ORE Design

30 January 2026

Document History

Date	Author	Comment
14 November 2019	Roland Lichters	initial release
4 December 2021	Roland Kapl	updates for release 6
26 April 2024	Roland Lichters	updates for release 12

Contents

Chapter 1

ORE Overview

1.1 Introduction

Open Source Risk Engine (ORE, https://opensourcerisk.org) was first released by Quaternion (https://www.quaternion.com) as free/open-source software in 2016, with annual to quarterly subsequent releases to date, the latest 11th version as of October 2023.

The Open Source Risk project aims at establishing a transparent peer-reviewed framework for pricing and risk analysis that serves as

- a benchmarking, validation, training, teaching reference
- an extensible foundation for tailored risk solutions.

ORE's analytics cover

- Financial instrument valuation for a range of derivatives products and bonds across six risk classes - interest rates, foreign exchange, inflation, equity, credit and commodity
- market risk analysis, sensitivity analysis, stress testing, Value at Risk
- credit exposure simulation
- CSA pricing and XVA calculation

ORE is based on QuantLib (https://quantlib.org), the "free/open-source library for quantitative finance", which in turn depends on the Boost C++ libraries (https://boost.org). The QuantLib project had its initial release in December 2000, and it has grown over almost two decades to over 500 thousand lines of code, driven by a small number of authors and larger number of contributors, including a number of Quaternion staff.

QuantLib provides the "infrastructure" of a fundamental class hierarchy covering

- dates and date arithmetic
- calendars
- schedules and schedule generation from rules

- yield/inflation/default term structures and bootstrap methods,
- financial instruments, models and model calibration,
- pricing engines, various pricing method templates (analytical, Monte Carlo simulation, finite difference),
- a range of underlying mathematical utilities including solvers, optimisers, interpolators etc.

ORE extends QuantLib, adding to QuantLib's class hierarchy what we have experienced as missing ingredients in our history as QuantLib users over the past 20 years. Moreover, and in contrast to QuantLib, ORE provides a risk application that is designed to be accessible to end users, providing

- a simple command-line application with input/output files
- transparent interfaces for trade data, market data, system configuration
- a detailed user guide with a large range of examples of ORE usage, see https://opensourcerisk.org/documentation

The purpose of this document is describing the layout and technical design of ORE, as well as its connection with QuantLib.

Like QuantLib, ORE is written in C++, and it is organized in a hierarchy of C++ libraries an application needs to be linked with, as indicated in figure 1.1.

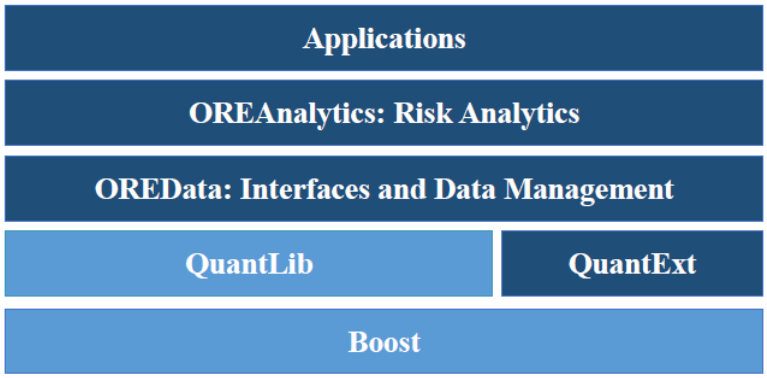

Figure 1.1: Hierarchy of ORE libraries.

The dark blue parts of indicate components of ORE, dependencies are indicated in light blue color.

QuantExt - Collection of QuantLib Extensions

Extensions to QuantLib such as financial instruments, pricing models, pricing engines and term structures are collected in the QuantExt library. The internal organization of QuantExt (folder structure, class hierarchy) mimics the structure of QuantLib, so that developers familiar with QuantLib will have little difficulty finding their way around QuantExt as well.

At an early stage of the Open Source Risk project we decided to keep these kind of extensions separate from QuantLib (rather than adding to a local copy of the

QuantLib library or immediately contributing these changes to the QuantLib project). This facilitates QuantLib release changes, swapping QuantLib in ORE for another/newer version of QuantLib.

Similarly, rather than patching a copy of QuantLib locally when identifying a bug or issue, we tend to copy the affected QuantLib class or part of the QuantLib class hierarchy to QuantExt and apply the correction in QuantExt. In doing so, we decouple from the QuantLib release cycle and bug fixes and facilitate quick fixes of parts of QuantLib that ORE critically depends on.

Some of the extensions that go into QuantExt may be contributed back into QuantLib with a delay, and then removed from QuantExt again. Examples are cross currency instruments and related engines and bootstrap helpers. Other extensions that are more relevant for the risk analytics in ORE may stay in QuantExt longer term, such as the cross asset model underlying the Monte Carlo scenario generation for exposure simulation and xVA in ORE.

QuantLib's focus is on the individual instrument and its valuation and analysis whereas ORE aims at portfolio analytics that may be more than simply additive in the instrument's contributions (such as xVA, Value at Risk, etc). QuantExt provides models and methods to support these analytics so that we might see QuantExt existing in parallel to QuantLib in the longer term.

OREData - Data Management, Translation and Assembly

OREData is a library beyond the scope of QuantLib and QuantExt. OREData serves at the high level two purposes summarized under the label of "data management"

- to **translate** between external (XML) representations of financial objects and internal (C++) representations of objects in memory that QuantLib and QuantExt can deal with
- to **assemble essential object hierarchies** underneath the abstract
  - **Portfolio** object containing pointers to all financial products "loaded"
  - **Market** objects containing bootstrapped curves and volatility surfaces of all kinds) with the help of various kinds of **configuration** and **convention** objects
  - **Model** objects used for market simulation or instrument pricing all with the help of a range of lower level **utilities** and **factories**. The functionality in OREData with QuantLib/QuantExt underneath is sufficient to build a market and bootstrap curves, load a portfolio from XML, calculate present values and project cash flows.

The key Portfolio, Market and Model objects are in turn used by the higher level analytics described in the next paragraph.

Furthermore, OREData provides supporting classes for reporting (see folder `OREData/ored/report`) and various utility classes for building the above mentioned objects, logging, screen output and serialization (see folder `OREData/ored/utility`).

OREAnalytics - Portfolio Risk Analytics

OREAnalytics is a library that comprises the classes for portfolio risk analytics, in particular Monte Carlo simulation-based analytics. It was originally designed to support exposure simulation and XVA, but also covers hypothetical scenario analysis (sensitivities, stress testing). The classes in OREAnalytics take the objects assembled in OREData (Portfolio, Market, Model) as an input. Key objects in OREAnalytics are then

- **SimMarket** and derived classes, all derived in turn from the Market base class in OREData: allows changing market points and updates of term structures after market moves
- **ScenarioGenerator** classes of several kinds (LGM, cross asset model, sensitivity, stress test): these generate Monte Carlo or hypothetical market scenarios (stored in **Scenario** objects); the scenarios in turn can be "applied" to the ScenarioSimMarket, and a Portfolio linked to the latter market then reacts to the market changes when repriced
- **NPVCube**: interface to stored portfolio NPVs or sensitivities
- **ValuationEngine**: builds an NPV cube given Portfolio, SimMarket, simulation DateGrid; builds a sensitivity cube
- **PostProcess**: Performs NPV cube aggregation, evaluates the collateral account evolution, computes exposure statistics and XVAs; key inputs into PostProcess are the Portfolio, NettingSet data, today's market, the NPV cube generated by ValuationEngine
- **OREApp**: Orchestrates the information flow from portfolio and market data loading to various analytics and reporting, supports e.g. a batch type work flow for ORE as illustrated in figure 1.2.

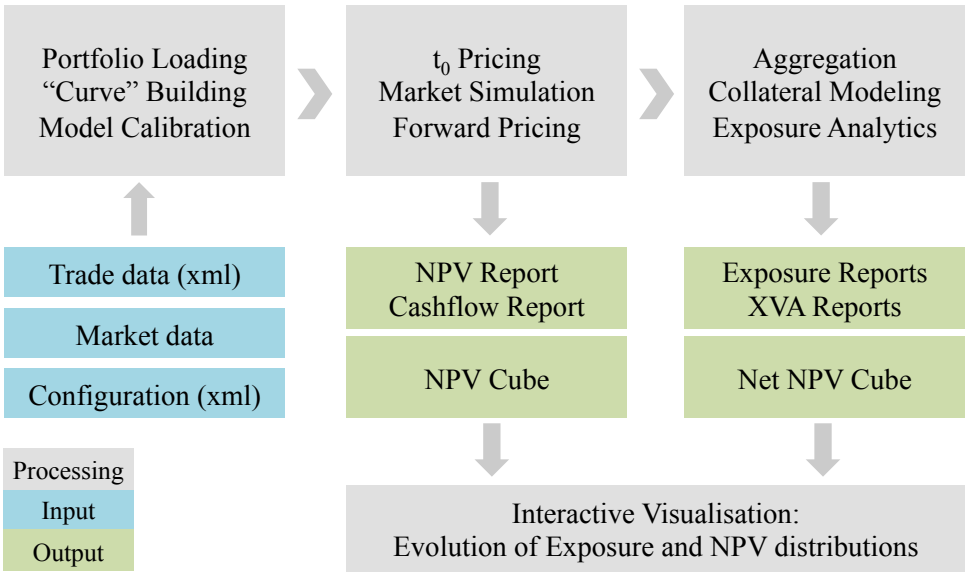

Figure 1.2: Information flow implemented in ORE App.

Applications

ORE comes with a command line application that wraps the OREApp class in OREAnalytics, with the following minimal main function.

```
int main(int argc, char** argv) {
        // ...
        if (argc != 2) {
                std::cout << endl << "usage: ORE path/to/ore.xml" << endl << endl;
                return -1;
        }
        string inputFile(argv[1]);
        boost::shared_ptr<Parameters> params = boost::make_shared<Parameters>();
        try {
                params->fromFile(inputFile);
                OREApp ore(params);
                return ore.run();
        } catch (const exception& e) {
                cout << endl << "an error occurred: " << e.what() << endl;
                return -1;
        }
}
```

The ORE User Guide at https://opensourcerisk.org/documentation discusses the usage of the ORE command line application in detail, the main ore.xml input file passed to the application and the various inputs referenced therein.

Unit Tests

All three libraries - QuantExt, OREData and OREAnalytics - are covered by unit test suites, again following the example of QuantLib. In total the test suites currently (as of release 11) comprise over 600 test cases.

Language Bindings

To open ORE up and make its components accessible from other programming languages such as Python and Java, we pursue the same approach as QuantLib and provide (a framework of) language bindings using SWIG, the Simple Wrapper Interface Generator (http://swig.org).

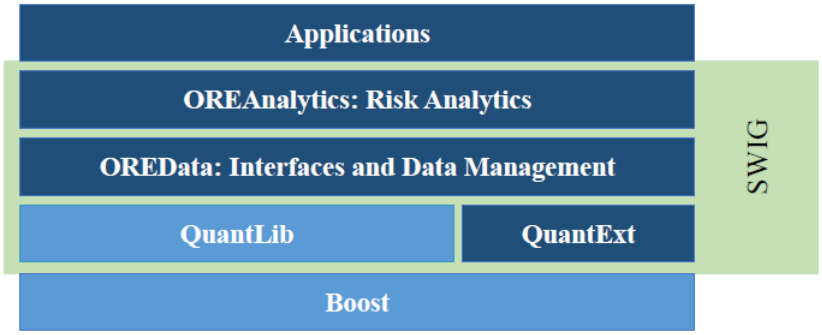

Figure 1.3: SWIG wrapper.

The ORE SWIG bindings are open source as well and provided in a separate repository https://github.com/opensourcerisk/ore-swig). This project provides language bindings for QuantLib, QuantExt, ORData and OREAnalytics as illustrated in figure 1.3. This allows, for example, running the ORE process that is encapsulated in the C++ call `ore.run()` in Python, but also querying members of the OREApp object in Python or calling into lower level ORData, QuantLib or QuantExt objects in Python. Refer to the examples in the ORE-SWIG project.

1.2 Library Stack

1.2.1 QuantExt

QuantExt contains ORE's extensions to QuantLib such as financial instruments, pricing models, pricing engines and term structures, to extend the product coverage across six asset classes ORE is aiming for. The internal organization of QuantExt (folder structure, class hierarchy) mimics the structure of QuantLib, so that developers familiar with QuantLib will have little difficulty finding their way around QuantExt as well. Moreover we follow closely the design of QuantLib since

- QuantLib's design is based on decades of quant developer experience and well tested in practice over the past 20 years
- we want to keep the option open that QuantExt code is migrated in part or entirely into QuantLib in the future

We do not elaborate on the QuantLib/QuantExt design and patterns, or the range of basic QuantLib objects here as this has been well covered by Luigi Ballabio, one of QuantLib's creators, in his reference book [3] with free drafts available in his blog posts [4]. In the following we sketch the current scope of the QuantLib extensions in QuantExt, as of the time of writing this text (Q1 2024). This concrete list is continuously changing and will be out of date at the time of the next release, i.e. check the on-line documentation at
https://www.opensourcerisk.org/docs/qle/index.html for an up-to-date picture.

- **Calendars**: Austria, Belgium, CME, Colombia, Cyprus, France, Greece, ICE, Ireland, Israel, Luxembourg, Malaysia, Mauritius, Netherlands, Peru, Philippines, Russia, Spain, Switzerland, UnitedArabEmirates; several additional calendars we added to QuantExt have been migrated into QuantLib by now; Note that ORE allows adding and amending calendars via configuration files and without code changes at the ORData level using amendedcalendar.xpp
- **Cashflows** and related pricers for IR/FX/INF/EQ/COM payoffs:
  - IR: Average overnight indexed coupon, Bond TRS cashflow, BRL CDI coupon, Capped/Floored Average BMA coupon, CMB coupon, Duration Adjusted CMS coupon, floating annuity coupon, IBOR FRA coupon, log-normal CMS spread pricer,, Overnight indexed coupon, scaled coupon, sub-period coupon
  - IR/FX: FX linked notional coupon and cashflow, Fixed Rate FX linked notional coupon, Quanto coupon pricer

  - INF: Non-standard capped/floored yoy inflation coupon, stripped capped/floored CPI coupon, stripped capped/floored YOY inflation coupon
  - EQ: Equity coupon, Equity margin coupon
  - Commodity: Commodity cashflow, commodity indexed average cashflow, Comodity indexed cashflow

- **Currencies**: Various currencies added to QuantExt in the past have been migrated into QuantLib by now, four precious metal related currencies are left in QuantExt (XAU, XAG, XPT, XPD)
  Note that currencies can be added now via configuration and without code changes at the OREData level using configurablecurrency.xpp

- **Indexes**: Inflation indices (BE HI CP, CA CPI, DE CPI, DK CPI, ES CPI, FR CPI SE CPI)Bond index, Commodity index, Commodity basis future index, Equity index, Compo Equity index, FX index, Inflation indices (CA CPI, Danish CPI, Swedish CPI), and a range of 48 IBOR indices connected with the additional currencies above;
  Note that additional overnight and IBOR indices can be created in ORE via conventions, i.e. without code changes in QuantExt or QuantLib

- **Instruments**: About 50 instruments across asset classes
  - IR: Average OIS, BRL CDI Swap, Cash-settled European Option, Deposit, Double Overnight Indexed Basis Swap, Fixed BMA Swap, Credit-linked Swap, Multi Currency Composite instrument, Multi-Leg Option, OIS Basis Swap, Sub Periods Swap, Tenor Basis Swap
  - IR/FX: FX Forward, Cross Currency Basis MTM Reset Swap, Cross Currency Basis Swap, Cross Currency Fixed/Float Swap, Cross Currency Swap, Currency Swap, Overnight Indexed Cross Currency Basis Swap
  - Equity/FX: Cliquet Option, Variance Swap, Equity Forward, Variance Swap, Vanilla Forward Option
  - Commodity: Commodity Average Price Option, Commodity Forward, Commodity Spread Option
  - Credit: Collateralized Bond Obligation, CDS Option, Credit Default Swap, Risk Participation Agreement, Synthetic CDO,
  - Bond: Bond Option, Bond Repo, Bond Total Return Swap, Convertible Bond, ASCOT, Forward Bond, Implied Bond Spread Helper

  Note that the scripted trade framework, which allows defining further instruments via payoff scripts, currently resides in OREData, see below

- **Math**: Various tools used in models and engines
  - Risk: Delta Gamma VaR, bucketed distribution, kendall rank correlation, covariance salvage
  - Par sensitivity conversion: block matrix inverse

  - Interpolation/Extrapolation: constant interpolation, quadratic interpolation, logquadratic interpolation, flat extrapolation
  - Regression: Nadaraya Watson Regression, Stabilized General Linear Least Squares Fit
  - Optimisation: Multi-threaded differential evolution
  - Classes used in the scripted framework: Basic CPU environment, compute environemnt, opencl environment, compiled formula, random variable operations, random variable LSM basis system
- **Methods**:
  - Monte Carlo: Brownian Bridge Path interpolator, multi-path generator, multi path variate generator, path generator factory, projected buffered multi-path generator and factory
  - Finite Difference: Black Scholes mesher, Black Scholes operator, defaultable equity jump diffusion Fokker Planck operator, Quanto helper
- **Models** used for pricing and market simulation for Exposure and xVA:
  - Model and model parameterization classes: Linear Gauss Markov Model (LGM), Cross Asset Model, Cross Asset Analytics, Cross Asset Model Parametrization (EQ BS Constant / Piecewise Constant, FX BS Constant/ Piecewise Constant, IR LGM1F Constant / Piecewise Linear / Constant HullWhite, INF DK, CR LGM1F), Projected Cross Asset Model, Gaussian 1d Cross Asset adaptor, Multi-Factor Hull-White model, Normal SABR and Normal SABR interpolation, Linkable Calibrated model, Defaultable Equity Jump Diffusion model, LGM Vectorised
  - Model calibration helpers: CMS Cap, CPI Cap/Floor, CDS Option, FX/EQ Option, Futures Option, representaive swaption and fxoption
  - Model Implied Termstructures: DK Zero Inflation, DK YOY Inflation, LGM Credit, LGM Yield, CIR++ Default, exact Bachelier implied volatility, implied price/yield term structure,
  - Basket credit models and methods: homogeneous pool, pool loss model, Gaussian Large Homogeneous Pool loss model, Hull White bucketing, transition matrix)
  - Carr-Madan arbitrage check
- **Pricing Engines**: About 60 pricing engines covering the additional instruments and models above
  - IR: Analytic LGM Swaption, Deposit, Swap Delta, Swap Multi Curve, LGM Convolution Solver, Numeric LGM Swaption, Deposit, LGM Convolution Solver, Monte Carlo AMC LGM Swap, MC AMC LGM Swaption, MC AMC Multi-Leg Option, Numeric LGM Multi-Leg Option
  - IR/FX: Analytic Cross Currency LGM FX Option, Barone Adesi Whaley, Cross Currency Swap, Currency Swap, FX Forward, Monte Carlo AMC

Currency Swap, MC AMC FX Forward, MC AMC FX Option, Overnight Index Cross Currency Basis Swap

  - INF: Analytic DK CPI Cap/Floor, CPI Bachelier Cap/Floor, CPI Black Cap/Floor, Analytic JY CPI Cap/Floor, Analytic JY YoY Cap/Floor
  - Equity/FX: Analytic Cross Asset LGM Equity Option, Equity Forward, FX Forward, Overnight Index Cross Currency Basis Swap, Analytic Barrier Option, Analytic Cash-Settled European Swaption, Analytic LGM FX Option, Analytic Digital American Option, Analytic Double Barrier Binary Option, Analytic Double Barrier Option, Analytic European Option, Analytic European Forward Option, Analytic Cross Asset LGM Equity Option, Barone Adesi Whaley, Equity Forward, Variance Swap General Replication, Volatility From Variance Swap
  - Commodity: Commodity Forward, Commodity APO, Commodity Schwartz Future Option, Commodity Spread Option, Commodity Swaption, Discounting Commodity Forward
  - Credit: Analytic LGM CDS Option, Black CDS Option, Black Index CDS Option, Midpoint CDS, Midpoint CDS Mutli-State, Midpoint Index CDS, Numerical Integration Index CDS Option, Analytic LGM CDS Option, Credit-Linked Swap, Index Tranche, Midpoint CDO, idpoint
  - Bond: Risky Bond, Accrual Bond Repo, Binomial Convertible, Black Bond Option, CBOMonte Carlo, Bond Repo, Bond TRS, Forward Bond, Risky Bond, Risky Bond Multi-State, Finite Difference Convertible Bond, Finite Difference Defaultable Equity Jump Diffusion, Intrinsic Ascot

- **Processes**: Cross Asset State Process, IR LGM1F State Process, IR HW State Process, CIR++ State Process, Commodity Schwartz State Process
- **Quotes**: Log Quote, Base Correlation Quote, Composite Vector Quote
- **Term Structures**: About 80 term structures and helper classes supporting the pricing across six asset classes
  - **Bootstrap Helpers**: Average OIS Rate Helper, Basis Two Swap Helper, BRL CDI Rate Helper, Cap/Floor Helper, Cross Currency Basis MTM Reset Swap Helper, Cross Currency Basis Swap Helper, Cross Currency Fix/Float Swap Helper, IMM FRA Rate Helper, Dated Stripped Optionlet and Adapter, Default Probability Helpers, Equity Forward Curve Stripper, Overnight Index Basis Swap / Cross Currency Basis Swap, Sub-Periods Swap, Tenor Basis Swap
  - **Black Volatility Term Structures**: Inverted Vol, Monotone Variance, Variance Moneyness, Surface with Delta, Surface with ATM, Sparse Surface,
  - **Dynamic Termstructures**: Black Vol Termstructure, Optionlet Vol Termstructure, Swaption Vol Matrix, YOY Optionlet Vol Termstructure
  - **Spreaded Termstructures**: Spreaded Optionlet Volatility, Spreaded Smile Section, Spreaded Swaption Volatility, Equity Vol Constant Spread, Hazard Spreaded Default Termstructure

  - **FX**: Black Vol Surface, Smile Section, Vanna Volga Smile Section
  - **Inflation**: Stripped CPI Volatility Structure, Stripped YOY Inflation Optionlet Volatility, YOY Inflation Optionlet Stripper, YOY Optionlet Volatility Surface, corrections of QuantLib's "KInterpolatedYoYOptionletVolatilitySurface"
  - **Correlation**: Correlation Termstructure, Flat Correlation Termstructure
  - **Others**: Cap/Floor Term Vol Curve, Cap/Floor Term Vol Surface, Cross Currency Price Termstructure, Discount Ratio Modified Curves, Price Term Structure and Adapter, Extension of QuantLib's Iterative Bootstrap, corrections and extensions to QuantLib's Optionlet Stripper
- **Time**: Year Counter, Futures Expiry Calculator

The current size of the QuantExt code base is about 125k source lines (excluding blank lines). ORE's size across the three libraries in ORE is 350k lines. QuantLib's size is about 476k lines.

1.2.2 ORE Data

The Data management library OREData, as stated above, hosts the classes for

- translating external trade, market and configuration data into financial objects that QuantExt and QuantLib can deal with, and
- assembling these into object hierarchies the higher level analytics can use conveniently (e.g. Portfolio, Market, Pricing Models, Simulation Models)

The OREData library currently comprises about 149 thousand lines of code and exceeds the size of QuantExt.

We start illustrating OREData's design by focusing on the Market objects.

1.2.2.1 Market Data

The market base class provides the interface to all kinds of term structure objects that might be needed in pricing across asset classes. Listing 1 shows a small excerpt, all member functions are pure virtual and are implemented in derived classes. It is then sufficient to pass a smart pointer respectively Handle to a base Market object into classes or functions that need to perform pricing. Relevant term structures, indices and quotes are queried by providing a single unique key or set of keys.

The Market interface covers

- Yield term structures by type and name, as shown in listing 1
- Swaption volatility structures by currency
- Discount curves by currency
- IBOR and Swap indices (with associated forward curves) by index name
- FX Spot quotes by currency pair
- FX volatility term structures by currency pair

- Default term structures by name
- Recovery rate quotes by name
- CDS volatility term structures by name
- Base correlation term structures by qualifier
- Zero and Year-on-Year inflation indices (with associated forward curves) by index name
- Cap/Floor volatility structures by currency
- Year on year Cap/Floor volatility and price surfaces by index name
- Zero inflation Cap/Floor volatility and price surfaces by index name
- Equity spot prices, dividend and forecasting curves by equity name
- Equity volatility term structures by equity name
- Security spreads by security ID
- Commodity price curves by commodity name
- Commodity volatility term structures by commodity name
- Generic correlation term structures by index pair
- Prepayment rate quotes by security ID

Listing 1: Market base class, pure virtual member functions with varying number and type of keys and common last argument indicating the "curve ID" to select one of several market configurations.

```
class Market {
public:
virtual ~Market() {}
virtual Date asofDate() const = 0;
//! \name Yield Curves
//@{
virtual Handle<YieldTermStructure>
discountCurve(const string& ccy,
const string& configuration = Market::defaultConfiguration) const = 0;
virtual Handle<IborIndex>
iborIndex(const string& indexName,
const string& configuration = Market::defaultConfiguration) const = 0;
virtual Handle<SwapIndex>
swapIndex(const string& indexName,
const string& configuration = Market::defaultConfiguration) const = 0;
//@}
//! \name Swaptions
//@{
virtual Handle<SwaptionVolatilityStructure>
swaptionVol(const string& ccy,
const string& configuration = Market::defaultConfiguration) const = 0;
// ...
//@}
//! \name Foreign Exchange
// ...
};
```

The `MarketImpl` class is derived from `Market`, and it essentially provides member variables (maps) to store the underlying term structures, indices and quotes. The `TodaysMarket` class is derived from `MarketImpl`, and it provides a constructor that builds a concrete Market instance. This build process involves various helper classes for each term structure across the asset classes, see folder `OREData/ored/marketdata`, as well as configuration helpers for each term structure in folder `OREData/ored/configuration`. Moreover, the build process involves

- reading market data in csv files and term structure configurations data XML files into internal data classes, and
- parsing/translating text into basic QuantLib/QuantExt objects such as Calendars, Periods, Currencies, Quotes etc., all supported by the utility code assembled in folder `OREData/ored/utilities`.

The `TodaysMarket` class is sufficient for portfolio pricing. Further derived classes that are essential for market simulation and scenario analysis are introduced in the next section on OREAnalytics.

1.2.2.2 Portfolio and CSA Data

Listing 2: Excerpt of the Portfolio class showing essential member functions.

```
class Portfolio {
public:
//! Get a Trade with the given trade id from the portfolio
boost::shared_ptr<Trade> get(const std::string& id) const;
//! Load using a default or user supplied TradeFactory
void load(const std::string& fileName,
const boost::shared_ptr<TradeFactory>& tf = boost::make_shared<TradeFactory>());
//! Load from an XML string using a default or user supplied TradeFactory
void loadFromXMLString(const std::string& xmlString,
const boost::shared_ptr<TradeFactory>& tf
= boost::make_shared<TradeFactory>());
//! Load from XML Node
void fromXML(XMLNode* node,
const boost::shared_ptr<TradeFactory>& tf
= boost::make_shared<TradeFactory>());
//! Save portfolio to an XML file
void save(const std::string& fileName) const;
//! Call build on all trades in the portfolio
void build(const boost::shared_ptr<EngineFactory>&);
//! Return trade list
const std::vector<boost::shared_ptr<Trade>>& trades() const { return trades_; }
// ...
};
```

The second large part of ORData deals with trade loading and building from external sources and assembling all trades in a `Portfolio` object. An excerpt is shown in listing 2. The `Portfolio` class provides several ways of loading a portfolio (from an XML file, an XML string, a single XML node), to store a portfolio to XML file, to access individual trades. When a portfolio is loaded from an external source, it is first represented in internal trade objects where all relevant fields are native (string, integer, double, bool) variables or standard C++ containers like vectors or maps.

In a subsequent step, initiated by a call to the `build` member function, the "raw" trade data is then translated into QuantLib/QuantExt objects up to the QuantLib/QuantExt instrument, linked to a QuantLib/QuantExt pricing engine, in turn linked to the relevant term structures provided by the `Market` class introduced in Section 1.2.2.1. The role of the `TradeFactory` class in the Portfolio member functions is to help constructing concrete instances of trade objects depending on the TradeType information in the trade XML representation. All trade objects derive from a common `Trade` base class shown in listing 3.

Listing 3: Excerpt of the Trade class showing essential member functions.

```
class Trade {
public:
class Trade : public XMLSerializable {
public:
// Constructor
// ....
//! Build QuantLib/QuantExt instrument, link pricing engine
virtual void build(const boost::shared_ptr<EngineFactory>&) = 0;
//! Return fixings that are relevant for pricing
virtual std::map<std::string, std::set<QuantLib::Date>>
fixings(const QuantLib::Date& settlementDate = QuantLib::Date()) const = 0;
//! \name Serialisation
//@{
virtual void fromXML(XMLNode* node);
virtual XMLNode* toXML(XMLDocument& doc);
//@}
//! \name Inspectors
//@{
const string& id() const { return id_; }
const string& tradeType() const { return tradeType_; }
const Envelope& envelope() const { return envelope_; }
const set<string>& portfolioIds() const { return envelope().portfolioIds(); }
const TradeActions& tradeActions() const { return tradeActions_; }
const boost::shared_ptr<InstrumentWrapper>& instrument() { return instrument_; }
const std::vector<QuantLib::Leg>& legs() { return legs_; }
const std::vector<string>& legCurrencies() { return legCurrencies_; }
const std::vector<bool>& legPayers() { return legPayers_; }
const string& npvCurrency() { return npvCurrency_; }
const Date& maturity() { return maturity_; }
//@}
//...
};
```

The `Trade` class provides access to the underlying QuantLib/QuantExt instrument and further trade information that is meaningful across all kinds of products. There is a range of concrete classes derived from `Trade` that perform the actual load and build process, see folder `OREData/ored/portfolio`. The list of ORE trade types up to release 6 in July 2021 was

- Bond
- Cap/Floor
- Commodity Forward, European Commodity Option
- Credit Default Swap
- Equity Forward, Equity Swap, European Equity Option
- Forward Bond
- Forward Rate Agreement
- FX Forward, FX Swap, European and American FX Option
- Swap

- European and Bermudan Swaption

The product coverage has then grown significantly to about 100 types over five quaterly releases, with Quaternion/Acadia contributing the bulk of formerly proprietary code to ORE. As of release 11:

- CompositeTrade
- ConvertibleBond
- Ascot
- TotalReturnSwap
- ContractForDifference
- CBO
- EquityPosition
- EquityOptionPosition
- BondPosition
- MultiLegOption
- IndexCreditDefaultSwap
- IndexCreditDefaultSwapOption
- SyntheticCDO
- BondOption
- BondTRS
- BondRepo
- CreditLinkedSwap
- CrossCurrencySwap
- InflationSwap
- Swap
- Swaption
- ForwardRateAgreement
- FxAverageForward
- FxForward
- FxOption
- FxAsianOption
- FxBarrierOption
- FxDoubleBarrierOption
- FxSwap

- FxDigitalOption
- FxEuropeanBarrierOption
- FxKIKOBarrierOption
- FxTouchOption
- FxDoubleTouchOption
- FxDigitalBarrierOption
- FxVarianceSwap
- CapFloor
- EquityOption
- EquityFutureOption
- EquityAsianOption
- EquityBarrierOption
- EquityDoubleBarrierOption
- EquityEuropeanBarrierOption
- EquityTouchOption
- EquityDoubleTouchOption
- EquityDigitalOption
- EquityForward
- EquitySwap
- EquityVarianceSwap
- EquityCliquetOption
- CommodityForward
- CommodityOption
- CommodityDigitalAveragePriceOption
- CommodityDigitalOption
- CommodityAsianOption
- CommoditySwap
- CommoditySwaption
- CommoditySpreadOption
- CommodityAveragePriceOption
- CommodityOptionStrip
- CommodityPosition

- CommodityVarianceSwap
- Bond
- ForwardBond
- CreditDefaultSwap
- CreditDefaultSwapOption
- Failed
- ScriptedTrade
- EquityBasketVarianceSwap
- FxBasketVarianceSwap
- CommodityBasketVarianceSwap
- RiskParticipationAgreement
- Autocallable_01
- DoubleDigitalOption
- EuropeanOptionBarrier
- PerformanceOption_01
- FxTaRF
- EquityTaRF
- CommodityTaRF
- FxWorstOfBasketSwap
- EquityWorstOfBasketSwap
- CommodityWorstOfBasketSwap
- EquityBestEntryOption
- FxBestEntryOption
- CommodityBestEntryOption
- FxAccumulator
- EquityAccumulator
- CommodityAccumulator
- FxWindowBarrierOption
- EquityWindowBarrierOption
- CommodityWindowBarrierOption
- FxGenericBarrierOption
- EquityGenericBarrierOption

- CommodityGenericBarrierOption
- FxBasketOption
- EquityBasketOption
- CommodityBasketOption
- FxRainbowOption
- EquityRainbowOption
- CommodityRainbowOption
- KnockOutSwap

The trade design is modular, and objects re-used within various trade types as member variables are

- Envelope
- Schedule
- OptionData
- LegData
- TradeActions

each of which comes with functions for loading from XML, building QuantLib/QuantExt objects.

CSA Data is represented in a single class, `NettingSetDefinition`, see folder `OREData/ored/portfolio`, also XML serializable like each trade. The "portfolio" of CSAs is then managed by class `NettingSetManager`, XML serializable like the `Portfolio` object.

1.2.2.3 Pricing Engines, Engine Factory

Each trade is linked with a pricing engine during the trade's build process. In order to limit the number of engines to be constructed (and to limit memory usage), ORE reuses engines as far as possible. This is achieved by the `EngineBuilder`, `EngineFactory` and `LegBuilder` classes. Currently, ORE provides 36 concrete "default" engine and leg builders when the `EngineFactory` is constructed, see orea/app/initbuilders.cpp. The design here is extensible so that developers can add their own engine builders to the factory when extending ORE. For this purpose the `EngineFactory` class provides the `addExtraBuilders` member function.

Figure 1.4 shows the relationships of these classes:

1.2.2.4 Pricing Models and Simulation Models

The construction and calibration of Pricing and Simulation models is encapsulated in the `LgmBuilder` and `CrossAssetModelBuilder` classes, see folder `OREData/ored/models`. Both are supported by associated "data" classes that hold the model/calibration configuration, XML serializable like portfolio and other configuration data in ORE.

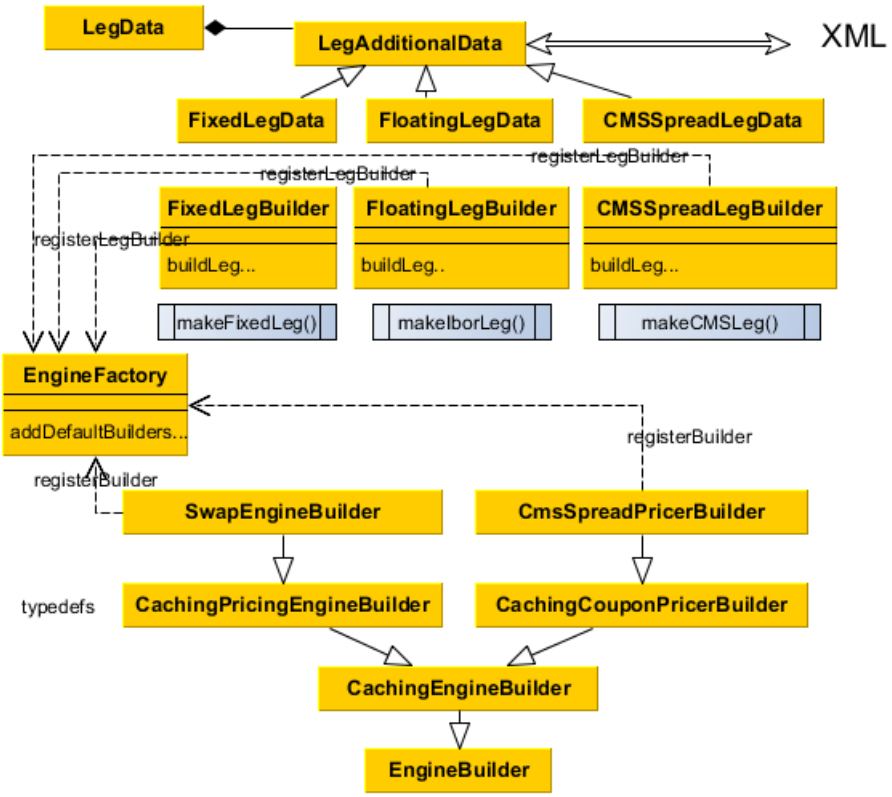

Figure 1.4: Collaboration Diagram of Instrument/Leg Builder and Factory Classes

1.2.2.5 Scripting

With the 11th release we added the scripted trade instrument which allows representing a variety of payoffs (cross asset classes, path-dependent, with multiple early exercise features) using a scripting language. The bulk of the related code can be found in OREData/ored/scripting and supports the construction and processing of the scripted trade instrument that is build in OREData/ored/portfolio/scriptedtrade.cpp. We drill deeper into the design of the scripted trade and its processing in Section 2.

1.2.3 ORE Analytics

The OREAnalytics library comprises the classes for portfolio risk analytics, in particular Monte Carlo simulation-based analytics. It was originally designed to support exposure simulation and XVA, but also covers hypothetical scenario analysis (sensitivities, stress testing). The classes in OREAnalytics depend on QuantLib, QuantExt and OREData, and the primary inputs into analytics are the Portfolio, Market and CrossAssetModel objects introduced in section 1.2.2.1.

The OREAnalytics library currently comprises about 80 thousand lines of code, somewhat smaller than QuantExt and OREData.

1.2.3.1 Simulation Market

The `TodaysMarket` class in OREData provides a snapshot of the market as of a given reference date. The Portfolio is linked to this market to produce the reference date's

valuations and cash flow projections. ORE's design to support analysis of how valuations change when the market moves under hypothetical scenarios with fixed reference date or under market changes through time is to "simply" link the portfolio to a market that change in these ways and then triggers repricing. The key element for the implementation is the `SimMarket` class, derived from the `MarketImpl` (which is derived form `Market`), i.e. it provides the same interfaces, and some additional and essential member functions, in particular the `update` function, as shown in listing 4, see folder `OREAnalytics/orea/simulation`.

Listing 4: Simulation Market base class.

```
class SimMarket : public data::MarketImpl {
public:
SimMarket(const Conventions& conventions) : MarketImpl(conventions), numeraire_(1.0) {}
//! Generate or retrieve market scenario, update market, notify termstructures and update fixings
virtual void update(const Date&) = 0;
//! Return current numeraire value
Real numeraire() { return numeraire_; }
//! Reset sim market to initial state
virtual void reset() = 0;
//! Get the fixing manager
virtual const boost::shared_ptr<FixingManager>& fixingManager() const = 0;
protected:
Real numeraire_;
};
```

The concrete class that derives from `SimMarket` and that actually implements `update`, `reset` and `fixingManager` is `ScenarioSimMarket`, as shown in listing 5, see folder `OREAnalytics/orea/scenario`.

Listing 5: Excerpt of the concrete Simulation Market class.

```
//! Simulation Market updated with discrete scenarios
class ScenarioSimMarket : public analytics::SimMarket {
public:
//! Constructors
// ...
//! Set scenario generator
boost::shared_ptr<ScenarioGenerator>& scenarioGenerator() { return scenarioGenerator_; }
//! Set aggregation data
boost::shared_ptr<AggregationScenarioData>& aggregationScenarioData() { return asd_; }
//! Set scenarioFilter
boost::shared_ptr<ScenarioFilter>& filter() { return filter_; }
//! Update market snapshot and relevant fixing history
void update(const Date& d) override;
//! Reset sim market to initial state
virtual void reset() override;
//! Return the fixing manager
const boost::shared_ptr<FixingManager>& fixingManager() const override { return fixingManager_; }
// ...
};
```

The `ScenarioGenerator` class that appears here is used to generate single or multiple market scenarios. This is used in ScenarioSimMarket's `update` function which then applies the generated scenario to the underlying data in `ScenarioSimMarket`. Moreover, the update call triggers QuantLib observer notification chains so that the portfolio that is linked to `ScenarioSimMarket` reacts to these changes with amended valuations when asked for NPVs next time. To finish this section we note that the

configuration for the simulation market's composition is externally done in ORE XML, and there is a configuration class `ScenarioSimMarketParameters` which is XML (de-)serializable, and used in the construction of the `ScenarioSimMarket` object.

1.2.3.2 Scenario Generation

OREAnalytics covers two types of Monte Carlo scenario generators that produce market evolutions - `LgmScenarioGenerator` for risk neutral interest rate scenarios only, `CrossAssetModelScenarioGenerator` for IR/FX/INF/EQ market scenarios, as well as sensitivity and stress testing scenario generators `SensitivityScenarioGenerator` and `stressScenarioGenerator`. All generators have associated XML (de-)serializable configuration classes which are used to construct the generators. The interface between any scenario generator and the `ScenarioSimMarket` is the `Scenario` object shown in listing 6. Its concrete derived classes hold the actual generated data.

Listing 6: Excerpt of the Scenario data class.

```
class Scenario {
public:
//! Return the scenario asof date
virtual const Date& asof() const = 0;
//! Get the scenario label
virtual const string& label() const = 0;
//! Set the scenario label
virtual void label(const string&) = 0;
//! Get Numeraire ratio n = N(t) / N(0) so that Price(0) = N(0) * E [Price(t) / N(t) ]
virtual Real getNumeraire() const = 0;
//! Set the Numeraire ratio n = N(t) / N(0) so that Price(0) = N(0) * E [Price(t) / N(t) ]
virtual void setNumeraire(Real n) = 0;
//! Check whether this scenario provides the data for the given key
virtual bool has(const RiskFactorKey& key) const = 0;
//! Risk factor keys for which this scenario provides data
virtual const std::vector<RiskFactorKey>& keys() const = 0;
//! Add an element to the scenario
virtual void add(const RiskFactorKey& key, Real value) = 0;
//! Get an element from the scenario
virtual Real get(const RiskFactorKey& key) const = 0;
//! clones a scenario and returns a pointer to the new object
virtual boost::shared_ptr<Scenario> clone() const = 0;
// ...
};
```

The scenario object refers to a unique reference date and contains an arbitrarily large number of data points that are identified by a RiskFactorKey. The key identifies the risk factor class (25 types so far in ORE, see the definition in `OREAnalytics/orea/scenario/scenario.hpp`, e.g. `DiscountCurve`, `IndexCurve`, `SwaptionVolatility`, etc.), a name (e.g. a currency or index name), and an integer indicating the position of the data item in a vector, matrix or cube. A market evolution or path is hence represented by a vector of scenarios. So far there is only one derived class from the base Scenario in ORE, `SimpleScenario`, which stores the scenario data in simple vectors and maps, also serializable.

1.2.3.3 Engine

The `OREAnalytics/orea/engine` folder comprises a number of calculators (wrapped into individual classes) for ORE's risk analytics beyond single pricing as of the reference date:

- **ValuationEngine** - performs a market simulation, prices a portfolio under scenarios, possibly through time, and fills a resulting NPV cube, with the help of the ValuationCalculator class; the ValuationEngine is used for Monte Carlo simulations of the market evolution but also the hypothetical scenario analytics below
- **SensitivityAnalysis**, also performed with the help of ValuationEngine and ValuationCalculator: This class wraps functionality to perform a sensitivity analysis for a given portfolio.
  - builds the "simulation" market to which sensitivity scenarios are applied,
  - builds the portfolio linked to this simulation market
  - generates sensitivity scenarios
  - runs the scenario "engine" to apply these and compute the NPV impacts of all required shifts
  - compiles first and second order sensitivities for all factors and all trades
  - fills result structures that can be queried
- **StressTest**, also performed with the help of ValuationEngine and ValuationCalculator: This class wraps functionality to perform a stress testing analysis for a given portfolio and
  - builds the "simulation" market to which stress scenarios are applied,
  - builds the portfolio linked to this simulation market
  - generates stress scenarios
  - runs the scenario "engine" to apply these and compute the NPV impacts of all required shifts
  - fills result structures that can be queried
  - writes stress test report to a file
- **ParametricVarCalculator**, as post-processor of a SensitivityStream, takes sensitivity data and a covariance matrix as an input and computes a parametric value at risk. The output can be broken down by portfolios, risk classes (IR, FX, EQ, ...) and risk types (delta-gamma, vega, ...)
- **CounterpartyCalculator/SurvivalProbabilityCalculator**, calculates the survival probability of a counterpart to be stored in the resulting cube
- **ParSensitivityAnalysis**
  The sensitivity analysis class above computes sensitivities to changes in zero rates, hazard rates, optionlet volatilities, inflation zero rates. The ParSensitivityAnalysis class turns these "raw" sensitivities into "par" sensitivities (i.e. sensitivities to changes in Deposit rates, Swap rates, CDS spreads etc.) via Jacobi transformation i.e. using the chain rule of differentiation. The Jacobi matrix for this purpose is given by the sensitivity of par instruments to the underlying "raw domain" risk factors, and it is computed in ORE by bump & revalue. This yields a possibly very large sparse Jacobi matrix. Transforming the

portfolio's raw sensitivities into par sensitivities involves multiplying the portfolio's raw sensitivity vector with the *inverse* of the Jacobi matrix. While conceptually simple, the implementation is involved and fills 2500+ lines of code in a single class. Sensitivity analysis in the raw domain is faster than direct sensitivity analysis in the par domain, because it avoids repeated curve bootstraps after changing market points that feed into curves.

- **MultithreadedValuationEngine**
  Quoting Luigi Ballabio: "*QuantLib is generally not thread-safe. The thread-local singleton pattern gives you per-thread singletons instead of global ones, but it doesn't prevent race conditions if you share objects between threads... I suggest to use multiple processes instead.*" In ORE we have chosen a middle way to parallise processing using "isolated", process-like, threads as follows: The engine builds the full portfolio once and prices it in the "main thread". Then it splits the portfolio into N parts with similar pricing time and spawns N threads. Within each thread, the engine sets the essential thread-local singleton (the evaluation date), builds a clone of the full market, builds a local scenario sim market, builds the respective sub-portfolio linked to the thread's local market, builds and runs the usual ValuationEngine; the engine then populates a "mini netting set NPV cube" for the sub-portfolio. Because the engine parallelises by splitting the portfolio and because market building causes a fixed computational overhead, we need a sufficiently large full portfolio and sufficiently large sub-portfolios to achieve a significant speed up with this approach, i.e. overall processing time not too far above the optimal limit of 1 / N times the single-threaded processing time.

- **AMCValuationEngine**
  This valuation engine uses American Monte Carlo to generate an NPV cube for a limited scope of products

  - Single Currency Swaps, built/priced with the MC LGM Swap Engine
  - Cross Currency Swaps, built/priced with the MC CAM Currency Swap Engine
  - European and Bermudan Swaptions, built/priced with the MC Multi Leg Option engine
  - FxForwards, built/priced with the MC CAM FX Forward Engine
  - FxOptions, built/priced with the MC CAM FX Option Engine

  All engines above are derived from MC Multi-Leg Base Engine and use AMC to price the instrument which involves computing regression polynomials in the training step which are then used to compute continuation values (as conditional expectations) for making exercise decisions in the pricing step. The AMCValuationEngine utilises these instrument pricing engines, i.e. uses the calibrated regression polynomials from the initial training step to generate NPV simulation paths (in the AMC Calculator that is returned as instrument additional result) and to build the NPV cube. This leads to significant speedup in the exposure simulation of complex portfolios containing Bermudan Swaptions, but even when simulating vanilla Swap portfolios.

  Note:

  - the AMCValuationEngine also supports multi-threading
  - The XVA and exposure simulation analytic (see below) can split the portfolio and cover part of it with AMCValuationEngine and the residual part using the "classic" ValuationEngine and finally consolidate the NPV cubes before post-processing for exposure and XVA reporting

1.2.3.4 Aggregation, Cubes, XVA and Post Process

For the calculation of XVA and exposures, ORE first applies the `ValuationEngine` above to build the required cubes (NPV cube on Nettingset level and trade level, Cpty Cube for survival probabilities), this is done in method `buildCube`.

The resulting cubes contain NPVs per Trade/Nettingset, future evaluation date and scenario. The cubes are then passed into the `PostProcess` class (see folder `OREAnalytics/ore/aggregation`).

The post processor is an orchestrating class performing following tasks (XML configurable) for the full portfolio and associated cube:

- compute netting set NPVs as of the reference date and find each netting set's maturity
- aggregate NPVs across trades per netting set, construct an NPV cube by netting set, future valuation date and scenario
- apply ORE's regression based dynamic initial margin model to project future Initial Margin per netting set and Monte Carlo path
- generate exposure evolutions per trade (expected positive and negative exposures, potential future exposures, etc.) without collateral
- generate collateral account balance evolutions per netting set, using CSA details in each NettingSet object (thresholds, minimum transfer amounts, independent amounts etc.)
- generate netting set exposure evolutions (after collateral)
- allocate netting set exposures to trade level
- compute XVAs (CVA, DVA, FCA, FBA, MVA, KVA) each netting set
- allocate netting set XVAs to trade level, excluding KVA

The `PostProcess` class does these tasks with the help of other classes (see folder `OREAnalytics/ore/aggregation`):

- `ValueAdjustmentCalculator`/ `StaticCreditXvaCalculator`/ `DynamicCreditXvaCalculator`: ValueAdjustmentCalculator defines an interface for derived classes to to perform the XVA calculations for all netting sets and along all paths. StaticCreditXvaCalculator calculates xva using survival probability from market, DynamicCreditXvaCalculator calculates xva using survival probability from each path
- `RegressionDynamicInitialMarginCalculator`/ `DynamicInitialMarginCalculator`: Dynamic Initial Margin calculation, fills

DIM cube per netting set that can be

  - returned to be further analyzed
  - used in collateral calculation
  - used in MVA calculation

- `ExposureCalculator`: Trade Exposure and Netting
  - Aggregation across scenarios per trade and date. This yields single trade exposure profiles, EPE and ENE
  - Aggregation of NPVs within netting sets per date and scenario. This prepares the netting set exposure calculation below
- `NettedExposureCalculator`: Netting set exposure and allocation to trades
  - Compute all netting set exposure profiles EPE and ENE using collateral if CSAs are given and active.
  - Compute the expected collateral balance for each netting set.
  - Allocate each netting set's exposure profile to the trade level such that the trade exposures add up to the netting set exposure.
- `ExposureAllocator`/ `RelativeFairValueNetExposureAllocator`/ `RelativeFairValueGrossExposureAllocator`/ `RelativeXvaExposureAllocator`/ `NoneExposureAllocator`: calculates EPE/ENE based on selected AllocationMethod
- `CollateralExposureHelper`: This class contains helper functions to aid in the calculation of collateralised exposures. It can be used to calculate margin requirements in the presence of e.g. thresholds and minimum transfer amounts, update collateral account details with e.g. new margin call info, and return collateralised exposures to the user/invoker.
- `CollateralAccount`: This class holds information corresponding to collateral cash accounts. It stores a balance as well as an asof date for the balance. The class also includes "margin" information relating to the most recent margin call (e.g. call amount, status, expected pay date. The idea is that this class can be updated on-the-run with new margin requirements and collateral balances, and the timestamps updated accordingly.
- `CVASpreadSensitivityCalculator`: Compute hazard rate and CDS spread sensitivities for a given exposure profile on an externally provided sensitivity grid.

1.2.3.5 Orchestration, ORE App

Finally, all analytics components described above are orchestrated by a single class `OREApp`, see folder `OREAnalytics/orea/app`, kicked off when OREApp's `run()` method is called.

The analytics supported by ORE

- Pricing

- Cashflows
- Sensitivities
- Stress
- XVA
- PFE (same analytics as XVA with a subset of results specific to PFE)
- VaR
- SIMM
- ParConversion
- Scenario
- ScenarioStatistic

are representated in several analytics classes (`PricingAnalytic, XvaAnalytic`, etc, see folder orea/app/analytics). These are all derived from the same `Analytic` base class and thus share several utilities including

- loading and building the initial market
- building the engine factory
- loading and building of the portfolio
- building pricing engines with links to the market term structures, linking engines to trades

All inputs into the ORE process – which analytics are run and their parameterization – are bundled in class `InputParameters` with member functions to manipulate every member variable which facilitates exposing OREApp to languages such as Python, see below. Alternatively, inputs can be provided in an instance of the `Parameter` class, essentially in the form of key/value pairs that can be loaded from XML. If the latter is used, as usual in the ORE command line application, then it gets parsed and converted into an InputParameters instance.

Results are generally kept in memory first (in-memory reports, NPV cubes, market cubes), when OREApp.run() is completed. They can be queried/extracted using a simple API (member functions of OREApp), written to files, or passed on in memory for further processing, e.g. when ORE is used in Python.

1.3 Unit Tests

All three ORE libraries (QuantExt, OREData and OREAnalytics) are covered by respective unit test suites with source code in folders

- `QuantExt/test`
- `OREData/test`
- `OREAnalytics/test`

with currently 272 test cases in QuantExt, 257 test cases in ORData and 78 test cases in OREAnalytics. Test suites are continuously extended when new functionality is added or when a bug is identified and fixed.

Chapter 2

Scripted Trade

The scripted trade module was introduced with the 11th ORE release in 2023. It provides a separate trade type in ORE which allows flexible definition of the instrument payoffs using a payoff script so that a new payoff does not require a code change and recompiling the code base, comprehensive tests, deployment etc. Scripted payoffs

- can be hybrid, i.e. depend on several IR indexes, FX rates, Inflation indexes, Equity or Commodity prices (as in Basket Options)
- can be path-dependent (as e.g. in Accumulators, TaRFs)
- can contain early exercise features (as e.g. in European/American/Bermudan Options)
- combinations of the latter two (as e.g. in Autocallables)

See the *Scripted Trade User Guide* in ORE [1] for a discussion of the language features and examples.

The Scripted Trade's XML is generic,

- contains the payoff script embedded into the trade XML, or references a script defined in a separate script library XML
- defines all variables that are used in the payoff script, supporting five types (Number, Event, Currency, Index, Daycounter)

see the scripted trade user guide that is part of each ORE release.

Notable features of the Scripted Trade that are relevant for performance

- integration into AMC exposure simulation
- fast NPV and XVA sensitivity calculation using AAD
- interface to external compute devices like GPUs for parallelisation

In the subsequent sections we follow the processing of a Scripted Trade through ORE from trade loading and parsing to pricing and exposure simulation. As we go, we briefly discuss the implementation in ORE, sketch the role of essential classes and point to the location of related files in the code base.

To fix the context, let us consider the template Vanilla Option example below

```
<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>
```

with the external script

```
<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>
    </Script>
  </Script>
</ScriptLibrary>
```

and related pricing engine configuration

```
<PricingEngines>
<PricingEngines>
  <Product type="ScriptedTrade">
    <Model>Generic</Model>
    <ModelParameters>
      <Parameter name="Model">BlackScholes</Parameter>
    </ModelParameters>
    <Engine>Generic</Engine>
    <EngineParameters>
      <Parameter name="Engine">FD</Parameter>
      <Parameter name="StateGridPoints">200</Parameter>
      <Parameter name="TimeStepsPerYear">24</Parameter>
    </EngineParameters>
  </Product>
```

2.1 Scripted Trade Build Process

When creating ORE's InputParameter instance we already load the portfolio (a single scripted trade) from XML and convert the XML representation into a `ScriptedTrade` class instance (see ored/portfolio/scriptedtrade.*pp).

Assuming we are after valuation only, we use the `PricingAnalytic` in orea/app/analytics/pricinganalytic.cpp and run its function `runAnalytic(...)`, see below. We see that we then first build the market, then build the portfolio, as is the case in almost all analytics.

```
void PricingAnalyticImpl::runAnalytic(
    const boost::shared_ptr<ore::data::InMemoryLoader>& loader,
    const std::set<std::string>& runTypes) {

    Settings::instance().evaluationDate() = inputs_->asof();
    ObservationMode::instance().setMode(inputs_->observationModel());

    QL_REQUIRE(inputs_->portfolio(), "PricingAnalytic::run: No portfolio loaded.");

    CONSOLEW("Pricing: Build Market");
    analytic()->buildMarket(loader);
    CONSOLE("OK");

    CONSOLEW("Pricing: Build Portfolio");
    analytic()->buildPortfolio();

    ...
```

`analytic()->buildPortfolio()` calls `portfolio()->build(...)` in ored/portfolio/portfolio.cpp, which causes a call to `ScriptedTrade::build()` in ored/portfolio/scriptedtrade.cpp. This in turn constructs a `ScriptedInstrument` object, see ored/scripting/scriptedinstrument.*pp, and attaches a `ScriptedInstrumentPricingEngine`, see ored/scripting/engines/scriptedinstrumentpricingengine.*pp. The scripted instrument is a slim class, while the scripted instrument pricing engine is worth looking into.

Let's see what happens when the engine builder is called on this harmless line 46 of `ScriptedTrade::build()`

```
    ...
    auto engine = builder->engine(id(), *this, engineFactory->referenceData(), engineFactory->iborFallbackConfig());
    ...
```

The second argument here (*this) means that we pass all Scripted Trade data (i.e. the payoff script and all parameters) to the engine builder in ored/portfolio/builders/scriptedrade.*pp.

2.2 Scripted Trade Engine Builder

As usual we start with reading the pricing engine parameters, i.e. reading pricingengine.xml content for the ScriptedTrade product, to populate model and engine parameters.

Then we retrieve the script code which may be located in a separate script library storage, not necessarily embedded into the trade itself.

2.2.1 Script Parser, Abstract Syntax Tree

The next essential step in the engine builder is parsing the script into an *Abstract Syntax Tree (AST)* unless we have cached the AST for this particluar script before:

```
...
ScriptedTradeScriptData script =
    getScript(scriptedTrade, ScriptLibraryStorage::instance().get(), purpose, true).second;

auto f = astCache_.find(script.code());

if (f != astCache_.end()) {
    ast_ = f->second;
    DLOG("retrieved ast from cache");
} else {
    ast_ = parseScript(script.code());
    astCache_[script.code()] = ast_;
    DLOGGERSTREAM("built ast:\n" << to_string(ast_));
}
...
```

The AST (see ored/scripting/ast.*pp) is an object representation of the script text – the parser unravels the script into a tree-like object of connected AST Nodes, representing variable definitions, basic operations and function calls, starting with the inputs and leading to the NPV as final result. When we process the "script" later on, we actually traverse its AST representation so that we do not need to repeatedly interpret or parse the script text.

The transformation of scripts into ASTs is covered by estensive unit tests which include generation of random ASTs, writing them as a script, parsing the script into an AST and comparing the two AST versions, see OREData/test/scriptparser.cpp.

2.2.2 Aside: Script Parser and Boost Spirit

The script parser in ored/scripting/scriptparser.*pp makes use of the Boost.Spirit library to generate the AST, see the call to `qi::phrase_parse` below.

```
...
ScriptParser::ScriptParser(const std::string& script) {
    ScriptGrammarIterator first(script.begin()), iter = first, last(script.end());
    ScriptGrammar grammar(first);
    success_ = qi::phrase_parse(iter, last, grammar, boost::spirit::qi::space);
...
```

The `ScriptGrammar` is the key input here beyond the script itself. The grammar is defined in ored/scripting/grammar.*pp, it encapsulates the definition of the scripting language with all keywords and language features as we have described in the *Scripted Trade User Guide*. And it is in the `grammar.hpp` header where we include the required boost spirit headers which allow that compact language definition and AST conversion:

```
#include <boost/phoenix.hpp>
#include <boost/spirit/include/qi.hpp>
#include <boost/spirit/include/support_line_pos_iterator.hpp>
```

The grammar file is, admittedly, cryptic at first glance. We refer the interested reader to the Boost Spirit documentation on boost.org and tutorials available online.

Note that this infrastructure has been used in a more confined way in ORE's formula based coupon, even before the Scripted Trade framework was developed. The formula based coupon is part of ORE's 12th release.

2.2.3 Context

Next in the engine building process is creating an instance of a "handy" struct, called `Context`, that encapsulates all constants and variables (scalars and arrays) that are required for the script or AST processing, mapping variable names to values. The context covers

- input variables
- local variables declared in the script code
- result variables, NPVs and additional results

We use a utility `makeContext` to create the struct, defined in ored/scripting/utilities.*pp.

```
...
auto context = makeContext(modelSize_, gridCoarsening_, script.schedulesEligibleForCoarsening(), referenceData,
                           events, numbers, indices, currencies, daycounters);
addAmcGridToContext(context);
addNewSchedulesToContext(context, script.newSchedules());

DLOG("Built initial context:");
...
```

and then we also ensure that all relevant schedule dates are captured in the context.

2.2.4 Static Analyser

Next, we run the `StaticAnalyser` (see ored/scripting/staticanalyser.*pp) which takes the root of the AST and the context as an input. It scans the AST and extracts several essential Date sets (by index name resp. pay currency), including

- index evaluation and forward dates, required for the MC simuilation grid and FD time grid, essential to set up the scripting model (see below)
- observation and payment dates used in the script's `PAY` function
- discount observation and pay dates used in the script's `DISCOUNT` function
- fixing and observation dates that occur in the script's `FWDCOMP` and `FWDAVG` functions
- regression dates where the script's `NPV` function needs to compute a conditional expectation

Moreover, using the static analyser results, we

- extract EQ, FX, IR, COM, INF index information from the script
- populate a map of required fixing dates by index
- extract payment currencies

Listing 7: Scripting model base class `Model`.

```
class Model : public LazyObject {
public:
    enum class Type { MC, FD };

    explicit Model(const Size n) : n_(n) {}

    // model type
    virtual Type type() const = 0;

    // result must be as of max(refdate, obsdate); refdate < paydate and obsdate <= paydate required
    virtual RandomVariable pay(const RandomVariable& amount, const Date& obsdate, const Date& paydate,
                               const std::string& currency) const = 0;

    // refdate <= obsdate <= paydate required
    virtual RandomVariable discount(...) const = 0;

    // refdate <= obsdate required
    virtual RandomVariable npv(...) const = 0;

    ...
};
```

- determine the base currency

Models, Processes, RandomVariables, Filters

Therafter we are ready to compile the pricing model and its related stochastic process. This can be as simple as building a BlackScholes model and process for a single asset Equity product, or more involved as in building a multi-factor cross asset model for a basket product.

The "models" used in the scripted trade framework, see ored/scripting/models,

- BlackScholes (file ored/scripting/models/blackscholes.*pp)
- BlackScholes with Local Volatility (file ored/scripting/models/localvol.*pp)
- GaussianCam (file (file ored/scripting/models/gaussiancam.*pp)

are actually model adaptors. Their role is to provide implementations of the model-dependent script functions (PAY, DISCOUNT, NPV, FWDCOMP, FWDAVG, ABOVEPROB, BELOWPROB). The model adaptors utilize QuantLib/QuantExt models/processes and OREData model builders "under the hood" to do the work.

All scripting models derive from the common base class `Model` in ored/scripting/models/model.hpp which declares a list of pure virtual functions (related to the script functions above) that need to be implemented by the derived classes. See the cut-down version of the `Model` base class below.

Note that the concrete scripting model adaptors also specify the numerical technique used, Monte Carlo or Finite Difference.

Focussing on the BlackScholes type model adaptors, we see that the concrete `BlackScholes` model inherits along the chain from `Model` ← `ModelImpl` ← `BlackScholesBase` ← `BlackScholes`, because it shares functionality with `LocalVol` that is implemented in `ModelImpl` and `BlackScholesBase`. Both `BlackScholes` and `LocalVol` use Monte Carlo simulation, whereas `FdBlackScholesBase` uses a Finite

Difference scheme.

The `GaussianCam` scripting model also derives from `ModelImpl`, and it uses Monte Carlo simulation because typically multi-factor.

If you look into the ored/scripting/models folder, you will see an additional adaptor hierarchy with class name endings "CG" that stands for *Computation Graph*. Which model hierarchy is built can be chosen in pricingengine.xml's ScriptedTrade section. We will get back to that in section **??**.

RandomVariable

The listing of the `Model` base class above also shows that the script processes objects of type `RandomVariable` rather than scalars. The random variable (and related objects) are defined in QuantExt (see folder qle/math). In a nutshell, `RandomVariable`s represent stochastic quantities (model states, IR/INF/FX rates or prices) as vectors of potential values, evolving in time along various Monte Carlo paths or following some finite difference scheme. The script engine processes entire `RandomVariable`s rather than one realisation at a time along a single path. The inspiration for using the RandomVariable object as "calculation primitive" is due to Fries [15].

RandomVariables currently allow a set of 18 operations (see struct `RandomVariableOpCode`)

- Add, Subtract, Negative, Mult, Div
- ConditionalExpectation
- IndicatorEq, IndicatorGt, IndicatorGeq
- Min, Max, Abs, Exp, Sqrt, Log, Pow
- NormalCdf, NormalPdf

Note:

- the ConditionalExpectation operation is *not pathwise*
- the Indicator, Min, Max, Abs and Sqrt operations are *not differentiable*

The RandomVariable is also the appropriate object to implement conditional expectations and differentiation of (smoothed) indicator functions which occurs in (not even too) complex derivative payoffs, see also Fries [16].

Filter

To cover the IF ... THEN ... ELSE language feature while processing the entire RandomVariable vector, we have introduced the concept of a `Filter`, also defined in qle/math/randomvariable.*pp. Filters are vectors of boolean's, implementing indicator functions, used to filter "branches" of the evolution of a RandomVariable when crossing an IF ... THEN ... ELSE point. "Alternative" results are finally combined, i.e. multiplied by their filter and added up. This is demonstrated in one of the worked examples, based on unit test case `testNestedIfThenElse` in OREData/test/scriptengine.cpp.

2.2.5 Scripted Instrument Pricing Engine

Getting back to the engine builder process in ored/portfolio/builder/scriptedtrade.cpp: Finally, we assemble all information and objects we have gathered in the engine builder process and create an instance of a `ScriptedInstrumentPricingEngine` which is returned to ScriptedTrade::build() to be linked to the `ScriptedInstrument`.

This engine delegates the work in turn to the `ScriptEngine` class, as we see in the following snippet of ore/scripting/engines/scriptedinstrumentpricingengine.cpp.

```
...
// set up script engine and run it
ScriptEngine engine(ast_, workingContext, model_);
auto paylog = boost::make_shared<PayLog>();
engine.run(script_, interactive_, paylog);
...
```

2.2.6 Script Engine and AST Runner

The ScriptEngine (in ored/scripting/scriptengine.*pp) then processes the AST given the context. The `ScriptEngine::run()` function itself is short, less than 70 lines, it delegates the work to the `ASTRunner` class, also defined in ored/scripting/scriptengine.cpp which forms the bulk of the 1200+ lines of code. The ASTRunner traverses the AST recursively (each node contains pointers to its argument nodes which may need processing of further argument nodes, etc.). Thus the ASTRunner "climbs" the AST tree up and down, storing interim results on the stack, until it arrives at the final result. The ASTRunner implmementation makes heavy use of the *visitor* pattern that is suited for this kind of recursive evaluation of the AST, yielding a particularly simple implementation.

Script parsing and script processing in the ScriptEngine is covered by unit tests in ORETest/test/scriptengine.cpp

2.3 Automatic Differentiation

Automatic differentiation (AD), also called algorithmic differentiation or computational differentiation, is a set of techniques for automatically augmenting computer programs with statements / code for the computation of derivatives (sensitivities) [5]. AD is hence an alternative approach to

- numerical differentiation using finite difference expressions

$$\frac{df(x)}{dx} \approx \lim_{h \to 0} \frac{f(x+h) - f(x)}{h},$$

- symbolic differentiation as implemented in computer algebra packages such as Mathematica, Maple, Maxima, etc.,
- symbolic differentiation "by hand", subsequently turned into computer code.

AD is well known in computer science since the 1980's with application e.g. in computational fluid dynamics, meteorology and engineering design optimization. It gained attention in the quantitative finance community more recently

[7, 8, 9, 10, 11, 12, 13], to name a few, where the seminal paper by Giles and Glasserman [7] addresses the problem of fast Greeks computation in a Libor Market Model setting. In January 2015 AD has even made it to the front page of RISK [14]. For a problem like CVA risk, where the underlying CVA numbers are computed by Monte Carlo techniques, "fast Greeks" are not only nice to have but a hard requirement, which explains the raised attention and the attempts to implement AD into Bank's quant libraries as stated e.g. in [12] and [14].

It can be shown [6] that the overall complexity of sensitivity calculation using AD is no more than four times greater than the complexity (number of operations) of the original valuation, an astonishing reduction in complexity compared to the conventional bump/revalue approach to sensitivity calculation, even if the realisitc overhead is closer to a factor of 10 than 4.

To apply AD, essentially the chain rule of differentiation, automatically "under the hood", it is key to extract the *Computation Graph* from an algorithm.

Consider the simple example of computing

$$y = (a + b) \times (b - c).$$

The computation graph for this case is shown in figure 2.1.

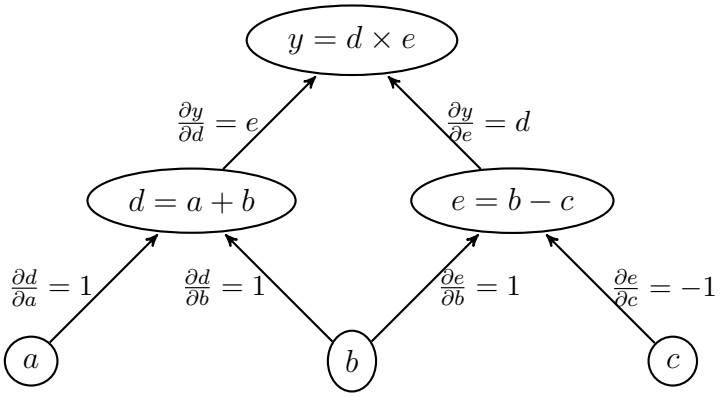

Figure 2.1: Computation Graph for $y = (a + b) \times (b - c)$

Applying the chain rule and collecting the derivatives along the edges of the graph, we can assemble the (forward) derivatives of y w.r.t. the input variables a, b, c

$$\begin{aligned}
\frac{\partial y}{\partial a} &= \frac{\partial y}{\partial d} \cdot \frac{\partial d}{\partial a} = e \\
\frac{\partial y}{\partial b} &= \frac{\partial y}{\partial d} \cdot \frac{\partial d}{\partial b} + \frac{\partial y}{\partial e} \cdot \frac{\partial e}{\partial b} = e + d \\
\frac{\partial y}{\partial c} &= \frac{\partial y}{\partial e} \cdot \frac{\partial e}{\partial c} = -d
\end{aligned}$$

The following section provides a brief introduction to the AD mechanics, *Forward* and *Backward* modes. The latter is also referred to as Adjoint Algorithmic Differentiation (AAD) and particularly important for fast sensitivity calculation.

2.3.1 Automatic Differentiation Basics

Consider the evaluation of some function $y = f(x_1, ..., x_n)$ of n input variables (e.g. an instrument value as a function of many market data points). Any such function evaluation can be expressed in terms of m intermediate variables to which we apply simple unary built-in functions (such as $\exp(x)$, $\log(x)$, $\sin(x)$, ...) and binary operations (addition, subtraction, multiplication, division).

AD is a "mechanical" application of the chain rule of differentiation to the series of intermediate results. The sequence can be evaluated in two ways, forward and backward direction, as well as combinations of both. The following sections sketch these procedures.

Forward Mode

The concept of this mode is the one that is easier to understand. We assume that our functions do not operate on real numbers but on *dual numbers* $x + \epsilon x'$ with an arithemtic defined by $\epsilon^2 = 0$. Basic operations – addition, subtraction, multiplication, division of dual numbers – follow from this definition as

$$\begin{aligned}
(x + \epsilon x') \pm (y + \epsilon y') &= x \pm y + \epsilon(x' \pm y') \\
(x + \epsilon x') \times (y + \epsilon y') &= x\, y + \epsilon(xy' + x'y) \\
(x + \epsilon x')/(y + \epsilon y') &= \frac{(x + \epsilon x') \times (y - \epsilon y')}{(y + \epsilon y') \times (y - \epsilon y')} = \frac{xy + \epsilon(x'y - y'x)}{y^2} \\
&= x/y + \epsilon(x'/y - y'x/y^2)
\end{aligned}$$

Function evaluations of dual arguments follow from their truncated Taylor expansion[1] (as all powers $\epsilon^2, \epsilon^3, \ldots$ vanish):

$$f(x + \epsilon x') = f(x) + \epsilon\, f'(x)\, x'$$

which also yields the chain rule

$$\begin{aligned}
f(g(x + \epsilon x')) &= f(g(x) + \epsilon\, g'(x)\, x') \\
&= f(g(x)) + \epsilon\, f'(g(x))\, g'(x)\, x'
\end{aligned}$$

by iterated application of the first truncated Taylor series. For $x' = 1$ the RHS of the latter is just the derivative of $f(\cdot)$ w.r.t. x. All we need to do is generalize all operations to *dual numbers* and evaluate $f(g(x + \epsilon))$. This yields the usual "primal" valuation $f(g(x))$ and the function's derivative mechanically by looking up the ϵ coefficient (second component) of the resulting dual number.

Generalizing to functions of n variables $x_1, ..., x_n$, and their dual generalization $(x_i + \epsilon x_i')$, the difference is in the truncated Taylor expansion for the function evaluations

$$f(x_1 + \epsilon x_1', \ldots, x_n + \epsilon x_n') = f(x_1, \ldots, x_n) + \epsilon \sum_{i=1}^{n} \frac{\partial f}{\partial x_i} x_i'.$$

This shows that we get any first order partial derivative by setting its *seed* x_i' to one and all others to zero. Hence this forward mode requires n passes / valuations to get

[1] Assuming that the derivative $f'(x)$ exists.

all n partial derivatives. The forward mode is efficient if we need to evaluate many functions f of few variables.

In finance it is often the other way around, few functions need sensitivity calculation w.r.t. many input variables. The *backward* mode, sketched in the following, is suited for this situation.

Backward Mode, Adjoint Algorithmic Differentiation (AAD)

The backward mode evaluates the chain rule along intermediate calculation steps in *reverse* order. Consider a sequence of evaluations [13]

$$X \to ... \to U \to V \to ... \to Y$$

which transforms the input X (of dimension n) to the outptut Y (of dimension m) using basic arithmetic operations and simple built-in function calls for which the derivative is built-in as well. We are interested in all partial derivatives

$$\frac{\partial Y_i}{\partial X_j}.$$

The derivative of the function $Y : \boldsymbol{R}^n \to \boldsymbol{R}^m, X \mapsto Y(X)$ in $\xi \in \boldsymbol{R}^n$ is a linear operator $DY(\xi) : \boldsymbol{R}^n \to \boldsymbol{R}^m$ which maps

$$\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \mapsto \begin{pmatrix} \frac{\partial Y_1}{\partial X_1}(\xi) & \cdots & \frac{\partial Y_1}{\partial X_n}(\xi) \\ \vdots & \ddots & \vdots \\ \frac{\partial Y_m}{\partial X_1}(\xi) & \cdots & \frac{\partial Y_m}{\partial X_n}(\xi) \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^n \frac{\partial Y_1}{\partial X_i}(\xi)\, x_i \\ \vdots \\ \sum_{i=1}^n \frac{\partial Y_m}{\partial X_i}(\xi)\, x_i \end{pmatrix}.$$

Because all variables are real, the adjoint operator of $DY(\xi)$ is

$$(DY(\xi)^* = \begin{pmatrix} \frac{\partial Y_1}{\partial X_1}(\xi) & \cdots & \frac{\partial Y_1}{\partial X_n}(\xi) \\ \vdots & \ddots & \vdots \\ \frac{\partial Y_m}{\partial X_1}(\xi) & \cdots & \frac{\partial Y_m}{\partial X_n}(\xi) \end{pmatrix}^\top = \begin{pmatrix} \frac{\partial Y_1}{\partial X_1}(\xi) & \cdots & \frac{\partial Y_m}{\partial X_1}(\xi) \\ \vdots & \ddots & \vdots \\ \frac{\partial Y_1}{\partial X_n}(\xi) & \cdots & \frac{\partial Y_m}{\partial X_n}(\xi) \end{pmatrix}.$$

For intermediate functions, the chain rule in conjunction with the multiplication rule for transposed matrices yields:

$$\begin{aligned} Y(X) &= Y(V(X)) \Rightarrow \\ (DY(\xi))^* &= (DY(V(\xi))\, DV(\xi))^* \\ &= (DV(\xi))^*\, (DY(V(\xi)))^* \end{aligned}$$

which allows us to compute the derivatives backwards, provided we can compute the adjoint operators. Hence, the backward mode considers *adjoint* variables

$$\bar{V}_k = \sum_{j=1}^m \bar{Y}_j \frac{\partial Y_j}{\partial V_k}$$

which collect the sensitivity of the result w.r.t. intermediate variable V_k. Likewise

$$\bar{U}_i = \sum_{j=1}^m \bar{Y}_j \frac{\partial Y_j}{\partial U_i} = \sum_{j=1}^m \bar{Y}_j \sum_k \frac{\partial Y_j}{\partial V_k} \frac{\partial V_k}{\partial U_i} = \sum_k \bar{V}_k \frac{\partial V_k}{\partial U_i}$$

Starting from the adjoint of the output we can eventually generate the adjoint of the input variables

$$\bar{X}_i = \sum_{j=1}^{m} \bar{Y}_j \frac{\partial Y_j}{\partial X_i} \tag{2.1}$$

by applying the chain rule to all intermediate results in reverse order

$$\bar{X} \leftarrow ... \leftarrow \bar{U} \leftarrow \bar{V} \leftarrow ... \leftarrow \bar{Y}.$$

Initializing the output adjoint for index j with one (and all other with zero) in (2.1) therefore yields the entire set of sensitivities of Y_i to all n input variables. Due to the formulation in terms of adjoint variables, AD in backward mode is also referred to as *Adjoint Algorithmic Differentiation (AAD)*. Recall that the overall complexity of sensitivity calculation using AAD is no more than four times greater than the complexity (in terms of number of operations) of the original valuation. The general rule is: The adjoint method should be used in case there are many inputs (curve points, volatilities, credit spreads etc.) but few outputs (prices, value adjustments). If there are few inputs but many outputs, the forward method is better.

Note that the reverse procedure uses partial derivatives of internal variables V_k w.r.t. internal variables U_i which are generally dependent on the values of internal variables. This means they need to be computed in a first forward *sweep* and then stored for usage in the backward sweep. This memory impact is different from the pure forward sweep which does not require storing intermediate results.

2.3.2 Computation Graphs in ORE

Operator Overloading is a typical approach to "recording" a computation graph automatically. See for example the attempts to integrate AAD with Operator Overloading into QuantLib [2] around 2014. See also the series of papers by Antoine Savine on the subject [18, 19, 20] and his books for a deeper dive [21, 22].

However, in ORE we apply AAD in the context of the scripted trade module only. Rather than following the operator overloading path introducing an "active" `Real` data type, replacing the "inactive" `Real` type across the entire ORE libraries, we extract the computation graph explicitly from the script in its AST representation in order to utilize AAD in the scripted trade processing only. Thus we avoid negative performance impact (recall the "factor 4" overhead) elsewhere and in the valuation of vanilla instruments. Moreover, we make the "AAD activation" of the scripted trade framework configurable, i.e. we can switch AAD on and off without recompiling the code base. How this works is discussed briefly in the following.

The `ComputationGraph` object is defined in qle/math/ad/computationgraph.*pp, alongside essential operations on the graph -- forward valuation, forward and backward derivative propagation. It is recommended to review the compact implementation of the these functions in qle/math/ad, see listings 8, 9 and 10.

To utilise the computation graph, ORE provides an additional hierarchy of scripting models and engines, as well as a configuration graph builder using the AST, summarized in table 2.1. Which side of the hierarchy is built, depends on configuration

Listing 8: Forward valuation on a computation graph in qle/ad/forwardvaluation.hpp

```
template <class T>
void forwardEvaluation(const ComputationGraph& g, std::vector<T>& values,
                       const std::vector<std::function<T(const std::vector<const T*>&)>>& ops,
                       std::function<void(T&)> deleter = {}, bool keepValuesForDerivatives = true,
                       const std::vector<std::function<std::pair<std::vector<bool>, bool>(const std::size_t)>>&
                           opRequiresNodesForDerivatives = {},
                       const std::vector<bool>& keepNodes = {}, const std::size_t startNode = 0,
                       const std::size_t endNode = ComputationGraph::nan, const bool redBlockReconstruction = false) {

    std::vector<bool> keepNodesDerivatives;
    if (deleter && keepValuesForDerivatives)
        keepNodesDerivatives = std::vector<bool>(g.size(), false);

    // loop over the nodes in the graph in ascending order

    for (std::size_t node = startNode; node < (endNode == ComputationGraph::nan ? g.size() : endNode); ++node) {

        // if a node is computed by an op applied to predecessors ...
        if (!g.predecessors(node).empty()) {

            // evaluate the node

            std::vector<const T*> args(g.predecessors(node).size());
            for (std::size_t arg = 0; arg < g.predecessors(node).size(); ++arg) {
                args[arg] = &values[g.predecessors(node)[arg]];
            }
            values[node] = ops[g.opId(node)](args);

            QL_REQUIRE(values[node].initialised(), "forwardEvaluation(): value at active node "
                                                   << node << " is not initialized, opId = " << g.opId(node));

            // then check if we can delete the predecessors
            if (deleter) {
                ...
            }     // if deleter
        }         // if !g.predecessors empty
    }             // for node
}
```

settings in pricingengine.xml (UseCG, UseAD). We can use the computation graph version even without activating AD.

Apparently these implementation have somewhat different scope (see localvol on the "classic" side, hwcg, and lgmcg on the "CG" side), which shows that the module is still in progress, on both sides with and without computation graph utilisation. Generally the duplication of code here (about 3500 lines of code in the models and engine classes of type CG) seems undesireable, disadvantageous from a development and maintenance effort point of view.

However, our plan is to continue following and building out the computation graph

models/model.hpp	models/modelcg.*pp
models/modelimpl.*pp	models/modelcgimpl.*pp
models/blackscholes.hpp	models/blackscholescg.*pp
models/blackscholesbase.*pp	models/blackscholescgbase.*pp
models/gaussiancam.*pp	models/gaussiancamcg.*pp
models/localvol.*pp	–
–	models/hwcg.*pp
–	models/lgmcg.*pp
engines/scriptedinstrumentpricingengine.*pp	engines/scriptedinstrumentpricingenginecg.*pp
–	computationgraphbuilder.*pp

Table 2.1: "Classic" scripting vs "Computation Graph", all files is ore/scripting.

Listing 9: Forward derivaties calculation on a computation graph in qle/ad/forwardderivatives.hpp

```
template <class T>
void forwardDerivatives(const ComputationGraph& g, const std::vector<T>& values, std::vector<T>& derivatives,
                        const std::vector<std::function<std::vector<T>(const std::vector<const T*>&, const T*)>>& grad,
                        std::function<void(T&)> deleter = {}, const std::vector<bool>& keepNodes = {},
                        const std::size_t conditionalExpectationOpId = 0,
                        const std::function<T(const std::vector<const T*>&)>& conditionalExpectation = {}) {

    if (g.size() == 0)
        return;

    // loop over the nodes in the graph in forward order

    for (std::size_t node = 0; node < g.size(); ++node) {
        if (!g.predecessors(node).empty()) {

            // propagate the derivatives from predecessors of a node to the node

            std::vector<const T*> args(g.predecessors(node).size());
            for (std::size_t arg = 0; arg < g.predecessors(node).size(); ++arg) {
                args[arg] = &values[g.predecessors(node)[arg]];
            }

            if (g.opId(node) == conditionalExpectationOpId && conditionalExpectation) {

                args[0] = &derivatives[g.predecessors(node)[0]];
                derivatives[node] = conditionalExpectation(args);

            } else {

                auto gr = grad[g.opId(node)](args, &values[node]);

                for (std::size_t p = 0; p < g.predecessors(node).size(); ++p) {
                    derivatives[node] += derivatives[g.predecessors(node)[p]] * gr[p];
                }
            }

            // the check if we can delete the predecessors

            if (deleter) {
                ...
            }     // if deleter
        }         // if !g.predecessors empty
    }             // for node
}
```

Listing 10: Backward derivaties calculation on a computation graph in qle/ad/backwardderivatives.hpp

```
template <class T>
void backwardDerivatives(const ComputationGraph& g, std::vector<T>& values, std::vector<T>& derivatives,
                         const std::vector<std::function<std::vector<T>(const std::vector<const T*>&, const T*)>>& grad,
                         std::function<void(T&)> deleter = {}, const std::vector<bool>& keepNodes = {},
                         const std::vector<std::function<T(const std::vector<const T*>&)>>& fwdOps = {},
                         const std::vector<std::function<std::pair<std::vector<bool>, bool>(const std::size_t)>>&
                             fwdOpRequiresNodesForDerivatives = {},
                         const std::vector<bool>& fwdKeepNodes = {}, const std::size_t conditionalExpectationOpId = 0,
                         const std::function<T(const std::vector<const T*>&)>& conditionalExpectation = {}) {

    if (g.size() == 0)
        return;

    std::size_t redBlockId = 0;

    // loop over the nodes in the graph in reverse order

    for (std::size_t node = g.size() - 1; node > 0; --node) {

        if (g.redBlockId(node) != redBlockId) {

            // delete the values in the previous red block
            ...

            // populate the values in the current red block
            ...

            // update the red block id
            redBlockId = g.redBlockId(node);
        }

        if (!g.predecessors(node).empty() && !isDeterministicAndZero(derivatives[node])) {

            // propagate the derivative at a node to its predecessors
            std::vector<const T*> args(g.predecessors(node).size());
            for (std::size_t arg = 0; arg < g.predecessors(node).size(); ++arg) {
                args[arg] = &values[g.predecessors(node)[arg]];
            }

            QL_REQUIRE(derivatives[node].initialised(),
                       "backwardDerivatives(): derivative at active node " << node << " is not initialized.");

            if (g.opId(node) == conditionalExpectationOpId && conditionalExpectation) {

                // expected stochastic automatic differentiaion, Fries, 2017
                args[0] = &derivatives[node];
                derivatives[g.predecessors(node)[0]] += conditionalExpectation(args);

            } else {

                auto gr = grad[g.opId(node)](args, &values[node]);

                for (std::size_t p = 0; p < g.predecessors(node).size(); ++p) {
                    ...
                    derivatives[g.predecessors(node)[p]] += derivatives[node] * gr[p];
                }
            }
        }

        // then check if we can delete the node
        ...

    } // for node
}
```

Listing 11: Scripting model base class `ModelCG` *using a computation graph.*

```
class ModelCG : public QuantLib::LazyObject {
public:
    enum class Type { MC, FD };

    explicit ModelCG(const QuantLib::Size n);

    // computation graph
    boost::shared_ptr<QuantExt::ComputationGraph> computationGraph() { return g_; }

    // result must be as of max(refdate, obsdate); refdate < paydate and obsdate <= paydate required
    virtual std::size_t pay(const std::size_t amount, const Date& obsdate, const Date& paydate,
                            const std::string& currency) const = 0;

    // refdate <= obsdate <= paydate required
    virtual std::size_t discount(...)

    // refdate <= obsdate required
    virtual std::size_t npv(const std::size_t amount, const Date& obsdate, const std::size_t filter,
                            const QuantLib::ext::optional<long>& memSlot, const std::size_t addRegressor1,
                            const std::size_t addRegressor2) const = 0;
    ...

    // CG / AD part of the interface
    virtual std::size_t cgVersion() const = 0;
    virtual const std::vector<std::vector<std::size_t>>& randomVariates() const = 0;
    virtual std::vector<std::pair<std::size_t, double>> modelParameters() const = 0;
    ...
};
```

path (adding more scripting models over time) and to keep the "classic" implementation available for validation and testing for part of the computation graph framework.

Key differences:

- `ScriptedInstrumentPricingEngineCG` calls the `ComputationGraphBuilder` which constructs the computation graph from the AST and delegates the pricing to the `forwardEvaluatiuon` function defined in qle/ad that operates on the computation graph; it does *not* use the `ASTRunner` any more
- the scripting models, starting with base class `ModelCG`, have a similar interface as the "non-CG" counterparts with `RandomVariable` arguments replaced with indexes (of type std::size_t) referencing nodes in the computation graph, see the cut-down listing of `ModelCG` in listing 11 and compare to the `Model` interface in listing 7; the scripting model classes provide additional member functions that reference random variates and model parameters on the graph;

2.3.3 NPV Sensitivities with AAD

Now we can utilise the backward sensitivity calculation on the computation graph to compute trade/portfolio sensitivities (for scripted trades). All we need to do for that purpose – as a user of ORE – is providing a scripted trade pricing engine configuration of the kind in listing 12.

i.e. with `Engine` set to MC and `UseAD` flag set to true. This triggers the construction of the computation graph version of scripting models and script engine.

When we run the usual pricing/sensitivity analytic in ORE, this will cause the generation of backward sensitivities in the first pricing call, and sensitivities are cached

Listing 12: Scripted Trade pricing engine configuration to generate AAD sensitivities: `UseAD=true`.

```
<PricingEngines>
  <Product type="ScriptedTrade">
    <Model>Generic</Model>
    <ModelParameters>
      <Parameter name="Model">GaussianCam</Parameter>
      ...
    </ModelParameters>
    <Engine>Generic</Engine>
    <EngineParameters>
      <Parameter name="Engine">MC</Parameter>
      <Parameter name="Samples">8192</Parameter>
      <Parameter name="RegressionOrder">4</Parameter>
      <Parameter name="TimeStepsPerYear">1</Parameter>
      <Parameter name="Interactive">false</Parameter>
      <Parameter name="BootstrapTolerance">1.0</Parameter>
      <Parameter name="ZeroVolatility">false</Parameter>
      <Parameter name="Interactive">false</Parameter>
      <Parameter name="UseAD">true</Parameter>
    </EngineParameters>
  </Product>
</PricingEngines>
```

as pricing engine additional results, see upper half of listing 13. When we then run the usual bump & revalue sensitivity scenarios and reprice the instrument repeatedly, then we read the cached sensitivities, scale them to the desired bump size and populate the sensitivity cube, see lower half of listing 13. This is fully integrated with the rest of the portfolio represented as "classic" trades.

2.3.4 XVA Sensitivities with AAD

The XVA sensitivity calculation using AAD is in development in ORE and in an experimental, proof of concept state at the time of writing this text.

ORE Example 56 demonstrates the current functionality using ORE's command line interface. Note that interface and implementation details are subject to change.

The key idea is to reuse ORE's ComputationGraph in the scripted trade framework. The case in Example 56 is particularly simple, a single Swap.

The AAD XVA sensitivity calculation is triggered in ore.xml as shown in listing 14, i.e. with parameters

- amc set to true, so that we run the exposure calculation with AMC
- amcCg set to true, so that we call a new `XvaEngineCG` (defined in orea/engine/xvaenginecg.*pp) that has the capability to generates both XVA and XVA sensitivities, when running the XVA analytic
- xvaSensitivityConfig set to a valid sensitivity config file

Moreover, we need to set `UseCG` set to `true` in the pricing engine configuration 15 used in the simulation phase, so that we build the trade using the computation graph scripting models.

When running the XvaAnalytic with the settings above, we will bypass the usual process and call into a new analytic `XvaAnalyticCG` defined in

Listing 13: Scripted Instrument Pricing Engine CG - this excerpt shows the caching of backward sensitivities upon the first pricing call (upper part), as well as their utilisation on subsequent calls to simulate NPVs in line with base NPV, sensitivities and risk factor bump sizes.

```
void ScriptedInstrumentPricingEngineCG::calculate() const {
    ...
    if (!haveBaseValues_ || !useCachedSensis_) {
        ...
        if (useCachedSensis_) {

            // extract sensis and store them

            std::vector<RandomVariable> derivatives(g->size(), RandomVariable(model_->size(), 0.0));
            derivatives[cg_var(*g, npv_ + "_0")] = RandomVariable(model_->size(), 1.0);
            backwardDerivatives(*g, values, derivatives, grads_, RandomVariable::deleter, keepNodes);

            sensis_.resize(baseModelParams_.size());
            for (Size i = 0; i < baseModelParams_.size(); ++i) {
                sensis_[i] = model_->extractT0Result(derivatives[baseModelParams_[i].first]);
            }
            DLOG("got backward sensitivities");

            // set flag indicating that we can use cached sensis in subsequent calculations

            haveBaseValues_ = true;
        }

    } else {

        // useCachedSensis => calculate npv from stored base npv, sensis, model params

        auto modelParams = model_->modelParameters();

        double npv = baseNpv_;
        DLOG("computing npv using baseNpv " << baseNpv_ << " and sensis.");

        for (Size i = 0; i < baseModelParams_.size(); ++i) {
            QL_REQUIRE(...)
            Real tmp = sensis_[i] * (modelParams[i].second - baseModelParams_[i].second);
            npv += tmp;
            DLOG(...)
        }
        results_.value = npv;
    }
    ...
}
```

orea/engine/xvaenginecg.*pp, as shown in listing 16.

This new analytic covers the following steps

- build today's market
- build a simulation market
- build the `GaussianCamCG` scripting model
- build the portfolio against the latter
- build the computation graph for all trades
- add nodes to the computation graph which sum the exposure over trades
- add nodes for the CVA calculation
- run a forward evaluation

Listing 14: ore.xml settings for XVA sendsitivity calculation using AAD: `amc=true`, `amcCg=true`, `xvaCgSensitivityConfigFile` *set.*

```
    <Analytic type="simulation">
      <Parameter name="active">true</Parameter>
      <Parameter name="amc">true</Parameter>
      <Parameter name="amcCg">true</Parameter>
      <Parameter name="xvaCgSensitivityConfigFile">xvasensiconfig.xml</Parameter>
      <Parameter name="amcTradeTypes">Swap</Parameter>
      <Parameter name="amcPricingEnginesFile">pricingengine_amc.xml</Parameter>
      ...
    </Analytic>
```

Listing 15: pricingengine.xml for XVA sensitivity calculation using AAD: `UseCG=true`.

```
  <Product type="ScriptedTrade">
    <Model>Generic</Model>
    <ModelParameters>
      <Parameter name="Model">GaussianCam</Parameter>
      ...
    </ModelParameters>
    <Engine>Generic</Engine>
    <EngineParameters>
      <Parameter name="Engine">MC</Parameter>
      <Parameter name="UseCG">true</Parameter>
      ...
    </EngineParameters>
  </Product>
```

- write exposure reports
- compute CVA as expectation over random variable values in the CVA node
- do a backward derivatives run
- fill the sensitivity cube by copying the AAD derivatives (or do repeated forward valuations for bump sensitivities); this is currently controlled by a hard-coded boolean `bumpCvaSensis`
- write the sensitivity report

i.e. it replaces the entire XVA analytic and the post processing of the NPV cube.

For testing/validation purposes (accuracy of results, performance) we can activate bump & revalue sensitivity calculation by setting the hard coded boolean `bumpCvaSensis=true` in orea/engine/xvaenginecg.cpp.

Listing 16: Call into the experimental XVA AAD-Sensitivity implementation, bypassing the ususal XVA run.

```
void XvaAnalyticImpl::runAnalytic(const boost::shared_ptr<ore::data::InMemoryLoader>& loader,
                                  const std::set<std::string>& runTypes) {
    if(inputs_->amcCg()) {
        LOG("XVA analytic is running with amc cg engine (experimental).");
        XvaEngineCG engine(
            inputs_->nThreads(), inputs_->asof(), loader, inputs_->curveConfigs().get(),
            analytic()->configurations().todaysMarketParams, analytic()->configurations().simMarketParams,
            inputs_->amcPricingEngine(), inputs_->crossAssetModelData(), inputs_->scenarioGeneratorData(),
            inputs_->portfolio(), inputs_->marketConfig("simulation"), inputs_->marketConfig("simulation"),
            inputs_->xvaCgSensiScenarioData(), inputs_->refDataManager(), inputs_->iborFallbackConfig());
        return;
    }
```

2.3.5 Dynamic Delta with AAD

The Dynamic Delta sensitivity calculation for IM modeling using AAD is in development in ORE and in an experimental, proof of concept state at the time of writing this text.

2.3.5.1 General Setup

The starting point are the conditional portfolio npvs on valuation times t

$$E\left(\sum_i \frac{X_i S_i(t_i)}{N(t_i)}\middle|\mathcal{F}_t\right)$$

where X_i are future flows paid at $t_i > t$, $S(t_i)$ is the fx rate to convert flow to base currency and $N(t_i)$ is the numeraire at time t_i and $\mathcal{F}_t$ is the filtration representing the infomration at time t. The goal is to compute sensitivities

$$\frac{\partial}{\partial p} E\left(\sum_i \frac{X_i S_i(t_i)}{N(t_i)}\middle|\mathcal{F}_t\right)$$

w.r.t. a vector of par market risk factors p. We write

$$\bar{X} := \sum_i \frac{X_i S_i(t_i)}{N(t_i)}$$

in what follows. We decompose the calculation of derivatives of $\bar{X}$ to par market risk factors into

- computing derivatives of the conditional portfolio npvs to model parameters m
- computing derivatives of model parameters m to zero market risk factors z
- computing derivatives of zero market risk factors z to par market risk factors p

and using the multivariate chain rule to combine the results, i.e.

$$\frac{\partial}{\partial p} = \frac{\partial}{\partial m}\frac{\partial m}{\partial z}\frac{\partial z}{\partial p}$$

Note: We generally decompose a portfolio into subportfolios and compute the dynamic delta on each subportfolio separately. See 2.3.5.6

2.3.5.2 Model Parameters

Table 2.2 lists the model parameters that we use[2]. Model parameters are generally identified by their type and a subset of additional keys (qualifier1, qualifier2, date, date2, date3, index, index2, hash).

[2] see modelcg.hpp, we only list non-derived parameters and those used in GaussianCamCG

parameter name	meaning
fix	historical index fixing
dsc	T0 ir discount
lgm_H	lgm1f parameter
lgm_Hprime	lgm1f parameter
lgm_zeta,	lgm1f parameter
fxbs_sigma,	fxbs parameter
logFxSpot,	T0 log fx spot
sqrtCorr,	model sqrt correlation
sqrtCov,	model sqrt covariance
corr,	model correlation
cov,	model cov
cam_corrzz,	ir-ir corr
cam_corrzx,	ir-fx corr

Table 2.2: Model parameters used by GaussianCamCG

2.3.5.3 Computing derivatives of conditional portfolio npvs to model parameters m

We use

$$\frac{\partial}{\partial m} E\left(\bar{X}\middle|\mathcal{F}_t\right) = E\left(\frac{\partial \bar{X}}{\partial m}\middle|\mathcal{F}_t\right)$$

and rely on the backward derivatives algorithm to compute pathwise derivatives

$$\frac{\partial \bar{X}}{\partial m}$$

to model parameters m.

2.3.5.4 Computing derivatives of model parameters m to zero market risk factors z

The relevant model parameters are dsc, lgm_zeta, fxbs_sigma and logFxSpot.

IR zero rate deltas

Suppose we have a sensitivity s of $\bar{X}$ to a model parameter of type dsc referencing a maturity time T, $T > t$, i.e.

$$s = \frac{\partial \bar{X}}{\partial P(0,T)}$$

with the deterministic T0 discount factor $P(0,T)$. The aim is to convert s to a sensitivity to the conditional zero rate as seen from t with maturity $T-t$. We write

$$Q(t,T,x(t)) := P(t,T)d$$

for the conditional zero bond with maturity T as seen from t conditional on the model state $x(t)$. In the LGM1F model we have

$$d = e^{-(H(T)-H(t))x(t)-\frac{1}{2}(H(T)^2-H(t)^2)\zeta(t)}$$

but the calculation goes through for any model where we can decompose $Q(t,T)$ as $P(t,T)\cdot d$. We write the deterministic zero bond

$$P(t,T) = e^{-z\cdot(T-t)}$$

in terms of the deterministic zero rate z and the conditional zero bond as

$$Q(t,T,x(t)) = e^{-z(T-t)}d =: e^{-\tilde{z}(T-t)}$$

We are ultimately interested in the derivative of $\bar{X}$ w.r.t. $\tilde{z}$. We have:

$$\frac{\partial \bar{X}}{\partial \tilde{z}} = \frac{\partial \bar{X}}{\partial P(t,T)}\frac{\partial P(t,T)}{\partial Q(t,T,x)}\frac{\partial Q(t,T,x)}{\partial \tilde{z}}$$

Since

$$\frac{\partial P(t,T)}{\partial Q(t,T,x)}\frac{\partial Q(t,T,x)}{\partial \tilde{z}} = \frac{1}{d}\left[-(T-t)Q(t,T,x)\right] = -(T-t)P(t,T)$$

we get

$$\frac{\partial \bar{X}}{\partial \tilde{z}} = -(T-t)\frac{\partial \bar{X}}{\partial P(t,T)}P(t,T)$$

and since

$$\frac{\partial \bar{X}}{\partial P(t,T)} = \frac{\partial \bar{X}}{\partial P(0,T)}\frac{\partial P(0,T)}{\partial P(t,T)} = \frac{\partial \bar{X}}{\partial P(0,T)}P(0,t)$$

we get the final formula to convert s to a market zero risk factor sensitivity:

$$\frac{\partial \bar{X}}{\partial \tilde{z}} = -(T-t)P(0,T)\frac{\partial \bar{X}}{\partial P(0,T)} = -(T-t)P(0,T)s$$

As a final step, we distribute the sensitivity on predefined buckets linearly.

FX deltas

Assume a sensitivity s of $\bar{X}$ to a model parameter of type logFxSpot:

$$s = \frac{\partial \bar{X}}{\partial \ln S(0)}$$

with $S(0)$ the T0 FX Spot rate. The log FX Spot rate $S(t)$ at time t is given by

$$\ln S(t) = \ln S(0) + \int_0^t \left(r(s) - d(s) - \frac{\sigma(s)^2}{2} \right) ds + \int_0^t \sigma(s) dW(s)$$

where $r(s)$ and $d(s)$ are the instantaneous, stochastic domestic and foreign short rates and $\sigma(s)$ is the volatility of $S(t)$. We are assuming $S(t)$ follows a GBM here, but the argument is valid more generally. We get

$$\frac{\partial \ln S(t)}{\partial \ln S(0)} = 1$$

and therefore

$$\frac{\partial \bar{X}}{\partial \ln S(t)} = \frac{\partial \bar{X}}{\partial \ln S(0)}$$

Note also that this is the derivative w.r.t. to a relative shift in $S(t)$ resp. $S(0)$.

IR vegas

A sensitivity s of $\bar{X}$ to a model parameter of type lgm_zeta

$$s = \frac{\partial \bar{X}}{\partial \zeta(T)}$$

with $T > t$ is directly distributed on the defined bucket structure linearly. The conversion to a market vega is done in par-conversion step, see below.

FX vegas

A sensitivity s of $\bar{X}$ to a model parameter of type fxbs_sigma

$$s = \frac{\partial \bar{X}}{\partial \sigma(T)}$$

with $T > t$ is converted to a market vega in the par-conversion step, see below.

2.3.5.5 Computing derivatives of zero market risk factors z to par market risk factors p

IR zero to par rate deltas

The conversion of IR zero to par rate deltas follow the usual procedure via Jacobian transformation. There are possible simplifications that can be exploited to speed up the conversion. We currently apply both of them in ore:

- state-independent par conversion
- time-independent par conversion

IR vega par conversion

The conversion of sensitivities to the lgm_zeta parameter to sensitivities to implied swaption vol is done in analogy to the IR delta par conversion by specifying a set of swaptions and calculating the Jaocbian of these swaptions to the lgm zeta parameter. The same possible simplifications as for IR par delta conversion exist and we apply both in ore currently:

- state-independent par conversion
- time-independent par conversion

FX vega par conversion

The conversion of sensitivities to the fxbs_sigma parameter to sensitivities to implied fx option vols is done in analogy to the IR delta par conversion by specifying a set of fx options and calculating the Jacobian of these fx swaptions to the fx bs sigma parameter. The same possible simplifications as for IR par delta conversion exist and we apply both in ore currently:

- state-independent par conversion
- time-independent par conversion

2.3.5.6 Portfolio decomposition

In general the calculations described in the previous sections are performed on suitable subsets of the whole portfolio. First of all, if the dynamic delta is required for each individual trade, these subsets consist of a single trade. If the dynamic delta is only required on the portfolio level, the portfolio is first decomposed into

- "plain vanilla" trades for which the calculations can be done on an the level of aggregated npvs
- complex trades which require an individual calculation for each trade

Linear trades like vanilla ir single and cross currency swaps and fx forwards fall into the first category while trades involving optionality and other exotic features fall into the second category.

2.3.5.7 Portfolio decomposition: Plain Vanilla Part

The plain vanilla part of the portfolio can be further subdivided by common regressor sets required to calculate conditional expectations. In general, conditional expectations for both values and derivatives of plain vanilla trades can be calculated using a simple linear regression with polynomial basis functions, as assposed to complex trades that might require more advanced models. The complexity of e.g. QR or SVD underlying the linear regression analysis is $O(mn^2)$ with m the number of observations, n the number of regressor variables, $m > n$, i.e. quadratic in the number of regressors. Therefore it is reasonable to run the regression analyses on subportfolios with a smallest possible set of regressors instead of combining trades with disjoint regressors.

2.3.5.8 Portfolio decomposition: Complex Part

The trades falling into the complex part of the portfolio are calculated one by one. As an example we consider a Bermudan swaption, i.e. an option to enter a vanilla underlying at multiple exercise times. Denote the exercise times by

$$t_1, t_2, \ldots, t_n$$

For each exercise time the Longstaff-Schwartz algorithm yields an exercise indicator e_i, $i = 1, \ldots, n$, which is 1 if the option is exercised at time t_i and 0 otherwise. We introduce

$$E_i := \sum_{j=1}^{i} e_j$$

indicating that the option was exercised on or before time t_i. Now consider a simulation time t_{sim} with an associated exercised value v_e representing the path value of the exercised-into underlying at t_{sim} and a future option value f_e representing the continuation value of the option at t_{sim} in case the option was not yet exercised. We proceed with calculating conditional expectations of v_e

$$V_e := E(v_e|\mathcal{F}_t) \tag{2.2}$$

and f_e

$$F_e := E(f_e|\mathcal{F}_t) \tag{2.3}$$

at $t = t_{sim}$ using an appropriate regression analysis. We set the trade exposure T at t_{sim} to

$$T := E_i V_e + 1_{F_e>0}(1 - E_i)F_e \tag{2.4}$$

where i is determined such that E_i is the relevant exercise indicator (set $i = 0$ and $E_0 := 0$ if $t_{sim} < t_1$). Not that we floor F_e at 0 which is not guaranteed due to artifacts of the regression model.

For the purpose of dynamic delta calculation we differentiate V_e and F_e w.r.t. a model parameter m by taking the conditional expectations of the derivatives of v_e

$$\frac{\partial V_e}{\partial m} = E\left(\frac{\partial}{\partial m} v_e \middle| \mathcal{F}_t\right) \tag{2.5}$$

and of f_e

$$\frac{\partial F_e}{\partial m} = E\left(\frac{\partial}{\partial m} f_e \middle| \mathcal{F}_t\right) \tag{2.6}$$

to recombine these to the derivative of T

$$\frac{\partial T}{\partial m} = E_i \frac{\partial V_e}{\partial m} + 1_{F_e>0} \, (1 - E_i) \frac{\partial F_e}{\partial m} \tag{2.7}$$

Note on indicator derivatives: We do not consider non-trivial indicator derivatives of E_i since this is an indicator of *past* exercise. The treatment of derivatives for *future* exercise indicators that are implicit in the calculation of F_e are not detailled here, we refer to the known result that such indicators do not require differentiation, see e.g. the discussion in [23], section 10. Finally, we do not consider derivatives of $1_{F_e>0}$ because this indicator reflects regression model artifcats only.

Note on decomposition: One may ask why we do not directly consider the path value T' of the exotic trade, i.e.

$$T' := E_i v_e + (1 - E_i) f_e$$

and the partial derivative

$$\frac{\partial T'}{\partial m} = E_i \frac{\partial v_e}{\partial m} + (1 - E_i) \frac{\partial f_e}{\partial m}$$

and then take the conditional expectation,

$$E\left(\frac{\partial T'}{\partial m}\middle|\mathcal{F}_t\right) = E\left(E_i \frac{\partial v_e}{\partial m} + (1 - E_i) \frac{\partial f_e}{\partial m}\middle|\mathcal{F}_t\right) \tag{2.8}$$

This would be more consistent with the plan vanilla trades and would not require a distinction of vanilla and exotic trades in the implementation. However, it would also not allow to floor F_e at zero as in 2.4 and - much more importantly - this would also require to include suitable regressors to explain past exercise decisions represented by E_i, such as e.g. suitable model states at all past exercise times $t_1, t_2, \ldots, t_i$, since E_i is part of the regressand, i.e. inside the conditional expectation operator in 2.8 as opposed to 2.7 where it is not part of the regressand.

To keep the implementation of the above decomposition approach as general as possible, we introduce

- component path values of the exotic trade $c_1, c_2, \ldots, c_N$
- a mini computation graph representing the calculation of the target exposure value $T = T(c_1, \ldots, c_N)$ as of function of the components c_i

The components in the example above are $c_1 = v_e$ and $c_2 = f_e$ and the mini computation graph represents the calculations in equations 2.2, 2.3, 2.4. The derivatives calculation then assigns $\partial V_e \partial m$ and $\partial F_e \partial m$ to c_1 resp. c_2 and replays the mini computation graph on these values to arrive at $\partial T \partial m$ from equation 2.7.

To give another example, for a TaRF we would have one component c_1 representing the path value f_e of the TaRF under the assumption that it was not terminated before the current simulation time. The function T would be given by

$$T(c_1) := (1 - E_i)E(c_1|\mathcal{F}_t)$$

where E_i denotes the indicator that is 1 if the structured was terminated before the simulation time. In addition we note that the regression model to calculate $E(c_1|\mathcal{F}_t)$ should include the accumulated profit of the TaRF as a regressor since this is important to explain the future termination probability.

2.4 External Compute Devices

In the application of ORE at industrial scale we occasionally hit cases where the performance of an ORE based service is not sufficient to keep processing times within agreed limits. One example are backtesting runs for particularly complex portfolios (including Accumulators and TaRFs) which require the repricing of the portfolio on several thousand market scenarios.

ORE's multi-threaded valuation engine helps utilising the available CPU cores, but the speed-up we can achieve with multi-threading in ORE is limited by the typically low number of cores and also the overhead in setting up markets and portfolios for each thread. The speed-ups we have achieved with multi-threading on our available hardware has been well below factor 10 so far.

Thanks to the growing popularity of AI tools and need for machine learning infrastructure graphics cards with hundreds of GPU cores have become more broadly available. One can easily rent AWS hardware with A100 Nvidia cards nowadays. Moreover, compiler support for such hardware has much improved over the past ten years. So it is time to utilise such devices with ORE to cut down run times by orders of magnitude where required by an SLA.

The computation graph representation of a scripted trade is also suited to build a particularly slim interface to an external compute device (GPU), because the computation graph is a very low level "atomic" representation of the algorithms we need to run. The idea is

- to compile the processing of a computation graph into a GPU kernal
- to run the computation graph multiple (thousands of) times with changing context
- to minimise the amount of data exchange between CPU and GPU

To support external devices, ORE provides a compute framework interface, defined in see qle/math/computeenvironment.*pp. This contains three class declarations, two of which (the latter two) need to be implemented in derived classes, see listings 18 and 17

- `ComputeEnvironment`: A thread-local singleton that exposes external compute frameworks to ORE. A new framework has to be registered to this singleton.
- `ComputeFramework`
- `ComputeContext`

Review the technical documentation for the compute environment interface in Docs/ComputeEnvironment at https://github.com/OpenSourceRisk/Engine.

Listing 17: Compute framework, provides the Compute Context for each device.

```
class ComputeFramework {
public:
    virtual ~ComputeFramework() {}
    virtual std::set<std::string> getAvailableDevices() const = 0;
    virtual ComputeContext* getContext(const std::string& deviceName) = 0;
};
```

Listing 18: The context in which calculations can be performed. The `ComputeFramework` *provides a pointer to the* `ComputeContext` *for each device.*

```
class ComputeContext {
public:
    ...
    virtual std::pair<std::size_t, bool> initiateCalculation(const std::size_t n,
                                                             const std::size_t id = 0,
                                                             const std::size_t version = 0,
                                                             const bool debug = false) = 0;
    virtual std::size_t createInputVariable(double v) = 0;
    virtual std::size_t createInputVariable(double* v) = 0;
    virtual std::vector<std::vector<std::size_t>> createInputVariates(const std::size_t dim,
                                                                      const std::size_t steps,
                                                                      const std::uint32_t seed) = 0;
    virtual std::size_t applyOperation(const std::size_t randomVariableOpCode,
                                       const std::vector<std::size_t>& args) = 0;
    virtual void freeVariable(const std::size_t id) = 0;
    virtual void declareOutputVariable(const std::size_t id) = 0;

    virtual void finalizeCalculation(std::vector<double*>& output, const Settings& settings) = 0;

    ...
};
```

This interface is not tied to the scripted trade framework in principle. But so far the only use case in ORE is linked to the scripted trade framework with computation graph: In order to utilise an external device with ORE we need to set a few pricing engine parameters as shown in listing 19.

Listing 19: Pricing engine configuration for using an external device.

```
<PricingEngines>
  <Product type="ScriptedTrade">
    ...
    <EngineParameters>
      ...
      <Parameter name="UseCG">true</Parameter>
      <Parameter name="UseExternalComputeDevice">true</Parameter>
      <Parameter name="ExternalComputeDevice">OpenCL/Apple/AMD Radeon Pro 5500M Compute Engine</Parameter>
    </EngineParameters>
  </Product>
  ...
</PricingEngines>
```

So far there are two implementations of the ComputeContext that can be selected via the `ExternalComputeDevice` parameter

- a dummy implementation called "BasicCpuContext" which will utilise the CPU cores to do the work, see qle/math/basiccpuenvironment.*pp
- an OpenCL reference implementation (in experimental state at the time of writing this text), see qle/math/openclenvironment.*pp

and a third implementation (CUDA) has been started. Both the OpenCL and CUDA implementations are expected to be completed by the time of the 13th ORE release mid 2024.

Chapter 3

ORE Integration

3.1 Language Bindings

To facilitate ORE adoption and integration it is useful to "bind" ORE with languages other than C++, because C++ development familiarity is less widely spread than e.g. Python or Java. Especially the younger generation of computer scientists are typically versatile in Python, because of the vast landscape of Python packages that one gets easily with "pip install ...", and perhaps in particular because of the popularity of machine learning and related Python tools such as TensorFlow.

SWIG

ORE language bindings are currently built using SWIG [24], following the QuantLib example. The project is located in the separate repository https://github.com/opensourcerisk/ore-swig.

The initial goal of the ORE-SWIG project was to provide a working *framework* that allows building *a single module* that provides access to classes and functions *across ORE and QuantLib*. Primarily we wanted to be able to execute the following few lines in Python, in order to kick off an ORE process.

```
from ORE import *
params = Parameters()
params.fromFile("Input/ore.xml")
ore = OREApp(params)
ore.run()
```

The secondary goal was to access and manipulate underlying ORE and QuantLib objects that are assembled during the OREApp.run() call. This is why we wanted to include QuantLib in our ORE module.

The QuantLib project has used SWIG for the purpose of wrapping QuantLib since the early 2000s, because the team has been aiming for exposing QuantLib to many languages including Java, Python, R and C#. To utilise the existing wrapper code base of QuantLib we therefore decided to also adopt the SWIG approach for ORE.

The main interest among ORE and QuantLib users in the industry seems to be in the Python language bindings, although we also supported clients in using the Java

module. If we just wanted to wrap ORE classes into Python and abandon access to the underlying QuantLib universe, then we would probably adopt a different framework such as boost.Python (https://github.com/boostorg/python) to build the module.

Current Scope

Since the first release of ORE-SWIG, ORE coverage has grown. However it is still patchy at about 11% of the classes in ORE wrapped (about 200/1850, see appendix A), compared to 44% coverage in QuantLib (about 920/2100).

As of today, users can query the `OREApp` class and retrieve portfolio, market and underlying term structures. Recently, while refactoring OREApp with

- separating analytics (see orea/app/analytics)
- ensuring results are kept primarily in-memory
- results can be retrieved via an extended API from OREApp
- separate file output of results on demand
- factoring out an input parameters object with API to set and get any member object
- input data instance can be passed to OREApp

we have extended the OREApp API and added the InputParameters API to the SWIG wrapper, for data management outside (before and after) calling `ORE.run()` and to facilitate ORE-based application development in Python.

We currently extend the SWIG wrapper scope mainly upon client request, rather than embarking on a systematic internal extension project.

Python Wheels

A notable addition in 2023 was establishing a process of building Python *wheels* using github actions post release. The wheels are published on the Python Package Index (PyPI) so that users can install the latest ORE Python module for their platform conveniently using the command

```
pip install open-source-risk-engine
```

again following the QuantLib example.

Similarly, we have added github actions to build ORE executables for Windows, macOS and Linux platforms automatically on github after each update of the github master branch, keeping the builds following a release, so that users can easily download the required executable and explore ORE without going through the complex build process from C++ sources.

When using the ORE wheel with pip install, we recommend the following steps to isolate the ORE environment

```
python -m venv ore_env
source ore_env/bin/activate (.\ore_env\Scripts\activate.bat)
python -m pip install open-source-risk-engine
```

Juyter Notebooks

The public Python wheels also make various Jupyter notebooks readily accessible that are published in the https://github.com/opensourcerisk/ore-swig repository's OREAnalytics-SWIG/Python/Examples/Notebooks folder. Juypter notebooks are convenient for prototyping ORE based applications, they can call into ORE, retrieve results in memory for post-processing including visualisations in the notebook, see a snapshot in figure 3.1. Run the notebook in the same environment as above

```
python -m pip install jupyterlab
python -m jupyterlab &
```

Launch ORE

Kick off a process in ORE, loading all inputs from Input/ore.xml and the files referenced therein. This is equivalent to using the ORE command line application.

[1]:
```
from ORE import *
import sys, time, math
sys.path.append('..')
import utilities

params = Parameters()
params.fromFile("Input/ore.xml")

ore = OREApp(params)
```

```
This should have loaded the main inputs ...
```

[2]:
```
ore.run()

utilities.checkErrorsAndRunTime(ore)
```

```
Run time: 61.14 sec
Errors: 0
```

```
The simulation process finally produces an NPV "cube": valuations for each trade, through time and
```

Uncollateralized Exposure

```
The simulation produces an NPV "cube": valuations for each trade, through time and across many sim
```

```
# Get exposure reports and plot ...
```

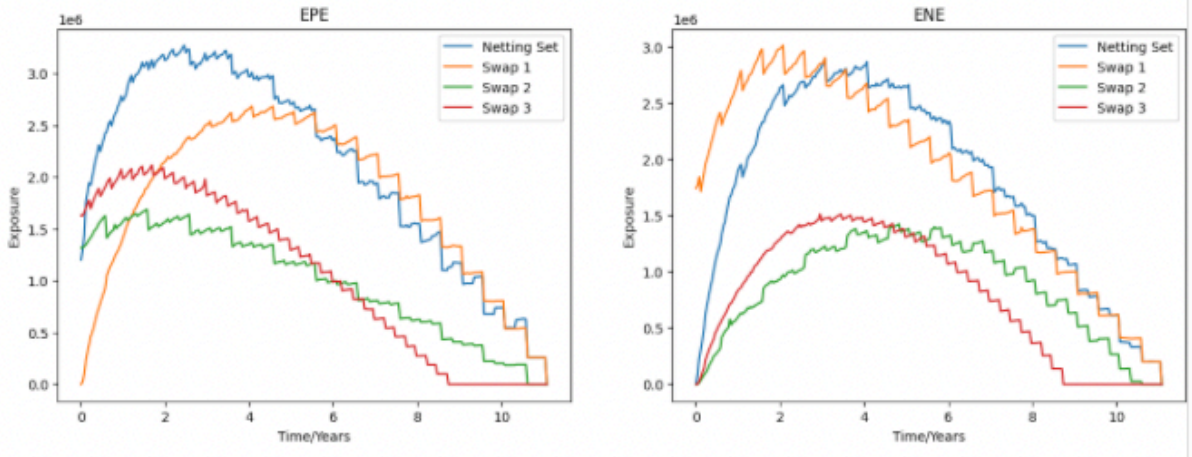

Figure 3.1: Jupyter notebook in OREAnalytics-SWIG/Python/Examples/Notebooks/Example_2, calling ORE for generating collateralised exposures, retrieving results in memory for post-processing including visualisation in the notebook.

Extending ORE SWIG

Developing the ORE SWIG wrapper involves adding interface files (with ending *.i) in folders

- QuantExt-SWIG/SWIG
- OREData-SWIG/SWIG
- OREAnalytics-SWIG/SWIG

where we – in a nutshell – add a header for each class we want to expose, including the desired subset of member function declarations. See any of the interface files in the folders above for examples and compare C++ class declarations to the interface definition in the *.i file. The `README.md` at https://github.com/opensourcerisk/ore-swig then provides links to tutorials for building the bindings and building Python wheels.

Extending the interface is simple in principle, but it is typically a slow and iterative process, because

- compilation time is long since the SWIG-generated C++ source file is particularly large (it covers all libraries - QuantLib, QuantExt, OREData and OREAnalytics)
- namespace conflicts occur regularly because all objects are imported from the QuantLib, QuantExt, ore::data, ore::analytics namespaces into the flat namespace; such conflicts typically occur between QuantExt and QuantLib where same class names may occur in both namespaces - these conflicts need to be resolved each time and may require renaming QuantExt classes or consolidating two classes into one
- compiler errors can be obscure and hard to trace to their source

The working bindings are hence precious, so that we build these internally upon any merge request into the ORE master branch.

3.2 ORE as a Service

To use ORE at industrial scale ORE needs to be deployable in an environment where applications communicate via services, possibly via the web, so that ORE should offer its functionality in the form of a service, too.

In the following we sketch a "service wrapper" around ORE that is not part of the public ORE on github, but proprietary software owned by Acadia. However, a similar public implementation with reduced scope is in progress at the time of writing this text and will be released in 2024.

RESTORE

RESTORE stands for **REST**ful API for **ORE**, a web service wrapper around ORE. REST stands for an architectural style that has been established in the 2000s for web application development. It claims amoung other things that

- clients and servers are clearly separated
- servers are stateless, do not maintain memory of past requests
- interfaces are "uniform", use the URI standard to identify required resources

There are various frameworks in the open source space that support the development of a server application and provide the technology for REST style client-server communication. Since ORE is written in C++, we were looking for such a framework in C++ too, free and open, and in 2016 we came across the C++ REST SDK, a Microsoft project for cloud based client-server communication in native C++.

Using this framework it is quite straight-forward to build the "restoreserver" application, an executable that can be kicked off in the command line. For production deployment, restore is built within a docker container. When the container is spun up it launches restoreserver which listens on a port for http requests. It is thus possible to launch multiple restore instances for parallel processing of requests.

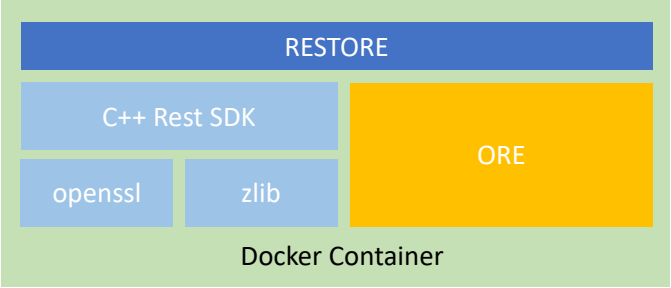

Figure 3.2: RESTORE wraps ORE and uses the C++ Rest SDK to implement a RESTful API. Additional dependencies are openssl for authentication and zlib for compression.

If restore receives e.g. a pricing request it will call into other services to retrieve trade data, market data, reference data and configurations, kick off the processing using ORE under the hood, send the results to a data service to manage, and finally send a response to the original request. Figure 3.3 illustrates the communication with and between services.

Before we look into restore implementation aspects, let's look at an example of how we need to structure a typical request and how to send a request to the restore server.

Let's assume we have launched restoreserver locally with

```
restoreserver --port=5003
```

to listen on port 5003.

Requests can now be sent to the server in many different ways. To test the server APIs we can use lightweight "clients" such Python scripts, Postman (see https://www.postman.com), or even Emacs. To use Python as a client, install the request package with "pip install requests" and send a simple GET request to the local restoreserver to check the version of its components:

```
import requests
import json
url = 'http://localhost:5003/api/version'
response = requests.get(url)
```

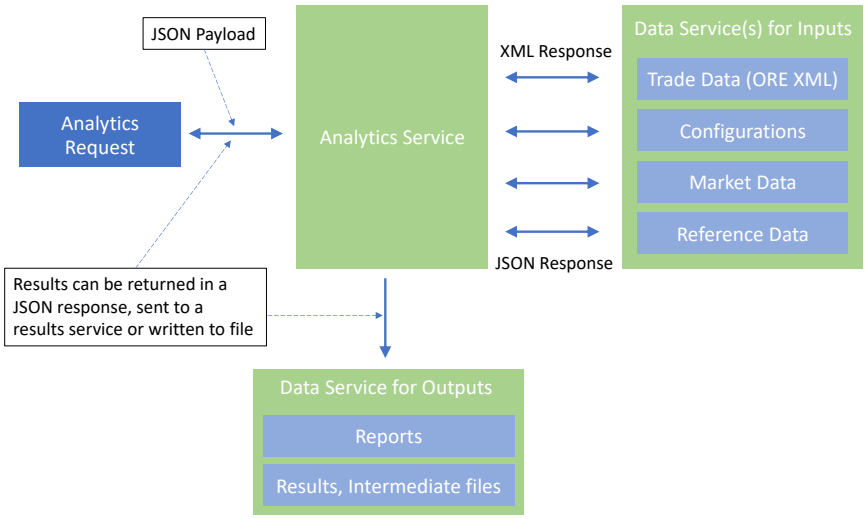

Figure 3.3: Request processing via the analytics service (RESTORE), data services for inputs and a results service.

```
print('Status:', response.status_code)
print('Response:', response.text)
print('Formatted response:', json.dumps(json.loads(response.text),indent=4))
```

Requests can be of type GET, PUT, POST, DELETE in restore. To ask restore to price a portfolio we need to send the following minimal POST request.

```
import requests
url = 'http://localhost:5003/api/pricer/default/analytics'
encoded_access_token = 'dG9rZW46dXNlcm5hbWU6cGFzc3dvcmQ='
header = { 'Authorization': 'Basic {}'.format(encoded_access_token),
           'Content-Type':'application/json'
}
body = { 'analytics': 'NPV',
         'baseCurrency': 'USD',
         'asof': 'YYYY-MM-DD',
         'portfolioUri': 'http://localhost:5000/file/portfolio.xml',
         'conventionsUri': 'http://localhost:5000/file/conventions.xml',
         'curveConfig': 'http://localhost:5000/file/curveconfig.xml',
         'scriptLibraryUri': 'http://localhost:5000/file/scriptlibrary.xml',
         'pricingEngineUri': 'http://localhost:5000/file/pricingengine.xml',
         'marketUri': 'http://localhost:5000/marketdata/market.txt/YYYY-MM-DD',
         'fixingsUri': 'https://localhost:5000/fixingdata/fixings.txt',
         'resultsPath': 'log',
         'reportUri': 'http://localhost:5000/report',
         'resultUri': 'http://localhost:5000/result',
         'outputAdditionalResults': 'true'
}
response = requests.post(url, params=body, headers=header)
print(response.text)
```

Note:

- The request is sent to a different URL or *end point* (pricer/default/analytics).
- We provide authorization details as part of the request header, this is typically required
- We specify the type of content in the body of the request, via the request header
- We pass the "body" as a second argument to the request. This body of the

request contains all information and essential URLs referencing familiar inputs that restore needs to retrieve in order to process the request.

- All URLs point to a single data service here that is running on the local host and listening on port 5000. We have implemented a simple data service using flask for testing/demo purposes which ultimately reads files from the data service' input directory; one could reference several different data services
- The report and result URIs are also pointing to the same data service, but different end points; these are used by the restore server to post results, in this case witten to files in the data service's output directory.
- The response in this case is thin, just reports the number of successfully processed trades; all results are found in the data service's output directory.

RESTORE wraps additional, so far proprietary *ORE+* functionality that has not been released to ORE yet, but essential for the service automation:

- Portfolio analyser: Determines the required market data, currencies, indexes and curves, dependent on portfolio and requested analytics
- Configuration builder: Build portfolio/analytics dependent configuration on the fly, using portfolio analser results – todaysmarket.xml, sensitivity.xml, simulation.xml – so that we do not need to provide these in the body of the request above
- Market data: Portfolio/analytics dependent tailoring of market data requests sent to the market data service
- Reference data: Retrieve portfolio dependent reference data such as equity dividends and stock splits, credit index constituents and Bond static data – so that users can simply reference a Bond via its security ID or a credit index by its RED code, and neither the user nor the service management team have to maintain such data manually

RESTORE Implementation

Source code components and their purpose

- **restoreserver.cpp**: Provide main function, set global http request parameters, build authentication header, manage pass-through headers (?), create a server instance (RestoreHandler), start the server

```
int main(int argc, char *argv[]) {
    ...
    RestoreHandler restoreHandler(address, swaggerDir, sslParameters, ptHeaders);
    try {
        // start the server
        restoreHandler.open().wait();
        ...
        restoreHandler.close().wait();
    } catch (...) {
      ...
    }
    ...
    return 0;
}
```

Overall size about 200 lines of code

- **restorehandler.*pp**: RestoreHandler implementation; register GET, POST, PUT, DEL methods with the listener, only the first two linked to fully implemented functions `handle_get` and `handle_post`, which is sufficient for restore's purpose; `handle_post` eventually delegates work to `handle_analytics` and further handlers, all defined outside restorehandler.cpp
  Overall size about 500 lines of code
- **handler_functions.*pp**: Reimplements the OREApp orchestration (eventually calling into ORE's analytics manager), retrieves inputs from various data service(s), manages sending results to the result service;
  Overall size about 2800 lines.

The full implementation of the restore application, used in Acadia's risk services, including all helper classes has less than 14k lines of code, still a relatively thin ($< 2\%$) but effective layer on top of core ORE/QuantLib.

Appendix A

SWIG Wrapper Scope

ORE SWIG wrapper scope as of the 11th release, 2023.

Table A.1: Wrapped classes in OREAnalytics

Class	SWIG Header
Analytic	orea_app.i
AnalyticsManager	orea_app.i
InputParameters	orea_app.i
MarketDataInMemoryLoader	orea_app.i
OREApp	orea_app.i
Parameters	orea_app.i
AggregationScenarioData	orea_cube.i
NPVCube	orea_cube.i

Table A.2: Wrapped classes in OREData

Class	SWIG Header
CalendarAdjustmentConfig	ored_calendarAdjustmentConfig.i
AverageOisConvention	ored_conventions.i
BMABasisSwapConvention	ored_conventions.i
CdsConvention	ored_conventions.i
Convention	ored_conventions.i
Conventions	ored_conventions.i
CrossCcyBasisSwapConvention	ored_conventions.i
CrossCcyFixFloatSwapConvention	ored_conventions.i
DepositConvention	ored_conventions.i
FXConvention	ored_conventions.i
FraConvention	ored_conventions.i
FutureConvention	ored_conventions.i
IRSwapConvention	ored_conventions.i
InflationSwapConvention	ored_conventions.i
OisConvention	ored_conventions.i
SecuritySpreadConvention	ored_conventions.i
SwapIndexConvention	ored_conventions.i
TenorBasisSwapConvention	ored_conventions.i
TenorBasisTwoSwapConvention	ored_conventions.i
ZeroRateConvention	ored_conventions.i
BaseCorrelationCurveSpec	ored_curvespec.i
CDSVolatilityCurveSpec	ored_curvespec.i
CapFloorVolatilityCurveSpec	ored_curvespec.i
CommodityCurveSpec	ored_curvespec.i
CommodityVolatilityCurveSpec	ored_curvespec.i
CorrelationCurveSpec	ored_curvespec.i
CurveSpec	ored_curvespec.i
DefaultCurveSpec	ored_curvespec.i
EquityCurveSpec	ored_curvespec.i

EquityVolatilityCurveSpec	ored_curvespec.i
FXSpotSpec	ored_curvespec.i
FXVolatilityCurveSpec	ored_curvespec.i
InflationCapFloorVolatilityCurveSpec	ored_curvespec.i
InflationCurveSpec	ored_curvespec.i
SecuritySpec	ored_curvespec.i
SwaptionVolatilityCurveSpec	ored_curvespec.i
YieldCurveSpec	ored_curvespec.i
YieldVolatilityCurveSpec	ored_curvespec.i
CSVLoader	ored_loader.i
InMemoryLoader	ored_loader.i
Loader	ored_loader.i
BufferLogger	ored_log.i
FileLogger	ored_log.i
Log	ored_log.i
Logger	ored_log.i
MarketImpl	ored_market.i
TodaysMarket	ored_market.i
BMASwapQuote	ored_marketdatum.i
BaseCorrelationQuote	ored_marketdatum.i
BasisSwapQuote	ored_marketdatum.i
BondOptionQuote	ored_marketdatum.i
BondOptionShiftQuote	ored_marketdatum.i
BondPriceQuote	ored_marketdatum.i
CPRQuote	ored_marketdatum.i
CapFloorQuote	ored_marketdatum.i
CapFloorShiftQuote	ored_marketdatum.i
CdsQuote	ored_marketdatum.i
CommodityForwardQuote	ored_marketdatum.i
CommodityOptionQuote	ored_marketdatum.i
CommoditySpotQuote	ored_marketdatum.i
CorrelationQuote	ored_marketdatum.i
CrossCcyBasisSwapQuote	ored_marketdatum.i
CrossCcyFixFloatSwapQuote	ored_marketdatum.i
DiscountQuote	ored_marketdatum.i
EquityDividendYieldQuote	ored_marketdatum.i
EquityForwardQuote	ored_marketdatum.i
EquityOptionQuote	ored_marketdatum.i
EquitySpotQuote	ored_marketdatum.i
FRAQuote	ored_marketdatum.i
FXForwardQuote	ored_marketdatum.i
FXOptionQuote	ored_marketdatum.i
FXSpotQuote	ored_marketdatum.i
HazardRateQuote	ored_marketdatum.i
ImmFraQuote	ored_marketdatum.i
IndexCDSOptionQuote	ored_marketdatum.i
InflationCapFloorQuote	ored_marketdatum.i
MMFutureQuote	ored_marketdatum.i
MarketDatum	ored_marketdatum.i
MoneyMarketQuote	ored_marketdatum.i
OIFutureQuote	ored_marketdatum.i
RecoveryRateQuote	ored_marketdatum.i
SeasonalityQuote	ored_marketdatum.i
SecuritySpreadQuote	ored_marketdatum.i
SwapQuote	ored_marketdatum.i
SwaptionQuote	ored_marketdatum.i
SwaptionShiftQuote	ored_marketdatum.i
YoYInflationSwapQuote	ored_marketdatum.i
YyInflationCapFloorQuote	ored_marketdatum.i
ZcInflationCapFloorQuote	ored_marketdatum.i
ZcInflationSwapQuote	ored_marketdatum.i
ZeroQuote	ored_marketdatum.i
EngineBuilder	ored_portfolio.i
EngineData	ored_portfolio.i
EngineFactory	ored_portfolio.i
Envelope	ored_portfolio.i
InstrumentWrapper	ored_portfolio.i
LegBuilder	ored_portfolio.i
Portfolio	ored_portfolio.i
Trade	ored_portfolio.i
TradeFactory	ored_portfolio.i
PlainInMemoryReport	ored_reports.i
GenericYieldVolCurve	ored_volcurves.i
LocalVolModelBuilder	ored_volcurves.i

SwaptionVolCurve	ored_volcurves.i

Table A.3: Wrapped classes in QuantExt

Class	SWIG Header
AverageOIS	qle_averageois.i
AverageONIndexedCouponPricer	qle_averageois.i
AverageOISRateHelper	qle_averageoisratehelper.i
Belgium	qle_calendars.i
CME	qle_calendars.i
Colombia	qle_calendars.i
Cyprus	qle_calendars.i
Greece	qle_calendars.i
ICE	qle_calendars.i
Ireland	qle_calendars.i
IslamicWeekendsOnly	qle_calendars.i
LargeJointCalendar	qle_calendars.i
Luxembourg	qle_calendars.i
Malaysia	qle_calendars.i
Netherlands	qle_calendars.i
Peru	qle_calendars.i
Philippines	qle_calendars.i
RussiaModified	qle_calendars.i
Spain	qle_calendars.i
Wmr	qle_calendars.i
FXLinkedCashFlow	qle_cashflows.i
FloatingRateFXLinkedNotionalCoupon	qle_cashflows.i
CrossCcySwap	qle_ccyswap.i
CrossCcySwapEngine	qle_ccyswap.i
AverageONIndexedCoupon	qle_coupons.i
AverageONLeg	qle_coupons.i
CapFlooredAverageONIndexedCouponPricer	qle_coupons.i
CappedFlooredAverageONIndexedCoupon	qle_coupons.i
QLEBlackCdsOptionEngine	qle_creditdefaultswap.i
QLECdsOption	qle_creditdefaultswap.i
CrossCcyFixFloatSwap	qle_crossccyfixfloatswap.i
DiscountingEquityForwardEngine	qle_equityforward.i
EquityForward	qle_equityforward.i
BEHICP	qle_indexes.i
BMAIndex	qle_indexes.i
BMAIndexWrapper	qle_indexes.i
BondFuturesIndex	qle_indexes.i
BondIndex	qle_indexes.i
CommodityFuturesIndex	qle_indexes.i
CommodityIndex	qle_indexes.i
CommoditySpotIndex	qle_indexes.i
ConstantMaturityBondIndex	qle_indexes.i
EquityIndex2	qle_indexes.i
FxIndex	qle_indexes.i
Name	qle_indexes.i
CommodityForward	qle_instruments.i
CrossCcyBasisMtMResetSwap	qle_instruments.i
CrossCcyBasisSwap	qle_instruments.i
Deposit	qle_instruments.i
DepositEngine	qle_instruments.i
DiscountingCommodityForwardEngine	qle_instruments.i
DiscountingFxForwardEngine	qle_instruments.i
DiscountingSwapEngineMultiCurve	qle_instruments.i
FxForward	qle_instruments.i
GeneralisedReplicatingVarianceSwapEngine	qle_instruments.i
OvernightIndexedBasisSwap	qle_instruments.i
Payment	qle_instruments.i
PaymentDiscountingEngine	qle_instruments.i
VarianceSwap2	qle_instruments.i
OvernightIndexedCrossCcyBasisSwap	qle_oiccbasisswap.i
OvernightIndexedCrossCcyBasisSwapEngine	qle_oiccbasisswap.i
BasisTwoSwapHelper	qle_ratehelpers.i
CrossCcyBasisMtMResetSwapHelper	qle_ratehelpers.i
CrossCcyBasisSwapHelper	qle_ratehelpers.i
CrossCcyFixFloatSwapHelper	qle_ratehelpers.i
ImmFraRateHelper	qle_ratehelpers.i

OIBSHelper	qle_ratehelpers.i
OICCBSHelper	qle_ratehelpers.i
SubPeriodsSwapHelper	qle_ratehelpers.i
TenorBasisSwapHelper	qle_ratehelpers.i
SubPeriodsCoupon1	qle_tenorbasisswap.i
SubPeriodsSwap	qle_tenorbasisswap.i
TenorBasisSwap	qle_tenorbasisswap.i
BlackVarianceSurfaceMoneynessForward	qle_termstructures.i
BlackVarianceSurfaceMoneynessSpot	qle_termstructures.i
BlackVolatilityWithATM	qle_termstructures.i
CreditCurve	qle_termstructures.i
CreditVolCurve	qle_termstructures.i
CreditVolCurveWrapper	qle_termstructures.i
FxBlackVannaVolgaVolatilitySurface	qle_termstructures.i
Name	qle_termstructures.i
PriceTermStructure	qle_termstructures.i
QLESwaptionVolCube2	qle_termstructures.i
SwapConventions	qle_termstructures.i
SwaptionVolCubeWithATM	qle_termstructures.i
SwaptionVolatilityConstantSpread	qle_termstructures.i
SwaptionVolatilityConverter	qle_termstructures.i

Bibliography

[1] Open Source Risk Engine, A Free Open-ource Platform for Risk Analytics and XVA, http://opensourcerisk.org, https://github.com/opensourcerisk

[2] QuantLib, A Free Open-Source Library for Quantitative Finance, http://www.quantlib.org

[3] L. Ballabio, "Implementing QuantLib - Quantitative Finance in C++: an inside look at the architecture of the QuantLib library", https://leanpub.com/implementingquantlib

[4] L. Ballabio, "Implementing QuantLib" blog posts on https://www.implementingquantlib.com

[5] Community portal to automatic differentiation (AD), http://www.autodiff.org

[6] A. Griewank. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. Society for Industrial and Applied Mathematics, 2000.

[7] M. Giles and P. Glasserman, Smoking adjoints: fast Monte Carlo Greeks, *RISK*, January 2006.

[8] M. Leclerc, Q. Liang, and I. Schneider. Fast Monte Carlo Bermudan Greeks. *RISK*, July 2009.

[9] M. Joshi and C. Yang. Fast Gamma computations for CDO tranches. *http://ssrn.com/paper=1689348*, 2010.

[10] L. Capriotti and M.B. Giles. Fast correlation Greeks by adjoint algorithmic differentiation. *RISK*, March, 2010.

[11] L. Capriotti. Fast Greeks by algorithmic differentiation. *Journal of Computational Finance*, 4:3–35, 2011.

[12] L. Capriotti and M.B. Giles. Algorithmic differentiation: Adjoint Greeks made easy. *http://ssrn.com/paper=1801522*, 2011.

[13] L. Capriotti, S. Lee, and M. Peacock. Real time counterparty credit risk management in Monte Carlo. *http://ssrn.com/paper=1824864*, 2011.

[14] N. Sherif. AAD vs GPU: banks turn to maths trick as chips lose appeal. *RISK*, January 2015.

[15] C. Fries, Stochastic Automatic Differentiation: Automatic Differentiation for Monte-Carlo Simulations, 2017, https://ssrn.com/abstract=2995695

[16] C. Fries, Automatic Backward Differentiation for American Monte-Carlo Algorithms - ADD for Conditional Expectations and Indicator Functions, 2017, https://ssrn.com/abstract=3000822

[17] C. Fries, Fast Stochastic Forward Sensitivities in Monte-Carlo Simulations Using Stochastic Automatic Differentiation (with Applications to Initial Margin Valuation Adjustments (MVA)), 2017, https://ssrn.com/abstract=3018165

[18] A: Savine, Computation Graphs for AAD and Machine Learning Part I: Introduction to Computation Graphs and Automatic Differentiation, 2019, https://www.researchgate.net/publication/337201082_Computation_Graphs_for_AAD_and_Machine_Learning_Part_I_Introduction_to_Computation_Graphs_and_Automatic_Differentiation

[19] A: Savine, Computation Graphs for AAD and Machine Learning Part II: Adjoint Differentiation and AAD, 2020 https://www.researchgate.net/publication/338662793_Computation_Graphs_for_AAD_and_Machine_Learning_Part_II_Adjoint_Differentiation_and_AAD

[20] A: Savine, Computation Graphs for AAD and Machine Learning Part III: Application to Derivatives Risk Sensitivities, 2020, https://www.researchgate.net/publication/340126636_Computation_Graphs_for_AAD_and_Machine_Learning_Part_III_Application_to_Derivatives_Risk_Sensitivities

[21] A: Savine and L. Andersen Modern Computational Finance: AAD and Parallel Simulations, Wiley, 2018

[22] A: Savine and J. Andreasen, Modern Computational Finance: Scripting for Derivatives and xVA Wiley, 2021

[23] V. Piterbarg, Computing Deltas of Callable Libor Exotics in Forward Libor Models, 2003, https://ssrn.com/abstract=396180

[24] Simplified Wrapper Interface Generator, https://www.swig.org