate',-0.000, ...
'sub_type','SWAP_FLOATING','spread',0.00);
float = float.set('maturity_date','24-Mar-2046','notional',100, ...
'compounding_type','simple','issue_date','26-Mar-2036', ...
'term',365,'notional_at_end',0);
float = float.rollout('base',r,'31-Mar-2016');
float = float.rollout('stress',r,'31-Mar-2016');
float = float.calc_value('30-Sep-2016','base',r);
float = float.calc_value('30-Sep-2016','stress',r);
v = Surface();
v = v.set('axis_x',30,'axis_x_name','TENOR', ...
'axis_y',45,'axis_y_name','TERM','axis_z',1.0,'axis_z_name','MONEYNESS');
v = v.set('values_base',0.376563388);
v = v.set('type','IRVol');
s = Swaption();
s = s.set('maturity_date','26-Mar-2036','effective_date','31-Mar-2016');
s = s.set('strike',0.045,'multiplier',1,'sub_type', 'SWAPT_PAY', ...
'model','normal','tenor',10);
s = s.set('und_fixed_leg','SWAP_FIXED','und_floating_leg','SWAP_FLOAT', ...
'use_underlyings',true);
s = s.calc_value('31-Mar-2016','base',r,v,fix,float);
s.getValue('base')
```

Dependencies of class:



4.14 Stochastic.help

`object` = *Stochastic*(`id`) [Octarisk Class]
`object` = *Stochastic*() [Octarisk Class]

Class for setting up Stochastic objects. A Stochastic instrument uses a risk factor with random variables (either uniform, normal or t-distributed) to draw values from a 1D Surface. The surface has exactly one value per given quantile [0,1]. This instrument type can be used to pre-calculate values in another risk system for a given risk factor distribution.

- STOCHASTIC: Stochastic instrument type is the default value.

In the following, all methods and attributes are explained and a code example is given.

Methods for Stochastic object *obj*:

- Stochastic(*id*) or Stochastic(): Constructor of a Stochastic object. *id* is optional and specifies id and name of new object.
- obj.set(*attribute*,*value*): Setter method. Provide pairs of attributes and values. Values are checked for format and constraints.
- obj.get(*attribute*): Getter method. Query the value of specified attribute.
- obj.calc_value(*valuation_date*,*scenario*, *riskfactor*, *surface*) Calculate the value of Stochastic instruments. Quantile values from a 1-dimensional surface are drawn based on (transformed) risk factor shocks.
- obj.getValue(*scenario*): Return Stochastic value for given *scenario*. Method inherited from Superclass *Instrument*
- Stochastic.help(*format*,*returnflag*): show this message. Format can be [plain text, html or texinfo]. If empty, defaults to plain text. Returnflag is boolean: True returns documentation string, false (default) returns empty string. [static method]

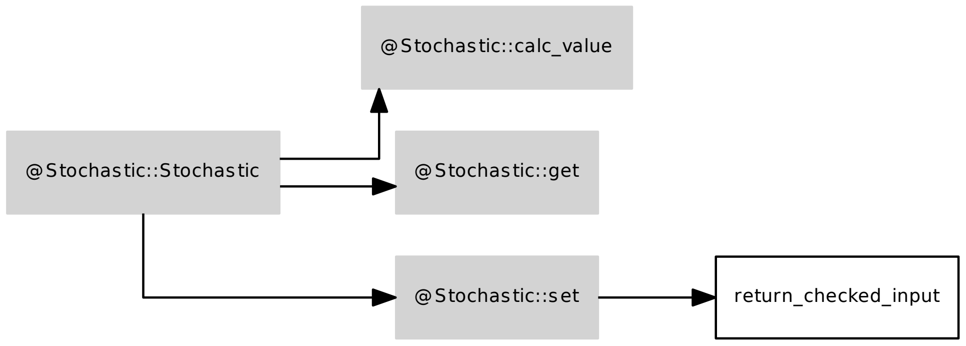
Attributes of Stochastic objects:

- *id*: Instrument id. Has to be unique identifier. (Default: empty string)
- *name*: Instrument name. (Default: empty string)
- *description*: Instrument description. (Default: empty string)
- *value_base*: Base value of instrument of type real numeric. (Default: 0.0)
- *currency*: Currency of instrument of type string. (Default: 'EUR') During instrument valuation and aggregation, FX conversion takes place if corresponding FX rate is available.
- *asset_class*: Asset class of instrument. (Default: 'stochastic')
- *type*: Type of instrument, specific for class. Set to 'Stochastic'.
- *value_stress*: Line vector with instrument stress scenario values.
- *value_mc*: Line vector with instrument scenario values. MC values for several *timestep_mc* are stored in columns.
- *timestep_mc*: String Cell array with MC timesteps. If new timesteps are set, values are automatically appended.
- *quantile_base*: Base quantile of stochastic curve. (Default: 0.5)
- *stochastic_riskfactor*: underlying risk factor objects. Shocks are transformed according to *stochastic_rf_type*
- *stochastic_curve*: underlying 1-dim surface with values per quantile.
- *stochastic_rf_type*: Risk factor transformation. Type can be ['normal','t','uniform'] (Default: 'normal')
- *t_degree_freedom*: degrees of freedom for t distribution (Default: 120)

For illustration see the following example: A stochastic value object is generated. Quantile values are given by a 1-dim volatility surface. The resulting Stress value ([95;100;105]) is retrieved:

```
disp('Pricing Pricing Stochastic Value Object')
r = Riskfactor();
r = r.set('value_base',0.5,'scenario_stress',[0.3;0.50;0.7],'model','BM');
value_x = 0;
value_quantile = [0.1,0.5,0.9];
value_matrix = [90;100;110];
v = Surface();
v = v.set('axis_x',value_x,'axis_x_name','DATE', ...
'axis_y',value_quantile,'axis_y_name','QUANTILE');
v = v.set('values_base',value_matrix);
v = v.set('type','STOCHASTIC');
s = Stochastic();
s = s.set('sub_type','STOCHASTIC','stochastic_rf_type','uniform', ...
't_degree_freedom',10);
s = s.calc_value('31-Mar-2016','base',r,v);
s = s.calc_value('31-Mar-2016','stress',r,v);
stress_value = s.getValue('stress')
```

Dependencies of class:



4.15 CapFloor.help

object = *CapFloor*(*id*) [Octarisk Class]

object = *CapFloor*() [Octarisk Class]

Class for setting up CapFloor objects. Plain vanilla caps and floors (consisting of caplet and floorlets) can be based on interest rates or inflation rates. Cash flows are generated according to different models (Black, Normal, Analytic) and subsequently discounted to calculate the CapFloor value.

- CAP: Plain Vanilla interest rate cap. Valuation model either ['black','normal','analytic']

- FLOOR: Plain Vanilla interest rate floor. Valuation model either ['black','normal','analytic']
- CAP_CMS: CMS interest rate cap. Valuation model either ['black','normal','analytic']
- FLOOR_CMS: CMS interest rate floor. Valuation model either ['black','normal','analytic']
- CAP_INFL: Cap on inflation expectation rates (derived from inflation index values). Analytical model only (cash flow value based on difference of inflation rate and strike rate)
- FLOOR_INFL: Floor on inflation rates (derived from inflation index values). Analytical model only (cash flow value based on difference of inflation rate and strike rate)

In the following, all methods and attributes are explained and a code example is given.

Methods for CapFloor object *obj*:

- CapFloor(*id*) or CapFloor(): Constructor of a CapFloor object. *id* is optional and specifies id and name of new object.
- obj.set(*attribute*,*value*): Setter method. Provide pairs of attributes and values. Values are checked for format and constraints.
- obj.get(*attribute*): Getter method. Query the value of specified attribute.
- obj.calc_value(*valuation_date*,*scenario*, *discount_curve*): Calculate the net present value of cash flows of Caps and Floors.
- obj.rollout(*valuation_date*,*scenario*, *reference_curve*, *vola_surface*): used for (CMS) Caps and Floors
- obj.rollout(*valuation_date*,*scenario*, *inflation_exp_rates*, *historical_inflation*, *consumer_price_index*): used for Inflation Caps and Floors Cash flow rollout for (Inflation) Caps and Floors.
- obj.calc_sensitivity(*valuation_date*,*scenario*, *reference_curve*, *vola_surface*, *discount_curve*) Calculate numerical sensitivities (durations, vega, theta) for the given CapFloor instrument.
- obj.calc_vola_spread(*valuation_date*,*scenario*, *discount_curve*, *volatility_surface*) Calibrate volatility spread in order to match the CapFloor price with the market price. The volatility spread will be used for further pricing.
- obj.getValue(*scenario*): Return CapFloor value for given *scenario*. Method inherited from Superclass *Instrument*
- CapFloor.help(*format*,*returnflag*): show this message. Format can be [plain text, html or texinfo]. If empty, defaults to plain text. Returnflag is boolean: True returns documentation string, false (default) returns empty string. [static method]

Attributes of CapFloor objects:

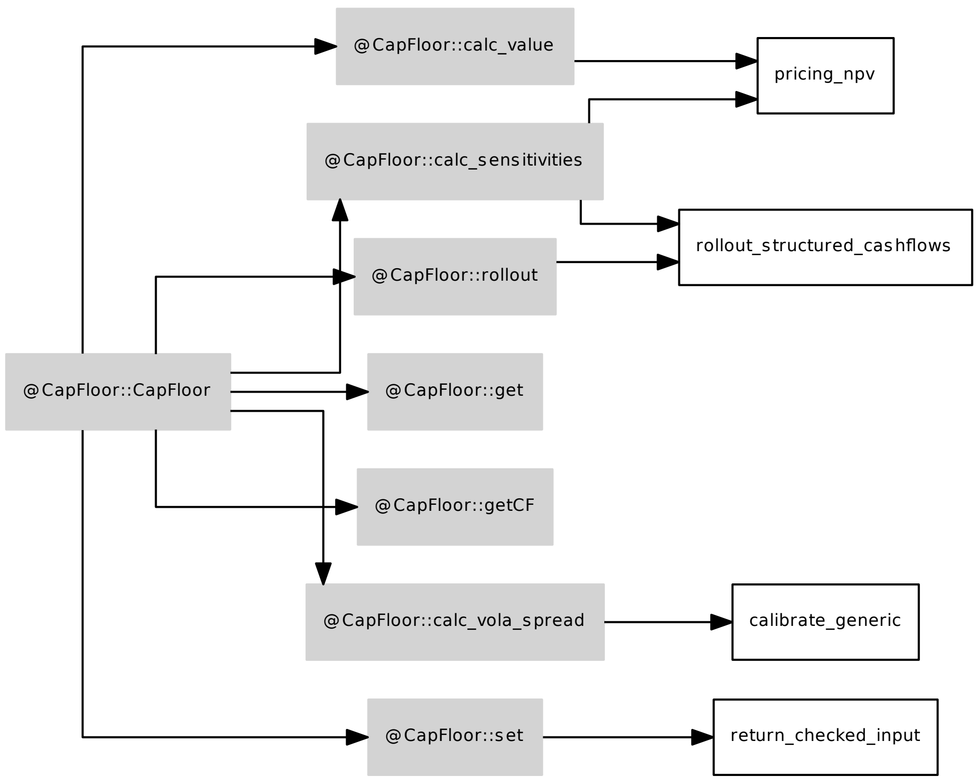
- *id*: Instrument id. Has to be unique identifier. (Default: empty string)
- *name*: Instrument name. (Default: empty string)
- *description*: Instrument description. (Default: empty string)
- *value_base*: Base value of instrument of type real numeric. (Default: 0.0)

- *currency*: Currency of instrument of type string. (Default: 'EUR') During instrument valuation and aggregation, FX conversion takes place if corresponding FX rate is available.
- *asset_class*: Asset class of instrument. (Default: 'derivative')
- *type*: Type of instrument, specific for class. Set to 'CapFloor'.
- *value_stress*: Line vector with instrument stress scenario values.
- *value_mc*: Line vector with instrument scenario values. MC values for several *timestep_mc* are stored in columns.
- *timestep_mc*: String Cell array with MC timesteps. If new timesteps are set, values are automatically appended.
- *model*: Valuation model for (CMS) Caps and Floors can be either ['black','normal','analytic']. Inflation Caps and Floors are valuated by analytical model only.

For illustration see the following example: A 2 year Cap starting in 3 years is priced with Black model. The resulting Cap value (137.0063959386) and volatility spread (-0.0256826614604929)is retrieved:

```
disp('Pricing Cap Object with Black Model')
cap = CapFloor();
cap = cap.set('id','TEST_CAP','name','TEST_CAP','issue_date','30-Dec-
2018', ...
'maturity_date','29-Dec-2020','compounding_type','simple');
cap = cap.set('term',365,'term_unit','days','notional',10000, ...
'coupon_generation_method','forward','notional_at_start',0, ...
'notional_at_end',0);
cap = cap.set('strike',0.005,'model','Black','last_reset_rate',0.0, ...
'day_count_convention','act/365','sub_type','CAP');
c = Curve();
c = c.set('id','IR_EUR','nodes',[30,1095,1460],'rates_base',[0.01,0.01,0.01], ...
'method_interpolation','linear');
v = Surface();
v = v.set('axis_x',365,'axis_x_name','TENOR','axis_y',90, ...
'axis_y_name','TERM','axis_z',1.0,'axis_z_name','MONEYNESS');
v = v.set('values_base',0.8);
v = v.set('type','IRVol');
cap = cap.rollout('31-Dec-2015','base',c,v);
cap = cap.calc_value('31-Dec-2015','base',c);
base_value = cap.getValue('base')
cap = cap.set('value_base',135.000);
cap = cap.calc_vola_spread('31-Dec-2015',c,v);
cap = cap.rollout('31-Dec-2015','base',c,v);
cap = cap.calc_value('31-Dec-2015','base',c);
vola_spread = cap.vola_spread
```

Dependencies of class:



4.16 Bond.help

object = *Bond*(*id*) [Octarisk Class]
object = *Bond*() [Octarisk Class]

Class for setting up various Bond objects. Cash flows are generated specific for each Bond sub type and subsequently discounted to calculate the Bond value. All bonds can have embedded options (Option pricing according to Hull-White model).

- FRB: Fixed Rate Bond
- FRN: Floating Rate Note: Calculate CF Values based on forward rates of a given reference curve.
- ZCB: Zero Coupon Bond
- ILB: Inflation Linked Bond
- CDS: Credit Default Swaps
- CASHFLOW: Cash flow instruments. Custom cash flow dates and values are discounted.
- SWAP_FIXED: Swap fixed leg

- SWAP_FLOATING: Swap floating leg
- FRN_CMS_SPECIAL: Special type floating rate notes (capitalized, average, min, max) based on CMS rates
- CMS_FLOATING: Floating leg based on CMS rates
- FRA: Forward Rate Agreement
- FVA: Forward Volatility Agreement
- FRN_FWD_SPECIAL: Averaging FRN: Average forward or historical rates of cms_sliding period
- STOCHASTICCF: Stochastic cash flow instrument (cash flows values are derived from an empirical cash flow distribution)

In the following, all methods and attributes are explained and a code example is given.
Methods for Bond object *obj*:

- Bond(*id*) or Bond(): Constructor of a Bond object. *id* is optional and specifies id and name of new object.
- obj.set(*attribute*,*value*): Setter method. Provide pairs of attributes and values. Values are checked for format and constraints.
- obj.get(*attribute*): Getter method. Query the value of specified attribute.
- obj.calc_value(*valuation_date*,*scenario*, *discount_curve*)
- obj.calc_value(*valuation_date*,*scenario*, *discount_curve*, *call_schedule*, *put_schedule*): Calculate the net present value of cash flows of Bonds (including pricing of embedded options)
- obj.rollout(*scenario*, *valuation_date*): used for FRB and CASHFLOW instruments
- obj.rollout(*scenario*, *valuation_date*, *reference_curve*, *vola_surface*): used for CMS_FLOATING or FRN_SPECIAL
- obj.rollout(*scenario*, *reference_curve*, *valuation_date*, *vola_surface*): used for FRN, FRA, FVA, SWAP_FLOATING
- obj.rollout(*scenario*,*valuation_date*, *psa_curve*, *psa_factor_surface*, *ir_shock_curve*): used for FAB with prepayments
- obj.rollout(*scenario*,*valuation_date*, *inflation_expectation_curve*, *historical_rates*, *consumer_price_index*): used for ILB
- obj.rollout(*scenario*,*valuation_date*, *riskfactor*, *cashflow_surface*): used for Stochastic CF instruments
- obj.rollout(*scenario*,*valuation_date*, *hazard_curve*, *reference_asset*,*reference_curve*): used for CDS Cash flow rollout for Bonds
- obj.calc_sensitivities(*valuation_date*,*discount_curve*, *reference_curve*) Calculate analytical and numerical sensitivities for the given Bond instrument.
- obj.calc_key_rates(*valuation_date*,*discount_curve*) Calculate key rate sensitivities for the given Bond instrument.
- obj.calc_spread_over_yield(*valuation_date*,*scenario*, *discount_curve*) or

- obj.calc_spread_over_yield(*valuation_date*,*scenario*, *discount_curve*, *call_schedule*, *put_schedule*) Calibrate spread over yield in order to match the Bond price with the market price. The interest rate spread will be used for further pricing.
- obj.calc_yield_to_mat(*valuation_date*): Calculate yield to maturity for given cash flow structure.
- obj.getValue(*scenario*): Return Bond value for given *scenario*. Method inherited from Superclass *Instrument*
- Bond.help(*format*,*returnflag*): show this message. Format can be [plain text, html or texinfo]. If empty, defaults to plain text. Returnflag is boolean: True returns documentation string, false (default) returns empty string. [static method]

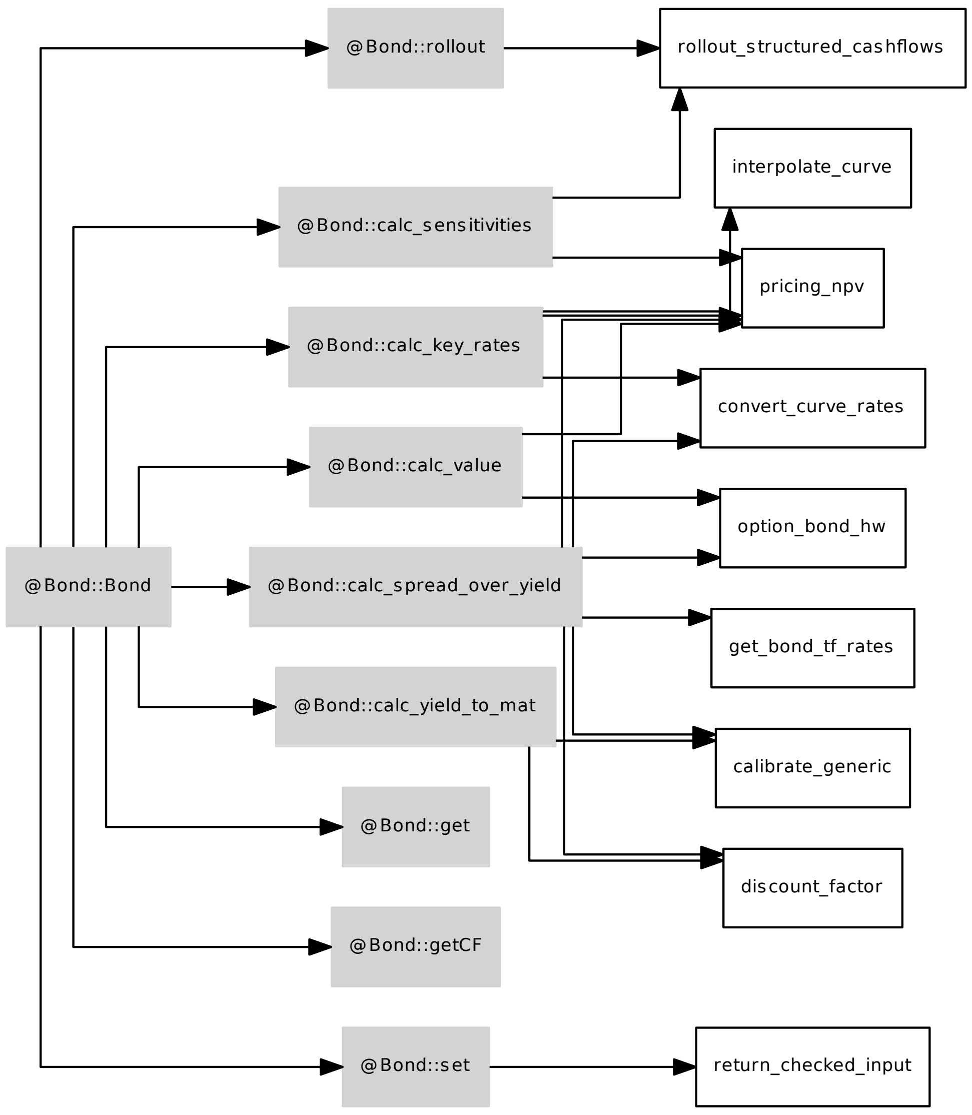
Attributes of Bond objects:

- *id*: Instrument id. Has to be unique identifier. (Default: empty string)
- *name*: Instrument name. (Default: empty string)
- *description*: Instrument description. (Default: empty string)
- *value_base*: Base value of instrument of type real numeric. (Default: 0.0)
- *currency*: Currency of instrument of type string. (Default: 'EUR') During instrument valuation and aggregation, FX conversion takes place if corresponding FX rate is available.
- *asset_class*: Asset class of instrument. (Default: 'derivative')
- *type*: Type of instrument, specific for class. Set to 'Bond'.
- *value_stress*: Line vector with instrument stress scenario values.
- *value_mc*: Line vector with instrument scenario values. MC values for several *timestep_mc* are stored in columns.
- *timestep_mc*: String Cell array with MC timesteps. If new timesteps are set, values are automatically appended.

For illustration see the following example: A 9 month floating rate note instrument will be calibrated and priced. The resulting spread over yield value (0.00398785481397732), base value (99.7917725092950) and effective duration (3.93109370316470e-005)is retrieved:

```
disp('Pricing Floating Rate Bond Object and calculating sensitivities')
b = Bond();
b = b.set('Name','Test_FRN','coupon_rate',0.00,'value_base',99.7527, ...
'coupon_generation_method','backward','compounding_type','simple');
b = b.set('maturity_date','30-Mar-2017','notional',100, ...
'compounding_type','simple','issue_date','21-Apr-2011');
b = b.set('term',3,'term_unit','months','last_reset_rate',-0.0024,'sub_Type','FRN
r = Curve();
r = r.set('id','REF_IR_EUR','nodes',[30,91,365,730], ...
'rates_base',[0.0001002740,0.0001002740,0.0001001390,0.0001000690], ...
'method_interpolation','linear');
b = b.rollout('base',r,'30-Jun-2016');
c = Curve();
c = c.set('id','IR_EUR','nodes',[30,90,180,365,730], ...
'rates_base',[0.0019002740,0.0019002740,0.0019002301,0.0019001390,0.001900069], .
'method_interpolation','linear');
b = b.set('clean_value_base',99.7527,'spread',0.003);
b = b.calc_spread_over_yield('30-Jun-2016',c);
b.get('soy')
b = b.calc_value('30-Jun-2016','base',c);
b.getValue('base')
b = b.calc_sensitivities('30-Jun-2016',c,r);
b.get('eff_duration')
```

Dependencies of class:



5 Octave Functions and Scripts

In the following sections you find the Octave function texinfo.

5.1 adapt_matlab

Delete unneccessary scripts

5.2 addtodatefinancial

[*newdatenum newdatevec*] = addtodatefinancial(*valdate*, *arg1*, *arg2*, *arg3*) [Function File]

Add or subtract given years, months or days to a given input date. End of month of input date will be preserved if no days are added. Both datenum and datevec format are returned. Explicit specification of years, months, days:

- *valdate*: start date for date manipulation
- *arg1*: years to add or subtract
- *arg2*: months to add or subtract (optional)
- *arg3*: days to add or subtract (optional)

Implicit specification of value and unit:

- *valdate*: start date for date manipulation
- *arg1*: value to add or subtract
- *arg2*: unit (days, months, years)

Single date input possible only. Example call:

```
 [newdatenum newdatevec] = addtodatefinancial('31-Mar-2016', 1, 'years')
newdatenum =  736785
newdatevec = [2017 3 31]
 [newdatenum newdatevec] = addtodatefinancial('31-Mar-2016', -1, -6, -
4)
newdatenum =  735869
newdatevec = [2014 9 27]
```

See also: addtodate.

5.3 aggregate_positions

[*position_struct position_failed_cell portfolio_value portfolio_shock*] = aggregate_positions (*position_struct*, *position_failed_cell*, *instrument_struct*, *index_struct*, *scennumber*, *scen_set*, *fund_currency*, *port_id*, *printflag*) [Function File]

Aggregate position and portfolio base, stress and MC scenarios shocks. Instrument MC values are aggregated and converted to fund currency based on position information (quantity and position ID referencing instrument ID).

Return total portfolio base, stress or MC values and position struct with new keys \'basevalue\', \'stresstests\' or \'mc_scenarios\' according to provided scenario set.

Input:

- *position_struct*: structure with position definitions (quantity and ID)
- *position_failed_cell*: failing position ids are stored in this cell
- *instrument_struct*: structure with instrument objects
- *index_struct*: structure with FX objects
- *scennumber*: number of stress or MC scenarios
- *scen_set*: Scenario set used for aggregation (e.g. base, stress, 250d or 1d)
- *fund_currency*: FX conversion of instrument values into fund currency
- *port_id*: Portfolio ID
- *printflag*: Boolean flag: true = print information about aggregation to stdout

See also: aggregate_position_base.

5.4 any2str

`[output type] = any2str(value)` [Function File]

Convert input value into string. Therefore a type dependent conversion is performed. One output string (a one-liner!) and the input type is returned. Conversion is supported for scalars, matrizes up to three dimensions, cells, boolean values and structs.

5.5 betainc_vec

`betainc_vec (x, a, b)`
`betainc_vec (x, a, b, tail)`

Compute the incomplete beta function ratio.
This function shadows the core function betainc. Performance improvement due to vectorized C++. Support for single precision was removed.
This is defined as

$$I_x(a,b) = \frac{1}{B(a,b)} \int_0^x t^{a-1}(1-t)^{b-1} dt$$

with *x* real in [0,1], *a* and *b* real and strictly positive. If one of the input has more than one components, then the others must be scalar or of compatible dimensions.
By default or if *tail* is `"lower"` the incomplete beta function ratio integrated from 0 to *x* is computed. If *tail* is `"upper"` then the complementary function integrated from *x* to 1 is calculated. The two choices are related as
betainc (*x*, *a*, *b*, `"lower"`) = 1 - betainc (*x*, *a*, *b*, `"upper"`).
Reference
A. Cuyt, V. Brevik Petersen, B. Verdonk, H. Waadeland, W.B. Jones *Handbook of Continued Fractions for Special Functions*, ch. 18.

See also: beta, betaincinv, betaln.

5.6 betainv_vec

`betainv_vec (x, a, b)`

For each element of *x*, compute the quantile (the inverse of the CDF) at *x* of the Beta distribution with parameters *a* and *b*. Performance improvement: Calling vectorized version of betacdf (batainc_vec). Supporting full double precision.

5.7 calcConvexityAdjustment

`[adj_rate adj] = calcConvexityAdjustment (valuation_date, instrument, r, sigma, t1, t2)` [Function File]

Return convexity adjustment to a given forward rate with specified forward start and end dates and forward volatility. For CMS Rate adjustments use function get_cms_rate.

Implementation of log-normal convexity adjustment according to H.P. Deutsch, Derivate und Interne Modelle, 4th Edition, Section 14.5 Convexity Adjustment. Normal model convexity adjustment just uses absolute volatility (sigma) instead of relative volatility (sigma*r). Input and output variables:

- *valuation_date*: valuation date [required]
- *instrument*: instrument struct or object (with model, basis) [required]
- *r*: forward rate [required]
- *sigma*: forward volatility (act/365 continuous) [required]
- *t1*: forward start date [required]
- *t2*: forward end date [required]
- *adj_rate*: OUTPUT: adjusted forward rate
- *adj*: OUTPUT: adjustment only

See also: timefactor.

5.8 calibrate_evt_gpd

`[chi sigma u] = calibrate_evt_gpd(v)` [Function File]

Calibrate sorted losses of historic or MC portfolio values to a generalized pareto distribution and returns chi, sigma and u as parameters for further VAR and ES calculation.

Implementation according to *Risk Management and Financial Institutions* by John C. Hull, 4th edition, Wiley 2015, section 13.6, page 292ff.

Explanation of Parameters:

- *v*: INPUT: sorted profit and loss distribution of all required tail events (1xN vector)
- *chi*: OUTPUT: Generalized Pareto distribution: shape parameter (scalar)
- *sigma*: OUTPUT: scale parameter (scalar)
- *u*: OUTPUT: location parameter(scalar)

5.9 calibrate_generic

`[calibrated_value retcode] = calibrate_generic(objf, x0, lb, ub)` [Function File]

Calibrate a given objective function according to start parameter and bounds. This function calls the generic optimizer fmincon.

See also: fmincon.

5.10 compile_oct_files

`compile_oct_files (path_octarisk)` [Function File]

Compile all .cc files in folder *path_octarisk*/oct_files and move successfully compiled .oct files to top folder.

5.11 convert_curve_rates

`[rate_target conversion_type] = convert_curve_rates (valuation_date, node, rate_origin, comp_type_origin, comp_freq_origin, dcc_basis_origin, comp_type_target, comp_freq_target, dcc_basis_target)` [Function File]

Convert a given interest rate from one compounding type, frequency and day count convention (dcc) into another type, frequency and dcc.

The following conversion formulas are applied: (the timefactor is depending on day count convention and days between valuation_date and valuation_date + node)). Convert

```
 from CONT ->   SMP:    (exp(rate_origin .* timefactor_origin) -1) ...
./ timefactor_target
 from SMP ->    CONT:   ln(1 + rate_origin .* timefactor_origin) ...
./ timefactor_target
 from DISC ->   CONT:   ln(1 + rate_origin./ comp_freq_origin) ...
.* (timefactor_origin .* comp_freq_origin) ./ timefactor_target
 from CONT ->   DISC:   (exp(rate_origin .* timefactor_origin ...
./ (comp_freq_target .* timefactor_target)) - 1 ) .* comp_freq_target
 from SMP ->    DISC:   ( (1 + rate_origin .* timefactor_origin) ...
.^(1./( comp_freq_target .* timefactor_target)) -1 ) .* comp_freq_target
 from DISC ->   SMP:    ( (1 + rate_origin ./ comp_freq_origin ) ...
.^(comp_freq_origin .* timefactor_origin) -1 ) ./ timefactor_target
 from CONT ->   CONT:   rate_origin .* timefactor_origin ./ timefactor_target
 from SMP ->    SMP:    rate_origin .* timefactor_origin ./ timefactor_target
 from DISC ->   DISC:   ( (1 + rate_origin ./ comp_freq_origin) ...
.^((comp_freq_origin .* timefactor_origin) ./ ( comp_freq_target ...
.* timefactor_target)) -1 ) .* comp_freq_target
```

Please note: compounding_freq is only relevant for compounding type DISCRETE. Otherwise it will be neglected. During object invocation, a default value for compounding_freq is set, even it is not required. Example call:

```
0.006084365 = convert_curve_rates(datenum('31-Dec-2015'),643,0.0060519888,'cont'
```

Input and output variables:

- *valuation_date*: base date used in timefactor calculation (datestr or datenum)
- *node*: number of days until second date used in timefactor calculation (scalar)
- *rate_origin*: interest rate between first and second date (scalar)
- *comp_type_origin*: compounding type of target rate: [simple, simp, disc, discrete, cont, continuous] (string)
- *comp_freq_origin*: compounding frequency of target rate: 1,2,4,12,52,365 or [daily,weekly,monthly,quarter,semi-annual,annual] (scalar or string)
- *dcc_basis_origin*: day-count basis of target rate(scalar)
- *comp_type_target*: compounding type of target rate: [simple, simp, disc, discrete, cont, continuous] (string)
- *comp_freq_target*: compounding frequency of target rate: 1,2,4,12,52,365 or [daily,weekly,monthly,quarter,semi-annual,annual] (scalar or string)
- *dcc_basis_target*: day-count basis of target rate(scalar)
- *rate_target*: OUTPUT: converted interest rate
- *conversion_type*: OUTPUT: conversion type from x to y

See also: timefactor.

5.12 correct_correlation_matrix

`[A_scaled pos_sem_def_bool] = correct_correlation_matrix(M)` [Function File]

Return a positive semi-definite matrix *A_scaled* to a given input matrix *M*. This function tests for indefiniteness of the input matrix and eventuallry adjusts negative Eivenvalues to 0 or slightly positive values via some iteration steps.
Reference: 'Implementing Value at Risk', Best, Philip W., 1998.

5.13 discount_factor

`df = discount_factor (d1, d2, rate, comp_type, basis, comp_freq)` [Function File]

Compute the discount factor for a specific time period, compounding type, day count basis and compounding frequency.

Input and output variables:

- *d1*: number of days until first date (scalar)
- *d2*: number of days until second date (scalar)
- *rate*: interest rate between first and second date (scalar)
- *comp_type*: compounding type: [simple, simp, disc, discrete, cont, continuous] (string)

- *basis*: day-count basis (scalar or string)
- *comp_freq*: 1,2,4,12,52,365 or [daily,weekly,monthly, quarter,semi-annual,annual] (scalar or string)
- *df*: OUTPUT: discount factor (scalar)

See also: timefactor.

5.14 doc_instrument

`= doc_instrument ()` [Function File]

This script contains all integration test cases for Octarisk's instruments. This script if part of the integration test suite.

5.15 doc_riskfactor

`object` `= Riskfactor ()` [Function File]

`object` `= Riskfactor (`*`name`*`, `*`id`*`, `*`type`*`, `*`description`*`, `*`model`*`, `*`parameters`*`)` [Function File]

Construct risk factor object. Riskfactor Class Inputs:

- *name* (string): Name of object
- *id* (string): Id of object
- *type* (string): risk factor type
- *description* (string): Description of object
- *model* (string): statistical model in list [GBM,BM,SRD,OU]
- *parameters* (vector): vector with values [mean,std,skew,kurt, ... start_value,mr_level,mr_rate,node,rate]

If no input arguments are provided, a dummy IR risk factor object is generated.
The constructor of the risk factor class constructs an object with the following properties:
Class properties:

- name: Name of object
- id: Id of object
- description: Description of object
- type: risk factor type
- model: risk factor model
- mean: first moment of risk factor distribution
- std: second moment of risk factor distribution
- skew: third moment of risk factor distribution
- kurt: fourth moment of risk factor distribution
- start_value: Actual spot value of object
- mr_level: In case of mean reverting model this is the mean reversion level
- mr_rate: In case of mean reverting model this is the mean reversion rate
- node: In case of a interest rate or spread risk factor this is the term node

- rate: In case of a interest rate or spread risk factor this is the term rate at the node
- scenario_stress: Vector with values of stress scenarios
- scenario_mc: Matrix with risk factor scenario values (values per timestep per column)
- timestep_mc: MC timestep per column (cell string)

property_value = Riskfactor.getValue((*base,stress,mc_timestep*),'abs') Riskfactor Method getValue
Return the value for a risk factor object. Specify the desired return values with a property parameter. If the second argument abs is set, the absolut scenario value is calculated from scenario shocks and the risk factor start value.
Timestep properties:

- base: return base value
- stress: return stress values
- any regular MC timestep (e.g. '1d'): return scenario (shock) values at MC timestep

property_value = Riskfactor.get (property) *object* = Riskfactor.set (property, value) Riskfactor Methods get / set
Get / set methods for retrieving or setting risk factor properties.

See also: Instrument.

5.16 estimate_parameter

`estimate_parameter()` [Function File]

Calculate statistics for historic time series. This script is not used in octarisk process, but it should simplify the parameter estimation for input parameters required for risk factor definitions.
The time series file (columns: risk factors, rows: historic values) will be generated by an python script (to be done).
The following statistic parameter are calculated: mean, volatility, skewness, kurtosis for simple, geometric and lognormal distributed value. Moreover mean reversion rates and levels are calculated assuming ergodic and regression methods.

5.17 fmincon

`[x obj info iter nf lambda] = fmincon(objf, x0, A, b, Aeq, beq, lb, ub)` [Function File]

Wrap basic functionality of Matlab's solver fmincon to Octave's sqp.
Non-linear constraint functions provided by fmincon's function handle `nonlincon` are NOT processed.
Return codes are also mapped according to fmincon expected return codes. Note:

This function mimics the behaviour of fmincon only.
In order to speed up minimizing, a bounded minimization algorithm fminbnd is used in a first try. If this fails, sqp algorithm is called.
Matlab:

```
A*x <= b
Aeq*x = beq
lb <= x <= ub
```

Octave:

```
g(x) = -Aeq*x + beq = 0
h(x) = -A*x + b >= 0
lb <= x <= ub
```

See the following example:

```
[x obj info iter] = fmincon (@(x)100*(x(2)-x(1)^2)^2 + (1-x(1))^2,[0.5,0],[1,2],1
x = [0.41494,0.17011]
obj =  0.34272
info =  1
iter =  6
```

Explanation of Input Parameters:

- *objf*: pointer to objective function
- *x0*: initial values
- *A*: inequality constraint matrix
- *b*: inequality constraint vector
- *Aeq*: equality constraint matrix
- *beq*: equality constraint vector
- *lb*: lower bound (required for fminbnd, defaults to -10)
- *ub*: upper bound (required for fminbnd, defaults to 10)

See also: sqp.

5.18 gammainc

`gammainc (`*x*`, `*a*`)`
`gammainc (`*x*`, `*a*`, `*tail*`)`

Compute the normalized incomplete gamma function.
This is defined as

$$\gamma(x,a) = \frac{1}{\Gamma(a)}\int_0^x t^{a-1}e^{-t}dt$$

with the limiting value of 1 as x approaches infinity. The standard notation is $P(a,x)$, e.g., Abramowitz and Stegun (6.5.1).
If *a* is scalar, then `gammainc (x, a)` is returned for each element of *x* and vice versa.
If neither *x* nor *a* is scalar, the sizes of *x* and *a* must agree, and `gammainc` is applied element-by-element. The elements of *a* must be nonnegative.
By default or if *tail* is `"lower"` the incomplete gamma function integrated from 0 to *x* is computed. If *tail* is `"upper"` then the complementary function integrated from *x* to infinity is calculated.

If *tail* is `"scaledlower"`, then the lower incomplete gamma function is multiplied by $\Gamma(a+1)\exp(x)x^{-a}$. If *tail* is `"scaledupper"`, then the upper incomplete gamma function is divided by the same quantity.
References:
M. Abramowitz and I. Stegun, *Handbook of mathematical functions* Dover publications, INC., 1972.
W. Gautschi, *A computational procedure for incomplete gamma functions* ACM Trans. Math Software, pp. 466–481, Vol 5, No. 4, 2012.
W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery *Numerical Recipes in Fortran 77*, ch. 6.2, Vol 1, 1992.

See also: gamma, gammainc, gammaln.

5.19 gammaincinv

`gammaincinv (`*y*`, a)`
`gammaincinv (`*y*`, a, `*tail*`)`

Compute the inverse of the normalized incomplete gamma function.
The normalized incomplete gamma function is defined as

$$\gamma(x,a) = \frac{1}{\Gamma(a)} \int_0^x t^{a-1} e^{-t} dt$$

and `gammaincinv (gammainc (x, a), a) = x` for each nonnegative value of *x*. If *a* is scalar, then `gammaincinv (y, a)` is returned for each element of *y* and vice versa.
If neither *y* nor *a* is scalar, the sizes of *y* and *a* must agree, and `gammaincinv` is applied element-by-element. The elements of *y* must be in $[0,1]$ and those of *a* must be positive.
By default or if *tail* is `"lower"` the inverse of the incomplete gamma function integrated from 0 to *x* is computed. If *tail* is `"upper"`, then the complementary function integrated from *x* to infinity is inverted.
The function is computed by standard Newton's method, by solving

$$y - \gamma(x,a) = 0$$

Reference: A. Gil, J. Segura, and N. M. Temme, *Efficient and accurate algorithms for the computation and inversion of the incomplete gamma function ratios*, SIAM J. Sci. Computing, pp. A2965–A2981, Vol 34, 2012.

See also: gamma, gammainc, gammaln.

5.20 generate_willowtree

value `= generate_willowtree (`*N*`, `*dk*`, `*z_method*`, `*willowtree_save_flag*`, `*path_static*`)` [Function File]

Computes the willow tree used e.g. for option pricing.
The willow tree is used as a lean and accurate option pricing lattice. This implementation of the willow tree concept is based on following literature:

- 'Willow Tree', Andy C.T. Ho, Master thesis, May 2000

- 'Willow Power: Optimizing Derivative Pricing Trees', Michael Curran, ALGO RESEARCH QUARTERLY, Vol. 4, No. 4, December 2001

Number of nodes must be in list [10,15,20,30,40,50]. These vectors are optimized by Currans Method to fulfill variance constraint (default: 20)
Variables:

- *N*: Number of timesteps in tree
- *dk*: timestep size of tree
- *z_method*: number of nodes per timestep
- *willowtree_save_flag*: boolean variable for saving tree to file
- *path_static*: path to directory if file shall be saved
- *Transition_matrix*: [output] optimized transition probabilities ("the Tree")
- *z*: [output] Z(0,1) distributed random variables used in tree

See also: option_willowtree.

5.21 getCapFloorRate

`[rate] = getCapFloorRate (CapFlag, F, X, tf, sigma, model)` [Function File]

Compute the forward rate of caplets or floorlets according to Black, Normal or analytical calculation formulas.
Input and output variables:

- *CapFlag*: model (Black, Normal) [required]
- *F*: forward rate (annualized) [required]
- *X*: strike rate (annualized) [required]
- *tf*: time factor until forward start date (in days) [required]
- *sigma*: swap volatility according to tenor, term and moneyness (act/365 continuous)[required]
- *model*: model (Black, Normal, Analytical) [required]
- *rate*: OUTPUT: adjusted forward rate

For Black model, the following formulas are applied:

```
Caplet_rate = (F*N( d1) - X*N( d2))
Floorlet_rate = (X*N(-d2) - F*N(-d1))
d1 = (log(F/X) + (0.5*sigma^2)*T)/(sigma*sqrt(tf))
d2 = d1 - sigma*sqrt(tf)
```

For Normal model, the following formulas are applied:

```
Caplet_rate = (F - X) * normcdf(d)  + sigma*sqrt(tf) * normpdf(d)
Floorlet_rate = (X - F) * normcdf(-d) + sigma*sqrt(tf) * normpdf(d)
d = (F - X) / (sigma*sqrt(tf));
```

For analytical model, the following formulas are applied:

```
Caplet_rate = max(0, F - X);
Floorlet_rate = max(0, X - F);
```

See also: swaption_bachelier, swaption_black76.

5.22 getFlooredSpotByFlooringForwardCurve

`[TermForward spotrates_floored forwardrates forwardrates_floored] = getFlooredSpotByFlooringForwardCurve(TermSpot, SpotRates, floor_rate, term_forwardrate, basis, comp_type, comp_freq, interp_method)` [Function File]

Compute the floored spot curve calculated by flooring the forward curve.
Explanation of Input Parameters:

- *TermSpot*: is a 1xN vector with all timesteps of the given curve
- *SpotRates*: is MxN matrix with curve rates defined in columns. Each row contains a specific scenario with different curve structure
- *floor_rate*: is a scalar, specifiying the floor applied to forward rates
- *term_forwardrate*: is a scalar, specifiying forward period
- *basis*: (optional) day count convention of instrument (default: act/365)
- *comp_type*: (optional) specifies compounding rule (simple, discrete, continuous (defaults to 'cont')).
- *comp_freq*: (optional) compounding frequency (default: annual)
- *interp_method*: (optional) specifies interpolation method for retrieving interest rates (defaults to 'linear').

Explanation of Output Parameters:

- *TermForward*: 1xN vector with all timesteps for output curves
- *spotrates_floored*: MxN matrix with floored spot curves
- *forwardrates*: MxN matrix with forward curves
- *forwardrates_floored*: MxN matrix with floored forward curves

See also: timefactor.

5.23 get_basis

`[basis ] = get_basis(dcc_string)` [Function File]

Map the basis for value according to a day count convention string. In order to introduce new day count conventions, add the basis to the cell and include the calculation method for the day count convention into the function timefactor().
The following mapping will be done for the input strings:

- *basis*: day-count basis (scalar)
  - *0* = actual/actual or act/act (1/1 mapped to act/act)
  - *1* = 30/360 SIA
  - *2* = act/360 or actual/360 or actual/360 Full
  - *3* = act/365 or actual/365 or actual/365 Full
  - *4* = 30/360 PSA

- *5* = 30/360 ISDA
- *6* = 30/360 European
- *7* = act/365 Japanese
- *8* = act/act ISMA
- *9* = act/360 ISMA
- *10* = act/365 ISMA
- *11* = 30/360E
- *13* = 30/360 or 30/360 German
- *14* = business/252
- *15* = act/364

See also: timefactor.

5.24 get_basket_volatility

`[basket_vola, basket_dict ] = get_basket_volatility (valuation_date, value_type, option, instrument_struct, index_struct, curve_struct, riskfactor_struct, matrix_struct, surface_struct)` [Function File]

Return diversified volatility for synthetic basket instruments. The diversified volatility is dependent on the option maturity and strike, so the volatility has to be calculated for each basket option valuation. The following two methods are implemented:

- *Levy 1992*: Levy uses a log-normal density as a first-order approximation to the true density. This holds for small maturities or volatilities only, since a weighted sum of log-normal distribution is not a log-normal distribution by itself.
- *VCV approach*: Assuming a normal distribution of underlying, the basket volatility is calculated by sigma = sqrt(w'*C*w) with Covariance Matrix C and the linear-combinations vector w.
- *Beisser*: Calculate lower limit of option prices by adjusting input volatilities and strikes. Also returns modified strike *basket_dict*.

5.25 get_bond_tf_rates

`[tf_vec rate_vec df_vec] = get_bond_tf_rates(valuation_date, cashflow_dates, cashflow_values, spread_constant, discount_nodes, discount_rates, basis, comp_type, comp_freq, interp_discount)` [Function File]

Compute the time factors, rates and discount factors for a given cash flow pattern according to a given discount curve and day count convention etc.
Pre-requirements:

- installed octave financial package
- custom functions timefactor, discount_factor, interpolate_curve, and convert_curve_rates

Input and output variables:

- *valuation_date*: Structure with relevant information for specification of the forward:

- *cashflow_dates*: cashflow_dates is a 1xN vector with all timesteps of the cash flow pattern
- *cashflow_values*: cashflow_values is a MxN matrix with cash flow pattern.
- *spread_constant*: a constant spread added to the total yield extracted from discount curve and spread curve (can be used to spread over yield)
- *discount_nodes*: tmp_nodes is a 1xN vector with all timesteps of the given curve
- *discount_rates*: tmp_rates is a MxN matrix with discount curve rates defined in columns. Each row contains a specific scenario with different curve structure
- *basis*: OPTIONAL: day-count convention of instrument (either basis number between 1 and 11, or specified as string (act/365 etc.)
- *comp_type*: OPTIONAL: compounding type of instrument (disc, cont, simple)
- *comp_freq*: OPTIONAL: compounding frequency of instrument (1,2,3,4,6,12 payments per year)
- *comp_type_curve*: OPTIONAL: compounding type of curve
- *basis_curve*: OPTIONAL: day-count convention of curve
- *comp_freq_curve*: OPTIONAL: compounding frequency of curve
- *interp_discount*: OPTIONAL: interpolation method of discount curve
- *sensi_flag*: OPTIONAL: boolean variable (calculate sensitivities) (default: linear)
- *tf_vec*: returns a Mx1 vector with time factors per cash flow date
- *rate_vec*: returns a Mx1 vector with rates per cf date
- *df_vec*: returns a Mx1 vector with discount factors per cf date method)

See also: timefactor, discount_factor, interpolate_curve, convert_curve_rates.

5.26 get_cms_rate_hagan

forward_rate= get_cms_rate_hagan(*valuation_date*, *value_type*, *swap*, *curve*, *sigma*, *payment_date*) [Function File]

Compute the cms rate of an underlying swap floating leg incl. convexity adjustment. The implementation of cms convexity adjustment is based on P.S. Hagan, Convexity Conundrums, 2003. There is a minor issue with Hagans formulas: An adjustment to the value of the swaplet / caplet / floorlet is being calculated. For calculation of this adjustment a volatility is required. The volatility has to be interpolated from a given volatility cube with a given moneyness. In case of swaplets, the moneyness can be assumed to be 1.0. For caplets / floorlets, the moneyness can be calculated as (cms_rate-X) or (cms_rate/X). Here either the adjusted cms rate or still the unadjusted cms rate can be used to calculate the moneyness.
Explanation of Input Parameters:

- *valuation_date*: valuation date
- *value_type*: value type (e.g. base or stress)
- *swap*: swap instrument object, underlying of cms swap
- *curve*: discount curve object
- *sigma*: volatility used for calculating convexity adjustment
- *payment_date*: payment date of cashflow

See also: discount_factor, timefactor, rollout_structured_cashflows.

5.27 get_cms_rate_hull

forward_rate= `get_cms_rate_hull`(*valuation_date*, *value_type*, *swap*, *curve*, *sigma*, *model*) [Function File]

Compute the cms rate of an underlying swap floating leg incl. convexity adjustment. The implementation of cms convexity adjustment is based on Hull: Option, Futures and other derivatives, 6th edition, page 734ff. Explanation of Input Parameters:

- *valuation_date*: valuation date
- *value_type*: value type (e.g. base or stress)
- *swap*: swap instrument object, underlying of cms swap
- *curve*: discount curve object
- *sigma*: volatility used for calculating convexity adjustment
- *model*: volatility model used for calculating convexity adjustment

See also: discount_factor, timefactor, rollout_structured_cashflows.

5.28 get_dependencies

`get_dependencies`(*path_octarisk*, *path_out*) [Function File]

Print a GraphViz .dot file containing all dependencies between Class methods and function names. All information is directly retrieved from all scriptnames and script source code. Comments and test cases are neglected. Final dot files are printed separately for all Classes and for an overall overview directly into the provided output folder. Compile .dot files with graphviz command:

```
dot -Tpdf Classname.dot -o Classname.pdf
dot -Tpng octarisk_dependencies.dot -o octarisk_dependencies.png
```

5.29 get_documentation

`get_documentation`(*type*, *path_octarisk*, *path_documentation*) [Function File]

Print documentation for all Octave functions in specified path. The documentation is extracted from the function headers and printed to a file 'functions.texi', 'functionname.html' or to standard output if a specific format (texinfo, html, txt) is set.
The path to all files has to be set in the variable path_documentation.

5.30 get_documentation_classes

`get_documentation_classes(`*`type, path_octarisk`*`,` *`path_documentation`*`)` [Function File]

Print documentation for all Octave Class definitions in specified path. The documentation is extracted from the static class methods and printed to a file 'functions.texi', 'functionname.html' or to standard output if a specific format (texinfo, html, txt) is set.

The path to all files has to be set in the variable path_documentation.

5.31 get_forward_rate

`forward_rate``= get_forward_rate(`*`nodes, rates`*`,` *`days_to_t1`*`,` *`days_to_t2`*`,` *`comp_type`*`,` *`interp_method`*`,` *`comp_freq`*`,,` *`basis`*`,` *`valuation_date`*`,` *`comp_type_curve`*`,` *`basis_curve`*`,` *`comp_freq_curve`* `,` *`floor_flag`*`)` [Function File]

Compute the forward rate calculated from interpolated rates from a yield curve. CAUTION: the forward rate is floored to 0.000001. Explanation of Input Parameters:

- *nodes*: is a 1xN vector with all timesteps of the given curve
- *rates*: is MxN matrix with curve rates defined in columns. Each row contains a specific scenario with different curve structure
- *days_to_t1*: is a scalar, specifiying term1 in days
- *days_to_t2*: is a scalar, specifiying term2 in days after term1
- *comp_type*: (optional) specifies compounding rule (simple, discrete, continuous (defaults to 'cont')).
- *interp_method*: (optional) specifies interpolation method for retrieving interest rates (defaults to 'linear').
- *comp_freq*: (optional) compounding frequency (default: annual)
- *basis*: (optional) day count convention of instrument (default: act/365)
- *valuation_date*: (optional) valuation date (default: today)
- *comp_type_curve*: (optional) compounding type of curve
- *basis_curve*: (optional) day count convention of curve
- *comp_freq_curve*: (optional) compounding frequency of curve
- *floor_flag*: (optional) Bool: flooring forward rates to 0.000001

See also: interpolate_curve, convert_curve_rates,timefactor.

5.32 get_gpd_var

`[`*`VAR ES`*`] = get_gpd_var(`*`chi, sigma`*`,` *`u`*`,` *`q`*`,` *`n`*`,` *`nu`*`)` [Function File]

Return Value-at-risk (VAR) and expected shortfall (ES) according to a generalized Pareto distribution.

Implementation according to *Risk Management and Financial Institutions* by John C. Hull, 4th edition, Wiley 2015, section 13.6, page 292ff.

Input and output variables:

- *chi*: GPD shape parameter one (float)
- *sigma*: GPD shape parameter two (float)
- *u*: offset level (float)
- *q*: quantile (float in [0:1])
- *n*: Number of scenarios in total distribution (integer)
- *nu*: number of tail scenarios (in doubt set to 0.025 * n) (integer)
- *VAR*: OUTPUT: Value-at-Risk according to the GPD
- *ES*: OUTPUT: Expected shortfall according to the GPD

Example call for calculation of VAR and ES for several confidence levels:

```
[VAR ES] = get_gpd_var(0.00001,1632.9,5930.8,[0.99;0.995;0.999],50000,1250)
```

5.33 get_marginal_distr_pearson

`[r type ] = get_marginal_distr_pearson (mu, sigma, skew, kurt, Z)` [Function File]

Compute a marginal distribution for given set of uniform random variables with given mean, standard deviation skewness and kurtosis. The mapping is done via the Pearson distribution family.
The implementation is based on the R package 'PearsonDS: Pearson Distribution System' and the function 'pearsonFitM' by
Martin Becker and Stefan Kloessner (2013)
R package version 0.97.
URL: http://CRAN.R-project.org/package=PearsonDS
licensed under the GPL >= 2.0
Input and output variables:

- *mu*: mean of marginal distribution (scalar)
- *sigma*: standard deviation of marginal distribution (scalar)
- *skew*: skewness of marginal distribution (scalar)
- *kurt*: kurtosis of marginal distribution (scalar)
- *Z*: uniform distributed random variables (Nx1 vector)
- *r*: OUTPUT: Nx1 vector with random variables distributed according to Pearson type (vector)
- *type*: OUTPUT: Pearson distribution type (I - VII) (scalar)

The marginal distribution type is chosen according to the input parameters out of the Pearson Type I-VII distribution family:

- *Type 0* = normal distribution
- *Type I* = generalization of beta distribution
- *Type II* = symmetric beta distribution
- *Type III* = gamma or chi-squared distribution
- *Type IV* = special distribution, not related to any other distribution

- *Type V* = inverse gamma distribution
- *Type VI* = beta-prime or F distribution
- *Type VII* = Student's t distribution

See also: discount_factor.

5.34 get_sub_object

`[`*`match_obj ret_code`*`] = get_sub_object(`*`input_struct`*`, `*`input_id`*`)` [Function File]

Return the object contained in a structure matching a given ID. Return code 1 (success) and 0 (fail).

5.35 get_sub_struct

`[`*`match_struct ret_code`*`] = get_sub_object(`*`input_struct`*`, `*`input_id`*`)` [Function File]

Return the sub-structure contained in a structure matching a given ID. Return code 1 (success) and 0 (fail).

5.36 harrell_davis_weight

`[`*`X`*`] = harrell_davis_weight (`*`scenarios`*`, `*`observation`*`, `*`alpha`*`)` [Function File]

Compute the Harrell-Davis (1982) quantile estimator and jacknife standard errors of quantiles. The quantile estimator is a weighted linear combination or order statistics in which the order statistics used in traditional nonparametric quantile estimators are given the greatest weight. In small samples the H-D estimator is more efficient than traditional ones, and the two methods are asymptotically equivalent. The H-D estimator is the limit of a bootstrap average as the number of bootstrap resamples becomes infinitely large.

Variables:

- *scenarios*: number of total scenarios
- *observation*: input vector for which HD weights shall be calculated
- *alpha*: quantile (e.g. 0.005)
- *X*: OUTPUT: HD-weight corresponding to observation vector

5.37 ind2sub_tril

`[ `*`r`*`, `*`c`*` ] =  ind2sub_tril (`*`N`*`, `*`idx`*`)` [Function File]

Convert a linear index to subscripts of a trinagular matrix.

An example of trinagular matrix linearly indexed follows

```
N = 4;
A = -repmat (1:N,N,1);
A += repmat (diagind, N,1) - A.';
A = tril(A)
```

```
=> A =
    1    0    0    0
    2    5    0    0
    3    6    8    0
    4    7    9   10
```

The following example shows how to convert the linear index '6' in the 4-by-4 matrix of the example into a subscript.

```
[r, c] = ind2sub_tril (4, 6)
=> r =  2
  c =  3
```

when *idx* is a row or column matrix of linear indeces then *r* and *c* have the same shape as *idx*.

See also: vech, ind2sub, sub2ind_tril.

5.38 instrument_valuation

`[ret_instr_obj] = instrument_valuation (instr_obj, valuation_date, scenario, instrument_struct, surface_struct, matrix_struct, curve_struct, index_struct, riskfactor_struct, path_static, scen_number, first_eval)` [Function File]

Valuation of instruments according to instrument type. The last four variables can be empty in case of base scenario valuation.
Variables:

- *instr_obj*: instrument, which has to be valuated
- *valuation_date*: valuation date
- *scenario*: scenario ['base','stress', MC timestep: e.g. '250d']
- *instrument_struct*: structure with all instruments in session
- *surface_struct*: structure with all surfaces in session
- *matrix_struct*: structure with all matrizes in session
- *curve_struct*: structure with all curves in session
- *index_struct*: structure with all indizes in session
- *riskfactor_struct*: structure with all riskfactors in session
- *para_object*: structure with required parameters
  - *para_object.path_static*: OPTIONAL: path to folder with static files
  - *para_object.scen_number*: OPTIONAL: number of scenarios
  - *para_object.scenario*: OPTIONAL: timestep number of days for MC scenarios
  - *para_object.first_eval*: OPTIONAL: boolean, first_eval == 1 means first evaluation
- *ret_instr_obj*: RETURN: evaluated instrument object

5.39 integrationtests

`integrationtests(`*`path_folder`*`)` [Function File]

Call integrationtests of specified functions and return test statistics.

Input parameter: path to folder with testdata. All integration test scripts have to be hard coded in this script.

5.40 interpolate_curve

`interpolate_curve (`*`nodes`*`, `*`rates`*`, `*`timestep`*`)` [Function File]

`interpolate_curve (`*`nodes`*`, `*`rates`*`, `*`timestep`*`, `*`interp_method`*`, `*`ufr`*`, `*`alpha`*`, `*`extrap_method`*`)` [Function File]

Calculate an interpolated rate on a curve for a given timestep.

Supported methods are: linear (default), moneymarket, exponential, loglinear, spline, smith-wilson, monotone-convex, constant (mapped to previous), previous and next.

A constant extrapolation is assumed, except for smith-wilson, where the ultimate forward rate will be reached proportional to reversion speed alpha. For all methods except splines a fast taylormade algorithm is used. For splines see Octave function interp1 for more details. Explanation of Input Parameters of the interpolation curve function:

- *nodes*: is a 1xN vector with all timesteps of the given curve
- *rates*: is MxN matrix with curve rates per timestep defined in columns. Each row contains a specific scenario with different curve structure
- *timestep*: is a scalar, specifiying the interpolated timestep on vector nodes
- *interp_method*: OPTIONAL: interpolation method
- *ufr*: OPTIONAL: (only used for smith-wilson): ultimate forward rate (default: last liquid point)
- *alpha*: OPTIONAL: (only used for smith-wilson): reversion speed to ultimate forward rate (default: 0.1)
- *extrap_method*: OPTIONAL: extrapolation method

See also: interp1, interp2, interp3, interpn.

5.41 load_correlation_matrix

`[`*`mktdata_struct id_failed_cell`*`] = load_mktdata_objects(`*`mktdata_struct, path_mktdata`*`, `*`file_mktdata`*`, `*`path_output`*`, `*`path_archive`*`, `*`tmp_timestamp`*`, `*`archive_flag`*`)` [Function File]

Load data from mktdata object specification file and generate objects with parsed data. Store all objects in provided mktdata struct and return the final struct and a cell containing the failed mktdata ids.

5.42 load_instruments

[*instrument_struct id_failed_cell*] = load_instruments(*instrument_struct, valuation_date*, *path_instruments*, *file_instruments*, *path_output*, *path_archive*, *tmp_timestamp*, *archive_flag*) [Function File]

Load data from instrument specification file and generate objects with parsed data. Store and return all objects in provided instrument structure. and a cell containing the failed instrument ids. The order of the final instrument structure is automatically set that all derivatives (OPT,SWAPT,SYNTH) are coming last.

5.43 load_matrix_objects

[*matrix_struct matrix_failed_cell*] = load_matrix_objects(*matrix_struct, path_mktdata*, *input_filename_matrix_index*) [Function File]

Load data from mktdata matrix object specification files and generate a struct with parsed data. Store all objects in provided struct and return the final struct and a cell containing the failed matrix ids.

5.44 load_mktdata_objects

[*mktdata_struct id_failed_cell*] = load_mktdata_objects(*mktdata_struct, path_mktdata*, *file_mktdata*, *path_output*, *path_archive*, *tmp_timestamp*, *archive_flag*) [Function File]

Load data from mktdata object specification file and generate objects with parsed data. Store all objects in provided mktdata struct and return the final struct and a cell containing the failed mktdata ids.

5.45 load_parameter

[*parameter_struct id_failed_cell*] = load_parameter(*path_parameter, filename_parameter*) [Function File]

Load data from parameter specification file and generate parameter object with parsed data.

5.46 load_positions

[*portfolio_struct id_failed_cell*] = load_positions(*portfolio_struct, valuation_date*, *path_positions*, *file_positions*, *path_output*, *path_archive*, *tmp_timestamp*, *archive_flag*) [Function File]

Load data from position specification file and generate objects with parsed data. Store all objects in provided position struct and return the final struct and a cell containing the failed position ids.

5.47 load_riskfactor_scenarios

[*riskfactor_struct rf_failed_cell*] = load_riskfactor_scenarios(*riskfactor_struct*, *M_struct*, *mc_timesteps*, *mc_timestep_days*) [Function File]

Generate MC scenario shock values for risk factor curve objects. Store all MC scenario shock values in provided struct and return the final struct and a cell containing all failed risk factor ids.

5.48 load_riskfactor_stresses

[*riskfactor_struct rf_failed_cell*] = load_riskfactor_stresses(*riskfactor_struct*, *stresstest_struct*) [Function File]

Generate stresses for risk factor objects (except curves). Store all stresses in provided struct and return the final struct and a cell containing all failed risk factor ids.

5.49 load_riskfactors

[*riskfactor_struct id_failed_cell*] = load_riskfactors(*riskfactor_struct*, *valuation_date*, *path_riskfactors*, *file_riskfactors*, *path_output*, *path_archive*, *tmp_timestamp*, *archive_flag*) [Function File]

Load data from riskfactor specification file and generate objects with parsed data. Store all objects in provided riskfactor struct and return the final struct and a cell containing the failed riskfactor ids.

5.50 load_stresstests

[*portfolio_struct id_failed_cell*] = load_stresstests(*portfolio_struct*, *valuation_date*, *path_stresstests*, *file_stresstests*, *path_output*, *path_archive*, *tmp_timestamp*, *archive_flag*) [Function File]

Load data from stresstest specification file and generate a struct with parsed data. Store all stresstests in provided struct and return the final struct and a cell containing the failed position ids.

5.51 load_volacubes

[*surface_struct vola_failed_cell*] = load_volacubes(*surface_struct*, *path_mktdata*, *input_filename_vola_index*, *input_filename_vola_ir*, *input_filename_vola_stochastic*) [Function File]

Load data from mktdata volatility surfaces / cubes specification files and generate a struct with parsed data. Store all stresstests in provided struct and return the final struct and a cell containing the failed volatility ids.

5.52 load_yieldcurves

[*rf_ir_cur_cell curve_struct*] = load_yieldcurves(*curve_struct*, *riskfactor_struct*, *mc_timesteps*, *path_output*, *saving*) [Function File]

Generate curve objects from risk factor objects. Store all curves in provided struct and return the final struct and a cell containing all interest rate risk factor currency / ratings.

5.53 octarisk

octarisk (*path_working_folder*) [Function File]

Full valuation Monte-Carlo risk calculation framework.
See www.octarisk.com for further information.

5.54 octarisk_gui

##

5.55 option_asian_levy

value = option_asian_levy (*CallPutFlag*, *S*, *X*, *T*, *r*, *sigma*, *divrate*, *n*) [Function File]

Compute the prices of european type asian average price call or put options according to Levy (1992) valuation formula. Convert all input parameter into continuously compounded values with act/365 day count convention.
The implementation is based on following literature:

- "Complete Guide to Option Pricing Formulas", Espen Gaarder Haug, 2nd Edition, page 190ff.

Variables:

- *CallPutFlag*: Call: "1", Put: "0"
- *S*: stock price at time 0
- *X*: strike price
- *T*: time to maturity in days
- *r*: annual risk-free interest rate (continuous, act/365)
- *sigma*: annualized implied volatility
- *divrate*: dividend rate p.a. (continuous, act/365)
- *n*: number of averaging dates (defaults to continuous: n = number of days to maturity)

See also: option_bs, option_asian_vorst90.

5.56 option_asian_vorst90

`value = option_asian_vorst90 (CallPutFlag, S, X, T, r, sigma, divrate)` [Function File]

Compute the prices of european type asian continously geometric average price call or put options according to Kemna and Vorst (1990) valuation formula. Convert all input parameter into continuously compounded values with act/365 day count convention. The implementation is based on following literature:

- "Complete Guide to Option Pricing Formulas", Espen Gaarder Haug, 2nd Edition, page 183ff.

Variables:

- *CallPutFlag*: Call: "1", Put: "0"
- *S*: stock price at time 0
- *X*: strike price
- *T*: time to maturity in days
- *r*: annual risk-free interest rate (continuous, act/365)
- *sigma*: annualized implied volatility (continuous, act/365)
- *divrate*: dividend rate p.a. (continuous, act/365)

See also: option_bs, option_asian_levy.

5.57 option_barrier

`[value] = option_barrier (CallPutFlag, UpFlag, S, X, H, T, r, sigma, q, Rebate)` [Function File]

Compute the prices of European call or put out or in barrier options.
Reference: Espen Gaarder Haug, "Complete Guide to Option Pricing Formulas", 2nd Edition, page 152ff.
Variables:

- *CallPutFlag*: Call: '1', Put: '0'
- *UpFlag*: Up: 'U', Down: 'D'
- *OutorIn*: 'out' or 'in' barrier option
- *S*: stock price at time 0
- *X*: strike price
- *H*: barrier
- *T*: time to maturity in days
- *r*: annual risk-free interest rate (continuously compounded)
- *sigma*: implied volatility of the stock price measured as annual standard deviation
- *q*: dividend rate p.a., continously compounded
- *Rebate*: Rebate of barrier option

5.58 option_binary

`[value] = option_binary (CallPutFlag, binary_type, S, X1, X2, T, r, sigma, divrate)` [Function File]

Compute the prices of European Binary call or put options according to Reiner and Rubinstein (Unscrambling the Binary Code, RISK 4 (October 1991), pp. 75-83) valuation formulas:

Option type Gap
A gap call option pays the difference (gap) between spot and either one of two strike values:

```
C(S,X1,X2,T) = X2*exp(-rT)*N(d)
P(S,X1,X2,T) = X2*exp(-rT)*N(-d)
d = (log(S/X1) + (r - divrate + 0.5*sigma^2)*T)/(sigma*sqrt(T))
```

Option type Cash-or-Nothing
A cash or nothing option pays the pre-defined amount X2 if the value is larger than the strike X1 (call option) or lower than the strike X1(put option):

```
C(S,X1,X2,T) = N(d)*X2*exp(-rT)
P(S,X1,X2,T) = N(-d)*X2*exp(-rT)
d = (log(S/X1) + (r - divrate - 0.5*sigma^2)*T)/(sigma*sqrt(T))
```

Option type Asset-or-Nothing
An asset or nothing option pays the future spot value S if the value is larger than the strike X1(call option) or lower than the strike X1 (put option):

```
C(S,X1,T) = S*N(d)*exp(-divrate*T)
P(S,X1,T) = S*N(-d)*exp(-divrate*T)
d = (log(S/X1) + (r - divrate + 0.5*sigma^2)*T)/(sigma*sqrt(T))
```

Option type Supershare
A supershare option has a payoff, if the future spot values lies between an lower bound X1 and upper bound X2, and is zero otherwise:

```
Value(S,X1,X2,T) = (S*exp(-divrate*T)/X1) * (N(d1) - N(d2))
d1 = (log(S/X1) + (r - divrate + 0.5*sigma^2)*T)/(sigma*sqrt(T))
d2 = (log(S/X2) + (r - divrate + 0.5*sigma^2)*T)/(sigma*sqrt(T))
```

All formulas are taken from Haug, Complete Guide to Option Pricing Formulas, 2nd edition, page 174ff.

Variables:

- *CallPutFlag*: Call: '1', Put: '0'
- *binary_type*: can be 'gap','cash','asset'
- *S*: stock price at time 0
- *X1*: strike price (lower bound for supershare or gap options)
- *X2*: payoff strike price (used for Gap and cash options, upper bound of supershare options)

- *T*: time to maturity in days
- *r*: annual risk-free interest rate (continuously compounded, act/365)
- *sigma*: implied volatility of the stock price measured as annual standard deviation
- *divrate*: dividend rate p.a., continously compounded

See also: option_willowtree, option_bs.

5.59 option_bjsten

`[value] = option_bjsten (CallPutFlag, S, X, T, r, sigma, divrate)` [Function File]

Calculate the option price of an American call or put option stocks, futures, and currencies. The approximation method by Bjerksund and Stensland is used.

The Octave implementation is based on a R function implemented by Diethelm Wuertz Rmetrics - Pricing and Evaluating Basic Options, Date 2015-11-09 Version 3022.85

References: Haug E.G., The Complete Guide to Option Pricing Formulas
Example taken from Reference:

```
price = option_bjsten(1,42,40,0.75*365,0.04,0.35,0.08)
price = 5.2704
```

Variables:

- *CallPutFlag*: Call: '1', Put: '0'
- *S*: stock price at time 0
- *X*: strike price
- *T*: time to maturity in days
- *r*: annual risk-free interest rate (continuously compounded)
- *sigma*: implied volatility of the stock price measured as annual standard deviation
- *divrate*: dividend rate p.a., continously compounded

See also: option_willowtree, option_bs.

5.60 option_bond_hw

`[value] = option_bond_hw (value_type,bond,curve,callschedule,putschedule)` [Function File]

Compute the value of a put or call bond option using Hull-White Tree model.
This script is a wrapper for the function pricing_callable_bond_cpp and handles all input and ouput data. Input data: value type, bond instrument, curve instrument, call and put schedule. References:

- Hull, Options, Futures and other derivatives, 7th Edition

See also: pricing_callable_bond_cpp.

5.61 option_bs

`[value delta gamma vega theta rho omega] = option_bs (CallPutFlag, S, X, T, r, sigma, divrate)` [Function File]

Compute the prices of european call or put options according to Black-Scholes valuation formula:

```
C(S,T) = N(d_1)*S - N(d_2)*X*exp(-rT)
P(S,T) = N(-d_2)*X*exp(-rT) - N(-d_1)*S
d1 = (log(S/X) + (r + 0.5*sigma^2)*T)/(sigma*sqrt(T))
d2 = d1 - sigma*sqrt(T)
```

The Greeks are also computed (delta, gamma, vega, theta, rho, omega) by their closed form solution.
Parallel computation for column vectors of S,X,r and sigma is possible.

Variables:

- *CallPutFlag*: Call: '1', Put: '0'
- *S*: stock price at time 0
- *X*: strike price
- *T*: time to maturity in days
- *r*: annual risk-free interest rate (continuously compounded, act/365)
- *sigma*: implied volatility of the stock price measured as annual standard deviation
- *divrate*: dividend rate p.a., continously compounded

See also: option_willowtree, swaption_black76.

5.62 option_lookback

`[value] = option_lookback (CallPutFlag, lookback_type, S, X1, X2, T, r, sigma, divrate)` [Function File]

Compute the prices of European Lookback call or put options of type floating strike or fixed strike.

Floating Strike options:
A floating strike lookback call / put gives you the right to buy / sell the the underlying security at the lowest / highest price observed during options lifetime. Pricing according to Goldman, Sosin and Gatto (1979) ("Path dependent options: Buy at the Low Sell at the High", Journal of Finance, 34(5), 1111- 1127) valuation formulas.

Fixed Strike options:
A fixed strike lookback call / put pays out the maximum of the difference between the highed observed price and the strike and 0 (call option) or the maximum of the difference between strike and lowest observed price and 0 (put option). Pricing according to Conze and Viswanathan (1991) ("Path dependent options: The Case of Lookback Options", Journal of Finance, 36, 1893 - 1907) formulas.

All formulas are taken from Haug, Complete Guide to Option Pricing Formulas, 2nd edition, page 141ff.

Variables:

- *CallPutFlag*: Call: '1', Put: '0'
- *lookback_type*: can be 'floating_strike','fixed_strike'
- *S*: stock price at time 0
- *X1*: strike price (or S_min or S_max for fixed strike)
- *X2*: payoff strike of fixed strike option
- *T*: time to maturity in days
- *r*: annual risk-free interest rate (continuously compounded, act/365)
- *sigma*: implied volatility of the stock price measured as annual standard deviation
- *divrate*: dividend rate p.a., continously compounded

See also: option_binary, option_bs.

5.63 option_willowtree

`value = option_willowtree (CallPutFlag, AmericanFlag, S, X, T, r, sigma, dividend, dk)` [Function File]

`value = option_willowtree (CallPutFlag, AmericanFlag, S, X, T, r, sigma, dividend, dk, nodes, path_static)` [Function File]

Computes the price of european or american equity options according to the willow tree model.

The willow tree approach provides a fast and accurate way of calculating option prices.
This implementation of the willow tree concept is based on following literature:

- 'Willow Tree', Andy C.T. Ho, Master thesis, May 2000
- 'Willow Power: Optimizing Derivative Pricing Trees', Michael Curran, ALGO RESEARCH QUARTERLY, Vol. 4, No. 4, December 2001

Example of an American Call Option with continuous dividends:
(365 days to maturity, vector with different spot prices and volatilities, strike = 8, r = 0.06, dividend = 0.05, timestep 5 days, 20 nodes): `option_willowtree(1,1,[7;8;9;7;8;9],8,365,0.06,[0.2;0.2;0.2;0.3;0.3;0.3],0.05,5,20)`

Variables:

- *CallPutFlag*: Call: '1', Put: '0'
- *AmericanFlag*: American option: '1', European Option: '0'
- *S*: stock price at time 0
- *X*: strike price
- *T*: time in days to maturity
- *r*: annual risk-free interest rate (cont, act/365)
- *sigma*: implied volatility of the stock price
- *dividend*: continuous dividend yield, act/365
- *dk*: size of timesteps for valuation points (default: 5 days)

- *nodes*: number of nodes for willow tree setup. Number of nodes must be in list [10,15,20,30,40,50]. These vectors are optimized by Currans Method to fulfill variance constraint (default: 20)

See also: option_binomial, option_bs, option_exotic_mc.

5.64 perform_rf_stat_tests

`[`*`retcode`*`] = perform_rf_stat_tests(`*`riskfactor_cell,riskfactor_struct,RndMat,distr_type`*`)` [Function File]

Perform statistical tests on risk factor shock vector. Return 1 if all tests pass, return 255 if at least one test fails.

5.65 pricing_forward

`[`*`theo_value`*` ] = pricing_forward (`*`valuation_date`*`, `*`forward`*`, `*`discount_curve_object`*`, `*`underlying_object`*`, `*`und_curve_object`*`)` [Function File]

Compute the theoretical value and price of FX, equity and bond forwards and futures.

Input and output variables:

- *valuation_date*: valuation date
- *forward*: forward object
- *discount_curve_object*: discount curve for forward
- *underlying_object*: underlying object of forward
- *und_curve_object*: discount curve object of underlying object

See also: timefactor, discount_factor, convert_curve_rates.

5.66 pricing_npv

`[`*`npv MacDur Convexity MonDur Convexity_alt`*`] = pricing_npv(`*`valuation_date, cashflow_dates`*`, `*`cashflow_values`*`, `*`spread_constant`*`, `*`discount_nodes`*`, `*`discount_rates`*`, `*`basis`*`, `*`comp_type`*`, `*`comp_freq`*`, `*`interp_discount`*`)` [Function File]

Compute the net present value, Macaulay Duration, Convexity and Monetary duration of a given cash flow pattern according to a given discount curve and day count convention etc.
Pre-requirements:

- installed octave financial package
- custom functions timefactor, discount_factor, interpolate_curve, and convert_curve_rates

Input and output variables:

- *valuation_date*: valuation date (preferred as datenum)

- *cashflow_dates*: cashflow_dates is a 1xN vector with all timesteps of the cash flow pattern
- *cashflow_values*: cashflow_values is a MxN matrix with cash flow pattern.
- *spread_constant*: a constant spread added to the total yield extracted from discount curve and spread curve (can be used to spread over yield)
- *discount_nodes*: discount_nodes is a 1xN vector with all timesteps of the given curve
- *discount_rates*: discount_rates is a MxN matrix with discount curve rates defined in columns. Each row contains a specific scenario with different curve structure
- *basis*: OPTIONAL: day-count convention of instrument (either basis number between 1 and 11, or specified as string (act/365 etc.)
- *comp_type*: OPTIONAL: compounding type of instrument (disc, cont, simple)
- *comp_freq*: OPTIONAL: compounding frequency of instrument (1,2,3,4,6,12 payments per year)
- *comp_type_curve*: OPTIONAL: compounding type of curve
- *basis_curve*: OPTIONAL: day-count convention of curve
- *comp_freq_curve*: OPTIONAL: compounding frequency of curve
- *interp_discount*: OPTIONAL: interpolation method of discount curve
- *sensi_flag*: OPTIONAL: boolean variable (calculate sensitivities) (default: linear)
- *npv*: returns a Mx1 vector with all net present values per scenario
- *MacDur*: returns a Mx1 vector with all Macaulay durations
- *Convexity*: returns a Mx1 vector with all convexities
- *MonDur*: returns a Mx1 vector with all Monetary durations
- *Convexity_alt*: returns a Mx1 vector with Convexity (alternative method)

See also: timefactor, discount_factor, interpolate_curve, convert_curve_rates.

5.67 print_class2dot

get all classes in folder and extract all Instrument classes

5.68 profiler_analysis

`profiler_analysis(`*`script_name`*`,`*`argument`*`,`*`depth`*`)` [Function File]

Call profiler for specified script and argument and return detailed statistics. Input variables:

- *script_name*: name of script as string [required]
- *argument*: first and only argument of script [required]
- *depth*: number of sub-functions to analyse [optional, default = 10]

5.69 replacement_script

`replacement_script(`*`replacement_list`*`)` [Function File]

Matlab Adaption of Octarisk Code Input files phrases to replace: wordlist_matlab.csv Format:(String;Replacement String;File) Input files for replacement: Automatical detection of all m.files in directory for replacement Output data: Rewritten m.files

See also: adapt_matlab.

5.70 return_checked_input

`[`*`retval`*`] = return_checked_input (`*`obj`*`, `*`val`*`, `*`prop`*`, `*`type`*`)` [Function File]

Return value with validated input values according to value type date, char, numeric, and boolean or special treatment for scenario values. Used for storing correct field values for classes or structs. The function itself is divided into two parts: special attributes with taylormade validation checks are used for type 'special', while a generic approach according to differenct types are performed in the second part.

5.71 rollout_structured_cashflows

`[`*`ret_dates ret_values accrued_interest`*`] =` [Function File]
`rollout_structured_cashflows (`*`valuation_date`*`, `*`value_type`*`, `*`instrument`*`, `*`ref_curve`*`, `*`surface`*`, `*`riskfactor`*`)`

Compute cash flow dates and cash flows values, accrued interests and last coupon date for fixed rate bonds, floating rate notes, amortizing bonds, zero coupon bonds and structured products like caps and floors, CM Swaps, capitalized or averaging CMS floaters or inflation linked bonds.

See also: timefactor, discount_factor, get_forward_rate, interpolate_curve.

5.72 save_objects

`[`*`riskfactor_struct rf_failed_cell`*`] =` [Function File]
`save_objects(`*`path_output, riskfactor_struct`*`, `*`instrument_struct`*`, `*`portfolio_struct`*`, `*`stresstest_struct`*`)`

Save provided structs for riskfactors, instruments, positions and stresstests.

5.73 scenario_generation_MC

`[`*`R distr_type`*` ] = cenario_generation_MC (`*`corr_matrix`*`, `*`P`*`,` [Function File]
`mc``, `*`copulatype`*`, `*`nu`*`, `*`time_horizon`*`)`

Compute correlated random numbers according to Gaussian or Student-t copulas and arbitrary marginal distributions within the Pearson distribution system.

See also: get_marginal_distr_pearson, mvnrnd, normcdf, mvtrnd ,tcdf.

5.74 struct2obj

`[obj] = struct2obj(s,verbose)` [Function File]

Converting structs into objects. Therefore the constructors of hard-coded classes are used to invoke objects and to set all structures attributes. The final object *obj* is returned. The optional *verbose* parameter sets the logging level.

5.75 swaption_bachelier

`[SwaptionBachelierValue] = swaption_bachelier (PayerReceiverFlag, F, X, T, r, sigma, m, tau)` [Function File]

Compute the price of european interest rate swaptions according to Bachelier Pricing Functions assuming normal-distributed volatilities. Fast implementation, fully vectorized.

```
C = ((F-X)*N(d1) + sigma*sqrt(T)*n(d1))*exp(-rT) * multiplicator(m,tau)
P = ((X-F)*N(-d1) + sigma*sqrt(T)*n(d1))*exp(-rT) * multiplicator(m,tau)
d1 = (F-X)/(sigma*sqrt(T))
```

Variables:

- *PayerReceiverFlag*: Call / Payer '1' (pay fixed) or Put / Receiver '0' (receive fixed, pay floating) swaption
- *F*: forward rate of underlying interest rate (forward in T years for tau years)
- *X*: strike rate
- *T*: time in days to maturity
- *r*: annual risk-free interest rate (continuously compounded)
- *sigma*: implied volatility of the interest rate measured as annual standard deviation
- *m*: Number of Payments per year (m = 2 -> semi-annual) (continuous compounding is assumed)
- *tau*: Tenor of underlying swap in Years

See also: option_bs.

5.76 swaption_black76

`[SwaptionB76Value] = swaption_black76 (PayerReceiverFlag, F, X, T, r, sigma, m, tau)` [Function File]

Compute the price of european interest rate swaptions according to Black76 pricing functions.

```
C = (F*N( d1) - X*N( d2))*exp(-rT) * multiplicator(m,tau)
P = (X*N(-d2) - F*N(-d1))*exp(-rT) * multiplicator(m,tau)
d1 = (log(S/X) + (r + 0.5*sigma^2)*T)/(sigma*sqrt(T))
d2 = d1 - sigma*sqrt(T)
```

Variables:

- *PayerReceiverFlag*: Call / Payer '1' (pay fixed) or Put / Receiver '0' (receive fixed, pay floating) swaption
- *F*: forward rate of underlying interest rate (forward in T years for tau years)
- *X*: strike rate
- *T*: time in days to maturity
- *r*: annual risk-free interest rate (continuously compounded)
- *sigma*: implied volatility of the interest rate measured as annual standard deviation
- *m*: Number of Payments per year (m = 2 -> semi-annual) (continuous compounding is assumed)
- *tau*: Tenor of underlying swap in Years

See also: swaption_bachelier.

5.77 swaption_underlyings

`[SwaptionValue] = swaption_underlyings (PayerReceiverFlag, F, X, T, r, sigma, m, tau)` [Function File]

Compute the price of european interest rate swaptions according to Black76 or Normal pricing functions using underlying fixed and floating legs.

See also: swaption_bachelier, swaption_black76.

5.78 test_io

`[success_tests total_tests] = test_io(path_testing_folder)` [Function File]

Perform integration tests for all functions which rely on input and output data. The functions have to be hard coded in this script and rely on validated output data. The storage and parsing process of objects is a little bit tricky. At first, all objects have to converted into structed and stored to a file. After the structs from the file have been parsed, all structe have to converted back again into objects using constructor and set methods. See section B.2 for an example.

5.79 test_oct_files

this is only a dummy function for containing all the oct file testing suites.

5.80 timefactor

`[tf dip dib] = timefactor(d1, d2, basis)` [Function File]

Compute the time factor for a specific time period and day count basis.

Depending on day count basis, the time factor is evaluated as (days in period) / (days in year)

Input and output variables:

- *d1*: number of days until first date (scalar)
- *d2*: number of days until second date (scalar)

- *basis*: day-count basis (scalar or string)
- *df*: OUTPUT: discount factor (scalar)
- *dip*: OUTPUT: days in period (nominator of time factor) (scalar)
- *dib*: OUTPUT: days in base (denominator of time factor) (scalar)

See also: discount_factor, yeardays, get_basis.

5.81 unittests

`unittests()` [Function File]

Call unittests of specified functions and return test statistics.

5.82 unvech

`m = unvech (v, scale)` [Function File]

Performs the reverse of `vech` on the vector *v*.

Given a Nx1 array *v* describing the lower triangular part of a matrix (as obtained from `vech`), it returns the full matrix.

The upper triangular part of the matrix will be multiplied by *scale* such that 1 and -1 can be used for symmetric and antisymmetric matrix respectively. *scale* must be a scalar and defaults to 1.

See also: vech, ind2sub, sub2ind_tril.

5.83 update_mktdata_objects

`[index_struct curve_struct id_failed_cell] = update_mktdata_objects(mktdata_struct, index_struct, riskfactor_struct, curve_struct)` [Function File]

Update all market data objects with scenario dependent risk factor and curve shocks. Return index struct and curve struct with scenario dependent absolute values.

Calculate reciprocal FX conversion factors for all exchange rate market objects (e.g. FX_USDEUR = 1 ./ FX_USDEUR). During aggregation and instrument currency conversion the appropriate FX exchange rate is always chosen by FX_BasecurrencyForeigncurrency) Volatility surfaces and cube MC and stress shocks are generated in script load_volacubes.m

5.84 betainc_lentz_vec

`f = betainc_lentz_vec(y,a,b)` [Loadable Function]

Continued fraction for incomplete gamma function (vectorized version). This function should be called from function batainc_vec.m only.

Input and output variables:

- *x*: x value to calcluate cumulative beta distribution
- *a*: first shape parameter
- *b*: second shape parameter
- *f*: return value of beta cdf at x for a and b

5.85 calc_sobol_cpp

`retvec` = *calc_sobol_cpp*(`scen`, *dim*, *directionfile*) [Loadable Function]

Calculate Sobol numbers for *scen* rows and *dim* columns for a given file with direction numbers *directionfile*. Implementation uses code of Frances Y. Kuo and Stephen Joe, 2008. taken from http://web.maths.unsw.edu.au/~fkuo/sobol/ License included in source code.

5.86 calc_vola_basket_cpp

`Volatility` = *calc_vola_basket_cpp*(*M1*, *TF*, *exponents*, *prefactors*) [Loadable Function]

Compute the diversified volatility of a basket of securities.
This function uses long double precision to handle large volatilities.
Input and output variables:

- *M1*: M1 of basket vola function
- *TF*: time factor in years
- *exponents*: Matrix with exponents (in columns) and different scenarios in rows
- *prefactors*: Matrix with prefactors (in columns) and different scenarios in rows
- *vola*: OUTPUT: basket volatility (vector)

5.87 calculate_npv_cpp

`retvec` = *calculate_npv_cpp*(`values`,*df*) [Loadable Function]

Calculate the sum of product of two matrizes along rows.
Input and output variables:

- *values*: Matrix or vector of values
- *df*: Matrix of discount factors
- *retvec*: Result: column vector with sums of product of each columns

5.88 gammainc_lentz_vec

`f` = *gammainc_lentz_vec*(*x*,*a*) [Loadable Function]

Continued fraction for incomplete gamma function (vectorized version). This function should be called from function gammainc_vec.m only.
Input and output variables:

- *x*: x value to calculate cumulative gamma distribution
- *a*: shape parameter
- *f*: return value of gamma cdf at x for a

5.89 interpolate_cubestruct

`retvec` = *interpolate_cubestruct*(`struct`,*xx*, *yy*, *zz*) [Loadable Function]

Interpolate cube values from all elements of an structure array of all cube fieldnames.

This function should be called from getValue method of Surface class which handles all input and ouput data. Please note that axis values needs to be sorted.
Input and output variables:

- *struct*: structure with the following fields (struct)
  - *id*: scenario ID (string)
  - *cube*: volatility cube (one volatility value per x,y,z coordinate) (NDArray)
  - *x_axis*: sorted vector of x-axis values (NDArray)
  - *y_axis*: sorted vector of y-axis values (NDArray)
  - *z_axis*: sorted vector of z-axis values (NDArray)
- *xx*: x coordinate used for interpolation of cube value
- *yy*: y coordinate used for interpolation of cube value
- *zz*: z coordinate used for interpolation of cube value (scalar or vector)
- *retvec*: return vector of scenario dependent interpolated volatility values (NDArray)

5.90 interpolate_curve_vectorized

`retvec =` [Loadable Function]
interpolate_curve_vectorized(`nodes`,`rates`,,`timesteps`)

Linear interpolation of a vector of timesteps for all scenarios of a rate matrix.
This function should be called from interpolate_curve only which handles all input and ouput data.
Input and output variables:

- *nodes*: Column vector of nodes
- *rates*: Matrix of column vector with rates (scenario = rows)
- *timesteps*: Column vector of timesteps to interpolate.
- *retvec*: Result: Matrix with interpolated rates (rows) for each timestep (column)

5.91 interpolate_curve_vectorized_mc

`retvec =` [Loadable Function]
interpolate_curve_vectorized_mc(`nodes`,`rates`,`timestep`)

Linear interpolation of a single timestep for all scenarios of a rate matrix.
This function should be called from interpolate_curve only which handles all input and ouput data.
Input and output variables:

- *nodes*: row vector of nodes
- *rates*: Matrix of column vector with rates (scenario = rows)
- *timestep*: Timestep to interpolate (integer)
- *retvec*: Result: Matrix with interpolated rates (rows) for each timestep (column)

5.92 interpolate_curvestruct

`retvec = ` *interpolate_curvestruct*(`struct`,*xx*) [Loadable Function]

Interpolate curve values from all elements of an structure array of all cube fieldnames. This function should be called from getValue method of Surface class which handles all input and ouput data. Please note that axis values needs to be sorted.

Input and output variables:

- *struct*: structure with the following fields (struct)
  - *id*: scenario ID (string)
  - *cube*: curve cube (one curve value per x coordinate) (NDArray)
  - *x_axis*: sorted vector of x-axis values (NDArray)
- *xx*: x coordinate used for interpolation of cube value (scalar or vector)
- *retvec*: return vector of scenario dependent interpolated volatility values (NDArray)

5.93 interpolate_surfacestruct

`retvec = ` *interpolate_surfacestruct*(`struct`,*xx*, *yy*) [Loadable Function]

Interpolate surface values from all elements of an structure array of all surface fieldnames.

This function should be called from getValue method of Surface class which handles all input and ouput data. Please note that axis values needs to be sorted.

Input and output variables:

- *struct*: structure with the following fields (struct)
  - *id*: scenario ID (string)
  - *surface*: volatility surface (one volatility value per x,y,z coordinate) (NDArray)
  - *x_axis*: sorted vector of x-axis values (NDArray)
  - *y_axis*: sorted vector of y-axis values (NDArray)
- *xx*: x coordinate used for interpolation of surface value
- *yy*: y coordinate used for interpolation of surface value (scalar or vector)
- *retvec*: return vector of scenario dependent interpolated volatility values (NDArray)

5.94 optimize_basket_forwardprice

`FSl = ` *optimize_basket_forwardprice*(`weights`, [Loadable Function]
S, *riskfree*, *r*, *sigma*, *K*, *lbound*, *ubound*) , *maxiter*)

Compute the optimized forward price of basket options according to Beisser et al.

This function is called by the script pricing_basket_options.

Published in: 'Pricing of arithmetic basket options by conditioning', G. Deelstra et al., Insurance: Mathematics and Economics 34 (2004) Pages 55ff

This function solves equation (33) of aforementioned paper for FSl(K) with bisection method for root finding.

Input and output variables:

- *weights*: weights of instruments in basket (vector)
- *S*: instrument spot prices (matrix: scenarios (rows), instruments (columns))
- *riskfree*: riskfree interest rate (vector)
- *r*: Weighted correlation coefficients (matrix: scenarios (rows), instruments (columns))
- *sigma*: instruments volatilities (matrix: scenarios (rows), instruments (columns))
- *K*: option strike values (vector)
- *lbound*: lower bound for root finding (suggestion: 0) (scalar)
- *ubound*: upper bound for root finding (suggestion: 1) (scalar)
- *maxiter*: maximum iterations in bisection method (set to 100) (scalar)
- *limit*: optimization limit for root finding (scalar)
- *FSl*: OUTPUT: Forward price (vector)

5.95 pricing_callable_bond_cpp

`Put Call` = *pricing_callable_bond_cpp*(*T*, *N*, *alpha*, *sigma_vec*, *cf_dates*, *cf_matrix*, *R_matrix*, *dt*, *Timevec*, *notional*, *Mat*, *K*) [Loadable Function]

Compute the put or call value of a bond option based on the Hull-White Tree. This function should be called from Octave script option_bond_hw.m which handles all input and ouput data. References:

- Hull, Options, Futures and other derivatives, 6th Edition
- Clewlow and Strickland, Implementing Derivatives Models, Page 255ff,. Chapter 9: Constructing Trinomial Trees for the short rate, 1st Edition

Input and output variables:

- *call_flag*: Boolean (true: call option, false: put option
- *T*: Bond Maturity in years
- *N*: Number of cash flow dates / call dates
- *alpha*: mean reversion parameter of Hull-White model
- *sigma_vec*: scenario dependent volatility
- *cf_dates*: row vector with cash flow dates (in days)
- *cf_matrix*: scenario dependent cash flow values
- *R_matrix*: scenario dependent discount rates for each cf date)
- *dt*: row vector with time steps between call dates
- *Timevec*: row vector with timesteps of cf_dates and a year after
- *notional*: bond notional)
- *Mat*: cash flow index of options maturity date
- *K*: Strike value
- *accr_int_mat*: scenario dependent interest cash flow values
- *american*: Boolean (true: american option, false: european option
- *Put*: OUTPUT: Putprices (vector)
- *Call*: OUTPUT: Callprices (vector)

5.96 pricing_option_cpp

***OptionVec* = *pricing_option_cpp*(*option_type*, *call_flag*, *S_vec*, *X_vec*, *T_vec*, *r_vec*, *sigma_vec*, *divrate_vec*, *n*)** [Loadable Function]

Compute the put or call value of different equity options.

This function should be called from Option class which handles all input and ouput data.

Input and output variables:

- *option_type*: Integer: Sets pricing engine (1=EU(BS),2=AM(CRR),3=ASIAN ARITHMETIC(MC))
- *call_flag*: Boolean: (true: call option, false: put option
- *S_vec*: Double: Spot prices (either scalar or vector of length m)
- *X_vec*: Double: Strike prices (either scalar or vector of length m)
- *T_vec*: Double: Time to maturity (days, act/365) (either scalar or vector of length m)
- *r_vec*: Double: riskfree rate (either scalar or vector of length m)
- *sigma_vec*: Double: volatility (annualized,act/365) (either scalar or vector of length m)
- *divrate_vec*: Double: dividend yield (cont,act/365) (either scalar or vector of length m)
- *n*: Integer: number of tree steps (AM) or number of MC scenarios (ASIAN)
- *OptionVec*: Double: OUTPUT: Option prices (columnn vector)

Example Call:

```
retvec = pricing_option_cpp(1,false,[10000;9000;11000],11000,365,0.01,[0.2;0.025;
retvec =
   1351.5596280726359
   1890.5481712408509
     83.4751762658461
```

Index