ORE Scripted Trade Module

Quaternion Risk Management

12 February 2024

Document History

Date	Author	Comment
16-04-2019	Peter Caspers	initial version
24-07-2019	Peter Caspers	phase 1 draft
02-10-2019	Peter Caspers	add interest rate indices
21-10-2019	Peter Caspers	add SORT and PERMUTE functions
29-10-2019	Peter Caspers	add product tag, local vol model
28-11-2019	Peter Caspers	add LOGPAY function and product tag
05-12-2019	Peter Caspers	add free style xml
06-01-2020	Peter Caspers	add HISTFIXING function
09-01-2020	Peter Caspers	add FWDCOMP function
16-01-2020	Peter Caspers	updates for ScheduleCoarsening, GaussianCam and LocalVol
13-02-2020	Peter Caspers	updates for Commodity
22-04-2020	Peter Caspers	add ABOVEPROB, BELOWPROB functions
18-08-2020	Peter Caspers	add DATEINDEX function
21-09-2020	Peter Caspers	add specifics on AMC
06-10-2020	Peter Caspers	extended version of LOGPAY() function
03-12-2020	Peter Caspers	additions for FD and calibration
01-07-2021	Peter Caspers	add case $d1 > d2$ for ABOVEPROB, BELOWPROB
13-09-2021	Nathaniel Volfango	update Index section and add note on payment currency
11-02-2022	Peter Caspers	extend FWDCOMP, add FWDAVG function (breaking change)
22-06-2023	Peter Caspers	add config flags for external devices
12-02-2024	Peter Caspers	additions for FD GaussianCam model

Breaking Changes

Date	Breaking Change
11-02-2022	If the (optional) spread is given in FWDCOMP(), a gearing must be given too

Contents

1 Summary

This document describes the ORE+ trade scripting module.

2 Trade Representation

2.1 General Structure

A scripted trade comprises

- external data that parametrises the payoff script
- the payoff script

In the following example, a plain vanilla equity call or put option is described by providing the expiry and settlement date, the strike, a put / call indicator and the underlying index as external data in the `Data` node, and the payoff script is referenced by the `ScriptName` node:

```
<Trade id="VanillaOption">
  <TradeType>ScriptedTrade</TradeType>
  <Envelope/>
  <ScriptedTradeData>
    <ScriptName>EuropeanOption</ScriptName>
    <Data>
      <Event>
        <Name>Expiry</Name>
        <Value>2020-02-09</Value>
      </Event>
      <Event>
        <Name>Settlement</Name>
        <Value>2020-02-15</Value>
      </Event>
      <Number>
        <Name>Strike</Name>
        <Value>1200</Value>
      </Number>
      <Number>
        <Name>PutCall</Name>
        <Value>1</Value>
      </Number>
      <Index>
        <Name>Underlying</Name>
        <Value>EQ-RIC:.SPX</Value>
      </Index>
      <Currency>
        <Name>PayCcy</Name>
        <Value>USD</Value>
      </Currency>
    </Data>
  </ScriptedTradeData>
</Trade>
```

The script itself is defined in a script library, which is part of the static data setup of the application like bond or credit index static data, as follows

```
<ScriptLibrary>
  <Script>
    <Name>EuropeanOption</Name>
    <ProductTag>SingleAssetOption({AssetClass})</ProductTag>
    <Script>
      <Code><![CDATA[
          Option = PAY(max( PutCall * (Underlying(Expiry) - Strike), 0 ),
                                                  Expiry, Settlement, PayCcy);
        ]]></Code>
      <NPV>Option</NPV>
      <Results>
        <Result>ExerciseProbability</Result>
      </Results>
    </Script>
  </Script>
  <Script>
  ...
  </Script>
  ...
</ScriptLibrary>
```

Many scripted trades can thus share the same script, and the payoff scripts can be managed and maintained centrally. Each scipt defines an exotic product type. And adding a product type to the library is a configuration rather than a code release change. For the ORE CLI the script library can be loaded by specifying the (optional) scriptLibrary parameter:

```
<ORE>
  <Setup>
  <Parameter name="asofDate">2016-02-05</Parameter>
  ...
  <Parameter name="scriptLibrary">scriptlibrary.xml</Parameter>
  </Setup>
  ...
</ORE>
```

In the pricer_app the script library can be loaded by specifying the (optional) script_library parameter, e.g. in config.txt

```
asof=2018-12-31
base_currency=USD
trade=scripted_fx_onetouch_option.xml
script_library=scriptlibrary.xml
```

Alternatively, the script can be inlined in the trade representation:

```
<Trade id="VanillaOption">
  <TradeType>ScriptedTrade</TradeType>
  <Envelope/>
  <ScriptedTradeData>
    <ProductTag>SingleAssetOption({AssetClass})</ProductTag>
    <Script>
      <Code><![CDATA[
          Option = PAY(max( PutCall * (Underlying(Expiry) - Strike), 0 ),
                                                  Expiry, Settlement, PayCcy);
      ]]></Code>
      <NPV>Option</NPV>
```

```
      <Results>
          <Result>ExerciseProbability</Result>
      <Results>
    </Script>
    <Data>
      ...
    </Data>
  </ScriptedTradeData>
</Trade>
```

The product tag node is optional both for scripts in the library and inlined script and used for the assignment of a pricing model and its parameters, see 4.2

2.2 Script Node

A script is described by

- the script code (in the `Code` node)
- the name of the variable used to populate the instrument's NPV result field (in the `NPV` node)
- an optional list of variables used to populate the additional result map in an instrument
- an optional specification of calibration strikes, see 4.8 for details
- an optional ScheduleCoarsening node specifying which schedules can be coarsened, see 4.7 for details on this
- an optional NewSchedules node specifying new schedules should be created before the script execution, this node contains
  - Name: a name for the new schedule to be created
  - Operation: an operation to perform, only Join is supported currently
  - Schedules: a list of source schedules
- an optional StickyCloseOutStates node specifying variables that should be held constant during an AMC exposure run with sticky close-out mpor mode, usually a list of exercise / barrier hit indicators, see 2.13 for more details on this
- an optional ConditionalExpectation node specifying a filter on model states to be used in NPV() and NPVMEM() functions. The filter is specified as a subnode ModelStates with one or several ModelState subnodes "Asset" (use EQ, FX, COMM components), "IR" (use interest rate states), "INF" (use inflation states). Applies to GaussianCam model only. If left empty, the full model state is used for conditional npv calculation.

Several script nodes can be used in parallel and are distinguished by an optional `purpose` attribute then. There must always be a script with empty purpose. Special purposes are

- `FD`: If a script with purpose FD is available this is preferred over the default script when an FD engine is built (as opposed to an MC engine)
- `AMC`: If a script with purpose AMC is available this is preferred over the default script when an AMC engine is built, i.e. an engine used within the AMC analytics type. See 2.13 for more details.

The keys in the instrument's additional results map are by default identical to the variable names used to populate them. They can be given a different name using the optional rename attribute. The variables can be scalars or arrays of any type which will be translated to QuantLib instrument result types `double` (for NUMBER), `QuantLib::Date` (for EVENT), `string` (for INDEX, CURRENCY, DAYCOUNTER) for scalars or vectors thereof for arrays.
The following additional results have a special meaning.

- `currentNotional` a number representing the current notional of a trade, if given this is displayed in the NPV report and - importantly - required as an input for trades which fall under the IM Schedule approach
- `notionalCurrency` the currency in which the currentNotional is given

```
<Script purpose="">
  <Code><![CDATA[
        NUMBER Payoff, ExerciseProbability;
        Payoff = PutCall * (Underlying(Expiry) - Strike);
        Option = LongShort * Quantity * PAY( max( Payoff, 0 ), Expiry, Settlement, PayCcy);
        IF Payoff > 0.0 THEN
            ExerciseProbability = 1;
        END;
    ]]></Code>
  <NPV>Option</NPV>
  <Results>
    <Result>ExerciseProbability</Result>
    <Result>currentNotional</Result>
    <Result rename="notionalCurrency">PayCcy</Result>
  </Results>
  <CalibrationSpec>
    <Calibration>
      <Index>Underlying</Index>
      <Strikes>
        <Strike>Strike</Strike>
      </Strikes>
    </Calibration>
  </CalibrationSpec>
  <ScheduleCoarsening/>
  <NewSchedules>
    <NewSchedule>
      <Name>ExerciseAndSimDates</Name>
      <Operation>Join</Operation>
      <Schedules>
        <Schedule>_AMC_SimDates</Schedule>
        <Schedule>ExerciseDates</Schedule>
      </Schedules>
    </NewSchedule>
  </Newschedules>
  <StickyCloseOutStates>
```

```
      <StickyCLoseOutState>ExerciseIndicator</StickyCloseOutState>
    </StickyCloseOutstates>
    <ConditionalExpectation>
      <ModelStates>
        <ModelState>Asset</ModelState>
      </ModelStates>
    </ConditionalExpectation>
  </Script>
```

2.3 Data Node

The `Data` node contains definitions for external variables that can be used in the script. These variables can either be scalars or one-dimensional arrays with fixed size. The supported data types are

- Event: a date
- Number: a real number
- Counter: an integer
- Index: an EQ, FX or COMM index
- Currency: a currency
- Daycounter: a day count convention

2.4 Event

An event is defined either as a scalar

```
<Event>
  <Name>Expiry</Name>
  <Value>2020-02-09</Value>
</Event>
```

or as an array of dates using ORE's ScheduleData node, e.g.

```
<Event>
  <Name>ExerciseDates</Name>
  <ScheduleData>
    <Rules>
     <StartDate>2016-02-06</StartDate>
     <EndDate>2016-05-06</EndDate>
     <Tenor>1D</Tenor>
     <Calendar>TARGET,US</Calendar>
     <Convention>F</Convention>
     <Rule>Forward</Rule>
   </Rules>
  </ScheduleData>
</Event>
```

using a rule based schedule or

```
<Event>
  <Name>ValuationDates</Name>
  <ScheduleData>
    <Dates>
      <Dates>
        <Date>2018-03-10</Date>
        <Date>2019-03-10</Date>
        <Date>2020-03-10</Date>
        <Date>2021-03-10</Date>
        <Date>2022-03-10</Date>
        <Date>2023-03-10</Date>
        <Date>2024-03-11</Date>
      </Dates>
    </Dates>
  </ScheduleData>
</Event>
```

using a list of dates. An array of dates can also be deduced from another, previously defined array by specifying a shift rule which is useful e.g. to generate fixing or payment schedules from accrual schedules, or notification and settlement schedules from exercise date schedules. Example:

```
<Event>
  <Name>FixingDates</Name>
  <DerivedSchedule>
    <BaseSchedule>AccrualDates</BaseSchedule>
    <Shift>-2D</Shift>
    <Calendar>TARGET</Calendar>
    <Convention>MF</Convention>
  </DerivedSchedule>
</Event>
```

2.5 Number

A scalar number is defined as

```
<Number>
  <Name>Strike</Name>
  <Value>2147.56</Value>
</Number>
```

and likewise an array of numbers as

```
<Number>
  <Name>EquityNotionalAmount</Name>
  <Values>
    <Value>100000000</Value>
    <Value>90000000</Value>
    <Value>80000000</Value>
  </Values>
</Number>
```

2.6 Index

Indices are defined as

```
<Index>
  <Name>Underlying</Name>
  <Value>EQ-RIC:.SPX</Value>
</Index>
```

in case of a scalar and

```
<Index>
  <Name>Underlyings</Name>
  <Values>
    <Value>EQ-UND1</Value>
    <Value>EQ-UND2</Value>
    <Value>EQ-UND3</Value>
    <Value>EQ-UND4</Value>
    <Value>EQ-UND5</Value>
  </Values>
</Index>
```

in case of a vector. Currently, Equity, FX, Commodity, Interest Rate and Inflation indices as well as generic indices are supported, the naming convention follows the standard ORE conventions, i.e.

- Equity: These are declared based on identifier type
  - `EQ-ISIN:{Name}:{Currency}:{Exchange}` for ISIN equities, e.g. `EQ-ISIN:NL0000852580:EUR:XAMS`,
  - `EQ-RIC:{Name}` for RIC equities, e.g. `EQ-RIC:.SPX`,
  - `EQ-FIGI:{Name}` for FIGI equities, e.g. `EQ-FIGI:BBG000BLNNV0`,
  - `EQ-BBG:{Name}` for BBG equities, e.g. `EQ-BBG:BARC LN Equity`.
- FX: `FX-SOURCE-CCY1-CCY2` with foreign currency `CCY1` and domestic currency `CCY2`. The order of the currencies here is important as they will have different meanings. For a natural payoff, `CCY2` must correspond to the instrument's payment currency. Otherwise, we have a quanto payoff. See section 2.10 for more information.
- Commodity: These are declared as `COMM-{Name}`, with possible extensions as follows (see below also for more information on the meanings of the extensions to the commodity indices):
  - `COMM-NYMEX:CL` (spot),
  - `COMM-NYMEX:CL-2020-09` (future with expiry 01 Sep 2020),
  - `COMM-NYMEX:CL-2020-09-15` (future with expiry 15 Sep 2020).
- Interest Rate: These are declared as `CCY-INDEX-TENOR`, e.g. `EUR-EURIBOR-6M`, `EUR-CMS-10Y`.
- Inflation: These are declared as `{Name}`, with possible extensions as follows(see below also for more information on the meanings of the extensions to the inflation indices):
  - `EUHICPXT` (constant/non-interpolated),

  - `EUHICPXT#F` (flat interpolation),
  - `EUHICPXT#L` (linear interpolation).

- Generic: These are declared as `GENERIC-{Name}`.

Notice that generic indices only provide historical fixings.
Within the scripting framework there are additional ways to reference commodity indices in a more dynamical fashion: When evaluating a commodity index using the index evaluation operator (see 3.8) as in

```
CommodityUnderlying(ObservationDate)
```

for

- a variable of the form `COMM-{Name}#N`, the index will be resolved to the $N+$ 1th future with expiry greater than ObservationDate for the given commodity underlying, $N \geq 0$. The parameter N corresponds to the field `FutureMonthOffset` commonly used in commodity trade xml schemas
- the above form can take one or two additional parameters `COMM-{Name}#N#D` or `COMM-{Name}#N#D#{Cal}` where D corresponds to `DeliveryRollDays` and `Cal` is the calendar used to roll the observation date forward before the next future expiry is looked up. If `Cal` is not given, it defaults to the null calendar (no holidays).
- a variable of the form `COMM-{Name}!N`, the index will be resolved to the Nth future relative to the month and year of the ObservationDate.

In general, if a commodity future is referenced, the observation date should be be less or equal to the expiry date of the future, since no historical fixing will in general be available after the future expiry date. In the future-looking simulation the observation of an expired future is possible, in this case the (model) value of the future is kept constant at its value at the expiry date for observation dates after the expiry date.
When evaluating an inflation index as in

```
CPIIndex(FixingDate)
```

the variable `CPIIndex` can be given in the following ways:

- a variable of the form `EUHICPXT` will evaluate the standard ORE inflation index, which is non-interpolated. This means the result will be constant for all FixingDates in an inflation period (i.e. usually constant for a calendar month)
- a variable of the form `EUHICPXT#F`, i.e. an ORE inflation index name followed by `#F` indicating a flat interpolation of the index, this form is equivalent to the previous form making the interpolation mode explicit as "flat"
- a variable of the form `EUHICPXT#L` where the suffix `#L` indicates linear interpolation of the index, i.e. the result will be the interpolated fixing between the usually monthly inflation index fixings

The use of the interpolation extensions is deprecated. Interpolation of fixings should be handled in the payoff script. The CPI indices are forecasting a flat fixing for given the inflation period. The use of interpolation is still supported but will be eventually removed in a later release.

2.7 Currency

A scalar currency variable is defined as

```
<Currency>
  <Name>PayCcy</Name>
  <Value>USD</Value>
</Currency>
```

and an array of currencies as

```
<Currency>
  <Name>BasketCurrencies</Name>
  <Values>
    <Value>USD</Value>
    <Value>GBP</Value>
    <Value>JPY</Value>
  </Value>
</Currency>
```

2.8 Daycounter

A scalar day count convention variable is defined as

```
<Daycounter>
  <Name>AccrualDayCounter</Name>
  <Value>Actual/360</Value>
</Daycounter>
```

and an array of day counters as

```
<Daycounter>
  <Name>LegAccrualDayCounters</Name>
  <Values>
    <Value>30/360</Value>
    <Value>Actual/360</Value>
  </Value>
</Currency>
```

2.9 Compact Trade XML

In addition to the trade xml described in 2.1, 2.2, 2.3 we support an alternative, more compact, format where the variable names are derived from the node names and the type is given by an attribute. The script must sit in the script library in this case (i.e. inlining is not possible) and is referenced via a name derived from the root node of the trade data. Consider the following example of a one touch option in the original format:

```
<Trade id="SCRIPTED_FX_ONE-TOUCH_OPTION">
  <TradeType>ScriptedTrade</TradeType>
  <Envelope>
    <CounterParty>CPTY_A</CounterParty>
    <NettingSetId>CPTY_A</NettingSetId>
    <AdditionalFields/>
  </Envelope>
  <ScriptedTradeData>
    <ScriptName>OneTouchOption</ScriptName>
```

```
    <Data>
      <Event>
        <Name>Settlement</Name>
        <Value>2020-08-01</Value>
      </Event>
      <Event>
        <Name>ObservationDates</Name>
        <ScheduleData>
          <Rules>
            <StartDate>2018-12-28</StartDate>
            <EndDate>2020-08-01</EndDate>
            <Tenor>1D</Tenor>
            <Calendar>US</Calendar>
            <Convention>U</Convention>
            <TermConvention>U</TermConvention>
            <Rule>Forward</Rule>
          </Rules>
        </ScheduleData>
        <ApplyCoarsening>true</ApplyCoarsening>
      </Event>
      <Number>
        <Name>BarrierLevel</Name>
        <Value>0.009</Value>
      </Number>
      <Number>
        <Name>Type</Name>
        <Value>-1</Value>
      </Number>
      <Number>
        <Name>LongShort</Name>
        <Value>1</Value>
      </Number>
      <Number>
        <Name>Amount</Name>
        <Value>10000000</Value>
      </Number>
      <Currency>
        <Name>PayCcy</Name>
        <Value>USD</Value>
      </Currency>
      <Index>
        <Name>Underlying</Name>
        <Value>FX-TR20H-USD-JPY</Value>
      </Index>
    </Data>
  </ScriptedTradeData>
</Trade>
```

In the compact format the same trade looks like this:

```
<Trade id="SCRIPTED_FX_ONE-TOUCH_OPTION">
  <TradeType>ScriptedTrade</TradeType>
  <Envelope>
    <CounterParty>CPTY_A</CounterParty>
    <NettingSetId>CPTY_A</NettingSetId>
    <AdditionalFields/>
  </Envelope>
  <OneTouchOptionData>
    <Settlement type="event">2020-08-01</Settlement>
```

```
    <ObservationDates type="event">
      <ScheduleData>
        <Rules>
          <StartDate>2018-12-28</StartDate>
          <EndDate>2020-08-01</EndDate>
          <Tenor>1D</Tenor>
          <Calendar>US</Calendar>
          <Convention>U</Convention>
          <TermConvention>U</TermConvention>
          <Rule>Forward</Rule>
        </Rules>
      </ScheduleData>
      <ApplyCoarsening>true</ApplyCoarsening>
    </ObservationDates>
    <BarrierLevel type="number">0.009</BarrierLevel>
    <BarrierType type="barrierType">DownIn</BarrierType>
    <LongShort type="longShort">Long</LongShort>
    <Amount type="number">10000000</Amount>
    <PayCcy type="currency">USD</PayCcy>
    <Underlying type="index">FX-TR20H-USD-JPY</Underlying>
  </OneTouchOptionData>
</Trade>
```

The supported types that must be specified in the `type` attribute are `number`, `event`, `currency`, `dayCounter` and `index`. In addition we support some custom types that are mapped to numbers internally and allow for a more natural representation of the trade:

- `bool` with a mapping `true` $\mapsto$ 1, `false` $\mapsto$ -1
- `optionType` with a mapping `Call` $\mapsto$ 1, `Put` $\mapsto$ -1, `Cap` $\mapsto$ 1, `Floor` $\mapsto$ -1
- `longShort` with a mapping `Long` $\mapsto$ 1, `Short` $\mapsto$ -1
- `barrierType` with a mapping `DownIn` $\mapsto$ 1, `UpIn` $\mapsto$ 2, `DownOut` $\mapsto$ 3, `UpOut` $\mapsto$ 4

Arrays of events are specified as in the example above (ObservationDates), for the other types the values are listed in value tags, e.g. an array of numbers is declared as

```
<MyNumberArray type="number">
  <Value>100.0</Value>
  <Value>200.0</Value>
  <Value>200.0</Value>
</MyNumberArray>
```

2.10 Payment Currency

The payoffs described in the Trade Specific Data for scripted trades usually involve a payment currency. Unless stated otherwise, this currency represents the *natural* payoff currency, i.e.

- for equity and commodity underlyings this should be the currency in which the underlying price is quoted
- for FX underlyings `FX-SOURCE-CCY1-CCY2` this should be the domestic (target, numeraire) currency `CCY2`.

If the payment currency is set to a different currency on the other hand, the resulting payoff is a true quanto payoff, i.e. the amount of the payoff is determined on the natural currency, but paid in a different currency *without* converting the amount to this latter currency using the fair FX Spot rate on the settlement date.
We are aware that a conversion from the natural payoff currency to a different settlement currency using the fair FX Spot rate is sometimes part of the terms and conditions of a trade. This conversion has no or small impact on the valuation and risk profile of a trade though and is therefore usually not part of the payoff modeling.
As an example consider a Forward Volatility Agreement on the FX pair GBP-EUR. The `PayCcy` should be set to EUR in this case, even if the forward premium is settled in GBP as it is usual market practice for this pair.
Notice that the above does not apply to fixed premiums, where the premium is given as a fixed number in the trade xml together with a premium currency in which this amount should be paid.

2.11 Convenience Trade Wrappers

We provide a number of trade wrappers that read a "normal" ORE XML and translates this internally to the script data structure. Examples include the trade types

- DoubleDigitalOption
- PerformanceOption_01
- Autocallable_01.

The implementation of a wrapper is usually done by inheriting from ScriptedTrade and implementing the fromXML(), toXML() and build() methods appropriately, see one of the classes on how this is done in detail.

2.12 SIMM product type deduction

The SIMM product type is deduced as follows

- if at least one commodity index is present in the script, it is set to "Commodity"
- else, if at least one equity index is present in the script, it is set to "Equity"
- otherwise it is set to "RatesFX"

2.13 Scripting and AMC

Scripted trades can be used in combination with AMC analytics type. Usually a separate script will be used within this analytics type because of the special inputs and outputs of AMC scripts as outlined below. To set up a separate script that should be used within the AMC analytics type, the attribute purpose should be given the value `AMC`, i.e. in the script library we could have

```
<Script>
  <Name>BermudanSwaption</Name>
  <!-- default script -->
  <Script>
  ...
  </Script>
  <!-- script that will be preferred in an AMC context -->
  <Script purpose="AMC">
  ...
  </Script>
</Script>
```

and similar for embedded scripts. A script that in run withing the AMC analytics type gets a special input event array `_AMC_SimDates` that contains the dates on which a conditional NPV is required as an output. This output has to be delivered in a number array `_AMC_NPV` which has the same size. Typically one will build a new schedule from the amc sim dates and other event dates using something like

```
<NewSchedules>
  <NewSchedule>
    <Name>ExerciseAndSimDates</Name>
    <Operation>Join</Operation>
    <Schedules>
      <Schedule>_AMC_SimDates</Schedule>
      <Schedule>ExerciseDates</Schedule>
    </Schedules>
  </NewSchedule>
</NewSchedules>
```

within the script node (see 2.2). When looping over the common event dates, the DATEINDEX() function can be used to distinguish the different kind of events.
The AMC analytics type supports several simulation modes w.r.t. mpor grids

- no close-out lag
- close-out lag with mpor mode actual date
- close-out lag with mpor mode sticky date

In the first two cases the `_AMC_SimDate` grid will consist of all valuation and close-out dates specified in the simulation setup. In the last case the `_AMC_SimDate` grid will consist only of the valuation date and two runs will be performed on these dates, one using the original and another one using a time-shifted stochastic process. Since it is not desirable that exercise decisions or barrier hit indicators are recomputed in the second run, it is possible to define a set of variables for which values are computed in the first run and then reused in these second run. Assignments to these variables in the second run are ignored. The definition of such variables is done in the script node (see 2.2):

```
<StickyCloseOutStates>
  <StickyCloseOutState>ExerciseIndicator</StickyCloseOutState>
</StickyCloseOutStates>
```

3 Scripting Language

3.1 Whitespace

Whitespace (space, tab, return, newline) is ignored during the parsing. All variable identifiers and keywords are case sensitive.

3.2 Keywords

The language uses keywords and predefined function names as listed in table 1 which may may not be used as variable identifiers.
In addition the following variable identifiers are automatically populated with special values when running the script engine on a trade:

- `TODAY`: the current evaluation date

3.3 Variables

Variables that can be used in the script are either

- externally defined variables, defined in the data node of the trade xml representation
- variables local to the script, declared within the script

Externally defined variables are protected from being modified by the script. All variables used within the script must be either externally defined or declared at the top of the script using

```
NUMBER continuationValue, exerciseValue, x[10];
```

which declares two scalars `continuationValue` and `exerciseValue` and an array `x` of size 10 (see 3.4 for more details on arrays). The only exemption to this rule is the variable declared in the `NPV` node of the script, which is definied implicitly as a scalar number.
Notice that within the script only variables of type Number can be declared.[1] All variables are initialised with 0. The scope of a variable declaration is always global to the script, multiple declarations of the same variable name are forbidden.
Variable identifiers are subject to the following restrictions

- must start with a character, then characters or numbers or underscores may follow
- no other special characters, no keywords or predefined functions allowed (see 3.2)
- e.g. `x` or `x_23`, `aValue` are valid identifiers
- identifiers starting with an underscore are technically allowed as well, but reserved for special use cases (e.g `_AMC_SimDates` and `_AMC_NPV` for AMC exposure generation)

[1] this restriction allows the static analysis of the script, see 6.1

Keyword	Context
IF	Control Flow
THEN	
ELSE	
END	
FOR	
IN	
DO	
NUMBER	Type Identifiers
OR	Logical Operators
AND	
abs	Functions
exp	
ln	
sqrt	
normalCdf	
normalPdf	
max	
min	
pow	
black	
dcf	
days	
PAY	Model dependent functions
LOGPAY	
NPV	
NPVMEM	
HISTFIXING	
DISCOUNT	
FWDCOMP	
FWDAVG	
ABOVEPROB	
BELOWPORB	
SIZE	Other Statements
DATEINDEX	
REQUIRE	
SORT	
PERMUTE	

Table 1: Reserved keywords.

3.4 Arrays, SIZE operator

Arrays are declared by specifying the size of the array in sqaure brackets, e.g.

```
NUMBER x[10], y[SIZE(ObservationDates)], z[5+3*v];
```

declares arrays

- `x` of size 10
- `y` with the same size as the array `ObservationDates`
- `z` with size $5 + 3v$ where v is a number variable

Once an array is delcared its size can not be changed. The ith element of an array a is accessed by `a[i]`, where i is an expression evaluating to a number. Here $i = 1, 2, 3, \ldots, n$, where n is the fixed size of the array, i.e. the subscripts start at 1 (as opposed to 0 as in some other languages).
The size of an array `a` can be evaluated by `SIZE(a)`. Only one dimensional arrays are supported. The array subscript must be *deterministic*, e.g.

```
IF Underlying > Strike THEN
  i = 1;
ELSE
  i = 2;
END;
Payoff = y[i];
```

is illegal since i in general will be path-dependent, but

```
IF Underlying > Strike THEN
  Payoff = y[1];
ELSE
  Payoff = y[2];
END;
```

is valid.[2]

3.5 Sorting Arrays: SORT and PERMUTE instructions

Given an array `x` of number type the statement

```
SORT (x);
```

will sort the array (pathwise) in ascending order. The statement

```
SORT (x, y);
```

will write a sorted version of `x` to `y` and leave `x` unchanged. The array `y` must be of number type and have the same size as `x`. The array `y` can also be equal to `x`, the statement `SORT(x,x);` is equivalent to `SORT(x);`. Finally the statement

```
SORT (x, y, p);
```

[2] the background is the simplicity and performance of the engine implementation

will write a sorted version of `x` to `y` and populate another array `p` with indices $1, \ldots,$ `SIZE`(x) such that $x[p[1]], \ldots, x[p[n]]$ is sorted. Here `p` must be an array with the same size as `x` and of number type.
A permutation `p` generated as above (or set up otherwise) can be used to sort an unrelated array `z` using

```
PERMUTE (z, p);
```

which will reorder the values of `z` as $z[1] \to z[p[1]]$, $z[2] \to z[p[2]]$ ... etc. The statement

```
PERMUTE (z, w, p);
```

will do the same, but write the result to `w` and leave `z` untouched.

3.6 Function DATEINDEX

Given an array `a` and a single date `d`, the expression

```
DATEINDEX(d, a, EQ)
```

returns 0 if the date d is not found in the array a and otherwise the (first) index i for which `a[i]` equals `d`. The variable d is required to be of type event. The variable a is only required to be an array, if the type of its elements are not event, the return value will always be zero indicating that d was not found in a. Similarly,

```
DATEINDEX(d, a, GEQ)
```

returns the index of the earliest date in a that is greater or equal than d, and

```
DATEINDEX(d, a, GT)
```

returns the index of the earliest date in a that is greater than d. If no such dates exists for `GEQ` or `GT`, the size of a plus 1 will be returned.

3.7 Instructions

A typical script comprises a sequence of instructions, each one terminated by `;`.

3.8 Index evaluation

Given an variable `index` of type Index its historical or projected fixing at a date d is evaluated using the expression `index(d)`. This is applicable to all index types. For example

```
Underlying(ObservationDate)
```

evaluates the index assigned to the variable Underlying at the date assigned to the variable ObservationDate. For FX, EQ, IR and COMM Spot indices this corresponds to a a fixing at the observation date in the usual sense. For COMM Future indices it is the observed future price at the observation date.
For INF indices the argument is the actual fixing date, which due to availability lags is observed at a later simulation time in models with dynamical inflation simulation. For

example in the GaussianCam model, this lag is defined as the number of calendar days from the zero inflation term structure base date to its reference date (adjusted to the first date of the inflation period to be consistent with the same adjustment applied to the base date). This means that when observing an inflation index at a fixing date d, the model state at $d + \text{lag}$ is used to make this observation.
The extended syntax

```
Underlying(ObservationDate, ForwardDate)
```

evaluates the projected fixing for ForwardDate as seen from ObservationDate.
This is applicable to FX, EQ, IR, INF and COMM Spot indices, but not to COMM Future indices, since for the latter the two concepts coincide (for ForwardDate ¡ Future-Expiry). If a forward date is given for the observation of a COMM future index, no error is thrown, but it will be ignored.
For inflation indices, the ForwardDate will be the actual fixing date again and the ObservationDate will be using a lagged state as explained above.
The ForwardDate must be greater or equal than the ObservationDate. If the Forward-Date is strictly greater than the ObservationDate the ObservationDate must not be a past date (for inflation indices it must not lie before the inflation term structure's base date), since the computation of projected fixings for past dates would involve the knowledge of past curves, i.e. past market data.
Notice also the further specifics of commodity and inflation indices in 2.6.

3.9 Comparisons == and !=

Compares two values, e.g. `x==y` or `x!=y`. This is applicable to all types. For a number the interpretation is "numerically equal".

3.10 Comparisons <, <=, >, >=

Compares two values `x<y`, `x<=y`, `x>y`, `x>=y`. Applicable to numbers and events, but not to currencies or indices. For numbers the interpretation is "less than, but not numerically equal", "less than or numerically equal", etc.

3.11 Operations +, -, *, /

Arithmetic operations `x+y`, `x-y`, `x*y`, `x/y`, applicable to numbers only.

3.12 Assignment =

Assignment `x = y`, only allowed for numbers within the script.

3.13 Logical Operators AND, OR, NOT

Connects results of comparisons or other logical expressions:

- `x<y AND z!=0`
- `x<y OR z!=0`

- `NOT(x==y)`
- AND has higher precedence than OR, e.g.
- `x<y AND y<z OR z!=0` same as {`x<y AND y<z`} `OR z!=0`, but
- `x<y AND` {`y<z OR z!=0`} requires parenthesis
- better always use parenthesis when mixing `AND` / `OR`

3.14 Conditionals: IF ... THEN ... ELSE ...

Conditional execution can be written as

```
IF condition THEN
    ... if-body ...
ELSE
    ... else-body ...
END
```

Examples:

```
IF x == y THEN
  z = PAY(X,d,p,ccy);
  w = 1;
END;
```

```
IF x == y THEN
  z = PAY(X,d,p,ccy);
ELSE
  z = 0;
  w = 0;
END;
```

where the `ELSE` part is optional. The body can comprise one ore more instructions, each of which must be terminated by `;`.

3.15 Loops: FOR ... IN ... DO

Loops are written as

```
FOR i IN (a,b,s) DO
... body ...
END
```

where `i` is a number variable identifier, and `a`, `b`, `s` are expressions that yield a result of type Number. The variable `i` must have been declared in the script before it can be used as a loop variable. The code in the body is executed for the values $i = a, a + s, \ldots$ until $a + ks > b$ if $s > 0$ or $a + ks < b$ if $s < 0$ for some integer $k > 0$. All values a, b, s must be integers and $s \neq 0$.
Example:

```
NUMBER i,x;
FOR i IN (1,100.1) DO x = x + i; END;
```

Here `a, b` must be deterministic, `i` must not be modified in the loop body. If a or b are modified in the loop body, still the initial values read at the start of the loop are used. The loop body can comprise one or more instructions, each of which must be terminated by `;`.

3.16 Special variable: TODAY

A constant event variable, set to the current evaluation date. This can e.g. be used to restrict exercise decisions to future dates, see 3.26 for an example.

3.17 Checks: REQUIRE

If the condition `C` is not true, a runtime error is thrown. Examples:

- `REQUIRE SIZE(ExerciseDates) == SIZE(SettlementDates);`
- `REQUIRE SIZE(Underlyings) == 2;`
- `REQUIRE Strike >= 0;`

3.18 Functions `min`, `max`, `pow`

Binary functions `min(x,y)`, `max(x,y)`, `pow(x,y)`, applicable to numbers only.

3.19 Functions `-`, abs, exp, ln, sqrt

Unary functions `-x`, `abs(x)`, `exp(x)`, `ln(x)`, `sqrt(x)`, applicable to numbers only.

3.20 Functions `normalPdf`, `normalCdf`

Returns the standard normal pdf $\phi(x)$ resp. cdf $\Phi(x)$, applicable to numbers only.

3.21 Function `black`

Implements the black formula `black(omega, obs, expiry, k, f, sigma)` with

$$\begin{aligned} \text{black} &= \omega \cdot (f\Phi(\omega d_1) - k\Phi(\omega d_2)) \\ d_{1,2} &= \frac{\ln(f/k) \pm \frac{1}{2}\sigma^2 t}{\sigma\sqrt{t}} \end{aligned}$$

where t is the (model's) year fraction between obs and expiry date, i.e.:

- `omega` is 1 (call) or -1 (put)
- `obs, expiry` are the observation / expiry dates
- `k, f` are the strike and the forward
- `sigma` is the implied volatility
- notice that no discounting is applied

3.22 Function `dcf`

The expression `dcf(dc, d1, d2)` returns the day count fraction for a day count convention `dc` and a period defined by dates `d1` and `d2`.

3.23 Function `days`

The expression `days(dc, d1, d2)` returns the number of days between `d1` and `d2` for a day count convention `dc`.

3.24 Function `PAY`

The expression `PAY(X, d, p, ccy)` calculates a discounted payoff for an amount X observed at a date d, paid at a date p in currency ccy, i.e.

$$\frac{XP_{ccy}(d,p)FX_{\text{ccy,base}}(d)}{N(d)} \tag{1}$$

where

- here P_{ccy} is the discount factor in currency ccy, FX is the FX spot from ccy to base and N is the model numeraire
- $d \leq p$ must hold
- if p lies on or before the evaluation date, the result is zero; X is not evaluated in this case. Note that X is evaluated in the LOGPAY function if past cashflows are included, see 3.25.
- avoids reading non-relevant past fixings from the index history
- if d lies before (but p after) the evaluation date, it is set to the evaluation date, i.e. the result is computed as of the evaluation date

3.25 Function `LOGPAY`

The expression `LOGPAY(X, d, p, ccy)` has the same meaning as `PAY(X, d, p, ccy)` (see 3.24) but as a side effect populates an internal cashflow log that is used to generate expected flows. The generated flow is

$$\frac{N(0)E\left(\frac{XP_{ccy}(d,p)FX_{\text{ccy,base}}(d)}{N(d)}\right)}{FX_{\text{ccy,base}}(0)P_{ccy}(0,p)} \tag{2}$$

which ensures that the flows discounted on T0 curves and converted with T0 FX Spots reproduce the NPV generated from `LOGPAY` expressions.
There is a second form `LOGPAY(X, d, p, ccy, legNo, type)` taking in addition

- a leg number `legNo`, which must evaluate to a determinisitc number
- a cashflow type `type`, which is an arbitrary string meeting the conventions for variable names

This additional information is used to populate the ORE cashflow report. If not given, `legNo` is set to 0 and type is set to `Unspecified`. Notice that cashflows will equal pay dates, pay currencies, leg numbers and types are aggregated to one number in the cashflow report.
A third form `LOGPAY(X, d, p, ccy, legNo, type, slot)` takes an additional parameter `slot` which must evaluate to a whole positive and deterministic number $1, 2, 3, \ldots$.

If several cashflows are logged into the same slot, previous results are overwritten. This is useful for scripts where tentative cashflows are generated that are later on superseded by other cashflows (e.g. for an American option).
Examples for the three forms are given below:

```
Payoff1 = LOGPAY( Notional * fixedRate, PayDate, PayDate, PayCcy);
Payoff2 = LOGPAY( Notional * fixedRate, PayDate, PayDate, PayCcy, 2, Interest);
Payoff3 = LOGPAY( Notional * fixedRate, PayDate, PayDate, PayCcy, 2, Interest, 3);
```

Here, Payoff1 will appear under leg number 0 and flow type "Unspecified" in the cashflow report. Payoff2 will appear under leg number 2 and flow Type "Interest". The same holds for Payoff3, but if any amounts were booked using the slot parameter 3 previously they will be overwritten with the current amount.
Note: If IncludePastCashflows in the pricing engine config is set to true then even if p lies on or before the evaluation date, a cashflow entry will be generated.

3.26 Function `NPV, NPVMEM`

The expression `NPV(X, d, [C], [R1], [R2])` calculates a conditional NPV of an amount X conditional on a date d, i.e.

$$E(X \,|\mathcal{F}_d \cap \mathcal{F}_C) \tag{3}$$

where $\mathcal{F}_d$ is the sigma algebra representing the information generated by the model up to d and $\mathcal{F}_C$ represents the additional condition C (if given). In an MC model 3 is computed using a regression against the model state at d. C can be used to filter the training paths, e.g. on ITM paths only. d must not lie before the evaluation date, but for convenience the scipt engines will treat d as if it were equal to the evaluation date in this case for the purpose of the NPV function evaluation.
The regressor can be enriched by (at most 2) additional variables R_i. A typical usage is the accumulated coupon in a target redemption feature which heavily influences the future conditional NPV but is not captured in the model state.
A typical usage of the NPV function is to decide on early exercises in the Longstaff-Schwartz algorithm:

```
NUMBER Payoff, d;
FOR d IN (SIZE(ExerciseDates), 1) DO
    IF ExerciseDates[d] > TODAY THEN
      Payoff = PAY( PutCall * (Underlying(ExerciseDates[d]) - Strike),
                    ExerciseDates[d], ExerciseDates[d], PayCcy);
      IF Payoff > 0 AND Payoff > NPV( Option, ExerciseDates[d], Payoff > 0) THEN
        Option = Payoff;
      END;
    END;
END;
Option = LongShort * Quantity * Option;
```

Here `TODAY` represents the evaluation date to ensure that only future exercise dates are evaluated, see 3.16.
Note: It is the users responsibility to use NPV() correctly to a certain extend: An example would be that X is composed from both past and future fixings w.r.t. the observation time t. In that case only the future fixings should be included in the argument of NPV(), whereas the past fixings are known and should just be added to the result of NPV().

The variant `NPVMEM(X, d, s, [C], [R1], [R2])` works exactly like `NPV(X, d, [C], [R1], [R2])` except that it takes an additional parameter `s` that must be an integer. If `NPVMEM()` is called more than once for the same parameter `s` a regression model representing the conditional npv will only be trained once and after that the trained model will be reused. The usual use case is for scripts used in combination with the AMC module where a regression model will be trained on a relative large number of paths (specified in the pricing engine configuration) and then reused in the global exposure simulation on a relatively small number of paths (specified in the xva simulation setup).

3.27 Function `HISTFIXING`

The expression `HISTFIXING(Underlying, d)` returns 1 if d lies on or before the reference date *and* the underlying has a historical fixing as of the date d and 0 otherwise.

3.28 Function `DISCOUNT`

The expression `DISCOUNT(d, p, ccy)` calculates a discount factor $P_{ccy}(d, p)$ as of d for p in currency *ccy*. Here d must not be a past date and $d \leq p$ must hold.

3.29 Functions `FWDCOMP` and `FWDAVG`

The `FWDCOMP()` and `FWDAVG()` functions are used to calculate a daily compounded or averaged rate over a certain period based on an overnight index such as USD-SOFR, GBP-SONIA, EUR-ESTER etc..
The rate is estimated as seen from an observation date looking *forward* from that date, even if fixings relevant for the rate lie in the past w.r.t. the observation date. In the latter case, an approximation to the true rate which is then dependent on the path leading from TODAY to the current model state at the observation date is calculated. This approximation is model-dependent. The only exception to this mechanics are historical fixings that are *known* as of TODAY. Such fixings are always taken into consideration with their true value.
More specifically, the `FWDCOMP()` and `FWDAVG()` functions take the following parameters. The parameters must be given in that order, and all parameters must be given in sequence up to the parameter "end" (last mandatory parameter) or the end of an optional parameter group (i.e. an optional parameter group must be given as a whole). Furthermore, all parameters must be deterministic.

- index [mandatory]: an overnight index `index`, e.g. EUR-EONIA, USD-SOFR, ...
- obs [mandatory]: an observation date obs $\leq$ start; if obs is $<$ TODAY it is set to TODAY, i.e. the result is as of TODAY in this case
- start [mandatory]: the value start date, this might be modified by a non-zero lookback
- end [mandatory]: the value end date, this might be modified by a non-zero lookback
- spread [optional group 1]: a spread, defaults to 0 if not given
- gearing [optional group 1]: a gearing, defaults to 1 if not given

- lookback [optional group 2]: a lookback period given as number of days, defaults to 0 if not given. This argument must be given as either a constant number or a plain variable, i.e. not as a more complex expression than either of these.
- rateCutoff [optional group 2]: a rate cutoff given as number of days, defaults to 0 if not given
- fixingDays [optional group 2]: the fixing lag given as number of days, defaults to 0 if not given. This argument must be given as either a constant number or a plain variable, i.e. not as a more complex expression than either of these.
- includeSpread [optional group 2]: a flag indicating whether to include the spread in the compounding, a value equal to 1 indicates 'true', -1 false, defaults to 'false' if not given
- cap [optional group 3]: a cap value, defaults to 999999 (no cap) if not given
- floor [optional group 3]: a floor value, defaults to -999999 (no floor) if not given
- nakedOption [optional group 3]: a flag indicating whether the embedded cap / floor should be estimated, a value equal to -1 indicates 'false' (capped / floored coupon rate is estimated), 1 'true' (embedded cap / floor rate is estimated), defaults to 'false' if not given
- localCapFloor [optional group3]: a flag indicating whether the cap / floor is local, a value equal to -1 indicates 'false', 1 'true', defaults to 'false' if not given.

Based on these parameters a rate corresponding to that computed for a vanilla floating leg is estimated, see the description in section "Floating Leg Data, Spreads, Gearings, Caps and Floors" for more details on this.

3.30 Functions `ABOVEPROB, BELOWPROB`

These functions are only available in Monte Carlo engines. The expression

```
ABOVEPROB(underlying, d1, d2, U)
```

returns the pathwise probability that the value of an index `underlying` lies at or above a number U for at least one time t between dates $d1$ and $d2$ conditional on the underlying taking the simulated path values at $d1$ and $d2$. The probability is by definition computed assuming a continuous monitoring. Similarly,

```
BELOWPROB(underlying, d1, d2, D)
```

returns the probability that the value of the underlying lies at or below D. Notice that $d1$ and $d2$ should be adjacent simulation dates to ensure that the results computed in the script are meaningful. This means the script should not evaluate the underlying at a date d with $d1 < d < d2$.
We note that U and D are not required to be deterministic quantities, although the common use case will probably be to have path-independent inputs.
Finally, if $d1 > d2$ both functions return 0.

4 Models

4.1 Pricing Engine Configuration

An example pricing engine configuration looks as follows.

```
<Product type="ScriptedTrade">
 <Model>Generic</Model>
 <ModelParameters>
   <!-- shared parameters -->
   <Parameter name="Model">BlackScholes</Parameter>
   <Parameter name="InfModelType">DK</Parameter>
   <Parameter name="BaseCcy">USD</Parameter>
   <Parameter name="EnforceBaseCcy">false</Parameter>
   <Parameter name="GridCoarsening">3M(1W),1Y(1M),5Y(3M),10Y(1Y),50Y(5Y)</Parameter>
   <Parameter name="IrReversion">0.0</Parameter> <!-- fallback for other ccys -->
   <Parameter name="IrReversion_EUR">0.0</Parameter>
   <Parameter name="IrReversion_GBP">0.0</Parameter>
   <Parameter name="FullDynamicIr">true</Parameter>
   <Parameter name="FullDynamicFx">true</Parameter>
   <Parameter name="ReferenceCalibrationGrid">400,3M</Parameter>
   <Parameter name="Calibration">Deal</Parameter>
   <!-- product specific parameters -->
   <Parameter name="Model_SingleAssetOption(EQ)">BlackScholes</Parameter>
   <Parameter name="Model_SingleAssetOption(FX)">BlackScholes</Parameter>
   <Parameter name="Model_SingleAssetOption(COMM)">BlackScholes</Parameter>
   <Parameter name="Model_SingleAssetOptionBwd(EQ)">BlackScholes</Parameter>
   <Parameter name="Model_SingleAssetOptionBwd(FX)">BlackScholes</Parameter>
   <Parameter name="Model_SingleAssetOptionBwd(COMM)">BlackScholes</Parameter>
   <Parameter name="Model_SingleUnderlyingIrOption">GaussianCam</Parameter>
   <Parameter name="Model_SingleUnderlyingIrOptionBwd">GaussianCam</Parameter>
   <Parameter name="Model_MultiUnderlyingIrOption">GaussianCam</Parameter>
   <Parameter name="Model_IrHybrid(EQ)">GaussianCam</Parameter>
   <Parameter name="Model_IrHybrid(FX)">GaussianCam</Parameter>
   <Parameter name="Model_IrHybrid(COMM)">GaussianCam</Parameter>
 </ModelParameters>
 <Engine>Generic</Engine>
 <EngineParameters>
   <!-- shared parameters -->
   <Parameter name="Engine">MC</Parameter>
   <Parameter name="Samples">10000</Parameter>
   <Parameter name="StateGridPoints">200</Parameter>
   <Parameter name="StateGridPoints_SingleUnderlyingIrOptionBwd">50</Parameter>
   <Parameter name="MesherEpsilon">1E-4</Parameter>
   <Parameter name="MesherScaling">1.5</Parameter>
   <Parameter name="MesherConcentration">0.1</Parameter>
   <Parameter name="MesherMaxConcentratingPoints">9999</Parameter>
   <Parameter name="MesherIsStatic">true</Parameter>
   <Parameter name="RegressionOrder">2</Parameter>
   <Parameter name="TimeStepsPerYear">24</Parameter>
   <Parameter name="Interactive">false</Parameter>
   <Parameter name="BootstrapTolerance">0.1</Parameter>
   <Parameter name="IncludePastCashflows">true</Parameter>
   <Parameter name="RegressionVarianceCutoff">1E-5</Parameter>
   <!-- product specific parameters -->
   <Parameter name="RegressionOrder_SingleAssetOption(EQ)">6</Parameter>
   <Parameter name="RegressionOrder_SingleAssetOption(FX)">6</Parameter>
   <Parameter name="RegressionOrder_SingleAssetOption(COMM)">6</Parameter>
```

```
    <Parameter name="Engine_SingleAssetOptionBwd(EQ)">FD</Parameter>
    <Parameter name="Engine_SingleAssetOptionBwd(FX)">FD</Parameter>
    <Parameter name="Engine_SingleAssetOptionBwd(COMM)">FD</Parameter>
    <Parameter name="Engine_SingleUnderlyingIrOption">FD</Parameter>
    <Parameter name="useAD_MultiAssetOptionAD(EQ)">true</Parameter>
    <Parameter name="useAD_MultiAssetOptionAD(FX)">true</Parameter>
    <Parameter name="useAD_MultiAssetOptionAD(COMM)">true</Parameter>
    <!-- MultiUnderlyingIrOption -->
    <!-- IrHybrid(EQ) -->
    <!-- IrHybrid(FX) -->
    <!-- IrHybrid(COMM) -->
  </EngineParameters>
</Product>
```

The model parameters have the following meaning:

- Model: The model to be used. Currently BlackScholes, LocalVolDupire, LocalVolAndreasenHuge, GaussianCam are supported
- BaseCcy: The default base ccy for the model. See 4.6 for the way the base ccy is determined for a model.
- EnforceBaseCcy: Enforce the base ccy from the model settings to be used, see 4.6 for more details.
- FullDynamicFx: by default, dynamic fx processes are generated only for FX indices on which the script explicitly depends; if this parameter is true, dynamic fx processes are also generated for payment and equity / commodity ccys not equal to the base ccy of the model or any of the currencies covered by those FX indices; if the parameter is false on the other hand zero volatility paths are assumed for these additional ccys
- FullDynamicIr: by default, dynamic ir processes are generated only for currencies of the IR indices on which the script depends; if the parameter is true, dynamic ir processes are also generated for currencies from fx indices, and payment and equity / commodity ccys not equal to the base ccy of the model; if the parameter is false zero volatility paths are assumed for these ccys
- InfModelType: Optional, defaults to "DK". For the GaussianCam model two flavours of inflation models are avaiable, Dodgson-Kainth ("DK") and Jarrow-Yildrim ("JY"). This parameter selects the flavour to use. Only applies to model = GaussianCam
- ReferenceCalibrationGrid: If given, only one calibration point per defined interval will be used (to avoid oscillations in the calibrated model volatility function or also to improve the calibration speed), only applicable to model = GaussianCam
- Calibration: Deal or ATM. For Deal the calibration spec of a script is used to calibrate a model. For ATM an ATM calibration is used. Optional, defaults to Deal.

The engine parameters have the following meaning:

- Engine: The engine to be used. Currently MC and FD is supported.

- Samples: The number of MC samples used. Only relevant for Engine = MC.
- StateGridPoints: The number of grid points in state direction. Only relevant for Engine = FD.
- MesherEpsilon: The FD mesher epsilon, optional, defaults to 1E-4.
- MesherScaling: The FD mesher scaling factor, optional, defaults tp 1.5.
- MesherConcentration: The FD mesher concentration parameter, optional, defaults to 0.1
- MesherMaxConcentrationPoints: The maximum number of mesher concentration points to use, optional, defaults to 9999, in which all calibration strikes from a script are used. For 1 only the first strike from the list is used, for 2 the first two strikes, etc.
- MesherIsStatic: If true, the mesher is built only once and reused under scenario / sensitivity computations. If false, the mesher is rebuilt for each repricing. Optional, defaults to false. For sensitivity runs it should be set to true.
- RegressionOrder: The order of the polynomial basis to compute conditional expectations via regression analysis. Applies to MC only.
- SequenceType: The sequence type used for pricing. Defaults to SobolBrownianBridge. Possible values SobolBrownianBridge, Burley2020SobolBrownianBridge, MersenneTwister, MersenneTwisterAnithetic, Sobol, Burley2020Sobol. Applies to MC only.
- PolynomType: The polynom type used for regression analysis. Defaults to Monomial. Possible values Monomial, Laguerre, Hermite, Hyperbolic, Legendre, Chebyshev, Chebychev2nd. Applies to MC only.
- TrainingSamples: If given, pricing and training are separate phases and traning phase is using this number of samples. Notice that NPVMEM() should be used to reuse regression coefficients from the training phase in the pricing phase. Applies to MC only.
- TrainingSeed: The seed used for rng in training phase. Only applies if TrainingSamples are given (and thus training and pricing phases are separate). Defaults to 43. Applies to MC only.
- TrainingSequenceType: The sequence type used for training. Only applies if TrainingSamples are given (and thus training and pricing phases are separate). See SequenceType for possible values. Defaults to MersenneTwister. Applies to MC only.
- SobolOrdering: Sobol sequence ordering for brownian bridge. Defaults to Steps. Possible values Steps, Factors, Diagonal. Applies to MC only.
- SobolDirectionIntegers: Sobol direction integers. Defaults to JoeKuoD7. Possible values Unit, Jaeckel, SobolLevitan, SobolLevitanLemieux, JoeKuoD5, JoeKuoD6, JoeKuoD7, Kuo, Kuo2, Kuo3. Applies to MC only.

- Seed: The seed for rng in pricing phase. Defaults to 42. Applies to MC only.
- TimeStepsPerYear: The number of time steps used to discretise the process. For 0 only the relevant simulation times are used. Otherwise at least the given number of step are used in the discretisation grid per year.
- CalibrationMoneyness: Moneyness of options used for smile calibration. Applies to the LocalVolAndreasenHuge model only. The moneyness is defined as a "standardised moneyness" $\ln(K/F)/\sigma\sqrt{t}$ with K strike, F ATMF forward, σ ATMF market vol, t option time to expiry
- BootstrapTolerance: tolerance for calibration bootstrap, only applies to model = GaussianCam
- IncludePastCashflows: if true, LOGPAY() will generate cashflow information for pay dates on or before the reference date. Optional, defaults to false.
- RegressionVarianceCutoff: Optional. Only relevant for MC models. If given, a coordinate transform and (possibly) a factor reduction is applied to the regressors used for conditional expectation calculation, such that $1-\epsilon$ of the total variance of regressors is kept, where ϵ the given parameter. This helps dealing with collinearity and also reducing the dimnensionality of the regression model.
- Interactive: If true an interactive session is started on script execution for debugging purposes; should be false except for debugging purposes
- UseAD: If true and RunType in the global pricing engine parameters is SensitivityDelta, a first order pnl expansion using AD sensitivities is used to compute scenario NPVs.
- UseCG: If true a computation graph is used to price trades instead of the runtime interpreter . If UseAD or UseExternalComputingDevice is true, this implies that UseCG is true irrespective of how it is configured.
- UseExternalComputingDevice: If true and RunType is not NPV (generating additional results) and AD sensitivities are *not* used, an external compute device is used for the calculations.
- UseDoublePrecisionForExternalCalculation: Use double precision for external computations. Defaults to false.
- ExternalDeviceCompatibilityMode: Only applies if UseCG is enabled. Defaults to false. If enabled, random number generation for internal calculations using the CG is aligned as closely as possible with what is usually implemented for external calculations, i.e. if enabled, the MersenneTwister random number generation is done in step-dimension-path order. If disabled, the "classic" order path-step-dimension is used.
- ExternalComputeDevice: The external compute device to use if UseExternalComputingDevice is effective.

4.2 Product Tags und pricing engine configuration

All parameters in the pricing engine configuration can be differentiated by product tags, see 2.2 for how to add these to a script definition. If a script has a product tag `TAG` assigned, parameters with suffix `_TAG` are relevant if given. If for a parameter no version with the product tag is given, the corresponding parameter without a tag is used as a fallback.
Product tags may contain the asset class variable `{AssetClass}` which is replaced by `EQ`, `FX` or `COMM` depending on the underlying asset class. If more than one index type eq, fx, comm occurs in the trade, the asset class variable is set to `HYBRID`. Notice that interest rate indices do not affect the asset class variable: If e.g. an equity trade contains an interest rate funding leg, the asset class shouuld still be EQ. Trades with exotic interest rate elements on the other hand can be distinguished by the product tag itself, see below for an example. If no eq, fx, comm index occurs in the trade, the asset class variable is left blank.
Table 2 gives a typical example for grouping scripts using the product tag. The suffix "AD" is meant to mark scripts that are suitable for AAD sensitivity computations.

Product Tag	Examples	Suitable Models	Suitable Engines
`SingleAssetOption({AssetClass})`	FX TaRF	BS, LV, GCAM	MC
`SingleAssetOptionAD({AssetClass})`	FX TaRF	BS, LV, GCAM	MC
`SingleAssetOptionBwd({AssetClass})`	EQ American Option	BS, LV, GCAM	MC, FD
`MultiAssetOption({AssetClass})`	EQ Autocallable	BS, LV, GCAM	MC
`MultiAssetOptionAD({AssetClass})`	EQ Autocallable	BS, LV, GCAM	MC
`SingleUnderlyingIrOption`	IR TaRN	LGM1F	MC
`SingleUnderlyingIrOptionBwd`	Bermudan swaption	LGM1F	MC, FD, Conv
`MultiUnderlyingIrOption`	Cross currency swaption	GCAM	MC
`IrHybrid({AssetClass})`	IR-EQ basket option	GCAM	MC

Table 2: Grouping trades by product tag (Example), models are abbreviated as BS = BlackScholes, LV = LocalVol, GCAM = GaussianCrossAsset

4.3 BlackScholes model

Models black scholes processes for EQ, FX, COMM underlyings. The strike slice of the input volatility is chosen as ATMF or as the first strike given in the list of calibration strikes for an index.
See 4.1 for the impact of the model parameter FullDynamicFx on the model setup.
For MC TimeStepsPerYear are ignored if the correlation structure is trivial, because then the process can be discretised exactly. Otherwise the given time steps per year are used to build a grid on which covariance matrices are computed assuming a constant volatility between the grid points. The actual MC paths are evoloved using these covariance matrices on the original (non-refined) time grid, i.e. taking large, exact steps again.
For FD TimeStepsPerYear, StateGridPoints, MesherEpsilon, MesherScaling, MesherConcentration, MesehMaxConcentrationPoints, MesherIsStatic are used, see the description of these parameters for their detailled interpretation.

Available Engine types: MC, FD

4.4 LocalVolDupire, LocallVolAndreasenHuge models

Models local volatility processes for EQ, FX, COMM underlyings using the full smile from input volatility term structures. To construct the local volatility surface either the

classic Dupire formula or the Andreasen-Huge method is used, see [1] for the latter. The parameter FullDynamicFx has an analoguous meaning as in the case of the BlackScholes model.
Calibration strike specifications (see 4.8) are not relevant for this model.
See 4.1 for the impact of the model parameter FullDynamicFx on the model setup.
Available Engine types: MC

4.5 GaussianCam models

Models black scholes processes for EQ, FX, COMM and LGM 1F processes for IR. For INF processes Dodgson-Kainth ("DK") and Jarrow-Yidlrim ("JY") models are available. The strike slice of the input volatility is chosen as ATMF / fair forward swap rate. Furthermore,

- For IR processes a strip of atm coterminal swaptions is used for the model calibration.
- For INF processes a strip of atm CPI cap/floors is used for the model calibration. For DK the reversion is calibrated and the volatility is fixed at 0.00050. For JY the index volatility is calibrated. The reversion of the real rate process is fixed at 0 and the volatility at 0.0030. The real rate process is assumed to be uncorrelated to the nominal rate and index process within the JY model, and also to all other IR, EQ, FX, COMM processes in the model.

Calibration strike specifications (see 4.8) are not yet supported by this model.
See 4.1 for the impact of the model parameters FullDynamicFx, FullDynamicIr on the model setup.
The FD model variant is supported for a single underlying IR model only. A single calibration strike is supported that can be specified for any IR model index appearing in the script. TimeStepsPerYear, StateGridPoints, MesherEpsilon are used, see the description of these parameters for their detailled interpretation.
Available Engine types: MC, FD

4.6 Base Currency Determination

The base currency of a model is determined using the following ruleset:

- If the model parameter EnforceBaseCcy is set to true, the base ccy is read from the model parameter BaseCcy
- Otherwise base ccy candidates are collected as the
  - target (domestic) currencies of underlying fx indices
  - if no fx indices are present in the model, all pay currencies that can occur
- If the set of base ccy candidates contains exactly one element, the base ccy is chosed as this currency
- Otherwise the base ccy from the model parameters is chosen to be the model base ccy

4.7 Grid Coarsening

Date schedules that are eligible for grid coarsening are listed in the (optional) Schedule-Coarsening subnode of the script (see 2.2)

```
<Script>
  <Code>...</Code>
  <NPV>...</NPV>
  <Results>...</Results>
  <ProductTag>...</ProductTag>
  <ScheduleCoarsening>
    <EligibleSchedule>ObservationDates</EligibleSchedule>
    <EligibleSchedule>KnockOutDates</EligibleSchedule>
  </ScheduleCoarsening>
</Script>
```

If a date schedule is eligible for grid coarsening the original grid is coarsened using the model parameter GridCoarsening if this is not empty (meaning no coarsening is applied). The parameter consists of a comma separated list of pairs of periods like 3M(1W),1Y(1M),5Y(3M),10Y(1Y),50Y(5Y). The coarsening procedure then works as follows (for the example rule):

- dates before or equal to the evaluation date are always kepts
- out to 3M in each subperiod of length 1W starting at the evaluation date at most one date of the original grid is kept in the result grid
- out to 1Y in each subperiod of length 1M starting at the last subperiod end date from the previous step at most one date of the original grid is kept
- etc. ... until all subperiods out to the last period 50Y are covered, all dates in the original grid that lie beyond the last subperiod are not present in the result grid

4.8 Calibration

The calibration approach for the model is specified on the engine parameter level (see 4.1). If the model parameter `Calibration` is set to `Deal`, information on the calibration instruments is extracted from the CalibrationSpc subnode of the script node (see 2.2). If this subnode is not given, the calibration falls back to ATMF, meaning that ATMF instruments (coterminals for IR) are used for calibration.

```
<Script>
  <Code>...</Code>
  <NPV>...</NPV>
  <Results>...</Results>
  <CalibrationSpec>
     <Calibration>
       <Index>Underlying</Index>
       <Strikes>
         <Strike>Strike</Strike>
         <Strike>KnockOutLevel</Strike>
       </Strikes>
     </Calibration>
   </CalibrationSpec>
 </Script>
```

The node CalibrationSpec can have one subnode Calibration per Index occuring in the script. For each index a list of calibration strikes can be specified. The list should start with the most important calibration strike and continue with strikes of decreasing importance. In the example (which could be a typcial setup for barrier option) the most important strike is given by the Strike variable and a secondary strike is given by the KnockOutLevel.
The usage of the calibration strikes is twofold:

- For the determination of calibration strikes. This is only relevant / supported by the BlackScholes and GaussianCam-FD model, which will use the first strike from the list to read the volatility from the market term structure (if Calibration is set to Deal in the model parameters).
- To determine concentration points for an FD mesher if `Engine` is set to `FD`. The first n strikes are used as concentration points where n is the minimum of specified strikes and the engine parameter `MesherMaxConcentrationPoints`. This is supported by the BlackScholes model only.

4.9 FX tags and correlation curves

FX indices contain a "tag", e.g. we can have two different EURUSD indices FX-ECB-EUR-USD and FX-CLOSE-EUR-USD with different historical fixings. This is respected in the script engine, i.e. historical fixings are retrieved using the exact FX index name including the tag.
For projection on the other hand we only have one set of market data per currency pair (FX Spot, forwarding curves and volatilities). This also holds for implied correlations, therefore correlation curves should be set up using the tag "GENERIC" always, e.g. the correlation between the pairs EUR-USD and JPY-USD shold be set up as FX-GENERIC-JPY-USD:FX-GENERIC-EUR-USD. The ordering of the indices in the pair follow the usual rules, i.e.

- COM < EQ < FX < IBOR < CMS
- smaller period < greater period for IBOR and CMS indices
- alphabetical order of index names for COM, EQ indices resp. of CCY1 then CCY2 for FX indices

5 Error Diagnostics and Debugging

5.1 Errors during parsing

The log file contains errors that occur during parsing showing the code context and type of error, e.g.

```
DEBUG  Getting script 'EuropeanOption' from library
DEBUG  parsing script (size 313)
ALERT  an error occured during script parsing:
NOTICE parsing stopped at L1:1:0
NOTICE expected ")" in L4:68:0:
NOTICE       Option = LongShort * Quantity * PAY(( max( Payoff, 0 ), Expiry, Settlement, PayCcy);
NOTICE                                                                 ^--- here
ALERT  [STEM] scripted trade could not be built due to parser errors, see log for more details.
```

5.2 Errors during runtime

The log file also contains errors that occur during the script execution, e.g.

```
<<<<<<<<<<
            Payoff = PutCall * (Underlying(Expiry) - Strike);
            ======
>>>>>>>>>>
Error during script execution: variable 'Payoff' is not defined. at L3:14:6
```

5.3 Tracing

If tracing is enabled in the engine parameters, the AST resulting from parsing is dumped to the log file, e.g.

```
 Sequence at L2:14:289
   DeclarationNumber at L2:14:35
     Variable(Payoff) at L2:21:6
       -
     Variable(ExerciseProbability) at L2:29:19
       -
   Assignment at L3:14:49
     Variable(Payoff) at L3:14:6
       -
     OperatorMultiply at L3:23:39
       Variable(PutCall) at L3:23:7
         -
       OperatorMinus at L3:33:29
         VarEvaluation at L3:34:18
           Variable(Underlying) at L3:34:10
             -
           Variable(Expiry) at L3:45:6
             -
         Variable(Strike) at L3:55:6
           -
   Assignment at L4:14:83
     Variable(Option) at L4:14:6
       -
     OperatorMultiply at L4:23:73
       Variable(LongShort) at L4:23:9
         -
       OperatorMultiply at L4:35:61
         Variable(Quantity) at L4:35:8
           -
         FunctionPay at L4:46:50
           FunctionMax at L4:51:16
             Variable(Payoff) at L4:56:6
               -
             ...
```

This can be used to track down parsing errors at a low level. Furthermore the context before the script engine is run is logged containing the variables from the data node of the scripted trade, e.g.

```
run script engine, context before run is:
Expiry                        (Event   )    const    February 9th, 2020
LongShort                     (Number  )    const    1.000000 (10000 det)
```

```
Option                          (Number  )              0.000000 (10000 det)
PayCcy                          (Currency)    const    USD
PutCall                         (Number  )    const    1.000000 (10000 det)
Quantity                        (Number  )    const    1000.000000 (10000 det)
Settlement                      (Event   )    const    February 9th, 2020
Strike                          (Number  )    const    2147.560000 (10000 det)
TODAY                           (Event   )    const    February 5th, 2016
Underlying                      (Index   )    const    EQ-SPX
```

which can be used to verify the input data used for the concrete trade pricing. Finally each execution step is logged with the AST node type currently processed, the value of the current evaluation and the code context which produced it

```
...
variable( Expiry ) at L3:45:6
expr value = February 9th, 2020
<<<<<<<<<<
             Payoff = PutCall * (Underlying(Expiry) - Strike);
                                                ======
>>>>>>>>>>

indexEval( EQ-SPX, February 9th, 2020 ) at L3:34:18
expr value = [1807.87,2534.44,1289.6,1541.2,3216.55,2120.7,1016.12,1159.3,2309.33,3898.24...]
<<<<<<<<<<
             Payoff = PutCall * (Underlying(Expiry) - Strike);
                                 ==================
>>>>>>>>>>
...
```

If the Interactive flag is set in the engine parameters, the single execution steps are displayed on the console output and the user is prompted for a command input which can be

- (c) to display the current state of the variables (context)
- (q) to continue the script execution without being promted any further
- return to execute the next step in the script

6 Implementation Details

6.1 Static Analysis

Extracts a superset of

- indices and associated observation dates from evaluation operator applications (see 3.8) as well as forward dates, if applicable
- observation dates from the PAY function (see 3.24) or from the DISCOUNT function (see 3.28) as well as payment dates / discount end dates; this is done per currency
- dates on which a conditional expectation is required from the NPV function (see 3.26)

that can possibly occur in a concrete script execution. The results from the static analysis are used to set up the model against which the script is eventually run.

6.2 Script Parser

Translates the script into an Abstract Syntax Tree (AST).

6.3 Script Analyzer

Extracts information from the AST and a concrete set of external data that is needed to set up and calibrate a model like the index names and the required simulation times for the indices.

6.4 Script Engine

Runs a script on the AST using a certain model type against a concrete model instance and given a concrete set of external data.

6.5 Model

Interface that a model must implement so that the script engine can be run against it. An example is the BlackScholes class.

References

[1] Andreasen J., Huge B.: Volatility Interpolation (2010) https://ssrn.com/abstract=1694972

[2] Caspers, Peter: Daily Spread Curves and Ester (September 30, 2019). Available at SSRN: https://ssrn.com/abstract=3500090 or http://dx.doi.org/10.2139/ssrn.3500090